# LOGISTIC Regression

## Why is Linear Regression *Supervised*?

Because in supervised learning, the model learns from labeled data (data where we already know the correct answers).

- In linear regression, we have:
    - Input (X) → features (like hours studied, number of ads shown).
    - Output (Y) → actual values we want to predict (like exam score, sales).

The algorithm "looks" at both X and Y during training and learns a line (best fit) that maps inputs to outputs.

---

**Example**:

- Dataset:
    - Hours studied (X): [1, 2, 3, 4, 5]
    - Exam score (Y): [40, 50, 60, 70, 80]

Here, we already know the scores (Y).
Linear regression learns the relationship → Score = 10 × Hours + 30.

Since the model is trained on input–output pairs (labeled data), it is supervised learning.

## Why Logistic Regression is *Supervised*

Just like linear regression, logistic regression is trained on labeled data (input features + known outputs).

- Input (X): features (age, income, symptoms, etc.)
- Output (Y): actual labels we already know, but in categories (0/1, yes/no, spam/not spam).

The model uses these labels during training to learn how to map inputs → probability of belonging to a class.

---

**Example:**

Suppose we want to predict if a person will buy a product.

- Data we have:
    - Age, Salary (inputs)
    - Buy (Y=1) or Not Buy (Y=0)

Since the dataset already has the correct answers (buy/not buy), logistic regression learns from it. That's why it is supervised learning.

## What is Logistic Regression?

- Logistic regression is a **machine learning algorithm** used when the output (dependent variable) has only **two possible outcomes** (yes/no, true/false, 0/1).
- Instead of predicting a number (like linear regression does), it predicts the **probability** of something happening.

---

**Examples:**

1. **Email Spam Detection**
    - Input: Features like words in the email, sender address, etc.
    - Output: Spam (1) or Not Spam (0).
2. **Loan Approval**
    - Input: Salary, credit score, number of dependents, etc.
    - Output: Approve loan (1) or Reject loan (0).
3. **Disease Prediction**
    - Input: Age, symptoms, test results.
    - Output: Has disease (1) or Does not have disease (0).

---

The "regression" part comes from using math equations, but the output is converted into a probability (between 0 and 1) using a **sigmoid function**.
Then, based on a threshold (usually 0.5), it decides **class 0 or class 1**.

## Why Logistic Regression was introduced

Linear regression was already there, but it has **limitations** when the target variable is **categorical (like yes/no, 0/1)**.

If we try to use linear regression for classification:

- It can give predictions < 0 or > 1 (not valid probabilities).
- It assumes a straight-line relationship, but probabilities actually follow an **S-shaped curve**.

---

**So Logistic Regression was introduced to:**

1. **Handle binary outcomes** (yes/no, 0/1).

2. **Model probabilities** correctly (always between 0 and 1).
3. Provide a **classification method** while still being mathematically simple and interpretable.

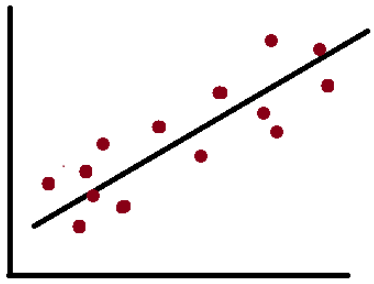
---

**Example**

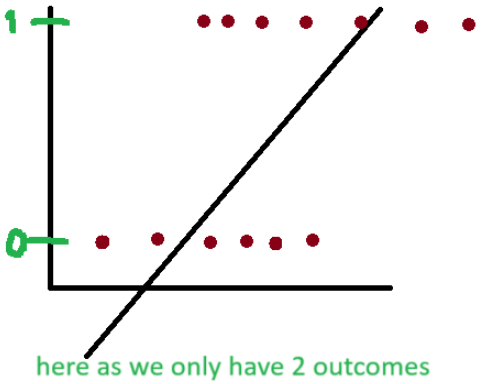
- **Linear regression problem:** Predict salary based on years of experience.
- **Logistic regression problem:** Predict whether a person will get a job (yes/no) based on years of experience.

Logistic regression solves classification tasks that linear regression **cannot handle properly**.

---

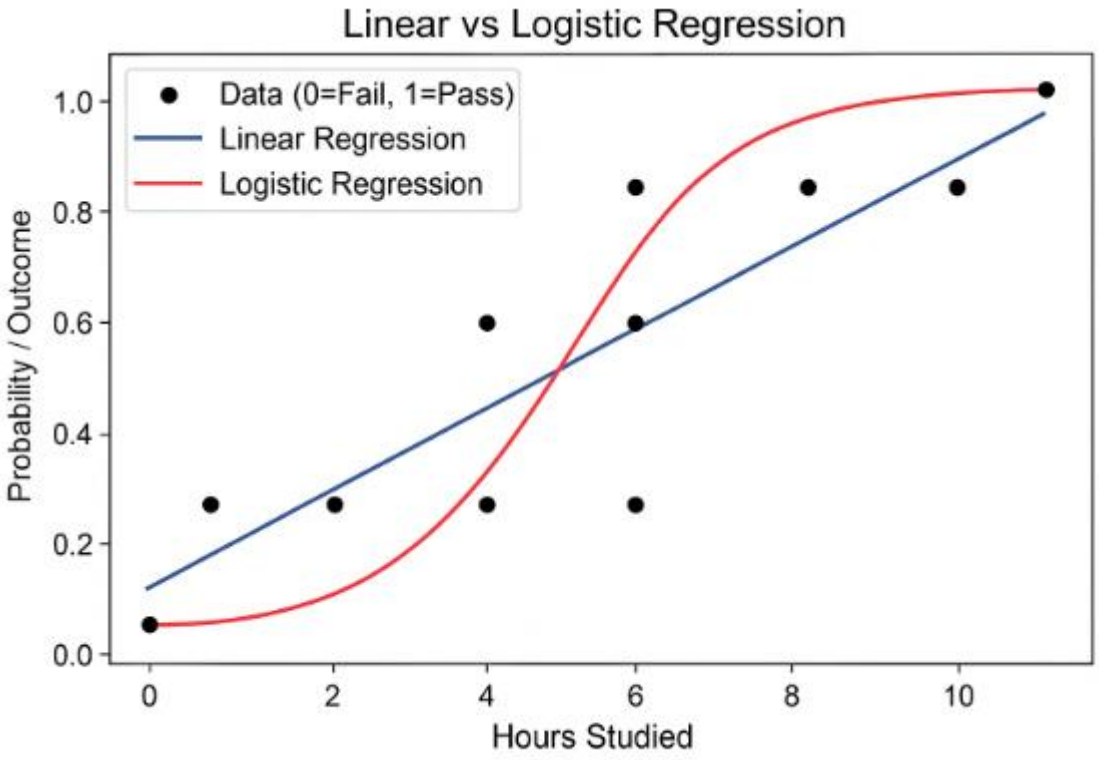


number are plotted with reg line




here as we only have 2 outcomes
either pas or fail
so tryng to plot the lines doest make sense

Logistic regression creates an S-shaped curve (sigmoid curve) because it models **probabilities**, which must always lie between 0 and 1.

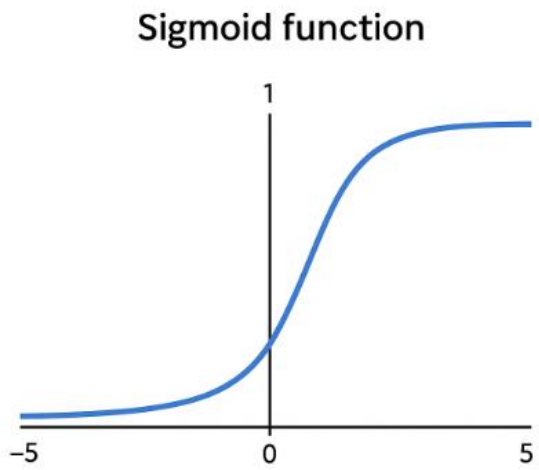



**sigmoid function**

A **sigmoid function** is a curve that **squashes any number into a range between 0 and 1**.

- Small numbers (very negative) → close to 0
- Big numbers (very positive) → close to 1
- Zero → exactly 0.5

It looks like an **S-shaped curve**, which is why we call it **"S-curve"**.




# odds ratio

**1. What is an odds ratio?**

- Logistic regression predicts the **probability** of something happening (like success/failure, yes/no).
- The **odds** of an event happening = probability it happens ÷ probability it does not happen.
- The **odds ratio (OR)** tells us **how the odds change** when a predictor changes by 1 unit.

**Easy way:**

- OR > 1 → odds **increase**
- OR < 1 → odds **decrease**
- OR = 1 → odds **stay the same**

---

**example**

Imagine you are studying whether **drinking coffee** affects the chance of **passing an exam**.

- Suppose logistic regression gives you:
- Odds ratio for coffee = 2

**Interpretation:**

- Students who drink coffee are **2 times more likely to pass** the exam than students who don't.
- If the OR was 0.5 → coffee drinkers would be **half as likely to pass**.

---

**3. Simple analogy**

Think of odds ratio like **multipliers**:

- OR = 3 → your chances triple
- OR = 0.3 → your chances drop to one-third

The **range of logistic regression** (the predicted values) is:

$0 \leq \text{predicted probability} \leq 1$

Logistic regression helps us implement a regression equation to measure the relationship between independent and dependent variables by finding the probability of an observation belonging to either of the two classes. Internally, it uses the sigmoid function, where we transform the regression equation into the logit function by taking the log of odds ratios. This algorithm allows us to predict, based on the values, whether an observation belongs to class 0 or 1.

**How do we decide which values falls under 0 and which under 1 we will use Threshold**

- Logistic regression predicts a probability that an event happens, a number between 0 and 1.
- The threshold is the cutoff point where we decide whether the prediction counts as "yes" (1) or "no" (0).
- Default threshold = 0.5.
    - If probability ≥ 0.5 → predict 1 (event happens)
    - If probability < 0.5 → predict 0 (event doesn't happen)

---

examples

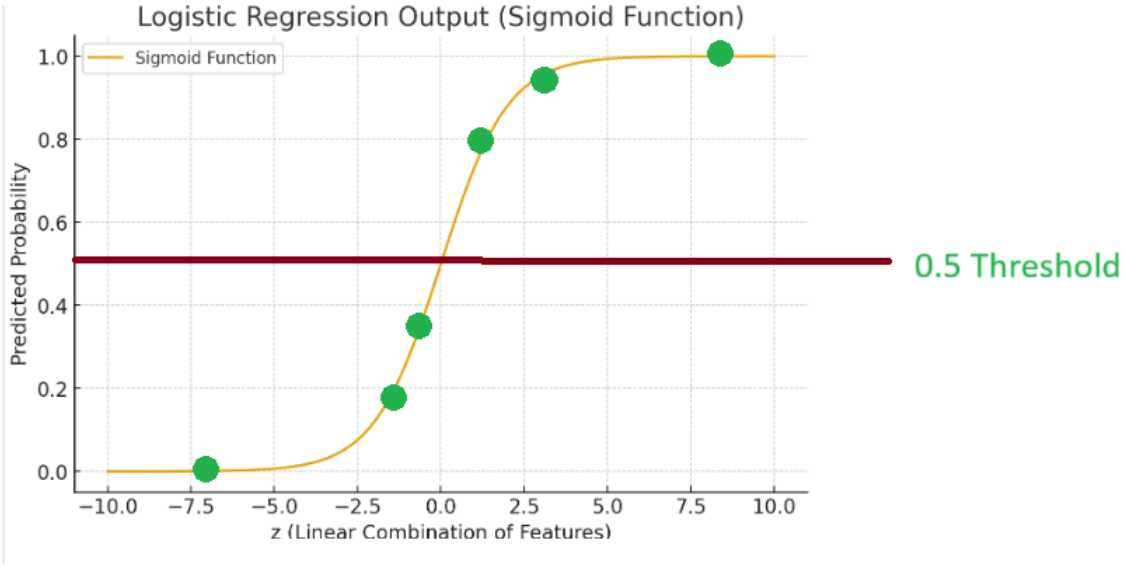
1. Email spam detection
    - Logistic regression predicts the probability that an email is spam.
    - Threshold = 0.5:
        - Probability ≥ 0.5 → classify as spam
        - Probability < 0.5 → classify as not spam
    - But if you want fewer spam emails to sneak in, you can lower the threshold to 0.3 → more emails marked as spam.
2. Medical test (disease prediction)
    - Probability that a patient has a disease.

- Threshold = 0.5 → patient predicted sick if probability ≥ 0.5.
- If it's a serious disease, doctors might lower the threshold to 0.2 → catch more possible cases (sacrifice some false positives to avoid missing real cases).

3. Loan approval
    - Predict probability someone will default on a loan.
    - Threshold = 0.5 → reject if probability ≥ 0.5.
    - Bank might set threshold higher, say 0.7 → only reject if very likely to default.

---

In short: Threshold = the line that says yes or no based on predicted probability. You can adjust it depending on what's more important: catching positives or avoiding false alarms.




**logistic regression model building into two phases**

Training Phase

- Input: Data with X (features) and Y (target / label)
- Process:
    - The model learns patterns in the data.
    - It figures out how each feature in X affects the probability of Y being 1 (yes) or 0 (no).
- Output: A trained logistic regression model, which has a "rule" or equation to convert X into a probability.

Example:

- Dataset: Emails with features like “number of links”, “contains word free”, etc. (X) and label “spam or not” (Y)
- After training, the model knows which features increase the chance of an email being spam.

---

Testing / Validation Phase

- Input: Only X (features) from new/unseen data
- Process:
    - The model calculates a probability for each observation (between 0 and 1).
    - It then compares the probability with a threshold (default = 0.5) to decide predicted Y (0 or 1).
- Output: Predicted labels (Y) for the new data

Example:

- New email comes in with “2 links” and “contains free”
- Model predicts probability = 0.7 → threshold 0.5 → classify as spam (Y = 1)

---

In short:

- Train: Learn the rule using X & Y
- Test: Apply the rule on new X to predict Y using the threshold

**Data encoding**

Data encoding is simply the process of converting data from one format into another so it can be used by computers or machine learning models.

In machine learning, encoding usually refers to converting categorical/text data into numbers, because models can only work with numbers, not words.

---

Example: Colors

You have a column Color:

Color

Red

Blue

| Color |
|---|
| Green |

Without encoding: ML models can't understand "Red," "Blue," or "Green."

With encoding:

1. Label Encoding: Assign numbers to each category

| Color | Encoded |
|---|---|
| Red | 0 |
| Blue | 1 |
| Green | 2 |

2. One-Hot Encoding: Make a column for each category

| Red | Blue | Green |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

In short:

- Data encoding = converting data to a format a computer can use.
- For ML, it usually means turning categorical/text data into numbers.

## 1. get_dummies

Idea: Turn categories into columns of 0s and 1s, showing which category each row belongs to.

example:

Imagine a small fruit shop selling Apple, Banana, Orange.

| Fruit |
|---|
| Apple |
| Banana |

Fruit

Orange

Apple

pd.get_dummies() turns this into:

| Apple | Banana | Orange |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 1 | 0 | 0 |

Each column says “Is it this fruit?” → 1 = yes, 0 = no.

---

Why use it?

- ML models can only work with numbers.
- One-hot avoids confusion: Apple ≠ Banana ≠ Orange (no “order” assumed).

---

Tip:

- pd.get_dummies(df['Fruit']) in Python does this automatically.

---

## Label Encoder

Label Encoder is a tool in Machine Learning that converts words (categories) into numbers so that the computer can understand them.
Computers cannot directly work with text, so we give each category a number.

---

\ Example:

Imagine you are making a model for a clothing shop.
You have a column called “Size” with values:

- Small
- Medium
- Large

The computer does not understand “Small”, “Medium”, “Large”.
So Label Encoder converts them into numbers like:

- Small → 0
- Medium → 1
- Large → 2

Now the data looks like this:

| Size (Text) | Size (Encoded) |
|---|---|
| Small | 0 |
| Medium | 1 |
| Large | 2 |

---

Label Encoder with a Example

Suppose you are building a model for a Car Selling Company.
You have a column called “Fuel Type” with values:

- Petrol
- Diesel
- CNG
- Electric

The computer does not understand these text values, so Label Encoder assigns a number to each:

- Petrol → 0
- Diesel → 1
- CNG → 2
- Electric → 3

Now the dataset changes like this:

| Car Model | Fuel Type (Text) | Fuel Type (Encoded) |
|---|---|---|
| Honda City | Petrol | 0 |
| Hyundai Creta | Diesel | 1 |
| Maruti Alto | CNG | 2 |
| Tesla Model 3 | Electric | 3 |
| Ford EcoSport | Petrol | 0 |
| Tata Nexon | Diesel | 1 |
| WagonR | CNG | 2 |
| MG ZS EV | Electric | 3 |

---

Now the computer can work with numbers 0,1,2,3 instead of words. This makes it easy for the machine learning model to process the data.

In short: Label Encoder = translator that changes text into numbers so the machine can use it for learning.

---

## 1. One-Hot Encoding (OHE)

One-Hot Encoding converts categorical values (like names, colors, or types) into numbers so that a machine learning model can understand them—but in a way that doesn't give any order or priority to the categories.

How it works:

- For each unique category, create a new column.
- Mark 1 if that row has that category, 0 otherwise.

example:

Suppose you have a column Fruit:

Fruit

Apple

Orange

| Fruit |
|---|
| Apple |
| Banana |

One-Hot Encoding result:

| Apple | Orange | Banana |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 0 | 1 |

Each fruit gets its own column. No numbers assigned to categories directly—so the model won't think "Apple > Orange > Banana."

---

2. get_dummies in Pandas

get_dummies is a function in Pandas that performs one-hot encoding automatically.

Example using the same data:

```
import pandas as pd


df = pd.DataFrame({'Fruit': ['Apple', 'Orange', 'Apple', 'Banana']})

pd.get_dummies(df['Fruit'])
```

Output:

| Apple | Banana | Orange |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |

Difference:

- One-hot encoding is the concept.
- get_dummies is a tool in Python/Pandas that implements that concept automatically.

If you have many categories, one-hot encoding creates many new columns. Sometimes people use Label Encoding instead—but that can wrongly suggest order if categories are non-numeric.

## manual approach

In **encoding**, a **manual approach** means you **convert categorical data to numbers by hand**, instead of using automatic tools like LabelEncoder or OneHotEncoder. You explicitly define the mapping yourself.

**Example (real-life):**

Suppose you have a column Color with values: Red, Green, Blue.
A manual encoding would be:

- Red → 0
- Green → 1
- Blue → 2

You just replace the text with these numbers in your dataset manually (or via a dictionary in code).

**Why use it:**

- Gives full control over the mapping.
- Useful when you want **specific order or custom values**.

## Feature Scaling

- In machine learning, feature scaling means adjusting the values of features (columns) so they are on a similar scale.
- Models that use distance or gradient calculations (like Logistic Regression, SVM, KNN, Neural Networks) perform better when features are scaled.
- Without scaling, a feature with large values (like salary in lakhs) can dominate over another with small values (like age in years).

---

Standard Scaling (Z-score Normalization)

**Standard Scaling** (also called Z-score Normalization) is a feature scaling technique that transforms the data so that each feature has:

- **Mean = 0**
- **Standard Deviation = 1**

n simple words: Standard Scaling makes different features comparable by bringing them to the same scale, without changing the shape of the distribution.

---

Example 1: Students' Data

You want to predict student performance based on Age and Exam Score.

| Student | Age (Years) | Exam Score (100) |
|---|---|---|
| A | 18 | 90 |
| B | 20 | 70 |
| C | 25 | 50 |
| D | 30 | 30 |

Problem:

- Age ranges from 18–30
- Exam Score ranges from 30–90
- Different scales → model may give more weight to Exam Score (larger range).

After Standard Scaling:

| Student | Age (Scaled) | Exam Score (Scaled) |
|---|---|---|
| A | -1.14 | 1.34 |
| B | -0.57 | 0.45 |
| C | 0.57 | -0.45 |
| D | 1.14 | -1.34 |

Now both features are on the same scale (around -2 to +2).

---

Example 2: Daily Life Analogy

Imagine you're comparing heights in centimeters and weights in kilograms of people.

- Heights ≈ 150–200 cm
- Weights ≈ 50–100 kg

If you don’t scale, the model might think “height” is more important (because numbers are bigger).
After scaling, both height and weight are adjusted to the same scale, so the model treats them fairly.

---

Key takeaway:

- Feature Scaling makes sure no feature dominates just because of its unit or range.
- Standard Scaling is one of the most commonly used methods.

## Evaluation Matrix for classification problems

### 1. Confusion Matrix

A **confusion matrix** is like a table that shows how many predictions were correct or wrong for each class.

| | **Predicted Yes** | **Predicted No** |
|---|---|---|
| **Actual Yes** | True Positive (TP) | False Negative (FN) |
| **Actual No** | False Positive (FP) | True Negative (TN) |

**example:**
Suppose you have a test to detect COVID:

- TP: Sick people correctly predicted as sick.
- TN: Healthy people correctly predicted as healthy.
- FP: Healthy people wrongly predicted as sick.
- FN: Sick people wrongly predicted as healthy.

---

### 2. Accuracy

**Accuracy** = How many predictions are correct overall.

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

**Example:**
If 100 people are tested and 90 predictions are correct, Accuracy = 90%.

**Limitation:** Not always good if classes are imbalanced (like 95 healthy vs 5 sick).

---

**3. Precision**

**Precision** = Of all predicted **Yes**, how many were actually Yes?

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

**Example:**
If the test predicts 10 people as sick, but only 8 really are, Precision = 8/10 = 0.8 (80%).
Think: “Of all flagged as sick, how many truly are?”

---

**4. Recall (Sensitivity)**

**Recall** = Of all actual **Yes**, how many did we correctly predict?

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

**Example:**
If there are 10 sick people and the test catches 8, Recall = 8/10 = 0.8 (80%).
Think: “Of all sick people, how many did we find?”

---

**5. F1 Score**

**F1 Score** = Harmonic mean(average of ratios) of Precision and Recall. Balances both.

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

**Example:**
If Precision = 0.8 and Recall = 0.8, F1 = 0.8.
Good when you want a balance between missing sick people (Recall) and wrongly flagging healthy people (Precision).

---

**Summary in simple words:**

- **Accuracy**: Overall correctness.

- **Precision**: How trustworthy the positive predictions are.
- **Recall**: How good we are at catching all positives.
- **F1 Score**: Balance of precision & recall.

---

# EVALUATION MATRIX FOR CLASSIFICATION

## CONFUSION MATRIX

| | Predicted Yes | Peres No |
|---|---|---|
| Actual Yes | True Positive | False Negative |
| Actual No | False Positive | True Negative |

## ACCURACY

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

90 out of 100 predictions are correct

## RECALL

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

8 out of 10 sick are correctly identified

## PRECISION

$$\frac{TP}{TP + FP}$$

8 out of 10 predicted sick are actually sick

## F1 SCORE

$$\frac{2 * Precision * Recall)}{Precision + Recall}$$

0.8

## What is Scaling?

Scaling means adjusting your data so that all features (columns) are on a similar range — usually between 0–1 or with mean 0 and standard deviation 1.

---

Why do we need scaling?

Because some models (like KNN, SVM, Logistic Regression) are distance-based —
if one feature has very large values, it will dominate the others.

---

### Example

Imagine you're ranking job candidates based on:

- Experience: 1–20 years
- Interview Score: 0–10

Without scaling, "Experience" (20) looks more important than "Interview Score" (10) —
even if both should be equally important.

After scaling, both features lie in a similar range (like 0–1),
so the model treats them fairly.

---

### Types of Scaling

There are mainly two common methods:

---

### Normalization (Min–Max Scaling)

Meaning:
Brings all values between 0 and 1.

Formula:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

| Symbol | Meaning | Example |
|---|---|---|
| **X** | Original value of a feature | Salary = 60,000 |
| **X_min** | Minimum value of that feature | 20,000 |

| Symbol | Meaning | Example |
|---|---|---|
| **X_max** | Maximum value of that feature | 100,000 |
| **X'** | Normalized value (between 0 and 1) | ? |

Example:

If a salary column has values 20,000 to 100,000:

- 20,000 → 0
- 100,000 → 1
- 60,000 → 0.5

Best for: When you know data has fixed boundaries (like percentages, ages, etc.)

---

## Standardization (Z-score Scaling)

Meaning:
Centers data around mean = 0 and standard deviation = 1.
It doesn't bound data between 0–1, but it adjusts the spread.

Formula:

$$Z = \frac{X - \mu}{\sigma}$$

| Symbol | Meaning | Example |
|---|---|---|
| **X** | Original value | Height = 170 cm |
| **μ (mu)** | Mean (average) of the feature | 160 cm |
| **σ (sigma)** | Standard deviation (spread) | 10 cm |
| **Z** | Standardized value (Z-score) | ? |

Example:

If height = 170 cm, mean = 160, std = 10 →
[
Z = (170 - 160)/10 = 1
]
That means 170 cm is 1 standard deviation above average.

Best for: Models assuming normal distribution (like Linear/Logistic Regression, PCA)

---

Difference

| Feature | Normalization | Standardization |
|---|---|---|
| Range | 0 to 1 | No fixed range |
| Center | Not necessarily 0 | Mean = 0 |
| Best for | Bounded data | Normally distributed data |
| Example use | Neural networks | Regression, PCA |

Real-world analogy

Think of scaling like converting different units:

- One person reports height in cm, another in feet.
- You can't compare them until you convert to the same unit (scale).

**Tunning**

1. Feature Selection (Picking the right variables)

Idea: Choose the features (columns) that truly impact your prediction. Using irrelevant data can confuse the model.

Real-life logistics example:

- Problem: Predict if a package will be delivered on time.
- Features you might have: Distance, Package Weight, Delivery Vehicle Type, Weather, Driver Experience, Package Color.
- Likely important features: Distance, Package Weight, Weather, Driver Experience.
- Irrelevant: Package Color (doesn't affect delivery time).

Only the selected features (Distance, Weight, Weather, Driver Experience) improve the model's prediction.

---

2. Dedicated Approach (Tailoring model to scenario)

The **dedicated approach** means you **tune your model specifically for a particular goal or metric** rather than using generic settings.

- Think of it like **customizing a phone for photography**. You don't just use default camera settings—you adjust brightness, ISO, focus mode, etc., to get the perfect shot for your purpose.
- In ML, this could mean focusing on **maximizing recall** instead of accuracy if missing a positive case is costly (like detecting fraud).

**Example:**

- You have a model to detect **fraudulent transactions**.
- You care more about catching **all frauds** (recall) than accidentally flagging a few good transactions (precision).
- So, you adjust the model specifically to **prioritize recall**—this is the dedicated approach.

---

**2. Adjusting the Threshold**

By default, many models like logistic regression classify a prediction as positive if the probability > 0.5.

- But sometimes 0.5 is **not ideal**. You can **adjust this threshold** to balance precision and recall for your goal.

**Example:**

- Suppose your spam email filter flags emails as spam if probability > 0.5.
- You notice some spam emails still reach your inbox.
- You lower the threshold to **0.3** so more emails are flagged as spam, catching more spam but also risking a few good emails getting flagged.

Mathematically:

$$\text{Predict Positive if } P(\text{positive}) > \text{threshold}$$

- **Threshold = 0.5** → default
- **Threshold = 0.3** → more sensitive (catch more positives)
- **Threshold = 0.7** → more strict (reduce false positives)

---

**3. Putting it Together**

1. **Choose your goal:** Maximize recall, precision, or balance F1-score.
2. **Use a dedicated approach:** Tune model specifically for that metric.

3. **Adjust the threshold:** Change the cutoff probability to optimize performance for your goal.

**Example with fraud detection:**

| Threshold | Precision | Recall |
|---|---|---|
| 0.5 | 90% | 70% |
| 0.3 | 80% | 90% |

- If your goal is **not missing any fraud**, pick **0.3** even if some good transactions are flagged.

## 3. Hyperparameter Tuning (Fine-tuning model settings)

Idea: Adjust the "settings" of the model to make it perform better.

Real-life logistics example:

- Problem: Predict on-time delivery using a decision tree.
- Hyperparameters:
  - Max depth → How many splits the tree can make
  - Min samples per leaf → Minimum packages needed to make a decision
- Tuning example:
  - Max depth too low → Underfitting → misses patterns → predicts all deliveries as on-time
  - Max depth too high → Overfitting → predicts perfectly on past data, fails on new Goal: Find the "sweet spot" for hyperparameters where accuracy on new data is maximized.

---

Summary Table for Logistics

| Tuning Step | Real-life Example | Key Idea |
|---|---|---|
| Feature Selection | Drop package color, keep weight, distance | Only use relevant info |
| Dedicated Approach | Separate models for cold-chain vs normal | Tailor to scenario |
| Hyperparameter Tuning | Max depth of decision tree | Fine-tune model settings |

## Key Points to Remember in Logistic Regression

1. **Works on Probability**

- Logistic regression predicts the **probability** of an event occurring (values between 0 and 1).
- Example: Probability of a package being delivered on time = 0.8 → 80% chance.

2. **Less Assumptions**
   - Unlike linear regression, it doesn't require:
     - Normal distribution of predictors
     - Homoscedasticity (constant variance)
     - Linear relationship between X and Y
   - Makes it more flexible for classification problems.
3. **Maximum Likelihood Estimation (MLE)**
   - Logistic regression uses **MLE** to find the best coefficients (weights) that make the observed outcomes most probable.
   - Think: "Choose coefficients that make our model's predictions match reality as closely as possible."
4. **Binary Outcome**
   - Usually used when the dependent variable has **2 classes** (e.g., On-time vs Delayed).
5. **Sigmoid Function**
   - Converts any input value to a probability between 0 and 1, forming the classic **S-shaped curve**.
6. **Threshold for Classification**
   - Default threshold = 0.5 (probability ≥ 0.5 → class 1, else class 0)
   - Can be adjusted depending on the business requirement.

**evaluate the performance of a classification model**, especially when dealing with **imbalanced classes**

**1. ROC Curve (Receiver Operating Characteristic Curve)**

**ROC** stands for **Receiver Operating Characteristic**. It's a curve that shows how well a classification model distinguishes between two classes (like "yes" vs "no" or "disease" vs "no disease").

- **X-axis:** False Positive Rate (FPR) → How often the model incorrectly predicts positive.
- **Y-axis:** True Positive Rate (TPR) → How often the model correctly predicts positive.

**Purpose:** It helps you see the trade-off between catching positives and avoiding false alarms at different thresholds.

**example:**

- Think of a **medical test** for a disease. If you set the test to be very sensitive, it will catch almost all sick people (high TPR) but might also flag some healthy people as sick (high FPR). The ROC curve shows this balance.
- **Axes:**
    - **X-axis:** False Positive Rate (FPR) → % of "no" that were wrongly predicted as "yes"
    - **Y-axis:** True Positive Rate (TPR) → % of "yes" that were correctly predicted
- **Interpretation:**
    - Curve **closer to top-left corner** → better model
    - Curve near diagonal (45° line) → model is guessing randomly

### 1. True Positive Rate (TPR / Sensitivity / Recall):

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

### 2. False Positive Rate (FPR):

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

**2. AUC (Area Under the Curve)**

- It measures the overall ability of a classification model to distinguish between positive and negative classes.
- The "curve" here is the ROC curve, which plots:
    - True Positive Rate (Recall / Sensitivity) on the Y-axis
    - False Positive Rate (1 – Specificity) on the X-axis
- AUC tells you how likely your model is to rank a random positive instance higher than a random negative one.
- Higher AUC → better model at separating the two classes.

The AUC (Area Under the Curve) value always ranges between 0 and 1:

- AUC = 1 → Perfect model (always distinguishes positive & negative correctly)

- AUC = 0.5 → No discrimination (model is random)
- AUC < 0.5 → Worse than random (model is predicting opposite of reality)

So the valid range is: $0 \leq AUC \leq 1$.

For practical purposes:

- 0.7–0.8 → Acceptable
- 0.8–0.9 → Good
- 0.9+ → Excellent

---

**3 Example**

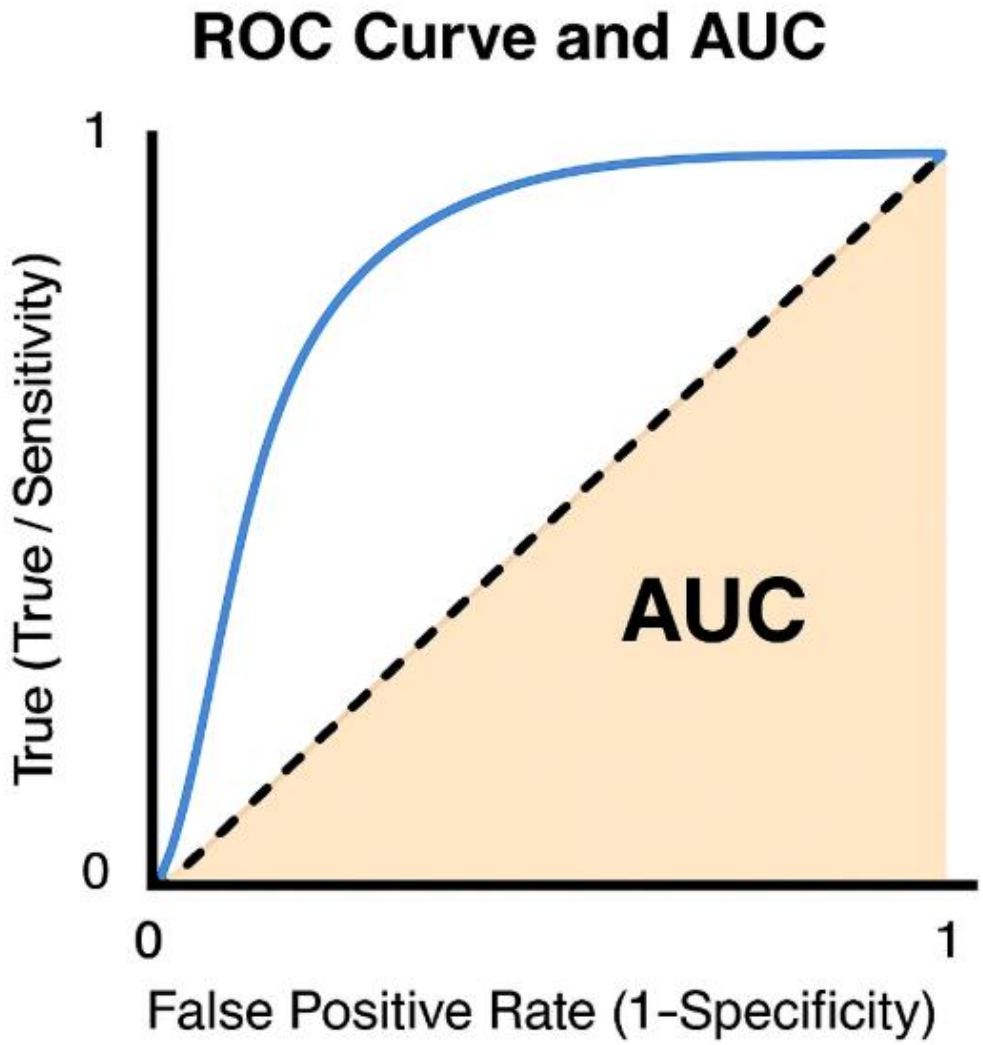
Imagine a **bank predicting loan defaults**:

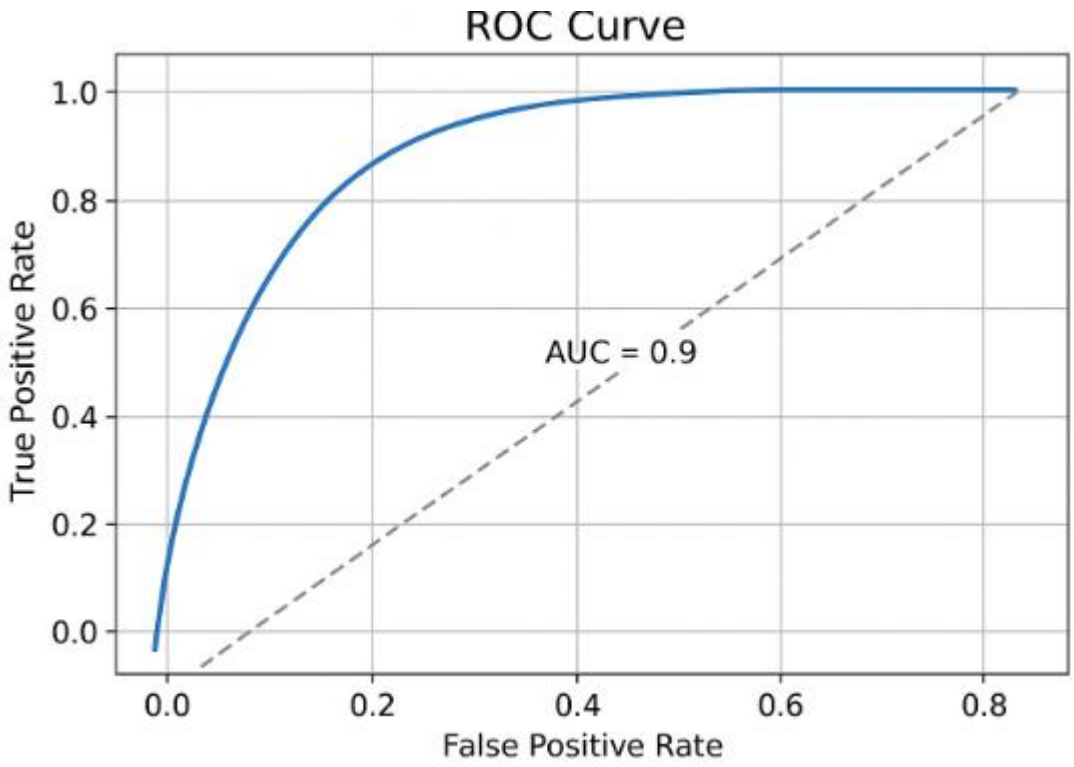
- Model predicts probability that a customer will default.
- **True Positive (TP):** Predicted default and customer actually defaults.
- **False Positive (FP):** Predicted default but customer does **not** default.
- **ROC Curve:** Shows trade-off between catching defaulters (TPR) and wrongly labeling safe customers as risky (FPR).
- **AUC:** Measures overall ability of the model to rank risky customers higher than safe ones.
- **Interpretation:**
    - AUC = 0.9 → model is very good at spotting risky customers
    - AUC = 0.6 → model is weak, but slightly better than random

**So ROC AUC is preferred when:**

- You want a **threshold-independent** evaluation.
- Classes are **imbalanced**.
- You care about **ranking predictions** rather than just classifying at one cutoff.

In short: confusion matrix & F1 = **detailed snapshot at one threshold**, ROC AUC = **overall ability to separate classes**.






**1. Underfitting – Model is too simple**

- **Meaning:** The model **cannot capture the patterns** in the data; it performs poorly on both training and new data.
- **example:**
  Imagine you are learning to recognize fruits, but you only remember **"all fruits are round."**
  - You call an apple round (correct)
  - You also call a banana round(wrong, not always round)
- **Takeaway:** The "rule" is too simple, so you miss important details.

---

**2. Overfitting – Model is too complex**

- **Meaning:** The model **memorizes the training data** perfectly, including noise, but fails on new/unseen data.
- **example:**
  Imagine you memorize **every single fruit in your basket**, including tiny scratches and spots.
  - You recognize your exact apples
  - But when you see a new apple with slightly different spots you get it wrong

- **Takeaway:** The model "learned too much detail" from the training set, so it cannot generalize.

---

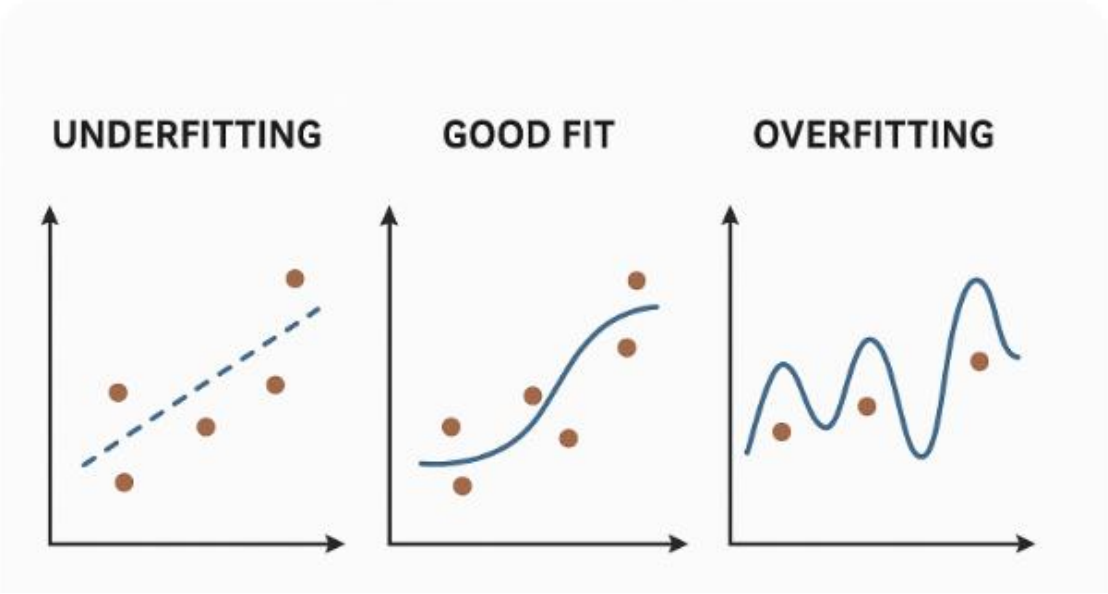
**3. Just right (good fit)**

- **Meaning:** The model learns the important patterns but ignores noise.
- **example:**
  You learn that **apples are usually red or green, round, and have a stem.**
  - You correctly identify most new apples
  - You don't get confused by random scratches or weird spots

---

**summary:**

| Problem | Learning style | Example |
|---|---|---|
| Underfitting | Too simple | "All fruits are round" |
| Overfitting | Too detailed/memorized | Memorizing every single apple's spot |
| Good fit | Just right | Learn key patterns, ignore noise |




**To Deal with the issue of overfitting and underfitting we use the following techniques**

1. **Feature selection:** Picking only the most important inputs (features) that help the model perform well.
2. **Regularization:** A technique to prevent overfitting by adding a penalty to large model coefficients:

- **L1 (Lasso):** Can shrink some coefficients to zero → also does feature selection.
- **L2 (Ridge):** Shrinks coefficients but keeps all → reduces impact of less important features.

**3.** **Cross Validation** – to find optimal model (always on training data )

1. **K-Fold CV:** Split data into **k equal parts**, train on k-1 parts, test on 1 part, repeat k times.
2. **Leave-One-Out CV (LOOCV):** Special case of k-fold where **k = number of samples**; each sample is used once as test.
3. **Stratified K-Fold CV:** Like k-fold, but ensures **each fold has the same class distribution** (useful for
4. **Leave-P-Out CV:** Test the model by leaving **p samples out** each time and training on the rest, repeating for all combinations.

Cross-validation is a way to **test how well a model will perform on unseen data** without having extra data lying around. It helps avoid **overfitting** (model memorizes the training data) and **underfitting** (model is too simple).

---

example

Imagine you are a teacher and you have **100 students**. You want to test if a new teaching method works well.

1. You **can't test it on all students at once** because you also want to know if it works for students you didn't teach.
2. So, you **split the students into 5 groups** (folds).

Then you do this:

- Teach 4 groups with the method, **test on the remaining 1 group**.
- Repeat this **5 times**, each time using a different group as the test group.
- At the end, you **average the test results** to see how well the method works overall.

This is exactly what **5-fold cross-validation** does in machine learning.

---

**Step-by-step in ML terms:**

1. Split your dataset into **k folds** (common: k = 5 or 10).
2. For each fold:
    - Use **k-1 folds for training**
    - Use the remaining fold for **testing/validation**
3. Repeat until every fold has been used as a test set once.
4. Average the results → this gives a **robust estimate** of model performance.

---

**Useful when:**

- You get a **more reliable estimate** of how your model will perform on new data.
- Helps detect **overfitting** early.
- Works even when you **don't have a separate test set**.

---

**Example in ML:**
You have 1000 movies with features like duration, genre, etc., and want to predict if a movie will be a hit. Using 5-fold cross-validation:

- Fold 1: Train on 800, test on 200
- Fold 2: Train on a different 800, test on another 200
- …repeat 5 times
- Average the accuracy → gives a realistic measure of your model's performance.

**K-Fold Cross-Validation**

Instead of testing a model on just one part of your data, you split the data into K parts and test the model K times, each time on a different part, to get a more reliable performance.

**Example**

Imagine you have **100 students** and you want to check how good a new teaching method is.

1. You **divide students into 5 groups** (K = 5).
   Each group has 20 students.
2. **First round:**
    - Use 4 groups (80 students) to train the method.

    - Test on the remaining 1 group (20 students).
3. **Second round:**
    - Use a different 4 groups to train.
    - Test on the group you didn't use before.
4. Repeat until **every group has been tested once**.
5. **Average the results** from all rounds → more reliable estimate of your method's effectiveness.

---

**Why K-Fold?**

- Reduces bias (not dependent on one random split).
- Reduces variance (more stable performance estimate).

---

- Common K values: 5 or 10.
- If K = number of data points → **Leave-One-Out CV**.

---

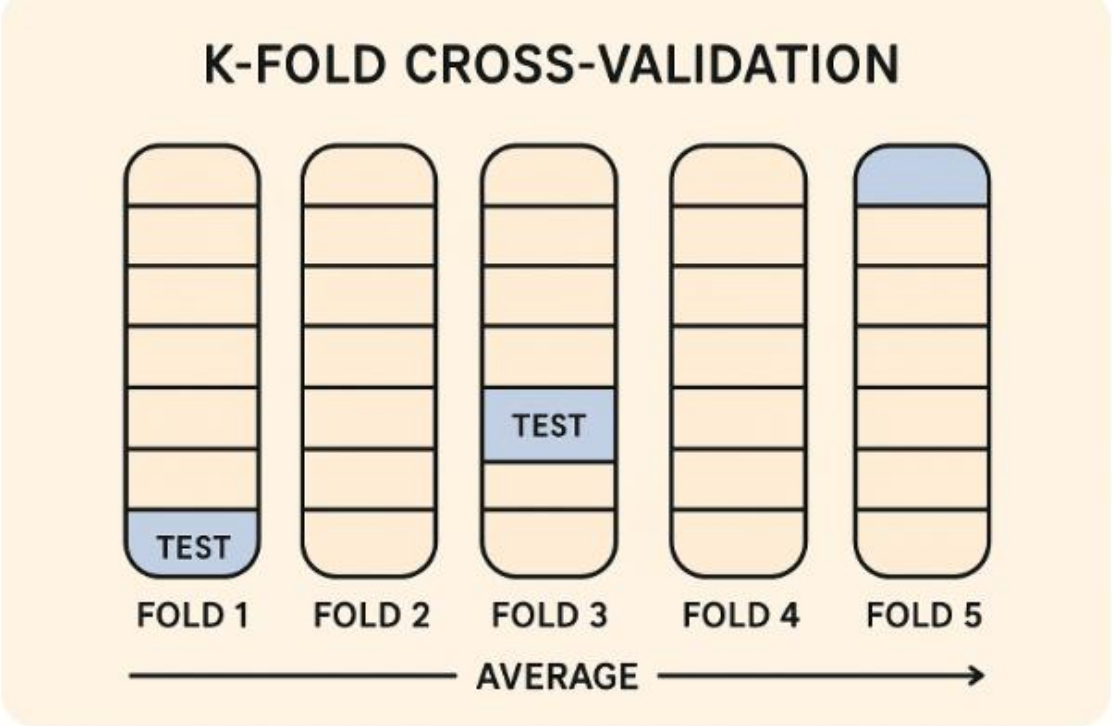
**Range**

Accuracy or score from **0 to 1** (or 0% to 100%).

- Higher → better model.
- Typically, k = 5 or 10.

The mean accuracy of the cross-validation model will be compared with the base model. If there is no major difference between the cross validation accuracy & base model, then we can say the model was trained good & can perform well on unseen data.

But, if there is a major difference between cross validation base model, which says that cross validation model was performing well compared to base model, so we can proceed further with cross-validation model instead of base model.




**Stratified K-Fold Cross-Validation** (when have imbalanced dataset)

It's like K-Fold, but ensures that each fold has the **same proportion of classes** as the original dataset.

---

**Example**

Imagine you have **100 students**, but now:

- 80 are **good performers**
- 20 are **struggling**

You want to test a new teaching method fairly.

1. If you just do regular K-Fold, some groups might end up with **all good students** and no struggling ones → **unfair test**.
2. **Stratified K-Fold:**
    - Split the students into 5 groups (K = 5) **but keep the ratio of 80:20 in each group**.
    - Each fold will have 16 good performers + 4 struggling students.
3. Then, like regular K-Fold:
    - Train on 4 groups → test on 1 group.
    - Rotate until every group has been tested.

4. Average the results → fair estimate of model performance for **all classes**, even the smaller ones.

---

**Why Stratified?**

- Ensures minority classes are represented in every fold.
- Prevents misleading performance scores when classes are imbalanced.

Always use Stratified K-Fold for **classification problems**, especially with imbalanced datasets.

Okay! Let's make it **super super simple** with a tiny story.

**Regular K-Fold vs Stratified K-Fold**

**Scenario:**

You have **10 students**:

- 8 **passed** (P)
- 2 **failed** (F)

You want to test a teaching method using **5-Fold CV** (split into 5 groups).

**Regular K-Fold**

- Randomly split into 5 groups (2 students each).
- Example split:
    - Fold 1: P, P
    - Fold 2: P, P
    - Fold 3: P, P
    - Fold 4: P, F
    - Fold 5: P, F

Problem: Some folds have **no failed students** → testing is not fair.

---

**Stratified K-Fold**

- Split into 5 groups **keeping pass/fail ratio same in all folds**.
- Each fold will have **1 pass + 0 or 1 fail**, keeping ratio 8:2.
- Example split:

- Fold 1: P, F
- Fold 2: P, P
- Fold 3: P, P
- Fold 4: P, P
- Fold 5: P, F

Every fold is **representative** of the whole dataset.

---

**In short:**

- Regular K-Fold = random split
- Stratified K-Fold = **split that preserves class ratios**

Sure! Let's break it down **super simply** with real-life examples.

---

**1. Leave-One-Out (LOO) Cross-Validation**

- **:** Take **one sample** from your dataset as the test set, and use **all the other samples** for training. Repeat this for **every single sample**.
- **Example:**
  Imagine you have **5 students** and you want to predict their exam scores using their study hours.
  - First, leave **Student 1** out → train on Students 2-5 → test on Student 1.
  - Then leave **Student 2** out → train on 1,3,4,5 → test on 2.
  - …do this for all 5 students.
- **Use case:** Works well when your dataset is **very small**, so you want to use **almost all data** for training each time.

---

**2. Leave-P-Out (LPO) Cross-Validation**

- **:** Instead of leaving **1 sample**, you leave **p samples** out as test data. Train on the rest. Repeat for all combinations of p samples.
- **Example:**
  Using the **same 5 students**, if p = 2:
  - Leave Students 1 & 2 out → train on 3,4,5 → test on 1 & 2.
  - Leave Students 1 & 3 out → train on 2,4,5 → test on 1 & 3.
  - …repeat for all combinations of 2 students.

- **Use case:** Useful if you want **more robust testing**, but it becomes **computationally heavy** for large datasets because the number of combinations grows fast.

---

**Difference:**

- LOO = leave 1 out
- LPO = leave p out (p can be more than 1)

**:** If your dataset is tiny → LOO or small p LPO is great. For large datasets → use K-Fold CV instead.\

Difference with example

**Leave-One-Out (LOO)**

| Test | Train |
|---|---|
| A | B, C, D, E |
| B | A, C, D, E |
| C | A, B, D, E |
| D | A, B, C, E |
| E | A, B, C, D |

Each student gets **tested once**, rest are used for training.

---

**Leave-Two-Out (LPO, p = 2)**

| Test | Train |
|---|---|
| A, B | C, D, E |
| A, C | B, D, E |
| A, D | B, C, E |
| A, E | B, C, D |
| B, C | A, D, E |
| B, D | A, C, E |
| B, E | A, C, D |
| C, D | A, B, E |
| C, E | A, B, D |
| D, E | A, B, C |

You **leave 2 students out each time**. As p increases, the number of combinations grows fast.

---

**Observation:**

- LOO is a **special case of LPO** with p = 1.
- LPO gives a more "robust" test but is **costlier to compute**.

# To select the most important feature without domain knowledge in feature selection – to check the contribution of features

## RFE (recursive feature elimination )

RFE is a **feature selection technique** that helps you pick the **most important features** for your model.

- It **recursively removes the least important features** and keeps the ones that contribute the most to prediction.
- Works well with models that can assign **feature importance**, like Logistic Regression, Random Forest, or SVM.

---

**Step-by-step idea**

1. Train the model on all features.
2. Check which feature contributes the least.
3. Remove that feature.
4. Repeat until you are left with the desired number of features.

---

**Example**

**Problem:** You want to predict whether a student will pass an exam.
**Features you have:**

- Hours studied
- Sleep hours
- Class attendance
- Social media time
- Breakfast eaten

**Step 1: Start with all features**

- Model is trained on all 5 features.

**Step 2: Check importance**

- Model shows “Breakfast eaten” contributes the least.

**Step 3: Remove least important feature**

- Remove “Breakfast eaten” and retrain the model with remaining 4 features.

**Step 4: Repeat**

- Next least important is “Social media time”. Remove it.
- Continue until only the top features remain, e.g., “Hours studied” and “Class attendance”.

**Result:** You now have the most important factors affecting exam performance:

- Hours studied
- Class attendance

This way, your model is **simpler, faster, and often more accurate**, because you removed noise or irrelevant features.

# To select the most important feature without domain knowledge

## Univariate Feature Selection

- It selects the **best features individually** based on their relationship with the **target variable**.
- It uses **statistical tests** (like chi-square for categorical data, ANOVA for continuous data).
- “Univariate” means **looking at one feature at a time**.

---

**Step-by-step idea**

1. Take **one feature**.
2. Measure how **strongly it’s related** to the target variable using a statistical test.
3. Rank all features based on the score.
4. Keep the top features, discard the weak ones.

---

**Example**

**Problem:** You want to predict whether a student will pass an exam.
**Features you have:**

- Hours studied (continuous)
- Sleep hours (continuous)
- Class attendance (continuous)

- Social media time (continuous)
- Breakfast eaten (categorical: Yes/No)

**Step 1: Test each feature individually**

- Use **ANOVA test** for continuous features.
- Use **chi-square test** for categorical features.

**Step 2: Rank features based on score**

| **Feature** | **Score** | **Decision** |
|---|---|---|
| Hours studied | 90 | Keep |
| Class attendance | 85 | Keep |
| Sleep hours | 40 | Maybe discard |
| Social media time | 20 | Discard |
| Breakfast eaten | 15 | Discard |

**Step 3: Select top features**

- Keep “Hours studied” and “Class attendance” because they are **most strongly related** to passing.

**Result:** You now have a simplified dataset with only the features that actually matter for prediction.