

Decision Tree Algorithm for Classification

A **Decision Tree** is a supervised learning algorithm used for both classification and regression tasks. It splits the dataset into subsets based on the value of input features. This splitting continues recursively until a stopping condition is met (like a pure class or a maximum depth). It uses metrics like **Entropy** and **Information Gain** to decide the best feature for splitting at each node.

✓ Key Concepts:

1. Entropy

Entropy measures the **impurity** or **uncertainty** in a dataset.

For a dataset S with binary classification (e.g., Yes/No), entropy is calculated as:

$$\text{Entropy}(S) = -p_+ \log_2(p_+) - p_- \log_2(p_-)$$

Where:

- p_+ is the proportion of positive examples (e.g., Yes)
- p_- is the proportion of negative examples (e.g., No)

If all examples belong to one class, entropy is 0 (pure). If split evenly, entropy is 1 (maximum impurity).

2. Information Gain

Information Gain measures the **reduction in entropy** after a dataset is split on a feature.

$$\text{IG}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \cdot \text{Entropy}(S_v)$$

Where:

- $\text{IG}(S, A)$ is the information gain by splitting on attribute A
- S_v is the subset of S where A has value v

The feature with the highest information gain is chosen for the split.

🔍 Tennis Play Example:

Suppose we have a dataset where we try to predict if someone will play tennis based on **Weather Conditions**:

Outlook	Temperature	Humidity	Wind	PlayTennis
---------	-------------	----------	------	------------

Sunny	Hot	High	Weak	No
-------	-----	------	------	----

Sunny	Hot	High	Strong	No
-------	-----	------	--------	----

Outlook Temperature Humidity Wind PlayTennis

Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Step-by-Step:

Step 1: Calculate Entropy of the Entire Dataset

There are 9 Yes and 5 No outcomes.

$$\text{Entropy}(S) = -9/14 \log_2(9/14) - 5/14 \log_2(5/14) \approx 0.940$$

Step 2: Calculate Information Gain for “Outlook”

Let's split by **Outlook**:

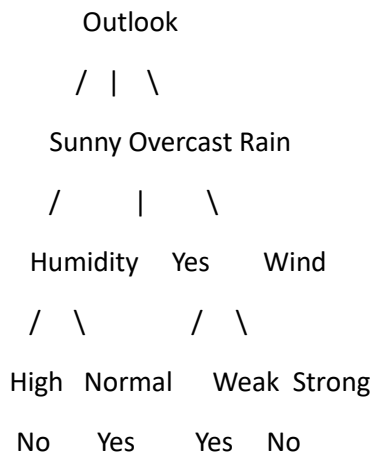
- **Sunny**: 5 samples → 2 Yes, 3 No → Entropy ≈ 0.971
- **Overcast**: 4 samples → 4 Yes → Entropy = 0
- **Rain**: 5 samples → 3 Yes, 2 No → Entropy ≈ 0.971

$$\text{IG}(S, \text{Outlook}) = 0.940 - \left(\frac{5}{14} \cdot 0.971 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.971 \right) \approx 0.940 - 0.693 = 0.247$$

Similar IG is calculated for other features (Temperature, Humidity, Wind), and the feature with the **highest IG** is chosen as the root node.

Let's say **Outlook** gives the highest IG—it becomes the **root node**. Then recursively repeat for each child node with the remaining features.

Final Decision Tree (Partial):



Summary:

- Decision Trees use **entropy** to measure impurity.
- **Information Gain** guides which feature to split on.
- The algorithm builds the tree by choosing the feature with the **highest gain** at each step.
- Trees can overfit—**pruning** is used to avoid that.

2. Logistic Regression – Explained

What is Logistic Regression?

Logistic Regression is a **supervised learning algorithm** used for **binary classification** problems. Despite the name "regression", it is used to **predict categorical outcomes**—specifically, whether an instance belongs to **class 0** or **class 1**.

It models the **probability** that a given input belongs to a particular class using a **logistic (sigmoid) function**.

Why Not Linear Regression?

If we used **Linear Regression** for classification, we might get outputs like -2, 1.5, or 5. But probabilities must lie in **[0, 1]**.

So, we apply the **sigmoid function** to map the linear combination of inputs to a valid probability.

Logistic Regression Model

Given input features $X = (x_1, x_2, \dots, x_n)$, logistic regression computes:

$$z = w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n = w^T x$$

Probability that $y=1: \hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$ $\text{Probability that } y = 1: \hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$

Where:

- w : weights (learned during training)
- $\sigma(z)$: **sigmoid function**
- \hat{y} : predicted probability that the class is 1

Sigmoid Function

The **sigmoid function** is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

It squashes any real-valued number into a value between 0 and 1, making it ideal for predicting **probabilities**.

Properties:

- As $z \rightarrow -\infty$, $\sigma(z) \rightarrow 0$
- As $z \rightarrow \infty$, $\sigma(z) \rightarrow 1$
- At $z=0$, $\sigma(z) = 0.5$

Binary Classification

- If $\hat{y} \geq 0.5$, predict **class 1**
- If $\hat{y} < 0.5$, predict **class 0**

Training Logistic Regression

Training involves adjusting the weights w to minimize the **log loss (binary cross-entropy)**:

$$\text{Loss} = -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$$

Where:

- y is the true label (0 or 1)
- \hat{y} is the predicted probability

We typically use **gradient descent** to minimize this loss.

Summary

Component	Description
Model Equation	$\hat{y} = \sigma(w^T x)$
Output	Probability between 0 and 1
Prediction	Class 1 if $\hat{y} \geq 0.5$, else Class 0
Activation Function	Sigmoid
Loss Function	Binary Cross-Entropy

3. Evaluating Supervised Classification Models

To assess the performance of a **supervised classification model**, especially for binary classification, we use various metrics derived from the **confusion matrix**.

Confusion Matrix

The **confusion matrix** is a table that summarizes the performance of a classification algorithm:


	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Evaluation Metrics with Formulas

1. Accuracy

Measures the overall correctness of the model.


$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

 **Limitation:** Misleading in imbalanced datasets.

2. Precision

Out of all predicted positives, how many were actually positive?

$$\text{Precision} = \frac{TP}{TP + FP}$$

 High precision means **few false positives**.

3. Recall (Sensitivity or True Positive Rate)

Out of all actual positives, how many were correctly predicted?

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

✅ High recall means **few false negatives**.

4. F1-Score

Harmonic mean of precision and recall. Best when you need a balance between them.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

5. ROC-AUC (Receiver Operating Characteristic - Area Under Curve)

- ROC Curve plots **True Positive Rate (Recall)** vs **False Positive Rate (FPR)**:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

- AUC** (Area Under Curve) summarizes this curve in a single number.
 - AUC = 1 → Perfect classifier
 - AUC = 0.5 → Random guess
-

Example:

Let's say we have a binary classifier tested on 100 instances:

	Predicted Positive	Predicted Negative
Actual Positive	40 (TP)	10 (FN)
Actual Negative	20 (FP)	30 (TN)

Compute Metrics:

- Accuracy:**

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total}} = \frac{40 + 30}{100} = 0.70$$

- Precision:**

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{40}{40 + 20} = 0.667$$

- Recall:**

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{40}{40 + 10} = 0.80$$

- F1-Score:**

$$= 2 \cdot 0.667 \cdot 0.800.667 + 0.80 \approx 0.727 = 2 \cdot \frac{0.667 \cdot 0.80}{0.667 + 0.80} \approx 0.727$$

- **FPR (for ROC):**

$$= \frac{FP}{FP + TN} = \frac{20}{20 + 30} = 0.40$$

(Full ROC curve would be generated by varying classification threshold.)

✅ Summary Table

Metric	Measures	Formula
Accuracy	Overall performance	$\frac{TP + TN}{Total}$
Precision	Quality of positive prediction	$\frac{TP}{TP + FP}$
Recall	Coverage of actual positives	$\frac{TP}{TP + FN}$
F1-Score	Balance between P & R	$2 \cdot \frac{P \cdot R}{P + R}$
ROC-AUC	Classifier's ability to discriminate classes	Area under ROC curve

4. Gradient Descent Algorithm – In Detail

✅ What is Gradient Descent?

Gradient Descent is an optimization algorithm used to **minimize a cost (loss) function** by iteratively moving in the direction of the **steepest descent**, as defined by the **negative of the gradient**.

It's widely used to train **machine learning models**, especially in **linear regression, logistic regression, and neural networks**.

🧠 Intuition:

- Imagine standing on a hill in the fog.
- You want to reach the lowest point (minimum cost).
- You take steps downhill, always moving in the direction of the **steepest slope**.

📚 Gradient Descent Algorithm

Let:

- θ : model parameters (weights)
- $J(\theta)$: cost function

- α : learning rate (step size)
- $\nabla J(\theta)$: gradient of the cost function

Update Rule:

$$\theta := \theta - \alpha \cdot \nabla J(\theta)$$

Repeat until convergence (when change in $J(\theta)$ is small).

Example – Linear Regression:

Cost function (Mean Squared Error):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x_i) - y_i)^2$$

Gradient:

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x_i) - y_i) \cdot x_{ij}$$

Update each parameter using the gradient descent rule.

Types of Gradient Descent

1. Batch Gradient Descent

- Uses **entire training dataset** to compute the gradient.
- Update happens **once per epoch**.

Pros:

- Converges smoothly
- Accurate gradient

Cons:

- Slow for large datasets

2. Stochastic Gradient Descent (SGD)

- Uses **one random data point** at a time to update weights.

Pros:

- Fast and memory efficient
- Good for online learning

Cons:

- Noisy updates → may oscillate

3. Mini-Batch Gradient Descent

- Uses a **small random subset (mini-batch)** of data (e.g., 32 or 64 samples).
- Combines benefits of both batch and SGD.

Pros:

- Faster than batch
- Less noisy than SGD
- Ideal for GPU/parallel processing

Comparison Table

Type	Data per Step	Speed	Convergence Stability	Usage
Batch	All data	Slow	Stable	Small datasets
SGD	Single sample	Fast	Noisy	Real-time, large
Mini-Batch	Small subset	Moderate	Balanced	Most common

Enhancements to Gradient Descent

- **Momentum:** Helps accelerate SGD by considering past gradients.
- **Adam** (Adaptive Moment Estimation): Combines momentum and adaptive learning rates.
- **RMSProp:** Adapts learning rate per parameter.

Summary

Gradient Descent is the backbone of most machine learning model training. Choosing the right variant and tuning parameters like the learning rate are key to fast and reliable convergence.

Would you like a plotted visualization of the descent path or comparison of these types in code?

5. K-Nearest Neighbors (KNN) Algorithm – In Detail

What is KNN?

K-Nearest Neighbors (KNN) is a **supervised learning** algorithm used for both **classification** and **regression** tasks. It is an **instance-based** or **lazy learning** algorithm, meaning it does **not learn an explicit model** during training. Instead, it stores the training data and makes predictions **based on the majority class (or average value)** of the **K nearest neighbors** during inference.

How KNN Works (Classification)

To classify a new point:

1. **Calculate distance** (e.g., Euclidean) between the new data point and all points in the training dataset.
 2. **Select the K closest points.**
 3. **Vote for the most frequent class** among the K neighbors.
 4. Assign that class to the new point.
-

Common Distance Metrics

For two points $x=(x_1,x_2,...,x_n)$ and $y=(y_1,y_2,...,y_n)$:

- **Euclidean Distance:**

$$d(x,y)=\sqrt{\sum_{i=1}^n(x_i-y_i)^2}$$

- **Manhattan Distance:**

$$d(x,y)=\sum_{i=1}^n|x_i-y_i|$$

- **Minkowski Distance** (general form):

$$d(x,y)=\left(\sum_{i=1}^n|x_i-y_i|^p\right)^{1/p}$$

Example:

Let's say you have the following training data for classifying whether a fruit is an apple (A) or an orange (O):

Weight (g) Diameter (cm) Class

150	7	Apple
170	8	Apple
140	7	Apple
130	6	Orange
160	9	Orange

To classify a new fruit with weight = 155g, diameter = 7.5cm:

1. Compute distance to all points.
2. Pick top **K=3** neighbors.
3. Suppose they are: Apple, Apple, Orange → Majority = Apple → Classify as Apple.

Mathematical Formulation (Classification)

Given:

- Training set: $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
- Query point: x_q
- Distance function: $d(x_i, x_q)$

Steps:

1. Compute distances $d(x_i, x_q)$
2. Select **K** points with the smallest distances
3. Predict:

$\hat{y}_q = \text{majority_vote}(y_i \text{ for } x_i \in K \text{ nearest neighbors})$

Choice of K – Its Effect

K Value	Behavior	Pros	Cons
Small (e.g., $K=1$)	More sensitive to noise	Captures local patterns	High variance, may overfit
Large (e.g., $K=n$)	Smoother decision boundary	More stable	May underfit, ignores local structure

- **Odd K** values are often used in binary classification to avoid ties.
- Usually chosen via **cross-validation**.

Bias-Variance Trade-off

- **Low K (e.g., 1):** Low bias, **high variance** → model fits training data closely.
- **High K:** High bias, **low variance** → smoother boundaries, less sensitive to noise.

Practical Notes

- KNN is **non-parametric**: No assumptions about data distribution.

- **Lazy learning:** No training phase, but slow at prediction time.
- Works best with **low-dimensional, normalized data**.
- Can be extended to **weighted KNN**: closer neighbors have more influence.

✅ Summary

Feature	Description
Algorithm Type	Supervised, instance-based
Prediction	Based on K closest training samples
Distance Metric	Typically Euclidean or Manhattan
Key Hyperparameter K (number of neighbors)	
Trade-off	Small K → overfitting; Large K → underfitting
