Machine Learning

Machine Learning is a discipline in computer science where we train machines on data so that they can make predictions without explicit programming

In traditional programming we have built logic and we give input, and it predicts the output

Logic, input Traditional software program output

In Machine Learning we give input and output build a logic so when you give input and output it should predict whether the output of that input is correct or not [model]

Input, output → ML Training → Logic[model]

Statistical Machine Learning

- **Supervised Learning** *(Training with labelled data)*:

  Supervised learning is a method in which the model is trained on a labelled dataset such as house price prediction.

  - **Regression** *(Predict continuous values)*:
    - Linear Regression
    - Multilinear Regression
    - Polynomial Regression
  - **Classification** *(Predict categories)*:
    - Binary Classification
    - Multiclass Classification
- **Unsupervised Learning** *(Training with unlabelled data)*:

  In Unsupervised Learning, we provide an unlabelled dataset to an ML program and then it learns to identify patterns and structures in data without any explicit guidance.

  - Clustering (e.g., K-Means, Hierarchical)
  - Dimensionality Reduction (e.g., PCA, t-SNE)

Supervised Learning Linear Regression

- **Coefficient = Slope**
  Yes! The coefficient in a linear regression equation represents the slope of the line. It tells you how much the Y value changes for a one-unit change in **X.**
- **Intercept** = **Where the line meets the Y-axis**
  The intercept is the point where the line crosses the Y-axis (i.e., when X = 0).
  It's not where the line meets both axes—just the Y-axis.

The intercept determines where the line starts on the Y-axis — that is, the value of Y when X = 0.

In the equation:

$$Y = mX + b$$

- b is the intercept.
- m is responsible for the angle (theta)
- m is coefficient
- m is weights in Deep Learning
- b is bias in deep learning
- It tells you the starting point of the line on the Y-axis.
- Visually, it's where the line crosses the Y-axis.

🔹 **1. Linear Models**

- Linear Regression: Models a straight-line relationship between input and output.
- Ridge Regression (L2 Regularization): Penalizes large coefficients to reduce overfitting.
- Lasso Regression (L1 Regularization): Can shrink some coefficients to zero (feature selection).
- Elastic Net: Combines L1 and L2 regularization.

🔹 **2. Polynomial Regression**

- Extends linear regression by adding polynomial terms (e.g., x^2, x^3) to capture non-linear relationships.

🔹 **3. Tree-Based Models**

- Decision Tree Regression: Splits data into regions and fits a constant in each.
- Random Forest Regression: Ensemble of decision trees; reduces overfitting.
- Gradient Boosting Regression (e.g., XGBoost, LightGBM): Builds trees sequentially to correct previous errors.

🔹 **4. Support Vector Regression (SVR)**

- Uses the principles of SVM to fit a function within a margin of tolerance.

🔹 **5. K-Nearest Neighbours Regression (KNN)**

- Predicts the output by averaging the values of the k-nearest data points.

🔹 **6. Neural Network Regression**

- Uses deep learning models (e.g., MLPs) to capture complex, non-linear relationships.

Simple Linear Regression

Purpose:

- To model the relationship between a dependent variable (target) and one independent variable (predictor).
- The goal is to find the best-fit line that minimizes the error between predicted and actual values.
- you change the coefficients the line will change

Equation of the Line:

$$Y = mX + b$$

- Y: Dependent variable (what you're trying to predict)
- X: Independent variable (input feature)
- M: Slope of the line (rate of change)
- b: Intercept (value of Y when X = 0)
- slopes == coefficients == weights responsible for the rotation of the line
- bias == intercept responsible for the lift the line

Interpretation:

- The slope M tells you how much Y changes for a unit change in X.
- The intercept b is the starting value of Y when X = 0.

  Dependent variable = Coefficient*independent variable + intercept

  Price = coefficient(m1)*price + intercept(b)

Multiple Linear Regression:

$$Y = b_1X_1 + b_2X_2 + \ldots + b_kX_k + b$$

Where:

- Y: Dependent variable (target/output)
- $X_1, X_2, \ldots, X_k$: Independent variables (features/input)
- $b_1, b_2, \ldots b_k$: Coefficients (slopes for each independent variable)
- b: Intercept (value of Y when all X's are 0)

Interpretation:

- Each coefficient $\mathbf{b_i}$ represents the **change in Y** for a **unit change in $\mathbf{X_i}$**, assuming all other variables are held constant.
- The intercept **b** is the baseline value of **Y** when all $\mathbf{X_i = 0}$.
- Dependent variable = Coefficient*independent variable 1+ Coefficient*independent variable 2…+intercept

Goal:

Identify the coefficients and intercept to predict the output

Find the best fit (it should either data points are around it or on it)

Example:

If you're predicting house prices:

Price=50×Area+10000×Bedrooms+20000 Price=50×Area+10000×Bedrooms+20000

- **Area** and **Bedrooms** are independent variables.
- **50** and **10000** are their respective coefficients.
- **20000** is the intercept.

📉 Cost Function (Error/Loss Function)

The **cost function** measures how well your model's predictions match the actual data. It tells you how far off your predicted values are from the actual values.

Actual value - predicted value = error

Errors are also called as Residual

Cost vs loss

Cost function refers to entire error, total error

Loss function = error only in one observation

Cost function = sse (sum of square error), rss (residual sum of squares)

🔹 **Basic Error (Residual):**

$$Error = \hat{y} - y$$

- $\hat{y}$: Predicted value from the model
- **y**: Actual value from the data

This is the **difference** between what your model predicts and what actually happened.

OLE = Ordinary Least Squares Estimation

MLE = Most Likelihood Estimation

🔹 **Cost Function in Linear Regression:**

Mean Absolute Error (MAE)

$$\frac{1}{n}\sum_{i=0}^{n} |y_i - \hat{y}_i|$$

Mean Squared Error (MSE)

$$\frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2$$

Root Mean squared error

$$\sqrt{\frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2}$$

Usually, we use the **Mean Squared Error (MSE)** as the cost function:

- It **squares** the errors so that negative and positive errors don't cancel out.
- Then it **averages** them over all data points.

Objective: To reduce the Error (how by finding the good coefficients)

Find the good coefficients to reduce the cost function

Derivatives and Partial Derivatives:

Derivative (Single-variable calculus)

- A derivative measures how a function changes as its input changes.
- It tells you the slope or rate of change of a function at a specific point.
- If you want to find a minimum point of a function
- First differentiation of function equal to zero
- $Y = x^2 \ = parabola$
- $x^n = nx^{n-1}$

Example:

If $f(x)=x^2$ then the derivative is $f'(x)=2x$

This means at any point *x*, the slope of the curve is 2x.

If: $f(x)=x^3$ Then the **derivative** is: $f'(x)= 3x^2$

🔍 **What does this mean?**

- For **every small change in *x***, the change in *y* (i.e., the slope of the curve) is approximately $\mathbf{3x^2}$.
- So, at:
    - x=1*x*=1, the slope is 3(1)2=3

- x=2x=2, the slope is 3(2)2=12
- x=3x=3, the slope is 3(3)2=27

This tells you how **steep** the curve is at any point.

Partial Derivative (Multivariable calculus)

- A partial derivative is used when a function has more than one variable.
- It measures how the function changes with respect to one variable, while keeping the others constant.

Example:

If f(x, y)=x^2+y^2 then Partial derivative with respect to x $\partial x/\partial f$=2x

Partial derivative with respect to y: $\partial y/\partial f$=2y

$$Cost(m,l) = 3 * m2 + 2 * l2 + 5 * m * l + 23$$

$$\partial cost/\partial m = 6 * m + 5 * l$$

$$\partial cost/\partial l = 4 * l + 5 * m$$

✅ **Here's the key idea:**

When you're taking the partial derivative with respect to m:

- You treat l as a constant.
- So, you only differentiate terms that contain m.
- Any term that does not contain m is treated as a constant and its derivative is 0.

🧠 **Why are they important in ML?**

- Derivatives help in optimization — finding the minimum of a cost function.
- Partial derivatives are used in gradient descent when dealing with functions of multiple variables (like weights in neural networks).

Chain Rule:

The chain rule is a fundamental concept in calculus used to compute the derivative of a composite function — that is, a function inside another function.

🔗 **Chain Rule Formula**

If you have a function: $y = f(g(x))$

Then the derivative is: $\frac{\partial x}{\partial y} = f'(g(x)) \cdot g'(x)$

🧠 **Intuition:**

You're **chaining** the derivatives together:

- First, take the derivative of the **outer function** (with the inner function still inside).
- Then multiply it by the derivative of the **inner function**.

🔍 **Example:**

$$Let's\ say: y = (3x + 2)4y = (3x + 2)4$$

This is a composite function:

- **Outer function**: $f(u) = u4$
- **Inner function**: g(x) = 3x + 2$g(x)$= 3x + 2
- Using the chain rule: $\frac{dy}{dx} = 4(3x + 2)^3 \cdot 3 = 12(3x + 2)3$

📘 **Gradient Descent: Theory Explained**

A trial-and-error method to find the optimal value for m(slope) and b(intercept)

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

The purpose of partial derivate is to measure how a function changes as of its variable is varied while keeping the other variable constant.

$$mse = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(m_{x_i} + b\right)\right)^2$$

How function mse changing with respect to m

$$\partial mse/\partial m = \frac{2}{n}\sum_{i=1}^{n} -x_i\left(y_i - \left(m_{x_i} + b\right)\right)$$

How function mse changing with respect to b

$$\partial mse/\partial b = \frac{2}{n}\sum_{i=1}^{n} -\left(y_i - \left(m_{x_i} + b\right)\right)$$

For $\partial mse/\partial b$ we will have an initial value b1 $b2 = b1 - \frac{\partial mse}{\partial b} * learining\ rate(alpha)$

Learning rate is direction

🔍 **Why Use Gradient Descent?**

In machine learning, we often want to find the **best parameters** (like weights in a model) that **minimize the error** (cost function). Gradient descent helps us do that.

Why MSE (and not MAE)?

⚖️ **Why MSE is Often Preferred:**

Feature	MSE	MAE
Penalizes large errors	✅ Yes (squares them)	❌ No (treats all errors equally)
Smooth gradient	✅ Differentiable everywhere	⚠️ Not differentiable at 0
Used in gradient descent	✅ Easier to optimize	⚠️ Harder due to non-smoothness
Sensitive to outliers	✅ Yes	❌ Less sensitive

🧠 **Key Insight:**

- **MSE is more sensitive to large errors**, which is useful when you want your model to **strongly penalize big mistakes**.
- **MAE is more robust to outliers**, but its **gradient is not smooth**, which can make optimization harder.

🧪 **When to Use MAE Instead?**

- When your data has **many outliers**.
- When you want **equal treatment** of all errors.
- In some **robust regression** techniques.

Model Evaluation: Metrics

📊 **1. Regression Metrics**

Used when the output is a **continuous value**.

Metric	Description
MSE (Mean Squared Error)	Average of squared differences between predicted and actual values. Penalizes large errors.
RMSE (Root Mean Squared Error)	Square root of MSE. Same units as the target variable.

Metric	Description
MAE (Mean Absolute Error)	Average of absolute differences between predicted and actual values. Less sensitive to outliers.
R^2 Score (Coefficient of Determination)	Measures how well the model explains the variance in the data. Ranges from 0 to 1 (higher is better).

📐 **R² Score (Coefficient of Determination)**

The R² score is a metric used to evaluate how well a regression model fits the data. It tells you the proportion of the variance in the dependent variable that is predictable from the independent variables.

🧮 **Formula:**

$$R^2 = 1 - \frac{\sum_{i=0}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n}(y_i - \bar{y}_i)^2}$$

$$R^2 = 1 - \frac{SSR}{SST}$$

SSR = Residual Sum of Squares (error) (square the residual and sum all of them)

SST = Total Sum of Squares (variance)

Residual = error of each observation (actual value – predicted value)

$\bar{y} = average\ of\ the\ actual\ values$

$R^2\ this\ much\ variation$ in output can explained by the input

📊 **SST (Total Sum of Squares)**

- Measures the **total variance** in the actual data.
- It compares each actual value to the **mean** of the actual values.
- Think of it as: *"How spread out is the actual data from the average?"*

📉 **SSR (Residual Sum of Squares)**

- Measures the **unexplained variance** — the part of the data the model **didn't** capture.
- It compares each actual value to the **predicted** value.
- Think of it as: *"How far off are the predictions from the actual values?"*

🔁 **How They Relate to R²:**

- If SSR is **small**, the model fits well → **R² is close to 1- over fitting**
- If SSR is **large**, the model fits poorly → **R² is close to 0 or negative**
- r-square value keep increase if you add more features (variable)

🧠 **Choosing the Right Metric:**

- **Imbalanced classification?** → Use **F1 Score**, **Precision**, **Recall**, or **ROC-AUC**
- **Regression with outliers?** → Use **MAE**
- **General regression?** → Use **RMSE** or **R^2**
- **Balanced classification?** → Use **Accuracy**

📈 **Polynomial Regression — Explained Simply**

Polynomial Regression is a type of regression analysis where the relationship between the independent variable *x* and the dependent variable *y* is modelled as an **nth-degree polynomial**.

Decide the degree based of the problem or by domain knowledge

🔍 **Why Use It?**

- When the data shows a **non-linear** relationship.
- Linear regression can't capture curves — polynomial regression can!

🧮 **Equation**

For degree 2:

$$y = b_0 + b_1 x + b_2 x^2 \ \ldots$$

For degree 3:

$$y = b_0 + b_1 x + b_2 x^2 + b_3 x^3 \ \ldots.$$

- b0,b1,b2,...: coefficients
- *x*: independent variable
- y: predicted value

📊 **How It Works:**

- You **transform** the original features into polynomial features (e.g.,x^2, x^3, x^4 etc.).
- Then apply **linear regression** on these transformed features.

✅ **Pros:**

- Captures **non-linear trends**.
- Still relatively simple to implement.

⚠️ **Cons:**

- Can **overfit** if the degree is too high.
- Sensitive to **outliers**.

🔍 **How to Identify a Polynomial Problem**

1. Visual Inspection (Plot the Data)

- Plot the data points (scatter plot).
- If the data forms a **curve** (not a straight line), it may be polynomial.
    - **U-shape or inverted U** → quadratic (degree 2)
    - **S-shape or wave** → cubic or higher

2. Check Residuals from Linear Regression

- Fit a **linear regression model**.
- Plot the **residuals** (errors).
    - If residuals show a **pattern** (like a curve), linear regression is not sufficient.
    - If residuals are **randomly scattered**, linear might be fine.

3. Try Polynomial Fits

- Fit models of increasing degree (e.g., degree 2, 3, 4).
- Compare metrics like:
    - **R^2 score** (higher is better)
    - **MSE** or **RMSE** (lower is better)
- Use **cross-validation** to avoid overfitting.

4. Domain Knowledge

- In physics, economics, or biology, some relationships are **naturally polynomial** (e.g., projectile motion, supply-demand curves).

🔁 **Multicollinearity — Explained Simply**: occurs when two or more independent variables are highly correlated, making it difficult to distinguish the individual effects on the dependent variable.

🧠 **Why is it a problem?**

- It becomes **hard to determine** the individual effect of each variable on the dependent variable.
- Coefficients can become **unstable** and **very sensitive** to small changes in the data.
- It can lead to **misleading interpretations** and **inflated standard errors**.

📉 **Example:**

Suppose you're predicting house prices using:

- Area in square feet
- Number of rooms
- Total built-up area

If Area and Built-up area are highly correlated, the model might struggle to decide **which one is actually influencing the price**.

🔍 **How to Detect Multicollinearity:**

1. **Correlation Matrix**: Look for high correlations between independent variables.
2. **Variance Inflation Factor (VIF)**:
    - VIF > 5 or 10 → potential multicollinearity.

🛠️ **How to Handle It:**

- **Remove** one of the correlated variables.
- **Combine** correlated variables into a single feature (e.g., using PCA).
- Use **regularization techniques** like Ridge or Lasso regression.

🔍 **1. Underfitting**

What it is:
The model is too simple to capture the underlying pattern in the data.

If your model is not trained very well on train data train error will be high(underfit)

Symptoms:

- High error on training and test data
- Poor performance overall

Causes:

- Model is not complex enough (e.g., linear model for curved data)
- Too few features
- Too much regularization
- Poor feature selection
- Inadequate validation
- Less epochs

Remedies:

- Use a more complex model (e.g., polynomial regression, deeper neural network)
- Add more relevant features
- Reduce regularization (e.g., lower L1/L2 penalty)

🔍 2. Overfitting

What it is:
The model is too complex and learns the noise in the training data instead of the actual pattern.

Focus too much on training(memorizing) rather than generalize

Model well trained on train data

Symptoms:

- Low error on training data
- High error on test data (poor generalization)

Causes:

- Model is too flexible or deep
- Too many features
- Not enough training data
- Poor model and Hyperparametere selection
- Poor feature selection
- Inadequate validation
- Apply regularization

Remedies:

- Use simpler models
- Apply regularization (L1, L2)
- Use cross-validation
- Prune decision trees or reduce neural network layers
- Collect more data
- Use dropout (in neural networks

  Assume your train error is low

  1. Test error might be low train error low and test error low ➔ Normal fit
  2. Test error might be high train error low and test error high ➔ overfit
  3. Trained very well on the train data train error will be high ➔ underfit

🧱 L1 and L2 Regularization — Explained Simply

Regularization is a technique used to prevent **overfitting** by adding a penalty to the loss function in machine learning models.

🔹 L1 Regularization (Lasso)

- Adds the **absolute value** of coefficients to the loss function:

$$Loss = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{n} | w_i |$$

Wi = weights, $\lambda = alpha$

Encourages **sparsity** pushes some weights to **exactly zero**.

- Useful for **feature selection**.

✅ **Pros**:

- Can eliminate irrelevant features
- Produces simpler, more interpretable models

🔹 **L2 Regularization (Ridge)**

- Adds the **squared value** of coefficients to the loss function:

$$Loss = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{n} w_i^2$$

- Encourages **small weights**, but rarely zero.
- Helps when features are **correlated**.

✅ **Pros**:

- Keeps all features but reduces their impact
- More stable and smooth optimization

🔄 Comparison Table

Feature	L1 (Lasso)	L2 (Ridge)
Penalty	Sum of absolute values	Sum of squared values
Feature selection	✅ Yes	❌ No
Coefficients	Can be zero	Shrinks toward zero
Use case	Sparse models	Multicollinearity

When to Use which?

L1 Regularization:

use when you need to feature selection, model interpretability, or expect only a few features to be relevant.

L2 Regularization:

Use when you want to keep all features, multicollinearity, or need a stable solution

Elastic Net:

Use when you want the benefits of both L1 and L2 especially with highly correlated features

⚖️ **Bias-Variance Trade-Off — A Core Concept in Machine Learning**

The **bias-variance trade-off** is about finding the right balance between **underfitting** and **overfitting** in your model.

Under fit ➔ train error is high ➔ high bias – low variance

Normal fit ➔ train error low and test error low ➔ low bias – low variance

Over fit ➔ train error low and test error high ➔ low bias – high variance

🔍 **Bias**

A measurement of how accurately a model can capture a pattern in a training dataset. If error is big the bias is high.

Red line bias curve ==bias^2 ➔ Train Error

Explains the about the how your model complexity increasing by more variables

- Error due to **wrong assumptions** in the learning algorithm.
- High bias → model is **too simple** → **underfitting**.
- Example: Using a linear model for curved data.
- BUVO = Bias Underfit Variance Overfit

🔍 **Variance**

- Error due to **model sensitivity** to small fluctuations in the training data.
- High variance → model is **too complex** → **overfitting**.
- Example: A high-degree polynomial that fits every training point perfectly.
- High variance: Train Error = low, Test Error = High
- Low variance: Train, Test Error = Low
- Green line ➔ variance curve ➔ Test error

📉 **Total Error = Bias² + Variance + Irreducible Error**

- **Goal**: Minimize total error by balancing bias and variance.
- You can't reduce both at the same time — improving one often worsens the other.

📊 **Visual Intuition:**

- **High Bias**: Predictions are consistently wrong in the same way.
- **High Variance**: Predictions vary wildly depending on the training data.
- **Just Right**: Low bias and low variance — the sweet spot! Balanced
- **Linear Regression is over fit when you use more variables model complexity**

Supervised Learning Classification:

is a type of machine learning where the model is trained on a labelled dataset, meaning that each training example is paired with an output label. The goal is for the model to learn a mapping from inputs to outputs so it can predict the label of new, unseen data.

Binary Classification

Multi class Classification

Key Concepts:

1. **Labeled Data**: Each input has a corresponding correct output (label).
2. **Training Phase**: The model learns from the labeled data.
3. **Testing Phase**: The model is evaluated on new data to assess its performance.

Common Algorithms:

- **Logistic Regression**
- **Decision Trees**
- **Random Forest**
- **Support Vector Machines (SVM)**
- **K-Nearest Neighbours (KNN)**
- **Naive Bayes**

Logistic Regression for Binary Classification

is a fundamental algorithm used when the target variable has two possible outcomes (e.g., 0 or 1, Yes or No, True or False).

🔍 **How It Works:**

Logistic Regression models the **probability** that a given input belongs to a particular class using the **logistic (sigmoid) function**:

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

Where:

- $Z(weights) = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$
- $x_1,\ x_2, x_3 \ldots.\ x_n = features$
- $e = Euler's\ number \sim 2.71828$
- $\sigma(z)$ outputs a value between 0 and 1, interpreted as a probability.

$$z = ßo + ß1 * salary$$

$$z = ßo + ß1x1$$

$$y = \frac{1}{1 + e^{-z}}$$

$$y = \frac{1}{1 + e^{-(ßo + ß1x1)}}$$

$$P(y = 1|x)$$

$$ßo = intercept$$

$$ß1 = coefficient$$

What is the probability that the person will buy the car given the x(salary)

✅ **Binary Classification Example:**

Let's say we want to predict whether a student passes (1) or fails (0) based on the number of hours studied.

Steps:

1. **Input Features**: Hours studied
2. **Output Label**: Pass (1) or Fail (0)
3. **Model**: Logistic Regression
4. **Decision Rule**: If predicted probability > 0.5 → Class 1 (Pass), else Class 0 (Fail)

Logistic Regression: Multiclass Classification:

In logistic regression, the cost function used is called Log Loss (also known as **Binary Cross-Entropy** Loss for binary classification or Categorical Cross-Entropy for multiclass).

Here you will have both global minima and local minima

Convex function U shaped curve, non-convex function

Cost function(mse) with logistic regression is non convex function

A **non-convex function** is a type of mathematical function where the line segment between two points on the graph **does not always lie above or on the graph** of the function. In simpler terms, it can have **multiple local minima and maxima**, making optimization more challenging.

📉 **Log Loss Formula (Binary Classification)**

For a single training example:

$$Log\ Loss\ = -\left[y.log(\hat{y}) + (1-y).log(1-\hat{y})\right]$$

$$When\ y\ =\ 1\ \ log\ loss\ = -log(p)$$

$$When\ y\ =\ 0\ \ \ log\ loss\ = -log(1-p)$$

Where:

Log loss aka binary cross entropy

· y is the true label (0 or 1)

· ŷ is the predicted probability of class 1

🧮 **For the Whole Dataset (Average over all samples):**

$$loss\ = -\frac{1}{n}\sum_{i=1}^{n}[y_i log\ (\hat{y}_i) + (1-y_i)log(1-\hat{y}_i)]$$

Where:

- m is the number of training examples
- $\hat{y}_i$ is the predicted probability for the i^{th} example
- Log loss aka multinominal cross entropy
- $y_i\ actual\ value$

✅ **Why Log Loss?**

- It penalizes **wrong confident predictions** more heavily.
- It is **convex**, which helps in optimization.

It works well with **probabilistic outputs**

Model Evaluation: Accuracy, Precision and Recall

🧮 **Classification Metrics**

True / False → Was the prediction correct?

Positive / Negative → What was the predicted class?

1. Confusion Matrix

False positives are actual Negatives

False Negatives are actual positives

A table that compares actual vs. predicted labels.

- Rows → Actual values
- Columns → Predicted values

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

- TP: Model says *Yes*, Actual is *Yes* ✅
- TN: Model says *No*, Actual is *No* ✅
- FP: Model says *Yes*, Actual is *No* ❌ (Type I error)
- FN: Model says *No*, Actual is *Yes* ❌ (Type II error)

Total observations = TP + TN + FP + FN

📌 **1. Accuracy**

- Definition: The ratio of correctly predicted observations (Positive and negative) to the total observations.
- Correctly predicted / all values in predictions
- Formula:

$$Accuracy = \frac{TP}{Total\ predictons}$$

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

✅ **Example:**

Suppose we have 100 predictions:

- TP = 40, TN = 50, FP = 5, FN = 5

$$Accuracy = \frac{40 + 50}{100} = \frac{90}{100} = 90\%$$

Good for: Balanced datasets.

Use **Precision, Recall, F1, ROC-AUC** when classes are imbalanced.

📌 **Error Rate:**

$$\boldsymbol{Error\ rate = 1 - Accuracy}$$

$1 - 90 = \ 10\%$ is the error rate of the model

📌 **Sensitivity (Recall, True Positive Rate – TPR):**
Out of **all actual positive**s, how many did the model **correctly predict as positive**?

$$Sentitvity = \frac{TP}{TP+FN}$$

$$\frac{TP(All\ correctly\ predicted\ positves)}{TP(All\ correctly\ predicted\ positves) + FN(predicted\ Negative\ but\ actually\ Positive) over\ all\ positives}$$

- **Focuses on detecting positives**.
- **Answers: *"If something is actually Positive, how often does my model catch it?"***
- **Sensitivity = Sick detected**

✅ Example: In disease detection → Sensitivity = proportion of sick patients correctly identified as sick.

📌 **Specificity (True Negative Rate – TNR)**
Out of all **actual negatives**, how many did the model **correctly predict as negative**?

$$Specificity = \frac{TN}{TN + FP}$$

$$\frac{TN(All\ correctly\ predicted\ Negatives)}{TN(All\ correctly\ predicted\ Negatives) + FP(predicted\ Positive\ but\ actually\ Negative) over\ all\ Negatives}$$

- Focuses on detecting **negatives**.
- Answers: *"If something is actually Negative, how often does my model correctly say it's Negative?"*

✅ Example: In disease detection → Specificity = proportion of healthy patients correctly identified as healthy.

Specificity = Safe detected

📌 **Precision (Positive Predictive Value – PPV)**
Out of all the predicted positives, how many are actually positive?

$$\boldsymbol{Precision = \frac{Tp}{Tp + Fp}}$$

- Answers: *"When my model says Positive, how often is it correct?"*
- Focuses on avoiding False Positives (FP).
- **Precision** = *"How right am I when I say Positive?"*

✅ Example: In spam detection, Precision means: *Of all the emails marked as spam, how many really were spam?*

📌 **Recall (Sensitivity, True Positive Rate – TPR)**
Out of all the actual positives, how many did the model correctly predict as positive?

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

- Answers: *"Of all the actual Positives, how many did I catch?"*
- Focuses on avoiding False Negatives (FN).
- **Recall** = *"How good am I at finding Positives?"*

✅ Example: In medical diagnosis, recall means: *Of all sick patients, how many did we correctly identify as sick*

📌 **F1 score**

Harmonic mean of precision and recall. Useful when classes are imbalanced.

$$F1\ Score\ = 2 * \frac{Precision * Recall}{Precision + Recall}$$

📌 **ROC Curve**

ROC (Receiver Operating Characteristic) curve is a plot between **True Positive Rate (TPR)** and **False Positive Rate (FPR)** at different classification thresholds.
It helps visualize how well a classifier can separate positive from negative classes.

- **X-axis:** False Positive Rate (FPR)
- **Y-axis:** True Positive Rate (TPR)

Formulas:

$$\boldsymbol{TPR}\ (\boldsymbol{Sensitivity},\ \boldsymbol{Recall}) = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}} = \frac{\boldsymbol{TP}}{\boldsymbol{P}}$$

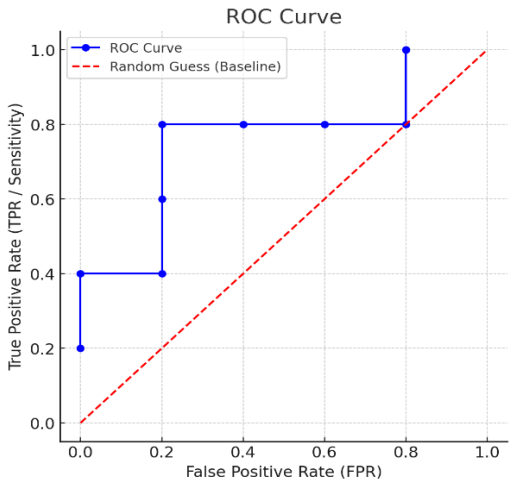
$$\boldsymbol{FPR} = \frac{\boldsymbol{FP}}{\boldsymbol{FP} + \boldsymbol{TN}} = \frac{\boldsymbol{FP}}{\boldsymbol{N}} = \boldsymbol{1 - Specificity}$$

📌 **AUC (Area Under the Curve)**
AUC measures the area under the ROC curve.

- AUC = 1 → Perfect model
- AUC = 0.5 → Random guess (baseline)
- Higher AUC means better classifier.

Record	Class	Prob	Model	TP	FP	TPR = TP/P (P=5)	FPR = FP/N (N=5)	Point (FPR,TPR)
1	P	0.90	P	1	0	1/5 = 0.2	0/5 = 0.0	(0.0, 0.2)
2	P	0.80	P	2	0	2/5 = 0.4	0/5 = 0.0	(0.0, 0.4)

Record	Class	Prob	Model	TP	FP	TPR = TP/P (P=5)	FPR = FP/N (N=5)	Point (FPR,TPR)
3	N	0.70	P	2	1	2/5 = 0.4	1/5 = 0.2	(0.2, 0.4)
4	P	0.60	P	3	1	3/5 = 0.6	1/5 = 0.2	(0.2, 0.6)
5	P	0.55	P	4	1	4/5 = 0.8	1/5 = 0.2	(0.2, 0.8)
6	N	0.54	P	4	2	4/5 = 0.8	2/5 = 0.4	(0.4, 0.8)
7	N	0.53	P	4	3	4/5 = 0.8	3/5 = 0.6	(0.6, 0.8)
8	N	0.51	P	4	4	4/5 = 0.8	4/5 = 0.8	(0.8, 0.8)
9	P	0.50	P	5	4	5/5 = 1.0	4/5 = 0.8	(0.8, 1.0)
10	N	0.40	N	5	4	5/5 = 1.0	4/5 = 0.8	(0.8, 1.0)



Interpretation:

- Model is good if ROC curve goes steeply upward (high TPR, low FPR).
- The more the curve bends towards the top-left, the better the classifier.

🔁 **Trade-off: Precision vs. Recall**

- **High Precision, Low Recall**: Very few false positives, but may miss many actual positives.
- **High Recall, Low Precision**: Catches most positives, but includes more false positives.

🔍 **Interpretation:**

- **Precision = 1.00**: All predicted positives were correct (no false positives).
- **Recall = 0.93**: The model correctly identified 93% of actual positives.
- **F1 Score = 0.96**: A harmonic mean of precision and recall, indicating a strong balance.
- **Accuracy = 97%**: Overall, the model made very few mistakes.

🧠 **Choosing the Right Metric:**

- **Imbalanced classification?** → Use **F1 Score**, **Precision**, **Recall**, or **ROC-AUC**
- **Regression with outliers?** → Use **MAE**
- **General regression?** → Use **RMSE** or **R^2**
- **Balanced classification?** → Use **Accuracy**

Classification Metrics Summary

Metric	Formula	Use Case	Example
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	Overall correctness of model (when classes are balanced).	Out of 100 patients, model predicted 90 correctly → Accuracy = 90%.
Error Rate	$\frac{FP+FN}{TP+TN+FP+FN} = 1 - Accuracy$	How often model is wrong.	If accuracy = 90%, error rate = 10%.
Sensitivity (Recall / TPR)	$\frac{TP}{TP+FN}$	Medical testing: probability of detecting disease when it's present.	Out of 50 sick patients, 45 identified → Sensitivity = 90%.
Specificity (TNR)	$\frac{TN}{TN+FP}$	Spam detection: probability of detecting "not spam" correctly.	Out of 50 non-spam emails, 47 marked correctly → Specificity = 94%.
Precision (PPV)	$\frac{TP}{TP+FP}$	When cost of false positives is high (e.g., email spam filter).	Out of 60 emails marked as spam, 54 were actually spam → Precision = 90%.
Recall	Same as Sensitivity $= \frac{TP}{TP+FN}$	When cost of false negatives is high (e.g., cancer detection).	Same as Sensitivity example above.
F1 Score	$2 * \frac{Precision * Recall}{Precision + Recall}$	Balances precision and recall when both are important.	Precision = 0.9, Recall = 0.9 → F1 = 0.9.
TPR (True Positive Rate)	$\frac{TP}{TP+FN} = \frac{TP}{P}$	Used in ROC curve (y-axis).	If P=50, TP=45 → TPR=0.9.

Metric	Formula	Use Case	Example
FPR (False Positive Rate)	$\boldsymbol{FPR = \frac{FP}{FP + TN} = \frac{FP}{N} = 1 - Specificity}$	Used in ROC curve (x-axis).	If N=50, FP=3 → FPR=0.06.
ROC Curve	Plot of TPR vs FPR at different thresholds	Compares classifiers visually.	Steeper curve → better model.
AUC (Area Under ROC)	Area under ROC curve (0.5 = random, 1 = perfect)	Measures overall separability.	AUC = 0.95 means very good classifier.

🌳 Decision Tree

1. Introduction

- A Decision Tree is a supervised learning algorithm used for classification and regression.
- It works by recursively splitting the dataset into smaller subsets based on feature values, forming a tree-like structure of nodes and branches.

2. How It Works (Classification Example)

1. Start at the Root Node (all data).
2. Select the best feature to split using a criterion (Entropy, Information Gain, Gini).
3. Split dataset into subsets (branches).
4. Continue splitting recursively until:
    - All samples in a node are of the same class (pure node), OR
    - Stopping condition (max depth, min samples, etc.) is reached.
5. Final nodes = Leaves (predicted class or value).

3. Splitting Criteria

(a) Entropy (ID3 Algorithm)

Measures randomness/impurity in the data.

$$\boldsymbol{Entropy(S)} = \sum_{i=1}^{n} p_i log_2(p_i)$$

- p_i = proportion of class iii in set SSS
- Entropy = 0 → Pure (all one class)
- Entropy = 1 → Maximum impurity (equal mix)

👉 Example: Toss a fair coin

$$p(H) = 0.5, p(T) = 0.5$$

$$H = -[0.5 \log_2 0.5 + 0.5 \log_2 0.5] = 1$$

(b) Information Gain

Measures **reduction in entropy** after splitting on an attribute.

$$IG(T, A) = H(T) - H(T \mid A)$$

- $H(T)$: Entropy of dataset before split
- $H(T \mid A)$: Weighted entropy after splitting by attribute A
- Higher IG → Better feature to split.

Gain Ratio (C4.5)

$$GainRatio(A) = \frac{IG(T, A)}{SplitInfo(A)}$$

(c) Gini Impurity (CART Algorithm)

Alternative impurity measure.

$$Gini\ (s) = 1 - \sum_{i=1}^{n} p_i^2$$

- Lower Gini → Purer node.
- Used in CART (Classification and Regression Trees).

👉 Example: If 7 Yes and 3 No in a node:

$$Gini = 1 - \left(\left(\frac{10}{7}\right)^2 + \left(\frac{10}{3}\right)^2\right) = 0.42$$

4. Regression Trees

- Instead of entropy/Gini, use:
    - **MSE (Mean Squared Error)**
    - **MAE (Mean Absolute Error)**
- Goal: Minimize variance in target values within nodes.

5. Tree Complexity & Pruning

- **Depth** = how many levels in the tree.
- **Deep tree → Overfitting** (memorizes training data).
- **Shallow tree → Underfitting** (too simple).

👉 Techniques:

- Set max depth, min samples per split, or use **pruning** (remove weak splits after training).

6. Advantages ✅

- Easy to understand & visualize.
- No feature scaling required.
- Handles both categorical & numerical data.

7. Disadvantages ⚠️

- **Overfitting** (trees grow too deep).
- **Unstable**: small data changes → different tree.
- Greedy splitting may not find the global best tree.

8. Common Algorithms

- **ID3** → Entropy + Information Gain.
- **C4.5** → Entropy + Gain Ratio (handles categorical + continuous).
- **CART** → Gini Index (Classification) / MSE (Regression).

9. Quick Example (Root Node Selection)

Suppose we have feature **Age** with 3 categories: Youth, Middle-aged, Senior.

1. Compute entropy for each category:

$$H(Youth) = -p_{yes}\log_2\left(p_{yes}\right) - p_{no}\log_2(p_{no})$$

(Similar for Middle-aged, Senior).

2. Compute weighted entropy for Age:

$$H(Age) = w_{youth}H(Youth) + w_{middle}H(MA) + w_{senior}H(Senior)$$

3. Compute Info Gain:

$$IG(Age) = H(Total) - H(Age)$$

4. Repeat for all features → Feature with **highest IG** = Root Node.

10. Summary

- **Entropy** → randomness in data.
- **Information Gain** → reduction in entropy after split.
- **Gini** → impurity measure used in CART.
- **Decision Trees** can classify & regress but are prone to **overfitting** → solved by **Ensemble Methods** (Random Forest, Gradient Boosted Trees).

🌞 **Naïve Bayes**

1. Core Idea

Naïve Bayes is a **probabilistic classifier** based on **Bayes' Theorem**, with the strong (naïve) assumption that all features are **independent given the class**.

- Goal: Estimate the probability of a class given observed feature.
- Example: “What is the probability an email is spam given it contains the words *free* and *lottery*?”

2. 📜 Bayes’ Theorem

$$(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

$$P(diamond|queen) = \frac{P(queen|diamond) * P(diamond)}{P(queen)}$$

Where:

- P(C|X) Posterior probability of class *C* given features X.
- P(X|C): Likelihood of features given class.
- P(C): Prior probability of class
- P(X): Evidence (probability of features)

3. 🧮 Naïve Assumption

Assumes that all features$x_1, x_2, \ldots, x_n$ are conditionally independent given the class C:

$$P(X|C) = P(x_1|C).P(x_2|C) \ldots . P(x_n|C)$$

This simplifies computation and works surprisingly well in practice.

4. 📦 Types of Naïve Bayes

Type	Use Case	Assumption
Gaussian NB	Continuous features (e.g., height, weight, sensor data)	Features follow a normal distribution
Multinomial NB	Discrete counts (e.g., word counts in text)	Features are count-based
Bernoulli NB	Binary features (yes/no, present/absent)	Features are binary

Quick Guide:

- Text data (word counts) → **Multinomial**
- Text data (binary word presence) → **Bernoulli**
- Continuous numerical data → **Gaussian**

5. ✅ Advantages

- Simple and fast.
- Works well with **high-dimensional data** (e.g., text).

- Requires less training data.

⚠️ **Limitations**

- Assumes feature independence (rare in real life).
- Zero probabilities can occur if a feature never appears with a class → handled by **Laplacian correction**.

6. 🔧 Laplacian Correction

- Problem: If a feature never occurs with a class, its probability = 0, making the entire product 0.
- Solution: Add a small smoothing term (usually 1).
- What is the probability that an email is spam given that it contains the word free and lottery?
- $p(spam) = 30\%$
- $p(free\_lottery) = 10\%$
- $p(free \mid spam) = 40\%$
- $p(lottery \mid spam) = 60\%$
- $p(free\_lottery \mid spam) = 0.4 * 0.6 = 0.24 = 24\%$
- $p(spam|free\_lottery) = \frac{p(free\_lottery|spam) * p(spam)}{p(free\_lottery)}$
  - $= \frac{0.24*0.3}{0.1}$
- $= 0.72 = 72\%$

9. Summary

- **Assumption**: Features are conditionally independent given class.
- **Laplacian correction**: Adds smoothing to avoid zero probabilities.
- **Hyperparameters**: Mainly smoothing (alpha), variance smoothing, and binarization.
- **Best suited for**: Text classification, spam detection, document categorization.

🌼 **Support Vector Machine (SVM)**

Support Vector Machine (SVM) is a powerful supervised learning algorithm used for classification and regression, but it's most commonly used for binary classification.

📌 **Core Idea:**

SVM tries to find the best boundary (hyperplane) that separates data points of different classes with the maximum margin(distance between the nearby datapoints and best fit line (border).we call these margin points as support vectors

In 2D:

- The hyperplane is a line.
- SVM finds the line that maximizes the distance between the nearest points of each class (called support vectors**).**

🧱 **Key Concepts:**

- Hyperplane: A decision boundary that separates classes.
- Support Vectors: Data points closest to the hyperplane; they "support" the boundary.
- Margin: Distance between the hyperplane and the support vectors. SVM maximizes this.

🔧 **Types of SVM:**

1. Linear SVM: Works when data is linearly separable.
2. Non-linear SVM: Uses kernel tricks (like RBF, polynomial) to transform data into higher dimensions where it becomes linearly separable.

✅ **Pros:**

- Works well in high-dimensional spaces.
- Effective when the number of features > number of samples.
- Robust to overfitting (especially with proper regularization).

⚠️ **Cons:**

- Not ideal for very large datasets.
- Less effective when classes overlap significantly.
- Choosing the right kernel and parameters can be tricky.

🔍 **What is a Kernel in SVM?**

A **kernel** is a mathematical function that transforms the input data into a higher-dimensional space to make it easier to separate with a hyperplane.

Kernal determines lot of things like iterations it will cost resources like liner will take more iterations

🧠 **Common Kernels:**

1. **Linear Kernel**:
    - No transformation; works well when data is linearly separable.
    - Decision boundary is a straight line.
2. **Polynomial Kernel**:
    - Transforms data into a higher-degree polynomial space.
    - Can model more complex, curved boundaries.
3. **RBF (Radial Basis Function) Kernel**:
    - Also known as the **Gaussian kernel**.
    - Maps data into infinite-dimensional space.
    - Great for non-linear problems with complex boundaries.

🧪 **Summary:**

Kernel	Use Case	Boundary Shape
Linear	Linearly separable data	Straight line
Polynomial	Curved but structured boundaries	Polynomial curves
RBF	Highly non-linear, flexible data	Complex curves

🧠 **When to Use Which Kernel**

Kernel Type	Use When...	Pros	Cons
Linear	- Data is linearly separable - Features are already well-scaled and informative	- Fast and simple - Works well with high-dimensional sparse data (e.g., text)	- Can't model complex relationships
Polynomial	- You suspect interactions between features - You want to model curved boundaries	- Can capture non-linear patterns - Degree can be tuned	- Slower than linear - Risk of overfitting with high degrees
RBF (Gaussian)	- Data is not linearly separable - You want a flexible, general-purpose kernel	- Very powerful - Can model complex boundaries	- Requires tuning of gamma and C - Slower on large datasets
Sigmoid (less common)	- You want a neural network-like behaviour	- Similar to a 2-layer neural net	- Rarely outperforms other kernels

🔧 **Tips for Choosing:**

1. **Start with Linear**: It's fast and often surprisingly effective.
2. **Try RBF if Linear Fails**: Especially if your data is clearly non-linear.
3. **Use Polynomial if You Know Feature Interactions Matter**.
4. **Use Cross-Validation** to compare performance across kernels.

🔧 **1. Regularization Parameter (C)**

- **What it does**: Controls the **trade-off between maximizing the margin and minimizing classification error**.
- **Low C**:
    - Allows more margin violations (i.e., misclassifications).
    - Results in a **wider margin** but possibly more errors.
    - Good for **generalization**.
- **High C**:
    - Tries to classify all training examples correctly.
    - Results in a **narrower margin** and may **overfit**.

🧠 Think of **C** as:

"How much do I care about **classifying every training point correctly**?"

🔬 **2. Gamma (γ) — Only for RBF, Polynomial, and Sigmoid Kernels**

- **What it does**: Controls the **influence of a single training example**.
- **Low gamma**:
    - Points far away from the decision boundary are considered.
    - Results in **smoother, more generalized** decision boundaries.
- **High gamma**:
    - Only points close to the boundary are considered.
    - Results in **tight, complex boundaries** — can **overfit**.

🧠 Think of **gamma** as:

"How far does the influence of a single training point reach?"

🌟 **K-Nearest Neighbors (KNN)**

1. Core Idea

KNN is a **non-parametric, instance-based learning algorithm** used for **classification** (and regression).

- It does not build a model but classifies a new data point based on the **majority class of its nearest neighbors**.
- Closeness is measured using **distance metrics** (e.g., Euclidean distance).

2. 🧮 Mathematics Behind KNN

Distance Between Two Points (Euclidean Distance)

For two points $A(x_1, y_1)$ and $B(x_2, y_2)$

$d(A, B) = \sqrt{(x_2, x_1)\text{^}2\ \ (y_2, y_1)\text{^}2}$

- Smaller distance → stronger relation.
- Larger distance → weaker relation.

👉 Other distance measures can also be used (Manhattan, Minkowski, Cosine similarity).

3. 🔧 Steps in KNN

1. **Choose KKK** (number of neighbors, a hyperparameter).
2. **Calculate the distance** between the new point and all existing data points.
3. **Rank neighbors**: Sort distances from smallest to largest.
4. **Pick K neighbors** (closest ones).
5. **Assign label**:
    - For classification → majority vote among neighbors.
    - For regression → average of neighbors' values.

4. 🔧 Hyperparameters

- **K (number of neighbors)**
    - Small KKK (e.g., 1–2): Sensitive to noise, risk of **overfitting**.
    - Large KKK: Smoother decision boundary, risk of **underfitting**.
    - Rule of thumb: K≈NK \approx \sqrt{N}K≈N (where NNN = number of observations).
- **Distance Metric**: Euclidean, Manhattan, Minkowski, Cosine.
- **Weighting**:
    - Uniform (all neighbors equal).
    - Distance-based (closer neighbors have more influence).

5. 📌 Important Questions

❓ When does KNN overfit?

- When KKK is too small (e.g., K=1):
    - Decision boundary becomes too complex.
    - Model memorizes training data → **high variance**.

❓ How to choose KKK?

1. KKK should ideally be an **odd number** (to avoid ties in classification).
2. A common heuristic:

K≈N

Example: If dataset has 100 observations → K=100=10K = \sqrt{100} = 10K=100=10.

    - If 10 is even → choose 9 or 11 (prefer 11 for better generalization).
3. Use **cross-validation** to find the best KKK.

6. ✅ Advantages

- Simple and intuitive.
- No training phase → fast to implement.
- Works well for small datasets.

⚠️ Limitations

- Computationally expensive at prediction time (needs distance calculation with all points).
- Sensitive to irrelevant/noisy features and feature scaling.
- Struggles with high-dimensional data (curse of dimensionality).

7. Example

Suppose we have data in 2D space:

- "Yes" category points: (2, 3), (3, 4)
- "No" category points: (1, 1), (2, 0)
- A new point (2.5, 2.5) arrives.

Steps:

1. Compute distances from (2.5, 2.5) to all 4 points.
2. Rank them.
3. Choose K=3K=3K=3.
4. Among 3 nearest neighbors, if 2 are "Yes" and 1 is "No", new point → **Yes**.

8. Summary

- **KNN principle**: Assigns a class based on the majority label of the nearest neighbors.
- **Key hyperparameter**: KKK.
- **Overfitting** when KKK is too small; **underfitting** when KKK is too large.
- Distance metric choice matters.
- Best for **small to medium datasets** with meaningful distance representation.

🌟 Ensemble Learning

1. What is Ensemble Learning?

Ensemble learning is a machine learning technique that **combines multiple models** (weak learners or strong learners) to improve performance compared to individual models.

Key benefits:

- Reduces **variance** (avoids overfitting)
- Reduces **bias** (avoids underfitting)
- Improves accuracy, stability, and generalization

Common strategies:

- Majority voting (classification)
- Averaging (regression)
- Weighted averaging

2. Types of Ensemble Methods

🔹 **A. Bagging (Bootstrap Aggregating)**

- **How it works**:
    1. Create random subsets of the training dataset using **sampling with replacement**.
    2. Train a base model (e.g., Decision Tree) on each subset in **parallel**.
    3. Aggregate predictions → **majority vote** (classification) or **average** (regression).
- **OOB (Out-of-Bag) data**:
    - Around **63%** of data is used in training each bootstrap sample.
    - Remaining **37%** is not selected → used as **test set (OOB error)**.
    - Provides built-in model validation and helps avoid overfitting.
- **Example**: **Random Forest**
    - Trains multiple decision trees.
    - Randomizes both **rows (samples)** and **columns (features)**.
    - At each split: selects p\sqrt{p}p features (if p = number of predictors).
    - Helps reduce correlation among trees → better performance.

🔹 **B. Boosting**

- **How it works**:
    - Models are trained **sequentially**.
    - Each new model focuses more on the errors of the previous ones.
    - Final prediction: weighted combination of all models.
- Goal: Turn **weak learners** (e.g., decision stumps) into a **strong learner**.
- Reduces **bias** (and can also reduce variance).

i. AdaBoost (Adaptive Boosting)

- Base model: **Decision stumps** (1-level decision trees).
- **Steps**:
    1. Initialize all samples with equal weights.
    2. Train a stump → calculate error.
    3. Increase weights of **misclassified samples**.
    4. Train next stump on updated weights.
    5. Repeat for multiple rounds.
    6. Final model = weighted sum of all stumps.
- **Performance weight** of each stump:

$$\alpha = \frac{1}{2}\log\left(\frac{1 - Error}{Error}\right)$$

- ✅ Focuses on **hard-to-classify examples**.
- ⚠️ Sensitive to noise and outliers.

ii. Gradient Boosting

- Extends boosting by using **gradient descent** to minimize loss.
- Works for both regression and classification.

Steps:

1. Start with a simple model (e.g., predicting mean of target values).
2. Calculate residuals (errors).
3. Train a tree on residuals.
4. Update model = previous model + learning_rate × new tree.
5. Repeat for multiple iterations.

- **Loss functions**:
    - Regression → Mean Squared Error
    - Classification → Logistic Loss
- **Hyperparameters**:
    - n_estimators: number of trees
    - learning_rate: how much each tree contributes
    - max_depth: complexity of trees

✅ Very accurate, handles nonlinear data.
⚠️ Computationally expensive, sensitive to tuning.

iii. XGBoost (Extreme Gradient Boosting)

- An optimized version of gradient boosting.
- **Key features**:
    - **Regularization (L1 & L2)** → prevents overfitting
    - **Parallelization** → faster training
    - **Tree pruning** → efficient splitting
    - **Handles missing values** automatically
    - Built-in **cross-validation**
- Widely used in **Kaggle competitions** and real-world tasks (fraud detection, ranking, forecasting).

🔹 **C. Stacking (Stacked Generalization)**

- Combines multiple models using a **meta-model**.
- Steps:
    1. Train base learners (e.g., SVM, Decision Tree, KNN).
    2. Generate predictions from these base learners.
    3. Train a **meta-learner** (e.g., Logistic Regression) on their outputs.
    4. Final prediction = meta-model's prediction.

✅ Captures strengths of multiple algorithms.
⚠️ More complex, needs careful tuning.

3. 📊 **Summary Table**

Method	Goal	Models trained	Combine by	Example
Bagging	Reduce variance	Parallel	Majority vote / Avg	Random Forest
Boosting	Reduce bias	Sequential	Weighted vote	AdaBoost, GBM, XGBoost
Stacking	Combine strengths	Parallel + meta	Meta-model	Stacked Classifier

✅ **Key Takeaways for Exams**:

- **Bagging** → Sampling with replacement, OOB error, Random Forest.
- **Boosting** → Sequential correction of errors, AdaBoost focuses on weights, Gradient Boosting uses residuals, XGBoost adds regularization + efficiency.
- **Stacking** → Uses meta-model to blend outputs of base models.

Class Imbalance

Handling **class imbalance** is crucial in classification problems where one class significantly outnumbers the other(s). If not addressed, models may become biased toward the majority class, leading to poor performance on the minority class.

⚖️ What is Class Imbalance?

Occurs when:

- One class has **much more data** than the other(s).
- Example: 95% "No Disease", 5% "Disease" — a model predicting "No Disease" always would still be 95% accurate but useless.

🧠 Why It's a Problem:

- Metrics like **accuracy** become misleading.
- The model may **ignore the minority class**, which is often the class of interest (e.g., fraud, disease).

🛠️ Techniques to Handle Class Imbalance:

🔁 1. Resampling Techniques

- **Oversampling**: Increase the number of minority class samples.
    - e.g., **SMOTE** (Synthetic Minority Over-sampling Technique)
    - SMOTE TOMEKlinks using near by points and calculating their distance to make a new data point so it belongs to the same class
- **Undersampling**: Reduce the number of majority class samples.
    - Risk: Losing valuable information.

⚖️ 2. Class Weights

- Assign **higher penalty** to misclassifying the minority class.
- Many models (e.g., LogisticRegression, SVM, RandomForest) support class_weight='balanced'.

📊 3. Evaluation Metrics

- Use metrics that reflect imbalance:
    - **Precision, Recall, F1 Score**
    - **ROC-AUC**
    - **Confusion Matrix**

🧠 4. Algorithmic Approaches

- Use models that are robust to imbalance:

- **Tree-based models** (e.g., XGBoost with scale_pos_weight)
- **Ensemble methods** (e.g., Balanced Random Forest)

✅ **Best Practice:**

Use a **combination** of:

- Resampling
- Class weighting
- Proper evaluation metrics

Scaling

🔍 **Why Scaling Matters in Machine Learning:**

Many algorithms (like **SVM**, **KNN**, **Logistic Regression**, and **Gradient Descent-based models**) are sensitive to the **scale of features**. If features are on very different scales, the model might give more importance to larger-scale features.

⚙️ **Types of Scaling:**

1. **Original Data**:
    - Features are in their raw form.
    - May have different ranges and units.
2. **Standardization (Z-score scaling)**:
    - Transforms data to have **mean = 0** and **standard deviation = 1**.
    - Formula: $Z = \frac{x-\mu}{\sigma}$
    - Best when data is normally distributed.

Where:

- x = original value
- μ = mean of the feature
- σ = standard deviation of the feature

3. **Normalization (Min-Max scaling)**:
   . Scales data to a **[0, 1]** range.
   . Formula:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Where:

- x = original value

- $x$min = minimum value of the feature
- $x$max = maximum value of the feature

📊 1. Standardization (Z-score Scaling)

Use when:

- Your data follows a **normal (Gaussian) distribution**.
- You're using algorithms that **assume standardized features** or are sensitive to feature variance.

Best for:

- **SVM**
- **Logistic Regression**
- **Linear Regression**
- **K-Means**
- **PCA (Principal Component Analysis)**

📈 2. Min-Max Normalization

Use when:

- You need features in a **bounded range**, typically [0, 1].
- Your data does **not follow a normal distribution**.
- You're using models that are **sensitive to the magnitude of input values**.

Best for:

- **Neural Networks**
- **K-Nearest Neighbours (KNN)**
- **Gradient Descent-based models**

📉 3. Max-Abs Scaling

Use when:

- Your data is **sparse** (lots of zeros), such as in text classification.
- You want to preserve **zero entries** and scale only based on the maximum absolute value.

Best for:

- **Sparse data**
- **Online learning algorithms** (like SGD)

🧠 Summary Table:

Scaling Method	Best When...	Algorithms That Benefit
Standardization	Data is Gaussian or has outliers	SVM, Logistic Regression, PCA
Min-Max	Data needs to be in [0, 1] range	Neural Nets, KNN
Max-Abs	Data is sparse	SGD, Text Models

Sklearn Pipeline

The **sklearn.pipeline.Pipeline** is a powerful tool in **Scikit-learn** that allows you to **chain together multiple steps** of a machine learning workflow into a single object. This makes your code cleaner, more modular, and easier to tune and reproduce.

🔧 **Why Use a Pipeline?**

- Ensures **consistent preprocessing** during training and testing.
- Simplifies **cross-validation** and **hyperparameter tuning**.
- Reduces the risk of **data leakage**.
- Makes your code more **organized and readable**.

🧱 **Typical Pipeline Structure:**

You can then use it like a regular model:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
pipeline = Pipeline([
  ('scaler', StandardScaler()),   # Step 1: Feature scaling
  ('svm', SVC(kernel='rbf'))     # Step 2: Model training
])
pipeline.fit(X_train, y_train)
predictions = pipeline.predict(X_test)
```

🧪 **With Cross-Validation or Grid Search:**

```
from sklearn.model_selection import GridSearchCV
param_grid = {
  'svm__C': [0.1, 1, 10],
  'svm__gamma': [0.01, 0.1, 1]
```

}

grid = GridSearchCV(pipeline, param_grid, cv=5)

grid.fit(X_train, y_train)

Note: Use stepname__parametername syntax to tune parameters inside the pipeline.

You can later on set the parameters by pipe.set_parametername.

K-Fold Cross Validation is a powerful technique used in machine learning to evaluate the performance of a model. It helps ensure that the model generalizes well to unseen data by reducing the risk of overfitting or underfitting.

🔍 **What is K-Fold Cross Validation?**

In K-Fold Cross Validation, the dataset is divided into K equal parts (folds). The model is trained and validated K times, each time using a different fold as the validation[Test] set and the remaining K-1 folds as the training set.

📊 **How It Works (Step-by-Step):**

1. Split the data into K folds (e.g., K=5).
2. For each fold:
    - Use the fold as the validation set.
    - Use the remaining K-1 folds as the training set.
    - Train the model and evaluate it on the validation set.
3. Average the performance across all K trials to get a more reliable estimate.

✅ **Benefits:**

- More reliable performance estimate than a single train/test split.
- Efficient use of data, especially useful when the dataset is small.
- Helps detect variance and bias in the model.

⚠️ **Considerations:**

- More computationally expensive than a single split.
- **For time-series data, use TimeSeriesSplit instead of regular K-Fold.**

Stratified K-Fold Cross Validation is a variation of K-Fold Cross Validation that ensures each fold has the same proportion of **class labels** as the original dataset. This is especially useful when dealing with imbalanced datasets.

🧠 **Why Use Stratified K-Fold?**

In regular K-Fold, the data is split randomly, which might lead to uneven class distributions in some folds. This can skew the model's performance evaluation. Stratified K-Fold solves this by preserving the class distribution in each fold.

If the main class have 30 70 distributions then stratified k fold make sure it maintains 30 70 split in all its samples (specially in test)

🔁 How It Works:

1. Split the dataset into K folds.
2. Ensure that each fold has approximately the same percentage of samples of each target class as the complete dataset.
3. Train and validate the model K times, each time using a different fold as the validation set.

✅ Benefits:

- Maintains class balance in each fold.
- Provides a more reliable estimate of model performance, especially for classification problems with imbalanced classes.

GridSearchCV is a powerful tool in scikit-learn used for hyperparameter tuning. It helps you find the best combination of hyperparameters for a machine learning model by exhaustively searching through a specified grid of values.

🔍 What is Hyperparameter Tuning?

Hyperparameters are the settings not learned from the data but set before training (e.g., number of trees in a random forest, learning rate in gradient boosting). Tuning them can significantly improve model performance.

⚙️ What is GridSearchCV?

GridSearchCV performs:

1. Exhaustive search over a grid of hyperparameter values.
2. Cross-validation to evaluate each combination.
3. Selection of the best model based on a scoring metric (e.g., accuracy, F1-score).

✅ Benefits:

- Finds the **optimal hyperparameters**.
- Uses **cross-validation** to avoid overfitting.
- Can be combined with **StratifiedKFold** for classification tasks.

RandomizedSearchCV is another hyperparameter tuning technique in **scikit-learn**, and it's often used as a faster alternative to **GridSearchCV**.

🔍 What is RandomizedSearchCV?

Instead of trying **every possible combination** like GridSearchCV, **RandomizedSearchCV**:

- **Randomly samples** a fixed number of parameter combinations from a specified distribution.
- Evaluates each combination using **cross-validation**.
- Returns the **best combination** based on a scoring metric.

⚙️ When to Use It?

- When the **parameter space is large**.
- When you want to **save time and computational resources**.
- When you're okay with a **good-enough solution** rather than the absolute best.

✅ Benefits:

- **Faster** than GridSearchCV for large search spaces.
- Can explore **more diverse combinations**.
- Still uses **cross-validation** for robust evaluation.

🔹 Unsupervised Learning

- **Definition:** A type of machine learning where the algorithm learns from **unlabeled data**.
- **Goal:** Discover **hidden patterns**, **group structures**, or reduce **dimensionality**.

Key Concepts

- **No Labels:** No predefined outcomes.
- **Pattern Discovery:** Algorithm learns from data itself.
- **Dimensionality Reduction:** Simplifies data while keeping important information.
- **Clustering:** Groups similar data points together.

Common Applications

- Customer segmentation
- Anomaly detection (fraud detection, outliers)
- Recommender systems
- Gene expression analysis
- Image and speech preprocessing

1. K-Means Clustering

Definition: Algorithm to partition data into **K non-overlapping clusters**.

Steps:

1. Choose number of clusters KKK (hyperparameter).
2. Initialize KKK centroids randomly.
3. Assign each data point to nearest centroid (using **distance metric**, usually Euclidean).

$$d = \sqrt{(x_2 - x_1)^2}(y_2 - y_1)^2$$

1. Recompute centroids (mean of assigned points).
2. Repeat steps 3–4 until centroids stop changing (convergence).

Optimal K:

- Use the **Elbow Method** → plot SSE (Sum of Squared Errors) vs. K.
- Choose K at the "elbow point" (where SSE stops decreasing significantly).

Pros:
✅ Simple, fast, scalable
✅ Works well with large datasets

Cons:
❌ Requires pre-specifying K
❌ Sensitive to outliers
❌ Assumes clusters are spherical and equal-sized

2. Hierarchical Clustering

Definition: Groups data based on similarity, producing a **hierarchy of clusters**.

Types:

- **Agglomerative (Bottom-Up):** Each observation starts as its own cluster, merges step by step.
- **Divisive (Top-Down):** All data starts in one cluster, splits recursively.

Steps (Agglomerative):

1. Treat each point as its own cluster.
2. Compute **distance matrix** between clusters.
3. Merge the closest clusters.
4. Update distances.
5. Repeat until all points are in one cluster.

Distance/Linkage Methods:

- **Single linkage:** Min distance between clusters.

- **Complete linkage:** Max distance.
- **Average linkage (Ward):** Mean distance (commonly used).

Visualization:

- **Dendrogram** → tree diagram showing merging process.
- Cut at a certain height to determine number of clusters.

Pros:
✅ No need to predefine K
✅ Produces hierarchy of clusters

Cons:
❌ Computationally expensive for large datasets
❌ Sensitive to noise/outliers

3. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

Definition: Clustering based on **density of data points**, can detect noise/outliers.

Key Parameters:

- ε\varepsilonε (epsilon): Radius for neighborhood search
- **minPts:** Minimum number of points in ε\varepsilonε-neighborhood to form a cluster

Types of Points:

- **Core point:** ≥ minPts neighbors within ε\varepsilonε
- **Border point:** Within ε\varepsilonε of a core point, but fewer than minPts
- **Noise point:** Neither core nor border

Steps:

1. For each point, find neighbors within ε\varepsilonε.
2. If ≥ minPts → core point → start cluster.
3. Expand cluster by adding all density-reachable points.
4. Label leftover points as noise.

Pros:
✅ Finds clusters of arbitrary shape
✅ Automatically detects outliers
✅ No need to specify K

Cons:
❌ Sensitive to parameter choice (ε\varepsilonε, minPts)
❌ Struggles with varying densities

4. PCA (Principal Component Analysis) — Dimensionality Reduction

Definition: Reduces number of variables while retaining most variance.

Key Idea:

- Transforms correlated features into a new set of **uncorrelated variables (principal components)**.
- **First component:** captures max variance.
- **Second component:** captures next most variance, orthogonal to first.

Math (Eigen Decomposition):

$$Av = \lambda v$$

- **Eigenvector (v):** Direction of variance
- **Eigenvalue (λ\lambdaλ):** Magnitude of variance

Steps:

1. Standardize data.
2. Compute covariance matrix.
3. Find eigenvalues & eigenvectors.
4. Rank eigenvalues (largest → most variance).
5. Select top components to reduce dimensions.

Pros:
✅ Reduces dimensionality → faster computation
✅ Removes redundancy (correlated features)
✅ Improves visualization in 2D/3D

Cons:
❌ Components are less interpretable
❌ Assumes linear relationships
❌ Sensitive to scaling

📊 Quick Comparison

Algorithm	Type	Needs K?	Strengths	Weaknesses
K-Means	Partitioning	Yes	Fast, scalable	Assumes spherical clusters
Hierarchical	Hierarchy-based	No	Dendrogram, hierarchy view	Expensive, sensitive
DBSCAN	Density-based	No	Arbitrary shape, detects noise	Sensitive to params

Algorithm	Type	Needs K?	Strengths	Weaknesses
PCA	Dim. Reduction	No	Handles correlation, reduces features	Loses interpretability

📘 Machine Learning – Model Guide

1. Model Selection Guide

1.1 Regression Models

- **Based on Relationship:**
  - **Simple Linear:** Linear Regression
  - **Moderately Linear:** Polynomial Regression
  - **Highly Non-linear:**
    - Decision Tree Regression
    - Random Forest Regression
    - XGBoost Regressor
    - SVM Regressor
- **Based on Dataset Size:**
  - **Small to Medium:**
    - Linear Regression
    - Polynomial Regression
    - Decision Tree Regression
    - SVM Regression
  - **Large:**
    - Random Forest Regression
    - XGBoost Regression

1.2 Classification Models

- **Based on Relationship:**
  - **Linear:** Logistic Regression
  - **Non-linear / Complex:** Decision Tree, Random Forest, XGBoost
- **Based on Data Type:**
  - **Text Data:** Naïve Bayes, SVM

    - **Categorical Data:** Decision Tree, Random Forest, XGBoost
    - **Numeric Data:** Logistic Regression, KNN, SVM, Random Forest, XGBoost
- **Based on Dataset Size:**
    - **Small to Medium:** Logistic Regression, SVM, Naïve Bayes
    - **Large:** Random Forest, XGBoost

2. Evaluation Metrics

2.1 Classification

- **Accuracy:** Overall correctness (best for balanced data).
- **Precision:** How many predicted positives are actually positive (useful for spam detection).
- **Recall:** Ability to capture actual positives (important for medical diagnosis).
- **AUC-ROC:** Measures discrimination between classes.
- **Confusion Matrix:** Summarizes TP, TN, FP, FN.

2.2 Regression

- **MAE (Mean Absolute Error):** Average magnitude of errors.
- **MSE (Mean Squared Error):** Penalizes large errors, sensitive to outliers.
- **RMSE (Root Mean Squared Error):** Easier to interpret (same unit as target).
- **R^2 Score:** Goodness of fit; variance explained by the model.

3. ML Project Life Cycle (10 Stages)

1. **Team Structure**
    - Business Team: CEO, Sales Head, AI Product Manager
    - Technical Team: AI Tech Lead, AI Engineers
2. **Requirements**
    - Define Scope of Work (SOW): Objectives, Deliverables, Timeline, Exclusions, Milestones
    - Tools: MS Docs, Excel, Jira, Notion
3. **Data Collection**
    - Internal: Databases (Oracle, MongoDB, S3)
    - External: Vendors, Web Scraping
    - Consider: Compliance, Access Control
    - Tools: SQL, MongoDB, BeautifulSoup
4. **Data Preparation**

- Cleaning (remove PII, duplicates, outliers, missing values)
- EDA (statistical analysis, visualization)
- Tools: Python, Pandas, Matplotlib, Seaborn, Spark

5. **Feature Engineering**
    - Cleaning, Transformation, Selection, Creation
    - Encoding (OHE, Label Encoding, Target Encoding)
    - Scaling (Standardization, Min-Max)
    - Dimensionality Reduction (PCA, t-SNE, UMAP)
    - Feature Selection (Correlation, VIF, Model-based methods)
6. **Model Selection & Training**
    - Choose appropriate ML model based on problem/data.
    - Tools: Python, Jupyter, scikit-learn
7. **Model Evaluation**
    - Use proper metrics (classification vs regression).
8. **Model Fine-Tuning**
    - Cross-validation
    - Hyperparameter tuning (GridSearchCV, RandomSearchCV)
9. **Model Deployment**
    - Training → Inference
    - Tools: FastAPI, Frontend integration
10. **Monitoring & Feedback**

- MLOps pipeline
- Model monitoring, retraining, Responsible AI

4. Feature Engineering (Deep Dive)

4.1 Key Steps

- **Understand Data:** Stats + visualization
- **Handle Missing Values:** Mean/Median, KNN imputation, drop rows/columns
- **Encoding:** Label, One-Hot, Target
- **Scaling:** Standardization, Min-Max
- **Transformation:** Log, sqrt, Box-Cox, polynomial features
- **Creation:** Combine features, extract from dates, text features (TF-IDF, sentiment)

- **Dimensionality Reduction:** PCA, t-SNE, UMAP
- **Selection:** Correlation, Mutual Info, Feature Importance, VIF

4.2 Variance Inflation Factor (VIF)

- **Purpose:** Detect multicollinearity among features
- **Formula:** VIF = $1 / (1 - R^2)$
- **Interpretation:**
    - VIF ≈ 1 → No correlation
    - VIF > 5 or 10 → Strong multicollinearity

✅ This cleaned-up structure makes it easier to study in **modules**:

1. Models (Regression vs Classification)
2. Evaluation Metrics
3. ML Project Life Cycle
4. Feature Engineering