Decision Tree from Scratch with Mushroom Dataset

Mubarak Oluwanishola Babslawal

Matriculation Number: 32189A

Machine Learning (DSE)

Date: September 1, 2024

University of Milan
Data Science in Economics Master

Plagiarism Declaration

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. 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

Listing 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. Training error It also supports three stopping criteria: 1. Minimum number of samples to be split at each node 2. The maximum depth of the tree 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 the desired (stopping criteria) including the minimum number of samples required to split, the maximum depth of the tree, the number of features to consider for each split, and the criterion used for splitting. 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

Listing 2: DecisionTree Class Initialization

```
class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=100,
        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
```

2.3 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.

Listing 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
```

2.3.1 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 through the information gain for each possible split and selects the best.

Listing 4: Splitting and Information Gain Calculation

```
def _information_gain(self, y, X_column, split_thresh):
    if self.split_using == "entropy":
        parent_loss = entropy(y)
    elif self.split_using == "gini":
        parent_loss = gini(y)
    elif self.split_using == "train_error":
        parent_loss = zero_one_loss(y, y)

    left_idxs, right_idxs = self._split(X_column, split_thresh)
    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)
    if self.split_using == "entropy":
        e_l, e_r = entropy(y[left_idxs]), entropy(y[right_idxs])
    elif self.split_using == "gini":
        e_l, e_r = gini(y[left_idxs]), gini(y[right_idxs])
    elif self.split_using == "train_error":
        e_l, e_r = zero_one_loss(y[left_idxs], y[left_idxs]),
            zero_one_loss(y[right_idxs], y[right_idxs])
    child_loss = (n_l / n) * e_l + (n_r / n) * e_r

    ig = parent_loss minus child_loss
    return ig
```

2.3.2 Splitting Criterion

The split function is responsible for executing the split. It simply passes the 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.

Listing 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:

    #since all thresholds have been converted into strings, for
        correct handling of nulls, X_column will need to be
        converted into strings too
        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 determined by the splitting criterion. Three criteria have been used for splitting, as determined by the user. Below are the mathematical notations for each and their implementation in the decision tree.

Entropy Entropy determines 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. Logarithm is used for scaling, since the logarithm in base 2 has the same highest point for entropy. This ensures the same result.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

In the code, `Entropy` is defined as follows:

Listing 6: Entropy function

```
def _entropy(self, y):
    counter = Counter(y.flatten())  # Convert to list
    ps = np.array([count / len(y) for count in counter.values()])
    return -np.sum([p * np.log2(p) for p in ps if p > 0])
```

Gini Impurity Gini impurity is based on the same idea as entropy. However, the implementation is different. Instead of using log_2, gini scales the values by squaring the probabilities (or proportions)

$$Gini(X) = 1 - \sum_{i=1}^{n} p(x_i)^2$$

Listing 7: Gini function

```
def _gini(self, y):
    counter = Counter(y.tolist())  # Convert to list
    ps = np.array([count / len(y) for count in counter.values()])
    return 1 - np.sum(ps ** 2)
```

Train Error Train error bases the splitting criterion on the number of samples that would be predicted wrongly if a single leaf value were applied to the node. For example, if there is a 60-40 split between the samples, 40% of the values would have been predicted wrongly. Thus the `train error` for that split would be 40%. The algorithm then works through the different possible splits to select the one with the lowest train error. This is implemented as below:

Listing 8: train error (as a split criterion)

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

2.3.3 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.

Listing 9: Tree Growth Process

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

```python
    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)
```

2.4 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.

Listing 10: Prediction Method

```python
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 provides several 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

This file contains functions for computing accuracy, recall, precision, and zero-one loss. These metrics are vital for assessing the model's performance.

Listing 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.

Listing 12: Zero-One Loss Calculation

```python
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.

Listing 13: Hyperparameter Tuning Function

```python
def tune(X, y, tune_on, limit):
    best_training_error = 1
    for i in range(1,limit):
        params = {tune_on: i}
        tree = DecisionTree(**params)
        tree.fit(X, y)
        y_pred = tree.predict(X)
        training_error = zero_one_loss(y, y_pred)
        if training_error == best_training_error:
            break
        if training_error < best_training_error:
            best_training_error = training_error
            best_tree = tree
    print(f"training error: {best_training_error}")
    return best_tree
```

4 Results and Discussion

4.1 Overview of the Experiments

Experiments were performed to analyze the performance of the model under various 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. Refer to the notebook linked in the appendix 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
- Numeric encoding

handling missing values, encoding categorical variables, and feature scaling. These steps were crucial to ensure that the dataset was in a suitable format for model training.

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 are particularly prominent in

1. veil-type: 94.8%
2. spore-print-color: 89.6%
3. veil-color: 87.9%
4. stem-root: 84.4% Other columns with missing values include:
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 I considered two options 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 x with 23% while the second highest value is with 21%. Mode imputation with 23% missing values will lead to the proportion of the first to second highest value becoming more than 2:1 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. Cite paper here

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'

Listing 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 we do not encode any of the predictors 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

Listing 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: 87.11%
    Precision: 93.40%
    Recall: 82.83%
```

Listing 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: 92.6%
    Precision: 97.06%
    Recall: 89.49%
```

Listing 17: Train error Tree

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

```
    Accuracy: 92.61%
    Precision: 97.08%
    Recall: 89.49%
```

Based on the above results, the train error model and the gini impurity models have similar performances across all evaluation metrics, while the entropy model perofrms worse on all three metrics.

It is also interesting to note that the precision is 97% which is higher than recall. This means that the system is optimized to catch all possible positives (poisonous mushrooms), at the expense of having a higher proportion of false positives (hence a lower recall). This is desirable since it is better to identify an edible mushroom as poisonous, and so it isn't eaten, than to make the error of identifying a poisonous mushroom as edible, which could be dangerous.

The next step however is to evaluate the presence of overfitting. 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 take a look at the training error for each of the models

Listing 18: Evaluating training errors

```
entropy_train = entropy_model.predict(X_train)
gini_train = gini_model.predict(X_train)
train_error_train = train_error_model.predict(X_train)
print(f"entropy train error: {zero_one_loss(y_train,
    entropy_train)}")
print(f"gini train error: {zero_one_loss(y_train,
    gini_train)}")
print(f"train_error train error: {zero_one_loss(y_train,
    train_error_train)}")
```

```
entropy train error: 11.95%
gini train error: 6.69%
train_error train error: 6.69%
```

The test errors on the 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: 12.89%
- Gini test error: 7.4%
- Train_error test error: 7.4%

Generally, the models performed worse on the test data compared to the training data. However, this performance difference is very small (less than 1%) and could easily ignored. This shows that the stopping criterion was valued appropriately 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 previous phase, and also because the model had a shorter training time. During the tuning, the training set is split into a training and validation set. This ensures that the models are evaluated based on their performance on data they haven't seen, rather than picking the most overfitted tree. The tree with the best performance is then retrained based on the full training dataset

The retrained tree achieves an accuracy of 99.77% on the test dataset

Listing 19: Hyperparameter Tuning

```
tuned_tree = tune(X_train, y_train,
    tune_on='max_depth',split_using='gini', start=15, stop=25)
```

```
    tree depth: 15, validation error: 0.010336710674444785
    tree depth: 16, validation error: 0.008801555623784669
    tree depth: 17, validation error: 0.006242963872684475
    tree depth: 18, validation error: 0.0066523385528605055
    tree depth: 19, validation error: 0.005935932862552451
    tree depth: 20, validation error: 0.004707808822024357
    tree depth: 21, validation error: 0.005117183502200389
    tree depth: 22, validation error: 0.005117183502200389
    tree depth: 23, validation error: 0.005833589192508443
    tree depth: 24, validation error: 0.004810152492068366
    training error: 0.004707808822024357
```

The tuned model continued to show an improved performance up until a max_depth of 20. After this point, the validation error started to increase, showing that the model had started to overfit.

A new tree was then trained using the same parameters from the best performing tree in the previous experiment, but using the full training set. This tree was then tested on the full test set.

Listing 20: Retraining the best tree

```
    #retraining the tree with the best validation error
    best_tree = DecisionTree(max_depth=20, split_using='gini')
    best_tree.fit(X_train, y_train)
    y_pred = best_tree.predict(X_test)
    print(accuracy(y_test, y_pred))
    print(precision(y_test, y_pred))
    print(recall(y_test, y_pred))
```

```
    Accuracy: 99.77%
    Precision: 99.81%
    Recall: 99.79%
```

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. While this was limited to one parameter, the use of several parameters for hyperparameter tuning can also help find smaller trees that achieve optimal performance

4.6 Conclusion

The experiments demonstrated that careful preprocessing, feature selection, and model tuning are essential for achieving high-performance machine learning models. The random forest model, in particular, stood out for its robust performance, making it well-suited for the classification tasks associated with the mushroom dataset (Section 6.1).

4.7 Appendix

4.7.1 The github repo

The github repo containing all code and analysis discussed can be found here: Repo Notebook with analysis