UTS

University of Technology Sydney

Student name: Thanh Thuy An Tran

Student ID: 14218768

32513 Advanced Data Analytics Algorithms

Assessment 2: Algorithm Implementation, Journal and Presentation – Decision Tree Classification Model

Executive Summary

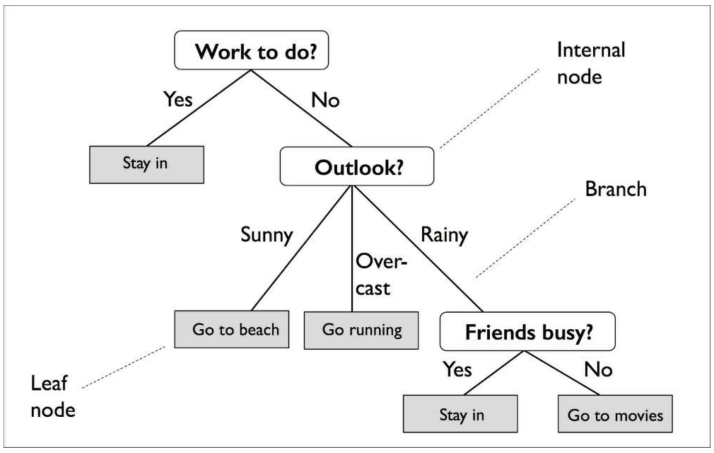
This comprehensive report provides a detailed exploration of decision tree classification, a powerful and widely used machine learning technique for solving classification problems. Decision trees offer interpretability, adaptability to various data types, and the ability to capture complex decision boundaries. The report covers the core concepts, algorithmic details, splitting criteria, pruning techniques, advantages, limitations, and real-world applications of decision tree classification.

Contents

1. Introduction

Decision tree classification is a supervised machine learning algorithm that is widely used for solving classification problems. It builds a tree-like model of decisions and their possible consequences, making it a versatile tool in predictive modelling across various domains.

This method classifies a population into branch-like segments that construct an inverted tree with a root node, internal nodes, and leaf nodes. The algorithm is non-parametric and can efficiently deal with large, complicated datasets without imposing a complicated parametric structure.



2. Decision Tree Basics

A decision tree is a hierarchical structure composed of nodes, branches, and leaves. Key terms include:

- **Root Node:** The top node representing the initial decision.
- **Internal Nodes:** Nodes that represent intermediate decisions.
- **Leaf Nodes:** Terminal nodes that provide the final classification or prediction.
- **Edges:** Connections between nodes, representing possible outcomes.

3. Decision Tree Algorithm

3.1. Initialization

- **Input Data:** The algorithm starts with a labeled dataset. Each data point is associated with a class label, indicating the correct classification.
- **Attributes:** The dataset consists of features or attributes that describe each data point. These attributes guide the decision-making process.
- **Stopping Criteria:** The algorithm defines stopping criteria to determine when to halt the tree-building process. Common stopping criteria include a maximum depth for the tree, a minimum number of samples required to split a node, or when all data points in a node belong to the same class.
- **Root Node:** The root node represents the entire dataset.

3.2. Selecting the Best Attribute (Feature Selection)

- **Measure of Impurity:** To decide which attribute to split on, the algorithm uses a measure of impurity or uncertainty. Common measures include Gini Impurity, Information Gain, or Entropy.

- **Splitting Criteria:** The algorithm evaluates each attribute based on the selected measure of impurity. It calculates the impurity for each possible split and selects the attribute that minimizes impurity (maximizes information gain) as the best attribute for splitting.

3.3. Splitting the Data

- **Branches:** Once the best attribute is selected, the dataset is split into subsets, each corresponding to a unique value of the selected attribute. This creates branches emanating from the current node.
- **Internal Nodes:** These new branches lead to internal nodes in the decision tree, each representing a decision based on the attribute's value.
- **Recursive Process:** The algorithm then recursively applies the same splitting process to each subset of data associated with the internal nodes. This process continues until one of the stopping criteria is met.

3.4. Handling Leaves (Terminal Nodes)

- **Leaf Nodes:** When a stopping criterion is met, a leaf node is created. Leaf nodes represent the final decision or prediction for a specific class.
- **Class Assignment:** The class label assigned to a leaf node is typically determined by a majority vote. In a binary classification problem, the class with more instances in the leaf node is selected as the predicted class.

3.5. Pruning (Optional)

- **Overfitting Prevention:** To prevent overfitting, pruning techniques may be applied. Pruning involves removing branches (subtrees) that do not significantly improve the tree's accuracy on validation data.

3.6. Final Decision Tree

- **Tree Completion:** Once the tree-building process is complete, a decision tree structure is formed with root nodes, internal nodes, and leaf nodes. This tree represents a set of decisions and predictions based on the input attributes.
- **Predictions:** To make predictions on new, unseen data, the algorithm follows the tree from the root node down to a leaf node, where the final class label is assigned.

The decision tree algorithm is known for its interpretability, as the resulting tree structure can be easily visualized and understood. It excels in capturing complex decision boundaries, making it a valuable tool for a wide range of classification tasks across various domains. However, it is important to address issues like overfitting through pruning and careful consideration of hyperparameters to ensure optimal performance in practice.

4. Splitting Criteria

4.1. Entropy

Entropy is a fundamental concept in decision trees and machine learning that quantifies the degree of disorder or mixing within a dataset. It helps decision trees make informed splits by measuring the impurity of data. Decision trees use entropy to determine how to split data at each level. The objective is to find splits that maximize "information gain," reducing entropy as much as possible. This ensures that splits result in subsets that are as pure as possible.

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

In the context of decision trees and machine learning:

- **Low Entropy**: If a dataset contains only one class (e.g., all red balls), it has low entropy because it's pure and not mixed up.
- **High Entropy**: If a dataset contains an equal number of multiple classes (e.g., a mix of red, blue, and green balls), it has high entropy because it's mixed up and impure.

In decision trees, we use entropy to decide how to split data at each level. The goal is to find splits that reduce the entropy as much as possible. This reduction in entropy is called "information gain." We want to split in a way that makes the classes purer on each side of the split. In summary, entropy helps decision trees figure out how to split data in a way that makes the resulting subsets as pure (homogeneous) as possible, which ultimately helps the tree make accurate predictions. It's a measure of disorder or uncertainty in your data.

4.2. Gini Impurity

Gini impurity is another crucial concept in decision trees, focusing on measuring the diversity or mixing of categories within a group. In decision trees, the goal is to minimize Gini impurity when making splits. By choosing attributes that result in groups with less mixing of categories, decision trees create purer subsets.

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

When building a decision tree, we want to split the data in a way that makes the groups as pure as possible. So, Gini impurity helps decision trees find the best way to divide data by picking the attribute that results in groups with the least mixing of categories. When the impurity decreases, it means the tree is making better decisions. In short, Gini impurity is like a tool that helps decision trees figure out the best way to split data into groups by focusing on minimizing mixing and maximizing purity. It helps the tree make smart decisions about how to categorize things.

When building a decision tree, we calculate the information gain for each attribute and select the attribute that gives the highest information gain as the splitting criterion at that node. This process is repeated recursively for each new branch, creating a tree structure that effectively separates the data based on the selected attributes.

4.3. Information gain

$$IG(Q) = S_O - \sum_{i=1}^{q} \frac{N_i}{N} S_i,$$

5. Crucial Tree Parameters

Technically, we can build a decision tree until each leaf has exactly one instance, but this is not common in practice when building a single tree because it will be *overfitted*, or too tuned to the training set, and will not predict labels for new data well. The most common ways to deal with overfitting in decision trees are as follows:

- ***Stopping***: To prevent model become overfitted, stopping rules must be applied when building a decision tree to prevent the model from becoming overly complex. Common parameters used in stopping rules include:
    - (a) the minimum number of records in a leaf.
    - (b) the minimum number of records in a node prior to splitting.
    - (c) the depth (i.e., number of steps) of any leaf from the root node.
- ***Pruning trees***: In some situations, stopping rules do not work well. An alternative way to build a decision tree model is to grow a large tree first, and then prune it to optimal size by removing nodes that provide less additional information by comparing the quality of the tree with and without that partition (comparison is performed using *cross-validation*, more on this below).
    - **Reduced Error Pruning:** Prune branches that do not significantly improve accuracy on a validation dataset.
    - **Minimum Description Length (MDL) Pruning:** Use a cost-complexity measure to trade-off tree complexity and accuracy.

6. Implementation

6.1. Defining Impurity Functions

The decision tree relies on impurity measures to make informed splits. I defined two crucial functions, **entropy** and **gini**, which serve as the basis for assessing impurity or disorder within a set of class labels. These functions capture the essence of decision tree splitting criteria.

```
# Define entropy and gini impurity functions
def entropy(y):
    _, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    return -np.sum(probabilities * np.log2(probabilities + 1e-10))

def gini(y):
    _, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    return 1 - np.sum(probabilities**2)
```

6.2. Mapping Criteria to Functions

Next, I created a dictionary named **criteria_dict**. This nifty mapping tool associates with criterion names, such as "entropy" or "gini," with their corresponding impurity functions. This dictionary simplifies the criterion selection process later on.

```
criteria_dict = {
    "entropy": entropy,
    "gini": gini,}
```

6.3. The Node Class

In the heart of the decision tree lies the **Node** class. Each instance of this class represents a node within the tree structure. It possesses attributes for storing feature indices, thresholds, class labels, and references to left and right child nodes. The **Node** class's existence is vital for accurately modelling the tree's hierarchical structure.

```
class Node:
    def __init__(self, feature_idx=None, threshold=None, labels=None, left=None,
right=None):
        self.feature_idx = feature_idx
        self.threshold = threshold
        self.labels = labels
        self.left = left
        self.right = right
```

6.4. Leaf Value Assignment Function

I implemented a function called **classification_leaf** to determine the class labels assigned to leaf nodes. This function identifies the most frequent class label within a set of labels, thus providing clarity on the predicted class for instances reaching leaf nodes.

```
def classification_leaf(y):
    return np.bincount(y).argmax()
```

6.5. The DecisionTree Class

The core of the decision tree classifier is the **DecisionTree** class. This class encapsulates the entire decision tree construction process and includes crucial methods:

- **__init__**: Initializes hyperparameters like **max_depth**, **min_samples_split**, **criterion**, and functions for criterion selection and leaf value assignment.
- **_functional**: Computes the impurity reduction for a potential split based on selected features and thresholds.
- **_build_tree**: Recursively constructs the decision tree by seeking the best splits and creating child nodes.
- **fit**: Fits the decision tree to the training data by invoking the **_build_tree** method and stores the root node.
- **predict_one** and **predict**: These methods facilitate making predictions. **predict_one** predicts the class label for a single data point by traversing the tree, while **predict** scales this process to make predictions for multiple data points.

```
class DecisionTree:
    def __init__(self, max_depth=np.inf, min_samples_split=2, criterion="gini"):
        # Initialization: Define hyperparameters and criterion function
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.criterion = criterion
        self._criterion_function = criteria_dict[criterion]  # Choose Gini or Entropy as
the impurity measure
```

```python
        self._leaf_value = classification_leaf  # Assign class labels to leaf nodes

    def _functional(self, X, y, feature_idx, threshold):
        # Calculate the impurity reduction for a potential split
        left_mask = X[:, feature_idx] < threshold
        right_mask = ~left_mask
        left_y = y[left_mask]
        right_y = y[right_mask]
        n_obj = X.shape[0]
        n_left = np.sum(left_mask)
        n_right = n_obj - n_left
        if n_left > 0 and n_right > 0:
            return (
                self._criterion_function(y)
                - (n_left / n_obj) * self._criterion_function(left_y)
                - (n_right / n_obj) * self._criterion_function(right_y)
            )
        else:
            return 0

    def _build_tree(self, X, y, depth=1):
        # Recursively build the decision tree
        n_samples, n_features = X.shape

        if len(np.unique(y)) == 1:
            return Node(labels=y)

        if depth >= self.max_depth or n_samples < self.min_samples_split:
            return Node(labels=y)

        max_criterion_value = -float('inf')
        best_split = None

        for feature_idx in range(n_features):
            thresholds = np.unique(X[:, feature_idx])

            for threshold in thresholds:
                criterion_value = self._functional(X, y, feature_idx, threshold)

                if criterion_value > max_criterion_value:
                    max_criterion_value = criterion_value
                    best_split = {
                        'feature_idx': feature_idx,
                        'threshold': threshold
                    }

        if best_split is not None:
            left_mask = X[:, best_split['feature_idx']] < best_split['threshold']
            right_mask = ~left_mask
            left_subtree = self._build_tree(X[left_mask], y[left_mask], depth + 1)
            right_subtree = self._build_tree(X[right_mask], y[right_mask], depth + 1)
            return Node(
                feature_idx=best_split['feature_idx'],
                threshold=best_split['threshold'],
                left=left_subtree,
                right=right_subtree
            )
        else:
            return Node(labels=y)

    def fit(self, X, y):
        # Fit the decision tree to the training data
        self.root = self._build_tree(X, y)

    def predict_one(self, node, X):
        # Recursively predict the class label for one data point
        if node.labels is not None:
            return self._leaf_value(node.labels)
        if X[node.feature_idx] < node.threshold:
            return self.predict_one(node.left, X)
        else:
            return self.predict_one(node.right, X)

    def predict(self, X):
        # Predict class labels for a set of data points
        return [self.predict_one(self.root, x) for x in X]
```

6.6. Model Evaluation

In the final stretch, I executed the main steps. I generated a synthetic classification dataset using **make_classification**, applied the decision tree fitting process to the data, and subsequently utilized the trained tree to make predictions.

```
from sklearn.datasets import (
    load_digits,
    make_classification,
    make_regression,
)
# Generate a make_classification dataset
X, y = make_classification(
    n_features=2, n_redundant=0, n_samples=400, random_state=17
)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=17
)
```

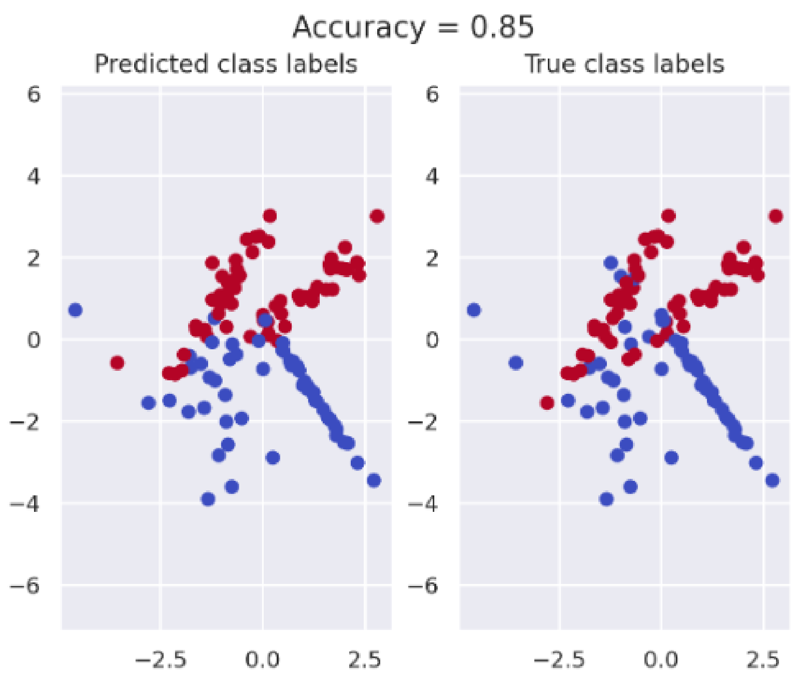
```
# Create and fit the decision tree model
clf = DecisionTree(max_depth=4, criterion="gini")
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Plot the results
plt.suptitle("Accuracy = {:.2f}".format(accuracy))
plt.subplot(121)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap=plt.cm.coolwarm)
plt.title("Predicted class labels")
plt.axis("equal")
plt.subplot(122)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=plt.cm.coolwarm)
plt.title("True class labels")
plt.axis("equal")
plt.show()
```

```
Accuracy: 0.85
```



In conclusion, decision tree classification is a powerful machine learning technique with a broad range of applications. Its ability to create interpretable models, handle mixed data types, and capture complex decision boundaries makes it a valuable tool for data scientists. However, careful consideration of overfitting and data bias is necessary to ensure that decision trees perform effectively in practice.

7. References

- Breiman, L., Friedman, J., Stone, C. J., & Olsen, R. A. (1984). Classification and Regression Trees. Chapman & Hall.
- Quinlan, J. R. (1986). Induction of Decision Trees. Machine Learning, 1(1), 81-106.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer.