Decision Tree from Scratch with Mushroom Dataset

Mubarak Oluwanishola Babslawal

Matriculation Number: 32189A

Machine Learning (DSE)

Date: September 2, 2024

University of Milan
Data Science in Economics Master

Plagiarism Declaration

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

This report discusses the implementation of a decision tree from scratch in python. The sections include the:

i. Scripting of the Decision Tree

ii. Testing of the various splitting criterion on the mushroom dataset

iii. Tuning on a single parameter

iv. Discussion of the results of the experiment.

Particular attention is paid to meeting the requirements of the assignment as detailed on the course website.

2 Discussion of the Scripts

The scripting of the Decision Tree class and its associated method are included in the `DecisionTree.py` file. This file is adapted from the solution by AssemblyAI [2]. On the other hand, the file utils.py contains other general machine learning functions to handle the data preprocessing and for assessing the models' performances.

2.1 Node Class

The `Node` class is a simple structure used to represent a node in the decision tree. Each node can either be an internal node, which holds a decision (split) based on a feature and threshold, or a leaf node, which holds a final predicted value. The class contains a single function that checks if the node is a leaf node

Code block 1: Node Class

```python
class Node:
    def __init__(self, feature: str = None, threshold=None,
        left=None, right=None, *, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

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

2.2 DecisionTree Class

The `DecisionTree` class implements the decision tree algorithm It supports three split criteria which are:

1. Entropy
2. Gini impurity
3. Misclassification error

1. Maximum depth of the tree
2. Minimum number of samples to be split at each node
3. Number of features to consider for each split

The following sections describe each section of the class.

2.2.1 Initialization

The constructor of the `DecisionTree` class allows customization of both stopping and splitting criteria. In the absence of any selection, it uses a maximum depth of 10 and splits on the entropy criterion. The definition of each splitting criterion is included in a subsequent section

Code block 2: DecisionTree Class Initialization

```
class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=10,
        n_features=None, split_using="entropy"):
        if split_using not in ('entropy', 'gini', 'train_error'):
            raise ValueError(f"split_using argument must be one of
                ('entropy', 'gini', 'train_error')")
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_features = n_features
        self.root = None
        self.split_using = split_using
```

2.2.2 Selecting the Optimal Splits

The core functionality of the decision tree algorithm is determining how to split the data at each node. The `_best_split()` method finds the optimal split based on the specified criterion. To find the optimal split, the best split function iterates through all possible splits using the specified criterion. Determining the optimality of the split is done by the information gain function.

Code block 3: Selecting the best split

```
def _best_split(self, X, y, feat_idxs):
    best_gain = -1
    split_idx, split_thresh = None, None
    for idx in feat_idxs:
        X_column = X[:, idx]
        thresholds = np.unique(X_column)
        for thresh in thresholds:
            gain = self._information_gain(y, X_column, thresh)
            if gain > best_gain:
                best_gain = gain
                split_idx = idx
                split_thresh = thresh
    return split_idx, split_thresh, best_gain
```

2.2.3 Information Gain

The information gain function simply compares the impurity/entropy/error (depending on split criterion selected) of the splitted samples, with the same before splitting. This value is then passed to the best split function which iterates using the information gain function for each possible split and selects the one that most reduces the impurity or error.

Code block 4: Splitting and Information Gain Calculation

```
    def _gain(self, y, X_column, threshold):
        if self.split_using == ’entropy’:
            parent_entropy = self._entropy(y)
            left_idxs, right_idxs = self._split(X_column,
                threshold)

            if len(left_idxs) == 0 or len(right_idxs) == 0:
                return 0

            n = len(y)
            n_l, n_r = len(left_idxs), len(right_idxs)
            e_l, e_r = self._entropy(y[left_idxs]),
                self._entropy(y[right_idxs])
            child_entropy = (n_l / n) * e_l + (n_r / n) * e_r

            information_gain = parent_entropy - child_entropy
            return information_gain

        elif self.split_using == ’gini’:
            ...
```

2.2.4 Splitting Criterion

The split function is responsible for executing the split. It simply passes all indexes with values below the threshold to the left node in the case of numeric variables, and the values above the threshold to the right node. For categorical variables, it uses a membership criterion since there is no ordinal relationship between the possible values. In this case it simply checks if the value matches the threshold value or not to determine the left and right splits.

Code block 5: Splitting for categorical and numeric variables

```
def _split(self, X_column, split_thresh):
    try:
        split_thresh = float(split_thresh)
        X_column = X_column.astype(float)
        left_idxs = np.argwhere(X_column <= split_thresh).flatten()
        right_idxs = np.argwhere(X_column > split_thresh).flatten()
    except ValueError:
        split_thresh = str(split_thresh)
        X_column = X_column.astype(str)
        left_idxs = np.argwhere(X_column.astype(str) ==
            split_thresh).flatten()
        right_idxs = np.argwhere(X_column.astype(str) !=
            split_thresh).flatten()

    return left_idxs, right_idxs
```

While much has been said about selecting an optimal split, the optimality of a split is highly dependent on the splitting criterion chosen. Three criteria have been used for splitting, based on user selection. Below are the mathematical notations for each and their implementation in the decision tree.

Entropy: Entropy evaluates how different the values in a split are. The more similar the values are, the lower the entropy. That is, assuming only two values in a sample, if there are 50% of each value, the entropy will be highest, while the entropy will be lowest if all the observations have the same value.

$$\psi_3(p) = -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p)$$

[1]

Code block 6: Entropy function

```
    def _entropy(self, y):
        counts = np.bincount(y.flatten())
        total_count = len(y)
        probabilities = counts / total_count
        probabilities = probabilities[probabilities > 0]

        entropy = -np.sum(probabilities * np.log2(probabilities) +
            (1 - probabilities) * np.log2(1 - probabilities)) / 2
        return entropy
```

Gini Impurity Gini impurity is another commonly used splitting criterion defined as:

$$\psi_2(p) = 2p(1 - p)$$

[1]

Code block 7: Gini function

```
    def _gini(self, y):
        counts = np.bincount(y.flatten())
        total_count = len(y)
        probabilities = counts / total_count
        probabilities = probabilities[probabilities > 0]

        gini = 2 * np.sum(probabilities * (1 - probabilities))
        return gini
```

Misclassification Error Misclassification error bases the splitting criterion on the number of samples that are wrongly classified. This can be considered as the "default" criterion for splitting trees. However, using a function this simple sometimes runs into errors where the tree is unable to find a split, or results in suboptimal splits. Hence, methods like gini and entropy were introduced to solve these problems. However, it has been included in this project as a kind of baseline, and evaluate how much of an improvement gini and entropy really bring to the table.

Code block 8: misclassification error

```
def _class_error(self, y):
    counter = Counter(y.flatten())  # Convert to list
    most_common_label, count = counter.most_common(1)[0]
    return 1 - (count / len(y))
```

2.2.5 Tree Growth

The tree is grown using the `fit()` and `_grow_tree()` methods. The `fit()` method initializes the process, while `_grow_tree()` handles the recursive splitting until the stopping criteria are met.

Code block 9: Tree Growth Process

```
    def fit(self, X, y):
        X = np.array(X)
        y = np.array(y)
        self.leaves = []
        self.n_features = X.shape[1] if not self.n_features else
            min(X.shape[1], self.n_features)
        self.root = self._grow_tree(X, y)

    def _grow_tree(self, X, y, depth=0):
        n_samples, n_feats = X.shape
        n_labels = len(np.unique(y))

        if depth >= self.max_depth or n_labels == 1 or n_samples <
            self.min_samples_split:
            leaf_value = self._most_common_label(y)
            self.leaves.append(leaf_value)
            return Node(value=leaf_value)

        feat_idxs = np.random.choice(n_feats, self.n_features,
            replace=False)
        best_feature, best_thresh, best_gain = self._best_split(X,
            y, feat_idxs)
        if best_gain == 0:
            leaf_value = self._most_common_label(y)
            self.leaves.append(leaf_value)
            return Node(value=leaf_value)

        left_idxs, right_idxs = self._split(X[:, best_feature],
            best_thresh)
        left = self._grow_tree(X[left_idxs, :], y[left_idxs],
            depth + 1)
        right = self._grow_tree(X[right_idxs, :], y[right_idxs],
            depth + 1)
        return Node(best_feature, best_thresh, left, right)
```

2.3 Prediction

The `predict()` method traverses the tree from the root to make predictions on new data. It uses the decision rules stored in the nodes to navigate through the tree until it reaches a leaf node, where it returns the predicted value.

Code block 10: Predict method

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

    def _traverse_tree(self, x, node):
        if node.is_leaf_node():
            return node.value

        if x[node.feature] <= node.threshold:
            return self._traverse_tree(x, node.left)
        return self._traverse_tree(x, node.right)
```

3 Utility Functions

The `utils.py` file contains utility functions that handle all other activities different from building the decision tree. These include:

1. Evaluation Metrics
2. Hyperparameter Tuning
3. Data Preprocessing

3.1 Evaluation Metrics

The evaluation metrics considered important for assessing the performance of the decision tree are accuracy, recall, precision, and zero-one loss. Accuracy defines the overall performance of the model. That is, how many values are predicted correctly. Precision and recall add more nuance to the performance evaluation, checking if the model is better at predicting positives or negatives. More discussion on this is included in the model comparison section where we evaluate the results of our models against these metrics.

Code block 11: Evaluation Metrics

```
def accuracy(y_actual, y_predicted):
    correct_predictions = np.sum(y_actual == y_predicted)
    accuracy = correct_predictions / len(y_actual)
    print(f"Accuracy: {accuracy}")
    return accuracy

def recall(y_actual, y_predicted):
    actual_positives = y_actual==1
    actual_negatives = y_actual==0
    predicted_positives = y_predicted==1
    predicted_negatives = y_predicted==0
    true_positives = np.sum(actual_positives & predicted_positives)
    false_negatives = np.sum(actual_positives &
        predicted_negatives)
    recall = true_positives/(true_positives+false_negatives)
    print(f"Recall: {recall}")
    return recall

def precision(y_actual, y_predicted):
    actual_positives = y_actual==1
    actual_negatives = y_actual==0
    predicted_positives = y_predicted==1
    predicted_negatives = y_predicted==0
    true_positives = np.sum(actual_positives & predicted_positives)
    false_positives = np.sum(actual_negatives &
        predicted_positives)
    precision = true_positives/(true_positives+false_positives)
    print(f"Precision: {precision}")
    return precision
```

3.2 Zero-One Loss

The `zero_one_loss()` function computes the error rate. It assigns a value of 1 to incorrect predictions and 0 to correct predictions. This essentially gives us the proportion of incorrect predictions. This metric is useful for training error and validation error evaluation, and comes in handy for hyperparameter tuning.

Code block 12: Zero-One Loss Calculation

```
def zero_one_loss(y_train, y_pred):
    incorrect_predictions = np.sum(y_train != y_pred)
    training_error = incorrect_predictions / len(y_train)
    return training_error
```

3.3 Hyperparameter Tuning

The `tune()` function allows for an exploration of hyperparameters to minimize the training error, thereby optimizing the model's performance. It follows the idea of `GridSearchCV`, but for simplicity, the tuning is performed on a single parameter.

Code block 13: Hyperparameter Tuning Function

```
    def evaluate_model(X_train, y_train, X_validate, y_validate,
        tune_on, split_using, i, cv_index):
        params = {tune_on: i, ’split_using’: split_using}
        tree = DecisionTree(**params)
        tree.fit(X_train, y_train)
        y_pred = tree.predict(X_validate)
        validation_error = zero_one_loss(y_validate, y_pred)
        print(f"Tree depth: {tree.max_depth}, Shuffle index:
            {cv_index}, Validation error: {validation_error}")
        return validation_error, i

    def tune(X, y, tune_on, split_using, start, stop,
        n_shuffles=5, n_jobs=-1):
        results_per_depth = {i: [] for i in range(start, stop)}

        for cv_index in range(n_shuffles):
            permuted_indices = np.random.permutation(X.shape[0])
            X_shuffled = X[permuted_indices]
            y_shuffled = y[permuted_indices]
            X_train, X_validate, y_train, y_validate =
                train_test_split(X_shuffled, y_shuffled,
                test_size=0.2)
            # Evaluate the model for each depth with
                parallelization
            results = Parallel(n_jobs=n_jobs)(
                delayed(evaluate_model)(X_train, y_train,
                    X_validate, y_validate, tune_on, split_using,
                    i, cv_index)
                for i in range(start, stop)
            )

            # Store the results for this shuffle
            for error, depth in results:
                results_per_depth[depth].append(error)

        mean_errors = {depth: np.mean(errors) for depth, errors in
            results_per_depth.items()}

        best_depth = min(mean_errors, key=mean_errors.get)
        best_error = mean_errors[best_depth]
        best_tree = DecisionTree(max_depth=best_depth,
            split_using=split_using)

        print(f"Best depth: {best_depth}, Split criterion:
            {split_using}, Mean validation error:
            {round(best_error * 100, 2)} %")
        return best_tree, [(f"tree_depth: {depth}",
            f"mean_validation_error: {error}") for depth, error in
            mean_errors.items()]
```

4 Results and Discussion

4.1 Overview of the Experiments

Experiments were performed to analyze the performance of the model under the three different splitting criterion. This section will give a brief overview of the major decisions taken during the data preprocessing phase, and then an analysis of the experiments follows. Thus, it includes the following sections:

- Data Preprocessing
- Model building and comparison
- Hyperparameter tuning on one of the models

Refer to the linked notebook for a step by step detail of the code and results.

4.2 Data Preprocessing

The data preprocessing stage included the following steps:

- Handling Missing Values/Feature Selection
- Numeric Encoding

4.2.1 Handling Missing Values

Many of the columns do not have any missing values, and this includes all of the numeric columns. However, missing values were present in the following columns

1. veil-type: 94.8%
2. spore-print-color: 89.6%
3. veil-color: 87.9%
4. stem-root: 84.4%
5. stem-surface: 62.4%
6. gill-spacing: 41%
7. cap-surface: 23.1%
8. gill-attachment: 16.2%
9. ring-type: 4%

All columns with more than 80% missing values were dropped as the sparse information is unlikely to influence the model in any meaningful way.

For the other columns two options were considered for handling the missing values:

1. **Mode imputation**: This would involve filling the missing values with the value of the most common value. The problem with this method is that it could lead to an overweighting of these mode values. For example, in the cap-surface column, the mode is 't' with 13.4% while the second highest value is 's' with 12.5%. Mode imputation with 23% missing values will lead to the proportion of the first to second highest value becoming almost 3:1 while in reality, the proportions are only slightly greater than 1:1. Research also shows that mode imputation doesn't improve the performance of classification models [3].

2. **Treat null as category**: The other option which was adopted was to treat the null as a separate category. To avoid excessive loss of data, and considering that certain information could be missing or difficult to verify for certain species, the presence or absence of data could be predictive in the model. Hence, the choice was made to treat null values as a separate category. This was done by forcing the columns to string, effectively codifying the null values as string with value 'nan'

   Code block 14: Forcing categorical variables to string

   ```
   for col in X:
   X.loc[:,col]=X.loc[:,col].astype(str) if
       X.loc[:,col].dtype == 'object' else X.loc[:,col]
   ```

4.3 Numeric Encoding

Decision trees are capable of handling both categorical and numeric variables. However, encoding could be a boost to performance. One-hot encoding would be inappropriate as it would lead to an unnecessary proliferation of columns. Label encoding on the other hand would be appropriate for columns where the categorical values have an ordinal nature. However, none of the columns in the model have this characteristic, so none of the features were encoded.

However, the target variable is encoded numerically since it is a binary variable.

4.4 Model Training and Evaluation

A tree was built on each of the splitting criterion, adopting a maximum depth of 10 for the trees. The following were the results

Code block 15: Entropy Tree

```
entropy_model = DecisionTree(split_using='entropy',
    max_depth=10)
entropy_model.fit(X_train, y_train)
entropy_pred = entropy_model.predict(X_test)
print(accuracy(y_test, entropy_pred))
print(precision(y_test, entropy_pred))
print(recall(y_test, entropy_pred))
```

```
Accuracy: 90.34%
Precision: 93.05%
Recall: 89.43%
```

The results above show that using entropy as the splitting criterion performs better on precision than other metrics. In the context of this model however, it may be a better idea to prioritize recall (i.e. have a higher proportion of false positives). In a real world application of this model, it would be better to discard edible mushroom and suffer some economic loss for example, rather than suffer the damage to health or loss of life that could result from false negatives.

Code block 16: Gini Tree

```
    gini_model = DecisionTree(split_using='gini', max_depth=10)
    gini_model.fit(X_train, y_train)
    gini_pred = gini_model.predict(X_test)
    print(accuracy(y_test, gini_pred))
    print(precision(y_test, gini_pred))
    print(recall(y_test, gini_pred))
```

```
    Accuracy: 90.42%
    Precision: 95.43%
    Recall: 87.06%
```

Splitting on Gini gives a similar performance in terms of accuracy. However, it performs better than entropy on precision but worse on recall. Just like for entropy, an improvement to this model would be to incentivize the identification of positives more highly such that the model will have a higher recall, even at the expense of more false positives. However, entropy performs a better job on recall than gini.

Code block 17: Misclassification error Tree

```
class_error_model = DecisionTree(split_using='class_error',
    max_depth=10)
class_error_model.fit(X_train, y_train)
class_error_pred = train_error_model.predict(X_test)
print(accuracy(y_test, class_error_pred))
print(precision(y_test, class_error_pred))
print(recall(y_test, class_error_pred))
```

```
    Accuracy: 80.7%
    Precision: 87.93%
    Recall: 75.96%
```

As expected, using the misclassification error as a splitting criterion produces the worst performance overall. This is because entropy and gini functions were designed as improvements over the misclassification error which in certain cases may fail to find a split. This means that the misclassification error must have failed to find the optimal split in several situations, causing it take longer to run, while also failing to achieve a higher score.

The expectation before starting the analysis was $\psi_1 < \psi_2 < \psi_3$ This hypothesis was not sufficiently proven in this implementation, given that entropy (ψ_3) and gini (ψ_2) performed very similarly on the test set. This may be because the dataset was carefully curated such that it was very easy for both models to achieve high accuracy scores on test data. On training data however, entropy performed slightly better as seen in the next section.

Absence of Overfitting Here we evaluate for any signs of overfitting. In simple terms, overfitting could be said to occur where the model performs worse on test data relative to training data. Hence to check if there is overfitting, we compare the training and test errors for each of the models.

Code block 18: Evaluating training errors

```
entropy_train = entropy_model.predict(X_train)
gini_train = gini_model.predict(X_train)
class_error_train = class_error_model.predict(X_train)
print(f"entropy training error: {zero_one_loss(y_train,
    entropy_train)}")
print(f"gini training error: {zero_one_loss(y_train,
    gini_train)}")
print(f"class_error training error: {zero_one_loss(y_train,
    class_error_train)}")
```

```
entropy training error: 9.04%
gini training error: 9.42%
class_error training error: 18.49%
```

The test errors on the other hand can be calculated by deducting the accuracy from 100%. Hence for each of the models we have the test errors as:

```
Entropy test error: 9.66%
Gini test error: 9.58%
Class\_error test error: 19.3%
```

Generally, the models performed very similarly on the test data compared to the training data with a difference of less than 1% in all three models. This shows that the stopping criterion appropriately helped to avoid overfitting on the data.

4.5 Hyperparameter Tuning

Hyperparameter tuning was performed to optimize model performance further. For the purpose of this analysis, the `tune` function was used to tune the model on the maximum depth parameter, using gini as the splitting criterion because it had the best performance on the single tree. The tuning involved a 5-fold cross validation process on a single parameter: maximum depth. The tree with the best performance is then retrained based on the full training dataset.

Code block 19: Hyperparameter Tuning

```
tuned_tree = tune(X_train, y_train,
    tune_on='max_depth',split_using='gini', start=19, stop=31)
```

```
    [('tree_depth: 19', 'mean_validation_error:
        0.003909528195681097'),
     ('tree_depth: 20', 'mean_validation_error:
         0.002619997953126599'),
     ('tree_depth: 21', 'mean_validation_error:
         0.0021492170709241634'),
     ('tree_depth: 22', 'mean_validation_error:
         0.001985467198853751'),
     ('tree_depth: 23', 'mean_validation_error:
         0.0018012485927745368'),
     ('tree_depth: 24', 'mean_validation_error:
         0.0018421860607921402'),
     ('tree_depth: 25', 'mean_validation_error:
         0.0018831235288097434'),
     ('tree_depth: 26', 'mean_validation_error:
         0.0016989049227305291'),
     ('tree_depth: 27', 'mean_validation_error:
         0.0019649984648449493'),
     ('tree_depth: 28', 'mean_validation_error:
         0.0018217173267833387'),
     ('tree_depth: 29', 'mean_validation_error:
         0.0019035922628185449'),
     ('tree_depth: 30', 'mean_validation_error:
         0.0017398423907481323')]
```

During tuning model continued to show an improved performance up until a `max_depth` of 26. After this point, the validation error started to increase, showing that the model had started to overfit. The optimal tree was then retrained on the entire training set and tested on the test set.

Code block 20: Retraining the best tree

```
    #retraining the tree with the best validation error on the
        entire dataset
    tuned_tree.fit(X_train, y_train)
    y_pred = tuned_tree.predict(X_test)
    ...
    Accuracy: 99.79%
    Precision: 99.85%
    Recall: 99.77%
```

The tuned model shows a significant improvement in all evaluation metrics with a performance of over 99% on each one. This shows how tuning can be used to achieve optimal results with decision trees.

Another point of note is that the amount of improvement was very little for subsequent trees near the minimum. While in this experiment, the tuning ranges were carefully selected to find the optimal point before the tree starts to overfit, in practice, one may consider using a slightly less accurate model, depending on the use case. For example, if the target is a 99% performance, a tree with depth 19 above or even lesser would achieve the goal just as well as the tree with depth 26 that was finally chosen, while saving time and machine power that would be involved in extending the tree to 26 levels.

While this was limited to one parameter, the tuning was able to showcase the kind of tradeoffs that data scientists would consider in practice when choosing models.

4.6 Conclusion

This article covered the entire process of building a decision tree, starting from writing a function that builds the trees, to data preprocessing, model building and hyperparameter selection. It showcased how a decision tree is built and the tradeoffs involved in the selection of parameters and trees. All code can be found in this github repo

References

[1] Nicol‘o Cesa-Bianchi. Tree predictors. Machine Learning — Statistical Methods for Machine Learning - University of Milan, March 2024. Unpublished course material.

[2] Assembly AI Github. Machine learning from scratch. https://github.com/AssemblyAI-Community/Machine-Learning-From-Scratch/blob/main/04

[3] Shaheen MZ. Memon, Robert Wamala, and Ignace H. Kabano. A comparison of imputation methods for categorical data. *Informatics in Medicine Unlocked*, 42:101382, 2023.