Tree predictors for binary classification

Aidana Akkaziyeva

Abstract

This report covers the development and analysis of a Decision Tree Classifier applied to a dataset to predict the class label ('e' or 'p') based on various mushroom characteristics. The entire process from data preprocessing to model evaluation is explained, including hyperparameter tuning using GridSearchCV to enhance performance.

1 Introduction

This report covers the development and analysis of a **Decision Tree Classifier** applied to a dataset to predict the class label ('e' or 'p') based on various mushroom characteristics. The entire process, from data preprocessing to model evaluation, is explained, including hyperparameter tuning using **GridSearchCV** with parallel processing to enhance performance.
This report documents the process of building and optimizing a **Decision Tree Classifier** for predicting the class of mushrooms (edible or poisonous) based on various mushroom characteristics. The dataset was preprocessed, and hyperparameter tuning was conducted using both **RandomizedSearchCV** and **GridSearchCV** with parallel processing to efficiently explore model configurations. Special attention was given to the potential for overfitting, and learning curves were plotted to evaluate model generalization.

2 Dataset Description

2.1 Purpose of the Dataset

The dataset used in this project is inspired by the Mushroom Data Set of J. Schlimmer [2]: https://archive.ics.uci.edu/ml/datasets/Mushroom. The primary purpose of the dataset is to simulate hypothetical mushrooms from 173 species, to classify them as edible or poisonous. It contains 61,069 mushrooms, with 353 samples for each species. Each sample is labeled as definitely edible or definitely poisonous. For simplicity, the mushrooms of unknown edibility have been combined with the poisonous class.

2.2 Target and Features

- **Class** (target): categorical with two values ('e' for edible, 'p' for poisonous).
- **Cap diameter:** Measures in cm.
- **Cap shape:** bell, conical, convex, flat, sunken, spherical, others.
- **Cap surface:** fibrous, grooves, scaly, smooth, shiny, leathery, silky, sticky, wrinkled, fleshy.
- **Cap color:** brown, buff, gray, green, pink, purple, red, white, yellow, blue, orange, black.
- **Bruises/Bleeds:** 't' for yes, 'f' for no.
- **Gill attachment:** adnate, adnexed, decurrent, free, sinuate, pores, none, unknown.
- **Gill spacing:** close, distant, none.
- **Gill color:** brown, buff, gray, green, pink, purple, red, white, yellow, blue, orange, black, 'f' for none.

- **Stem height:** Measures in cm.
- **Stem width:** Measures in mm.
- **Stem root:** bulbous, swollen, club, cup, equal, rhizomorphs, rooted.
- **Stem surface:** fibrous, grooves, scaly, smooth, shiny, leathery, silky, sticky, wrinkled, fleshy, 'f' for none.
- **Stem color:** brown, buff, gray, green, pink, purple, red, white, yellow, blue, orange, black, 'f' for none.
- **Veil type:** 'p' for partial, 'u' for universal.
- **Veil color:** brown, buff, gray, green, pink, purple, red, white, yellow, blue, orange, black, 'f' for none.
- **Has ring:** 't' for yes, 'f' for no.
- **Ring type:** cobwebby, evanescent, flaring, grooved, large, pendant, sheathing, zone, scaly, movable, none, unknown.
- **Spore print color:** brown, buff, gray, green, pink, purple, red, white, yellow, blue, orange, black, 'f' for none.
- **Habitat:** grasses, leaves, meadows, paths, heaths, urban, waste, woods.
- **Season:** spring, summer, autumn, winter.

3 Preprocessing

3.1 Missing Values

The dataset originally comprised over 61,000 samples with both categorical (17) and numerical features (3). Upon inspection, it was found that some columns had a significant number of missing values, all of which were categorical. Several columns had between 14,000 and 51,000 missing values, while others had fewer, with counts as low as 2,000 to 9,800. To address this, a threshold of 80% for missing values was set. As a result, the following columns were removed:

- cap-surface (14120 missing values)
- gill-attachment (9884 missing values)
- gill-spacing (25063 missing values)
- stem-root (51538 missing values)
- stem-surface (38124 missing values)
- veil-type (57892 missing values)
- veil-color (53656 missing values)
- spore-print-color (54715 missing values)

Rows with the remaining missing values were dropped resulting in a dataset with 58595 rows and 14 columns (target included).

3.2 Encoding Categorical Features

The cleaned dataset consisted of both numerical and categorical features. Since the chosen model was a tree-based classification algorithm, categorical features were transformed using one-hot encoding using `one_hot_encoding` and `custom_label_encoding` from `classes.py`. After preprocessing, the data was split into training and test sets (80% for training, 20% for testing) using `custom_train_test_split`.

Algorithm 1 custom_train_test_split

Input: X, y, test_size = 0.2, random_state = None
if random_state is not None **then**
    Set the random seed to random_state
end if
Get total number of samples: $n \leftarrow X.shape[0]$
Create an array of indices: $indices \leftarrow [0, 1, \ldots, n-1]$
Shuffle the indices
Split point: $split\_idx \leftarrow n \times (1 - test\_size)$
Get train and test indices:
    $train\_indices \leftarrow indices[: split\_idx]$
    $test\_indices \leftarrow indices[split\_idx :]$
Split X into X_train and X_test using $train\_indices$ and $test\_indices$
Split y into y_train and y_test using $train\_indices$ and $test\_indices$
Output: X_train, X_test, y_train, y_test =0

4 Hyperparameter Tuning and Time Estimation

4.1 Overview of Hyperparameter Search

To speed up the hyperparameter tuning process, 20% of the training data is randomly sampled using the above `cl.custom_train_test_split()` function. For this project, a randomized hyperparameter search was employed to optimize the performance of the decision tree classifier. The hyperparameter grid that was used is outlined below:

```
param_grid = {
    'max_depth': [10, 20, 30, 40, 50],
    'min_samples_split': [5, 10, 15, 20],
    'criterion': ['gini', 'entropy', 'misclassification_error']
}
```

The search space includes 5 possible values for the maximum 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 parameter combinations.

4.2 RandomizedSearchCV vs. GridSearchCV

Given the characteristics of our dataset and the hyperparameter grid, RandomizedSearchCV emerged as the more suitable choice for the following reasons.

4.2.1 Efficiency in High-Dimensional Spaces

GridSearchCV evaluates every possible combination of hyperparameters, leading to a combinatorial explosion in the search space, particularly with multiple hyperparameters. With the parameter grid defined above there are 60 combinations to evaluate. As the number of hyperparameters increases, the computational cost of grid search grows significantly[3].

In contrast, RandomizedSearchCV randomly samples a specified number of combinations (e.g., `n_iter=10`), allowing for a broader exploration of the hyperparameter space without evaluating every single combination. This not only speeds up the tuning process but can also yield competitive results.

4.2.2 Reducing Overfitting

By reducing the time spent on exhaustive searches, RandomizedSearchCV minimizes the risk of overfitting. With a large parameter space, it can provide a balanced approach between exploration and exploitation, ensuring a more efficient search for optimal parameters.

In summary, for our hyperparameter tuning efforts, RandomizedSearchCV provides a more practical and efficient approach compared to GridSearchCV, making it particularly well-suited for high-dimensional parameter spaces like the one encountered in this project.

4.3 Pseudo Code for Randomized Search

Algorithm 2 Randomized Search CV

function RandomizedSearchCV(model, param_grid, X_train, y_train, n_iter, cv, random_state)
  $best\_score \leftarrow -\infty$
  $best\_params \leftarrow None$
  Randomly set seed based on *random_state*
  for $i \leftarrow 1$ to *n_iter* **do**
    Sample a combination of parameters from *param_grid*
    Set model parameters to the sampled combination
    Perform k-fold cross-validation on the model with *cv* folds
    Compute the average cross-validation score for the current model
    if current score > best_score **then**
      $best\_score \leftarrow$ current score
      $best\_params \leftarrow$ current combination of parameters
  return $best\_params, best\_score$

4.4 Time Estimation for Randomized Search

The overall time taken for hyperparameter tuning depends on multiple factors, including the size of the dataset, the complexity of the decision tree, and the number of cross-validation folds. To estimate the generalization performance of each parameter configuration, we applied 5-fold cross-validation (*cv=5*). Thus, for each iteration, the model was trained and evaluated on 5 different train-validation splits. With 10 iterations and 5 cross-validation folds, this resulted in a total of 50 model evaluations:

$$\text{Total Evaluations} = n\_iter \times cv = 10 \times 5 = 50 \tag{1}$$

Our sampled dataset contains approximately 9000 samples and 77 features, which is a moderate size. Training a decision tree for one set of parameters on this dataset, assuming deeper trees (e.g., max_depth = 50), could take anywhere between 10 seconds to 1 minute, depending on the maximum depth and the criterion used.

Assuming an average of 30 seconds per evaluation fold, the total estimated time for the randomized search with 50 model evaluations can be calculated as follows:

$$\text{Total Time} = 50 \times 30 \text{ seconds} = 1500 \text{ seconds} = 25 \text{ minutes} \tag{2}$$

4.5 System Resources and Parallel Processing

To expedite the hyperparameter search, we employed parallel processing using *n_jobs=-1*, which allows the algorithm to utilize all available CPU cores[4]. Parallel processing significantly reduces the time taken for the search, depending on the number of CPU cores available. While the search space was moderately large, careful resource management and parallelization allowed us to conduct the hyperparameter search efficiently resulting in 20 minutes of runtime.

4.6 Optimal Hyperparameters

The hyperparameter tuning process yielded the following optimal parameters for the decision tree model:

- **Max Depth:** 40
- **Min Samples Split:** 10
- **Criterion:** Entropy

4.6.1 Interpretation of Best Parameters

The parameter `max_depth` is set to 40, indicating that the decision tree can be quite deep, allowing the model to capture complex patterns within the data. However, this also increases the risk of overfitting, particularly when the training dataset is not large enough.

The `min_samples_split` parameter, configured to 10, specifies that a node must contain at least 10 samples to be eligible for splitting. This constraint helps to prevent the creation of nodes based on very few samples, reducing the likelihood of overfitting.

The choice of `criterion` as `entropy` signifies that the model uses information gain to evaluate the quality of splits. This criterion focuses on reducing uncertainty in the data, aiming to create branches that maximize class separation, which can lead to a more balanced decision tree.

The best average cross-validation score achieved during tuning was:

$$\textbf{Accuracy} : 0.9554\,(95.54\%)$$

This cross-validation score indicates that the model was able to correctly classify approximately 96% of the instances across multiple validation folds, suggesting good generalization performance during training.

5 Model Description

The project implements a binary decision tree classifier. Each tree node either splits the data based on a specific feature and threshold or acts as a leaf node providing a prediction [1]. The main components of the decision tree model include:

- **TreeNode Class:** A recursive structure that holds the feature index, split threshold, and pointers to child nodes. It also determines if a node is a leaf.
- **DecisionTree Class:** This class contains methods for calculating split criteria, such as Gini impurity, entropy, and misclassification error. It recursively builds the tree based on these metrics.
- **Splitting Criteria:** The decision tree supports three splitting criteria—Gini impurity, entropy, and misclassification error. These are used to measure the quality of a split and to identify the best feature and threshold for each node.

The model is trained recursively by evaluating possible splits at each node and selecting the one that minimizes the impurity or error, as defined by the chosen splitting criterion. The process ends when the stopping criteria are met: either the tree reaches a specified maximum depth or the node cannot be split further due to insufficient samples.

The training dataset is fed into the `fit()` function of the `DecisionTree` class, which begins the recursive process of building the tree.

5.1 TreeNode Class

The `TreeNode` class is designed to represent a node within a decision tree, where the data is split based on a feature and a defined threshold.

The constructor of the `TreeNode` class initializes several important parameters:

- **feature**: An integer representing the index of the feature used for making the split at this node.
- **threshold**: A numeric value that determines the criterion for splitting the data, effectively serving as the decision boundary.
- **left** and **right**: References to the child nodes, which represent the outcomes of the decision made at the current node.
- **is_leaf**: A boolean flag indicating whether the current node is a leaf node, which means it does not further split the data.

- **prediction**: A value that holds the predicted outcome if the node is a leaf, representing the final classification for the input data.

The `predict` method within this class is responsible for generating predictions based on input feature values. When invoked, the method first checks if the current node is a leaf node. If it is, the method returns the stored prediction. If the node is not a leaf, the method evaluates the value of the relevant feature against the specified threshold. Based on this comparison, it recursively calls itself on either the left or right child node, guiding the traversal down the tree until a leaf node is reached, at which point the prediction is returned.

5.2 DecisionTree Class

The `DecisionTree` class provides functionality for building and evaluating a decision tree based on specified parameters, such as the maximum depth of the tree, the minimum number of samples required to split an internal node, and the criterion used for evaluating splits.

5.2.1 Constructor

The constructor initializes the following parameters:

- **max_depth**: An optional parameter that limits the maximum depth of the tree. If set to `None`, the tree will grow until all leaves are pure or until it contains fewer samples than `min_samples_split`.
- **min_samples_split**: This parameter specifies the minimum number of samples required to split an internal node. The default value is 2, meaning that a node must have at least two samples to be considered for splitting.
- **criterion**: A string that specifies the function to measure the quality of a split. The default is `'gini'`, but it can also be set to `'entropy'` or `'misclassification_error'`.
- **root**: This attribute is initialized to `None` and will later store the root node of the decision tree.

5.2.2 Parameter Management

The class includes methods for managing parameters:

- `get_params`: Returns the parameters of the decision tree as a dictionary. This is useful for model inspection and tuning.
- `set_params`: Accepts keyword arguments to update the parameters of the decision tree. This method enables easy adjustment of model settings.

5.2.3 Split Criteria Functions

The class contains several private methods to compute different criteria used for evaluating the quality of a split:

- `_gini`: Calculates the Gini impurity of the target labels `y`. A lower Gini score indicates a better split.
- `_entropy`: Computes the entropy of the target labels, which quantifies the amount of disorder or uncertainty. Similar to Gini, lower values indicate a more homogeneous set.
- `_misclassification_error`: Calculates the misclassification error rate, which represents the proportion of incorrect predictions. The goal is to minimize this error in the splits.

5.3 Best Split Function

Algorithm 3 Best Split Function

function BESTSPLIT(X, y)
  $best\_feature, best\_threshold \leftarrow None, None$
  $best\_score \leftarrow \infty$
  $m, n \leftarrow \text{shape}(X)$
  for $feature \in [0, n-1]$ **do**
    $thresholds \leftarrow \text{unique}(X[:, feature])$
    for $threshold \in thresholds$ **do**
      $left\_mask \leftarrow (X[:, feature] \leq threshold)$
      $right\_mask \leftarrow (X[:, feature] > threshold)$
      if $\text{sum}(left\_mask) < min\_samples\_split$ or $\text{sum}(right\_mask) < min\_samples\_split$ **then**
      end if
      if $criterion =' gini'$ **then**
        $score \leftarrow \frac{\text{sum}(left\_mask) \cdot Gini(y[left\_mask]) + \text{sum}(right\_mask) \cdot Gini(y[right\_mask])}{m}$
      else if $criterion =' entropy'$ **then**
        $score \leftarrow \frac{\text{sum}(left\_mask) \cdot Entropy(y[left\_mask]) + \text{sum}(right\_mask) \cdot Entropy(y[right\_mask])}{m}$
      else if $criterion =' misclassification\_error'$ **then**
        $score \leftarrow \frac{\text{sum}(left\_mask) \cdot MisclassificationError(y[left\_mask]) + \text{sum}(right\_mask) \cdot MisclassificationError(y[right\_mask])}{m}$
      end if
      if $score < best\_score$ **then**
        $best\_score \leftarrow score$
        $best\_feature \leftarrow feature$
        $best\_threshold \leftarrow threshold$
  return $best\_feature, best\_threshold$

6 Final Evaluation

6.1 Zero-One Loss Evaluation

The zero-one loss metric provides insight into the classification accuracy of the model on both training and test datasets.

- **Training Zero-One Loss:** 0.75%
- **Test Zero-One Loss:** 1.26%

6.1.1 Model Performance and Overfitting Assessment

The model demonstrated good performance on the test set, with the following metrics:

- **Accuracy**: 98.74%
- **Precision**: 98.98%
- **Recall**: 98.72%
- **F1 Score**: 98.85%

The confusion matrix indicates that the model correctly identified:

- True Positives (TP): 5320
- True Negatives (TN): 4370
- False Positives (FP): 55
- False Negatives (FN): 69

These metrics suggest that the model is reliable and effectively minimizes false positives and negatives. The small difference between the training zero-one loss (0.75%) and the test zero-one loss (1.26%) indicates that the model generalizes well to unseen data.

Based on the strong performance metrics and the lack of significant overfitting, pruning may not be necessary at this stage.

7 Conclusion

In this report, the application of machine learning techniques to address a classification problem using decision trees was explored. A systematic approach involving data preprocessing, hyperparameter tuning, and performance evaluation was employed to develop a robust model capable of accurately classifying instances based on the given features.

After fine-tuning the model's parameters using randomized search cross-validation, I identified optimal settings that greatly improved its performance. The final model produced impressive results, with an accuracy of 98.74%, precision of 98.98%, and recall of 98.72%. These findings indicate the model's high reliability in predicting target classes while minimizing false positives and false negatives.

Additionally, the assessment of overfitting revealed that the model generalizes well to unseen data, as evidenced by the relatively small discrepancy between training and test performance.

The results of this study emphasize the effectiveness of decision trees in classification tasks and stress the significance of optimizing hyperparameters and evaluating models in the development of machine learning solutions. Future research could focus on exploring ensemble methods, such as random forests, to enhance performance and robustness.

References

[1] Chen, L. (2019). Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar: Foundations of machine learning, second edition. Statistical Papers, 60(5), 1793–1795. https://doi.org/10.1007/s00362-019-01124-9

[2] Wagner, D., Heider, D., Hattab, G. (2021). Secondary Mushroom [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C5FP5Q.

[3] GridSearchCV. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[4] joblib.Parallel — joblib 1.5.dev0 documentation. (n.d.). https://joblib.readthedocs.io/en/latest/generated/joblib.Parallel.html

8 Appendix

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 I 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.