📘 Decision Tree – Play Tennis Example

🌳 1. Introduction

A **Decision Tree** is a supervised machine learning algorithm that makes decisions by splitting data into branches based on conditions.

It works like asking a sequence of **questions** or **if-else statement**, where each question reduces uncertainty and brings us closer to the final decision (Yes/No).

🧩 2. Key Terms

1️⃣ Entropy

A measure of impurity (disorder) in the data.

- Entropy = 0 → Pure (all Yes or all No)
- Entropy = 1 → Maximum impurity (mixed equally)

$$ Entropy(S) = -p_{yes}\log_2(p_{yes}) - p_{no}\log_2(p_{no}) $$

2️⃣ Information Gain

Reduction in entropy after splitting on an attribute.

$$ Gain(S, A) = Entropy(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|}Entropy(S_v) $$
The attribute with **highest Information Gain** becomes the **root node**.

3️⃣ Root Node

The first and most important split in a decision tree.
It provides the **highest reduction in impurity**.

📊 3. Play Tennis Dataset

Weather	Temperature	Humidity	Wind	Play Tennis?
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No

Weather	Temperature	Humidity	Wind	Play Tennis?
Overcast	Hot	High	Weak	Yes
Rainy	Mild	High	Weak	Yes
Rainy	Cool	Normal	Weak	Yes
Rainy	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rainy	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rainy	Mild	High	Strong	No

Rainy | Cool | Normal | Strong | No |

🔢 4. Entropy of Full Dataset

Total entries = 14
Yes = 9
No = 5

$$ p_{yes} = \frac{9}{14}, \quad p_{no} = \frac{5}{14} $$$$ Entropy(S) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) -\frac{5}{14}\log_2\left(\frac{5}{14}\right) $$
Final Entropy:
$$ Entropy(S) = 0.94 $$

Dataset summary: total = 14, Yes = 9, No = 5

---

2) Information Gain formula

For an attribute (A) with possible values (v):

$$ Gain(S,A) = Entropy(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|}Entropy(S_v) $$
We compute (Entropy(S_v)) for each value (v), then the weighted sum, then the gain.

---

3) Gain calculations (step-by-step)

A. **Weather** (values: Sunny, Overcast, Rainy)

Counts & entropies:

- Sunny: 5 instances (Yes=2, No=3)
  $$Entropy(Sunny) = -\frac{2}{5}\log_2\frac{2}{5}-\frac{3}{5}\log_2\frac{3}{5}\approx 0.971$$
- Overcast: 4 instances (Yes=4, No=0)
  $$Entropy(Overcast) = 0.000$$
- Rainy: 5 instances (Yes=3, No=2)
  $$Entropy(Rainy) \approx 0.971$$

Weighted entropy:

$$ E_{Weather} = \frac{5}{14}(0.971)+\frac{4}{14}(0.000)+\frac{5}{14}(0.971) \approx 0.694 $$
Information Gain:

$$ Gain(S,Weather) = 0.940 - 0.694 \approx \mathbf{0.247} $$

B. **Temperature** (values: Hot, Mild, Cool)

Counts & entropies:

- Hot: 4 (Yes=2, No=2)
  $$Entropy(Hot)= -\frac{2}{4}\log_2\frac{2}{4}-\frac{2}{4}\log_2\frac{2}{4}=1.000$$
- Mild: 6 (Yes=4, No=2)
  $$Entropy(Mild)\approx 0.918$$
- Cool: 4 (Yes=3, No=1)
  $$Entropy(Cool)\approx 0.811$$

Weighted entropy:

$$ E_{Temp} = \frac{4}{14}(1.000)+\frac{6}{14}(0.918)+\frac{4}{14}(0.811) \approx 0.911 $$
Information Gain:

$$ Gain(S,Temperature) = 0.940 - 0.911 \approx \mathbf{0.029} $$

C. **Humidity** (values: High, Normal)

Counts & entropies:

- High: 7 (Yes=3, No=4)
  $$Entropy(High)\approx 0.985$$
- Normal: 7 (Yes=6, No=1)
  $$Entropy(Normal)\approx 0.592$$

Weighted entropy:

$$ E_{Humidity} = \frac{7}{14}(0.985)+\frac{7}{14}(0.592) \approx 0.788 $$
Information Gain:

$$ Gain(S,Humidity) = 0.940 - 0.788 \approx \mathbf{0.152} $$

D. **Wind** (values: Weak, Strong)

Counts & entropies:

- Weak: 8 (Yes=6, No=2)
  $$Entropy(Weak)\approx 0.811$$
- Strong: 6 (Yes=3, No=3)
  $$Entropy(Strong)=1.000$$

Weighted entropy:

$$ E_{Wind} = \frac{8}{14}(0.811)+\frac{6}{14}(1.000) \approx 0.892 $$
Information Gain:

$$ Gain(S,Wind) = 0.940 - 0.892 \approx \mathbf{0.048} $$

---

4) Final table (rounded)

Attribute	Weighted Entropy	Information Gain
Weather	0.694	**0.247**
Humidity	0.788	0.152
Wind	0.892	0.048
Temperature	0.911	0.029

---

5) Conclusion — Why **Weather** is the root

- **Weather** yields the **largest Information Gain (0.247)** among all attributes.
- That means splitting on **Weather** reduces the dataset entropy the most (creates purer child nodes), so it is chosen as the **root node**.

---

6) Next steps (after root)

After choosing Weather as root, the tree is constructed recursively:

- For branch **Sunny** → compute gains again among remaining features (Humidity, Wind, Temperature) using only Sunny rows → choose best split (Humidity in this dataset).

- For **Overcast** → becomes pure (all Yes) → stop.
- For **Rainy** → compute gains among remaining features → choose best split (Wind in this dataset).

This process repeats until all leaf nodes are pure or stopping criteria are met.

-

🧮 5. Information Gain (Final Results Only)

Attribute	Information Gain
Weather	**0.247**
Humidity	0.151
Wind	0.048
Temperature	0.029

✔️ Highest IG → **Weather**

So, **Weather becomes the Root Node**.

🌳 6. Final Decision Tree (Play Tennis)

🎯 7. Applications of Decision Trees

- Weather prediction
- Medical diagnosis
- Loan approval systems
- Fraud detection
- Customer behavior prediction
- Game AI decision making
- Student performance classification

📝 8. Summary

- Decision Trees split data to reduce impurity.
- **Entropy** measures impurity.
- **Information Gain** measures reduction in impurity.
- Attribute with highest IG becomes **root node**.
- For the Play Tennis dataset → **Weather** is the root.

---

9. Gini Impurity

📌 What is Gini Impurity?

Gini Impurity tells us **how mixed or impure** a node is.

Intuition:

- If a node has **only one class** → pure → Gini = 0
- If a node has **mixed classes** → impure → Gini > 0
- Higher Gini = worse split
- Lower Gini = better split

Decision Tree tries to **reduce Gini** as much as possible.

📘 Formula

For a node with classes and probabilities ($p_1, p_2, ..., p_k$):

$$ Gini = 1 - \sum_{i=1}^{k} p_i^2 $$

For binary classification (Yes/No):

$$ Gini = 1 - (p_{yes}^2 + p_{no}^2) $$

🟦 Example 1: Pure Node

Data: 10 samples → all "Yes"

- ($p_{yes}=1$)
- ($p_{no}=0$)

$$ Gini = 1 - (1^2 + 0^2) = 0 $$

✔ Pure
✔ No impurity

🟧 Example 2: Mixed Node

Data: 5 Yes, 5 No

- (p_{yes}=0.5)
- (p_{no}=0.5)

$$ Gini = 1 - (0.5^2 + 0.5^2) = 1 - (0.25 + 0.25) = 0.5 $$

This is **maximum impurity** in binary classification.

🔍 Why is Gini Used?

✔️ 1. Fast to compute

No logarithms → very efficient.

✔️ 2. Gives very similar results to Entropy

Most of the time, both choose the **same** split.

✔️ 3. More sensitive to purity

Gini reacts quickly to class mixing.

✔️ 4. Works well for classification trees

It is the **default criterion** in sklearn:

```
DecisionTreeClassifier(criterion="gini")
```

criterion	Meaning
`"gini"`	Split based on Gini impurity
`"entropy"`	Split based on Information Gain
`"log_loss"`	Uses probabilistic impurity

In [ ]:

🌳 Decision Tree Hyperparameters

Decision Trees can easily **overfit**, so we use hyperparameters to control the tree's growth and improve performance.

Here are the most important hyperparameters in a Decision Tree Classifier.

1️⃣ criterion (Impurity Measure)

Controls **how splits are chosen**.

Options:

- `"gini"` → Gini Impurity (default, faster)
- `"entropy"` → Information Gain (uses log, slower)
- `"log_loss"` → entropy-like, probability-based

Example:

```
DecisionTreeClassifier(criterion="entropy")
```

2 What is max_depth?

- **max_depth** controls the **maximum number of levels** in a Decision Tree from the **root node** down to the **leaf nodes**.
- It is one of the most important hyperparameters because it **directly affects model complexity**:
  - **Too high** → tree grows very deep → may **overfit** training data.
  - **Too low** → tree is shallow → may **underfit** the data.

```
               Weather      ← Depth 1
       /          |        \
   Sunny       Rainy     Overcast  ← Depth 2
     /                       \
  Humidity                  Windy    ← Depth 3
/   \                     /     \
High   Low               Strong  Weak ← Depth 4
```

Controls how deep the tree can grow.

Large depth (20): captures all patterns → risk of overfitting

Small depth (3 or 4): generalizes well → reduces overfitting

Example

If max_depth=2, the model will only split the tree twice → simple model.

3. What is min_samples_split?

- **min_samples_split** controls the **minimum number of samples (rows)** a node must have **before it can be split**.
- If a node has fewer samples than this value, **splitting is not allowed**.
- Helps **prevent overfitting** by stopping tiny nodes from being split.

```
                 ROOT NODE
              (5 samples/rows)
                     |
       --------------------------------
```

```
    |                |                |
  Sunny            Rainy          Overcast
(2 rows)         (2 rows)          (1 row)
```

- **Default = 2** (can overfit small datasets)
- Use **cross-validation** to tune for your dataset.

4. What is min_samples_leaf?

- **min_samples_leaf** sets the **minimum number of samples (rows) required in a leaf node**.
- Unlike `min_samples_split` which controls **when a node can split**, `min_samples_leaf` controls **the size of the final leaf nodes** after splitting.
- Helps **prevent tiny, meaningless leaves** that may cause overfitting.

```
                ROOT NODE (6 samples)
                          |
         -------------------------------
         |                |             |
    Sunny (2)        Rainy (3)     Overcast (1)

- Suppose `min_samples_leaf = 2`
- Split of **Overcast node** would create a leaf with 1 sample → **not allowed**
- Any split that would create a leaf with **fewer than 2 samples** is rejected.
```

5. What is max_features?

- **max_features** controls **how many features (columns)** the tree can consider **when looking for the best split** at each node.

- It introduces **randomness** and can help **reduce overfitting**.

- Default = None (all features considered)

- Total features = 4 (Weather, Temperature, Humidity, Wind)

- Suppose `max_features = 2`

    - At each node, the tree **randomly selects 2 features** to consider for splitting.
    - Example:
        - Node 1: Features selected → Weather & Humidity
        - Node 2: Features selected → Temperature & Wind
    - Reduces overfitting by not always using all features.

- Tree only checks **Weather** and **Humidity** to decide the split.
- Other features (Temp & Wind) are **ignored for this node**.

- **Introduces randomness** → good for **ensemble methods** like Random Forest.
- Can **reduce overfitting** by limiting the features considered at each split.
- Works differently depending on tree type:
  - **DecisionTreeClassifier** / **DecisionTreeRegressor** → limits features per node
  - **RandomForest** → strongly recommended to set max_features < total features

---

6. What is max_leaf_nodes?

- **max_leaf_nodes** sets the **maximum number of leaf nodes (endpoints)** in the tree.
- Helps **control tree complexity** by limiting the number of final decision nodes.

- Prevents **overfitting** by keeping the tree **simpler**.
- Smaller value → simpler tree → may underfit
- Larger value → more leaves → may overfit

7. What is min_impurity_decrease?

- **min_impurity_decrease** sets the **minimum reduction in impurity** required to make a split.
- Parent Node: Impurity = 0.8
- Split would reduce impurity to 0.78

-min_impurity_decrease = 0.05

- → 0.02 < 0.05 → split not allowed

Hyperparameter	Meaning	Analogy / Visualization	Effect
max_depth	Maximum levels in tree	Building height	Controls over/underfitting
min_samples_split	Minimum samples to split a node	Minimum students to divide class (before)	Stops tiny nodes from splitting
min_samples_leaf	Minimum samples in a leaf	Minimum team size (after splitting)	Prevents tiny leaves
max_features	Maximum features to consider at each split	Textbooks student can check	Introduces randomness, reduces overfitting
max_leaf_nodes	Maximum number of leaf nodes	Maximum number of final teams	Limits complexity, prevents overfitting
min_impurity_decrease	Minimum impurity reduction to split	Only split if improvement is worth it	Avoids unnecessary weak splits

better Tuning

- Use **cross-validation** to select the best values.
- Combine hyperparameters for **better control over tree complexity**:
  - `max_depth` + `min_samples_split` + `min_samples_leaf` → control overfitting
  - `max_features` + `min_impurity_decrease` → improve generalization
  - `max_leaf_nodes` → simplify final tree

Essemble Techinques

Ensemble Learning (Bagging, Boosting, Gradient Boosting)

What is Ensemble Learning?

Ensemble Learning is a technique in machine learning where multiple models (weak learners) are combined to produce a more accurate and stable prediction.

Key idea: A group of weak learners together forms a strong learner.

Bagging (Bootstrap Aggregating)

Definition

Bagging is an ensemble method where multiple **independent** models are trained on **bootstrapped samples** (sampling with replacement) from the dataset.
Their predictions are then combined.

- For classification → **Majority Vote**
- For regression → **Average**

Goal

To reduce **variance** and prevent overfitting.

How Bagging Works

1. Create multiple bootstrapped datasets.
2. Train one weak learner (usually decision tree) on each dataset.
3. Aggregate all predictions using voting or averaging.

Intuition

Each model sees slightly different data, produces slightly different results.
Combining them reduces overall error.

Examples

- Random Forest
- Bagged Decision Trees

Boosting

Definition

Boosting is a sequential ensemble technique where each new model focuses on **correcting the errors** made by previous models.
Models are combined using **weighted voting** or **weighted averaging**.

Goal

To reduce **bias** and convert weak learners into a strong learner.

How Boosting Works

1. Train the first model.
2. Identify misclassified samples and increase their weights.
3. Train the next model focusing on difficult samples.
4. Combine all models with weighted contributions.

Intuition

Early models fail on hard samples.
Later models focus more on those, improving accuracy gradually.

Examples

- AdaBoost
- Gradient Boosting

- XGBoost
- LightGBM
- CatBoost

Gradient Boosting (GBM)

Definition

Gradient Boosting improves models step-by-step by using the **gradient of the loss function**.
Each new model fits the **residual errors** (actual − predicted) of the previous model.

Goal

To minimize the loss function by learning in the direction of the **negative gradient**.

How Gradient Boosting Works

1. Start with a simple model.
2. Calculate residuals (errors).
3. Train a new weak model to predict these residuals.
4. Update predictions using a learning rate.
5. Repeat for many steps.

Intuition

Each new weak learner adds a small correction.
Many small corrections combine into a powerful model.

Examples

- Gradient Boosting Machine (GBM)
- XGBoost
- LightGBM
- CatBoost

Summary Table

Technique	Goal	Training Style	Key Idea	Final Output
Bagging	Reduce variance	Parallel	Train on bootstrapped samples	Majority vote / Average
Boosting	Reduce bias	Sequential	Focus on misclassified samples	Weighted vote
Gradient Boosting	Minimize loss	Sequential	Learn from residuals using gradients	Sum of weak learners

Bagging

Boosting

Random Forest

In [ ]:

```

```