Classification and Regression Tree for classification

12/12/2025

Team member: Lihao Xu, Zheyu Shi, Haowen Qin, Yaqi Sun

Github repo: https://github.com/Kaitlyn-Sun/Classification-and-Regression-Tree-for-classification

Overview

- Mathematical Intuition
- Implementation Methodology
- Evaluation
- Interesting features and challenges

Mathematical Intuition

Mathematical Framework

- Representation
- Loss function
- Optimizer
- Model evaluation

Representation

- Binary decision tree:
- Each internal node corresponds to a test of the form:

$$x_j < \tau$$

- Each leaf node stores a class distribution and produces a label prediction through:

$$\dot{y} = arg \max_k c_k(t)$$

- Hypothesis class:

$$H = \{\text{All binary trees with axis} - \text{aligned threshold splits}\}$$

Loss Function

- Shannon Entropy:
- For a node containing sample from X classes, the empirical class distribution is:

$$p_k = \frac{c_k}{n}, k = 1\ldots, K$$

- The entropy of node is defined:

$$H(p) = -\sum_{k=1}^{K} p_k log p_k$$

Loss Function

- Gini Impurity:
- It is CART default impurity measurement, defined as:

$$G(p) = 1 - \sum_{k=1}^{K} p_k^2$$

Optimizer

- It uses a greedy, top-down recursive algorithm, often called recursive binary splitting, which usually consists of:

- 1. Greedy split selection

- 2. Prepruning

- 3. Postpruning

Greedy Split Selection

- At each node:
- 1. Enumerate candidate features
- 2. Enumerate candidate threshold
- 3. Compute all impurity decrease
- 4. Select the best (j, τ) among them

Prepruning

- Multiple stopping conditions are checked before searching for the best split, and the node becomes a leaf whenever any of the following criteria are satisfied:
- 1. Node is pure.
- 2. Reaching the maximum depth.
- 3. Node has less than minimum samples to split.

Postpruning

- For each internal node t, let T_t denote the subtree rooted at t and $L(T_t)$ is their leaves, the effective complexity parameter associated with node t is:

$$g(t) = \frac{R(T) - (T_t)}{|L(T_t)| - 1}$$

- 1. For the current tree, we need to calculate all g(t) for every internal node.
- 2. Find the minimum value, denoted as a_k
- 3. Prune all nodes t with $g(t)=a_k$
- 4. Repeat on the smallest tree, only leave the root node.

Model Evaluation

- Accuracy rate:

$$Acc = \begin{cases} \frac{1}{n}\sum_{i=1}^{n} 1[\dot{y}_i = y_i], if\ no\ sample\ weight \\ \frac{\sum_{i=1}^{n} \omega_i 1[\dot{y}_i = y_i]}{\sum_{i=1}^{n} \omega_i}, else \end{cases}$$

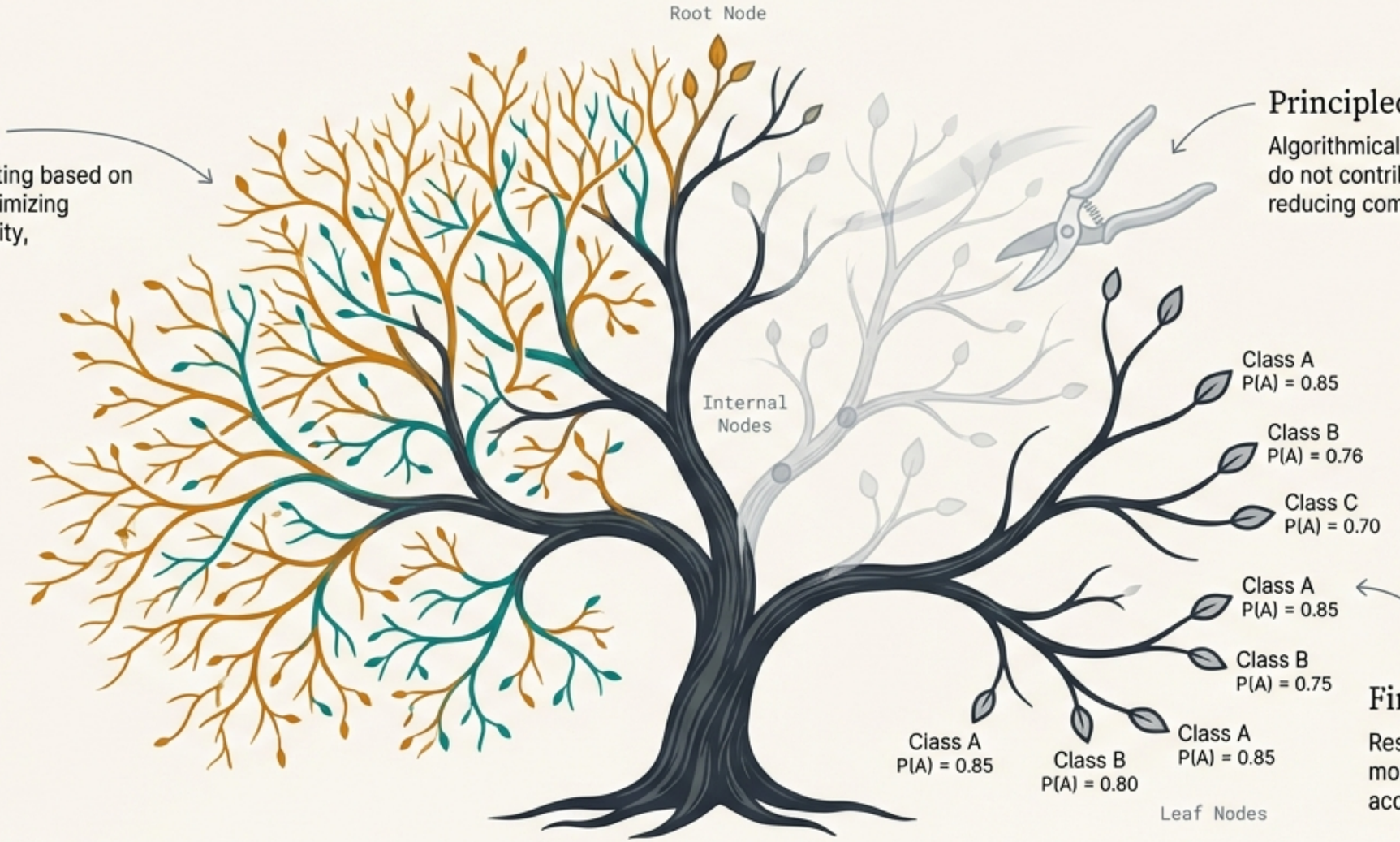
Implementation Methodology

Anatomy of a Decision Tree: From Greedy Growth to Principled Pruning

A step-by-step walkthrough of the core algorithms that build, refine, and operate a decision tree classifier.

Greedy Growth

Recursive binary splitting based on information gain, maximizing immediate homogeneity,



Principled Pruning

Algorithmically removing branches that do not contribute to generalization, reducing complexity and overfitting.

Final, Elegant Structure

Resulting in a simplified, robust model optimized for predictive accuracy on unseen data.

The Blueprint: Defining the Tree's Behavior

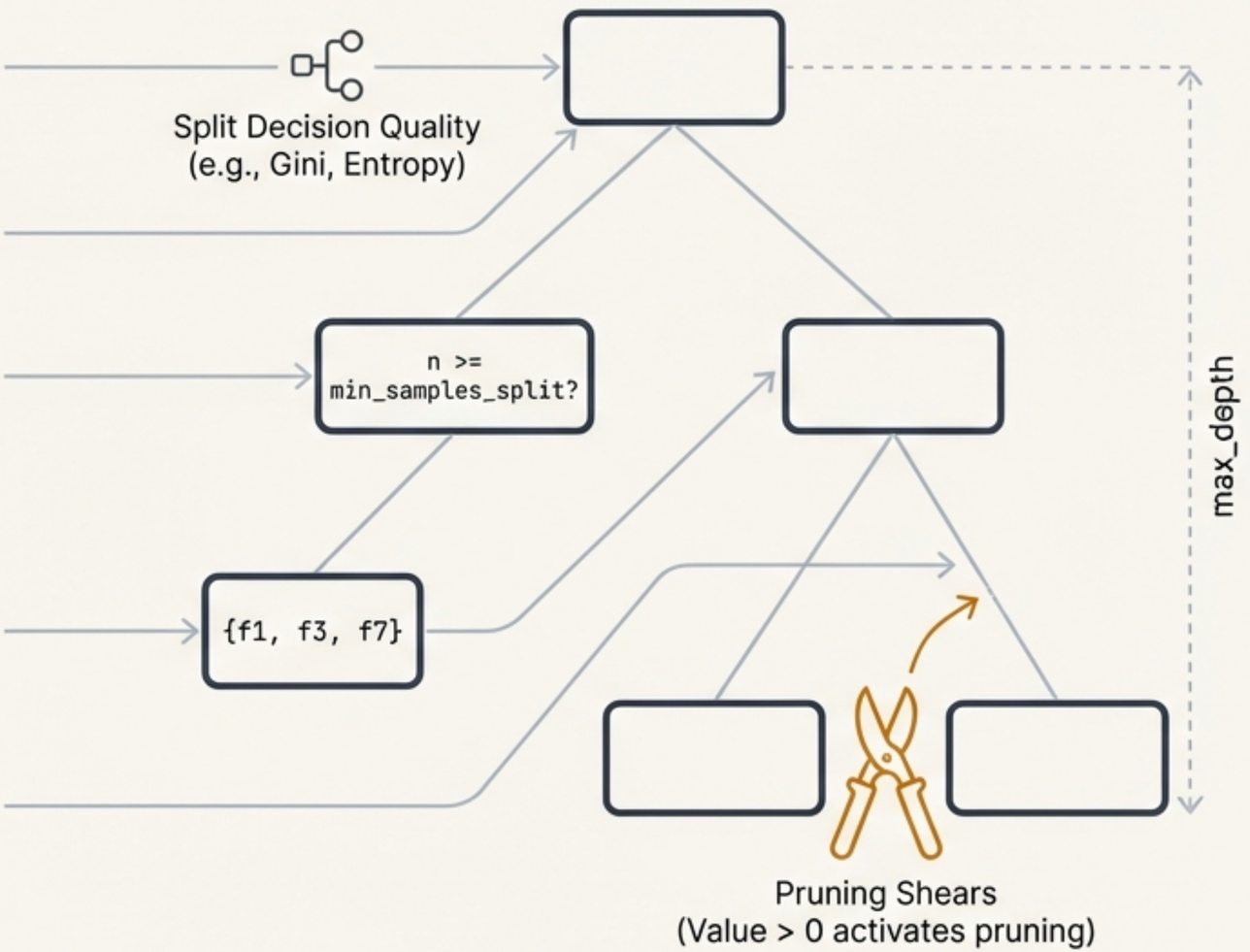
`criterion` – {"gini", "entropy"} – The function to measure the quality of a split.

`max_depth` – The maximum depth of the tree. A primary tool to control complexity.

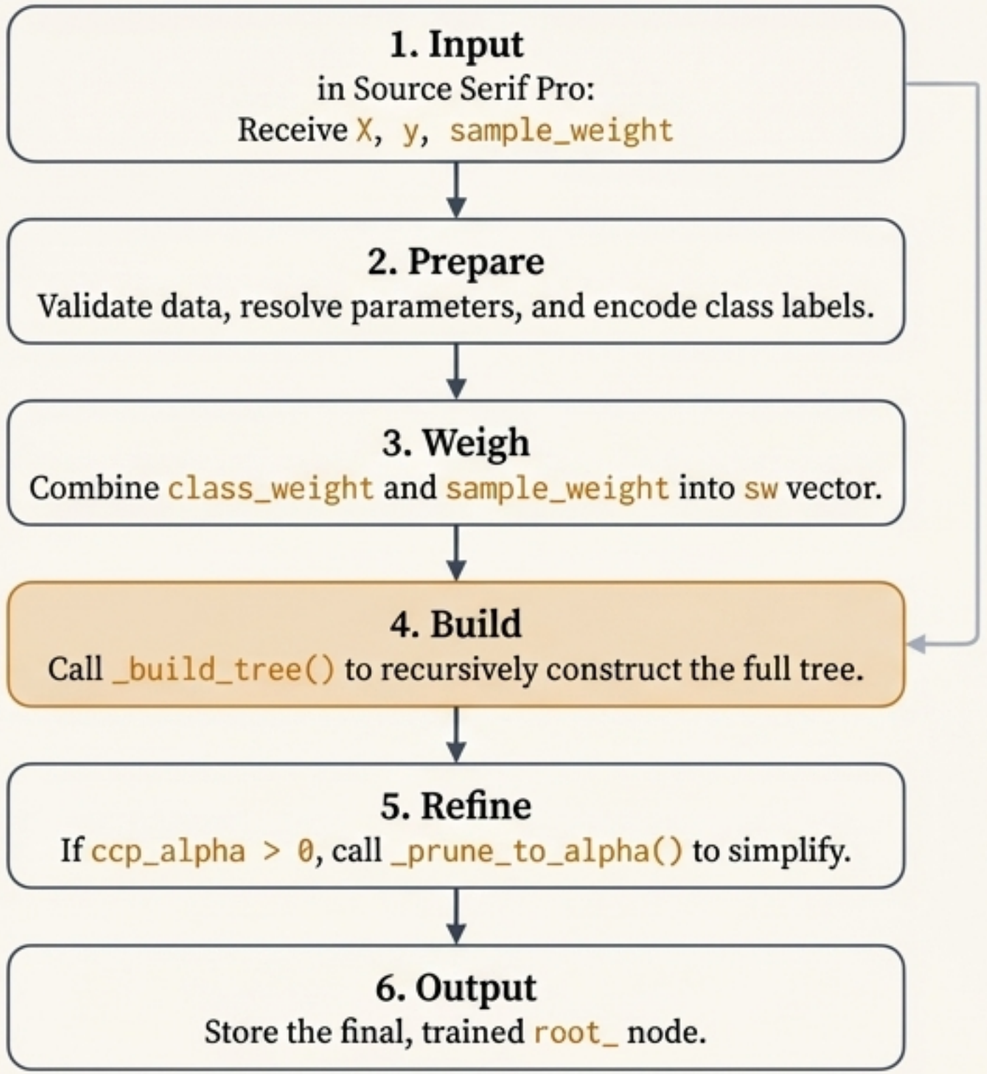
`min_samples_split` – The minimum number of samples required to consider splitting a node.

`max_features` – The number of features to consider when looking for the best split. Introduces randomness and reduces variance.

`ccp_alpha` - The complexity parameter used for Minimal Cost-Complexity Pruning. A value > 0 activates the pruning process.



The Entry Point: Orchestrating the Build with `fit()`




```
function fit(X, y, sample_weight):
    // 1. Prepare data and parameters


    // 2. Calculate final sample weights


    // 3. Build the tree recursively from the root


    // 4. Prune the tree if a complexity parameter is set
```

The Core Engine: `_build_tree()`'s Recursive Logic

`_build_tree` is a recursive function that constructs the tree in a top-down fashion. For each node, it first calculates its properties (like impurity and majority class). Then, it decides if it should become a **leaf node** (a stopping point) or an **internal node** (a decision point).

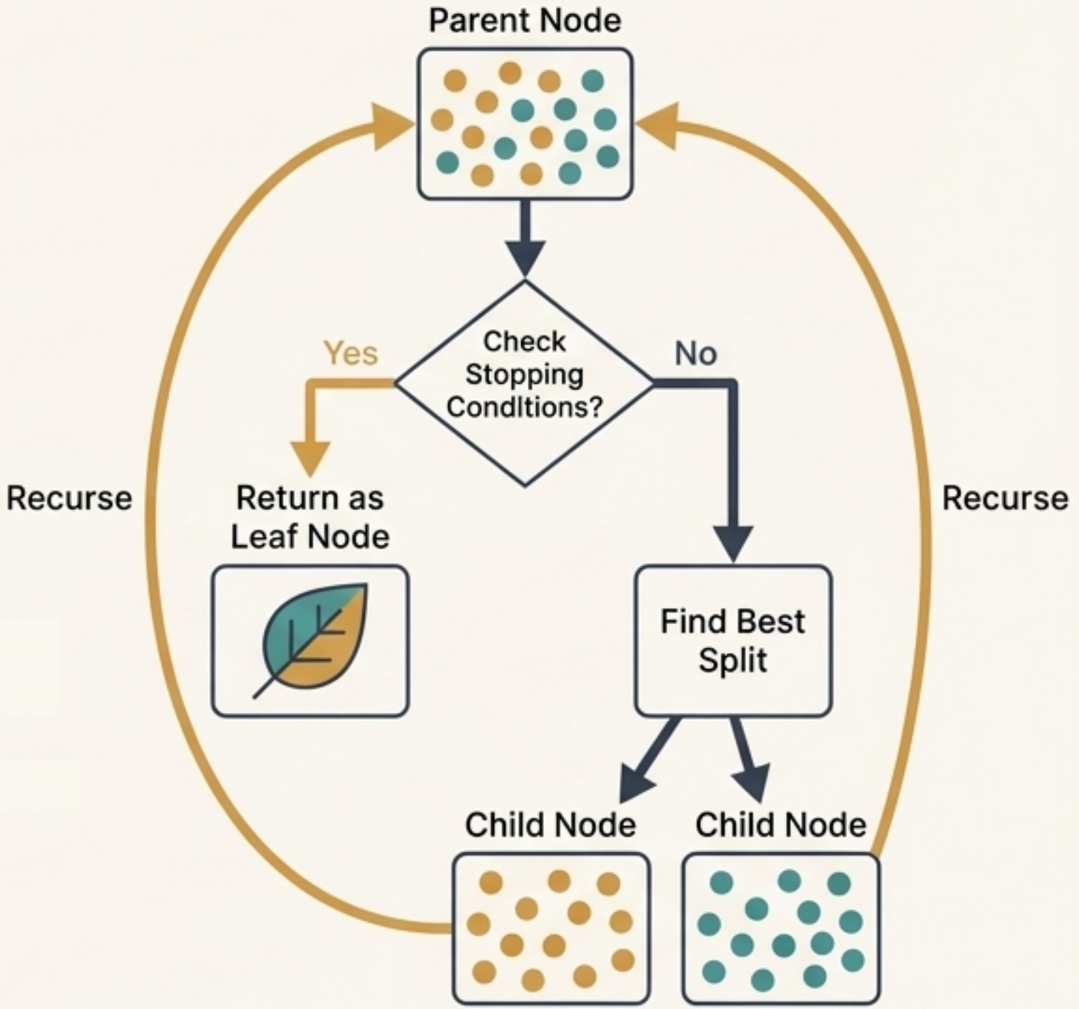
```
function _build_tree(data, depth):
    // 1. Create a node and calculate its initial properties

    // 2. Check stopping conditions (base cases for recursion)


    // 3. Find the best possible split for the current data

    // 4. Check if the split is valid and worthwhile


    // 5. Recurse: build left and right subtrees
```



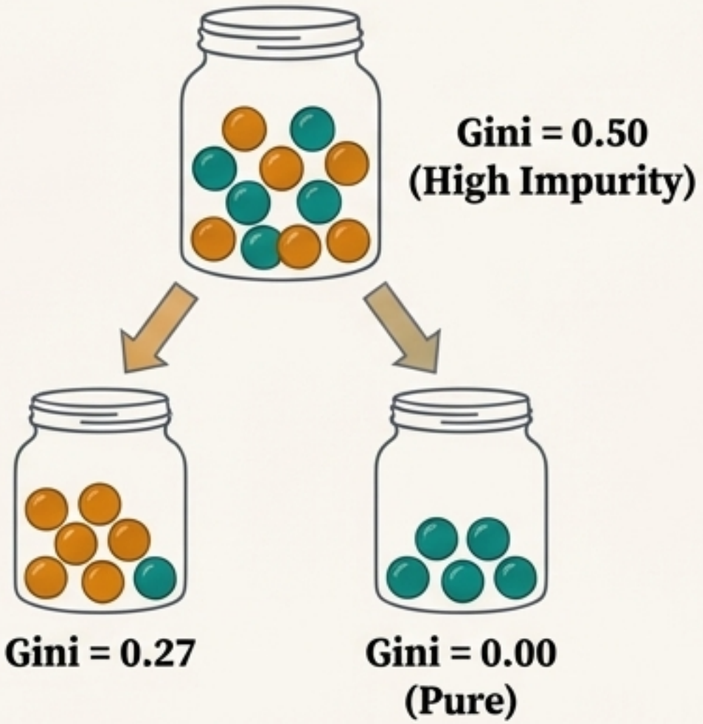
Quantifying Disorder: The Role of Gini Impurity and Entropy

A good split reduces disorder. It takes a ‘mixed’ group of samples and creates two ‘purer’ groups. Gini and Entropy are two ways to measure this disorder.

Gini Impurity

$$Gini = 1 - \sum(p_i)^2$$

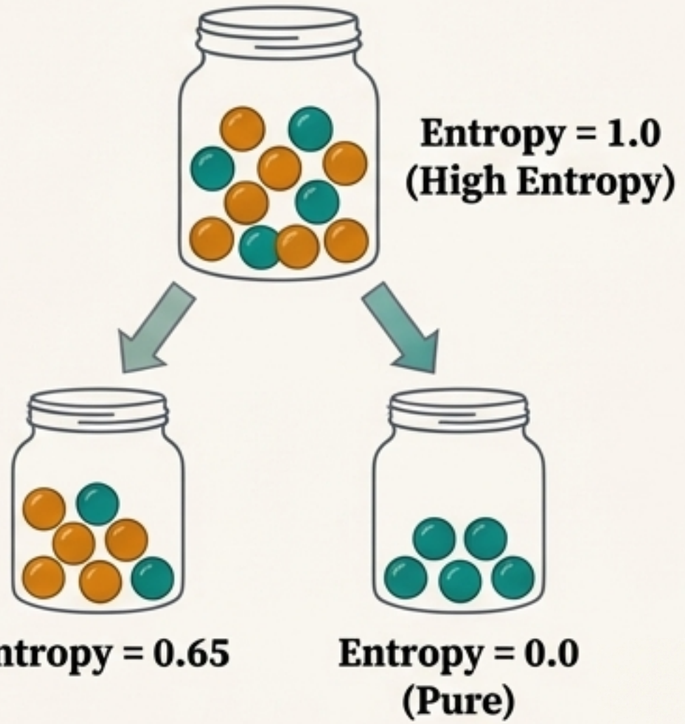
Measures the probability of incorrectly classifying a randomly chosen sample if it were randomly labeled according to the distribution of labels in the node. Lower is better.



Entropy

$$Entropy = -\sum(p_i * \log_2(p_i))$$

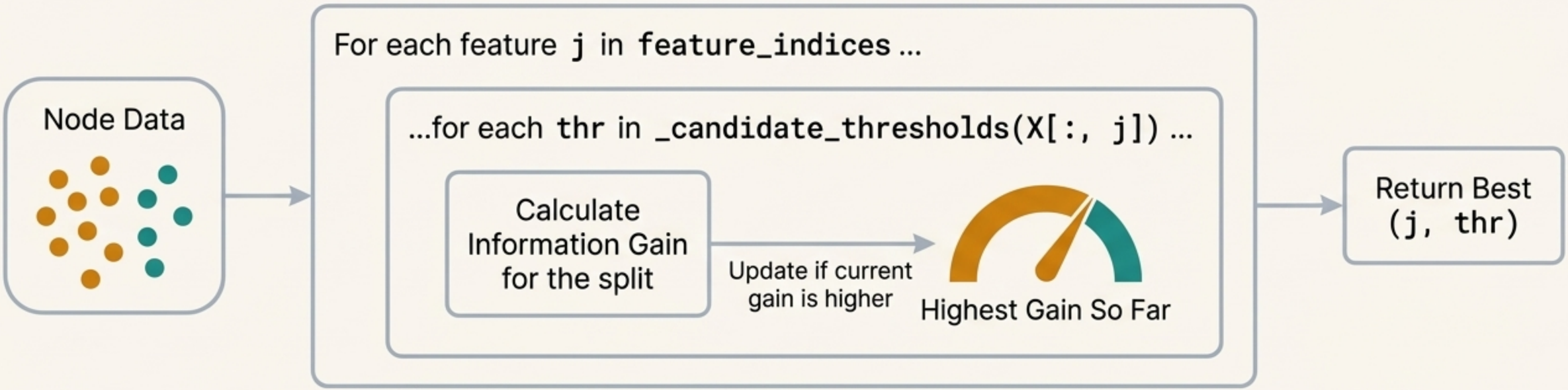
Measures the uncertainty or ‘surprise’ in the data, rooted in information theory. Lower is better.



The Decisive Moment: The `_find_best_split` Algorithm

This function performs a greedy search. It iterates through a subset of features and, for each feature, all potential thresholds to find the single split that yields the greatest **Information Gain**.

```
Information Gain = Impurity(parent) - [Weighted Average Impurity of Children]
```

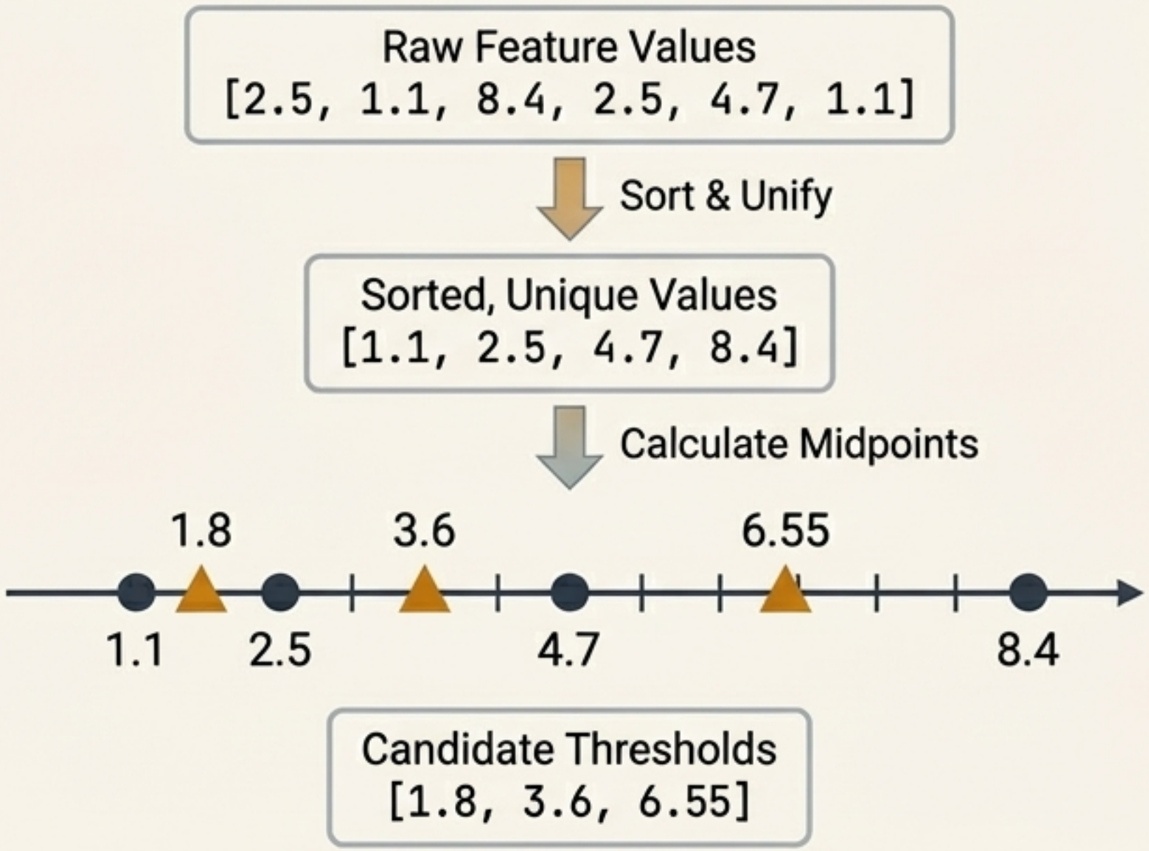


Key Insight: The algorithm is "greedy" because it makes the best choice at the current step without considering if that choice will lead to the best overall tree.

Choosing the Questions: Generating Candidate Thresholds

The Problem: For a feature with continuous values, there are infinite potential thresholds. How do we test them efficiently?

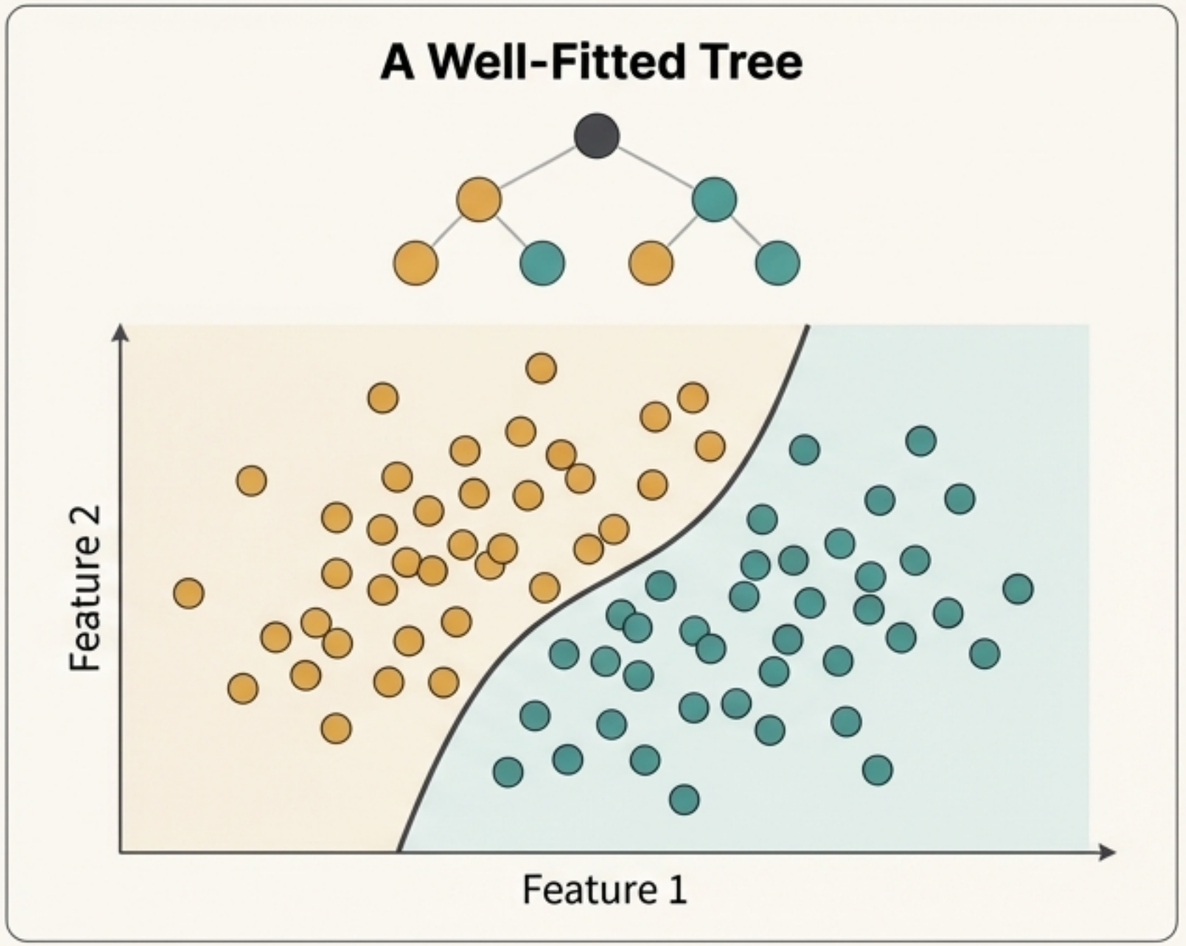
The Solution: The optimal split point must lie between two adjacent unique values of the feature. We only need to test the midpoints.

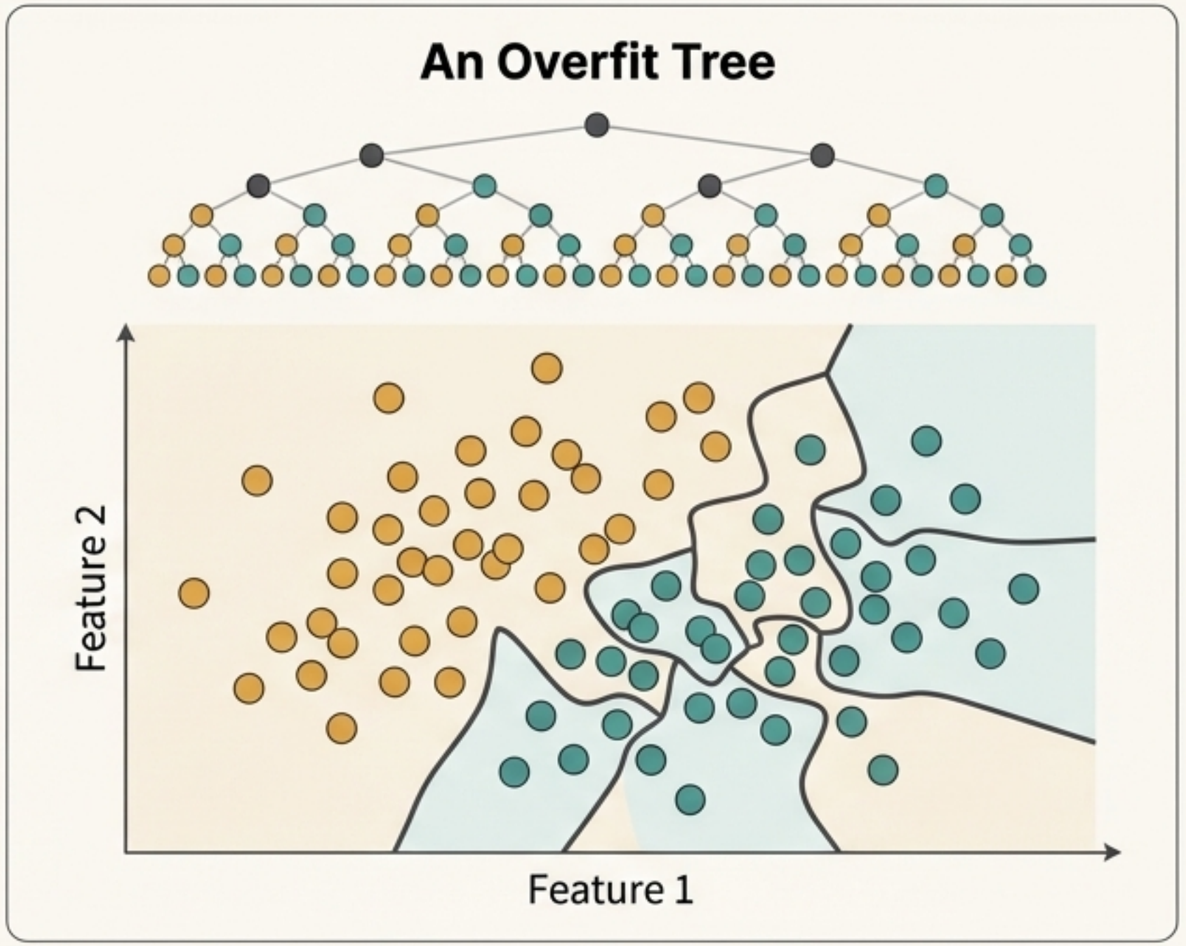



```
function _candidate_thresholds(feature_values):
    unique_vals = sort(unique(feature_values))
    // Return midpoints between adjacent unique values
    return (unique_vals[:-1] + unique_vals[1:]) / 2.0
```

The Peril of Perfection: Overfitting the Training Data

The greedy, recursive nature of `_build_tree` will continue splitting until all leaves are pure. This creates a model that memorizes the training data, including its noise.

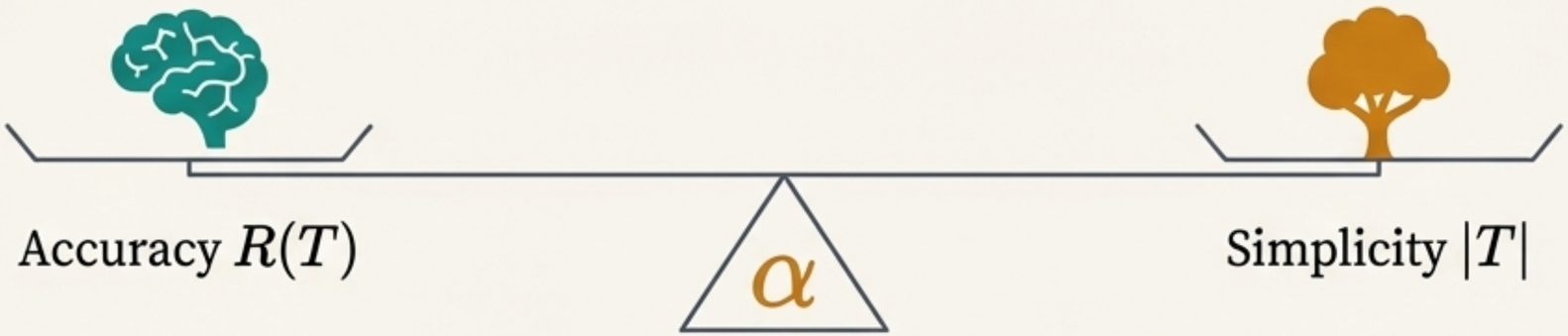




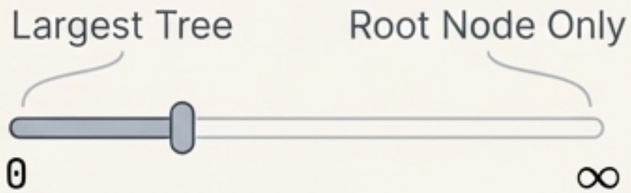
The Resolution: Minimal Cost-Complexity Pruning

Instead of stopping tree growth early with heuristics, we grow a large tree and then prune it back to find the best balance between its performance and its size.

$$R_a(T) = R(T) + \alpha * |T|$$



- $R(T)$: The total weighted impurity of the tree's terminal nodes (leaves). This is the **misclassification cost.**
- $|T|$: The number of leaves in the tree. This is the **complexity penalty.**
- α: The complexity parameter. It controls the trade-off.



Identifying the Weakest Link

The Strategy

Pruning happens iteratively. In each step, we find and prune the internal node that is the “weakest link.”

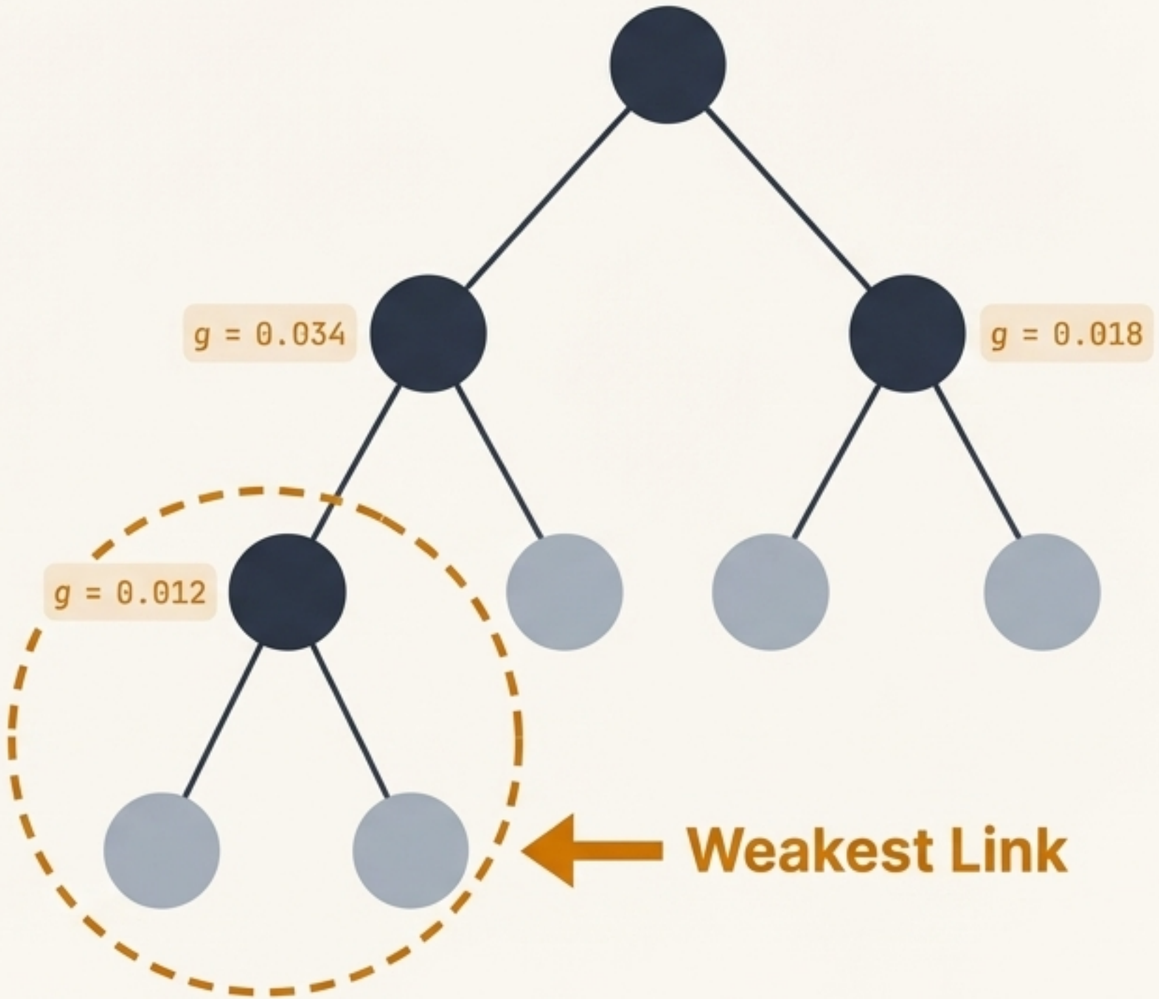
Definition

The weakest link is the node that provides the least amount of impurity reduction relative to the complexity it adds.

The Effective Alpha Formula

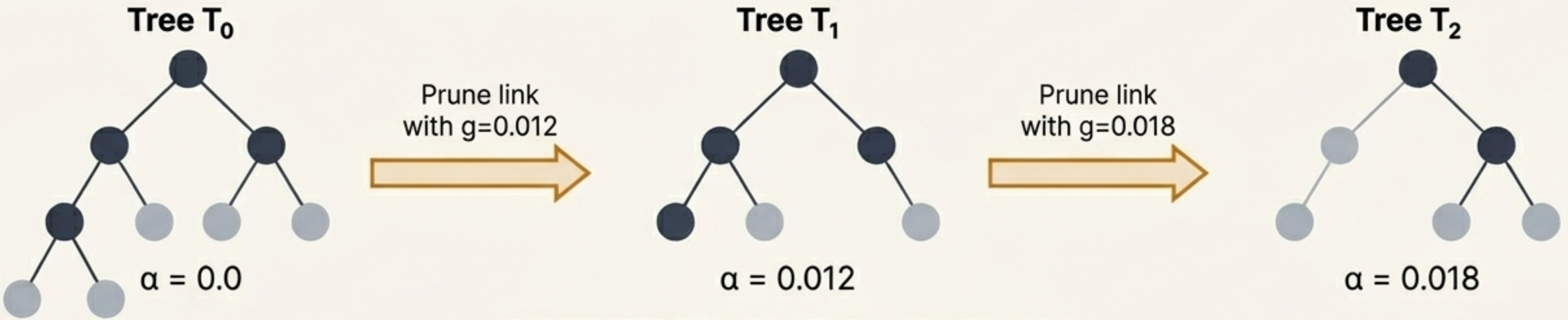
$$g(t) = \frac{R(t) - R(T_t))}{|T_t| - 1}$$

$R(t)$: Impurity of node t if it were a leaf.
$R(T_t)$: Total impurity of the subtree rooted at t.
$|T_t|$: Number of leaves in the subtree at t.

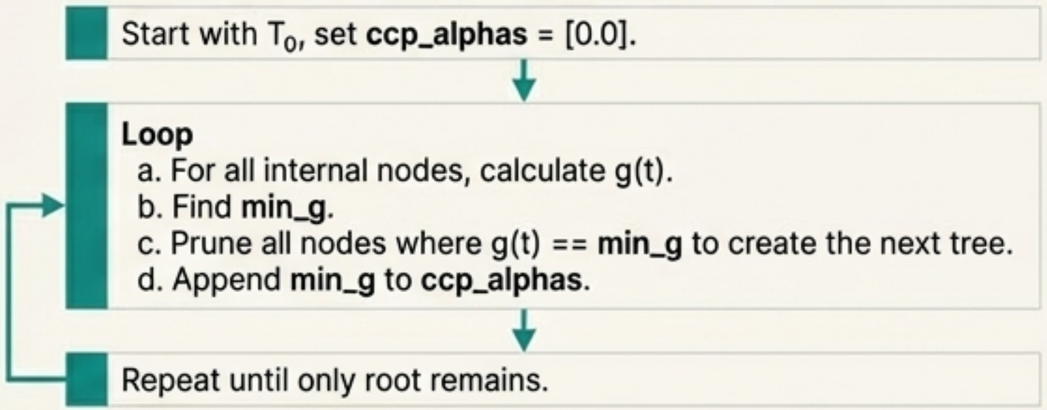


Generating the Full Pruning Path

By repeatedly finding the weakest link and pruning it, we generate a finite sequence of subtrees, each one optimally simple for a specific range of α.



Flowchart of cost_complexity_pruning_path

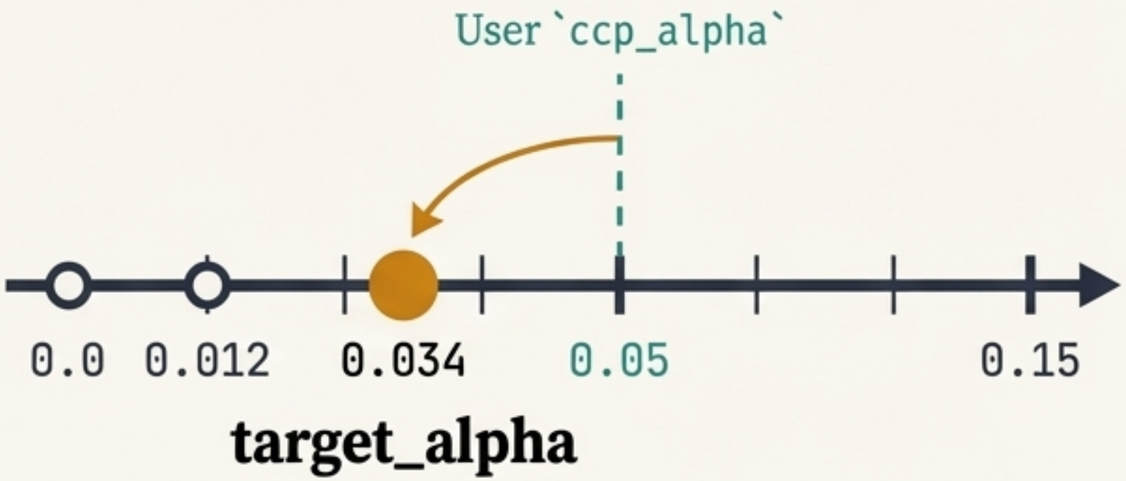


Final Selection: Pruning to the Target Alpha

The Goal: Given a user-specified `ccp_alpha`, select the correct subtree from the pruning path.

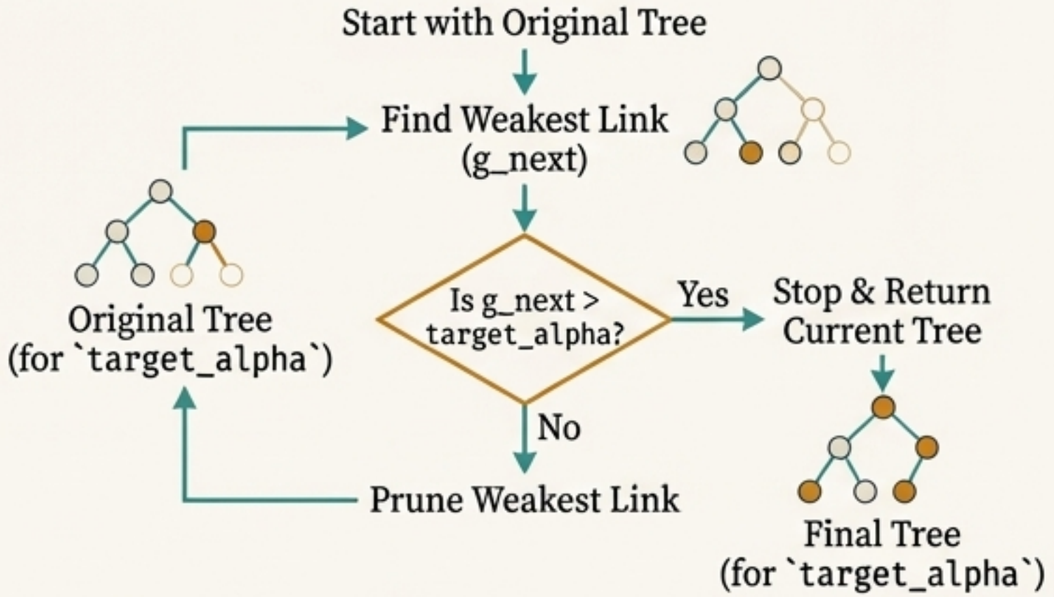
Step 1: Find the Target Alpha

The code first generates the entire `ccp_alphas` sequence on a *clone* of the tree. It then finds the largest value in this sequence that is less than or equal to the user's `ccp_alpha`. This becomes the `target_alpha`.



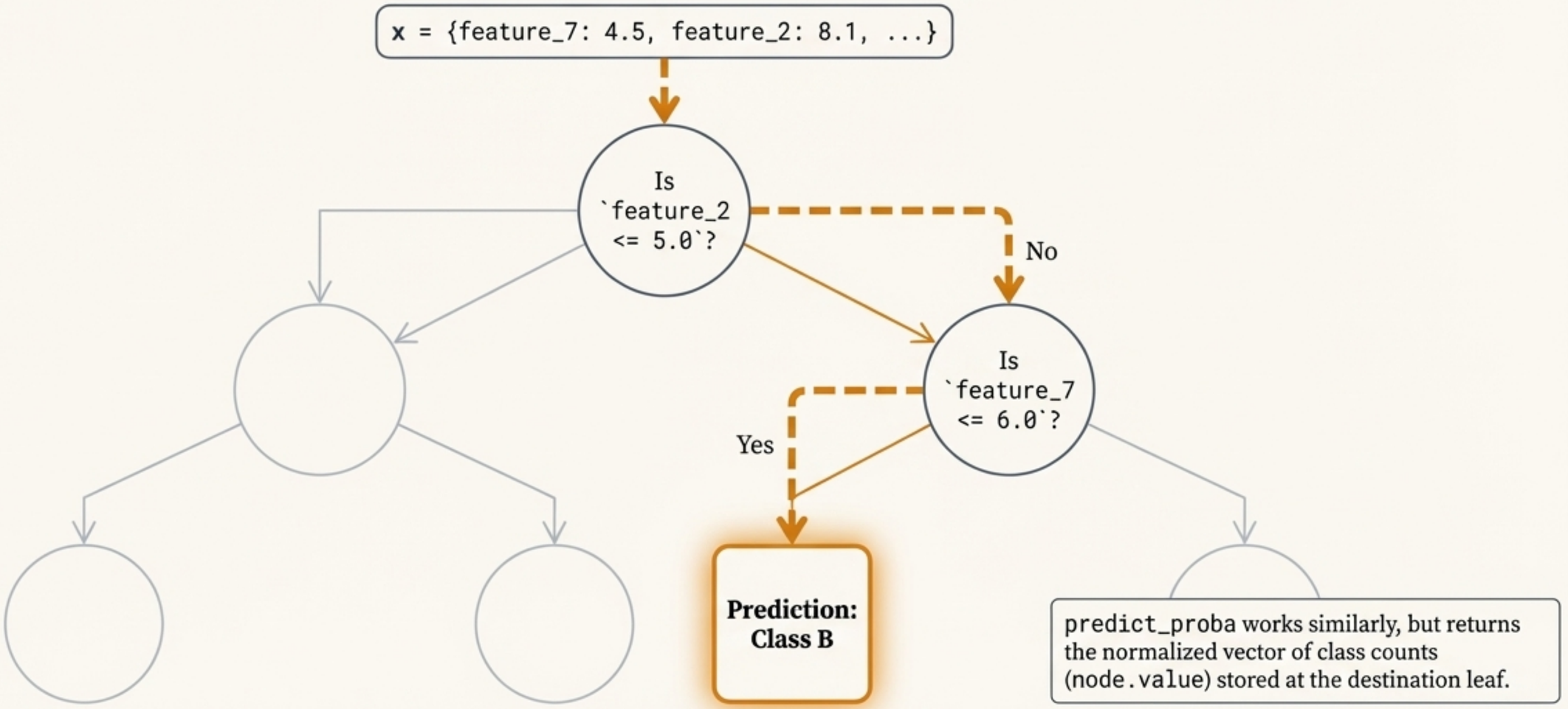
Step 2: Prune the Real Tree

Now, working on the *original* tree, the algorithm iteratively finds and prunes weakest links. It stops pruning as soon as the next weakest link's effective alpha is greater than the `target_alpha`, ensuring the final tree corresponds exactly to the chosen `target_alpha`.



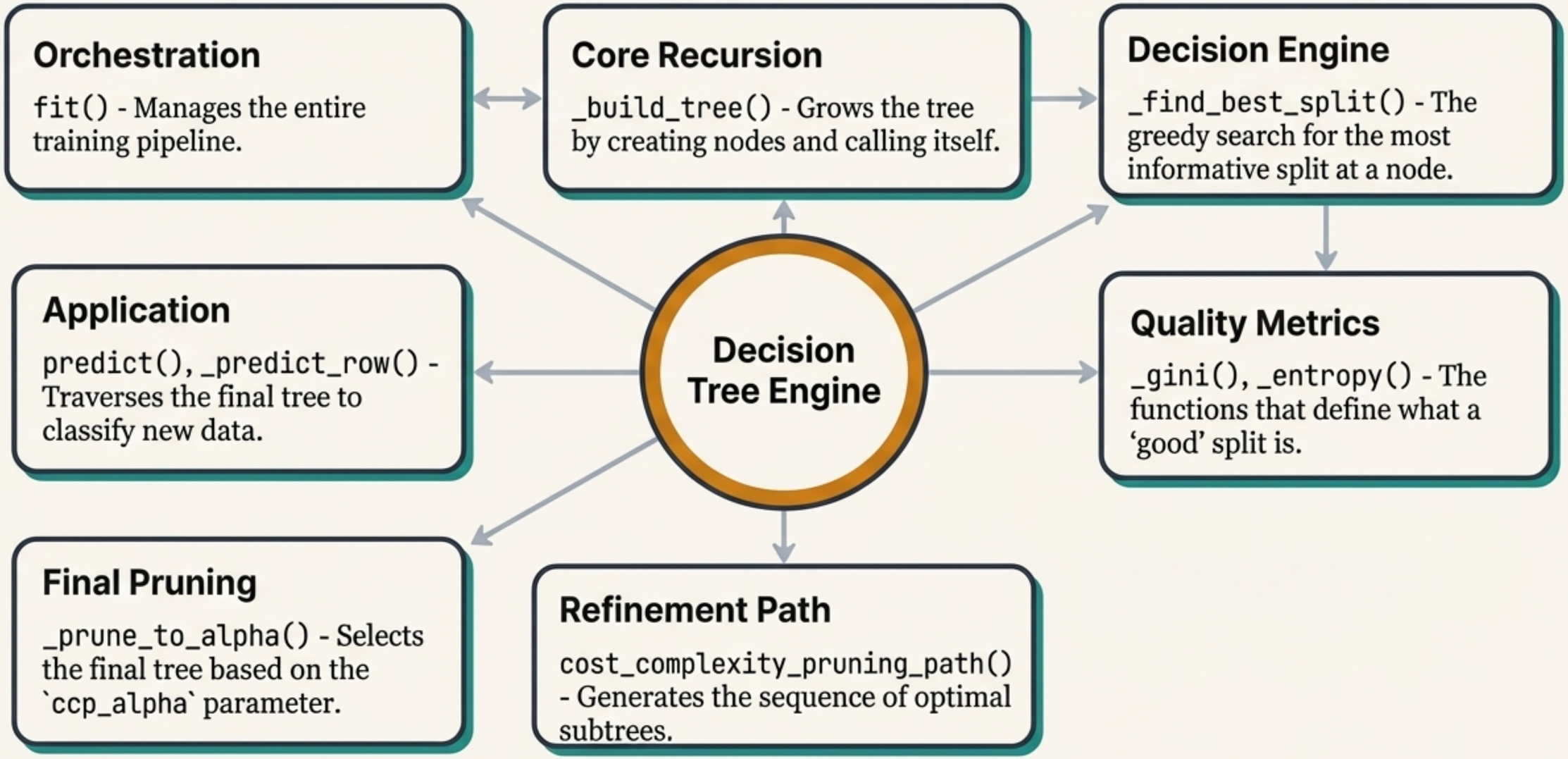
Putting the Tree to Work: The Prediction Path

To predict the class for a new data sample, we drop it down the tree from the root. At each internal node, the sample's feature value is compared to the node's threshold, determining whether to go left or right. This continues until a leaf node is reached. The majority class of that leaf node is the prediction.



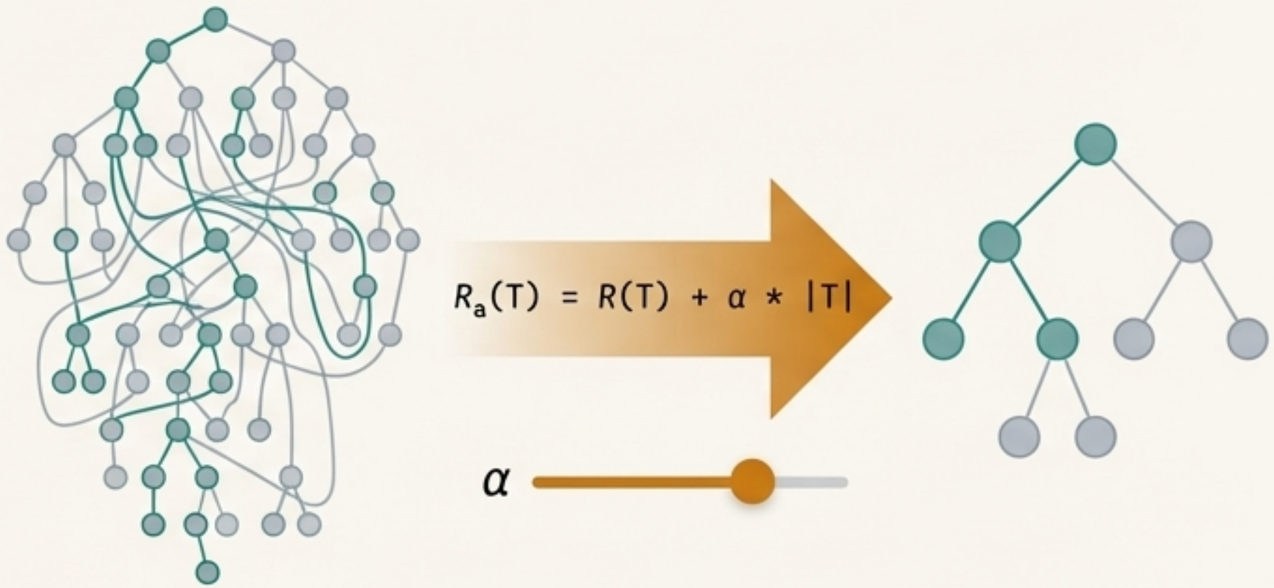
`predict_proba` works similarly, but returns the normalized vector of class counts (`node.value`) stored at the destination leaf.

The Algorithmic Toolkit: A Functional Summary



From Unchecked Growth to a Balanced Model

- **Greedy Construction**
  The tree is initially built using a greedy, top-down recursive algorithm, aiming to create the purest possible leaves.
- **The Overfitting Problem**
  This unchecked growth leads to an overly complex model that captures noise and doesn't generalize well.
- **Principled Refinement**
  Cost-Complexity Pruning provides a deterministic method to simplify the tree, finding the optimal trade-off between misclassification cost and model complexity.



Overfit Tree

Balanced Tree

This two-phase process—greedy growth followed by systematic pruning—is a powerful technique for managing the bias-variance tradeoff and building a high-performing classifier.

Evaluation

Experimental Setup: My CART vs Sklearn

- Dataset: Iris Dataset
- Train / Test split: 70% / 30%
- Same hyperparameters for both models
- Evaluation metrics: Accuracy, Prediction agreement ratio, Confusion matrix

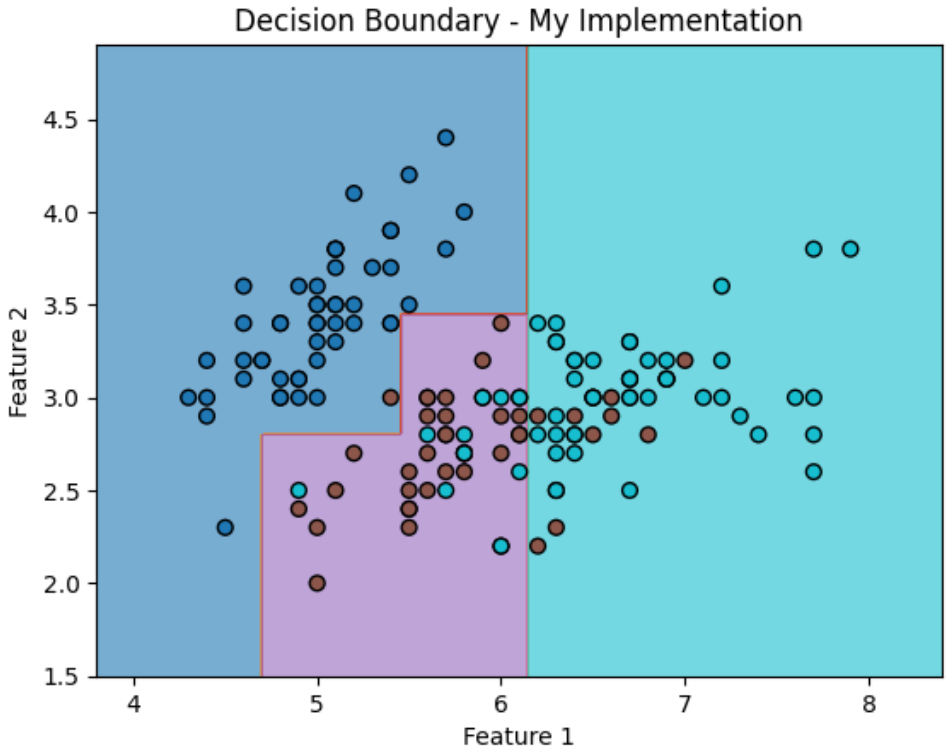
```
Prediction agreement ratio: 0.9555555555555556
My Accuracy      : 0.933333
Sklearn Accuracy : 0.977778
our implementation successfully reproduce the sklearn results.
```

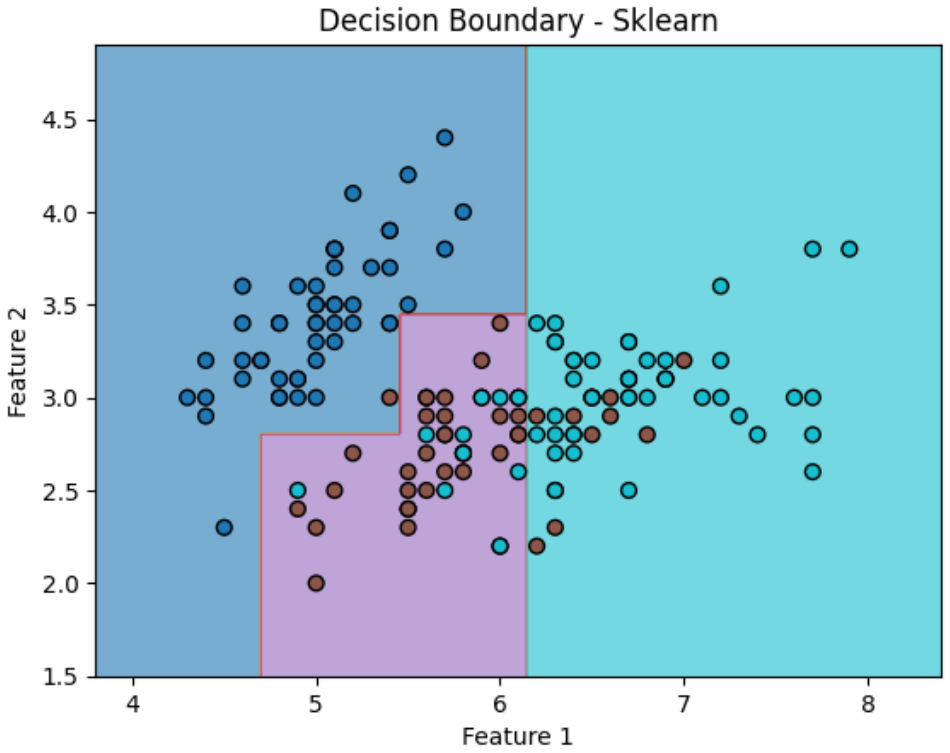
Conclusion:

- Over 95% of test samples receive the same prediction.
- Accuracy difference is small and within acceptable range.

Decision Boundary Visualization

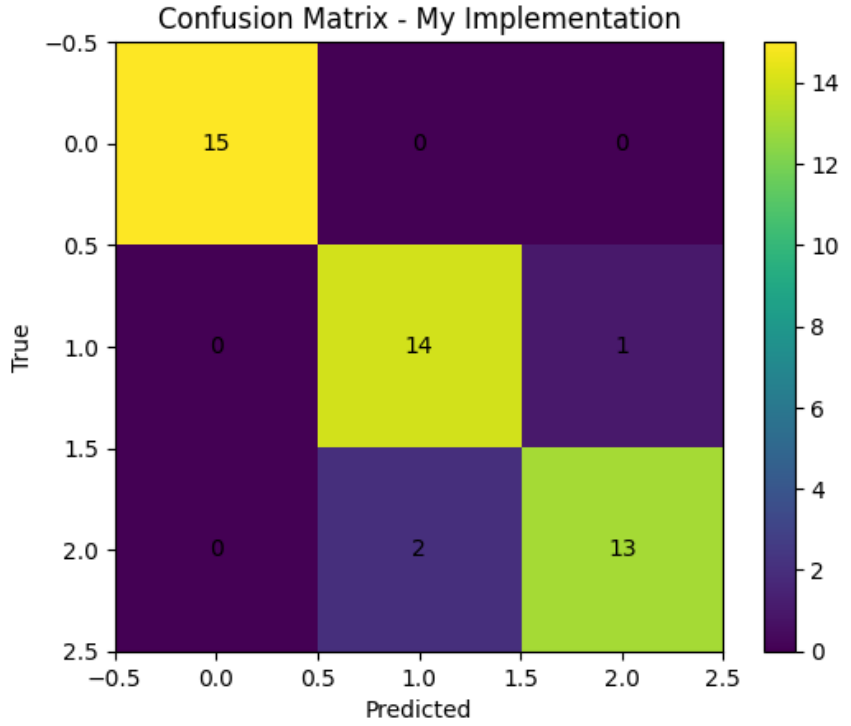
- Trained on first two features of Iris
- My CART vs Sklearn CART
- Very similar partition structure
- Differences only near class boundaries

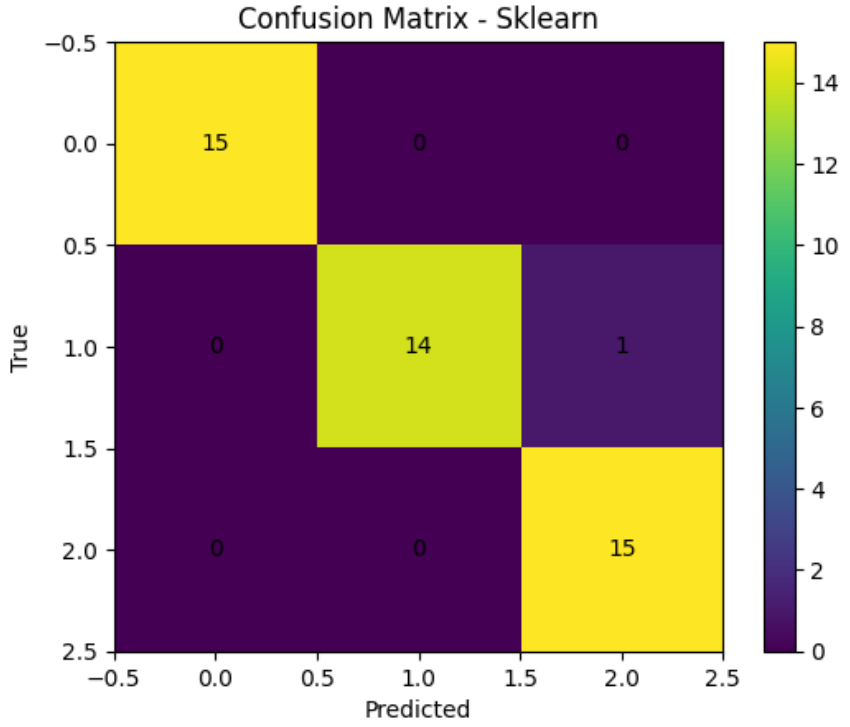




Confusion Matrix Comparison

- Only 2 samples are predicted differently
- Errors mainly occur near the third class



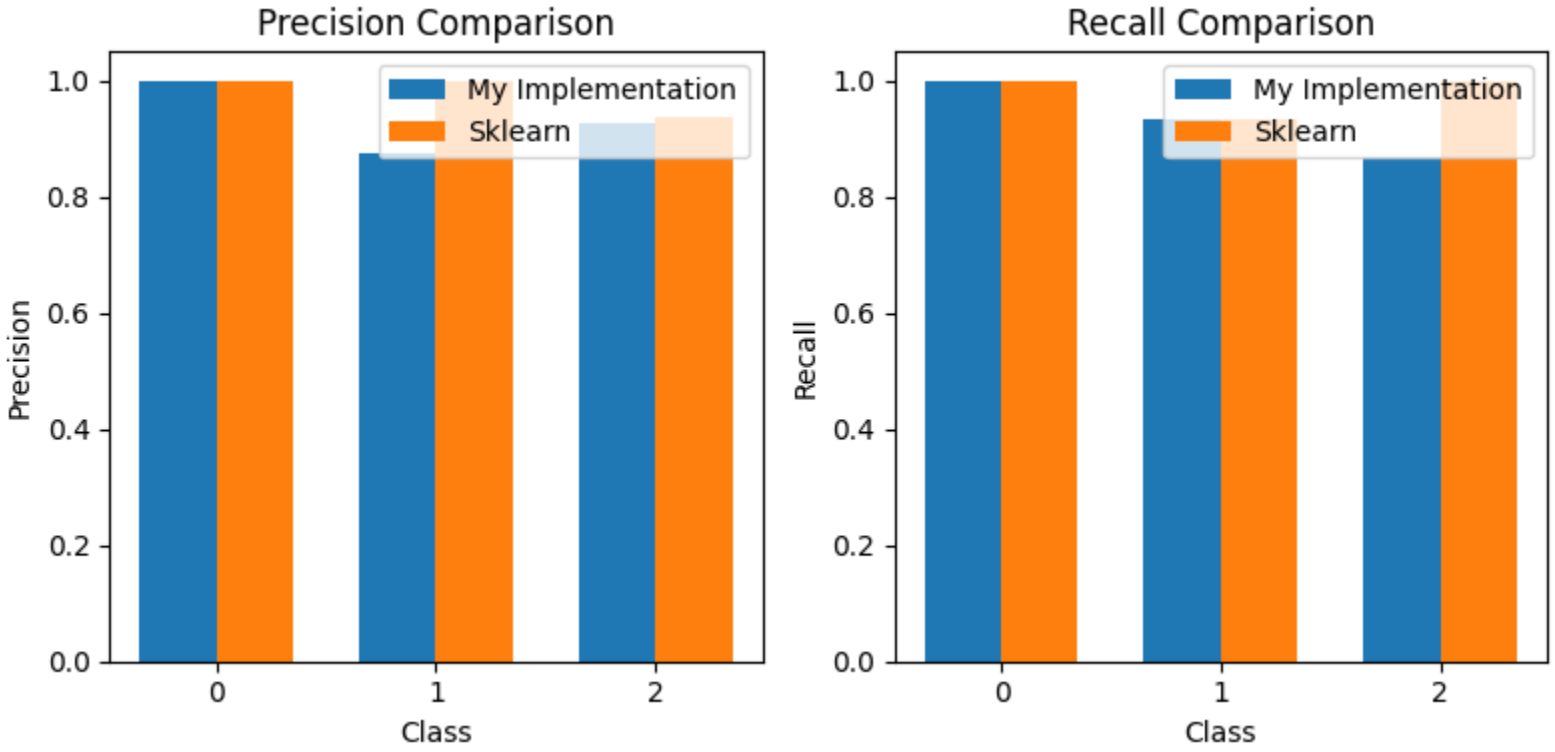


Conclusion:

- Over 95% of test samples receive the same prediction.
- Accuracy difference is small and within acceptable range.

Precision & Recall Comparison

- Precision and recall exhibit similar trends across all classes
- Differences between the two models remain small



Further validation on the Wine dataset

- Motivation: Iris is simple → need harder benchmark
- Dataset: Wine (13 features, 3 classes)
- Same dataset split and hyperparameters as Iris
- Fair comparison, no re-tuning
- Evaluation metrics: Prediction agreement, Classification accuracy

```
Prediction agreement ratio: 0.9814814814814815
My Accuracy       : 0.981481
Sklearn Accuracy : 0.962963
Our implementation successfully reproduce the sklearn results.
```

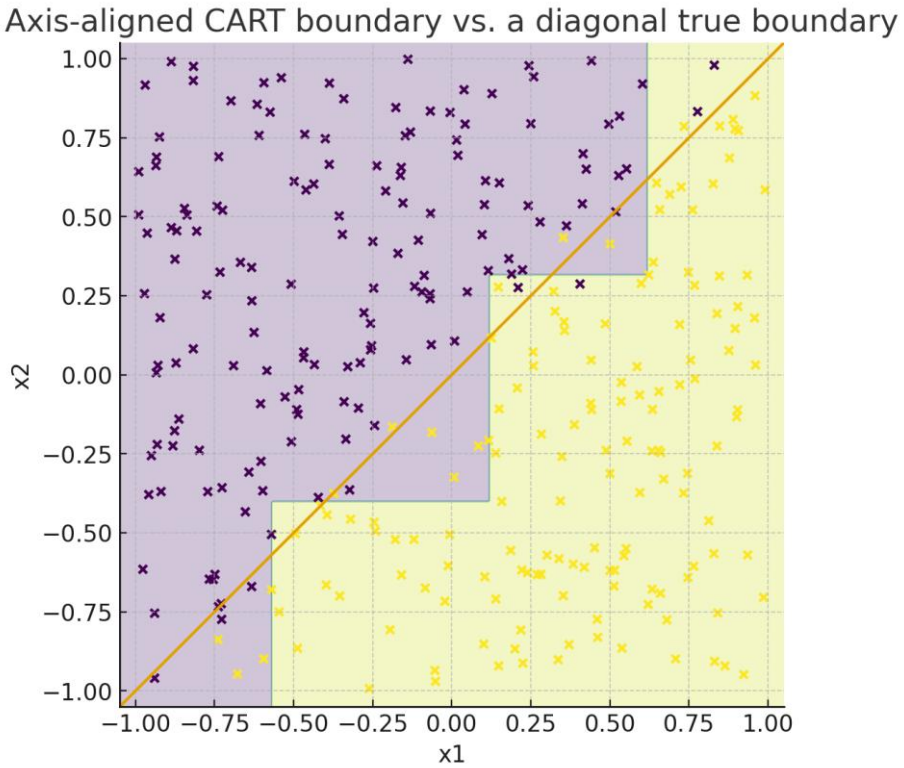
Final Validation Conclusion

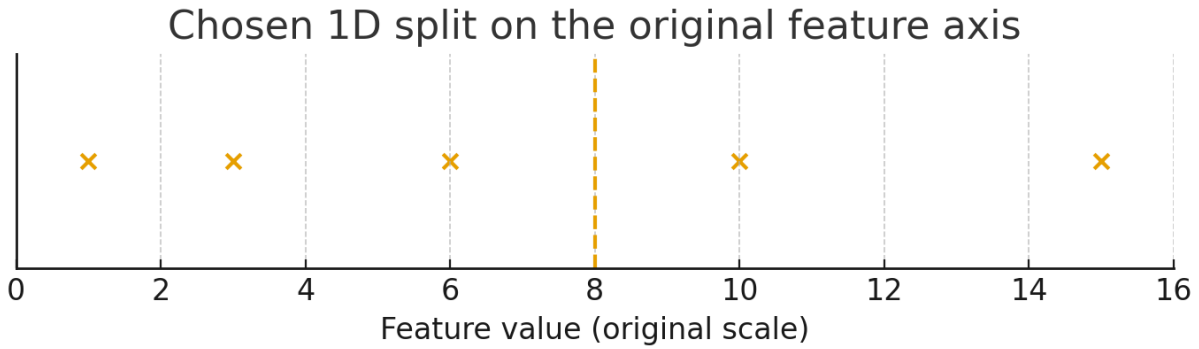
- Our implementation successfully reproduces Sklearn's behavior
- High agreement (>95%)
- Similar decision boundaries
- Nearly identical confusion matrices

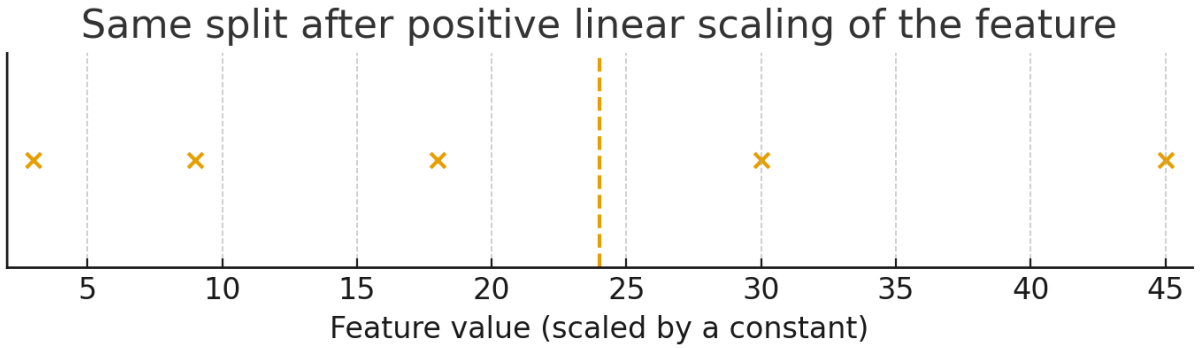
Interesting features and challenges

Interesting feature

- Extremely greedy, not necessarily globally optimal
- Natural axis alignment splitting (the splitting format is typically $x_j \leq t \; vs \; x_j > t$)
- Insensitive to feature scaling







Challenge

- Aligning with sklearn’s expected interface and behaviour
- Managing complex stopping and pruning logic
- It is hard to make our implementation as fast as sklearn.

Thank you for listening
If you have any questions, please feel free to reach out