Decision Trees: Mathematical Foundations and Implementation

Albert Villanueva

July 9, 2025

Contents

1 Introduction to Decision Trees

Decision trees are supervised learning algorithms used for both classification and regression tasks. They create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. The resulting model is represented as a tree structure where:

- Internal nodes represent tests on attributes
- Branches represent outcomes of tests
- Leaf nodes represent class labels or values

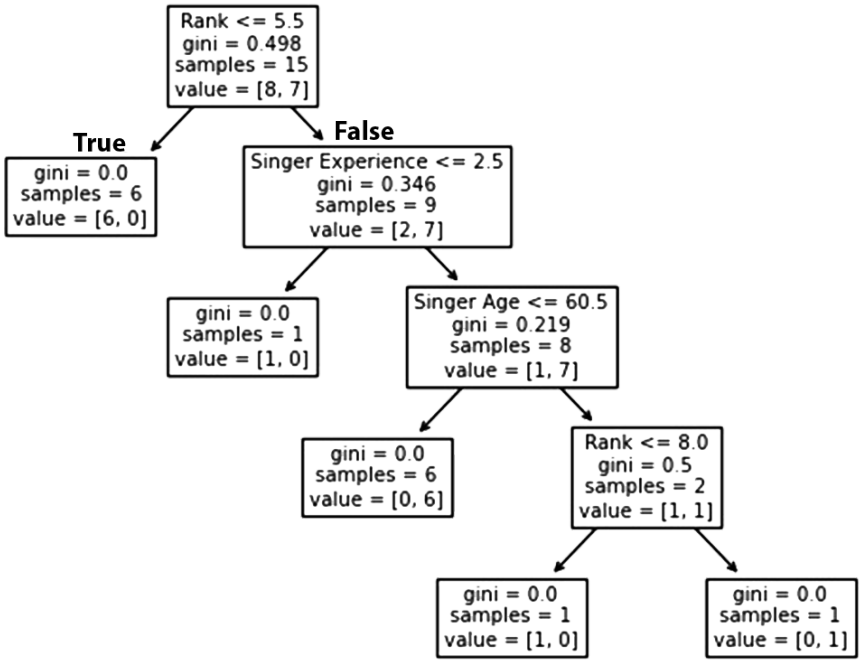

Figure 1: Example of a decision tree structure showing internal nodes with splitting conditions, branches representing outcomes, and leaf nodes with class predictions

2 Mathematical Foundations

2.1 Impurity Measures

The quality of a split in a decision tree is measured using impurity measures. The goal is to find splits that create the most homogeneous (pure) child nodes.

2.1.1 Gini Impurity

The Gini impurity measures the probability of incorrectly classifying a randomly chosen element if it were randomly labeled according to the distribution of labels in the subset.

For a node t with C classes, the Gini impurity is defined as:

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

where p_i is the proportion of samples belonging to class i at node t.

For a binary split creating left and right child nodes, the weighted Gini impurity is:

$$\text{Gini}_{\text{split}} = \frac{n_{\text{left}}}{n} \cdot \text{Gini}(\text{left}) + \frac{n_{\text{right}}}{n} \cdot \text{Gini}(\text{right}) \tag{2}$$

where n_{left} and n_{right} are the number of samples in the left and right child nodes, and $n = n_{\text{left}} + n_{\text{right}}$.

2.1.2 Information Gain and Entropy

Entropy measures the average amount of information needed to identify the class of a sample:

$$\text{Entropy}(t) = -\sum_{i=1}^{C} p_i \log_2(p_i) \tag{3}$$

Information gain measures the reduction in entropy after a split:

$$\text{InfoGain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \cdot \text{Entropy}(S_v) \tag{4}$$

where S is the set of samples, A is the attribute, and S_v is the subset of S for which attribute A has value v.

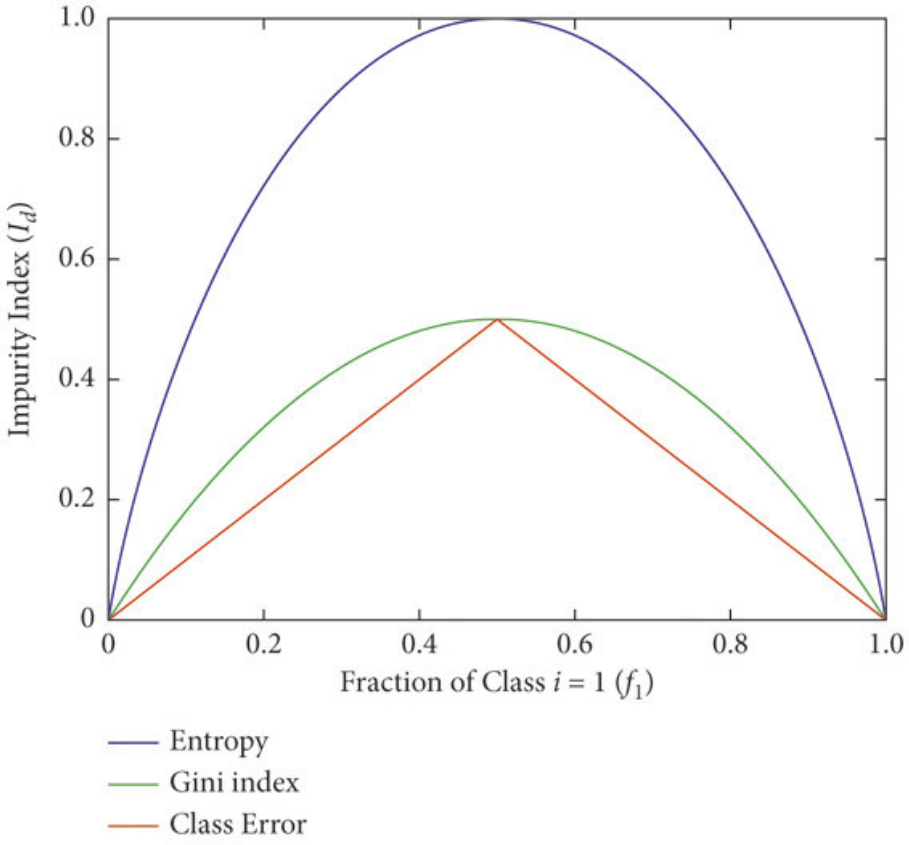

Figure 2: Comparison of Gini impurity and entropy for binary classification. Show curves plotting both measures against the probability of class 1, demonstrating their similar behavior with Gini being slightly faster to compute.

2.2 Splitting Criteria

For continuous features, we need to determine optimal threshold values. For a feature X_j and threshold θ, we create a binary split:

$$\text{Left child: } X_j \leq \theta \tag{5}$$

$$\text{Right child: } X_j > \theta \tag{6}$$

The optimal threshold θ^* minimizes the impurity measure:

$$\theta^* = \arg\min_{\theta} \text{Impurity}_{\text{split}}(\theta) \tag{7}$$

3 Tree Construction Algorithm

The decision tree construction follows a greedy, top-down approach:

Algorithm 1 Decision Tree Construction

function BuildTree(S, Features, depth)
if StoppingCriterion(S, depth) **then**
  return CreateLeaf(S)
end if
bestFeature, bestThreshold ← FindBestSplit(S, Features)
if bestFeature is None **then**
  return CreateLeaf(S)
end if
$S_{\text{left}} \leftarrow \{(x, y) \in S : x_{\text{bestFeature}} \leq \text{bestThreshold}\}$
$S_{\text{right}} \leftarrow \{(x, y) \in S : x_{\text{bestFeature}} > \text{bestThreshold}\}$
leftChild ← BuildTree(S_{left}, Features, depth+1)
rightChild ← BuildTree(S_{right}, Features, depth+1)
return CreateNode(bestFeature, bestThreshold, leftChild, rightChild)

3.1 Stopping Criteria

Common stopping criteria include:

- Maximum depth reached: depth $\geq$ max_depth
- Minimum samples for split: $|S| <$ min_samples_split
- Pure node: All samples belong to the same class
- No improvement in impurity measure

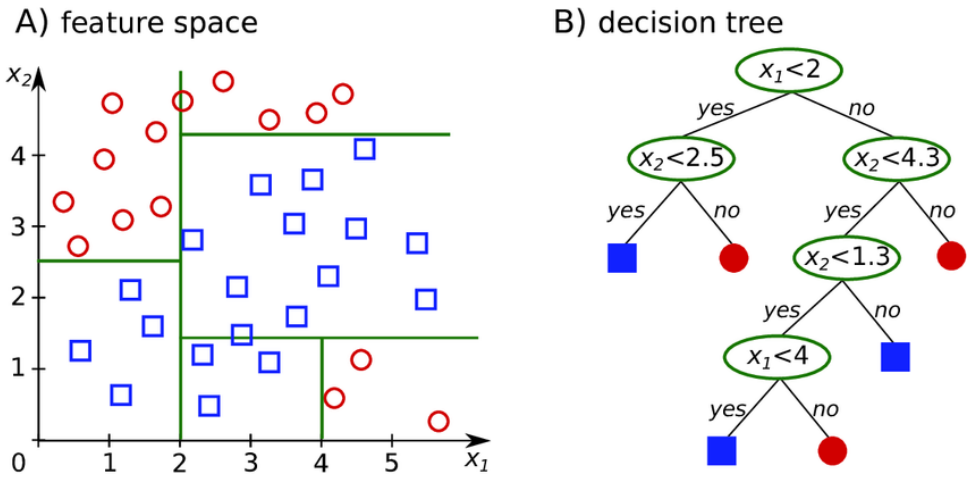

Figure 3: Step-by-step visualization of tree construction showing how the algorithm recursively splits the data space. Display the progressive partitioning of a 2D feature space with decision boundaries at each step.

4 Overfitting and Regularization

Decision trees are prone to overfitting, especially when grown deep. Common regularization techniques include:

4.1 Pre-pruning (Early Stopping)

- Limiting maximum depth
- Setting minimum samples per split/leaf
- Requiring minimum impurity decrease

4.2 Post-pruning

Post-pruning involves growing a full tree and then removing branches that don't provide significant improvement on validation data.

The cost-complexity pruning uses a complexity parameter α:

$$R_\alpha(T) = R(T) + \alpha|T| \tag{8}$$

where $R(T)$ is the error rate of tree T and $|T|$ is the number of leaves.

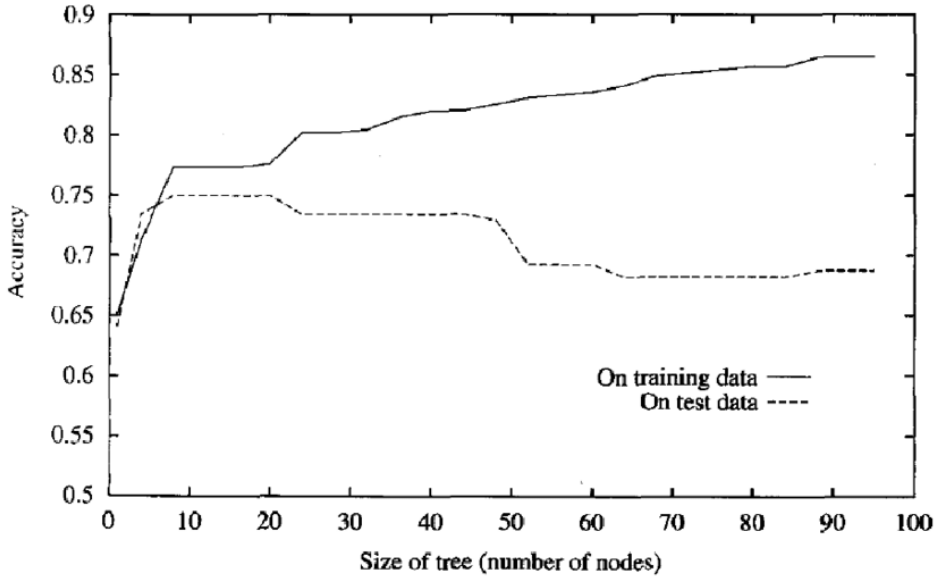

Figure 4: Illustration of overfitting in decision trees. Show training vs validation accuracy curves as tree depth increases, demonstrating how validation accuracy peaks and then decreases while training accuracy continues to improve.

5 Custom Decision Tree Implementation

5.1 Class Structure Overview

The custom implementation consists of two main classes:

- `CustomDecisionTreeNode`: Represents individual nodes in the tree
- `CustomDecisionTree`: The main classifier implementing the decision tree algorithm

5.2 Node Class Analysis

```
class CustomDecisionTreeNode:
    def __init__(self, feature_index=None, threshold=None,
                 left=None, right=None, value=None):
        self.feature_index = feature_index  # Feature used for
            splitting
        self.threshold = threshold          # Threshold value for split
        self.left = left                    # Left child node
        self.right = right                  # Right child node
        self.value = value                  # Leaf node prediction
```

The node class implements a binary tree structure where:

- Internal nodes store `feature_index` and `threshold`
- Leaf nodes store the predicted `value`
- Navigation is handled through `left` and `right` pointers

5.3 Main Classifier Methods

5.3.1 Initialization

```
def __init__(self, max_depth=None, min_samples_split=2):
    self.max_depth = max_depth
    self.min_samples_split = min_samples_split
    self.root = None
```

The constructor accepts hyperparameters for regularization:

- `max_depth`: Limits tree depth to prevent overfitting
- `min_samples_split`: Minimum samples required to create a split

5.3.2 Training Method

The `fit` method initiates the recursive tree building process:

```
def fit(self, X, y):
    self.root = self._build_tree(X, y, depth=0)
```

5.3.3 Tree Construction

The `_build_tree` method implements the recursive algorithm:

$$\text{StoppingCondition} = \begin{cases} \text{True} & \text{if depth} \geq \text{max\_depth} \\ \text{True} & \text{if } |\text{unique}(y)| = 1 \\ \text{True} & \text{if } |y| < \text{min\_samples\_split} \\ \text{False} & \text{otherwise} \end{cases} \tag{9}$$

5.3.4 Best Split Selection

The `_best_split` method finds the optimal feature and threshold by minimizing Gini impurity:

$$(\text{feature}^*, \theta^*) = \arg\min_{\text{feature},\theta} \text{Gini}_{\text{split}}(\text{feature}, \theta) \tag{10}$$

$$\text{where } \theta \in \{\text{unique values of feature}\} \tag{11}$$

5.3.5 Gini Impurity Calculation

The implementation uses the weighted Gini impurity formula:

```
def _gini_index(self, left_y, right_y):
    m_left, m_right = len(left_y), len(right_y)
    m = m_left + m_right

    gini_left = 1.0 - sum((np.sum(left_y == c) / m_left) ** 2
                          for c in np.unique(left_y)) if m_left > 0 else
                              0
    gini_right = 1.0 - sum((np.sum(right_y == c) / m_right) ** 2
                           for c in np.unique(right_y)) if m_right > 0
                               else 0

    return (m_left / m) * gini_left + (m_right / m) * gini_right
```

5.3.6 Prediction Method

Prediction traverses the tree from root to leaf:

Algorithm 2 Tree Traversal for Prediction

function Predict(input, node)
if node.value is not None **then**
  return node.value
end if
if input[node.feature_index] $\leq$ node.threshold **then**
  return Predict(input, node.left)
else
  return Predict(input, node.right)
end if

5.4 Key Implementation Features

1. **Binary Splits Only**: Uses binary splits on continuous features
2. **Gini Impurity**: Uses Gini as the splitting criterion
3. **Greedy Search**: Evaluates all possible thresholds for each feature
4. **Majority Vote**: Leaf nodes predict the most common class
5. **Regularization**: Supports depth limiting and minimum sample constraints

5.5 Computational Complexity

- **Training Time**: $O(n \cdot m \cdot \log n)$ where n is samples and m is features
- **Prediction Time**: $O(\log n)$ for balanced trees, $O(n)$ worst case
- **Space Complexity**: $O(n)$ for storing the tree structure

6 Usage Example

```python
import numpy as np
from sklearn.datasets import make_classification

# Generate sample data
X, y = make_classification(n_samples=1000, n_features=4,
                           n_classes=2, random_state=42)

# Split data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# Train custom decision tree
clf = CustomDecisionTree(max_depth=5, min_samples_split=10)
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Evaluate accuracy
accuracy = clf.score(X_test, y_test)
print(f"Accuracy: {accuracy:.3f}")
```

7 Advantages and Limitations

7.1 Advantages

- Easy to understand and interpret
- Requires little data preparation
- Handles both numerical and categorical data
- Can model non-linear relationships
- Automatically performs feature selection

7.2 Limitations

- Prone to overfitting
- Can create overly complex trees

- Biased toward features with more levels
- Can be unstable (small data changes can result in different trees)
- Difficulty in optimal tree construction (NP-complete)

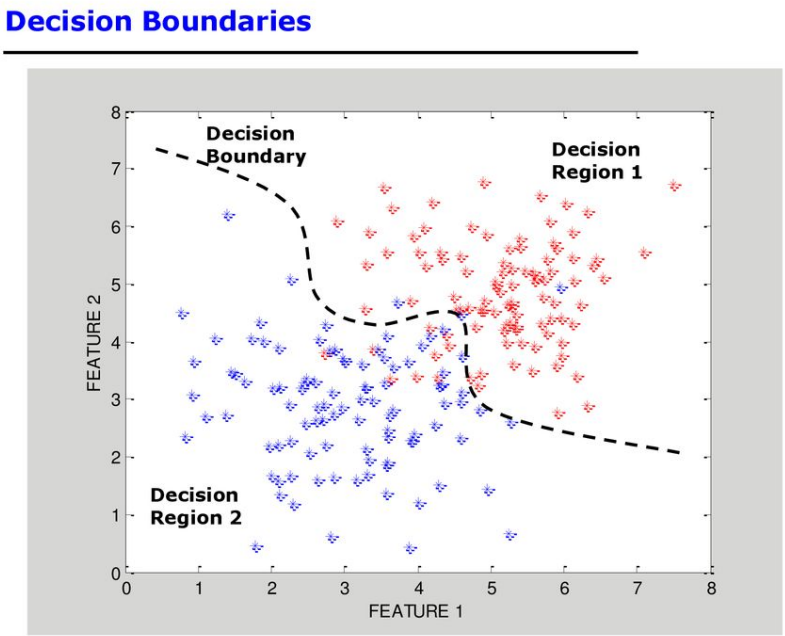

Figure 5: Decision boundaries created by a decision tree on a 2D dataset. Show rectangular regions corresponding to different classes, illustrating how decision trees partition the feature space with axis-parallel splits.

8 Conclusion

Decision trees provide an intuitive and powerful approach to classification problems. The custom implementation demonstrates the core concepts while maintaining simplicity and clarity. The mathematical foundations, particularly the Gini impurity measure and recursive splitting algorithm, form the backbone of more advanced ensemble methods like Random Forests and Gradient Boosting.

Understanding the implementation details helps in:

- Debugging and optimizing tree-based models
- Implementing custom splitting criteria
- Extending to ensemble methods
- Understanding hyperparameter effects

References

[1] Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees*. Wadsworth & Brooks/Cole Advanced Books & Software.

[2] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1), 81-106.

[3] Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.