SUPERVISED LEARNING

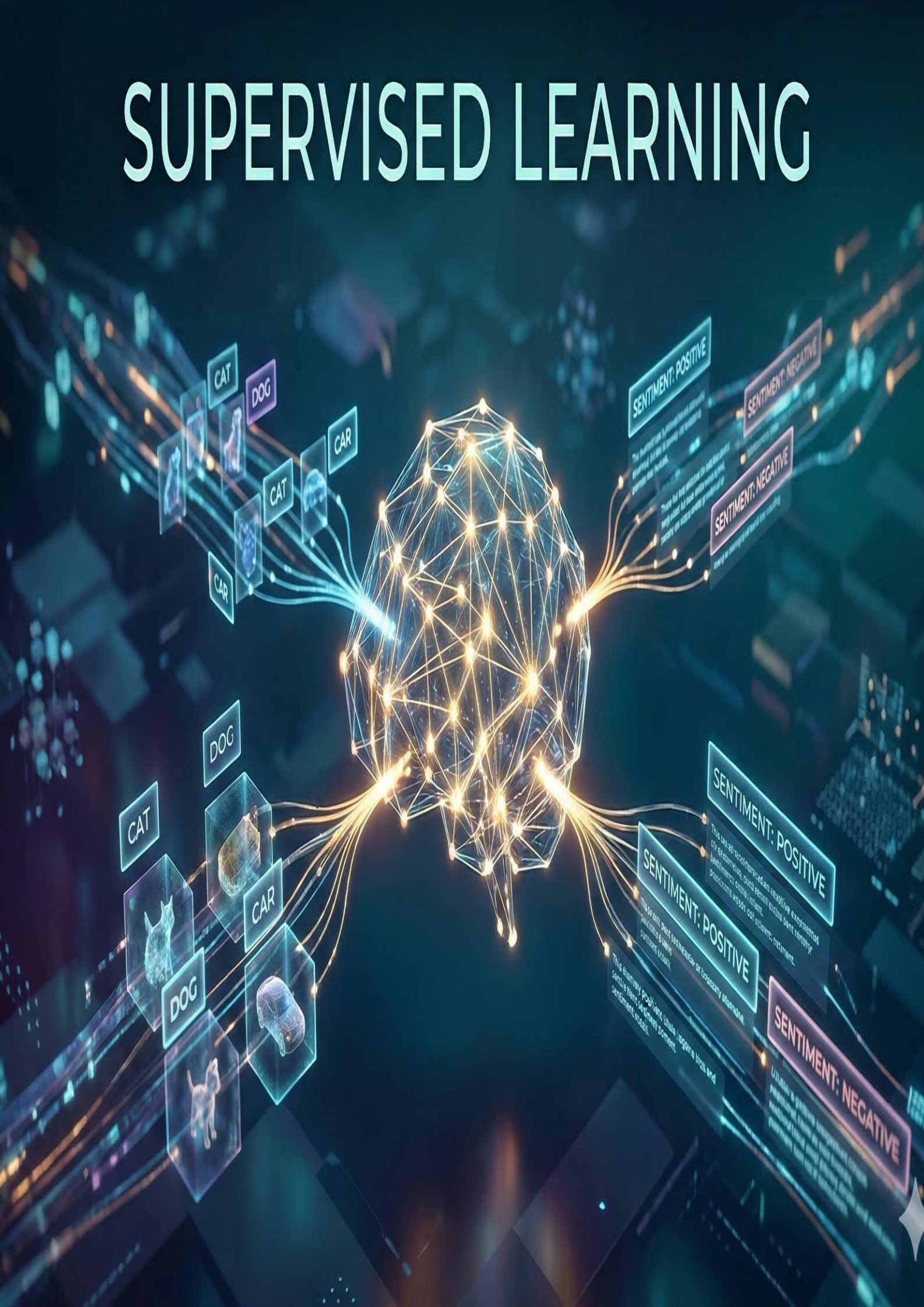


Table of Contents

Introduction

Purpose

The purpose of this work is to bridge theoretical knowledge with practical application. By learning the core concepts behind each model—such as cost functions, decision boundaries, and ensemble strategies—it aims to guide practitioners and students in choosing the right algorithm for the right problem.

Scope

This document provides a comprehensive overview of the theoretical foundations of Artificial Intelligence (AI) and Machine Learning (ML). It focuses on widely used supervised models, including linear regression, logistic regression, decision trees, random forests, gradient boosting, XGBoost, support vector machines (SVM), and K-nearest neighbors (KNN). The scope extends from mathematical concepts to real-world applications across domains like prediction and classification.

Procedure

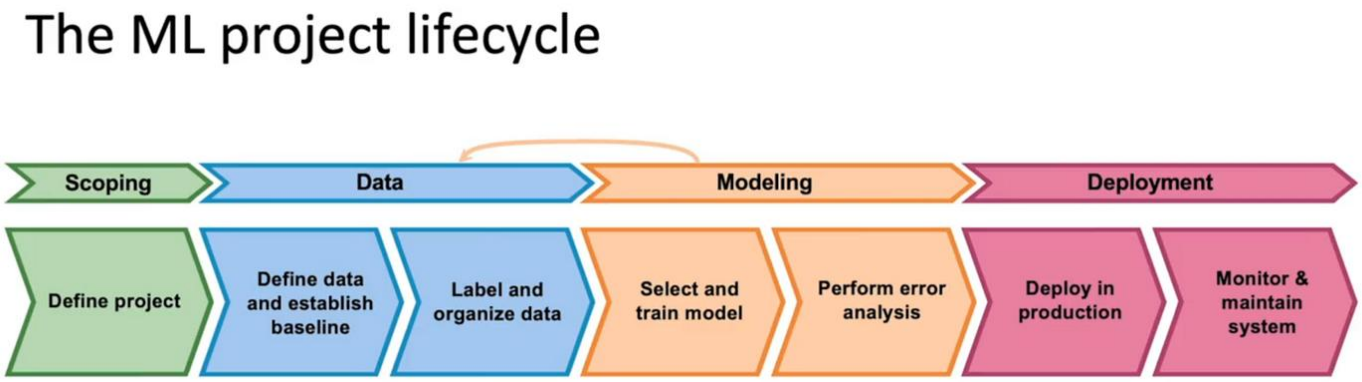
The study follows a structured five-step procedure:

1. **Theoretical Study:** Reviewing mathematical foundations (e.g., entropy, optimization).
2. **Implementation:** Writing code examples for training and evaluating models.
3. **Visualization:** Using plots to illustrate model fits and predictions.
4. **Evaluation:** Comparing models using metrics like accuracy, F1-score, and RMSE.
5. **Guidelines:** Providing decision-making rules for when to use each model.

Brief Problem

In modern applications, datasets are increasingly large, complex, and diverse. A central challenge is selecting the most suitable ML model, as some excel at handling high-dimensional data while others are more interpretable but less scalable. This document addresses this by offering a structured approach to understanding and evaluating these algorithms.

Full Machine Learning Project Cycle



Made by Youssef Amgad Elkhatib

Important Concepts

Loss Function

- Definition: Measures the error **for a single training example**.

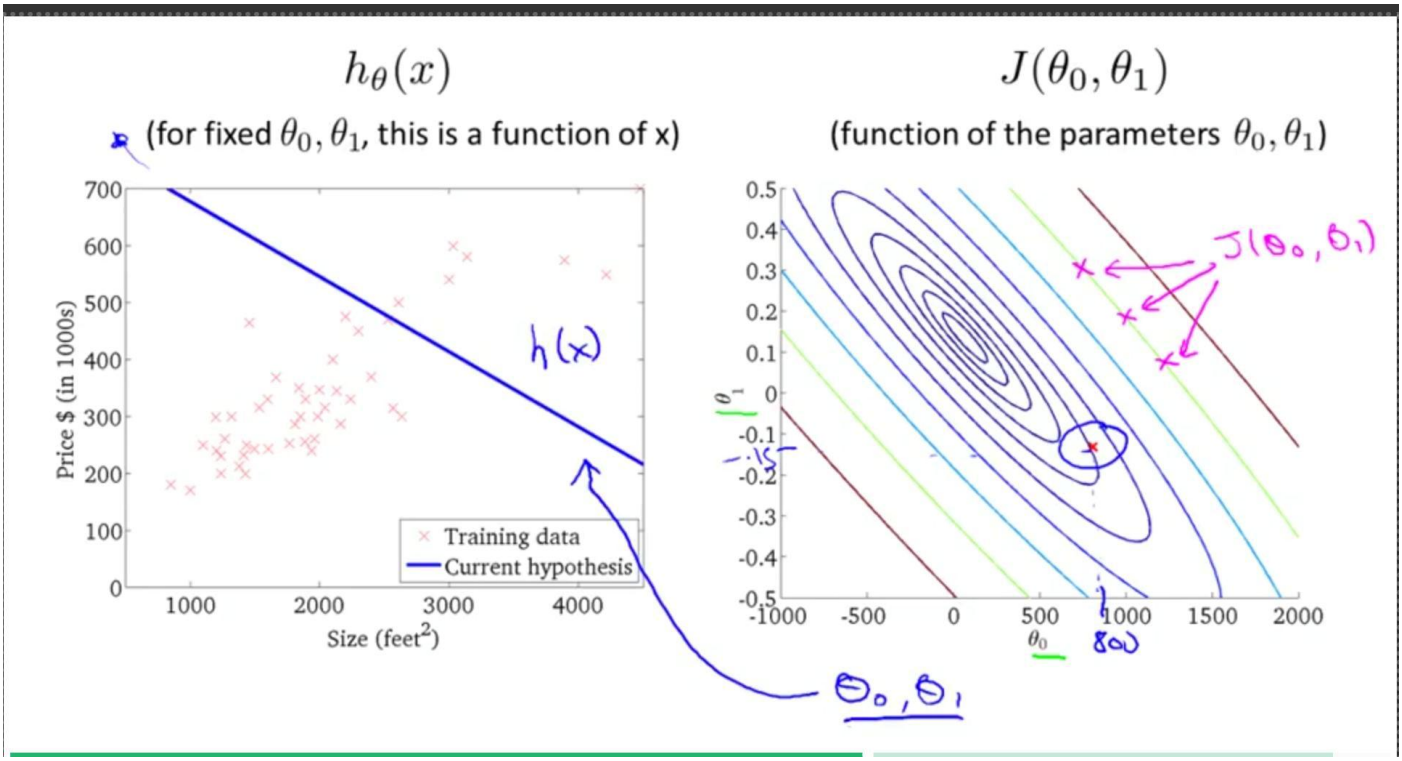
Cost Function

- Definition: The **average (or sum)** of all loss values across the dataset.
- Used for optimization (minimization with gradient descent).

So: Loss = per sample, Cost = average over dataset.

The 2D Contour View

- **The Axes:** The X and Y axes represent your parameters.
- **The Lines:** Every point on a single colored line has the **exact same cost** (error).
- **The Bullseye:** The center of the smallest inner ellipse is the Global Minimum. This is the goal. Gradient Descent is essentially taking steps from the outer rings toward this center "bullseye".



Made by Youssef Amgad Elkhatib

Underfitting, Overfitting and Generalization

1. Underfitting

- Happens when the model is **too simple** to capture the underlying patterns.
- It performs poorly on **training data** and **test data**.
- Cause: High **bias**.
- Example: Using a straight line to fit data that is quadratic.

---

2. Overfitting

- Happens when the model is **too complex** and learns not just the pattern but also the noise in training data.
- Performs very well on **training data**, but poorly on **unseen/test data**.
- Cause: High **variance**.
- Example: A curve that passes through every training point, even outliers.

---

3. Generalization

- A model's ability to **perform well on unseen data** (not just the training set).
- Good generalization lies in between underfitting and overfitting.
- Goal of ML: train models that generalize well.

Bias & Variance

Bias

- **Definition**: Error due to simplifying assumptions made by the model.
- High bias = model ignores important patterns (underfitting).
- Mathematically:
    - If the true relationship is y=f(x), but the model assumes y=ax+b, then bias is the difference between expected prediction and true value.
- **Example**:
    - Predicting house prices using only "size" while ignoring "location" or "age".

Variance

- **Definition**: Error due to **too much sensitivity** to training data.
- High variance = model learns noise, not just signal (overfitting).
- Mathematically:
    - Variance measures how much predictions change if we train the model on a different dataset sampled from the same distribution.
- **Example**:
    - A decision tree that perfectly memorizes the training set but fails on new houses.

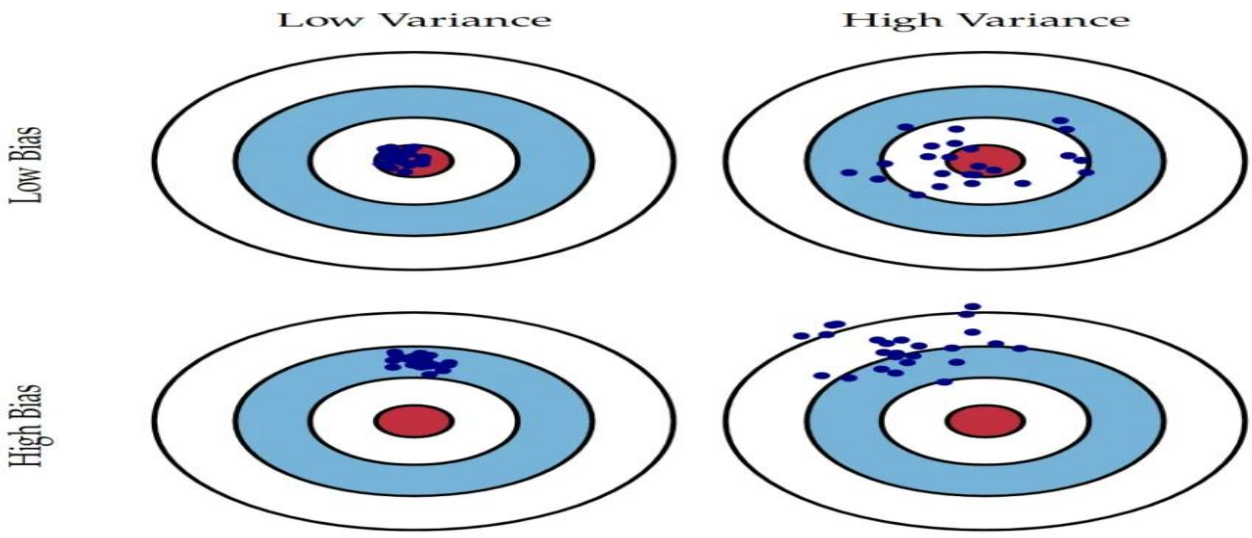


Fig. 1 Graphical illustration of bias and variance.

Case 1: High Bias (Underfitting)

Metric	Value
Train Accuracy	LOW
CV Accuracy	LOW
Gap (Train vs CV)	SMALL

Meaning:

The model cannot even fit the training data.

Example:
Train = 60%
CV = 58%

Both bad → **high bias**

Case 2: High Variance (Overfitting)

Metric	Value
Train Accuracy	HIGH
CV Accuracy	LOW
Gap	LARGE

Example:
Train = 98%
CV = 75%

Model memorized training data → **high variance**

Case 3: Good Generalization (Ideal) – Both Low

Metric	Value
Train Accuracy	HIGH
CV Accuracy	HIGH
Gap	SMALL

Example:
Train = 95%
CV = 93%

This is what we want.

Case 4: High Bias + High Variance

This is less common but possible.

Metric	Value
Train Accuracy	LOW
CV Accuracy	VERY LOW
Gap	LARGE

Example:
Train = 70%
CV = 50%

Model:

- Too simple (bias)
- Also unstable (variance)

Regularization and Lambda (λ)

Regularization controls **model complexity**.

It adds a penalty to large weights:

$$J = Loss + \lambda \sum w^2$$

Effect of Lambda Values

Lambda	**Bias**	**Variance**	**Result**
Very Low	Low	High	Overfitting
Medium	Balanced	Balanced	Best generalization
Very High	High	Low	Underfitting

Baseline Performance

Baseline = performance of a **simple model or human-level**.

It tells you:

"How good should my model realistically be?"

Diagnosing with Baseline

Assume baseline accuracy = 90%.

Case A: High Bias

Train = 75%
Baseline = 90%

Huge gap → model underfitting.

Made by Youssef Amgad Elkhatib

Case B: High Variance

Train = 95%
CV = 85%
Baseline = 93%

Train close to baseline, CV far → variance.

Case C: Good Model

Train = 92%
CV = 91%
Baseline = 90%

The model is performing near optimal.

Case D: High Variance & High Bias

Train = 72%
CV = 61%
Baseline = 90%

The model is terrible.

How to Solve these Problems?

Adding More Data

Problem	Add More Data?
High Bias	❌ No
High Variance	✅ Yes

How to Fix High Bias

If underfitting:

Solutions:

- Use a more complex model
- Reduce regularization (lower λ)
- Add more features
- Try adding polynomial features

How to Fix High Variance

If overfitting:

Solutions:

- Add more data
- Increase regularization (higher λ)
- Try a smaller set of features
- Reduce model complexity

Parameters vs Hyperparameters

1. Parameters

- **Definition**: Internal values that a model **learns automatically** from the training data.
- Adjusted during training through optimization (e.g., gradient descent).
- Model's "knowledge".

✅ Examples:

- **Linear Regression** → coefficients (weights) w, intercept b.
- **Neural Networks** → weights and biases of neurons.
- **Logistic Regression** → coefficients for each feature.

⚡ Key: Parameters are **learned by the model**.

---

2. Hyperparameters

- **Definition**: External settings chosen **before training** to control how the model learns.
- They are **not learned from data**, but instead set manually or tuned.
- Affect the speed, accuracy, and generalization of training.

✅ Examples:

- **Learning rate** (step size in gradient descent).
- **Number of hidden layers/neurons** in a neural network.
- **Batch size & number of epochs**.
- **k** in k-Nearest Neighbors.
- **C, kernel, gamma** in SVM.

⚡ Key: Hyperparameters are **set by the practitioner**.

Hyperparameter Tuning

- **Tuning = searching for hyperparameter values that give the best model performance on unseen data.**
- **Goals: get best generalization, avoid overfitting to the training set, and be efficient with compute.**

Grid Search (exhaustive)

What it is

- Try *every combination* of values in a user-specified grid.

Pros

- Simple and deterministic.
- Good when search space is small and you want full coverage.

Cons

- Explodes combinatorially with many hyperparameters.
- Wasteful if many parameters don't matter (looks at unimportant combinations).

When to use

- Small number of hyperparameters with a few candidate values.

Random Search

What it is

- Sample n_iter random combinations from distributions or lists of values.

Pros

- Much more efficient when only a few hyperparameters strongly affect performance.
- Easy to search continuous or very large spaces (sample learning rates on log scale, etc).

Cons

- Stochastic: results vary by random seed (but reproducible with random_state).
- Can still miss narrow sweet spots unless n_iter is large enough.

When to use

- Large search spaces, continuous hyperparameters, limited compute budget.

Cross-Validation

Purpose

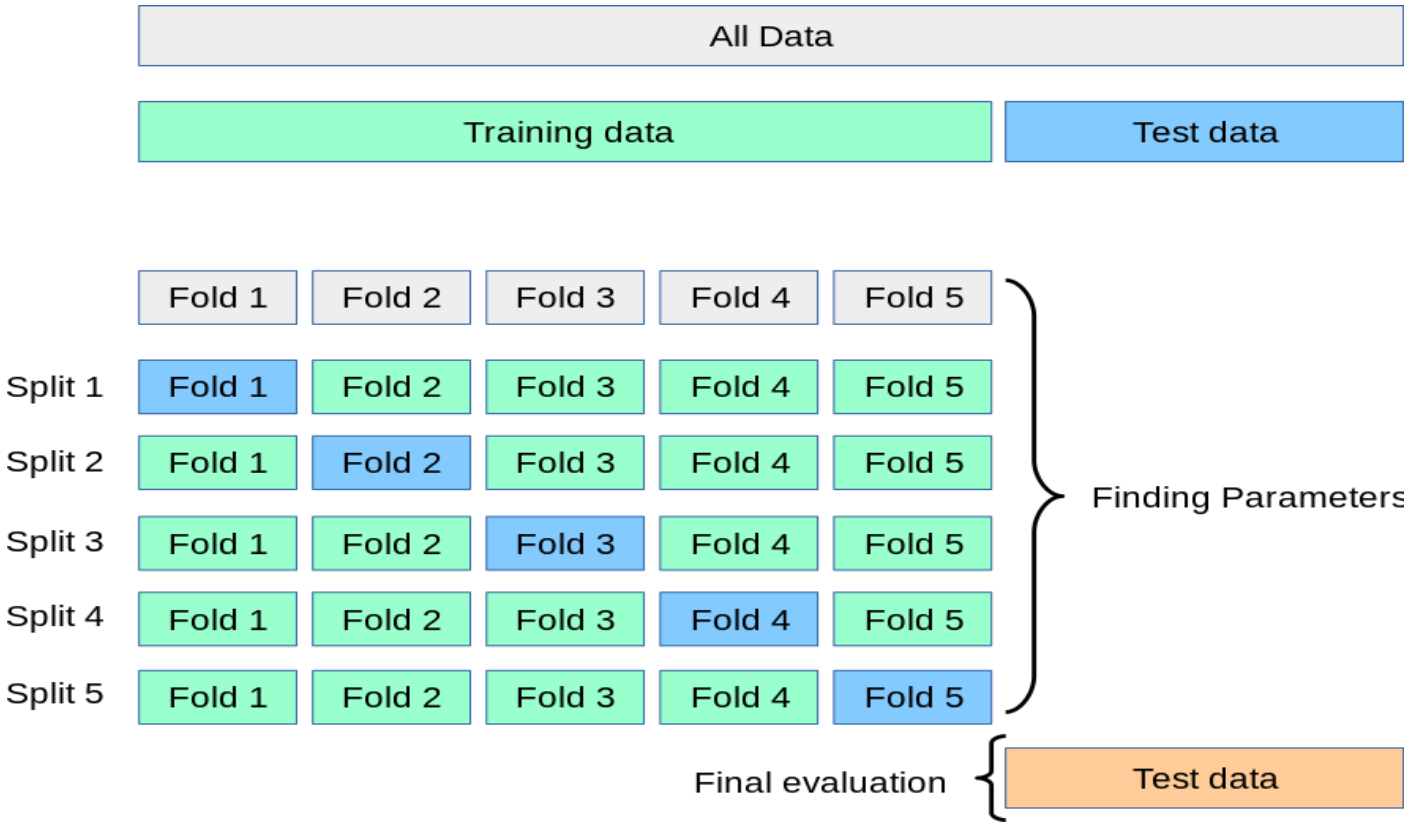
- Estimate model performance on unseen data.
- Use CV as the evaluation inner loop when tuning hyperparameters so the score reflects generalization.

Basic K-fold

- Split data into k folds. Train on k-1, validate on the held-out fold; repeat and average scores.
- Common choices: k=5 or k=10.

Choosing k

- Larger k (e.g., 10) → lower bias in estimate but higher variance and compute.
- Smaller k (e.g., 5) → faster, sometimes adequate.



Cross-Validation Main Parameters

Parameter	Description	Common Values / Notes
n_splits	Number of folds (k) the data is split into.	Typical: 5 or 10
shuffle	Whether to shuffle data before splitting.	True (recommended for IID data), False for time series
random_state	Seed for reproducibility when shuffle=True.	Any integer (e.g., 42)
train_size	(Some CV variants) Proportion/size of training split.	Float (0.8 = 80%) or int
test_size	(Some CV variants) Proportion/size of validation/test split.	Float (0.2 = 20%) or int
groups	(For GroupKFold / LeaveOneGroupOut) Keeps group samples together.	Array of group labels
max_train_size	(For TimeSeriesSplit) Limits training set size in each split.	Integer or None (no limit)

Key Cross-Validation Variants

CV Variant	Use Case	Key Characteristics
KFold	General-purpose CV	Splits dataset into *k* equal folds. May shuffle if specified.
StratifiedKFold	Classification (esp. imbalanced)	Preserves class proportions in each fold.
RepeatedKFold	Reduce variance in CV estimates	Repeats KFold multiple times with different splits.
LeaveOneOut (LOO)	Very small datasets	Uses 1 sample as test, rest as train. Expensive.
LeavePOut (LPO)	Tiny datasets, detailed evaluation	Leaves out *p* samples each time. More expensive than LOO.
GroupKFold	Grouped samples (e.g., patients, users)	Ensures all samples of a group are in the same fold.
LeaveOneGroupOut (LOGO)	Group-based evaluation	Leaves out one group at a time as test.
TimeSeriesSplit	Sequential / time-series data	Respects temporal ordering (no shuffling). Training grows with each split.
RepeatedStratifiedKFold	Classification + imbalanced classes	Like RepeatedKFold but keeps class proportions.

Correlation

Definition

- Correlation measures the **strength and direction of the relationship** between two variables.
- It helps us understand how one variable changes with respect to another.
- Range: $-1 \leq r \leq 1$

Types of Correlation

Correlation Value (r)	Meaning
r = 1	Perfect positive correlation (variables move together exactly).
r =−1	Perfect negative correlation (variables move in exact opposite directions).
r = 0	No correlation (no linear relationship).
0 < r < 10	Positive correlation (as X increases, Y tends to increase).
−1 < r < 0	Negative correlation (as X increases, Y tends to decrease).

Common Correlation Measures

Method	When to Use	Formula / Description
Pearson correlation	Continuous numeric variables, linear relationships	Measures linear correlation between X and Y
Spearman's rank correlation	Ordinal data or non-linear monotonic relationships	Uses ranks instead of raw values
Kendall's Tau	Small datasets or ordinal variables	Based on concordant and discordant pairs
Point-Biserial correlation	One continuous + one binary variable	Special case of Pearson correlation
Cramer's V / Chi-square-based	Categorical variables	Measures association strength

Why Correlation Matters in ML

Feature selection:

Highly correlated features → multicollinearity (bad for linear regression, logistic regression).

Solution: drop/reduce correlated features (e.g., using PCA).

Data understanding:

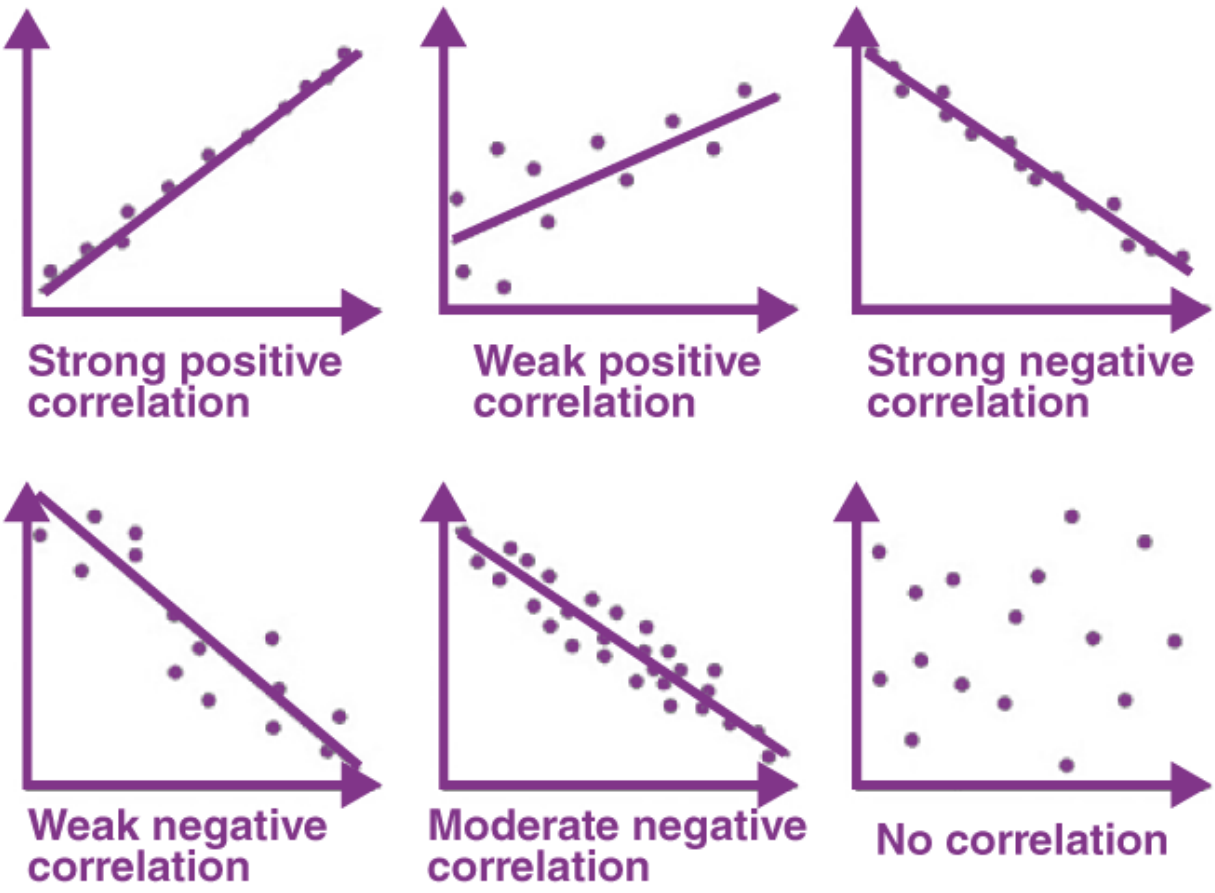
- Find relationships between variables before modeling.

Target correlation:

- Helps identify which features are most predictive.

Assumptions in models:

- Linear models assume low correlation among independent variables.



Random Seed

Definition

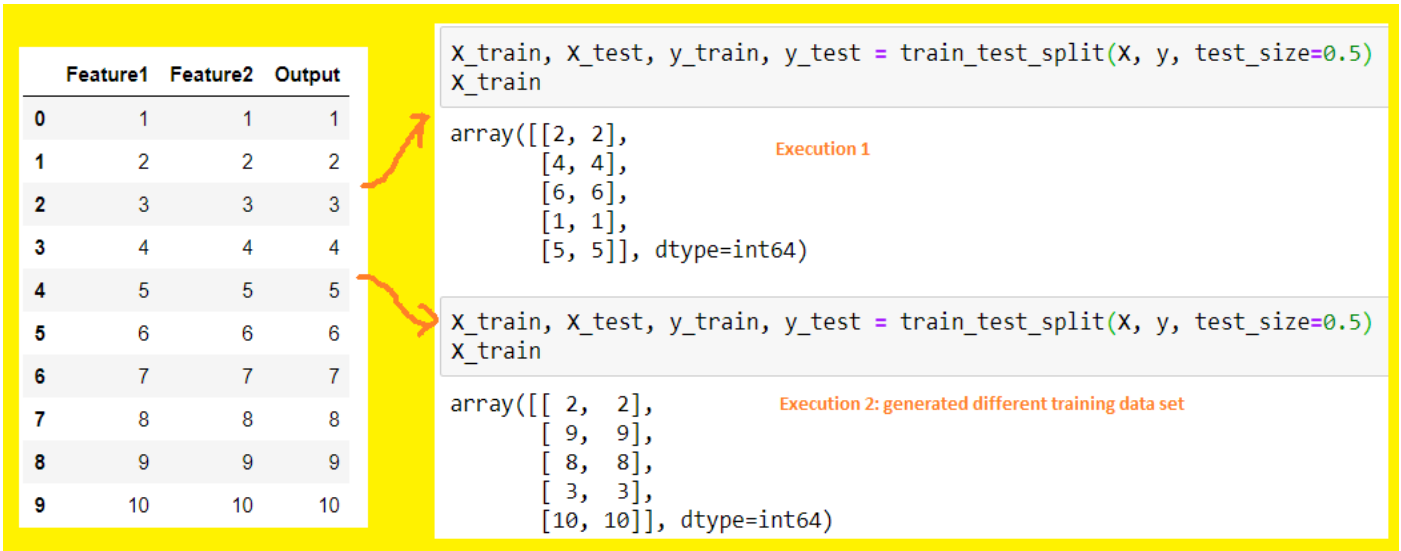
- A **random seed** is an initial value used by a random number generator to produce a sequence of random numbers.
- In AI/ML, randomness is used in tasks like shuffling data, initializing weights in neural networks, and splitting datasets.

Why It Matters

- Without fixing a seed, each run may produce **slightly different results**, making experiments inconsistent.
- Setting a seed ensures **reproducibility** → you and others can replicate the same experiment with identical results.

Where It's Used in ML/DL

1. **Data Splitting** → Train/test split shuffling.
    - Example: train_test_split(data, test_size=0.2, random_state=42)
2. **Weight Initialization** → Neural networks start with different random weights if no seed is fixed.
3. **Sampling / Augmentation** → Random sampling of data, random image flips/rotations.
4. **Optimization Algorithms** → Some stochastic algorithms rely on randomness.



	Feature1	Feature2	Output
0	1	1	1
1	2	2	2
2	3	3	3
3	4	4	4
4	5	5	5
5	6	6	6
6	7	7	7
7	8	8	8
8	9	9	9
9	10	10	10

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
X_train
```

```
array([[2, 2],
       [4, 4],
       [6, 6],
       [1, 1],
       [5, 5]], dtype=int64)
```

Execution 1

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
X_train
```

```
array([[ 2,  2],
       [ 9,  9],
       [ 8,  8],
       [ 3,  3],
       [10, 10]], dtype=int64)
```

Execution 2: generated different training data set

Multi-Class Classification vs Multi-label Classification

Multi-Class Classification

Each example belongs to ONE and ONLY ONE class.

Think:

“Pick exactly one option.”

Examples:

- Handwritten digit → 0–9
- Animal image → cat OR dog OR bird
- Disease type → flu OR covid OR malaria

Multi-Label Classification

Each example can belong to MULTIPLE classes at the same time.

Think:

“Pick ALL that apply.”

Examples:

- Photo tags → beach + sunset + people
- Music genre → rock + pop
- Medical diagnosis → diabetes + hypertension

Regression

Linear Regression

Linear Regression models the relationship between input features x and a continuous target variable y.

---

1. Model (Hypothesis Function)

$$\hat{y} = h_\theta(x) = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

or in vector form:

$$\hat{y} = w^T x + b$$

where:

- $w = (w_1, w_2, \ldots, w_n)$ **are the weights.**
- **b is the bias (y-intercept).**
- $(x_1, \ldots, x_n)$ **are the input features.**

---

2. Actual Function

$$y = \hat{y} + e$$

Where e is the error

---

3. Cost Function (Mean Squared Error, MSE)

We measure the difference between predicted and actual values:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - (w^T x^{(i)} + b))^2$$

This is the cost function we want to minimize.

Made by Youssef Amgad Elkhatib

Gradient Descent (GD)

What is a Gradient?

Think of the loss function as a mountain. The gradient tells you:

- Which direction is uphill (increasing loss)
- So we move **the opposite way** to go downhill and minimize loss.

In multi-dimensional space (D parameters):

$$\nabla L = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_D}]$$

Each optimizer **updates weights differently based on the gradient**.

How It Works

Gradient descent is an iterative optimization algorithm used to minimize a cost (loss) function(convex best).

Steps:

1. Initialize x_0 $(starting\ point\ on\ curve)$ randomly

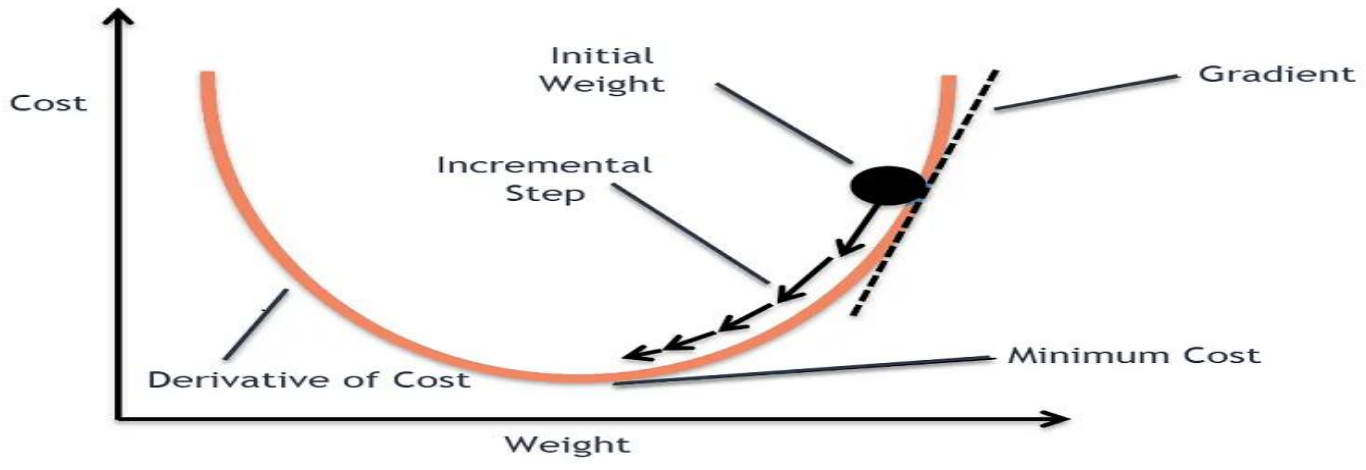
2. Loop until convergence

3. we **update** the weights iteratively:

$$w^{(t+1)} = w^{(t)} - \eta \nabla_w J(w^{(t)})$$

where:

- η is the **learning rate** (step size) -> hyperparameter,
- $\nabla_w J(w^{(t)})$is the gradient (derivative) at the current weights.



Variants

Batch GD

- Uses the **entire dataset** for one update
- Very stable, but slow

Stochastic GD (SGD)

- Updates using **one sample at a time**
- Very noisy but fast
- May bounce around minima

Mini-Batch GD

- Uses small subsets (16, 32, 64, …)
- Standard in deep learning
- Best balance of speed + stability

The Normal Equation: A "One-Shot" Solution

While Gradient Descent reaches the minimum by taking small, incremental steps down the cost function "bowl," the **Normal Equation** is a mathematical shortcut that jumps straight to the minimum in a single move.

Instead of iterating multiple times, you compute the optimal parameters **analytically** using linear algebra.

The Formula

$$\theta = (X^T X)^{-1} X^T y$$

Where:

- X is the design matrix (input features)
- y is the vector of target values (labels)
- θ is the vector of optimal parameters you want to find

Why don't we always use it?

At first glance, the Normal Equation looks ideal:

- No learning rate α to tune
- No iterative optimization

However, it has a major drawback: **scalability**.

Computing the matrix inverse $(X^T X)^{-1}$ is computationally expensive, with a time complexity of approximately $O(n^3)$.

- With **100 features**, it's fast and practical.
- With **100,000 features**, it becomes infeasible — your computer will struggle or even freeze.

Learning Rate (LR)

What It Is

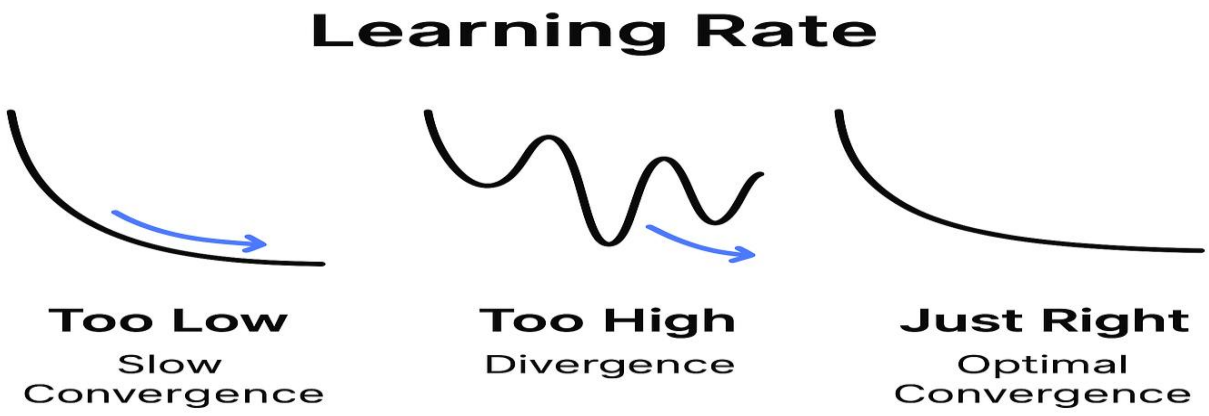
- The **learning rate** controls how much the model's parameters (weights) are updated during each training step.
- It's the **step size** in the optimization process (usually gradient descent).

Why It Matters

- If **too high** → the model overshoots the minimum, loss may bounce around or diverge.
- If **too low** → the model learns very slowly, may get stuck in local minima.
- A **good learning rate** balances speed and stability, converging efficiently to a low loss.

Techniques for Managing LR

1. **Fixed LR** → one constant value throughout training.
2. **Learning Rate Schedules** → gradually change LR:
    - **Step decay**: reduce LR after fixed epochs.
    - **Exponential decay**: shrink LR continuously.
    - **Cosine annealing**: smoothly decay and restart.
3. **Adaptive Methods** → optimizers like Adam, RMSprop adjust LR per parameter automatically.
4. **Learning Rate Finder** → start with very small LR, increase exponentially, find the "sweet spot."



4. Regularization in Linear Regression

Regularization helps prevent overfitting by penalizing large weights.

(a) Ridge Regression (L2 Regularization)

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - (w^T x^{(i)} + b))^2 + \lambda \sum_{j=1}^{n} w_j^2$$

- **Penalizes large weights.**
- **Keeps all features but shrinks coefficients.**

(b) Lasso Regression (L1 Regularization)

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\left((y^{(i)} - (w^T x^{(i)} + b)\right)^2 + \lambda \sum_{j=1}^{n} |w_j|$$

- **Forces some weights to become zero → great for feature selection.**

(c) Elastic Net (Combination of L1 and L2)

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - (w^T x^{(i)} + b))^2 + \lambda_1 \sum_{j=1}^{n} |w_j| + \lambda_2 \sum_{j=1}^{n} w_j^2$$

- **Elastic Net shrinks coefficients and also sets some to zero**
- **Can keep groups of correlated features together while still applying sparsity**

5. Code Example of Linear Regression

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
import numpy as np

# Example data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2.3, 4.1, 6.2, 8.1, 10.5])  # target values

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mse)   # RMSE = sqrt(MSE)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R² Score:", r2)
```

---

6. Parameters of Linear Regression

Parameter	Type	Default	Description
fit_intercept	bool	True	Whether to calculate the intercept bb. If False, model assumes data is already centered.
copy_X	bool	True	Whether to copy the feature matrix X. If False, X may be overwritten.
n_jobs	int or None	None	Number of jobs to run in parallel for computation (None = 1 core, -1 = all cores).
positive	bool	False	If True, forces coefficients w to be non-negative.
tol	float	1e-6	Stopping tolerance.

7. Code Example Ridge Regression

```
from sklearn.linear_model import Ridge
import numpy as np

# Sample dataset
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([1, 2, 3, 3.5, 5])

# Ridge with alpha(lambda)=1.0
ridge = Ridge(alpha=1.0, fit_intercept=True)
ridge.fit(X, y)

print("Ridge Coefficients:", ridge.coef_)
print("Ridge Intercept:", ridge.intercept_)
```

---

8. Parameters of Ridge Regression

Parameter	Type	Default	Description
alpha	float	1.0	Regularization strength (λ\lambda).
fit_intercept	bool	True	Whether to fit the bias term.
copy_X	bool	True	Whether to copy input data.
max_iter	int	None	Max iterations for solver.
tol	float	1e-4	Stopping tolerance.
solver	str	'auto'	Solver method (auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga', 'lbfgs').
positive	bool	False	Enforces non-negative weights.

9. Code Example Lasso Regression

```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1, fit_intercept=True)
lasso.fit(X, y)

print("Lasso Coefficients:", lasso.coef_)
print("Lasso Intercept:", lasso.intercept_)
```

---

10. Parameters of Lasso Regression

Parameter	Type	Default	Description
alpha	float	1.0	Regularization strength (λ\lambda).
fit_intercept	bool	True	Whether to fit intercept.
copy_X	bool	True	Copy input data.
max_iter	int	1000	Maximum iterations.
tol	float	1e-4	Tolerance for stopping.
selection	str	'cyclic'	Coordinate descent rule (cyclic or random).
positive	bool	False	Forces positive weights.
Warm_start	bool	False	When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.
Precompute	bool	False	Whether to use a precomputed Gram matrix to speed up calculations. The Gram matrix can also be passed as argument. For sparse input this option is always False to preserve sparsity.

11. Code Example Elastic Net

```
from sklearn.linear_model import ElasticNet

elastic = ElasticNet(alpha=0.1, l1_ratio=0.5, fit_intercept=True)
elastic.fit(X, y)

print("Elastic Net Coefficients:", elastic.coef_)
print("Elastic Net Intercept:", elastic.intercept_)
```

---

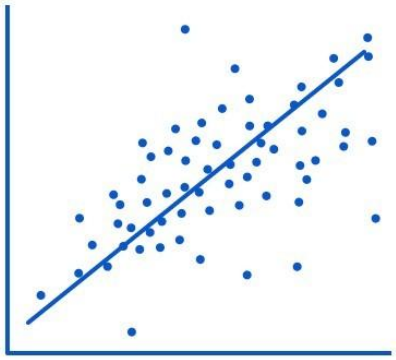
12. Parameters of Elastic Net

Parameter	Type	Default	Description
alpha	float	1.0	Regularization strength (λ\lambda).
l1_ratio	float	0.5	Mix between L1 (Lasso) and L2 (Ridge). l1_ratio = 0 the penalty is an L2 penalty. For l1_ratio = 1 it is an L1 penalty.
fit_intercept	bool	True	Whether to fit bias.
copy_X	bool	True	Copy input data.
max_iter	int	1000	Max iterations.
tol	float	1e-4	Stopping tolerance.
selection	str	'cyclic'	Coordinate descent rule.
positive	bool	False	Forces coefficients ≥ 0.
Warm_start	bool	False	When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.
Precompute	bool	False	Whether to use a precomputed Gram matrix to speed up calculations. The Gram matrix can also be passed as argument. For sparse input this option is always False to preserve sparsity.

13. After Training Attributes of Linear Regression

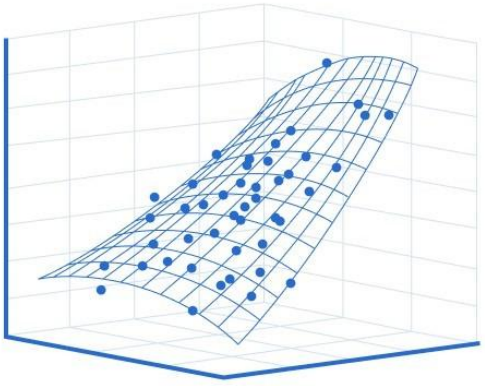
Attribute	Description
coef_	Coefficients (weights) of the model.
intercept_	Bias term (intercept).
rank_	Rank of the feature matrix X.
singular_	Singular values of X.
n_features_in_	Number of input features.
feature_names_in_	Names of features seen during fit (if input is DataFrame).

Simple Linear Regression



One Feature

Multiple Linear Regression



Multiple independent Features

Polynomial Regression

1. Formula

For a **single feature x** and polynomial degree d:

$$\hat{y} = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$$

For **multiple features**, we can create polynomial combinations (cross-terms).

---

2. Cost Function

Same as Linear Regression (**Mean Squared Error**):

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - (w^T x^{(i)} + b))^2$$

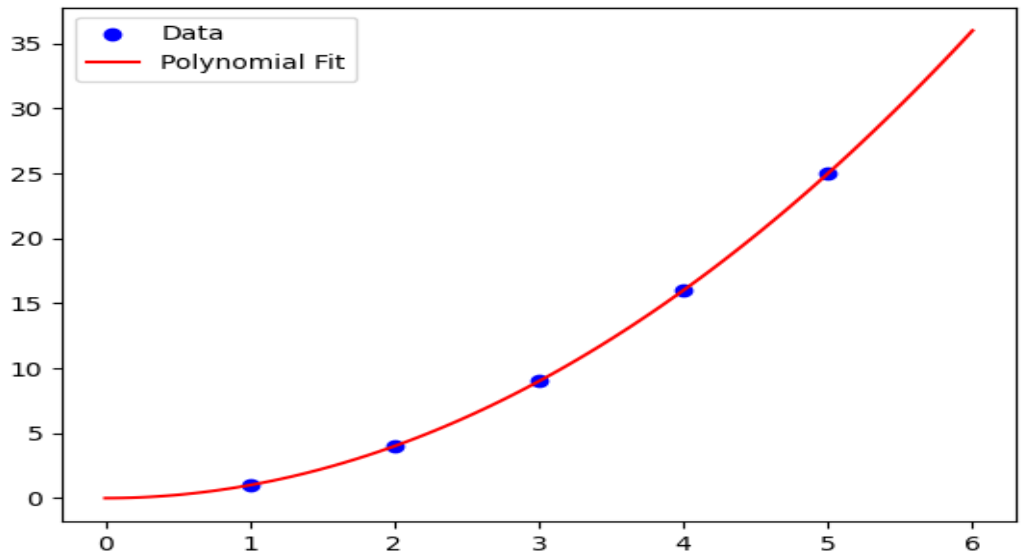
---

3. Regularization

Polynomial regression can easily overfit when the degree is high.
To prevent this, we often apply Ridge/Lasso/ElasticNet regularization.

4. Code Example

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Sample data
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([1, 4, 9, 16, 25])  # quadratic relation

# Transform features into polynomial (degree 2)
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X)

# Fit model
model = LinearRegression()
model.fit(X_poly, y)

# Predict
X_test = np.linspace(0, 6, 100).reshape(-1, 1)
X_test_poly = poly.transform(X_test)
y_pred = model.predict(X_test_poly)

# Plot
plt.scatter(X, y, color="blue", label="Data")
plt.plot(X_test, y_pred, color="red", label="Polynomial Fit")
plt.legend()
plt.show()

print("Coefficients:", model.coef_)
print("Intercept:", model.intercept_)
```



5. Parameters of Polynomial Feature

Parameter	Type	Default	Description
degree	int	2	The degree of the polynomial features to be generated. Example: degree=2 → x, x^2.
interaction_only	bool	False	If True, only interaction features are produced (e.g., x1x2) and no powers like x1^2.
include_bias	bool	True	If True, includes a bias column of ones (useful for intercept in regression). If False, removes it.
order	{'C','F'}	'C'	Order of output array: 'C' (row-major) or 'F' (column-major). Controls how polynomial terms are arranged.

---

6. Attributes after fitting

Attribute	Description
powers_	Array of exponents for each output feature. Example: for 2D with degree=2 → [[0,0],[1,0],[0,1],[2,0],[1,1],[0,2]].
n_input_features_	Number of original input features.
n_output_features_	Number of generated polynomial + interaction features.

Classification

Logistic Regression

Unlike **Linear Regression** (which predicts continuous values), **Logistic Regression** is used for **classification problems** (e.g., predicting 0/1, yes/no, spam/not spam).

It predicts the **probability** that a data point belongs to a class using the **logistic (sigmoid) function**:

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

Where:

- $z = \mathbf{w}^T\mathbf{x} + b$
- Output σ(z) is always between **0 and 1**.

---

1. Logistic Regression Model

$$P(y=1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b) = \sigma(z)$$

Decision rule:

$$\hat{y} = 0 \; if \sigma(z) < 0.5$$

$$\hat{y} = 1 \; if \; \sigma(z) \geq 0.5$$

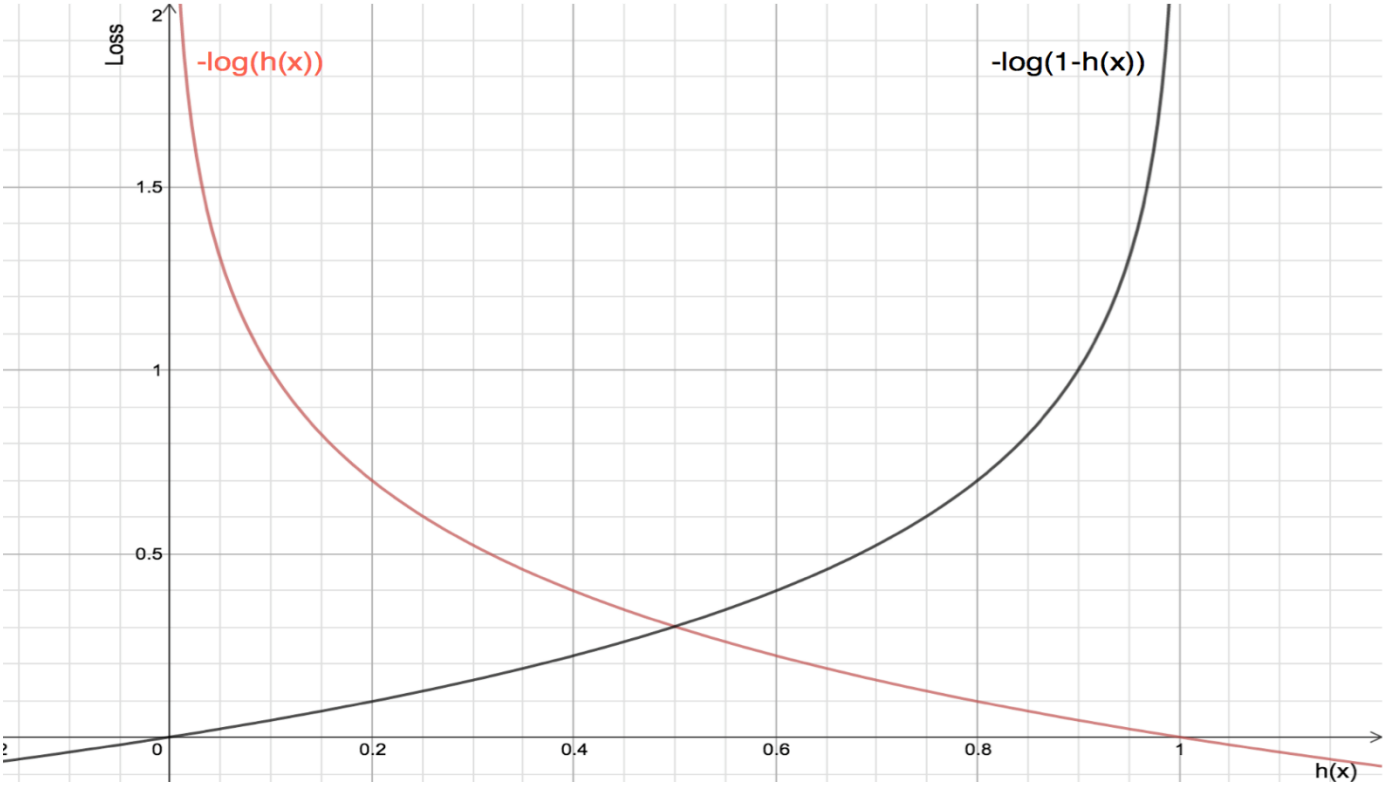
2. Cost Function (Log Loss / Binary Cross-Entropy)

Linear regression uses MSE, but for classification, it doesn't work well. Instead, logistic regression minimizes the log loss:

$$J(\mathrm{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

where:

- $y^{(i)}$ is the true label (0 or 1).
- $\hat{y}^{(i)}$ is the predicted probability.
- m is the number of training samples.



3. Regularization

To prevent overfitting, there is built-in regularization:

- **L1 Regularization (Lasso):** Encourages sparsity in weights.
- **L2 Regularization (Ridge):** Shrinks weights smoothly.
- **Elastic Net** regularization (mix of L1 & L2).

4. Multi-Class Logistic Regression

One-vs-Rest (OvR)

Idea: Break a multi-class problem into multiple binary problems.

- If you have **K classes**, train **K binary classifiers**:
    - Class 1 vs all others
    - Class 2 vs all others
    - …
    - Class K vs all others
- During prediction:
    - Each classifier outputs a probability.
    - The class with the **highest probability** is chosen.

Example (3 classes: Cat, Dog, Rabbit):

- Model 1: Cat vs (Dog+Rabbit)
- Model 2: Dog vs (Cat+Rabbit)
- Model 3: Rabbit vs (Cat+Dog)

If you input a picture of a Dog:

- Model 1 says "20% Cat"
- Model 2 says "90% Dog"
- Model 3 says "10% Rabbit"

→ Final prediction = Dog

✅ Pros: Works with any binary classifier, simple, fast.
❌ Cons: Classifiers can disagree, probabilities may overlap (less elegant).

Multinomial Logistic Regression (a.k.a. Softmax Regression)

Idea: Train one model for all classes at once.

- Uses a **softmax function** instead of sigmoid:
- Ensures that all probabilities sum to 1.
- Directly gives probabilities for each class.

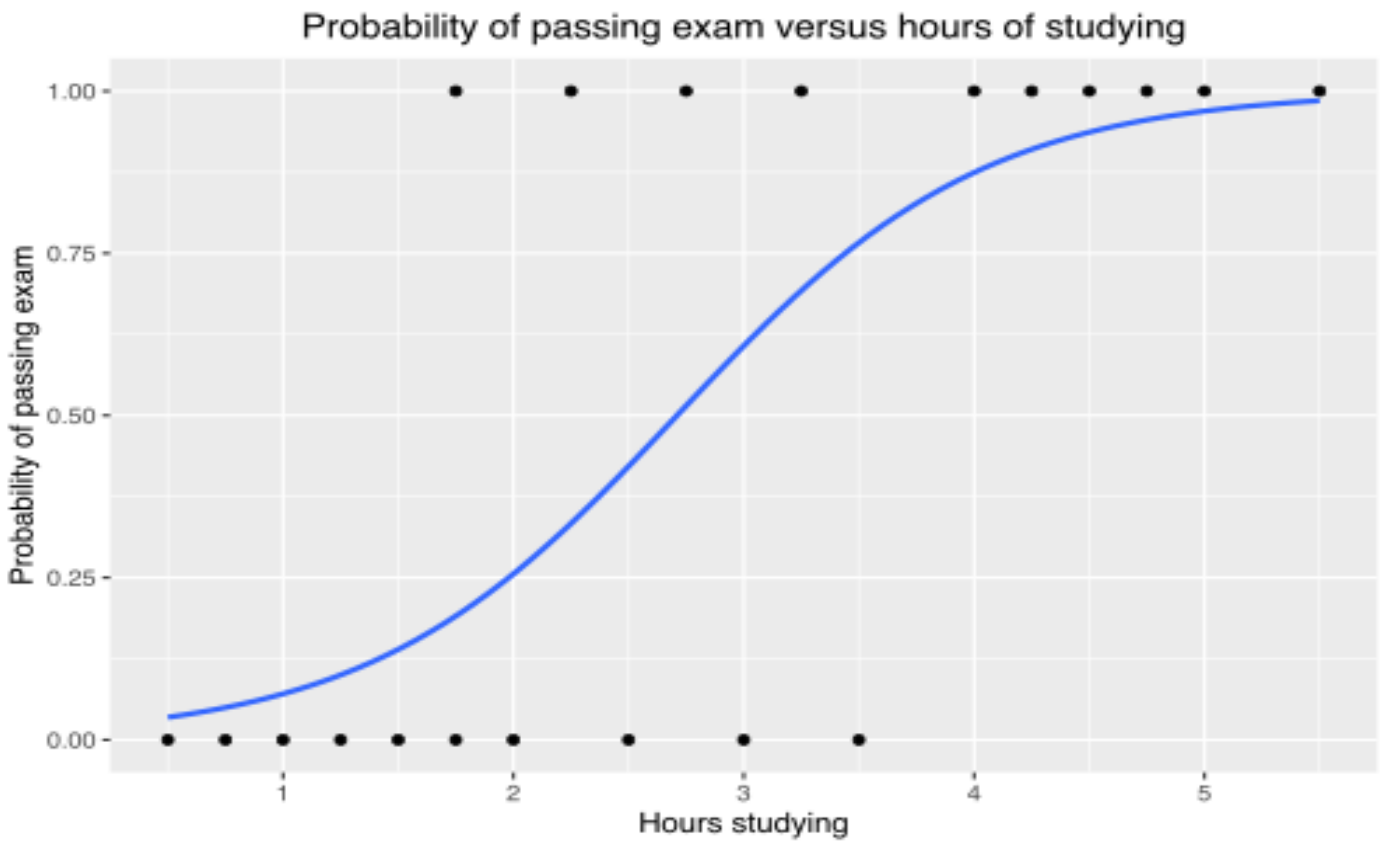
Example (3 classes: Cat, Dog, Rabbit):

For an input image:

- P(Cat) = 0.05
- P(Dog) = 0.90 ✅
- P(Rabbit) = 0.05

→ Final prediction = Dog

✅ Pros: Elegant, consistent probabilities, best for large datasets.
❌ Cons: Requires solvers that support multinomial (not liblinear).



5. Code Example (Binary Classification)

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import numpy as np

# Example data (X: features, y: binary labels)
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([0, 0, 0, 1, 1])

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Logistic Regression Model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)  # probability estimates

# Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

6. Parameter Table for Logistic Regression

Parameter	Type	Default	Description
penalty	{'l1', 'l2', 'elasticnet', 'none'}	'l2'	Specifies the regularization norm.
dual	bool	False	Solves dual or primal optimization. Only for 'liblinear' solver with penalty='l2'.
tol	float	1e-4	Tolerance for stopping criteria.
C	float	1.0	Inverse of regularization strength (λ=1/C\lambda = 1/C). Smaller C → stronger regularization.
fit_intercept	bool	True	Whether to add an intercept (bias term).
intercept_scaling	float	1	Used only when solver='liblinear' and fit_intercept=True.
class_weight	dict, "balanced" or None	None	Adjusts weights inversely proportional to class frequencies.
random_state	int, RandomState or None	None	Controls randomness for solvers and shuffling.
solver	{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}	'lbfgs'	Algorithm to use for optimization.
max_iter	int	100	Maximum number of iterations for solvers.
multi_class	{'auto', 'ovr', 'multinomial'}	'auto'	'ovr': one-vs-rest, 'multinomial': true multinomial logistic regression.
warm_start	bool	False	Reuse solution from previous fit to speed up convergence.
n_jobs	int or None	None	Number of CPU cores to use (only for 'liblinear').
l1_ratio	float, optional	None	For penalty='elasticnet', balance between L1 (0–1).

- **lbfgs, newton-cg, sag, saga → support multinomial.**
- **liblinear → supports only binary and OvR, not multinomial.**

When you set multi_class='auto' (the default), scikit-learn looks at:

1. If there are only **2 unique classes** → it automatically runs **binary logistic regression**.
2. If there are **>2 classes** → it chooses **multinomial logistic regression** *if the solver supports it*, otherwise falls back to **OvR**.

Both Classification and Regression

Decision Tree

A **Decision Tree** is a **flowchart-like structure**:

- **Root Node**: Represents the entire dataset, the first split.
- **Internal Nodes**: Represent decision points (tests on features).
- **Branches**: Represent the outcome of the test at a node.
- **Leaves**: Terminal nodes that give the final prediction.
    - For **classification**: each leaf corresponds to a class label.
    - For **regression**: each leaf corresponds to a numerical value (mean of samples).

Example:

- Root Node: *Is age > 30?*
- Left Branch: *Yes → Income?*
- Right Branch: *No → Predict "Student"*

1. Decision Tree Algorithms

There are three major algorithms used to construct decision trees:

1. Accuracy

- Splits based on **the accuracy** of the classification

2. ID3 (Iterative Dichotomiser 3)

- Uses **Entropy** and **Information Gain** to decide splits.
- Chooses the feature with **maximum information gain**.

3. CART (Classification and Regression Trees)

- The most widely used (implemented in scikit-learn).
- Uses **Gini Impurity** (classification) or **MSE** (regression).
- Produces a **binary tree** (always 2 branches per node).

4. C4.5 (successor of ID3)

- Similar to ID3 but uses **Gain Ratio** (normalizes Information Gain).
- Better for multi-class classification.

2. How ID3 Works

Entropy

Entropy measures **impurity** (uncertainty) in the dataset.

$$Entropy(S) = -\sum_{i=1}^{C} p_i \log_2(p_i)$$

where p_i is the proportion of samples in class i.

- Low entropy → high purity (samples mostly in one class).
- High entropy → mixed classes.

Average Information (Expected Entropy after split)

The **average entropy of the child nodes** resulting from a split.

$$I(A) = \sum \frac{p_i + n_i}{p + n} \cdot Entropy(A)$$

Information Gain

The **expected reduction in entropy** achieved by splitting the dataset on a particular attribute.

$$Gain = Entropy(S) - I(A)$$

where:

- S = dataset
- A = feature used to split

The feature with **highest IG** is chosen for splitting (in ID3).

3. How CART Works

1. **Start at the root node** with all training data.
2. For each feature and possible split value:
    - Partition the data into **two subsets** (left branch, right branch).
    - Calculate the **impurity** of the split.
    - Choose the split that minimizes impurity.
3. Repeat recursively on each subset until:
    - Maximum depth is reached, OR
    - Node has fewer than min_samples_split, OR
    - Node is pure (all samples same class).

Example 1: Entropy, Information Gain (ID3)

Suppose we have a dataset for predicting whether to play tennis based on **Weather**.

Weather	Play
Sunny	No
Sunny	No
Overcast	Yes
Rain	Yes
Rain	No
Overcast	Yes

👉 Target variable: **Play (Yes/No)**

Step 1: Calculate Entropy of the target (Play)

We have:

- Yes = 3
- No = 3

$$p(Yes) = \frac{3}{6} = 0.5, \quad p(No) = \frac{3}{6} = 0.5$$

Entropy:

$$H(Play) = -\sum p_i \log_2(p_i) = -[0.5\log_2(0.5) + 0.5\log_2(0.5)]$$

$$H(Play) = 1.0$$

Step 2: Entropy after splitting on Weather

- **Sunny** → 2 samples, all **No** → Entropy = 0
- **Overcast** → 2 samples, all **Yes** → Entropy = 0
- **Rain** → 2 samples (Yes=1, No=1) →

$$H(Rain) = -[0.5\log_2(0.5) + 0.5\log_2(0.5)] = 1.0$$

Step 3: Weighted Average Entropy

$$H(Weather) = \frac{2}{6}(0) + \frac{2}{6}(0) + \frac{2}{6}(1) = \frac{2}{6}(1) = 0.333$$

Step 4: Information Gain

$$IG(Weather) = H(Play) - H(Weather) = 1.0 - 0.333 = 0.667$$

✅ So, splitting on **Weather** reduces uncertainty the most with **IG = 0.667**.

Example 2: CART (Gini Index)

Now let's compute with **Gini** for the same dataset.

Step 1: Gini of target (Play)

$$Gini(Play) = 1 - \sum p_i^2 = 1 - (0.5^2 + 0.5^2) = 0.5$$

Step 2: Gini after splitting on Weather

- **Sunny** → 2 samples (No=2, Yes=0) →

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

- **Overcast** → 2 samples (Yes=2, No=0) →

$$Gini(Overcast) = 0$$

- **Rain** → 2 samples (Yes=1, No=1) →

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

Step 3: Weighted Average Gini

$$Gini(Weather) = \frac{2}{6}(0) + \frac{2}{6}(0) + \frac{2}{6}(0.5) = 0.167$$

Step 4: Gini Gain (aka impurity decrease)

$$\Delta Gini = Gini(Play) - Gini(Weather) = 0.5 - 0.167 = 0.333$$

✅ CART would also choose **Weather** here, but using **Gini instead of Entropy**.

4. Key Differences between Cart and ID3

Feature	ID3	CART
Splits	Multiway (can create >2 branches)	Binary (always 2 branches)
Criterion	Entropy / Info Gain	Gini, Entropy (classification), MSE/MAE (regression)
Output	Classification only	Classification + Regression
Normalization	Uses Gain Ratio in C4.5 (fix for bias)	No need, works well with binary splits

---

5. How Splitting Works With Continuous Values

Suppose we have feature:

Age: [18, 22, 25, 30, 40]

The tree considers splits like:

Age < 20?

Age < 23.5?

Age < 27.5?

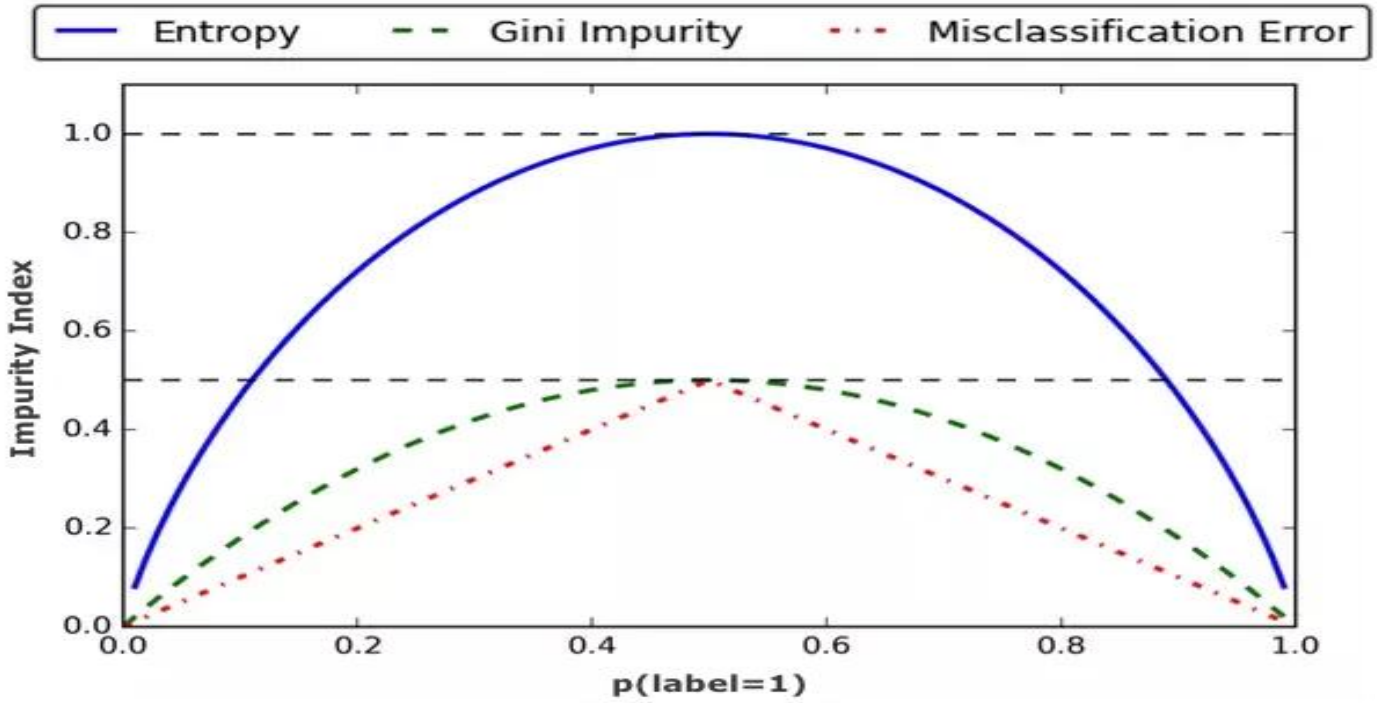
Age < 35?

These split points are typically:

Midpoints between sorted values.

6. When to stop splitting

- When a node is 100% pure
- When splitting a node will exceed max depth
- When improvements in purity score are below a threshold
- When the number of examples in a node is below a threshold



7. Cost Functions in Decision Trees

Depending on whether we're solving classification or regression:

Classification Cost

- **Cart (Gini Index)**:

$$Gini(S) = 1 - \sum_{i=1}^{C} p_i^2$$

- **Entropy (used in ID3)**:

$$Cost(S) = Entropy(S)$$

Regression Cost (Cart only)

- **Mean Squared Error (MSE)** at node:

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

- **Mean Absolute Error (MAE)** also possible:

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

8. Code Example

```
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.datasets import load_iris, load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error

# Classification Example (Iris Dataset)
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

clf = DecisionTreeClassifier(criterion="entropy", max_depth=3, random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Classification Accuracy:", accuracy_score(y_test, y_pred))

# Regression Example (Diabetes Dataset)
X, y = load_diabetes(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

reg = DecisionTreeRegressor(criterion="squared_error", max_depth=3, random_state=42)
reg.fit(X_train, y_train)
y_pred = reg.predict(X_test)
print("Regression MSE:", mean_squared_error(y_test, y_pred))
```

9. DecisionTreeClassifier() Parameters

Parameter	Type	Default	Description	Possible Values
criterion	str	"gini"	Function to measure split quality	"gini", "entropy", "log_loss"
splitter	str	"best"	Strategy to choose split	"best", "random"
max_depth	int or None	None	Maximum depth of tree	Any int ≥ 1, or None (unlimited)
min_samples_split	int or float	2	Minimum samples required to split an internal node	int ≥ 2, or float in (0,1.0] (fraction of samples)
min_samples_leaf	int or float	1	Minimum samples required at a leaf node	int ≥ 1, or float in (0,1.0]
min_weight_fraction_leaf	float	0.0	Minimum weighted fraction of samples per leaf	Float in [0, 0.5]
max_features	int, float, str, or None	None	Number of features to consider at each split	None, int (absolute), float (fraction), "sqrt", "log2"
random_state	int, RandomState, or None	None	Controls randomness (important if splitter="random")	Any int, None
max_leaf_nodes	int or None	None	Limits number of leaf nodes	int ≥ 2, or None
min_impurity_decrease	float	0.0	Split only if impurity decrease ≥ value	Float ≥ 0
class_weight	dict, "balanced", or None	None	Weights associated with classes	None (all weights = 1), "balanced", {class_label: weight}
ccp_alpha	non-negative float	0.0	Complexity parameter for Minimal Cost-Complexity Pruning	Float ≥ 0

Parameter	Type	Default	Description	Possible Values
monotonic_cst	array-like of int of shape (n_features,) or None	None	Monotonicity constraints for features (experimental)	-1 (decreasing), 0 (no constraint), 1 (increasing), None

---

10. DecisionTreeRegressor() Parameters

Parameter	Description
criterion	"squared_error" (default), "friedman_mse", "absolute_error", "poisson".
splitter, max_depth, min_samples_split, etc.	Same as classifier.
ccp_alpha	Same pruning regularization.

- A **Decision Tree Regressor** is a model that:
  Predicts a **continuous numerical value** by recursively splitting the data into regions and assigning a prediction to each region.
  Instead of predicting classes, it predicts numbers(certain numbers put in leaf nodes from training).

Random Forest

A **Random Forest** is an **ensemble learning algorithm** that builds multiple **Decision Trees** and combines their outputs to improve accuracy and reduce overfitting.

- For **classification** → takes a **majority vote** across trees.
- For **regression** → takes the **average prediction** across trees.

It's based on two core ideas:

1. **Bagging (Bootstrap Aggregating):** Train each tree on a different random sample of the data.
2. **Feature Randomness:** At each split, only a random subset of features is considered.

---

1. How Random Forest Works

1. Draw **bootstrap samples** from the training set (with replacement).
2. For each sample, grow a **decision tree**:
    - At each node, instead of considering all features, consider a **random subset**.
    - Split using the best feature among those.
3. Repeat for N trees (e.g., 100).
4. Aggregate predictions:
    - Classification → majority voting.
    - Regression → average.

---

2. Why Random Forest?

- **Decision Trees** → low bias, high variance (overfitting).
- **Random Forest** → reduces variance by averaging many diverse trees.
- Improves **generalization** without requiring heavy tuning.

3. How Random Forest Works (step-by-step example)

Let's say we have this dataset to predict whether a student **passes or fails** an exam:

Hours Studied	Attendance	Pass (Y/N)
5	80	Pass
1	40	Fail
4	70	Pass
2	30	Fail
3	65	Pass

A single Decision Tree might overfit: e.g., if Hours Studied > 2.5 → predict Pass.
A Random Forest builds many trees on different bootstrapped samples (random subsets of data), using different subsets of features.

- **Tree 1** might split on *Hours Studied*.
- **Tree 2** might split on *Attendance*.
- **Tree 3** might split on both but in a different order.

When predicting a new student (say Hours=2, Attendance=60):

- Tree 1 → Pass
- Tree 2 → Fail
- Tree 3 → Pass

Final Random Forest prediction = Majority Vote = Pass

4. Cost Function

Each individual tree uses the **same cost function as CART**:

- Classification → Gini or Entropy
- Regression → MSE, MAE (Gini)

But Random Forest minimizes **out-of-bag error (OOB)** as an ensemble measure.

---

5. Key Concepts

- **Bootstrap sampling:** each tree sees ~63% of training data, rest is "out-of-bag" (OOB).
- **Out-of-Bag Error:** can be used as an unbiased estimate of test error (like built-in cross-validation).
- **Feature Importance:** measured by how much a feature reduces impurity across all trees.

---

6. Code Example RandomForestClassifier

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load dataset
X, y = load_iris(return_X_y=True)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Random Forest Classifier
rf = RandomForestClassifier(
    n_estimators=100,     # number of trees
    criterion="gini",     # split function
    max_depth=None,       # grow full trees
    random_state=42,
    oob_score=True        # use out-of-bag score
)
rf.fit(X_train, y_train)

# Predictions
y_pred = rf.predict(X_test)
# Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print("OOB Score:", rf.oob_score_)
print(classification_report(y_test, y_pred))
print("Feature Importances:", rf.feature_importances_)
```

7. Code Example RandomForestRegressor

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Load dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Random Forest Regressor
rf_reg = RandomForestRegressor(
    n_estimators=100,
    criterion="squared_error",  # regression cost function
    max_depth=None,
    random_state=42,
    oob_score=True
)

rf_reg.fit(X_train, y_train)

# Predictions
y_pred = rf_reg.predict(X_test)

# Evaluation
print("MSE:", mean_squared_error(y_test, y_pred))
print("R² Score:", r2_score(y_test, y_pred))
print("OOB Score:", rf_reg.oob_score_)
print("Feature Importances:", rf_reg.feature_importances_)
```

8. RandomForestClassifier() Parameters

Parameter	Type	Default	Description	Possible Values
n_estimators	int	100	Number of trees in the forest	int ≥ 1
criterion	str	"gini"	Split quality function	"gini", "entropy", "log_loss"
max_depth	int or None	None	Max depth of each tree	int ≥ 1, or None
min_samples_split	int or float	2	Minimum samples required to split a node	int ≥ 2 or float (0,1]
min_samples_leaf	int or float	1	Minimum samples required in a leaf	int ≥ 1 or float (0,1]
min_weight_fraction_leaf	float	0.0	Min weighted fraction of samples per leaf	Float [0, 0.5]
max_features	int, float, str, or None	"sqrt"	Number of features considered per split	int, float, "sqrt", "log2", None
max_leaf_nodes	int or None	None	Maximum number of leaf nodes	int ≥ 2, or None
min_impurity_decrease	float	0.0	Split if impurity decrease ≥ value	Float ≥ 0
bootstrap	bool	True	Whether to use bootstrap samples	True / False
oob_score	bool	False	Whether to use out-of-bag samples to estimate generalization	True / False
n_jobs	int	None	Number of CPU cores used	None, -1 (all cores), or int
random_state	int, RandomState, or None	None	Controls randomness	Any int, None
verbose	int	0	Controls verbosity when fitting	int (0 = silent)
warm_start	bool	False	If True, reuse solution of previous fit and add more estimators	True / False

Parameter	Type	Default	Description	Possible Values
class_weight	dict, "balanced", or None	None	Weights associated with classes	None, "balanced", dict
ccp_alpha	non-negative float	0.0	Complexity parameter for pruning	Float ≥ 0
max_samples	int, float, or None	None	Number of samples for each tree if bootstrap = True	int, float in (0,1], or None
monotonic_cst	array-like or None	None	Monotonic constraints for features (experimental)	-1, 0, 1

---

9. RandomForestRegressor() Parameters

The same as the classifier but different Criterion options {"squared_error", "absolute_error", "friedman_mse", "poisson"}, and there is no class weight since Regression has continuous targets

SVM

Support Vector Machine (SVM) is a **supervised learning algorithm** used for **classification** (most common) and **regression (SVR)**.

It works by:

- Mapping data into a high-dimensional space.
- Finding the **optimal hyperplane** that separates classes with the **maximum margin**.
- The **margin** is the distance between the **decision boundary (hyperplane)** and the closest data points (called **support vectors**).
  SVM tries to find the hyperplane that maximizes this margin → gives better generalization.
- Uses **support vectors** (critical data points near the boundary) to define this hyperplane.

---

1. How it Works

1. Linear SVM (Linearly separable data)

- Finds the line (2D), plane (3D), or hyperplane (n-D) that best separates two classes.
- Margin = distance between hyperplane and nearest data points (support vectors).
- Goal: Maximize the margin.

Equation of decision boundary:

$$f(x) = w \cdot x + b$$

Prediction rule:

$$\hat{y} = \text{sign}(w \cdot x + b)$$

---

2. Non-linear SVM (Kernel Trick)

- When data isn't linearly separable, SVM applies a **kernel function** to project data into higher dimensions.
- Common kernels:
  - **Linear**
  - **Polynomial**
  - **RBF (Radial Basis Function)**
  - **Sigmoid**

3. Soft Margin & Regularization

- Real-world data has noise → allow misclassification with penalty.
- Cost function (for classification):

$$J(w) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i$$

Where:

- C: Regularization parameter (trade-off between margin size and misclassification).
- ξ: Slack variable (error penalty).

---

2. SVM Error Function (Classification)

Error formula = C * (Classification Error) + (Distance Error)

---

3. The Kernel Trick (RBF)

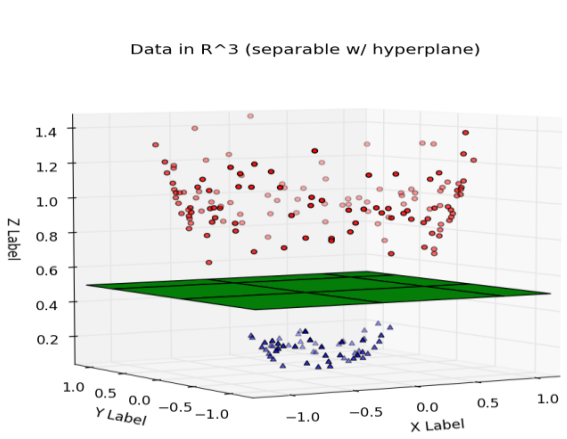
Sometimes data is not linearly separable in the original feature space. RBF is used as a similarity function.

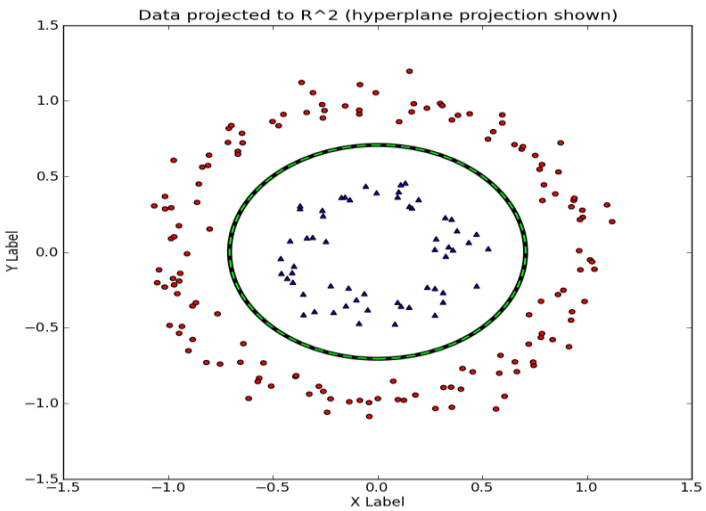
The trick:

- Map data into **higher-dimensional space** where a linear separation is possible.
- But explicitly transforming to higher dimensions is expensive.

Solution → Kernel Trick:

- Use a **kernel function K(x, x')** that computes the inner product in the higher-dimensional space **without explicitly mapping**.





4. Log Loss vs Hinge Loss

Both are **loss functions for classification**, but with different philosophies.

Log Loss (used in Logistic Regression)

- Outputs **probabilities**.
- Penalizes **wrong but confident predictions heavily**.
- Smooth, differentiable everywhere.

Hinge Loss (used in SVM)

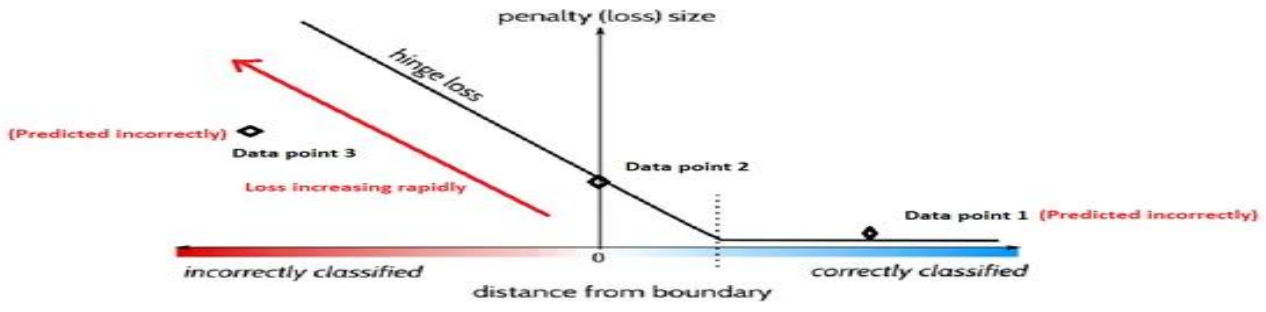
- If correctly classified **and outside the margin** → loss = 0.
- If inside the margin (even if correct) → some loss.
- If misclassified → big loss.

Difference

- **Log Loss**: cares about probability calibration (output probabilities).
- **Hinge Loss**: only cares about being on the right side of the margin, not probabilities.
- **SVM → Hinge Loss** is preferred because it enforces the **max-margin principle**.
- **Logistic regression** can give well-calibrated probabilities, while **SVM** gives strong boundaries but not probabilities (unless we apply Platt scaling).

Why Hinge Loss Fixes Issues of Log Loss

- **Log loss** keeps punishing even when the prediction is confidently correct (p = 0.99).
- **Hinge loss** ignores well-classified examples outside the margin → focuses learning on **hard/misclassified points**.
- This makes SVM more robust to noise and helps generalization by enforcing **maximum margin** rather than overfitting to probability estimates.



5. SVM for Regression (SVR)

Instead of classification, SVR tries to fit a function within a margin of tolerance (epsilon).

Cost function:

$$J(w) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} L_{\varepsilon}(y_i - f(x_i))$$

Where L_{ε} is the epsilon-insensitive loss (errors smaller than ϵ\epsilonϵ are ignored).

---

6. Code Example

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load dataset (Iris)
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train SVM with RBF kernel
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale')
svm_model.fit(X_train, y_train)

# Predict
y_pred = svm_model.predict(X_test)

# Evaluate
print("Accuracy:", accuracy_score(y_test, y_pred))
```

7. Parameters Table

Parameter	Type	Default	Description
C	float	1.0	Regularization parameter. Higher C = less tolerance for errors.
kernel	str	'rbf'	Type of kernel: 'linear', 'poly', 'rbf', 'sigmoid', or callable.
degree	int	3	Degree of polynomial kernel (if poly).
gamma	{'scale','auto'} or float	'scale'	Kernel coefficient. High gamma → tighter decision boundary.
coef0	float	0.0	Independent term in poly/sigmoid kernel.
shrinking	bool	True	Whether to use shrinking heuristic.
probability	bool	False	Enable probability estimates (slower).
tol	float	1e-3	Tolerance for stopping criterion.
cache_size	float	200	Cache size (MB) for kernel computation.
class_weight	dict or 'balanced'	None	Handles imbalanced classes.
verbose	bool	False	Print training logs.
max_iter	int	-1	Maximum iterations (-1 = no limit).
decision_function_shape	str	'ovr'	'ovr' (One-vs-Rest) or 'ovo' (One-vs-One).
break_ties	bool	False	Break ties in multiclass decisions.
random_state	int	None	Controls randomness (only for probability=True).

Gradient Boosting

Gradient Boosting is an **ensemble learning method** that builds a model in **stages** (like boosting), where each new model corrects the errors (residuals) of the previous one.

- Base learners: usually **shallow decision trees** (weak learners).
- Unlike Bagging (Random Forest), boosting builds trees **sequentially** (not independently).
- Uses **gradient descent** on the loss function to minimize errors.

---

1. Cost Function

- **Regression (MSE loss)**:

→ Learners fit residuals directly.

- **Classification (Log Loss)**

→ Learners fit the difference between actual labels and predicted probabilities.

---

2. Code Example: Regression with Gradient Boosting

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import GradientBoostingRegressor

# Load data
X, y = fetch_california_housing(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train GB model
gbr = GradientBoostingRegressor(
    n_estimators=200,
    learning_rate=0.05,
    max_depth=3,
    random_state=42
)
gbr.fit(X_train, y_train)

# Predictions
y_pred = gbr.predict(X_test)

print("MSE:", mean_squared_error(y_test, y_pred))
```

---

3. Code Example: Classification with Gradient Boosting

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load dataset
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train GB Classifier
gbc = GradientBoostingClassifier(
    n_estimators=150,
    learning_rate=0.1,
    max_depth=3,
    random_state=42
)
gbc.fit(X_train, y_train)

# Predictions
y_pred = gbc.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

---

4. Parameters of Gradient Boosting (sklearn)

Parameter	Type	Default	Description
loss	str	"squared_error"	Loss function: "squared_error", "absolute_error", "huber", "log_loss".
learning_rate	float	0.1	Shrinks contribution of each tree.
n_estimators	int	100	Number of boosting stages (trees).
subsample	float	1.0	Fraction of samples per tree (like bagging).
criterion	str	"friedman_mse"	Function for split quality.
max_depth	int	3	Max depth of individual trees.
min_samples_split	int	2	Minimum samples to split a node.
min_samples_leaf	int	1	Minimum samples in a leaf.
max_features	int/float/str	None	Features considered at each split.
random_state	int	None	Seed for reproducibility.

👉 In short:

- **Random Forest** = averaging many independent trees (bagging).
- **Gradient Boosting** = sequential correction of errors via gradients.

AdaBoost (Adaptive Boosting)

AdaBoost (Adaptive Boosting) is an **ensemble method** that combines multiple **weak classifiers** (typically *Decision Stumps*, i.e., one-level decision trees) to create a **strong classifier**.

- "Adaptive" means the algorithm adjusts the weights of misclassified samples — focusing more on hard examples in subsequent rounds.

So, instead of training one big complex model, AdaBoost trains a sequence of simple models, each one *correcting* the mistakes of the previous.

---

1. How AdaBoost Works (Step-by-Step)

Weights are updated using **Odd ratio (scaling factor), which is the number of correctly classified points divided by the number of incorrectly classified points**

Steps:

- Train the first learner
- **Increase the weight** of the data samples that learner **predicts incorrectly**
- **Decrease the weight** of the data samples that learner **predicts correctly**
- Train another learner using **the modified dataset**
- Repeat steps 2,3,4 for all other learners

Make the prediction using the weighted **voting** based on the performance of each learner

2. Code Example

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# 1. Generate a sample dataset
X, y = make_classification(n_samples=1000, n_features=10,
                           n_informative=5, n_redundant=0,
                           random_state=42)

# 2. Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# 3. Initialize base (weak) learner
base_clf = DecisionTreeClassifier(max_depth=1)

# 4. Create AdaBoost model
ada_clf = AdaBoostClassifier(
    estimator=base_clf,
    n_estimators=100,
    learning_rate=0.5,
    algorithm='SAMME.R',  # 'SAMME' for discrete; 'SAMME.R' uses probabilities
    random_state=42
)

# 5. Train model
ada_clf.fit(X_train, y_train)

# 6. Evaluate
y_pred = ada_clf.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

3. Parameters of Adaboost

Parameter	Type	Default	Description
estimator	estimator object	DecisionTreeClassifier(max_depth=1)	The base (weak) learner to boost — typically a shallow decision tree (decision stump). You can use any classifier supporting sample_weight.
n_estimators	int	50	Number of boosting rounds (number of weak learners). Higher → more complex model but slower and possibly overfitting.
learning_rate	float	1.0	Weight applied to each classifier's contribution. Lower values make learning slower but can improve generalization.
algorithm	str	'SAMME.R'	Boosting algorithm: 'SAMME.R' uses probabilities (real boosting, faster, better) while 'SAMME' uses discrete class outputs.
random_state	int or None	None	Controls randomization for reproducibility when using stochastic base learners.
base_estimator *(deprecated)*	estimator object	None	Old name for estimator. Will be removed in future versions; use estimator instead.
weight_trim_quantile	float	0.0	Used to trim extreme sample weights. Rarely needed, helps with noisy data.
n_iter_no_change	int or None	None	Stops early if no improvement after given number of iterations (only used with validation sets).
validation_fraction	float	0.1	Fraction of training data reserved for early stopping validation (used with n_iter_no_change).

Parameter	Type	Default	Description
tol	float	1e-4	Tolerance for early stopping. Training stops if improvement is below this threshold.
estimator_params	dict	{}	Parameters passed to the base estimator at each boosting iteration.

XGBoost

- **XGBoost** = **Extreme Gradient Boosting**.
- It's an **ensemble learning algorithm** based on **gradient boosting**, but optimized for **speed, performance, and scalability**.
- It builds trees sequentially (like Gradient Boosting) but with many **engineering improvements**:
    - Regularization (L1 & L2)
    - Efficient handling of missing values
    - Tree pruning with **max depth**
    - Parallelization
    - Handling of sparse data
    - Early stopping

💡 Think of it as **Gradient Boosting on steroids**.

---

1. Key Concepts Behind XGBoost

- Starts with a weak learner (like a shallow tree).
- Builds models **sequentially**, each one learning from the **residual errors** of the previous model.
- Uses **gradient descent** on a differentiable loss function.
- Uses **similarity** instead of Gini and entropy (square of sum of labels of set / (number of points in the set + lambda (regularization)

Steps:

- Start with a naive learner that predicts 0.5 in all cases
- Calculate the **residual (difference** between the predicted and the label)
- Build a **new learner** to predict the **residual**
- **Multiply** the predictions of the learner by learning rate (η)
- **Combine** all predictions to get the prediction
- Calculate the **new residuals**

2. Example: Classification with XGBoost

```
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load data
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train model
model = xgb.XGBClassifier(
    n_estimators=200,
    max_depth=4,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42
)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))

# Feature importance
import matplotlib.pyplot as plt
xgb.plot_importance(model)
plt.show()
```

3. Example: Regression with XGBoost

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error

# Load data
X, y = fetch_california_housing(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train model
reg = xgb.XGBRegressor(
    n_estimators=300,
    max_depth=5,
    learning_rate=0.05,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42
)
reg.fit(X_train, y_train)

# Evaluate
y_pred = reg.predict(X_test)
print("RMSE:", mean_squared_error(y_test, y_pred, squared=False))
```

4. Parameters of XGBoost

Here's a **parameter table** for both **Classifier** and **Regressor**:

Parameter	Type	Default	Description
booster	str	"gbtree"	Booster type: "gbtree", "gblinear", "dart".
n_estimators	int	100	Number of boosting rounds (trees).
learning_rate (eta)	float	0.3	Step size shrinkage to prevent overfitting.
max_depth	int	6	Max depth of trees. Higher → more complex model.
min_child_weight	int	1	Minimum sum of instance weights in a child.
gamma	float	0	Minimum loss reduction to make a split.
subsample	float	1.0	Fraction of training instances used for each tree.
colsample_bytree	float	1.0	Fraction of features per tree.
colsample_bylevel	float	1.0	Fraction of features per split.
lambda (reg_lambda)	float	1	L2 regularization term.
alpha (reg_alpha)	float	0	L1 regularization term.
scale_pos_weight	float	1	Balances positive/negative classes (for imbalance).
random_state	int	None	Seed for reproducibility.
tree_method	str	"auto"	GPU/CPU optimizations: "hist", "gpu_hist".
eval_metric	str	depends	Metric: "rmse", "mae", "logloss", "error".
objective	str	depends	Loss: "reg:squarederror", "binary:logistic", "multi:softmax", etc.

KNN

- **KNN = K-Nearest Neighbors** is a **lazy learner** (no explicit training, it just stores the data).
- For classification: it assigns the class based on the **majority vote** of the nearest k neighbors.
- For regression: it takes the **average (or weighted average)** of the nearest k neighbors.

---

1. How it Works

1. Choose k (number of neighbors).
2. For a new data point:
    - Compute the **distance** (commonly Euclidean) to all training points.
    - Find the **k closest points**.
    - For classification → predict the **most common class**.
      For regression → predict the **average/weighted value**.
3. Done!

---

2. Distance Metrics

Euclidean Distance

- It's the most common distance used in KNN.
- The shortest straight line between the two points.

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

- **Example**
    - x=(3,4), y=(6,8)

$$d(x,y) = \sqrt{(6-3)^2 + (8-4)^2} = \sqrt{9+16} = \sqrt{25} = 5$$

Manhattan Distance

- Also called **L1 distance** or "city-block distance."
- Instead of a straight line, you move along the **grid lines**
- Imagine you must walk blocks, not diagonally.

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

- **Example**
  - x=(3,4), y=(6,8)

$$d(x,\ y)\ = |6-3| + |8-4| = 3 + 4 = 7$$

Minkowski Distance (general form)

- A **generalization** of both Euclidean and Manhattan.
- Controlled by the parameter p.
- If p=1 → Manhattan distance
- If p=2 → Euclidean distance
- If p→∞ → Chebyshev distance (max absolute difference)

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

- **Example**
  - x=(3,4), y=(6,8), p=3

$$d(x,y) = (|6-3|^3 + |8-4|^3)^{1/3} = (27+64)^{1/3} = (91)^{1/3} \approx 4.48$$

Chebyshev Distance

- Instead of summing distances (like Manhattan) or squaring (like Euclidean), Chebyshev looks at the **largest coordinate difference**.
- Points: p=(1,2,3), q=(4,6,5)

|p1−q1|=|1−4|=3 |p2−q2|=|2−6|=4 |p3−q3|=|3−5|=2

DChebyshev(p,q)=max(3,4,2)= 4

Hamming Distance (for categorical)

- Measures the **number of positions where the values differ**.
- Used when features are **binary (0/1)** or **categorical**.
- Used in **error detection/correction** (like QR codes, coding theory).

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

- **Example 1 (binary vectors):**
  - x=(1,0,1,1), y=(1,1,0,1)
    Differences at positions 2 and 3 → Hamming distance = 2 (or 2/4=0.5 if normalized)

- **Example 2 (strings):**
  - "karolin" vs "kathrin" → differ in 3 positions → Hamming distance = 3.

---

3. Cost Function (for Classification)

KNN doesn't optimize weights like Logistic or SVM. Instead, the **error** is based on misclassification:

$$J(\hat{y},y) = \frac{1}{N}\sum_{i=1}^{N} 1(\hat{y}_i \neq y_i)$$

where $\mathbf{1}$ is the indicator function.

---

4. Cost Function (for Regression)

Prediction = **average (or weighted average)** of the target values of the nearest neighbors.

Unlike linear/logistic regression, KNN doesn't **learn parameters with gradient descent**. It's a **lazy learner** → it just stores the data, and "cost" is only computed when evaluating predictions.

So:

- No cost function during training (because there's no training).
- But when evaluating performance → **MSE, RMSE, MAE, or R^2** are used.

$$J(\hat{y},y) = \frac{1}{N}\sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

5. Code Example (Classification)

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load dataset
X, y = load_iris(return_X_y=True)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize KNN
knn = KNeighborsClassifier(n_neighbors=5, metric="minkowski", p=2)

# Train (fit)
knn.fit(X_train, y_train)

# Predict
y_pred = knn.predict(X_test)

# Evaluate
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

6. Parameters of KNeighborsClassifier

Parameter	Type	Default	Description
n_neighbors	int	5	Number of neighbors.
weights	{'uniform', 'distance'} or callable	'uniform'	Weighting function for neighbors.
algorithm	{'auto', 'ball_tree', 'kd_tree', 'brute'}	'auto'	Algorithm to compute nearest neighbors.
leaf_size	int	30	Leaf size for BallTree/KDTree.
p	int	2	Power parameter for Minkowski distance (p=1 → Manhattan, p=2 → Euclidean).
metric	str/callable	'minkowski'	Distance metric.
metric_params	dict	None	Extra parameters for the metric.
n_jobs	int	None	Number of parallel jobs.

Ensemble Learning

Ensemble learning is a technique where **multiple models** (often called *weak learners*) are combined to produce a **stronger model**.
The main idea: *a group of weak models can outperform a single strong model*.

Why Ensembles Work?

1. **Reduce Variance** → Bagging (e.g., Random Forest).
2. **Reduce Bias** → Boosting (e.g., AdaBoost, Gradient Boosting, XGBoost).
3. **Improve Predictions** → By combining models, we cancel out individual weaknesses.

Types of Ensemble Methods

1. Bagging (Bootstrap Aggregating)

- Trains many models on **random subsets** of the training data.
- Predictions are averaged (regression) or voted (classification).
- Helps reduce **variance** (overfitting).

Key steps:

- **Sample data with replacement** → bootstrap samples.
- Train a weak learner (like a decision tree) on each.
- Aggregate results.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.metrics import accuracy_score

# Load dataset
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Bagging with Decision Trees
bag_clf = BaggingClassifier(
   estimator=DecisionTreeClassifier(),
   n_estimators=50,      # number of trees
   max_samples=0.8,      # % of data sampled per tree
   bootstrap=True,       # sampling with replacement
   random_state=42
)

bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)

print("Bagging Accuracy:", accuracy_score(y_test, y_pred))
```

Parameter	Type	Default	Description
estimator	object	None (DecisionTreeClassifier used by default)	The base learner.
n_estimators	int	10	Number of base learners to fit.
max_samples	int/float	1.0	Number (or fraction) of samples to draw for each learner.
max_features	int/float	1.0	Number (or fraction) of features per learner.
bootstrap	bool	True	Sample with replacement (bootstrap).
bootstrap_features	bool	False	Bootstrap features.
oob_score	bool	False	Whether to compute out-of-bag score.
warm_start	bool	False	If True, reuse previous learners when adding more.
n_jobs	int	None	CPUs to use in parallel.
random_state	int	None	Reproducibility.
verbose	int	0	Controls logging.

2. Stacking (Stacked Generalization)

- Trains **different types of models**
- A meta-model learns how to best combine the predictions from several different base models.
- Unlike bagging/boosting, it blends diverse learners.

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import StackingClassifier

# Base models
estimators = [
   ('dt', DecisionTreeClassifier(max_depth=4, random_state=42)),
   ('knn', KNeighborsClassifier(n_neighbors=5)),
   ('svm', SVC(probability=True, kernel="linear", random_state=42))
]

# Stacking (meta-model = Logistic Regression)
stack_clf = StackingClassifier(
   estimators=estimators,
   final_estimator=LogisticRegression(),
   passthrough=False   # if True → also pass original features to meta-learner
)

stack_clf.fit(X_train, y_train)
y_pred = stack_clf.predict(X_test)

print("Stacking Accuracy:", accuracy_score(y_test, y_pred))
```

Parameter	**Type**	**Default**	**Description**
estimators	list of (str, estimator)	**Required**	Base models to stack.
final_estimator	estimator	LogisticRegression() (Classifier) / RidgeCV() (Regressor)	Meta-learner that combines base models.
cv	int, cross-validation	5	CV splitting strategy for training meta-model.
stack_method	str/list	"auto"	How to pass base learners' outputs to final estimator.
n_jobs	int	None	CPUs for parallel training.
passthrough	bool	False	If True, pass original features + base learner outputs.

3. Boosting

- Models are trained **sequentially**.
- Each model tries to **fix the mistakes** of the previous one.
- Helps reduce **bias**.

Key steps:

- Train a weak model.
- Increase weights of misclassified/misfit points.
- Train next model with focus on hard points.
- Combine all models (weighted sum).

```
from sklearn.ensemble import AdaBoostClassifier

# AdaBoost with Decision Trees
ada_clf = AdaBoostClassifier(
   estimator=DecisionTreeClassifier(max_depth=1),  # weak learners
   n_estimators=100,
   learning_rate=0.5,
   random_state=42
)

ada_clf.fit(X_train, y_train)
y_pred = ada_clf.predict(X_test)

print("AdaBoost Accuracy:", accuracy_score(y_test, y_pred))
```

Parameter	Type	Default	Description
estimator	object	None (DecisionTreeClassifier(max_depth=1))	Base learner.
n_estimators	int	50	Number of weak learners.
learning_rate	float	1.0	Shrinks contribution of each weak learner.
random_state	int	None	Reproducibility.

4. Voting

Voting is one of the **simplest ensemble methods**.
It combines **different models** and makes a decision based on their predictions.

1. Hard Voting

- Each classifier votes for a class.
- The class with the **majority vote** wins.
- Example: If predictions are [0, 1, 1, 1], the majority is **1**.

2. Soft Voting

- Instead of voting for a class directly, each classifier predicts **probabilities**.
- The class with the **highest average probability** wins.
- Usually performs better than hard voting if classifiers can estimate probabilities (predict_proba must be available).

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score

# Load dataset
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Define base models
log_clf = LogisticRegression(max_iter=1000, random_state=42)
knn_clf = KNeighborsClassifier(n_neighbors=5)
dt_clf = DecisionTreeClassifier(max_depth=4, random_state=42)
svm_clf = SVC(probability=True, kernel="linear", random_state=42)  # probability=True for soft voting

# Voting Classifier (Hard Voting)
hard_voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('knn', knn_clf), ('dt', dt_clf), ('svm', svm_clf)],
    voting='hard'
)

# Voting Classifier (Soft Voting)
soft_voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('knn', knn_clf), ('dt', dt_clf), ('svm', svm_clf)],
    voting='soft'
)
```

```python
# Fit models
hard_voting_clf.fit(X_train, y_train)
soft_voting_clf.fit(X_train, y_train)

# Predictions
hard_pred = hard_voting_clf.predict(X_test)
soft_pred = soft_voting_clf.predict(X_test)

# Evaluate
print("Hard Voting Accuracy:", accuracy_score(y_test, hard_pred))
print("Soft Voting Accuracy:", accuracy_score(y_test, soft_pred))
```

Parameter	Type	Default	Description
estimators	list of (str, estimator)	**Required**	List of models (name, model) to combine.
voting (Classifier only)	{"hard", "soft"}	"hard"	Hard = majority vote, Soft = average probabilities.
weights	list of floats	None	Weights for each estimator when combining predictions.
n_jobs	int	None	Number of CPU cores to use in parallel.
flatten_transform	bool	True	If True, concatenate results from transformers.
verbose	bool/int	False	Controls verbosity.

When to use Each Model

Regression Models (predicting continuous values)

1. Linear Regression

- **Use when:**
    - Relationship between features & target is **approximately linear**.
    - Few features, low complexity, interpretability matters.
- **Don't use when:** data has complex nonlinear patterns.

---

2. Polynomial Regression

- **Use when:**
    - Relationship is nonlinear but can be approximated with polynomials.
    - Data has curves or quadratic-like behavior.
- **Beware:** Higher degree → overfitting.

Classification Models (predicting categories)

3. Logistic Regression

- **Use when:**
    - Binary or multiclass classification with **linearly separable data**.
    - You want **probabilities** and interpretability.
- **Don't use when:** decision boundary is highly nonlinear.

Both Classification and Regression

4. Support Vector Machine (SVM)

- **Use when:**
    - Data has **clear separation** between classes.
    - Works well in **high-dimensional spaces** (text, image features).
- **Use kernel trick (RBF, Poly):** when boundary is nonlinear.
- **Don't use when:** dataset is very large → training becomes slow.

---

5. Decision Trees

- **Use when:**
    - You want interpretability (easy to explain with rules), fast.
    - Nonlinear relationships, mixed data types.
- Good for tabular data, but not unstructured data.
- **Don't use when:** you need high accuracy on complex data (trees overfit).

---

6. Random Forest

- **Use when:**
    - You want better accuracy than a single tree.
    - You want robustness (averaging reduces variance).
- **Good for:** tabular datasets, feature importance analysis.
- **Don't use when:** real-time predictions with strict latency (RF is slower).

7. Gradient Boosting

- **Use when:**
    - You want strong predictive performance.
    - Data is tabular, possibly imbalanced.
- **Good for:** small to medium datasets.
- **Don't use when:** dataset is huge → can be slow.

---

8. XGBoost & Adaboost

- **Use when:**
    - You want **state-of-the-art performance** on structured/tabular data.
    - Handling missing values automatically.
- **Faster & optimized version** of gradient boosting.
- **Don't use when:** dataset is mostly unstructured (images, text → deep learning may work better).

---

9. K-Nearest Neighbors (KNN)

- **Use when:**
    - Data is small to medium.
    - Decision boundary is complex and non-linear.
    - Works for both classification & regression.
- **Don't use when:**
    - Data is high-dimensional (curse of dimensionality).
    - Large dataset → predictions become slow.

---

10. Neural Networks

- **Use when:**
    - There is a system of multiple models, as it is easier to string together
    - Good for structured and unstructured data
    - For Transfer learning
- **Disadvantages:**
    - Slow

General Rule of Thumb:

- If you **need interpretability** → Linear, Logistic, Decision Trees.
- If you **need accuracy on tabular data** → Random Forest, Gradient Boosting, XGBoost.
- If you **have complex boundaries** → SVM, KNN.

Classification Evaluation Metrics

Confusion Matrix

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

---

1. Accuracy

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

Definition

The ratio of correct predictions to total predictions.

Good when

Classes are balanced.

Problem:

Misleading with imbalanced datasets (e.g., 95% negatives → always predicting "negative" gives 95% accuracy but useless).

2. Precision (a.k.a. Positive Predictive Value)

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

Definition

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

Good when

Cost of false positives is high (e.g., spam detection, cancer diagnosis alerts).

---

3. Recall (a.k.a. Sensitivity / True Positive Rate)

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

Definition

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

Good when

Cost of false negatives is high (e.g., disease detection, fraud detection).

4. F1-Score

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

Definition

Harmonic mean of Precision and Recall.

Why it exists

Precision and Recall often conflict. Increasing Precision usually decreases Recall, and vice versa.

Solution

F1-score combines them using the harmonic mean (not simple average, because harmonic mean punishes extreme imbalance).

Good when

Need a balance between precision and recall.

Problem

Does not consider true negatives.

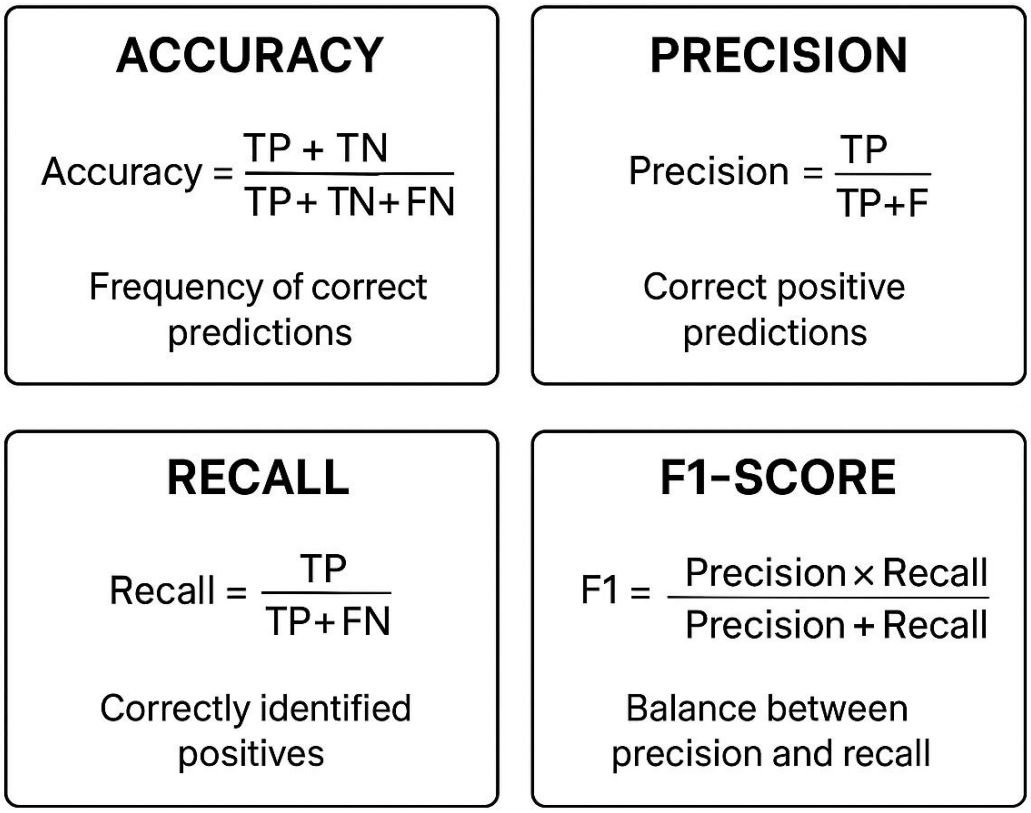
When is F1-Score "Good"?

There's no universal number, it depends on your problem and data balance, but here are the rules of thumb:

- **F1 ≈ 1.0 (close to 1):** Excellent — model balances Precision & Recall very well.
- **F1 > 0.8:** Strong performance (usually good in most real-world tasks).
- **F1 0.6 – 0.8:** Decent, but might need improvement depending on stakes.
- **F1 < 0.5:** Weak — usually means either Precision or Recall is very low.

When Should You Worry?

1. **If you see a big Precision–Recall imbalance:**
    - Example: Precision = 0.9, Recall = 0.1 → F1 ≈ 0.18 (very poor).
    - Even if Precision looks good, F1 exposes the fact Recall is terrible.
2. **If the problem is high-stakes:**
    - **Medical diagnosis**: Missing positives (low Recall) can be dangerous → low F1 is a warning.
    - **Spam detection**: Blocking too many legit emails (low Precision) → low F1 means unhappy users.
3. **If dataset is imbalanced:**
    - In fraud detection, positive cases may be <1%.
    - A high Accuracy (99%) might hide that F1 is terrible because model predicts almost all negatives.
    - Here, F1 is the **real performance measure**.



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

ROC Curve

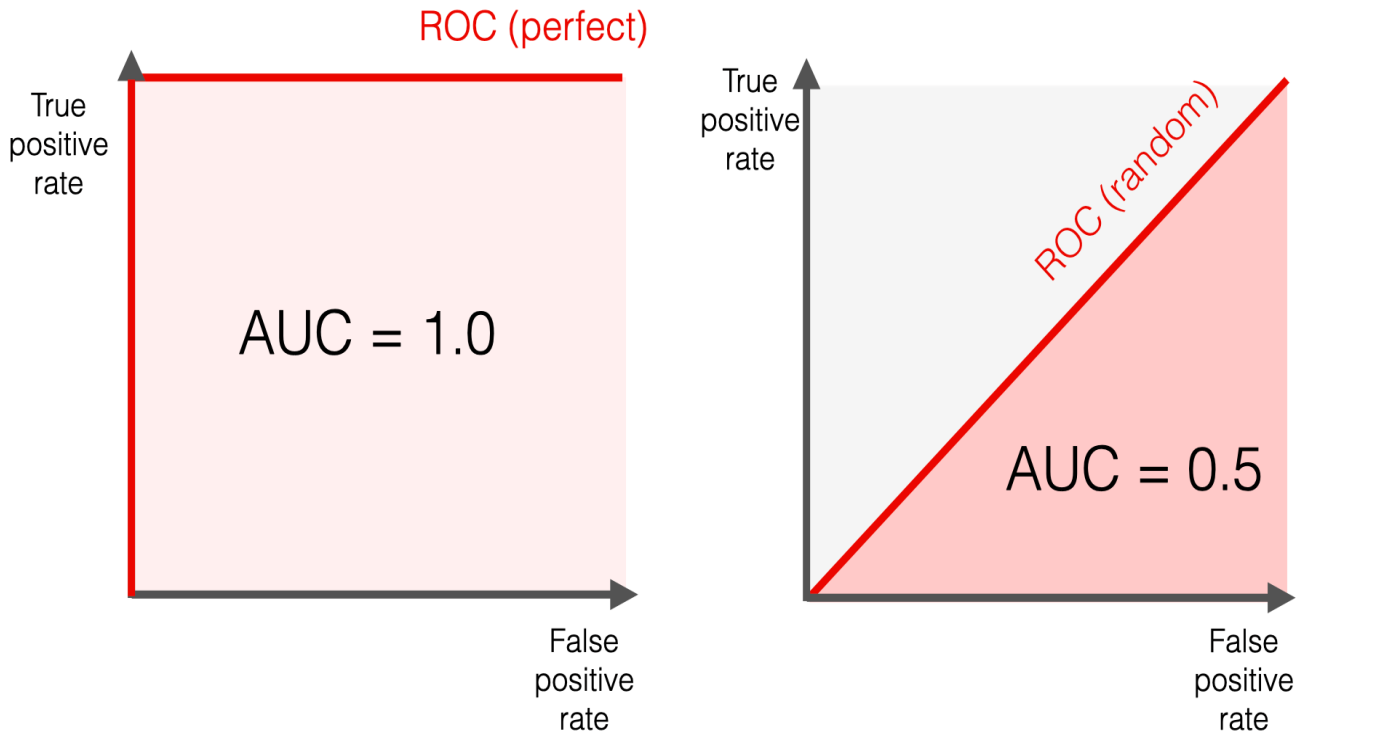
Plots True Positive Rate (Recall) vs False Positive Rate at various thresholds.

- False Positive Rate:

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

AUC (Area Under the Curve)

- measures the performance of a binary classification model by quantifying its ability to distinguish between positive and negative classes
- Ranges from 0.5 (random guessing) to 0.7 (70% chance positives rank above negatives) to 1.0 (perfect classifier).
- **Class imbalance resistant** (much better than accuracy for skewed data).



Regression Evaluation Metrics

1. Mean Absolute Error (MAE)

Formula:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - \hat{y}_i\right|$$

- Measures the **average magnitude of errors** (absolute difference).
- Easy to interpret in the same units as the target variable.
- Drawback: Treats all errors equally (outliers not penalized heavily).

---

2. Mean Squared Error (MSE)

Formula:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2$$

- Squared differences make **larger errors penalized more heavily**.
- Sensitive to outliers.
- Used a lot in optimization because it's differentiable.

---

3. Root Mean Squared Error (RMSE)

Formula:

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

- Same as MSE but **in the original unit of the target variable**.
- Interpreted as the **standard deviation of prediction errors**.

4. R^2 Score (Coefficient of Determination)

Formula:

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

- Measures **how well predictions capture variability in continuous outcomes**.
- Value ranges:
  - R^2 =1 → perfect fit
  - R^2 = 0 → no better than predicting mean
  - R^2 < 0 → worse than predicting mean
  - 0< R^2 < 1

Conclusion

In conclusion, there is no universal algorithm suitable for all tasks; the choice depends on the nature of the problem, dataset characteristics, and trade-offs between interpretability and accuracy. While foundational models like linear regression offer simplicity, ensemble methods like XGBoost provide high performance at the cost of complexity. Ultimately, the effectiveness of AI solutions relies on aligning the right algorithm with the right problem.

Feedback & Contribution

This is a living document. If you have feedback, suggestions, or additional insights that could improve it, I welcome your input. Sharing ideas helps keep this material accurate, relevant, and valuable for everyone.

Copyright & Usage

Acknowledgments

This document is authored and copyrighted by **[Youssef Amgad Elkhatib]**.
Special thanks to the contributors who provided valuable feedback and suggestions for improvements:

Name	Contribution