Scientific Machine Learning 12
Decision Trees

Donghyun Ko

January 4, 2026

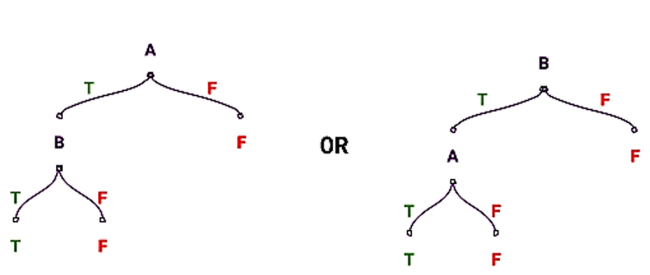
What you will learn. In this posting, we walk through the intuition and mathematics of decision trees—how they split data using simple yes/no questions, how they differ for classification and regression, what "impurity" means, and how metrics such as Gini, Entropy, and Information Gain decide each split. We conclude with pruning, overfitting control, and ensemble variations such as Random Forest and AdaBoost.

Contents

1 Introduction

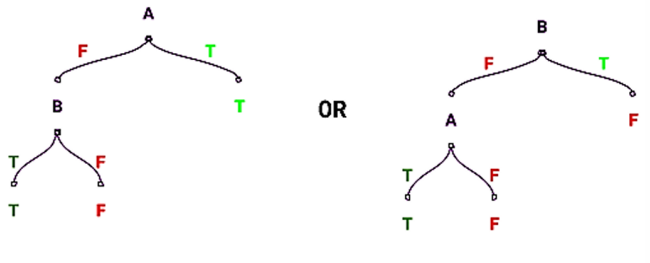
When you hear the term *decision tree*, the image that may first come to your mind might not be a mathematical model, but rather a branching path of reasoning: a series of "if–then" choices leading to a final decision. Before diving into real-world applications, let us warm up with something every computer science students meet early, which is the logical gates. Just as a logical gate outputs True or False depending on its inputs, a tree-like model can represent this sequence of decisions. For instance, an AND gate requires both inputs A and B to be True to produce a True output. Each branch of the tree corresponds to a simple question: "Is A True?" If yes, "Is B True?" and only then does the model say "True." Similarly, the OR and XOR gates can also be expressed as branching trees, demonstrating how hierarchical rules can represent complex logic with simple binary questions.

A	B	A AND B
F	F	F
F	T	F
T	F	F
T	T	T



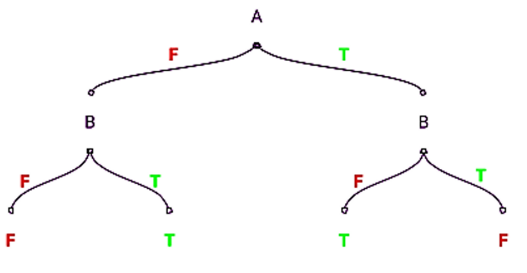
The AND logical gate represented as a tree structure.

A	B	A OR B
F	F	F
F	T	T
T	F	T
T	T	T



The OR logical gate in tree form, combining alternative paths.

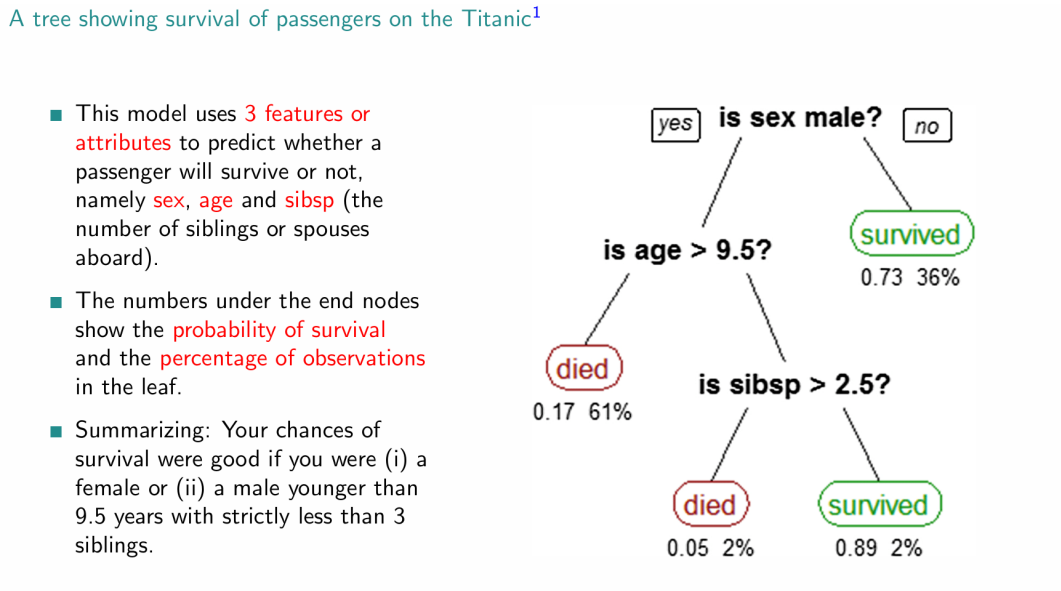
A	B	A XOR B
F	F	F
F	T	T
T	F	T
T	T	F



The XOR logical gate modeled as a two-layer decision tree.

These toy examples show how decisions can be represented recursively with each question narrowing possibilities until a single outcome remains. Another good example to introduce

the concept of decision tree is the *Titanic survival chance*. Now imagine applying these gate logic to real data: instead of Boolean inputs A and B, we use features such as age, gender, or number of siblings to decide whether a passenger survived the Titanic disaster.



A decision tree predicting survival on the Titanic.

In this famous example, the model learns a sequence of human-like rules: *If the passenger is female, predict survived. Else, if male and younger than 9.5 years with fewer than 3 siblings, also predict survived; otherwise, died.* Each path corresponds to a simple question, but together they capture meaningful patterns hidden in data.

This step-by-step reasoning process is exactly what a **Decision Tree (DT)** models. A decision tree is a supervised learning method that predicts a target value by applying a series of branching rules. At every split, the model asks a question about one feature—*if the condition holds, go left; otherwise, go right.* The journey ends at a leaf node, which holds the prediction: a class label for classification or a numeric estimate for regression. Unlike linear regression or logistic models, a decision tree does not assume a pre-defined equation between input and output. It simply discovers the rules directly from data, which is why it's called a *non-parametric* model. The deeper the tree, the more specific and complex these rules become. In practice, two main variants exist. A **classification tree** deals with categorical outcomes—such as spam vs. not spam, survive vs. die, pass vs. fail—whereas a **regression tree** predicts continuous quantities such as housing price or drug effectiveness. Together, they are collectively referred to as **CART** (*Classification and Regression Trees*), one of the most powerful and interpretable families of models in machine learning.

2 Fundamentals of Decision Trees

A **Decision Tree (DT)** is a *non-parametric supervised learning* method used for both classification and regression tasks. Non-parametric means that it does not assume a specific mathematical form between input and output. Instead, the model learns a series of simple, human-like *'if–then–else'* rules directly from data. In essence, a decision tree automatically searches for ways to **split a dataset based on conditions** that make the resulting groups as homogeneous as possible. At each step, the algorithm asks: *"Which feature and threshold divide the data best?"* This sequence of binary questions gradually builds a tree-shaped structure that can be interpreted as a set of decision rules.

Two main variants of decision trees:

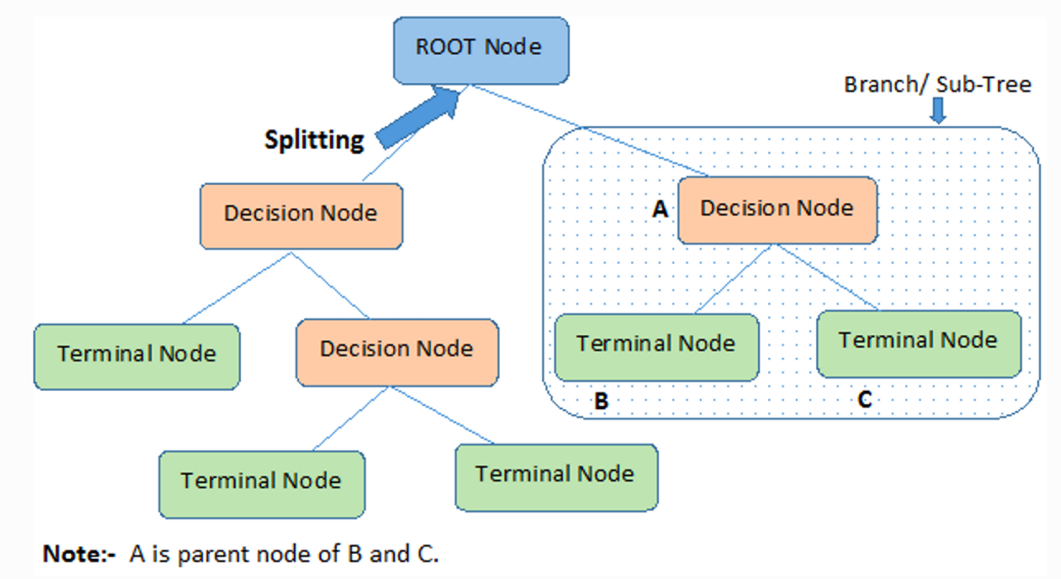
- **Classification Tree:** targets are *categorical* (e.g. survival/died, spam/not spam, pass/fail).
- **Regression Tree:** targets are *continuous* (e.g. house price, drug effectiveness).

Both rely on the same principle of hierarchical splitting and are therefore collectively known as **CART: Classification and Regression Trees.**

Important concepts related to DTs To understand a tree's anatomy, let's briefly summarize its key components.

- **Root Node:** represents the entire training dataset, from which the tree grows.
- **Splitting:** the process of dividing a node into two or more sub-nodes according to some condition.
- **Decision (Internal) Node:** a node that can still be split further—it has arrows both entering and leaving.
- **Leaf (Terminal) Node:** a node that cannot be split any further; it gives the final prediction.
- **Branch (Sub-tree):** a subsection of the full tree that behaves like a smaller tree itself.
- **Parent and Child Nodes:** when a node divides into sub-nodes, it becomes their parent; the sub-nodes are its children.
- **Pruning:** the process of cutting off weak or unnecessary branches to prevent overfitting and simplify the tree.

Each node in the tree acts as a test case for some feature/attribute, and each edge descending from the node corresponds to the possible answers to the decision rule. This process is recursive in nature and is repeated for every sub-tree rooted at the new node.



Important concepts of a Decision Tree.

Each new observation is classified or predicted by **traversing the tree from the root to a leaf**. At each step, a condition is tested—such as "Is salary > \$50,000 ?"—and the data follow the corresponding branch. When a leaf is reached, the model outputs its decision: either a label (classification) or a numerical estimate (regression). Because each node tests a feature and each edge represents a possible answer, the process is inherently **recursive**, repeating itself for every sub-tree rooted at a new node.

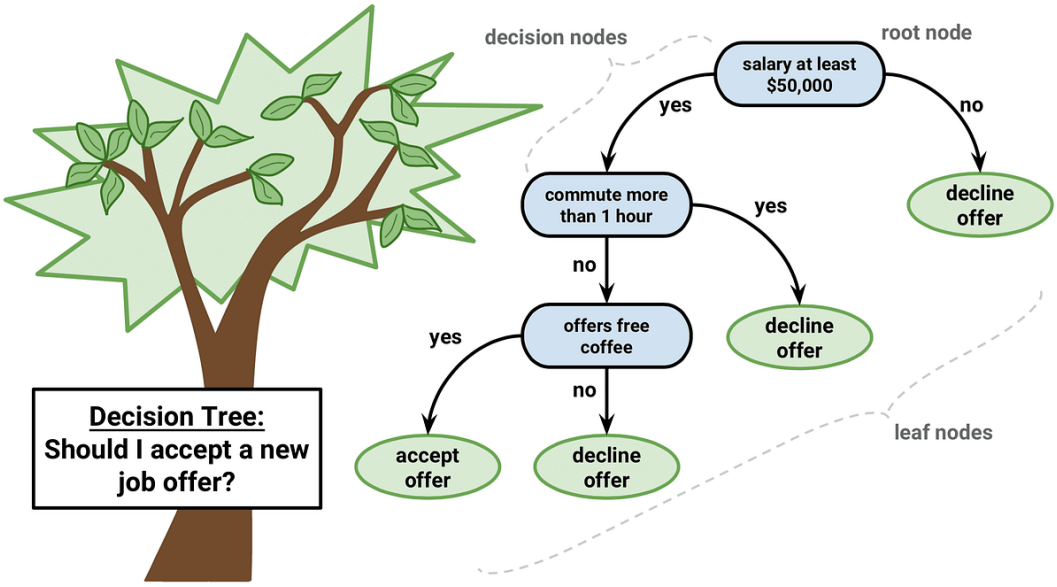


Illustration of a Decision Tree.

3 Process of a Classification Tree

We are going to illustrate the full training loop of a *classification tree* using a tiny dataset with two binary features:

$$\textbf{Feature A}, \quad \textbf{Feature B} \in \{\texttt{Yes}, \texttt{No}\},$$

and the target **Heart Disease** $\in \{\texttt{Yes}, \texttt{No}\}$. Let the dataset contain $n = 100$ patients. Throughout, node impurity is measured by the *Gini index*

$$\text{Gini}(S) \;=\; 1 - \sum_{k \in \{\text{Yes,No}\}} p_k^2,$$

and when a node S is split into children $S_1, \ldots, S_m$, the *weighted impurity* is

$$\text{WGini}(S \to \{S_j\}) \;=\; \sum_{j=1}^{m} \frac{|S_j|}{|S|} \,\text{Gini}(S_j)$$

Step 1. Pick the root feature by comparing weighted impurity

Assume the class counts after splitting the whole dataset by each feature are:

	Yes	No	total
A=Yes	30	10	40
A=No	20	40	60

	Yes	No	total
B=Yes	35	15	50
B=No	15	35	50

If we split by A:

$$\begin{aligned}
\text{Gini}(A{=}\texttt{Yes}) &= 1 - \left(\tfrac{30}{40}\right)^2 - \left(\tfrac{10}{40}\right)^2 = 1 - (0.75^2 + 0.25^2) = 0.375, \\
\text{Gini}(A{=}\texttt{No}) &= 1 - \left(\tfrac{20}{60}\right)^2 - \left(\tfrac{40}{60}\right)^2 = 1 - (\tfrac{1}{3}^2 + \tfrac{2}{3}^2) \approx 0.4444, \\
\text{WGini}(\text{split on } A) &= \tfrac{40}{100} \cdot 0.375 + \tfrac{60}{100} \cdot 0.4444 \approx \boxed{0.4167}
\end{aligned}$$

If we split by B:

$$\begin{aligned}
\text{Gini}(B{=}\texttt{Yes}) &= 1 - (0.7^2 + 0.3^2) = 0.42, \\
\text{Gini}(B{=}\texttt{No}) &= 1 - (0.3^2 + 0.7^2) = 0.42, \\
\text{WGini}(\text{split on } B) &= 0.5 \cdot 0.42 + 0.5 \cdot 0.42 = \boxed{0.42}
\end{aligned}$$

Because $0.4167 < 0.42$, **Feature A** becomes the root node.

Step 2. Recurse on each child node

We now consider the two nodes created by A.

Node $A =$ `Yes` (40 samples). Suppose the distribution within this node by B is:

$$B = \texttt{Yes}: \ (24 \text{ Yes}, 6 \text{ No}) \ (30 \text{ total}), \qquad B = \texttt{No}: \ (6 \text{ Yes}, 4 \text{ No}) \ (10 \text{ total})$$

Gini per child:

$$\text{Gini}(B{=}\texttt{Yes}) = 1 - (0.8^2 + 0.2^2) = 0.32, \quad \text{Gini}(B{=}\texttt{No}) = 1 - (0.6^2 + 0.4^2) = 0.48$$

Weighted impurity inside this node:

$$\text{WGini} = \tfrac{30}{40} \cdot 0.32 + \tfrac{10}{40} \cdot 0.48 = \boxed{0.36}$$

Splitting A=`Yes` on B yields a weighted Gini of 0.36, which is below the unsplit impurity 0.375, so the reduction is

$$0.375 - 0.36 = 0.015$$

Viewed purely by impurity, this is an improving split and we would proceed. In practice, we split only if this reduction exceeds the user–defined `'min_impurity_decrease'` (one of stopping rules to prevent overfitting); otherwise we stop and keep A=`Yes` as a leaf.

Node $A =$ `No` (60 samples). Assume the distribution by B is:

$$B = \texttt{Yes}: \ (11, 9) \ (20 \text{ total}), \qquad B = \texttt{No}: \ (9, 31) \ (40 \text{ total})$$

Ginis and weighted impurity:

$$\begin{aligned} \text{Gini}(11/9) &= 1 - (0.55^2 + 0.45^2) = 0.495, \\ \text{Gini}(9/31) &= 1 - (0.225^2 + 0.775^2) = 0.34875, \\ \text{WGini} &= \tfrac{20}{60} \cdot 0.495 + \tfrac{40}{60} \cdot 0.34875 = \boxed{0.3975} \end{aligned}$$

The unsplit impurity is 0.4444. A split on B reduces it by 0.0469, i.e.,

$$0.4444 - 0.0469 = 0.3975$$

By impurity alone, this is a beneficial split. In practice, we proceed only if this reduction exceeds the user–set `min_impurity_decrease` (a stopping rule to prevent overfitting); otherwise we keep the node as a leaf.

Step 3. Decide when to stop (stopping rules)

A node becomes a **leaf** when any of the following holds (typical settings):

- **Purity threshold:** $\text{Gini}(S)$ is below a user-defined limit (e.g., near 0).
- **Minimum samples per leaf:** $|S| < n_{\min}^{\text{leaf}}$ (e.g. $n_{\min}^{\text{leaf}} = 10$).
- **Maximum depth:** the path length from root has reached a limit (e.g., depth = 5).
- **Minimum impurity decrease:** Best candidate split reduces impurity by less than $\Delta_{\min}$.

When no rule triggers, the algorithm repeats Step 2 on each child (recursive partitioning).

Step 4. Make predictions at leaves

Each leaf stores either:

- the **majority class** for hard classification; or
- the **class proportion** $p(\texttt{Yes})$ for probabilistic output.

A new patient is classified by evaluating the tests from the root down to a leaf and returning the leaf's label or probability.

Step 5. (Optional) Handle numeric features

For a numeric feature X, sort its unique values $x_{(1)} < \cdots < x_{(m)}$ and test thresholds at midpoints

$$t_j = \frac{x_{(j)} + x_{(j+1)}}{2}, \quad j = 1, \ldots, m-1$$

For each t_j, split into $\{X < t_j\}$ and $\{X \geq t_j\}$, compute the weighted Gini, and pick the threshold with the smallest value. This procedure is applied inside *each* node using only the samples that reach that node.

Step 6. (Optional) Prune the grown tree

After growing a large tree, pruning removes weak subtrees using cost–complexity pruning. For a tree T with $|T|$ leaves and training loss $R(T)$,

$$R_\alpha(T) = R(T) + \alpha|T|$$

is minimized over $\alpha \geq 0$. Increasing α favors simpler trees and often improves generalization.

Summary.

- Compare features at each node by weighted Gini; choose the smallest.
- Recurse on children until a stopping rule fires; otherwise keep splitting.
- Leaves predict by majority vote or by class probability.
- Numeric features use midpoint thresholds; large trees can be pruned with cost–complexity.

4 Process of a Regression Tree

Regression trees apply the same recursive partitioning idea, but the target variable is *continuous* rather than categorical. Instead of predicting classes, each leaf stores a constant numeric value—typically the *mean response* of samples that fall into that group.

Step 1. Problem Setup

Suppose we wish to predict a patient's **drug effectiveness score** from two continuous features:

$$X_1 = \text{dosage (mg)}, \qquad X_2 = \text{treatment duration (weeks)}$$

Unlike linear regression, which assumes a single straight line, regression trees divide the input space into smaller regions and fit a constant value within each. This produces a *piecewise-constant model* that adapts to nonlinear patterns automatically.

Step 2. Candidate Splits and Cost Function

At each node, the algorithm searches for the split that minimizes the total *sum of squared errors (SSE)* after division:

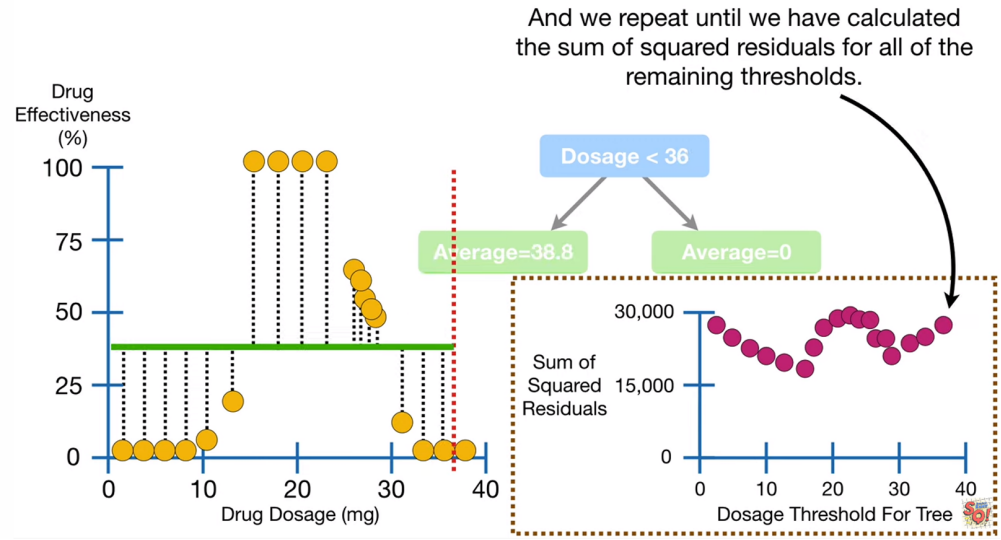
$$\text{SSE}(S) = \sum_{i \in S} (y_i - \bar{y}_S)^2,$$

where $\bar{y}_S$ is the mean target value of node S. For every feature X_j and candidate threshold t,

$$\text{SSE}_{\text{split}}(t) = \text{SSE}(S_L) + \text{SSE}(S_R),$$

and the optimal split is

$$(X_j^*, t^*) = \arg\min_{j,t} \text{SSE}_{\text{split}}(t)$$



Candidate splitting based on SSE curve.

Equivalently, we can think of minimizing the *weighted variance* of child nodes.

Step 3. Using weighted variance (equivalent to minimizing SSE)

Imagine a single-feature dataset (x_i, y_i) representing drug dosage versus response. We test thresholds between sorted x_i values—for instance:

$$t \in \{5.5,\ 10.5,\ 14.5,\ 18.5,\ 22.5\}$$

For each threshold:

- Split data into $S_L = \{x_i < t\}$ and $S_R = \{x_i \geq t\}$.
- Compute means $\bar{y}_L, \bar{y}_R$ and their variances.
- Evaluate the weighted variance:

$$\mathrm{Var}_{\text{weighted}}(t) = \frac{|S_L|}{n}\mathrm{Var}(S_L) + \frac{|S_R|}{n}\mathrm{Var}(S_R)$$

The threshold with the smallest weighted variance becomes the split. For instance, if Dosage < 14.5 yields the largest variance reduction, it becomes the first partition.

Step 4. Recursive Splitting

After the first cut, the same procedure is repeated on each sub-region. Within each region, the algorithm again tries all remaining features and thresholds, always minimizing SSE. This *recursive binary splitting* continues until a stopping condition is met (e.g., minimum node size, maximum depth, or minimal variance reduction).

Step 5. Predictions and Model Shape

Every terminal node (leaf) predicts the average target value of the samples within it:

$$\widehat{y}(x) = \bar{y}_{\text{leaf}(x)}$$

As a result, the final prediction function resembles a step-wise curve that locally averages the data—flexible yet interpretable.

Summary.

- Splits minimize within-node variance or SSE.
- Leaves predict the mean of the target values.
- The piecewise-constant surface adapts to nonlinear and interaction effects naturally.

5 CART (Classification and Regression Trees)

The CART algorithm constructs both classification and regression trees using the same principle: *recursive binary splitting* guided by an impurity or cost metric. The process belongs to the family of **Top-Down Induction of Decision Trees (TDIDT)**, a greedy strategy that repeatedly divides the data to minimize impurity.

Step 1. Recursive Binary Splitting

Growing a tree means deciding **which feature to split on**, **what threshold to use**, and then recursing on the resulting children. Let S be a node with $n = |S|$ samples and let $I(\cdot)$ denote node impurity:

$$I(S) = \begin{cases} \text{Gini or entropy} & \text{(classification)} \\ \text{variance or SSE}/n & \text{(regression)} \end{cases}$$

For each feature X_j and candidate threshold t, split S into $S_L = \{x_i : x_{ij} < t\}$ and $S_R = \{x_i : x_{ij} \geq t\}$. The *weighted post–split impurity* is

$$W(j,t) \;=\; \frac{|S_L|}{|S|} I(S_L) \;+\; \frac{|S_R|}{|S|} I(S_R),$$

and the *impurity decrease* is

$$\Delta(j,t) \;=\; I(S) - W(j,t)$$

CART chooses the split that maximizes the decrease (equivalently, minimizes W):

$$(j^*, t^*) \;=\; \arg\max_{j,t} \Delta(j,t) \;=\; \arg\min_{j,t} W(j,t)$$

Candidate thresholds.

- **Continuous features:** use midpoints between sorted unique values of X_j.
- **Categorical features:** CART uses *binary* splits; when categories are many, an order based on class proportions (classification) or mean response (regression) is often used, then a single cut on that order is evaluated.

Computational note. Presorting each feature once allows scanning thresholds in linear time with running counts/sums, so evaluating all splits is $O(p\, n \log n)$ for presort $+ O(p\, n)$ per node. This procedure is *recursive*: after selecting (j^*, t^*), the same search is applied to each child node (*recursive partitioning*). Because the choice is made to optimize the metric *locally* at every step, CART is a **greedy algorithm**. (Whether the split is actually taken is governed by the stopping rules in Step 2, e.g., requiring $\Delta(j^*, t^*) >$`min_impurity_decrease`.)

Step 2. When to Stop Splitting

If no limit is set, a decision tree can keep splitting until each training sample is isolated in its own leaf—yielding **100% training accuracy** (or zero training SSE for regression) but poor generalization due to overfitting. To prevent this, CART uses *early stopping* rules that halt growth when a candidate split is not sufficiently beneficial or the node is too small/too deep.

Impurity–gain check. Let $I(\cdot)$ denote node impurity (Gini/entropy for classification; variance or SSE for regression). For a node S split into S_L, S_R, the improvement is

$$\Delta \;=\; I(S) \;-\; \frac{|S_L|}{|S|}\, I(S_L) \;-\; \frac{|S_R|}{|S|}\, I(S_R)$$

We stop if $\Delta \le \tau$, where $\tau =$`min_impurity_decrease` is a user–set tolerance (regularization) to prevent overfitting.

Common stopping criteria.

- **Small node:** `min_samples_split` (do not split if the node has too few samples) or `min_samples_leaf` (each child must have at least this many samples).
- **Shallow benefit:** $\Delta \le \tau$ (`min_impurity_decrease`) or $\Delta \le 0$ (no positive gain).
- **Depth cap:** `max_depth` reached (longest root–to–leaf path limit) or a cap on the number of leaves (`max_leaf_nodes`).
- **Purity/constancy:** All labels identical (classification) or all target values identical/constant (regression), so no further reduction is possible.

These criteria control model complexity during growth. After early stopping, a tree can still be simplified with *post–pruning* (e.g., cost–complexity pruning) to further reduce overfitting.

Step 3. Pruning the Overgrown Tree

Even with early–stopping rules, a grown tree can still be too specific to the training set. **Post–pruning** simplifies such a tree by removing weak or unnecessary branches to improve generalization.

Two common approaches.

- **Reduced–error pruning.** Hold out a validation set and iteratively prune a node (replace its subtree by a leaf predicting the majority class / mean response) if doing so does *not* increase the validation error. This proceeds bottom–up from the leaves.
- **Cost–complexity pruning (weakest–link).** Balance fit and size via a penalty parameter α such that:

  $$R_\alpha(T) \;=\; R(T) \;+\; \alpha\,|T|,$$

  where $R(T)$ is the empirical loss (e.g., resubstitution misclassification rate or SSE) and $|T|$ is the number of leaves. Increasing α favors smaller trees.

Weakest–link algorithm (CART). For each internal node t with subtree T_t, define its *effective* complexity value

$$\alpha_t \;=\; \frac{R(t) \;-\; R(T_t)}{|T_t| - 1},$$

i.e., the increase in training loss per leaf removed if we collapse T_t to a single leaf. At each pruning step, remove the subtree with the smallest α_t (the *weakest link*). This produces a nested sequence

$$T_0 \supset T_1 \supset \cdots \supset T_K,$$

from the full tree T_0 down to the root tree T_K. Select the final subtree by choosing α via cross–validation or a validation set (e.g., "one–standard–error" rule for a simpler, competitive tree).

Pre–pruning vs. post–pruning. Early–stopping rules (`min_samples_leaf`, `max_depth`, `min_impurity_decrease`, etc.) are *pre–pruning*: they limit growth during training.
Cost–complexity and reduced–error pruning are *post–pruning*: they first grow a tree (greedy, possibly large) and then cut it back.

Step 4. Common Variants and Metrics

Several decision–tree learners share the same top–down induction idea but differ in their split metrics:

- **ID3:** Information gain (entropy).
- **C4.5:** Gain ratio (addresses multi–valued attributes); handles missing values.
- **C5.0:** Efficient successor to C4.5 with boosting options.
- **CART:** Gini impurity (classification) or variance/SSE (regression); binary splits; cost-complexity pruning.
- **CHAID:** Chi–square tests of association for multiway splits.
- **MARS:** Piecewise–linear regression splines inspired by tree partitioning.

CART grows a tree via recursive, greedy splits that minimize impurity, then controls overfitting with *pre–pruning* (stopping rules) and *post–pruning* (especially cost–complexity/weakest–link). This balance between flexibility and simplicity underlies its strong practical performance.

6 Attribute-Selection Metrics

At each node of a decision tree, we must pick the attribute (and threshold) that makes the *children as homogeneous as possible*. The idea is simple: if the children are "cleaner" (more uniform) than the parent, then the split is useful. Different algorithms quantify "cleaner" with different metrics. In the following, we introduce the most common ones.

Before we start: how a split is scored

Suppose that a node S is split into children $S_1, \ldots, S_m$ (binary for CART, multiway for CHAID). Let $I(\cdot)$ be an impurity measure (Gini, entropy, variance, χ^2–based score, ...). We always combine children by a size-weighted average:

$$\text{PostSplit}(S) \;=\; \sum_{k=1}^{m} \frac{|S_k|}{|S|}\, I(S_k), \qquad \Delta \;=\; I(S) \;-\; \text{PostSplit}(S) \quad \text{(improvement)}$$

Pick the split with the largest improvement Δ (or the smallest PostSplit value).

1) Gini Impurity (CART, classification)

Intuition. "How often would I guess the class wrong if I label randomly according to this node's class proportions?" Lower is better.

$$I_G(S) = 1 - \sum_{i=1}^{J} p_i^2,$$

where p_i is the fraction of class i in the node (sums to 1).

- $I_G = 0$ when all samples are the same class (perfectly pure).
- I_G is largest when classes are evenly mixed.

Mini example. If a node has 6 Yes and 2 No: $p_{\text{Y}} = 0.75,\ p_{\text{N}} = 0.25$.

$$I_G = 1 - (0.75^2 + 0.25^2) = 1 - (0.5625 + 0.0625) = 0.375$$

A split is good if it produces children with *smaller* weighted Gini.

Tips. Fast to compute; default in CART/Random Forests. Entropy and Gini often rank splits similarly.

2) Entropy and Information Gain (ID3, C4.5, classification)

Intuition. Entropy measures *uncertainty/surprise*. Information Gain (IG) asks: "By how much did the split reduce uncertainty?"

$$H(S) = -\sum_i p_i \log_2 p_i, \qquad IG(S, A) = H(S) - \sum_{a \in A} P(a)\, H(S_a)$$

Binary shortcut. For Yes-probability p: $H(p) = -p \log_2 p - (1-p)\log_2(1-p)$ (max at $p = 0.5$).

Mini example (Play Golf). If $H(S) = 0.940$ and splitting on `Outlook` yields weighted child entropy 0.693, then $IG = 0.940 - 0.693 = 0.247$.

Tips.

- log base only changes the scale, not the ranking.
- IG *over-favors* attributes with many unique values (e.g., ID). See Gain Ratio.

3) Gain Ratio (C4.5, classification)

Problem. IG is biased toward attributes that split data into many tiny groups. **Fix.** Normalize IG by the attribute's own "split information" (how scattered the partitions are).

$$\text{GainRatio}(A) \;=\; \frac{IG(S,A)}{\text{SplitInfo}(A)}, \qquad \text{SplitInfo}(A) = -\sum_{a\in A} P(a)\,\log_2 P(a)$$

Interpretation. Prefer attributes that both *reduce entropy a lot* and *don't fragment the data excessively*. Choose the attribute with the largest Gain Ratio.

4) Reduction in Variance (Regression trees)

Intuition. For continuous targets, a pure node has values tightly clustered around their mean. A good split makes each child's values *less spread out*.

$$\text{Var}(S) = \frac{1}{|S|}\sum_{i\in S}(y_i - \bar{y}_S)^2, \qquad \text{PostSplit} = \frac{|S_L|}{|S|}\text{Var}(S_L) + \frac{|S_R|}{|S|}\text{Var}(S_R)$$

Pick the split with the largest variance reduction (equivalently, the largest SSE decrease).

Mini example. If a split turns one scattered group into two tight clusters (each near its own mean), variance drops a lot ⇒ great split.

Tips. Robust to monotone rescalings of y. Still sensitive to outliers; consider robust variants if needed.

5) Chi-Square Statistic (CHAID, classification; multiway splits allowed)

Intuition. "Are class proportions *significantly different* across the branches of this attribute?" Compare observed vs. expected counts assuming independence.

$$\chi^2 \;=\; \sum \frac{(O-E)^2}{E}$$

Usage. Large χ^2 (small p-value) ⇒ attribute and target are associated ⇒ a promising split.

Practical notes.

- Works naturally with categorical targets; CHAID allows *multiway* splits.
- Expected counts in each cell should be reasonably large (rule of thumb: ≥ 5) for the test to be reliable.

Putting it together (how trees actually choose)

1. For each attribute and candidate threshold (or category partition), compute children and their impurity by the chosen metric.
2. Combine children via a size-weighted average; compute improvement Δ.
3. Pick the split with the largest Δ. (Later, Step 2's stopping rules decide whether the gain is *big enough* to accept.)

Quick comparison and guidance

- **Gini vs. Entropy:** Very similar rankings; Gini is slightly faster. Either is fine.
- **Use Gain Ratio** if an attribute has many distinct values (mitigates IG bias).
- **Regression:** Use variance/SSE reduction.
- **CHAID / χ^2:** Good for categorical targets and multiway splits; check expected counts.

Key takeaway

All metrics share the same goal: **find splits that make children as homogeneous as possible**. They differ mainly in *how* they quantify homogeneity (misclassification chance, uncertainty, spread, or statistical association).

7 Discussions and Variations

Decision Trees (DTs) are one of the most intuitive machine learning models. They mimic human reasoning, making predictions through a series of simple questions. For example, "Is the temperature above 30°C?" or "Is the color red?". Each question narrows down possibilities until a final decision (or prediction) is reached. Because of this transparent and visual structure, DTs are often used as the starting point for learning more advanced models.

Advantages of CART

CART (Classification and Regression Tree) has many appealing properties that make it practical and easy to use:

- **Easy to understand and explain.** Decision Trees resemble a flowchart, so anyone can trace a prediction path from the root to a leaf and see exactly why a decision was made. This interpretability makes DTs valuable in domains like medicine, finance, and policy-making.
- **Performs feature selection automatically.** Trees *implicitly select important features* by splitting only on variables that actually reduce impurity. Features that do not contribute are naturally ignored. In other words, feature selection is built into the learning process.
- **Works with both numerical and categorical data.** Unlike algorithms such as logistic regression or SVM, DTs can handle discrete categories (e.g., color = red, blue, green) and continuous features (e.g., temperature, height) in the same model.
- **Handles nonlinear relationships easily.** Because the model recursively splits the space, it can naturally represent complex and nonlinear decision boundaries that linear models would struggle to capture.
- **Minimal data preparation.** Decision Trees don't require feature scaling or normalization. Missing values or different measurement units are usually not problematic, since the splits depend on ordering or equality checks rather than distance.
- **Supports multiple outputs.** CART can be extended to handle multi-output regression or classification, where multiple targets are predicted at once.
- **Intuitive to visualize and communicate.** You can literally "draw" the model. This makes it ideal for reports, educational materials, and non-technical audiences.

Disadvantages of CART

Despite its strengths, a DT also has several weaknesses that must be managed carefully:

- **Overfitting (too complex trees).** If allowed to grow without restriction, a tree can memorize the training data perfectly— splitting again and again until every leaf contains only a few samples. This leads to **overfitting**: excellent accuracy on training data but poor performance on new data. To prevent this, we use *stopping rules* (e.g., `max_depth`, `min_samples_leaf`) or apply pruning.

- **Instability (high variance).** A small change in the data (even one or two samples) can completely alter the structure of the tree. This instability occurs because each split depends heavily on local impurity measures. Ensemble methods like **bagging** and **boosting** are often used to stabilize results.
- **Greedy and locally optimal.** At every node, the algorithm chooses the best split *locally*, not necessarily the best global combination of splits. Therefore, the final tree may not represent the overall best structure possible.
- **Biased toward dominant classes.** When the dataset is unbalanced (e.g., 90% "No" and 10% "Yes"), the tree tends to predict the majority class. Before training, we should balance the data or adjust class weights.
- **Piecewise-constant prediction.** In regression trees, the prediction within each region is a constant mean. This can create sharp jumps at region boundaries, making the model less smooth than other regressors.

How Ensemble Methods Improve CART

To overcome the instability and overfitting issues, modern ML uses **ensemble learning**. The idea is simple: instead of relying on one deep tree, build many trees and combine them. Each tree contributes a "vote" or an average, leading to better generalization.

1. Bagging (Bootstrap Aggregating).

- Train multiple trees on randomly resampled subsets of the data (with replacement).
- Each tree is slightly different because it sees a different sample of data.
- Final prediction = average (for regression) or majority vote (for classification).
- Reduces variance and increases stability.
- The **Random Forest** is the most popular form of bagging, where each tree also uses a random subset of features at each split.

2. Boosting.

- Trees are built *sequentially*, not independently. Each new tree focuses on correcting the mistakes of the previous ones.
- Misclassified examples are given higher weights, forcing the model to learn difficult patterns.
- Classic examples: **AdaBoost**, **Gradient Boosting**, and modern variants like **XGBoost** and **LightGBM**.
- Produces a strong model by combining many shallow "weak" trees.

3. Rotation Forest.

- Each tree is trained on data that has been rotated using **Principal Component Analysis (PCA)** applied to a random subset of features.
- This transformation increases diversity between trees, which improves ensemble performance and prevents all trees from learning the same splits.

Summary: When and Why to Use Trees

- Use **single trees** when interpretability is key — for example, to explain medical or policy decisions.
- Use **ensemble trees** (e.g., Random Forest, Gradient Boosting) when predictive accuracy is more important than simplicity.
- Always monitor for **overfitting** (too deep trees) and **variance** (unstable structure), and apply pruning or ensembles if necessary.

In essence: A single Decision Tree is like one person's opinion—simple, clear, and easy to follow. But by combining many diverse trees through bagging or boosting, we form a well-balanced "committee" that is much wiser and more reliable. This is why Decision Trees, despite their simplicity, form the foundation of many of today's most powerful algorithms.

This is a posting that I summarized with study-purpose and is adapted from lecture notes of NE-795 (Scientific Machine Learning), given by Professor Xu Wu, North Carolina State University.