Decision Tree Predictors for Binary Classification on the Secondary Mushroom Dataset

Piscitelli John Carlo

1 Introduction

Decision tree models are widely used in machine learning for classification tasks due to their interpretability and effectiveness in handling both numerical and categorical data. This report outlines the implementation and evaluation of decision tree models trained on the Secondary Mushroom Dataset to classify mushrooms as either edible or poisonous.

The primary goal is to construct a decision tree model from scratch, that achieves high classification accuracy while maintaining model simplicity. Three splitting criteria—Gini impurity, entropy, and squared impurity—were explored, alongside different stopping conditions to prevent overfitting.

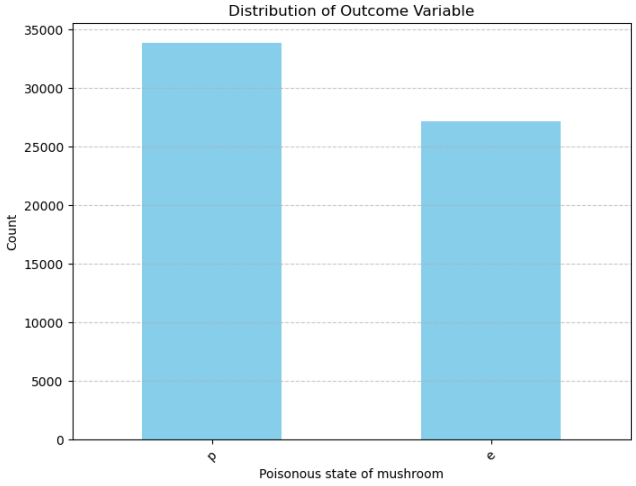
2 The Datasaet

2.1 The Secondary Mushroom Datasaet

The Secondary Mushroom Dataset is a simulated dataset designed for binary classification tasks, distinguishing between edible ('e') and poisonous ('p') mushrooms.

It expands on the original Mushroom Dataset, featuring 61,069 instances and 20 attributes that capture various morphological characteristics. The dataset is comprised of both continuous and categorical variables, defining different physical characteristics of the mushrooms.

The dataset contains slightly more examples of Poisonous mushrooms rather than Edible.



2.2 Data Pre-Processing

The dataset contains different features with missing values, in particular, as the table below shows, there are few attributes such as *veil-type*, *veil-color*, *spore-print-color*, *stem-root*, and *stem-surface* containing more than 50% missing values, with some of these approaching 90%. The missing values are all contained in categorical variables.

Variable	% Missing
class	0.000000
cap-diameter	0.000000
cap-shape	0.000000
cap-surface	23.121387
cap-color	0.000000
does-bruise-or-bleed	0.000000
gill-attachment	16.184971
gill-spacing	41.040462
gill-color	0.000000
stem-height	0.000000
stem-width	0.000000
stem-root	84.393064
stem-surface	62.427746
stem-color	0.000000
veil-type	94.797688
veil-color	87.861272
has-ring	0.000000
ring-type	4.046243
spore-print-color	89.595376
habitat	0.000000
season	0.000000

Table 1: Percentage of missing values in each feature.

The following handling procedures have been taken:

- **Drop columns with more than 50% missing values:** Features such as *veil-type*, *veil-color*, *spore-print-color*, *stem-root*, and *stem-surface* have mostly missing values. The relevancy of these is therefore marginal: these features introduce too much uncertainty into the model, leading to unreliable predictions and inefficient training. Therefore, these columns will be removed from the dataset.
- **Replace missing values with a "Missing" category for the remaining features:** For categorical variables with moderate missing values (e.g., *cap-surface*, *gill-attachment*, *gill-spacing*, and *ring-type*), we replace NaN values with a new category labeled as `Missing`. This ensures that no information is lost and allows the model to potentially learn from the absence of data as a meaningful feature.

Since the dataset consists mostly of categorical features, we apply *one-hot encoding*. One-hot encoding creates a new binary column for each unique category in a categorical variable, assigning a value of 1 if the category is present and 0 otherwise.

3 Decision Trees

Decision trees is a machine learning algorithm used for classification tasks. They are based on hierarchical structures consisting of nodes, where each internal node represents a decision based on a specific feature, and each leaf node represents a class label.

In this project, we implement decision tree predictors from scratch to classify mushrooms as poisonous or edible using the Secondary Mushroom Dataset. The decision tree follows a top-down, greedy approach, selecting splits based on an impurity criterion such as *Gini* index, *Entropy*, or *Squared Impurity*. The tree expands based on defined stopping criteria to avoid overfitting.

3.1 Tree Implementation Breakdown

3.1.1 Node Class

Our Decision Tree model is built using the Node class as its core structure. Each node in the tree represents either a decision point or a final classification. If it's a decision point, it stores a feature and a threshold, splitting the data into two groups based on whether the value is above or below the threshold. If it's a leaf, it holds the final predicted class.

Listing 1: Node Class

```
class Node:
    def __init__(self, feature=None, threshold=None, value=
        None, depth=0):
        self.feature = feature
        self.threshold = threshold
        self.value = value
        self.depth = depth
        self.left = None
        self.right = None

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

3.1.2 Decision Tree Implementation

The approach to building a Decision Tree model starts with building a DecisionTree class that provides a structured way to create and train decision trees. This class is designed to handle training, making predictions, and evaluating performance using a recursive tree-building approach.

Listing 2: DecisionTree Class

```
class DecisionTree:
Procedure DecisionTree(criterion, max_depth,
    min_samples_split, impurity_threshold, feature_names):
    SET self.criterion = criterion
```

```
    SET self.max_depth = max_depth
    SET self.min_samples_split = min_samples_split
    SET self.impurity_threshold = impurity_threshold
    SET self.feature_names = feature_names
    SET self.root = None
    SET self.tree_depth = 0

    Select impurity function based on criterion
```

The grow_tree method is responsible for constructing the decision tree recursively. It starts at the root and determines the best feature and threshold to split the dataset, aiming to maximize impurity reduction. At each step, it checks stopping conditions—such as if all samples belong to the same class, the number of samples is too small, the maximum depth is reached, or impurity is below a threshold. If any of these conditions are met, the node becomes a leaf with the majority class.

If a valid split is found, the dataset is divided into left and right subsets, ensuring neither is empty. he method then calls itself recursively to grow the left and right subtrees, increasing depth at each step. This process continues until all nodes are either decision nodes or leaf nodes. The final tree is a combination of internal nodes, which store decision rules, and leaf nodes, which store class labels.

Listing 3: GrowTree

```
procedure grow_tree(X, y, depth):
    num_samples, num_features = shape of X
    current_impurity = compute impurity using selected
        criterion
    update tree_depth = max(tree_depth, depth)

    if all samples belong to the same class or num_samples <
         min_samples_split:
        return leaf node with majority class

    if max_depth is set and depth     max_depth:
        return leaf node with majority class

    if impurity_threshold is set and current_impurity <
        impurity_threshold:
        return leaf node with majority class

    best_gain = 0, best_feature = none, best_threshold =
        none
    parent_impurity = current_impurity

    for each feature in X:
        for each unique threshold in the feature column:
            gain = call _gain(feature column, y, threshold,
                parent_impurity)
```

```
            if gain > best_gain:
                best_gain = gain
                best_feature = feature
                best_threshold = threshold

    if best_gain is 0:
        return leaf node with majority class

    left_mask = X[:, best_feature]     best_threshold
    right_mask = not left_mask

    if left or right subset is empty:
        return leaf node with majority class

    left_node = recursive call _grow_tree(left subset, depth
        + 1)
    right_node = recursive call _grow_tree(right subset,
        depth + 1)

    return internal node with best_feature, best_threshold,
        left_node, right_node
```

The gain procedure evaluates how well a feature split improves class separability, calculating impurity reduction before and after the split. The best split is chosen based on the maximum information gain.

Listing 4: Gain

```
procedure _gain(feature_column, y, threshold,
    parent_impurity)

    left_mask = feature_column     threshold
    right_mask = not left_mask

    if left subset or right subset is empty
        return 0

    left_impurity = compute impurity(y_left)
    right_impurity = compute impurity(y_right)
    weighted_impurity = weighted sum of left_impurity and
        right_impurity

    return parent_impurity - weighted_impurity
```

The majority_class procedure identifies the most frequent class in a given subset of labels. Used when a node needs to become a leaf.

Listing 5: Majority Tree

```
procedure _majority_label(y)
    return most frequent class in y
```

The _traverse function is responsible for navigating the decision tree recursively. If the current node is a leaf, it directly returns the stored class label. Otherwise, it evaluates the feature value of the row against the node's threshold. If the feature value is less than or equal to the threshold, the function moves to the left child; otherwise, it moves to the right child. This process continues until a leaf node is reached, at which point the function returns the predicted class label.

The predict method takes a dataset applies the _traverse function to each row, starting from the root node of the decision tree. It returns an array of predicted class labels, where each row in X is classified based on the learned tree structure.

Listing 6: Predict and Traverse

```
procedure predict(X)
    return array of _traverse(row) for each row in X

procedure _traverse(row, node)
    if node is a leaf
        return node.value
    if row[node.feature]     node.threshold
        return call _traverse(row, node.left)
    else
        return call _traverse(row, node.right)
```

3.2 Impurity Measures

To determine the best split at each node, we calculate impurity using one of three methods:

Gini Impurity

The *Gini* index measures the probability of misclassification if a random sample were classified according to the class distribution in a node.

$$Gini(y) = 1 - \sum_{i=1}^{C} p_i^2 \tag{1}$$

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

Entropy

Entropy measures the uncertainty or disorder in a dataset.

$$Entropy(y) = -\sum_{i=1}^{C} p_i \log_2 p_i \tag{2}$$

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

Squared Impurity

The *squared impurity* is another measure. It is a variation of the Gini Index. It is computed as:

$$Squared(y) = \sum_{i=1}^{C} \sqrt{p_i(1 - p_i)} \tag{3}$$

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

3.3 Stopping Criteria

To prevent overfitting and ensure that the decision tree does not grow excessively complex, we apply specific stopping criteria. These conditions determine when to halt further splitting of nodes.

Maximum Depth

The parameter `max_depth` sets an upper limit on how deep the tree can grow. A deeper tree can capture more complex patterns, but it also increases the risk of overfitting by memorizing the training data.

Minimum Samples per Node

The parameter `min_samples_split` specifies the minimum number of samples required for a node to be split. If a node contains fewer samples than this threshold, further splitting is not allowed, and the node becomes a leaf.

Impurity Threshold

The parameter `impurity_threshold` prevents unnecessary splits by ensuring that a node is only split if its impurity is above a predefined value. Once impurity falls below this threshold, the node is considered sufficiently pure and is converted into a leaf node.

3.4 Zero-One Loss

To evaluate model performance, we calculate both training and test errors using zero-one loss.

$$\text{Training Error} = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \mathbb{1}(y_i \neq \hat{y}_i)$$

$$\text{Test Error} = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} \mathbb{1}(y_i \neq \hat{y}_i)$$

where n_{train} and n_{test} are the number of samples in the training and test sets, respectively, and with y_i being the true label, $\hat{y}_i$ the predicted label, and

$\mathbb{1}(y_i \neq \hat{y}_i)$ is an indicator function that returns 1 if the prediction is incorrect and 0 otherwise

4 Model Implementation

4.1 Hyperparameter Tuning

Hyperparameters are the configuration parameters of a machine learning model which are defined before the training process begins. Choosing the right hyperparameter is important because it directly impacts the model's performance.

To select good hyperparameters, we perform a grid search, systematically trying different combinations of impurity measures and stopping criteria. For each combination, we train a tree and measure how well it performs. By comparing the performance across all configurations, we identify which hyperparameters yield the best results for our dataset.

We used 5-fold cross validation to assess each hyperparameter combination more reliably. In 5-fold CV, the training dataset is split into five equal parts ("folds"). For each fold, we train the model on the other four folds and evaluate it on the held-out fold. This process repeats five times so that every data point becomes part of a test fold once. We then average the resulting performance metrics across all five folds to get a more robust estimate.

Listing 7: Grid Search

```
procedure grid_search(X, y, criteria, stop_criterion,
    stop_values, feature_names, n_splits=5, n_jobs=-1):
    determine all parameter combinations (each combination
        is one impurity measure and one stop value).
    for each combination:
       a) Perform K-fold splitting of (X, y).
       b) For each fold:
          - Train a DecisionTree using the specific impurity
              measure and stopping criterion value.
          - Compute zero-one loss on both the folds
              training set and test set.
          - Record tree depth and leaf count.
       c) Average the training loss, test loss, depth, and
           leaf count across all folds.
       d) Print or log these averaged metrics.
       e) Store the results in a list (including the
           hyperparameters and metrics).
    return the list of all results.
```

	criterion	max_depth	avg_train_loss	avg_test_loss	avg_tree_depth	avg_leaf_count
9	entropy	10	0.152359	0.155583	10.0	93.0
10	entropy	11	0.112706	0.113458	11.0	108.6
11	entropy	12	0.065838	0.066749	12.0	124.6
12	entropy	13	0.047114	0.049207	13.0	141.6
13	entropy	20	0.001407	0.004565	20.0	229.6
14	entropy	25	0.000020	0.003582	25.0	250.2
15	entropy	30	0.000000	0.003541	25.6	250.8
16	entropy	40	0.000000	0.003541	25.6	250.8
17	entropy	50	0.000000	0.003541	25.6	250.8
0	gini	10	0.113955	0.118084	10.0	96.6
1	gini	11	0.077275	0.080401	11.0	119.6
2	gini	12	0.044218	0.045195	12.0	139.8
3	gini	13	0.025612	0.028636	13.0	162.2
4	gini	20	0.003060	0.006775	20.0	242.0
5	gini	25	0.000921	0.004851	25.0	262.8
6	gini	30	0.000036	0.004053	29.8	279.6
7	gini	40	0.000000	0.004053	30.8	281.4
8	gini	50	0.000000	0.004053	30.8	281.4
18	squared	10	0.215792	0.216784	10.0	48.8
19	squared	11	0.204815	0.205486	11.0	60.6
20	squared	12	0.165454	0.166677	12.0	71.4
21	squared	13	0.139315	0.141398	13.0	82.8
22	squared	20	0.022081	0.024849	20.0	159.4
23	squared	25	0.002881	0.005793	25.0	196.0
24	squared	30	0.000287	0.002968	29.4	208.4
25	squared	40	0.000000	0.002784	31.8	212.0
26	squared	50	0.000000	0.002784	31.8	212.0

Figure 1: Hyperparameter tuning on max_depth

	criterion	min_samples_split	avg_train_loss	avg_test_loss	avg_tree_depth	avg_leaf_count
6	entropy	2	0.000000	0.003541	25.6	250.8
7	entropy	5	0.000092	0.003562	25.4	247.2
8	entropy	10	0.000445	0.003521	25.2	239.0
9	entropy	25	0.001361	0.003991	25.0	225.4
10	entropy	50	0.003311	0.006120	24.8	211.0
11	entropy	100	0.006668	0.009047	24.6	193.0
0	gini	2	0.000000	0.004053	30.8	281.4
1	gini	5	0.000169	0.004155	30.8	274.8
2	gini	10	0.000425	0.004278	30.8	267.6
3	gini	25	0.001141	0.004749	30.8	257.0
4	gini	50	0.003526	0.006877	30.6	239.0
5	gini	100	0.008397	0.011299	30.2	213.6
12	squared	2	0.000000	0.002784	31.8	212.0
13	squared	5	0.000097	0.002886	31.6	209.2
14	squared	10	0.000271	0.002845	31.6	205.8
15	squared	25	0.001259	0.003766	30.4	194.2
16	squared	50	0.002988	0.005445	30.2	182.8
17	squared	100	0.006611	0.008822	29.6	168.0

	criterion	impurity_threshold	avg_train_loss	avg_test_loss	avg_tree_depth	avg_leaf_count
7	entropy	0.00	0.000000	0.003541	25.6	250.8
8	entropy	0.01	0.000087	0.003500	25.6	245.0
9	entropy	0.10	0.003608	0.006857	22.8	176.0
10	entropy	0.20	0.009487	0.012056	21.8	138.2
11	entropy	0.25	0.011534	0.014082	21.8	129.6
12	entropy	0.50	0.041787	0.045359	19.8	83.2
0	gini	0.00	0.000000	0.004053	30.8	281.4
1	gini	0.01	0.001735	0.005322	27.6	219.6
2	gini	0.10	0.024578	0.028042	19.8	108.0
3	gini	0.20	0.062445	0.065807	14.4	65.4
4	gini	0.25	0.079900	0.084106	13.6	55.8
5	gini	0.50	0.445093	0.445093	0.0	1.0
14	squared	0.00	0.000000	0.002784	31.8	212.0
15	squared	0.01	0.000000	0.002784	31.8	212.0
16	squared	0.10	0.000159	0.002743	31.8	204.4
17	squared	0.20	0.000834	0.003664	31.8	190.4
18	squared	0.25	0.001781	0.004442	31.8	177.2
19	squared	0.50	0.010941	0.012772	31.4	134.6

Figure 2: Hyperparameter tuning on min_samples_split and impurity_threshold

4.2 Observations

The tables above show the result of a `grid_search` run on stopping criteria `max_depth`, `min_samples_split`, and `impurity_threshold` on three different impurity measures, *Entropy*, *Gini*, and *Squared Impurity*.

Using `max_depth` as a stopping criterion, the tree grows up until the `max_depth` that's been tested is reached. Regardless of the impurity measure used, at some point, while setting very high thresholds, the tree stops growing. Pruning is not required, as the tree does not grow excessively.

`min_samples_split` specifies the minimum number of samples contained in a node for it to be further split. As `min_samples_split` increases, the model tends to stop splitting sooner; however, both training loss and test loss remain low, with marginal differences among each other, and change only marginally. Hence, the tree remains fairly robust even when forced to have larger node sizes before splitting.

Observing the tuning on `impurity_threshold`, when the threshold is very low, the tree is allowed to split until it almost perfectly fits the training set. In these cases, the train loss is effectively zero and the test loss stays small. As the threshold grows, the tree stops splitting earlier, and we see a gradual rise in both training and test losses. The depth and leaf count drop accordingly, reflecting more aggressive stopping.

We can observe how this stopping criterion obtains a good compromise in terms of lower leaf count and shallower depth, specifically when we use `impurity_threshold` of 0.25 and *Gini* impurity measure. The resulting tree is simpler to interpret and computationally efficient while retaining strong classification results. This makes the `impurity_threshold`-based approach the best choice.

4.3 Conclusions

Thanks to the Hyperparameter tuning phase, performed through a Grid Search approach, we were able to identify the best combination of hyperparameters for our purposes. Specifically, we were looking for a DecisionTree model which compromised predictive power with tree size and complexity.

The hyperparameters of choice are *Gini* Impurity and `impurity_thresold` equivalent to 0.25. We have therefore performed 5-fold cross-validation to make prediction and evaluate our model. Below are the metrics obtained, averaged across all folds, with the confusion matrix and the Decision Tree obtained specifically on the last fold.

	Predicted 0	Predicted 1
Actual 0	4839	553
Actual 1	483	6338

Metric	Value
Average Training Loss	0.07853
Average Testing Loss	0.08068
Average Tree Depth	15.00
Average Leaf Count	55.80

The average training loss of 0.07853 and average testing loss of 0.08068 indicate that the model has strong predictive power, as these values are very close. This suggests that the model generalizes well and does not suffer from significant overfitting. Additionally, with an average tree depth of 15 and an average leaf count of 55.80, the model remains moderately complex—neither excessively deep nor overly complicated. Given this balanced performance and controlled complexity, there is no need for further pruning; the tree is already optimized for both accuracy and generalization.