An Introduction to Decision Trees

Machine Learning Class Handout

November 7, 2025

Abstract

This handout provides a concise overview of Decision Trees, a fundamental algorithm in machine learning. We will cover their structure, training process, common splitting criteria for both classification and regression tasks, and essential techniques for regularization to prevent overfitting.

Contents

1 What is a Decision Tree?

A Decision Tree is a supervised machine learning algorithm that is versatile enough for both classification and regression tasks. It operates by building a model in the form of a tree structure, much like a flowchart.

The tree consists of several types of nodes:

- **Internal Nodes (Decision Nodes):** These nodes represent a "test" on a specific feature (e.g., "Is Outlook = Sunny?"). Each branch coming out of an internal node corresponds to an outcome of the test. The single, topmost node in the tree is called the **Root Node**, which is the special internal node where the process begins and represents the entire dataset.

- **Leaf Nodes (Terminal Nodes):** These are the final nodes at the bottom of the tree. They represent the outcome or decision. For classification, this is a class label (e.g., "Play Tennis"). For regression, it's a continuous value (e.g., "House Price = $300,000").

To make a prediction for a new data point, it is "dropped" from the root node. At each internal node, its feature value is tested, and it follows the corresponding branch. This process is repeated until it reaches a leaf node, which provides the final prediction.

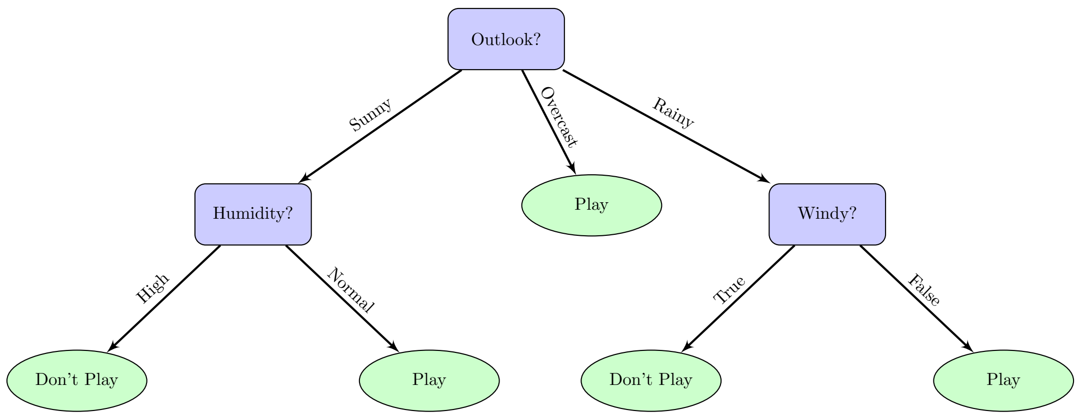

Figure 1: A simple Decision Tree for a "Play Tennis" classification problem. This example uses *categorical* splits.

1.1 Binary Feature Splits

The example in Figure 1 shows splits on *categorical* features (e.g., 'Sunny', 'Overcast', 'Rainy'). However, a more common approach, especially for trees that handle numerical data, is to use **binary splits**.

In this structure, each internal node tests a single feature against a numerical threshold. This results in exactly two branches (hence, "binary"):

- One branch for data points where the condition is **True** (e.g., `Age > 30`).

- One branch for data points where the condition is **False** (e.g., `Age <= 30`).

This method is the standard for algorithms like **CART** (Classification and Regression Trees), as it can be applied to both numerical features (e.g., `Temperature > 25.5`) and categorical features (by asking, e.g., `Outlook == Sunny?`).

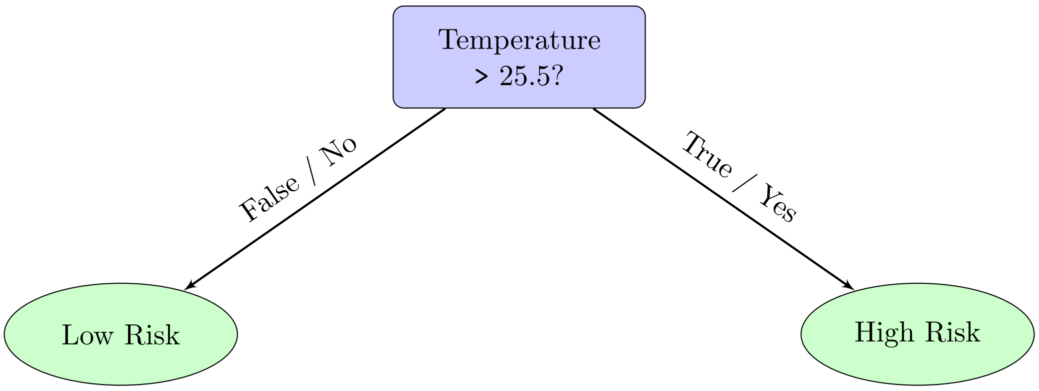

Figure 2: Example of a single **binary split** on a numerical feature.

2 How Decision Trees are Trained

2.1 The Ideal Goal

The ideal goal when training a decision tree is to find the **smallest possible tree** that perfectly fits the training data. A smaller tree is simpler, easier to interpret, and generally less likely to overfit.

Unfortunately, finding the globally optimal (smallest) decision tree is an **NP-hard** problem.

2.2 The Realistic Solution: A Greedy Algorithm

Since the ideal solution is not feasible, we use a heuristic approach: a **greedy algorithm**. The most common one is called **ID3** (or its successors, **C4.5** and **CART**).

This algorithm builds the tree in a top-down, recursive, divide-and-conquer fashion:

1. Start at the root node with the entire training dataset.
2. Find the "best" feature and the "best" split-point for that feature to divide the data. "Best" is determined by a splitting criterion (see Section 3).
3. This split divides the current node's data into two or more subsets.
4. Create new child nodes for each of these subsets.
5. **Recursively** repeat steps 2-4 for each new child node.
6. The recursion stops for a branch when a **stopping criterion** is met (e.g., the node is "pure" – all samples belong to the same class, or a regularization limit is hit). Then we create a terminal node with the best prediction for the dataset we have (i.e., major class or mean of the data).

This is "greedy" because at each step, it makes the decision that is *locally optimal* (i.e., the best split for the *current* node) without considering if this choice will lead to a globally optimal tree.

2.3 A Simple Worked Example

Let's trace the greedy algorithm on a tiny dataset to predict a user's 'Class'. We'll use two categorical features.

ID	**Finished Uni** (X1)	**Owns Car** (X2)	**Class** (Target)
1	No	No	A
2	Yes	Yes	A
3	No	Yes	A
4	Yes	No	B
5	Yes	Yes	A

Goal: Find the best first split at the root node.

1. **Initial State (Root Node):** The dataset is {4 'A', 1 'B'} (impure).
2. **Evaluate Potential Split 1: ‘Owns Car‘ (X2)**
   - *Branch 'Yes':* Samples {2, 3, 5} → {3 'A', 0 'B'}. **This node is pure!**
   - *Branch 'No':* Samples {1, 4} → {1 'A', 1 'B'}. Still impure.

   This split is good, as it creates one perfectly pure node.
3. **Evaluate Potential Split 2: ‘Finished Uni‘ (X1)**
   - *Branch 'Yes':* Samples {2, 4, 5} → {2 'A', 1 'B'}. Still impure.
   - *Branch 'False':* Samples {1, 3} → {2 'A', 0 'B'}. **This node is pure!**

   This split is also good. In this simple case, both features provide the same quantitative information gain.
4. **Greedy Choice (Step 1):** Let's assume the algorithm greedily picks `Owns Car` as the first split.
5. **Recursion (Step 2):**
   - The *Branch 'Yes'* node is pure ({3 'A', 0 'B'}), so it becomes a **Leaf Node** with the prediction **'A'**. The recursion stops for this branch.
   - The *Branch 'No'* node is impure ({1 'A', 1 'B'}), containing Samples {1, 4}. The algorithm must **recursively** find the best split for this new, smaller dataset.
6. **Greedy Choice (Step 2.1):** Now at the new internal node (Samples {1, 4}), the algorithm evaluates remaining features. The only feature left to split on is ‘Finished Uni‘.
   - **Evaluate Split on ‘Finished Uni‘**:
   - *Branch 'Yes':* Sample {4} → {0 'A', 1 'B'}. **Pure!**
   - *Branch 'No':* Sample {1} → {1 'A', 0 'B'}. **Pure!**
7. **Result:** The algorithm selects this split. Both resulting child nodes are pure, so they become leaves. The final tree (Figure 3) now has a depth of 2.

3 Typical Splitting Criteria

The "best" split is found by maximizing some score/metric. For classification, we typically use **Information Gain**; for regression, we minimize **Variance**. These metrics quantify the "purity" of the child nodes created by a split.

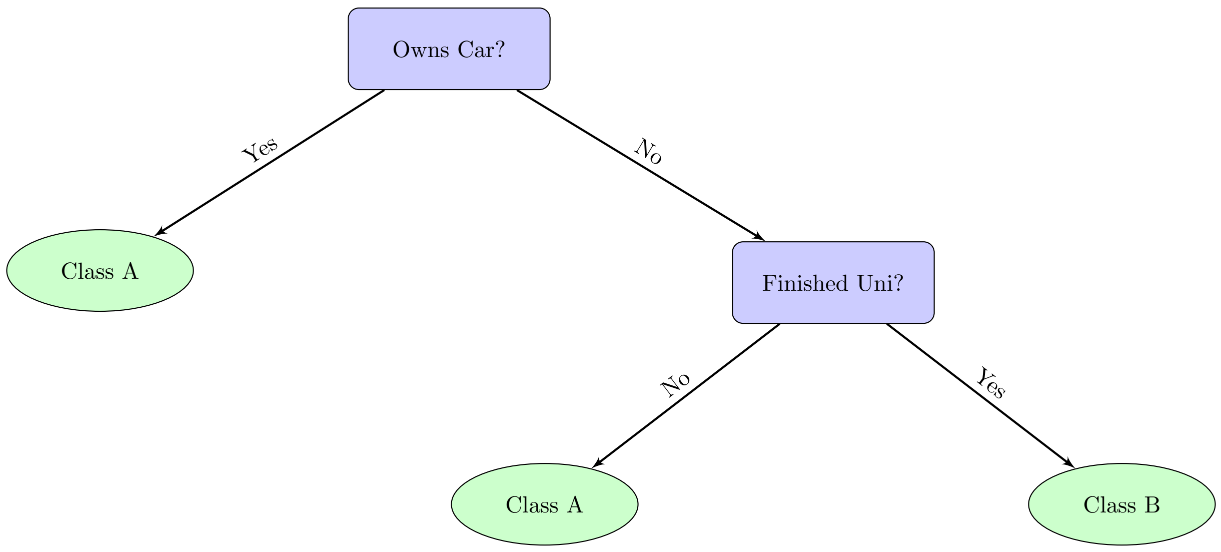

Figure 3: The final, depth-2 tree from our worked example.

3.1 Classification: Entropy and Information Gain

In classification, we want splits that make the resulting child nodes as "pure" as possible (i.e., containing samples from a single class). **Entropy** is a measure of impurity or uncertainty.

For a set of samples S, its entropy is defined as:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

where c is the number of classes and p_i is the proportion of samples in S that belong to class i.

- $H(S) = 0$ if the node is pure (all samples are one class).
- $H(S) = 1$ (for a 2-class problem) if the node is perfectly impure (50%/50% split).

When we perform a binary split on a node S, we divide it into two subsets, S_{left} and S_{right}. The **Information Gain** is the reduction in entropy achieved by this split:

$$IG(S, \text{split}) = H(S) - \left(\frac{|S_{\text{left}}|}{|S|} H(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} H(S_{\text{right}}) \right)$$

The term on the right is the weighted average entropy of the children. The algorithm picks the feature and split-point that **maximizes** the Information Gain.

3.2 Regression: Variance Reduction

In regression, the leaf nodes predict a continuous value, typically the mean of the target values of all samples in that leaf. We want to create splits so that the samples in the leaves have target values that are as close to each other as possible.

The impurity of a node S is measured by the **variance** of its target values:

$$Var(S) = \frac{1}{|S|} \sum_{i \in S} (y_i - \bar{y})^2$$

where $\bar{y}$ is the mean target value of the samples in S.

The algorithm chooses the split (feature and split-point) that **minimizes** the weighted average variance of the resulting child nodes. For a binary split, this is:

$$\text{WeightedAvgVar} = \frac{|S_{\text{left}}|}{|S|} Var(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} Var(S_{\text{right}})$$

This is equivalent to maximizing the **Variance Reduction**, which is defined as $Var(S) - \text{WeightedAvgVar}$.

4 Regularization (Preventing Overfitting)

If allowed to grow indefinitely, a decision tree will "memorize" the training data, leading to **overfitting**. It will perform perfectly on data it has seen but fail to generalize to new, unseen data.

To prevent this, we use **regularization** techniques, which are strategies to limit the tree's complexity. This is also known as **pruning**.

- **Limiting Maximum Depth (`max_depth`):** This is a **pre-pruning** technique. We simply stop the tree from growing past a certain depth (number of levels). This is one of the most effective and common regularization methods.
- **Minimum Samples for a Split (`min_samples_split`):** This **pre-pruning** technique specifies the minimum number of samples a node must have in order to be considered for splitting. If a node has fewer samples than this threshold, it becomes a leaf, even if it's impure.
- **Minimum Samples for a Leaf (`min_samples_leaf`):** A similar **pre-pruning** method. A split is only considered valid if it results in *both* of its child nodes having at least `min_samples_leaf` samples. This prevents creating "slivers"—leaves with very few samples.
- **Post-Pruning (e.g., Cost-Complexity Pruning):** This is an alternative strategy where the tree is grown to its full depth first. Then, branches that add little predictive power (and are likely just fitting noise) are "pruned" away. This is often more effective than pre-pruning but is more computationally complex.