Decision Trees

1 Introduction to Decision Trees

Decision trees are a type of machine learning model used for both classification and regression tasks. The structure resembles a flowchart, making decision trees intuitive to understand. They consist of:

- **Internal Nodes**: Represent decisions based on the value of a specific feature.
- **Branches**: Correspond to the possible outcomes of those decisions.
- **Leaf Nodes**: Indicate the final decision or prediction.

At each node, a decision is made by testing a feature, and the data is divided into branches based on the outcome. In this lecture, we will focus on constructing and evaluating a decision tree.

2 Building a Decision Tree

The goal is to create a tree that can accurately classify data points. The process involves recursively selecting features to split the data, aiming to reduce uncertainty with each split. Note that different trees can be constructed from the same dataset, and we will later learn how to select the one that generalizes best to new data.

2.1 Example: Weather and Tennis

Let's consider building a decision tree for this dataset on deciding whether to play tennis based on weather conditions:

Day	Outlook	Temperature	Humidity	Wind	Tennis?
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	No
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

Table 1: Weather conditions and whether to play tennis.

Step-by-Step Process for Building the Tree

To construct a decision tree:

1. **Start with the entire dataset at the root node.** At this point, the data includes all examples, both positive (Yes) and negative (No) for playing tennis.
2. **Choose the best feature to split on.** The goal is to select a feature that divides the dataset into groups where the target variable (Tennis?) is as pure as possible (i.e., mostly Yes or No). We will consider different criteria to measure impurity, such as misclassification error, Gini impurity or entropy.
3. **Split the dataset based on the chosen feature.** Create branches corresponding to the possible values of the feature.
4. **Repeat the process recursively for each branch.** Continue splitting the data until all examples are classified (either all Yes or all No) or no further features remain.

First Split: Outlook

We start by splitting on the **Outlook** feature:

- **Sunny**: Days 1, 2, 8, 9, 11.

- **Overcast**: Days 3, 7, 12, 13.
- **Rain**: Days 4, 5, 6, 10, 14.

Observations:

- When Outlook is **Overcast**, all examples are labeled Yes for playing tennis. Thus, we can classify these examples directly and stop further splitting for this branch.
- For **Sunny** and **Rain**, we need additional splits because the labels are mixed.

Second Split: Sunny Days

We now consider splitting the **Sunny** branch. The next feature to use could be **Humidity**:

- **High Humidity**: Days 1, 2, 8 (No).
- **Normal Humidity**: Days 9 (Yes) and 11 (No).

At this point:

- When Humidity is **High**, the outcome is always No, so we can classify these directly.
- For **Normal Humidity**, we still have mixed labels (Yes and No), so further splits are needed.

Recursive Splitting and Stopping Criteria

The process continues recursively:

- For the **Rain** branch, split based on **Wind**:
  - **Weak Wind**: Days 4, 5, 10 (Yes).
  - **Strong Wind**: Days 6, 14 (No).
- The algorithm stops branching when all remaining examples at a node have the same label (pure node), or no more features can be split.

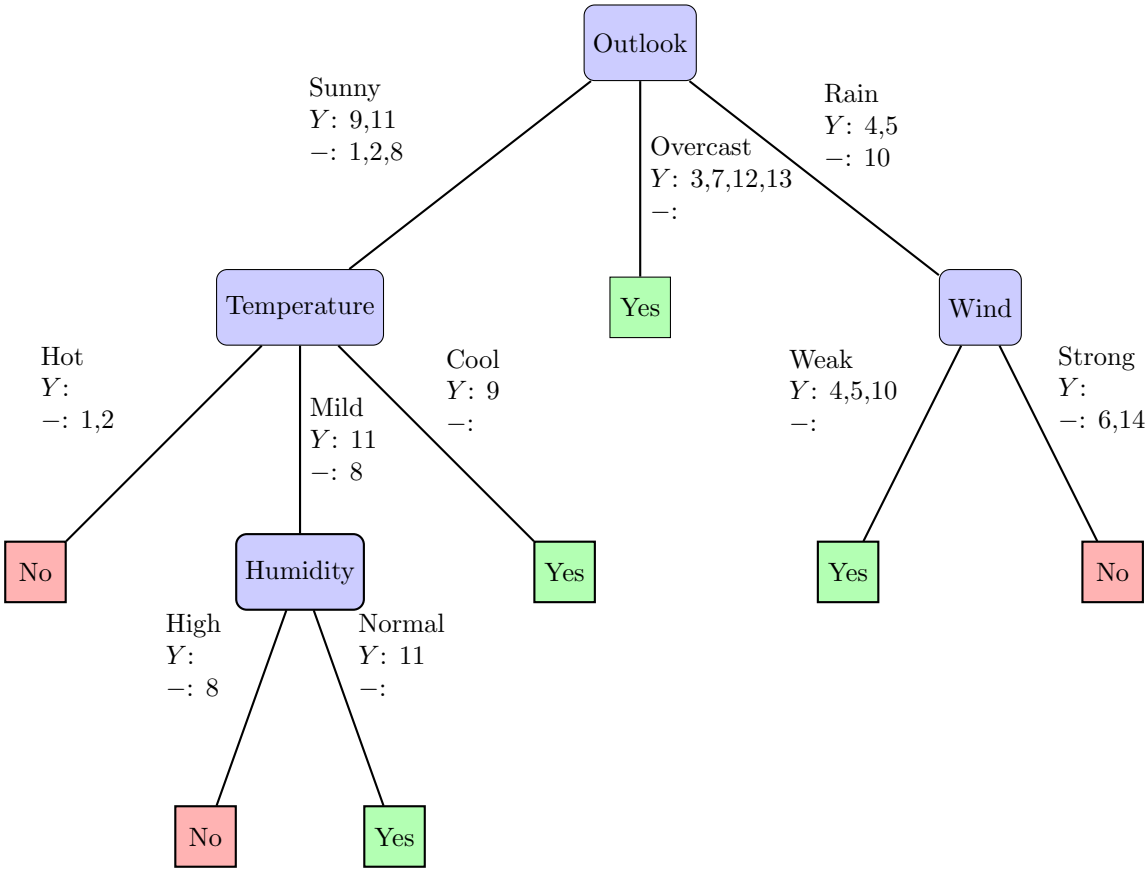


Figure 1: Final decision tree for predicting whether to play tennis based on weather conditions.

2.2 Base Cases to Stop Recursion

- **Pure Nodes**: A node where all instances belong to the same class. These nodes are leaf nodes and no further splitting is necessary.
- **No Features Left**: If there are no remaining features to split on and the node is not pure, majority voting is used for classification.
- **Handling Missing Values**: If a feature is absent in some instances, it can be managed using the majority class of the parent node.

2.3 Evaluating Splits: Impurity Measures

When building a decision tree, an important step is determining the best feature to split on at each decision node. To do this, we need a way to measure how "impure" the data is before and after each potential split. Impurity refers to how mixed the target labels (e.g., Yes or No for a classification task) are in a given subset of the data. Ideally, we want each split to reduce impurity as much as possible, bringing us closer to a state where each branch contains only a single class.

There are several common ways to quantify impurity:

- **Gini Impurity**: This measure calculates the probability of misclassifying a randomly chosen element from the dataset. A lower Gini impurity indicates that most examples in a node belong to the same class.
- **Entropy (Information Gain)**: Entropy measures the uncertainty or disorder within a node. Splitting on a feature that reduces entropy results in a higher information gain, meaning the split brings more clarity to the classification.
- **Misclassification Error**: This metric looks at the proportion of misclassified instances within a node.

For this lecture, we will focus on **misclassification error**. As we continue to build the tree, our goal will be to reduce impurity at each step.

To quantify impurity in a binary classification problem, we define the probabilities p_0 and p_1, where:

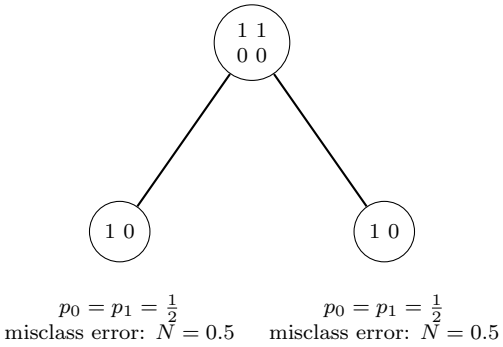
- p_0 is the proportion of examples with the label '0' in a node,
- p_1 is the proportion of examples with the label '1'.

Even when a node is impure, we still need to assign it a classification. This can be done using **majority vote**, where we classify the node based on the class that occurs most frequently $\max\{p_0, p_1\}$. By classifying according to the majority, we minimize the misclassification error, which is always less than or equal to 0.5. If the error exceeded 0.5, we would simply classify the node the other way to improve accuracy.

3 Constructing a Tree to Minimize Impurity

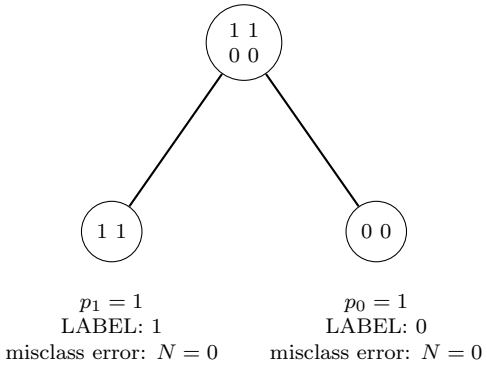
Let's consider a simple binary dataset: {1, 1, 0, 0}. Based on this dataset, we can construct two possible decision trees.

The first tree splits the dataset into two equal parts, with each leaf containing one positive and one negative instance:



In this case, both leaves have equal proportions of '1' and '0', resulting in $p_0 = \frac{1}{2}$ and $p_1 = \frac{1}{2}$. The misclassification error for each leaf is 0.5, and splitting the data like this does not reduce the impurity.

Now let's consider a second tree, where each leaf contains only one class:



In this tree, each leaf contains only one class: the left leaf contains two '1's, and the right leaf contains two '0's. The misclassification error for both leaves is 0, as all instances in each leaf are of the same class.

Clearly, the second tree is better since it completely separates the classes, reducing the misclassification error to 0. This is an example of how choosing the right features for splitting can lead to a more informative, less impure tree. Thus, the goal is to choose features that maximize the purity of the resulting subsets, minimizing the overall misclassification error.

4 Decision Tree Algorithms Are Greedy

Decision tree algorithms are considered greedy because they make the best local choice at each step (selecting the feature that maximizes impurity reduction). This approach is efficient but may not always yield the optimal global solution. Therefore, it is common to generate lots of different decision trees in a greedy way and then choose the one that generalizes best to new data.

5 Bias-Variance Tradeoff

When selecting between different decision tree models, it is essential to choose the one that balances the **bias-variance tradeoff**.

In the decision tree we built on the tennis decision, we achieved a perfect fit to the training data, meaning the model classified every single point correctly. While this might seem ideal, it is a classic example of **overfitting**, where the model becomes too specific and memorizes the training data rather than generalizing from it. A model that overfits often struggles with new, unseen test data because it fails to capture broader patterns beyond the training set. This is an instance of **low bias** and **high variance**, where the model fits the training data perfectly but may perform poorly on new data.

This idea is similar to what we encounter in **k-nearest neighbors (k-NN)**. When we set $k = 1$, the model has full flexibility to classify every point in the training data, but this flexibility often leads to **overfitting**, just as we saw in the decision tree that perfectly classified the training data.

There is a clear advantage to using a simpler model. While a simpler decision tree may not perfectly classify all training data points (leading to a higher **bias**), it is likely to generalize better to unseen data due to lower **variance**. For example, a model that has an **accuracy** of less than perfect on the training data, such as a **20% error rate**, might exhibit higher **bias** but lower **variance**, making it more robust to new data. This is why, in practice, we often prefer smaller decision trees that focus on the main trends in the data.

6 Addressing Overfitting

Techniques to address overfitting include:

- **Pruning**: Removes branches that have little impact on classification accuracy.
- **Limiting Tree Depth**: Restricts the number of levels in the tree to

reduce complexity.

- **Minimum Samples for Splits**: Ensures that splits are made only when a node contains a sufficient number of samples.

7 Model Selection Techniques

To evaluate how well a decision tree generalizes to unseen data, you can use:

1. **Holdout Method**: This technique involves splitting the dataset into separate *training* and *testing* sets. The model is trained on the training set and evaluated on the test set to assess its performance. The goal is to build a model that generalizes well from the training data to unseen test data, minimizing the **generalization error**: the error on the test set. Data are often shuffled first (e.g., if they were compiled by different sources) to ensure that the training and test sets are representative and reduce bias from the original data order.

2. **Cross-Validation**: In this method, the dataset is split into multiple subsets or *folds*. The model is trained and evaluated multiple times, each time using a different fold as the test set and the remaining folds as the training set. This helps obtain a more reliable estimate of the model's ability to generalize to new data. The specific steps are:

   (a) Partition the dataset k times to create k folds (also known as **validation sets**).

   (b) Each fold is used as a test set, while the remaining folds are used for training.

   (c) The accuracy is averaged across all partitions to approximate the model's overall performance. A typical choice is $k = 10$ for **k-fold cross-validation**, but other values can be chosen depending on the dataset and the problem.

   An extreme form of cross-validation is **leave-one-out cross-validation**, where $k = n$ (the sample size), meaning each instance in the dataset is used as a test set exactly once, while the remainder is used for training.