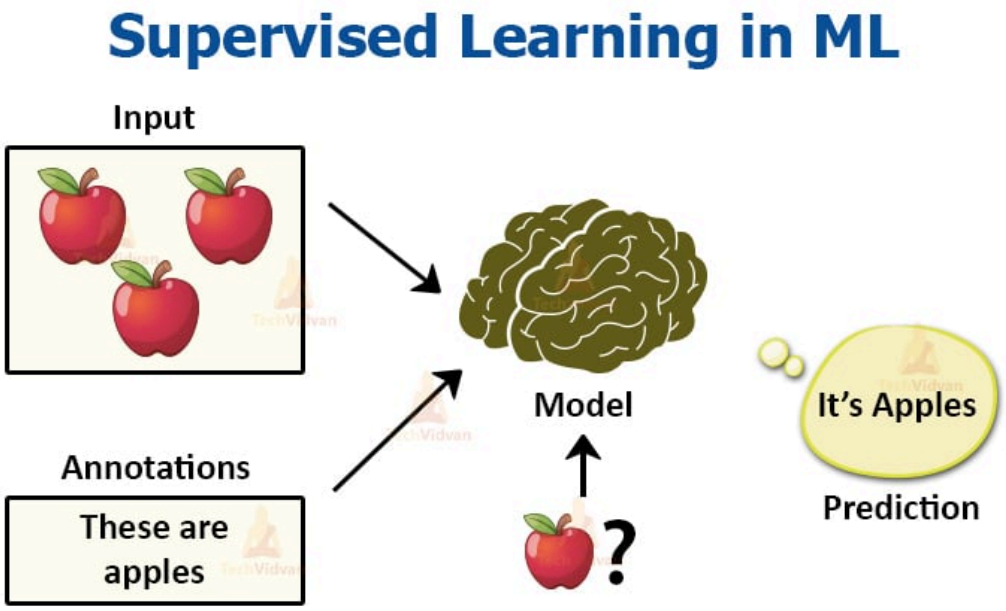
📘 Supervised Machine Learning (ML)

🧠 What is Supervised Machine Learning?

Supervised Learning is a type of Machine Learning where:

- The model is **trained on labeled data**.
- It **learns the relationship** between **input features (X)** and **output labels (Y)**.
- The goal is to **predict outcomes** for new, unseen data.



🗒️ Example:

Hours Studied (X)	Marks Scored (Y)
2	50
4	70
6	90

The model learns a mapping like:

f(X) = Y → f(4) = 70

🧪 Applications

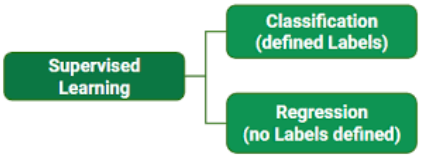
- Email spam detection
- Loan approval
- Disease prediction
- Stock price prediction
- House price estimation

🧩 Components

- **Input Features (X)**: Independent variables (e.g., age, salary)
- **Output Label (Y)**: Dependent variable (e.g., approved = Yes/No)
- **Model**: Learns from training data
- **Loss Function**: Measures error
- **Optimizer**: Reduces error (e.g., Gradient Descent)

🔀 Types of Supervised Learning

There are mainly **two** types:



🔵 1. Regression

- **Output is continuous**
- Predicts quantities like price, salary, temperature

📌 Algorithms:

- Linear Regression
- Decision Tree Regressor
- Random Forest Regressor
- SVR (Support Vector Regression)

📊 Example:

Predict house price based on area, location, etc.

🔴 2. Classification

- **Output is categorical**
- Classifies data into groups like "spam" or "not spam"

📌 Algorithms:

- Logistic Regression
- K-Nearest Neighbors (KNN)
- Decision Tree Classifier

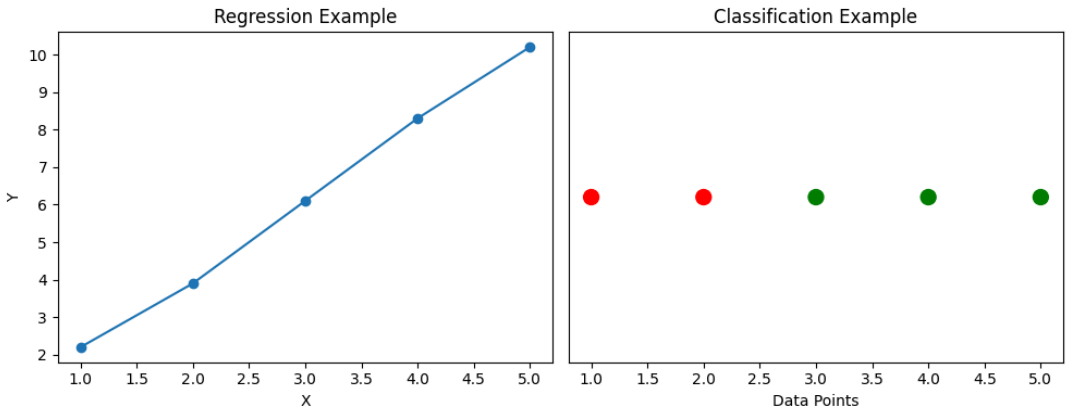
- Random Forest Classifier
- Support Vector Machine (SVM)
- Naive Bayes

📊 **Example:**

Detect whether a tumor is **benign** or **malignant**

📉 Visualizing Regression vs Classification

```
import matplotlib.pyplot as plt

# Dummy regression plot
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
x = [1, 2, 3, 4, 5]
y = [2.2, 3.9, 6.1, 8.3, 10.2]
plt.plot(x, y, marker='o')
plt.title('Regression Example')
plt.xlabel('X')
plt.ylabel('Y')

# Dummy classification plot
plt.subplot(1, 2, 2)
x1 = [1, 2, 3, 4, 5]
y1 = ['No', 'No', 'Yes', 'Yes', 'Yes']
colors = ['red' if val == 'No' else 'green' for val in y1]
plt.scatter(x1, [1]*5, c=colors, s=100)
plt.yticks([])
plt.title('Classification Example')
plt.xlabel('Data Points')

plt.tight_layout()
plt.show()
```



📘 Simple Linear Regression (SLR)

🔎 What is Simple Linear Regression?

Simple Linear Regression is a type of **regression algorithm** used to predict a **continuous value** based on **one independent variable**.

🔁 Formula:

$$Y = mX + c$$

Where:

- Y = predicted value (e.g., Package)
- X = independent variable (e.g., CGPA)
- m = slope (coefficient)
- c = intercept

📈 Real-Life Example:

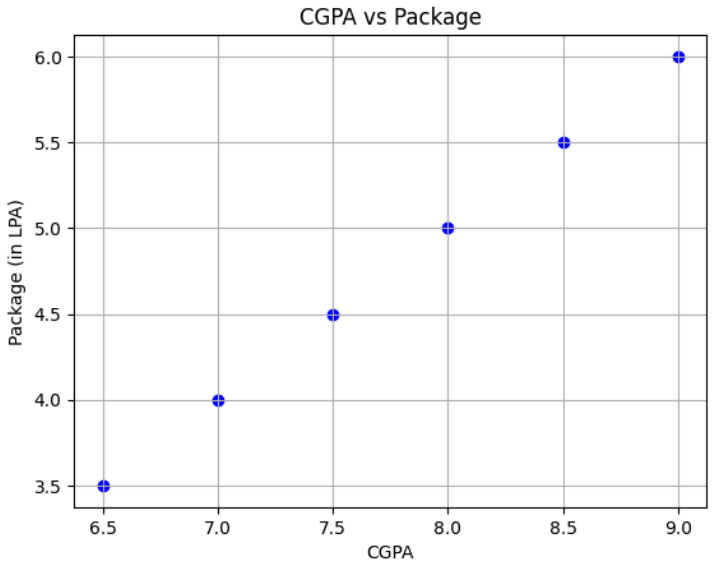
Predicting a student's **salary package** (Y) based on their **CGPA** (X).

📦 Dataset Example

CGPA	Package (LPA)
6.5	3.5
7.0	4.0
7.5	4.5
8.0	5.0
8.5	5.5
9.0	6.0

📊 Visualize the Data

```
import pandas as pd
import matplotlib.pyplot as plt

# Sample dataset
data = {
    'CGPA': [6.5, 7.0, 7.5, 8.0, 8.5, 9.0],
    'Package': [3.5, 4.0, 4.5, 5.0, 5.5, 6.0]
}

df = pd.DataFrame(data)

# Scatter plot
plt.scatter(df['CGPA'], df['Package'], color='blue')
plt.title('CGPA vs Package')
plt.xlabel('CGPA')
plt.ylabel('Package (in LPA)')
plt.grid(True)
plt.show()
```



⚙️ Step-by-Step Model Building

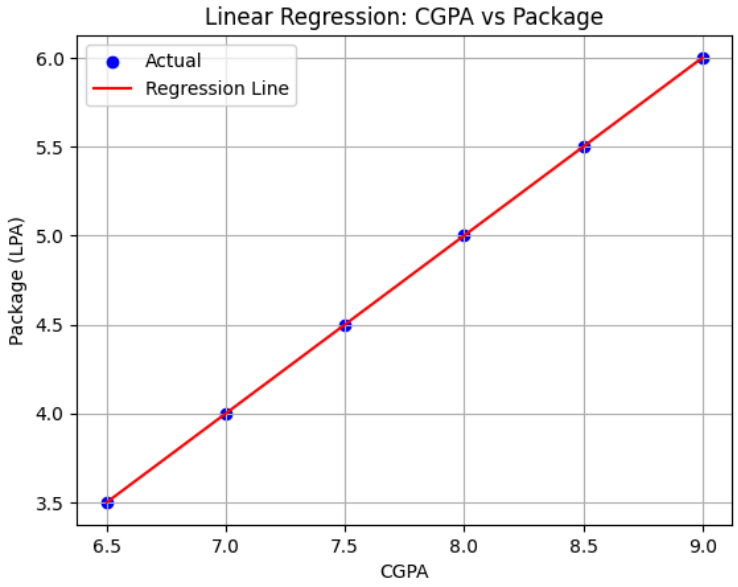
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# 1. Features & Labels
X = df[['CGPA']]  # Independent variable (2D)
y = df['Package']  # Dependent variable

# 2. Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

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

# 4. Predict
y_pred = model.predict(X_test)

# 5. Evaluation
print("Slope (m):", model.coef_[0])
print("Intercept (c):", model.intercept_)
print("R2 Score:", r2_score(y_test, y_pred))
```

🔍 Predict New Package Based on CGPA

```
# Predict package for CGPA = 8.2
cgpa_input = [[8.2]]
predicted_package = model.predict(cgpa_input)
print(f"Predicted Package for CGPA 8.2 is: ₹{predicted_package[0]:.2f} LPA")
```

📈 Visualize Regression Line

```
# Plot the line with training data
plt.scatter(X, y, color='blue', label='Actual')
plt.plot(X, model.predict(X), color='red', label='Regression Line')
plt.title('Linear Regression: CGPA vs Package')
plt.xlabel('CGPA')
plt.ylabel('Package (LPA)')
plt.legend()
plt.grid(True)
plt.show()
```



📊 Output Sample

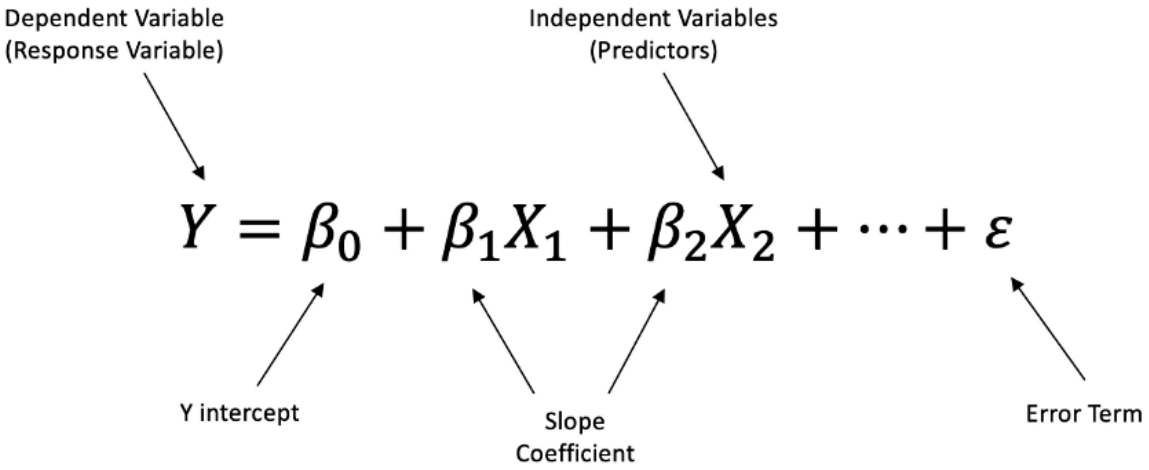
Slope (m): 1.0
Intercept (c): -3.0
R2 Score: 1.0
Predicted Package for CGPA 8.2 is: ₹5.20 LPA

Multiple Linear Regression - Notes

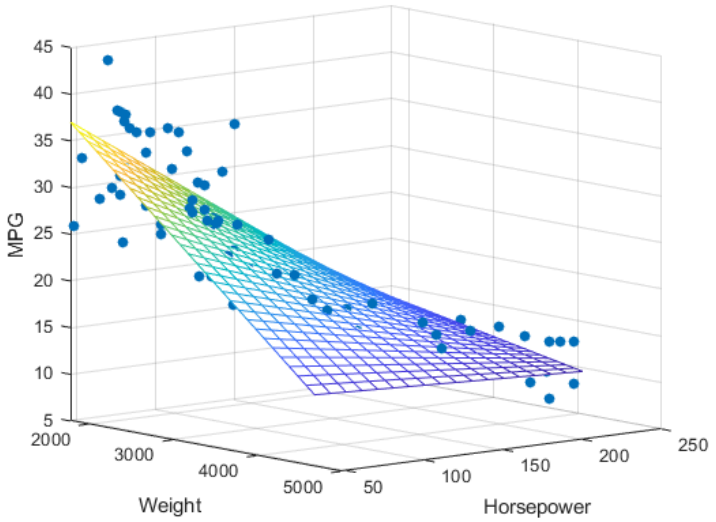
What is Multiple Linear Regression?

Multiple Linear Regression is a supervised machine learning algorithm that predicts a continuous target variable based on **two or more independent variables**. It is an extension of simple linear regression which uses only one independent variable.

Equation:



- **Y** = dependent variable (target)
- **X1, X2, ..., Xn** = independent variables (features)
- **b0** = intercept
- **b1, b2, ..., bn** = coefficients
- **\epsilon** = error term



Assumptions of Multiple Linear Regression:

1. Linearity: Relationship between features and target is linear.
2. Independence: Observations are independent.
3. Homoscedasticity: Constant variance of errors.
4. Normality: Residuals are normally distributed.
5. No multicollinearity: Independent variables should not be highly correlated.

Steps to Implement Multiple Linear Regression

1. Import Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

2. Create Custom Dataset

```
data = pd.DataFrame({
    "CGPA": [6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 9.8, 8.2, 7.7],
    "Experience": [0, 1, 1, 2, 2, 3, 3, 4, 2.5, 1.5],
    "Package": [3.0, 3.5, 4.0, 4.5, 5.0, 6.0, 6.5, 7.0, 5.2, 4.2]
})
print(data)
```

3. Define Features and Target

```
X = data[["CGPA", "Experience"]]   # example features
y = data["Package"]                # target variable
```

4. Train-Test Split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

5. Train the Model

```
model = LinearRegression()
model.fit(X_train, y_train)
```

6. Make Predictions

```
y_pred = model.predict(X_test)
```

7. Evaluate the Model

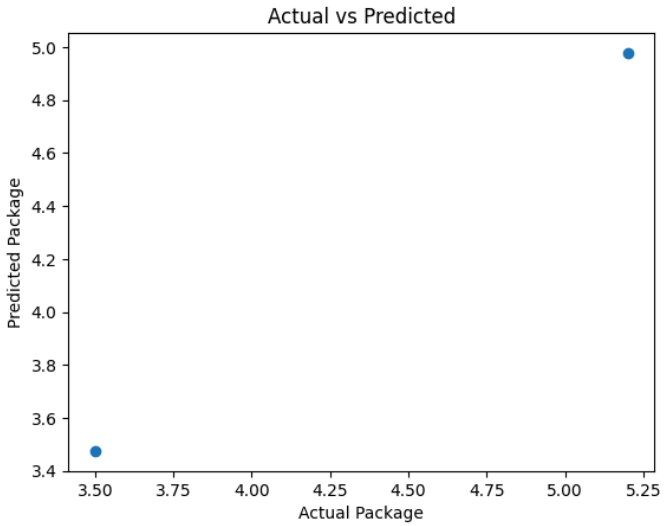
```
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
print("R-squared Score:", r2_score(y_test, y_pred))
```

8. Coefficients

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

Visualization Example (Optional)

```
# Plotting actual vs predicted
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Package")
plt.ylabel("Predicted Package")
plt.title("Actual vs Predicted")
plt.show()
```



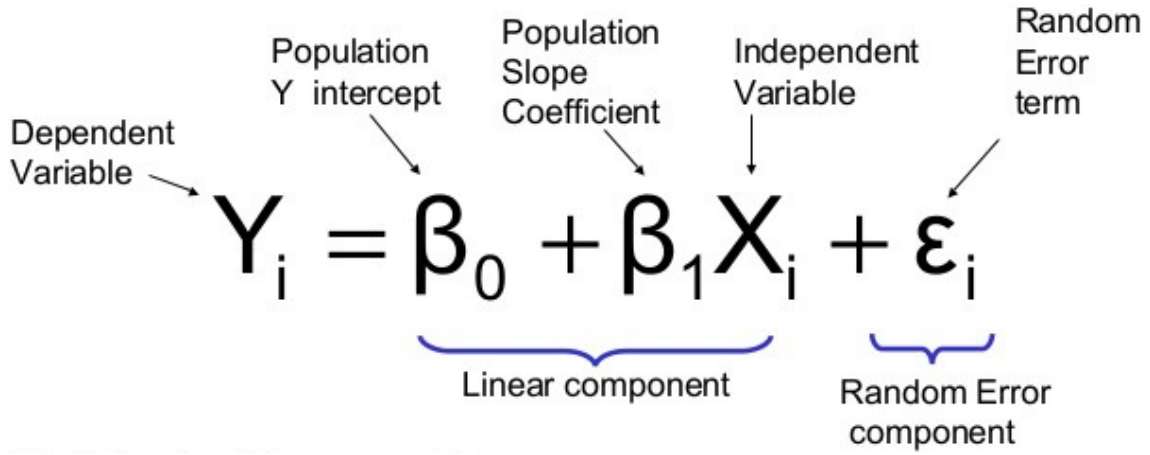
Applications:

- Predicting salary based on education, experience, etc.
- Estimating house prices based on area, location, age.
- Sales prediction using marketing budget, region, seasonality, etc.

📌 Polynomial Regression

1. What is Polynomial Regression?

Polynomial Regression is an extension of Linear Regression where the relationship between the independent variable xxx and dependent variable yyy is modeled as an nnn-degree polynomial.



It's useful when the data shows a **curved** or **non-linear** relationship.

---

2. Why Use Polynomial Regression?

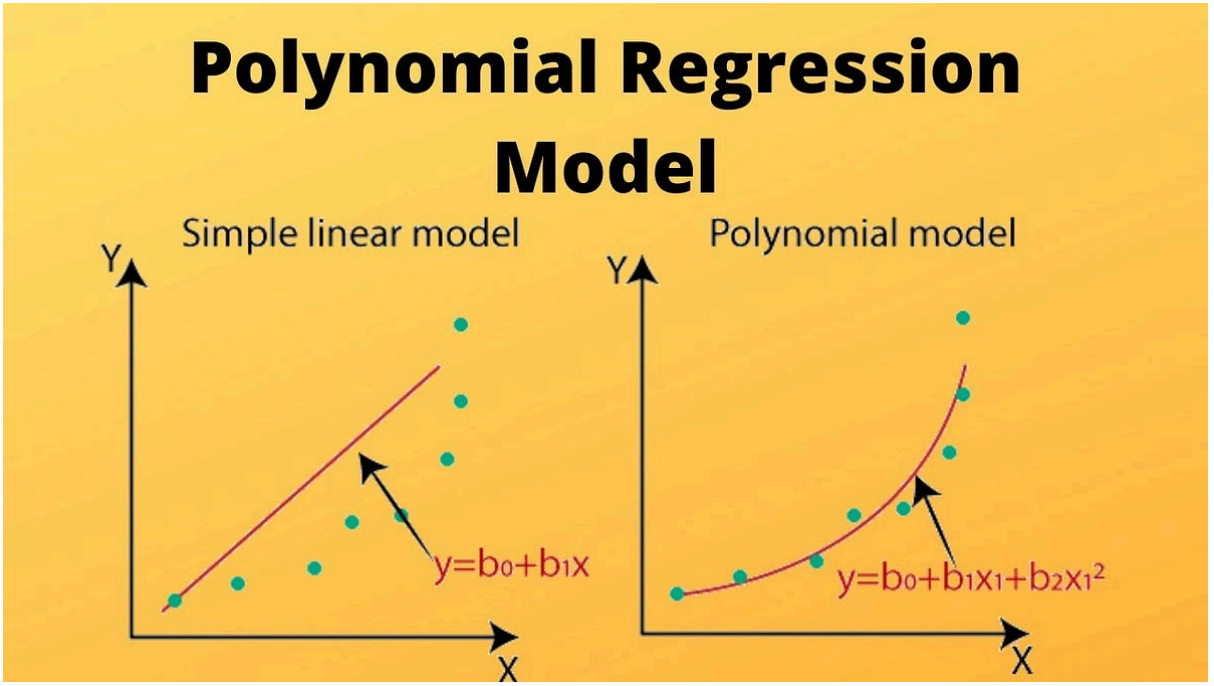
- Linear regression fails to capture curves in data.
- Polynomial regression adds **powers of features** to model non-linearity.
- It still uses **Linear Regression under the hood**, but on **transformed polynomial features**.

---

3. Steps in Polynomial Regression

1. Load dataset
2. Identify independent (XXX) and dependent (YYY) variables
3. Transform features to polynomial form using PolynomialFeatures
4. Train a linear regression model on transformed features

5. Visualize predictions vs actual values

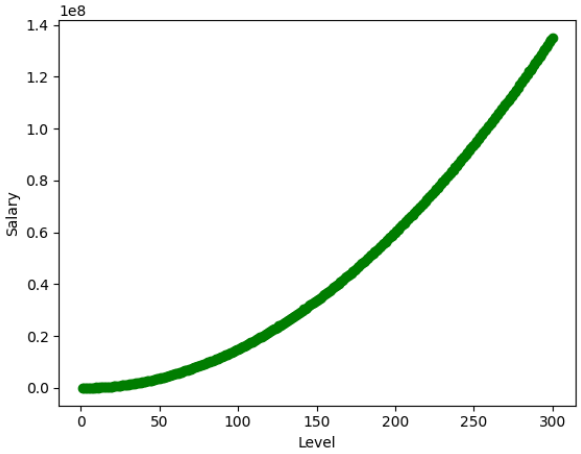


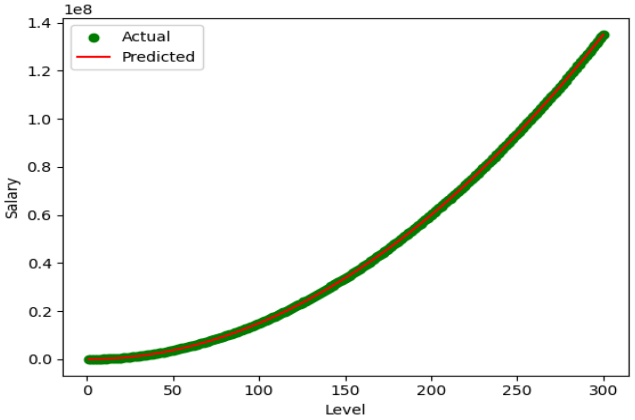
---

4. Example with Your Code

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Step 1: Load dataset
dataset = pd.read_csv("Data_Set.csv")
print(dataset.head(3))

# Step 2: Visualize data
plt.scatter(dataset["Level"], dataset["Salary"], color='green')
plt.xlabel("Level")
plt.ylabel("Salary")
plt.show()

# Step 3: Correlation
```

```
print(dataset.corr())

# Step 4: Prepare data
x = dataset[["Level"]]
y = dataset[["Salary"]]

# Step 5: Transform features to polynomial
pf = PolynomialFeatures(degree=2)
pf.fit(x)
x_poly = pf.transform(x)

# Step 6: Split data
x_train, x_test, y_train, y_test = train_test_split(x_poly, y, test_size=0.2, random_state=42)

# Step 7: Train model
lr = LinearRegression()
lr.fit(x_train, y_train)

# Step 8: Model accuracy
print("Accuracy:", lr.score(x_test, y_test) * 100)

# Step 9: Visualization
plt.scatter(dataset["Level"], dataset["Salary"], color='green')
plt.plot(dataset["Level"], lr.predict(pf.transform(dataset[["Level"]])), color='red')
plt.xlabel("Level")
plt.ylabel("Salary")
plt.legend(["Predicted", "Actual"])
plt.show()
```





Key Points

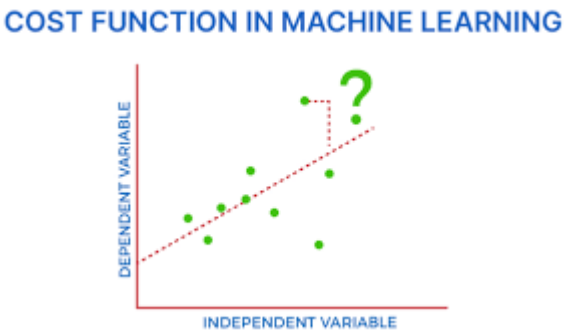
- **Degree selection:** Higher degree → more flexibility, but risk of overfitting.
- **Overfitting:** Too many polynomial terms can make the model fit training data too perfectly but fail in new data.
- **Use cross-validation** to choose the right degree.

Cost Function in Machine Learning

Definition

A **Cost Function** (also called **Loss Function** or **Error Function**) measures **how well a model's predictions match the actual target values**.
It calculates the **difference between predicted values** (y^\hat{y}y^) and **actual values** (yyy) and returns a **single number** representing the error.

The goal of training is to **minimize** the cost function.



Mathematical Representation

For a dataset with m samples:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Loss}(\hat{y}^{(i)}, y^{(i)})$$

- $J(\theta)$ → Cost function value
- m → Number of training examples
- $\hat{y}$ → Predicted value
- y → Actual value
- θ → Model parameters

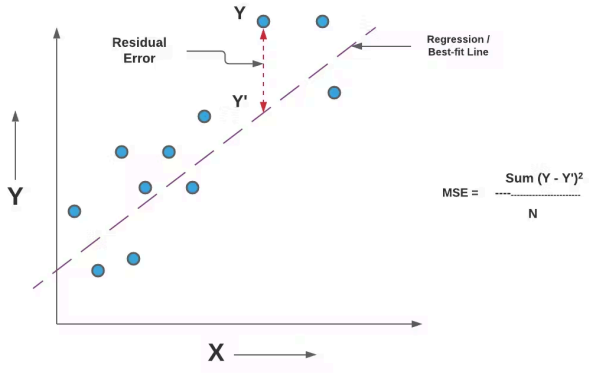
Types of Cost Functions

1. Mean Squared Error (MSE) (Used in Regression)

$$\text{MSE} = \boxed{\frac{1}{n}\sum_{i=1}^{n}\boxed{(Y_i - \hat{Y}_i)^2}}$$

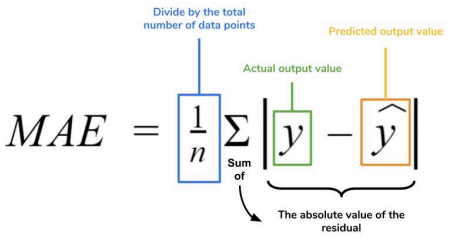
Mean — Error — Squared

- Squares the error → penalizes large errors more.
- Always positive.
- Common in **Linear Regression**.



---

2. Mean Absolute Error (MAE) (Used in Regression)



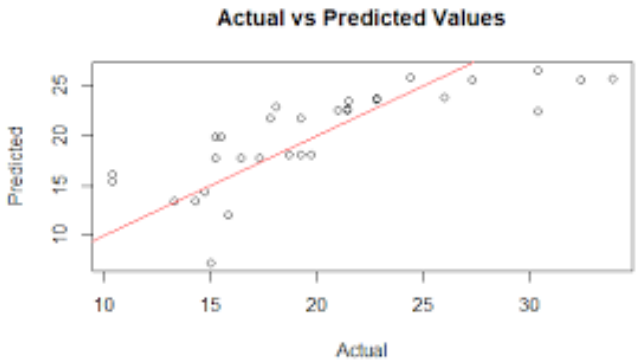
- Takes the **absolute difference** between predicted and actual values.
- Less sensitive to outliers compared to MSE.

---

3. Root Mean Squared Error (RMSE)

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

- Square root of MSE.
- Has same units as the target variable.



Regularization in Machine Learning

Definition

Regularization is a technique used to **reduce overfitting** in machine learning models by **adding a penalty to the model's complexity**.
It discourages the model from learning **too complex patterns** or **memorizing noise** in the training data.

Why Regularization is Needed

- Complex models can fit the **training data perfectly** but fail on **new/unseen data** → **overfitting**.
- Regularization adds **constraints or penalties** to reduce overfitting and improve **generalization**.

How Regularization Works

In linear regression, the cost function is modified as:

$$J(\theta) = \text{Loss} + \text{Penalty Term}$$

Where:

- **Loss** → Original cost function (MSE)
- **Penalty Term** → Controls model complexity

Types of Regularization

1. L1 Regularization (Lasso Regression)

Cost Function=

$$\text{Cost Function} = \text{MSE} + \lambda \sum_{j=1}^{n} |\theta_j|$$

- Adds **absolute value** of coefficients as penalty.
- Can **shrink some coefficients to zero**, effectively performing **feature selection**.
- Useful when you want a **sparse model**.

---

2. L2 Regularization (Ridge Regression)

Cost Function=

$$\text{Cost Function} = \text{MSE} + \lambda \sum_{j=1}^{n} \theta_j^2$$

- Adds **squared value** of coefficients as penalty.
- Reduces the magnitude of coefficients but **doesn't make them zero**.
- Useful when you have **multicollinearity** or many small/medium-sized features.

Key Parameters

- **λ (lambda) / alpha** → Regularization strength
    - Higher λ → more penalty → simpler model
    - Lower λ → less penalty → more flexible model

Visual Example

- Without regularization: Model fits all training points → Overfitting

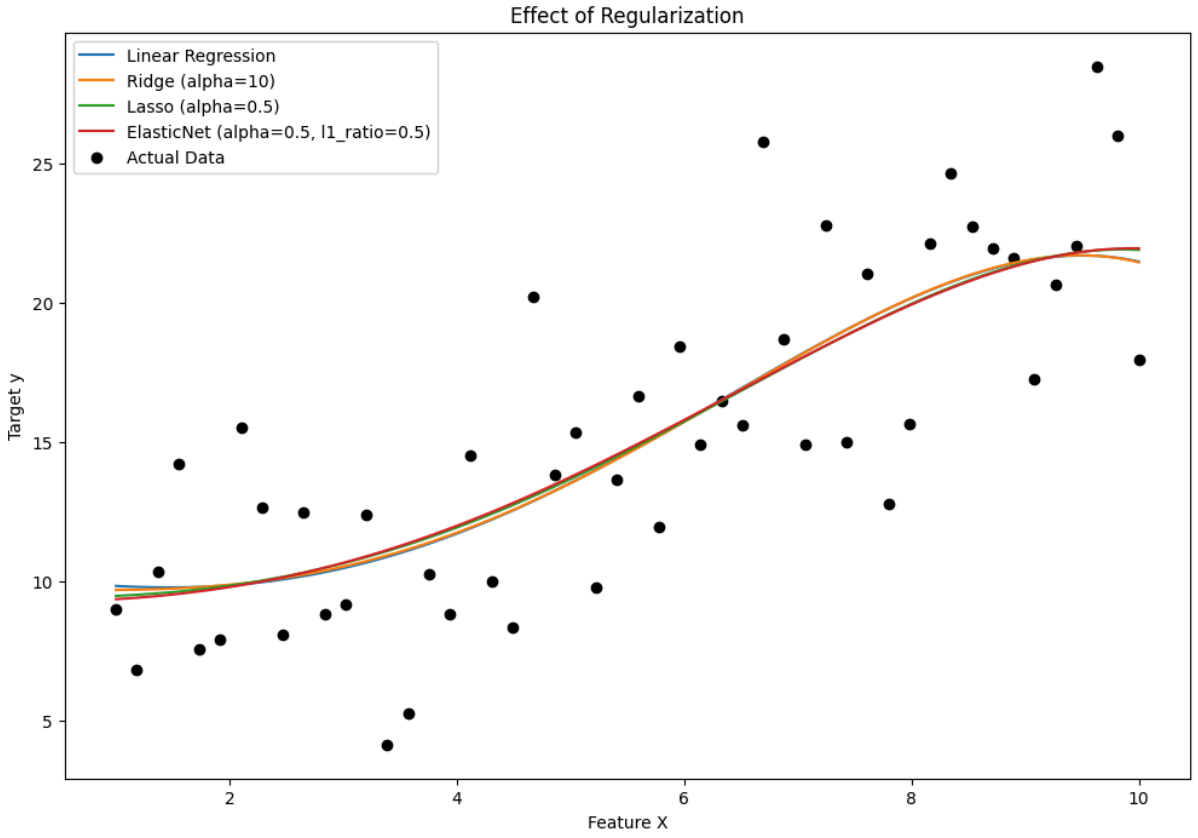
- With regularization: Model is smoother → Better generalization

---

Regularization Example in Python

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Step 1: Create a custom dataset
np.random.seed(42)
X = np.linspace(1, 10, 50).reshape(-1,1)
y = 2*X.flatten() + 5 + np.random.randn(50)*4  # linear with noise

# Introduce some more features to show regularization effect
X = np.hstack([X, X**2, X**3, X**4])  # polynomial features

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 2: Train models
models = {
    "Linear Regression": LinearRegression(),
    "Ridge (alpha=10)": Ridge(alpha=10),
    "Lasso (alpha=0.5)": Lasso(alpha=0.5),
    "ElasticNet (alpha=0.5, l1_ratio=0.5)": ElasticNet(alpha=0.5, l1_ratio=0.5)
}

# Step 3: Fit and predict
plt.figure(figsize=(12,8))
X_plot = np.linspace(1,10,100).reshape(-1,1)
X_plot_poly = np.hstack([X_plot, X_plot**2, X_plot**3, X_plot**4])

for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_plot_poly)
    plt.plot(X_plot, y_pred, label=name)
```

```
# Step 4: Plot actual points
plt.scatter(X[:,0], y, color='black', label='Actual Data')
plt.xlabel("Feature X")
plt.ylabel("Target y")
plt.title("Effect of Regularization")
plt.legend()
plt.show()
```



Explanation

1. We created a **non-linear dataset** with polynomial features.
2. We trained:
    - **Linear Regression** → no regularization.
    - **Ridge** → L2 regularization.
    - **Lasso** → L1 regularization.

- **ElasticNet** → combination of L1 + L2.

3. The **plot shows** how:
    - Linear Regression overfits the noisy data.
    - Regularized models are smoother and generalize better.

All Source Code:- Github

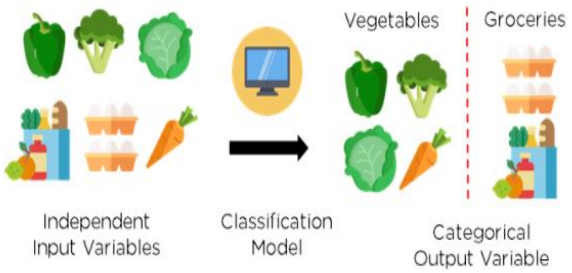
Classification Analysis

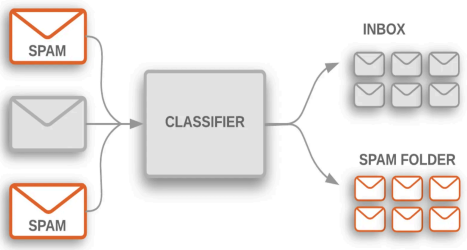
1. What is Classification Analysis?

Classification is a **supervised machine learning** technique used to predict **categorical outcomes** (discrete classes) based on input features.
Unlike regression (which predicts continuous values), classification assigns data points to **predefined labels**.

Examples:

- Spam email detection (Spam / Not Spam)
- Disease diagnosis (Positive / Negative)
- Image recognition (Dog / Cat / Bird)
- Credit approval (Approved / Rejected)





---

2. How Classification Works

1. **Training phase** – The algorithm learns patterns from labeled data.
2. **Testing phase** – The model predicts labels for new, unseen data.
3. **Evaluation** – Accuracy, Precision, Recall, and F1-score are calculated to measure performance.

---

3. General Workflow

1. **Data Collection** – Gather labeled data.
2. **Data Preprocessing** – Handle missing values, encode categorical variables, normalize features.
3. **Model Selection** – Choose a classification algorithm.
4. **Training** – Fit the model with training data.
5. **Prediction** – Classify new data points.
6. **Evaluation** – Use metrics to check accuracy.

---

4. Types of Classification Algorithms

A. Binary Classification

Predicts one of two possible outcomes.
Example: Fraud (Yes / No)

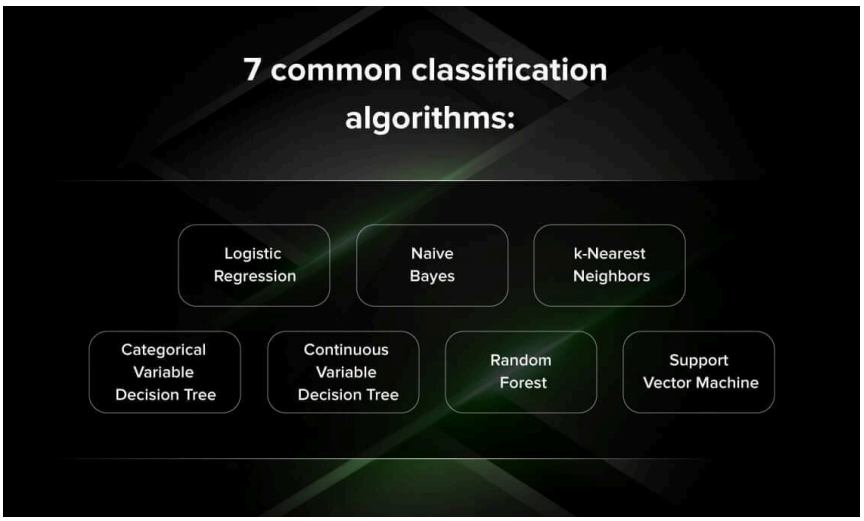
B. Multi-Class Classification

Predicts one out of three or more classes.
Example: Handwritten digit recognition (0–9)

C. Multi-Label Classification

Each instance can belong to multiple categories.
Example: Movie tagging (Action, Drama, Comedy)

5. Common Classification Algorithms



Algorithm	Description	Advantages	Disadvantages
Logistic Regression	Linear model for binary classification.	Simple, interpretable.	Not suitable for complex non-linear problems.
K-Nearest Neighbors (KNN)	Classifies based on closest data points.	No training phase, simple.	Slow for large datasets.
Support Vector Machine (SVM)	Finds the optimal boundary between classes.	Works well with high-dimensional data.	Memory-intensive, not great for large datasets.
Decision Tree	Splits data based on feature values.	Easy to visualize.	Can overfit without pruning.
Random Forest	Ensemble of decision trees.	High accuracy, reduces overfitting.	Less interpretable.
Naive Bayes	Based on probability (Bayes theorem).	Works well with text data.	Assumes feature independence.
Neural Networks	Deep learning model for complex patterns.	High accuracy, adaptable.	Needs lots of data and computing power.

---

6. Evaluation Metrics

- **Accuracy:**

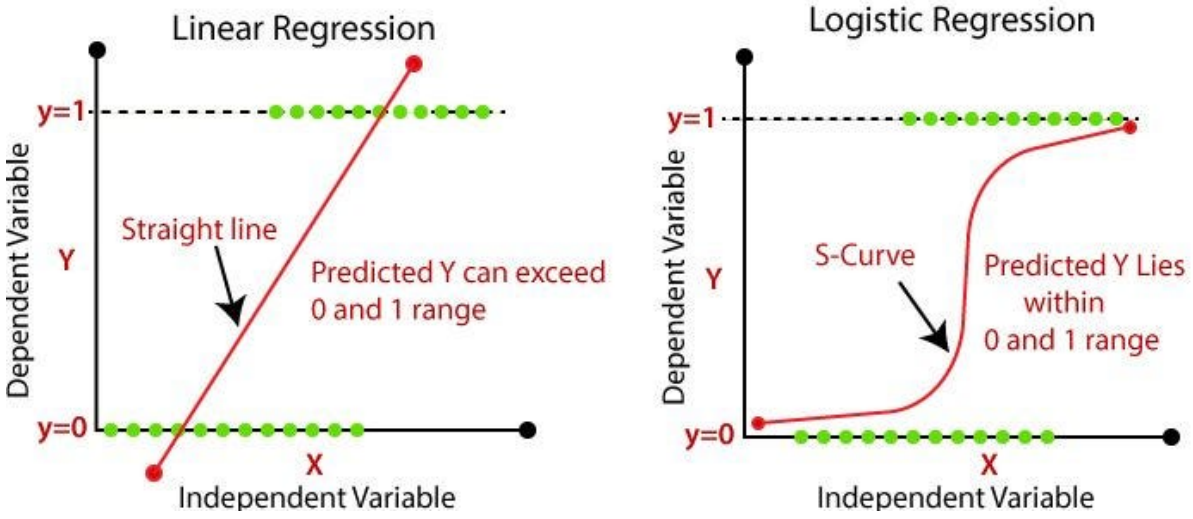
$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

- **Precision:** How many predicted positives are actually correct.
- **Recall (Sensitivity):** How many actual positives were predicted correctly.
- **F1-Score:** Harmonic mean of precision and recall.
- **Confusion Matrix:** Table showing true positives, true negatives, false positives, false negatives.

Logistic Regression – Notes

1. What is Logistic Regression?

- Logistic Regression is a **supervised learning algorithm** used for **classification problems**.
- Despite the name, it is used for predicting **categorical outcomes** (e.g., Yes/No, Pass/Fail, Spam/Not Spam).
- It works by estimating the **probability** that an instance belongs to a particular class.



---

2. Why Not Linear Regression for Classification?

- Linear Regression outputs **continuous values** — not probabilities between 0 and 1.
- Logistic Regression applies a **sigmoid function** to map predictions to the range **(0, 1)**, making them interpretable as probabilities.

---

3. Sigmoid Function

The **sigmoid function** transforms any real value into a probability:

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

Where:

- $z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$
- Output range: (0, 1)

---

4. Decision Boundary

- Logistic Regression predicts **1** if the probability ≥ 0.5, else predicts **0**.
- The **threshold** can be adjusted based on the problem.

---

5. Cost Function

Since logistic regression deals with probabilities, we use **Log Loss (Cross-Entropy Loss)**:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]$$

---

6. Steps in Logistic Regression

1. Import libraries & load dataset
2. Preprocess data (handle missing values, encoding, scaling)
3. Split into training & test sets
4. Train logistic regression model
5. Predict on test data

6. Evaluate using metrics (accuracy, precision, recall, F1-score, ROC curve)

---

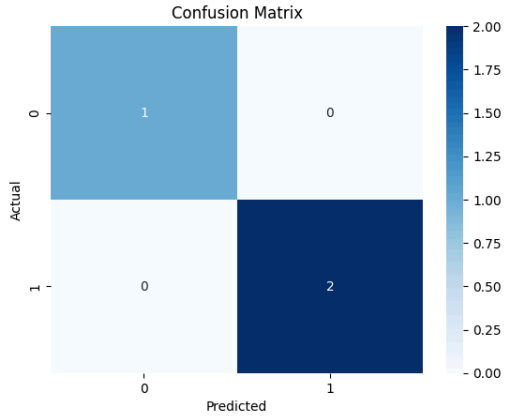
7. Example – Logistic Regression in Python

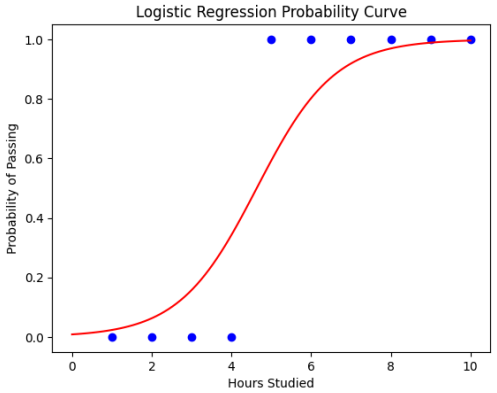
Let's create a **custom dataset** for "Exam pass prediction based on hours studied."

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns

# Step 1: Create custom dataset
data = pd.DataFrame({
    'Hours_Studied': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'Pass': [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
})

# Step 2: Features & target
X = data[['Hours_Studied']]
y = data['Pass']

# Step 3: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Train Logistic Regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Step 5: Predictions
y_pred = model.predict(X_test)

# Step 6: Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Confusion Matrix
```

```python
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# Step 7: Probability curve
import numpy as np
X_range = np.linspace(0, 10, 100).reshape(-1, 1)
y_prob = model.predict_proba(X_range)[:, 1]

plt.plot(X_range, y_prob, color='red')
plt.scatter(X, y, color='blue')
plt.xlabel("Hours Studied")
plt.ylabel("Probability of Passing")
plt.title("Logistic Regression Probability Curve")
plt.show()
```

```
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         2

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3
```





---

8. Key Points

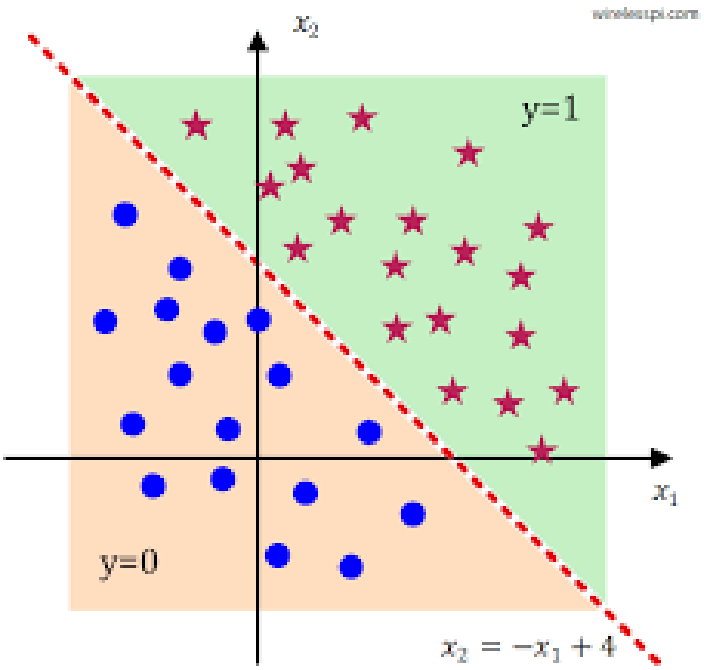
- **Output:** Probability of belonging to a class.
- **Evaluation metrics:** Accuracy, Precision, Recall, F1-score, ROC-AUC.
- **Best for:** Binary classification problems.
- **Limitations:** Assumes linear relationship between independent variables and log-odds.

Multiple Input Logistic Regression – Notes with Visualization

1. What is Multiple Input Logistic Regression?

Multiple Input Logistic Regression is a classification algorithm used when:

- Target variable is **categorical** (binary or multi-class).
- There are **two or more independent variables**.
- Predicts the **probability** of the target class using a **logistic (sigmoid) function**.



---

2. Logistic Function Formula

$$P(Y=1) = \frac{1}{1 + e^{-(b_0 + b_1 X_1 + b_2 X_2 + \ldots + b_n X_n)}}$$

---

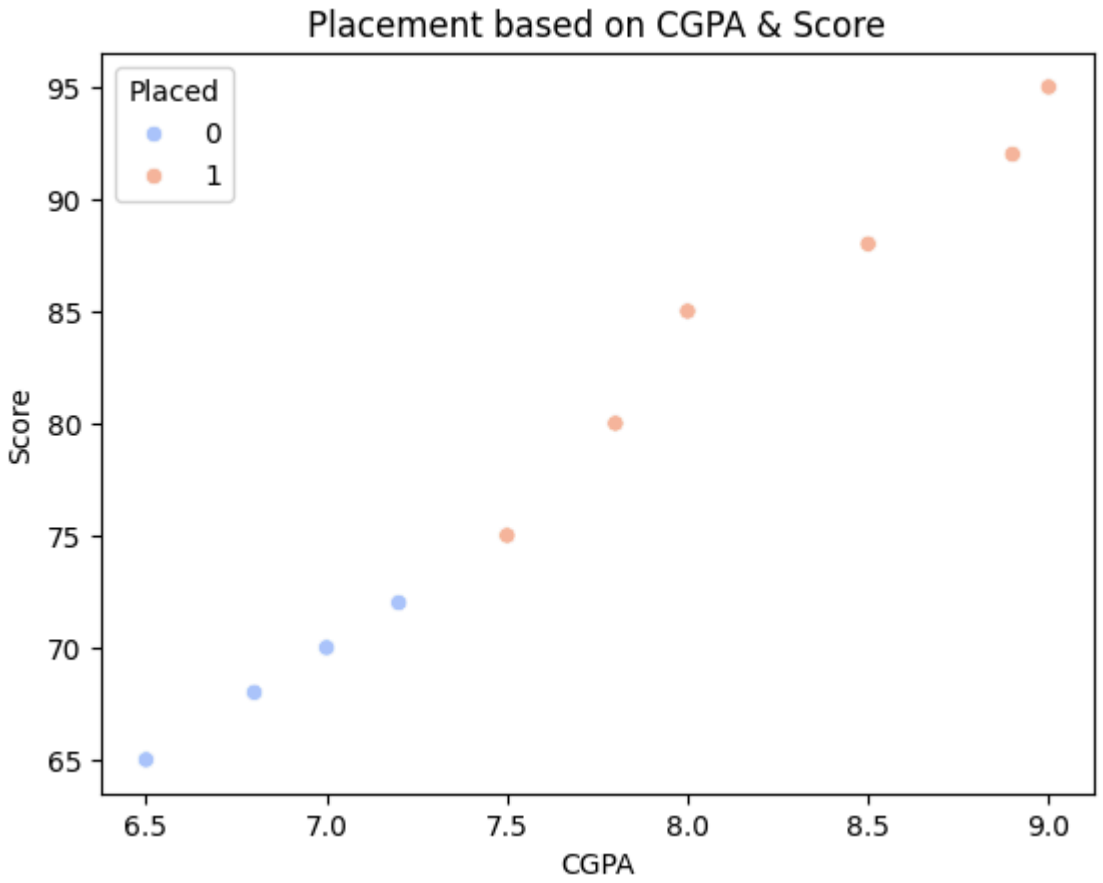
3. Implementation Steps

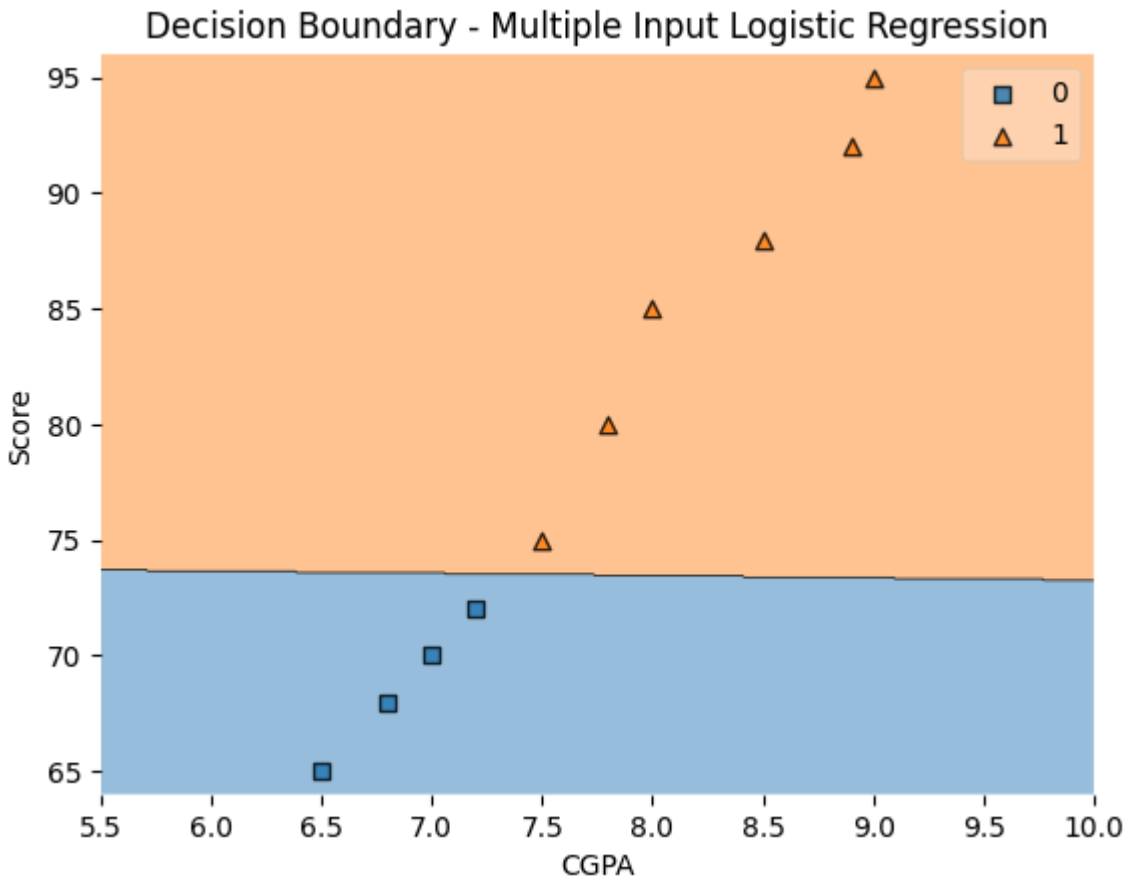
```
# Step 1: Import Libraries
```

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from mlxtend.plotting import plot_decision_regions

# Step 2: Create Dataset
data = pd.DataFrame({
    "CGPA": [6.5, 7.0, 8.0, 7.8, 8.5, 7.2, 6.8, 8.9, 9.0, 7.5],
    "Score": [65, 70, 85, 80, 88, 72, 68, 92, 95, 75],
    "Placed": [0, 0, 1, 1, 1, 0, 0, 1, 1, 1]
})

# Step 3: Features & Target
X = data[["CGPA", "Score"]]
y = data["Placed"]

# Step 4: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 5: Train Model
lr = LogisticRegression()
lr.fit(X_train, y_train)

# Step 6: Evaluation
y_pred = lr.predict(X_test)
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Step 7: Scatter Plot of Data
sns.scatterplot(data=data, x="CGPA", y="Score", hue="Placed", palette="coolwarm")
plt.title("Placement based on CGPA & Score")
plt.show()

# Step 8: Decision Boundary Plot
plot_decision_regions(X.to_numpy(), y.to_numpy(), clf=lr)
```

```
plt.xlabel("CGPA")
plt.ylabel("Score")
plt.title("Decision Boundary - Multiple Input Logistic Regression")
plt.show()
```

```
Confusion Matrix:
 [[1 0]
 [0 1]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1

    accuracy                           1.00         2
   macro avg       1.00      1.00      1.00         2
weighted avg       1.00      1.00      1.00         2
```





---

4. Key Notes

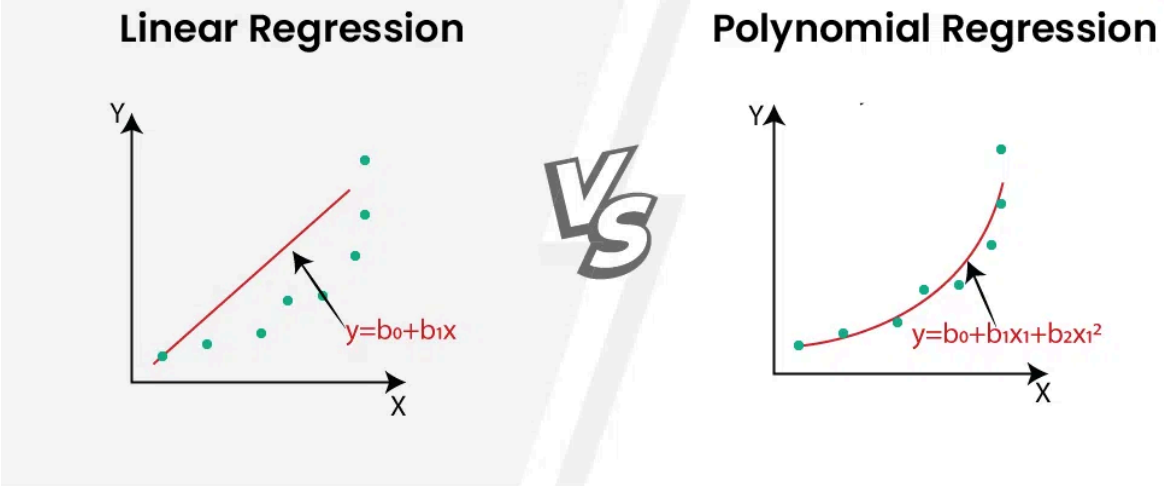
- **Multiple features** improve classification power compared to single-feature logistic regression.
- Decision boundaries can be visualized using plot_decision_regions from mlxtend.
- Works best when the relationship between **log-odds** and inputs is **linear**.

Polynomial Input Logistic Regression – Notes

---

1. What is Polynomial Input Logistic Regression?

- Standard Logistic Regression assumes a **linear relationship** between the independent variables and the log-odds.
- But in real-world datasets, decision boundaries are often **non-linear**.
- **Polynomial Input Logistic Regression** transforms input features into **higher-degree polynomial features** so the model can capture **non-linear decision boundaries**.



---

2. Formula

For **two features** X_1, X_2:

- Linear Logistic Regression:

$$P(Y=1) = \frac{1}{1+e^{-(b_0+b_1X_1+b_2X_2)}}$$

- Polynomial (degree=2):

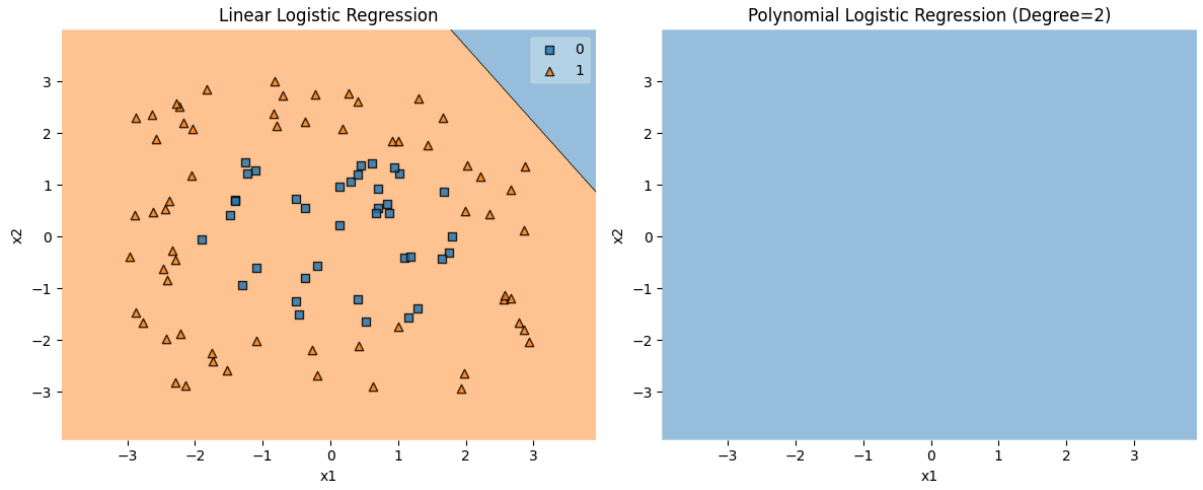
$$P(Y=1) = \frac{1}{1+e^{-(b_0+b_1X_1+b_2X_2+b_3X_1^2+b_4X_2^2+b_5X_1X_2)}}$$

This allows the model to learn **curved boundaries.**

3. Example Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from mlxtend.plotting import plot_decision_regions

# Step 1: Create custom dataset (non-linear separable)
np.random.seed(0)
x1 = np.random.uniform(-3, 3, 100)
x2 = np.random.uniform(-3, 3, 100)
y = (x1**2 + x2**2 > 4).astype(int)  # Circle decision boundary

X = pd.DataFrame({"x1": x1, "x2": x2})

# Step 2: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Logistic Regression (Linear)
lr_linear = LogisticRegression()
lr_linear.fit(X_train, y_train)

# Step 4: Polynomial Feature Transformation
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X)

# Step 5: Logistic Regression (Polynomial)
lr_poly = LogisticRegression(max_iter=1000)
lr_poly.fit(X_poly, y)

# Step 6: Visualization
plt.figure(figsize=(12, 5))

# Linear Logistic Regression
plt.subplot(1, 2, 1)
plot_decision_regions(X.to_numpy(), y, clf=lr_linear)
```

```
plt.title("Linear Logistic Regression")
plt.xlabel("x1")
plt.ylabel("x2")

# Polynomial Logistic Regression
plt.subplot(1, 2, 2)
plot_decision_regions(
    X_poly, y, clf=lr_poly,
    filler_feature_values={2:0, 3:0, 4:0}  # fix extra polynomial terms
)
plt.title("Polynomial Logistic Regression (Degree=2)")
plt.xlabel("x1")
plt.ylabel("x2")

plt.tight_layout()
plt.show()
```



---

4. Comparison: Linear vs Polynomial Logistic Regression

Aspect	Linear Logistic Regression	Polynomial Logistic Regression
Decision Boundary	Straight line / plane	Curved, flexible
Fit to Data	Works well for linearly separable data	Works for **non-linear patterns**

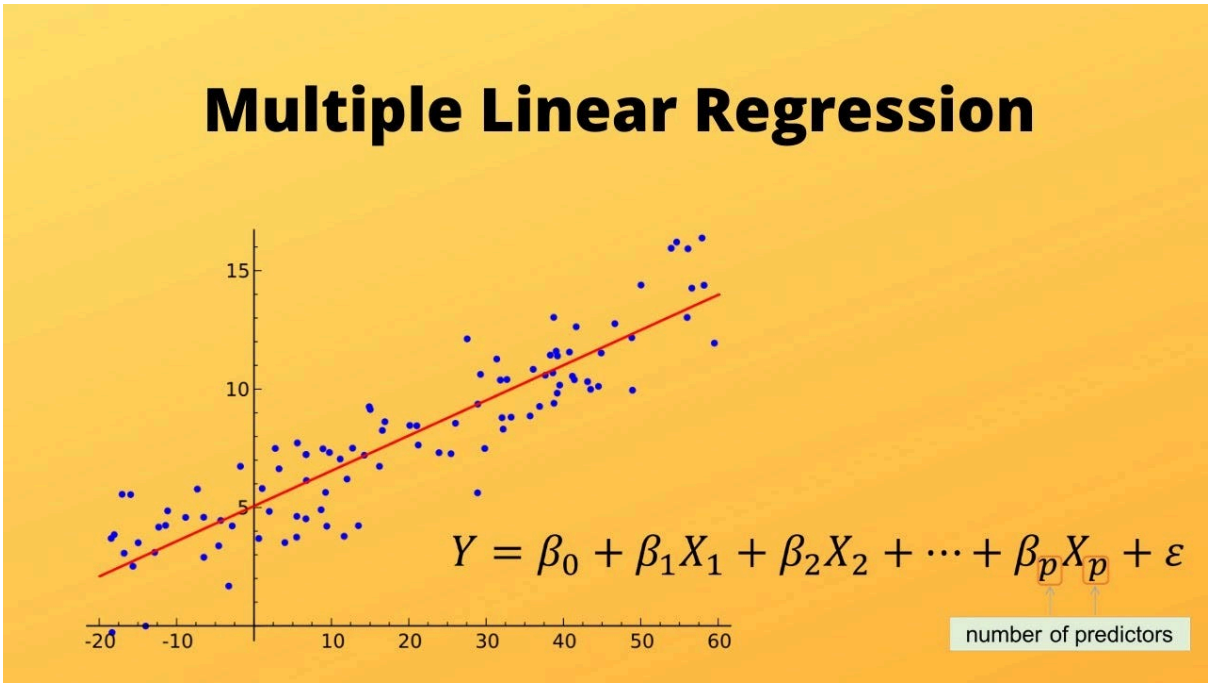
Complexity	Simple, easy to interpret	More complex (higher features)
Overfitting	Less prone	Higher risk if degree is too high
Use Case	Simple binary classification	Circular, spiral, or complex boundaries

---

📘 Notes on Multiclass Classification

🔹 1. What is Multiclass Classification?

- In **binary classification**, we classify data into 2 classes (0/1).
- In **multiclass classification**, we classify data into **3 or more classes** (e.g., predicting digits 0–9, classifying animals: dog, cat, bird, etc.).



Logistic Regression can be extended to multiclass problems using:

1. **OvR (One-vs-Rest / One-vs-All)**
2. **Multinomial Logistic Regression (Softmax Regression)**

🔹 2. One-vs-Rest (OvR) Method

- Idea: Train **one classifier per class**.
- For **k classes**, train **k binary classifiers**.

- Each classifier predicts probability: "Is this sample class i or not?".
- The class with the **highest probability** is chosen.

✅ Default strategy in **sklearn LogisticRegression** (if solver = liblinear).

Function Used:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(multi_class="ovr", solver="liblinear")
clf.fit(X_train, y_train)
```

🔹 3. Multinomial Logistic Regression (Softmax)

- Instead of building multiple binary classifiers, it uses a **single model**.
- Applies **Softmax function** to output probabilities across all classes.
- Probability for class i:

$$P(y = i|x) = \frac{e^{\theta_i \cdot x}}{\sum_{j=1}^{k} e^{\theta_j \cdot x}}$$

The class with the **highest probability** is chosen.

✅ Used when multi_class="multinomial" with solver = lbfgs, newton-cg, or saga.

Function Used:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(multi_class="multinomial", solver="lbfgs")
clf.fit(X_train, y_train)
```

🔹 4. Comparison: OvR vs Multinomial

Feature	OvR (One-vs-Rest)	Multinomial (Softmax)
Model Type	k binary models	One unified model

Training Speed	Faster	Slower
Accuracy	Sometimes lower	Generally better
Default in sklearn	Yes (liblinear)	Yes (lbfgs, saga)

🔹 5. Example Code (Iris Dataset 🌸)

```python
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

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

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

# OvR Logistic Regression
ovr_model = LogisticRegression(multi_class="ovr", solver="liblinear")
ovr_model.fit(X_train, y_train)
print("OvR Accuracy:", accuracy_score(y_test, ovr_model.predict(X_test)))

# Multinomial Logistic Regression
multi_model = LogisticRegression(multi_class="multinomial", solver="lbfgs", max_iter=500)
multi_model.fit(X_train, y_train)
print("Multinomial Accuracy:", accuracy_score(y_test, multi_model.predict(X_test)))
```

✅ **Key takeaway**:

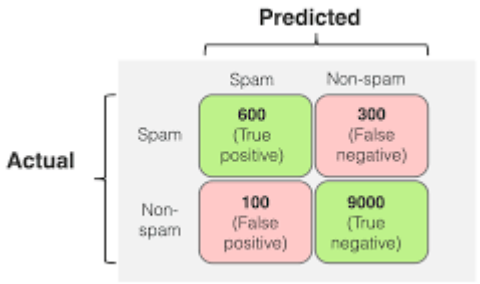
- **OvR** = simple, fast, but less accurate for correlated classes.
- **Multinomial (Softmax)** = slower, but usually better accuracy & probability calibration.

📊 Confusion Matrix – Notes

🔹 What is a Confusion Matrix?

A **Confusion Matrix** is a performance evaluation tool for classification models.
It shows how many predictions were **correct** and **incorrect**, broken down by each class.

It is a **square matrix** comparing **Actual values (True labels)** vs **Predicted values (Model output)**.



🟩 Structure of Confusion Matrix (Binary Classification)

	Predicted Positive (1)	Predicted Negative (0)
Actual Positive (1)	**True Positive (TP)**	**False Negative (FN)**
Actual Negative (0)	**False Positive (FP)**	**True Negative (TN)**

🔹 Terminologies

1. **True Positive (TP)** → Model predicted **Positive**, and it was actually **Positive**.
2. **True Negative (TN)** → Model predicted **Negative**, and it was actually **Negative**.
3. **False Positive (FP)** → Model predicted **Positive**, but it was actually **Negative**. (Type I Error)
4. **False Negative (FN)** → Model predicted **Negative**, but it was actually **Positive**. (Type II Error)

🔹 Important Metrics from Confusion Matrix

- **Accuracy** → Overall correctness of the model.

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

- **Precision** → Out of predicted positives, how many are correct.

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

- **Recall (Sensitivity / TPR)** → Out of actual positives, how many are correctly predicted.

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

- **F1-Score** → Balance between Precision & Recall.

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

- **Specificity (TNR)** → Out of actual negatives, how many are correctly predicted.

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

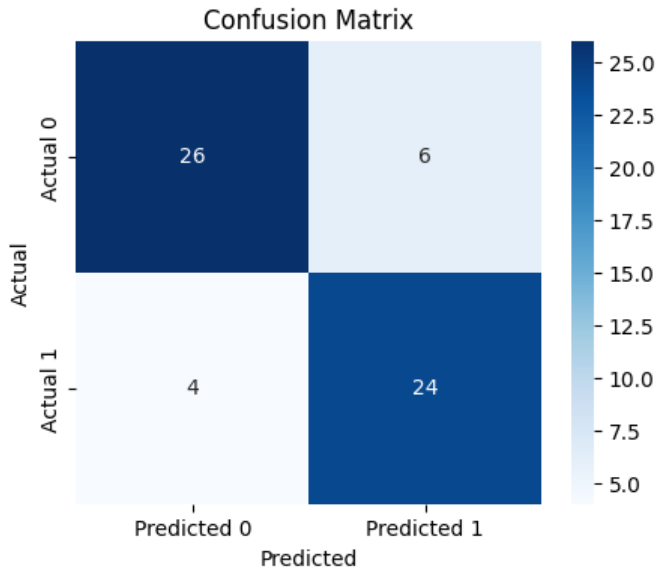
🔹 Why is Confusion Matrix Important?

✔ Helps to understand **where the model is going wrong**
✔ Shows **type of errors** (FP vs FN)
✔ Better than Accuracy for **imbalanced datasets**
✔ Used in **Medical AI, Fraud Detection, Spam Filters** etc.

🔹 Python Example

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

# Step 1: Create dataset (fixed params)
X, y = make_classification(n_samples=200,
                           n_features=2,
                           n_informative=2,   # all features are useful
                           n_redundant=0,     # no redundant features
                           n_classes=2,
                           random_state=42)

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

# Step 3: Train model
model = LogisticRegression()
model.fit(X_train, y_train)

# Step 4: Predictions
y_pred = model.predict(X_test)

# Step 5: Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

# Step 6: Plot Confusion Matrix
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Predicted 0","Predicted 1"],
            yticklabels=["Actual 0","Actual 1"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
```

```
plt.show()

# Step 7: Classification Report
print(classification_report(y_test, y_pred))
```



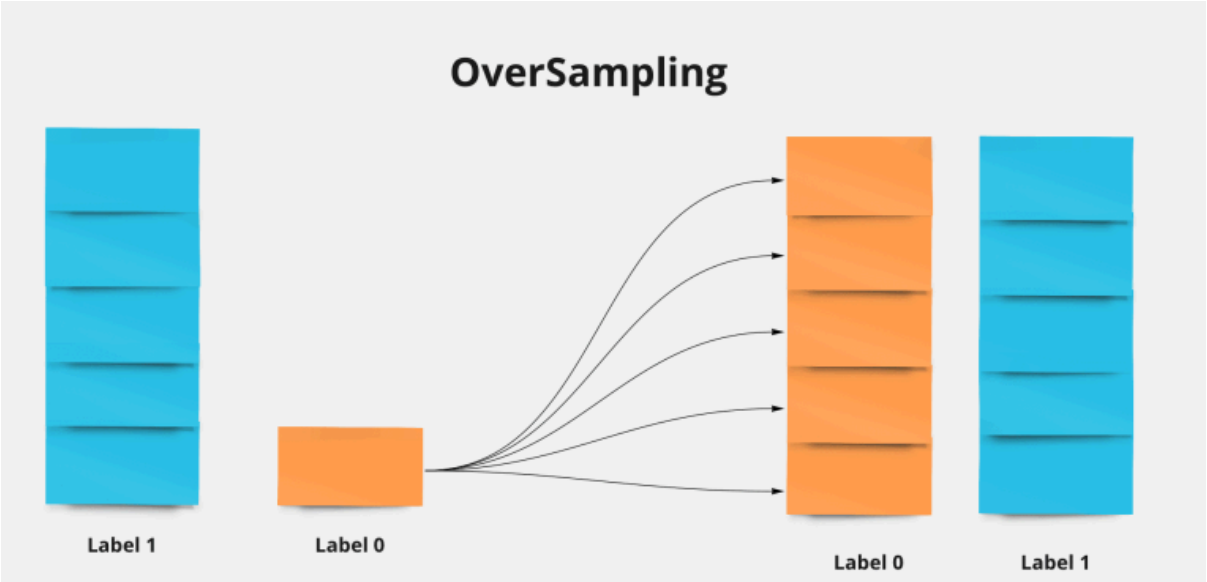
	precision	recall	f1-score	support
0	0.87	0.81	0.84	32
1	0.80	0.86	0.83	28
accuracy			0.83	60
macro avg	0.83	0.83	0.83	60
weighted avg	0.84	0.83	0.83	60

👉 This gives you a **Confusion Matrix heatmap + Precision, Recall, F1-score report**.

📘 Imbalanced Datasets & Handling Techniques

🔹 What is Imbalanced Dataset?

- An **imbalanced dataset** occurs when the number of observations in each class is **not equally distributed**.
- Example: Fraud detection →
    - Fraud = 1%
    - Non-Fraud = 99%
- Issue: ML models become biased towards the majority class.



⚖️ Why is it a Problem?

- Accuracy becomes misleading.
- Model may predict only majority class.
- Rare but important cases (fraud, disease detection) may be ignored.

🔹 Solutions

1. Random Oversampling

- Increases minority class samples by **duplicating them randomly** until classes are balanced.
- Advantage: No loss of data.
- Disadvantage: Overfitting (same minority samples repeated).

```
from imblearn.over_sampling import RandomOverSampler
from collections import Counter

X_res, y_res = RandomOverSampler(random_state=42).fit_resample(X, y)
print("Before:", Counter(y))
print("After Oversampling:", Counter(y_res))
```

---

2. Random Undersampling

- Reduces majority class samples by **randomly removing them** to match minority class.
- Advantage: Faster training, less memory.
- Disadvantage: Loss of important majority class data.

```
from imblearn.under_sampling import RandomUnderSampler

X_res, y_res = RandomUnderSampler(random_state=42).fit_resample(X, y)
print("Before:", Counter(y))
print("After Undersampling:", Counter(y_res))
```

🔹 Comparison

Method	Pros ✅	Cons ❌
Oversampling	Keeps all data, balances dataset	Overfitting risk
Undersampling	Faster training, less memory usage	Loss of information from majority

---

✅ **Tip**: Instead of just over/undersampling, we can use **SMOTE (Synthetic Minority Oversampling Technique)** to generate **synthetic samples**.

📘 Naïve Bayes Classifier

🔹 1. Introduction

- **Naïve Bayes** is a **probabilistic classifier** based on **Bayes' Theorem**.
- Assumption: Features are **independent** (this is the "naïve" part).
- Works well for **text classification, spam detection, sentiment analysis**, etc.

🔹 2. Bayes' Theorem

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

Where:

- P(Y∣X)P(Y|X)P(Y∣X) → Posterior probability (class given features)
- P(X∣Y)P(X|Y)P(X∣Y) → Likelihood (features given class)
- P(Y)P(Y)P(Y) → Prior probability of class

- P(X)P(X)P(X) → Evidence (scaling factor)

🔹 3. Why "Naïve"?

- It assumes **independence between features**:

$$P(X|Y) = P(x_1|Y) \cdot P(x_2|Y) \cdot \ldots \cdot P(x_n|Y)$$

This makes computation **fast and simple**, but not always realistic.

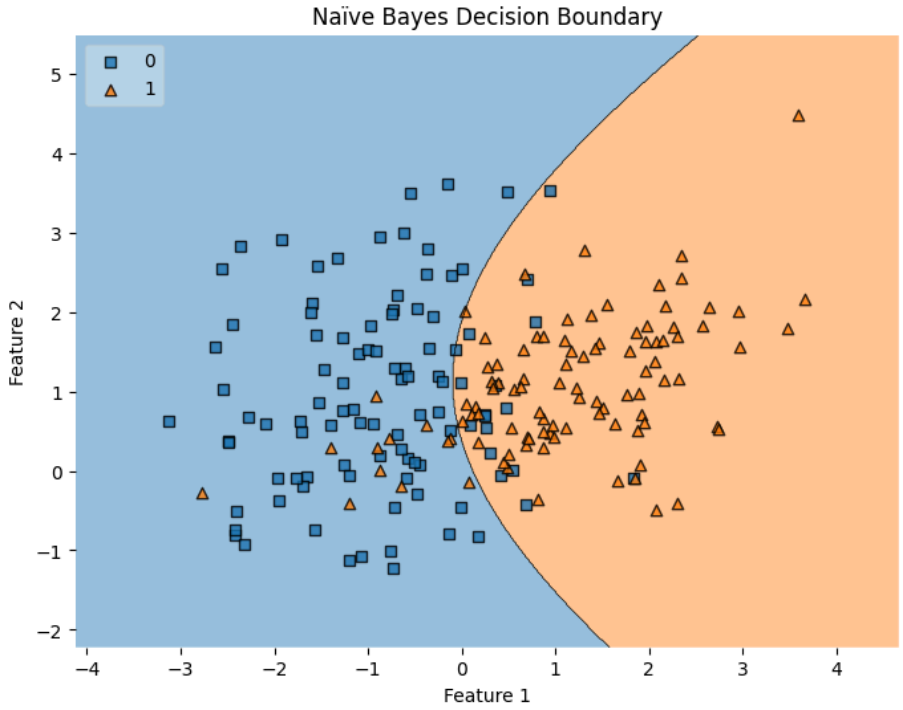
🔹 4. Types of Naïve Bayes

1. **Gaussian Naïve Bayes** → Continuous features (assumes normal distribution).
2. **Multinomial Naïve Bayes** → Text classification (word counts).
3. **Bernoulli Naïve Bayes** → Binary features (0/1 like word present or not).

🔹 5. Python Example with Graph

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from mlxtend.plotting import plot_decision_regions

# Step 1: Create Dataset
X, y = make_classification(n_samples=200, n_features=2,
                           n_classes=2, n_redundant=0,
                           n_clusters_per_class=1, random_state=42)

# Step 2: Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Step 3: Train Naïve Bayes
model = GaussianNB()
model.fit(X_train, y_train)

# Step 4: Plot decision boundary
plt.figure(figsize=(8,6))
plot_decision_regions(X, y, clf=model, legend=2)
plt.title("Naïve Bayes Decision Boundary")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```



🔹 6. Advantages

✅ Very fast & simple
✅ Works well with small datasets
✅ Handles high-dimensional text data (spam, sentiment)
✅ Requires little training data

🔹 7. Disadvantages

❌ Assumes features are independent (not always true)
❌ Performs poorly when features are highly correlated
❌ Not great with continuous data unless distribution is normal

---

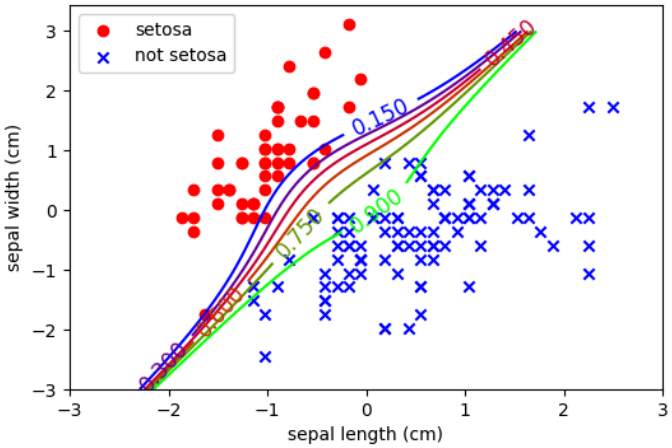
🔹 8. Applications

- Spam email filtering
- Sentiment analysis
- Document categorization
- Medical diagnosis

📊 The graph above (decision boundary) helps visualize how Naïve Bayes classifies regions in feature space.

🔹 What is a Non-Linear Model?

- **Linear Model**: Assumes a **straight-line relationship** between input features and output. Example: Linear Regression, Logistic Regression.
- **Non-Linear Model**: Captures **complex, curved, non-linear relationships** in data.
    - Handles cases where data **cannot be separated by a straight line/plane**.
    - Examples: **Decision Tree, Random Forest, SVM (with kernels), Neural Networks, kNN**.



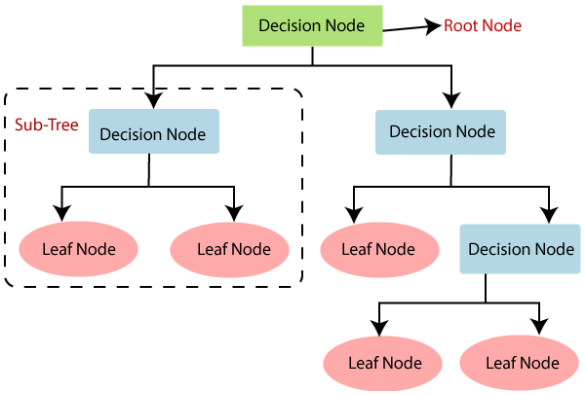
👉 In short:

- Linear = "Straight line fitting"
- Non-linear = "Flexible curves & splits that capture complexity"

🔹 What is a Decision Tree?

- A **non-linear supervised learning algorithm**.
- Works for **Classification & Regression** tasks.
- Splits data into branches based on conditions (like if/else rules).

- Final output → Leaf node (class or value).



Example:

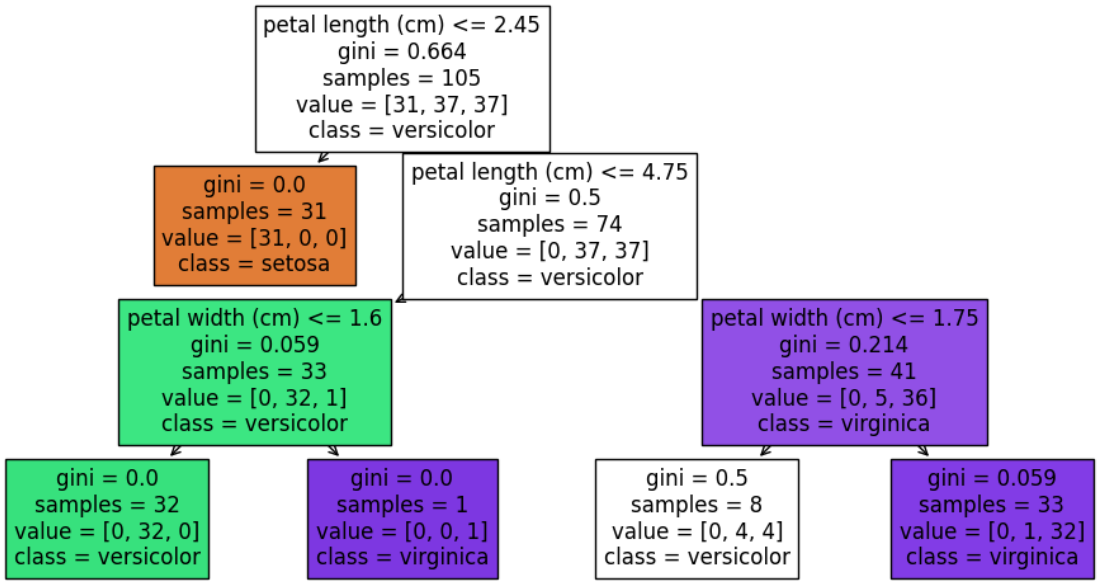
- If Age > 30 → Yes
- Else If Salary > 50k → Yes
- Else → No

🔷 Decision Tree Code Example (Classification)

```
# Step 1: Import libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score

# Step 2: Load dataset (Iris dataset)
iris = load_iris()
X = iris.data
y = iris.target

# Step 3: Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```python
# Step 4: Train Decision Tree
clf = DecisionTreeClassifier(criterion="gini", max_depth=3, random_state=42)
clf.fit(X_train, y_train)

# Step 5: Predictions
y_pred = clf.predict(X_test)

# Step 6: Evaluate
print("Accuracy:", accuracy_score(y_test, y_pred))

# Step 7: Plot Decision Tree
plt.figure(figsize=(12,6))
plot_tree(clf, feature_names=iris.feature_names, class_names=iris.target_names, filled=True)
plt.show()
```

Accuracy: 100.0



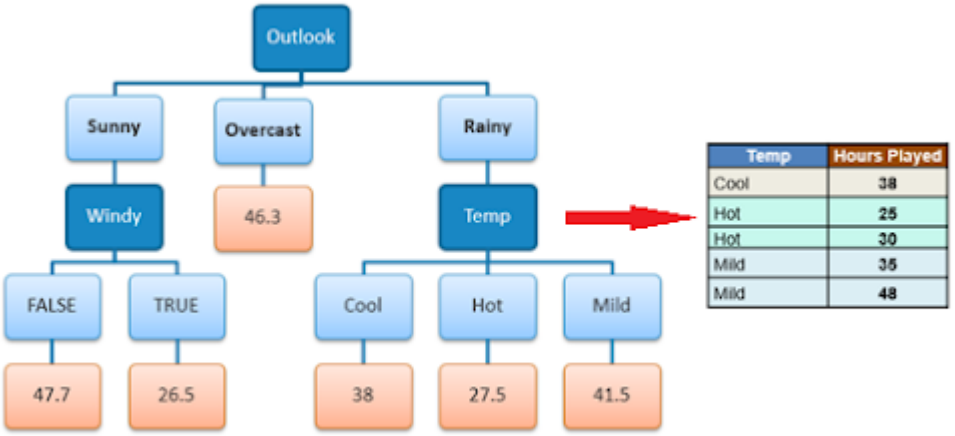
🔹 Output:

✅ Prints **Accuracy**
🌳 Shows a **Decision Tree Graph** (splits with conditions)

🌳 Decision Tree Regression – Notes

🔹 What is it?

- A **non-linear supervised ML algorithm** used for regression tasks.
- Splits the dataset into regions by learning **decision rules** (if-else conditions).
- Each region predicts a **constant value** (average of target values in that region).



🔹 How it Works

1. Start with all data in the root node.
2. Choose the best **split point** (feature & threshold) that minimizes error.
    - Error often measured using **MSE (Mean Squared Error)**.
3. Split into left and right child nodes.
4. Repeat until stopping condition (max depth, min samples, etc.).
5. Final prediction = **mean of target values** in the leaf node.

🔹 Advantages

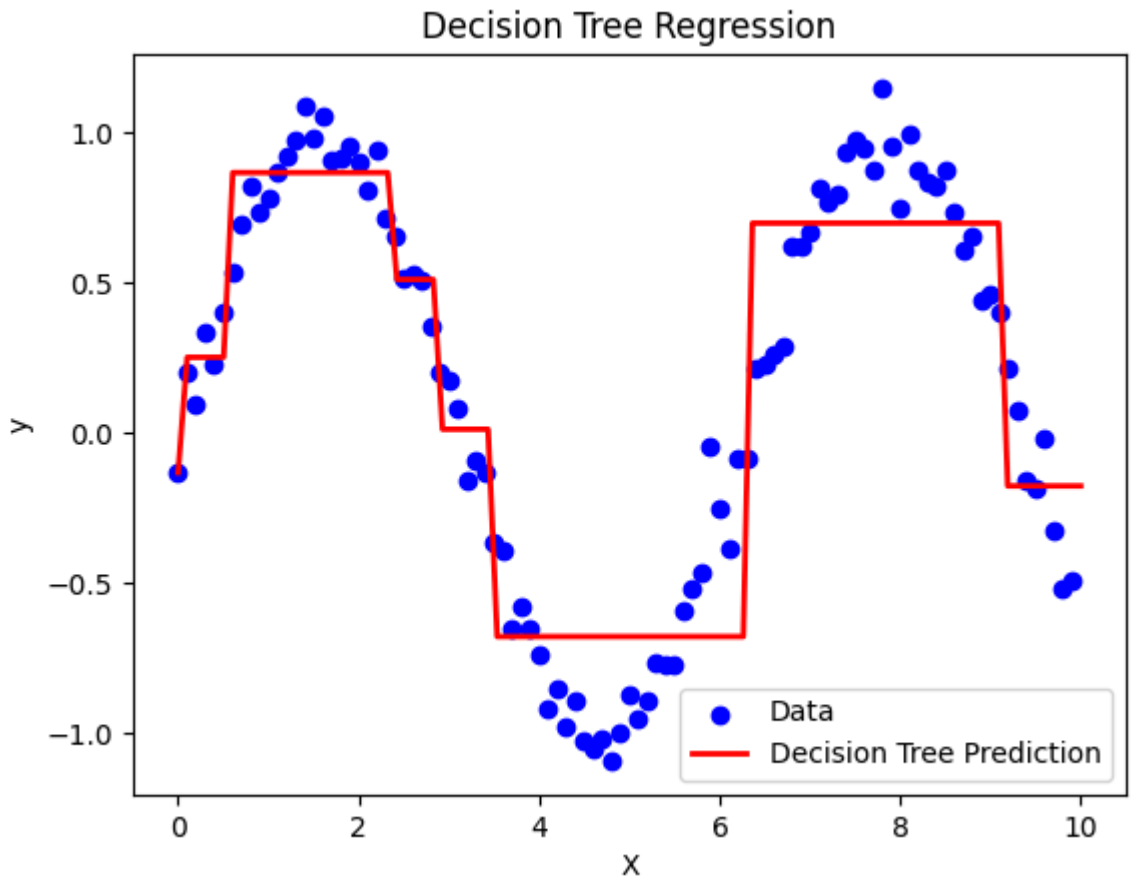
✅ Handles **non-linear** and complex relationships.
✅ Easy to visualize and interpret.
✅ No need for feature scaling.
✅ Can automatically capture feature interactions.

🔹 Limitations

❌ Prone to **overfitting** (high variance).
❌ Not smooth prediction (piecewise constant).
❌ Sensitive to small data changes.
👉 Solution → Use **Random Forest / Gradient Boosting** for better generalization.

🔹 Python Example

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

# Dataset
X = np.arange(0, 10, 0.1).reshape(-1, 1)
y = np.sin(X).ravel() + np.random.randn(len(X)) * 0.1  # noisy sine wave

# Train Decision Tree Regressor
regressor = DecisionTreeRegressor(max_depth=3, random_state=42)
regressor.fit(X, y)

# Predictions
X_test = np.linspace(0, 10, 100).reshape(-1, 1)
y_pred = regressor.predict(X_test)

# Plot
plt.scatter(X, y, color="blue", label="Data")
plt.plot(X_test, y_pred, color="red", linewidth=2, label="Decision Tree Prediction")
plt.xlabel("X")
plt.ylabel("y")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```



🔹 Graphical Intuition

- The **red line** shows a step-like function.
- Predictions are **piecewise constant**, not smooth (like linear regression).

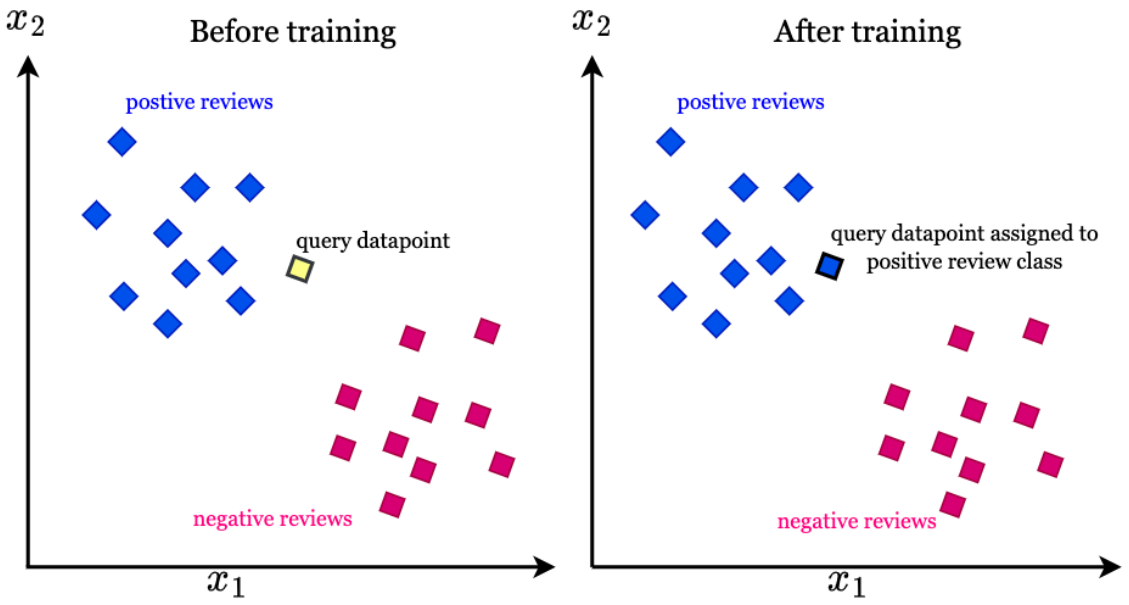
🔹 K-Nearest Neighbors (KNN)

📌 What is KNN?

- A **non-parametric, supervised learning algorithm** used for both **classification** and **regression**.
- Works on the principle:

  "A data point is classified/predicted based on the majority or average of its nearest neighbors."

- Uses a **distance metric** (commonly Euclidean distance) to find the *k* closest points.



🟢 KNN Classification

✔️ How it works:

1. Choose the number of neighbors k.
2. For a new data point, find the k nearest points from the training dataset.

3. Assign the **majority class label** among neighbors.

Example:

If k=5 and among neighbors → 3 are "Dog", 2 are "Cat", the prediction = **Dog** 🐶

🔵 KNN Regression

✔️ How it works:

1. Choose k.
2. Find the k nearest neighbors.
3. Take the **average of their target values** as prediction.

Example:
If neighbors have prices [200, 220, 240], prediction = (200+220+240)/3 = 220.

🔹 Advantages

✅ Simple and intuitive.
✅ Works for both classification & regression.
✅ No training step (lazy learner).
✅ Can capture non-linear decision boundaries.

🔹 Limitations

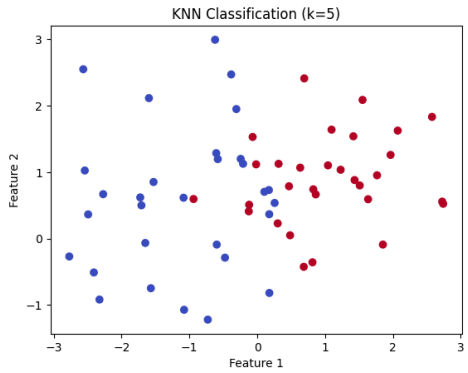
❌ Computationally expensive for large datasets (needs to compute distance for all points).
❌ Sensitive to irrelevant features → needs **feature scaling (Normalization/Standardization)**.
❌ Choosing optimal k is tricky.

🐍 Python Code – Classification & Regression

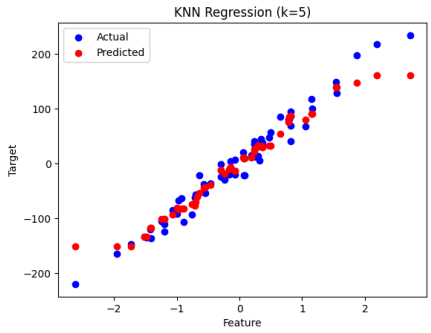
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification, make_regression
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.metrics import accuracy_score, mean_squared_error

# ======================
# 1. KNN Classification
# ======================
# Create synthetic dataset (fixed!)
Xc, yc = make_classification(
    n_samples=200, n_features=2, n_informative=2,
    n_redundant=0, n_classes=2,
    n_clusters_per_class=1, random_state=42
)

# Split data
Xc_train, Xc_test, yc_train, yc_test = train_test_split(
    Xc, yc, test_size=0.3, random_state=42
)

# Train KNN Classifier
knn_clf = KNeighborsClassifier(n_neighbors=5)
knn_clf.fit(Xc_train, yc_train)
yc_pred = knn_clf.predict(Xc_test)

print("Classification Accuracy:", accuracy_score(yc_test, yc_pred))

# Plot classification
plt.scatter(Xc_test[:, 0], Xc_test[:, 1], c=yc_pred, cmap="coolwarm", marker="o")
plt.title("KNN Classification (k=5)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()


# ======================
```

```
# 2. KNN Regression
# ======================
# Create synthetic regression dataset
Xr, yr = make_regression(
    n_samples=200, n_features=1, noise=15, random_state=42
)

# Split data
Xr_train, Xr_test, yr_train, yr_test = train_test_split(
    Xr, yr, test_size=0.3, random_state=42
)

# Train KNN Regressor
knn_reg = KNeighborsRegressor(n_neighbors=5)
knn_reg.fit(Xr_train, yr_train)
yr_pred = knn_reg.predict(Xr_test)

print("Regression MSE:", mean_squared_error(yr_test, yr_pred))

# Plot regression results
plt.scatter(Xr_test, yr_test, color="blue", label="Actual")
plt.scatter(Xr_test, yr_pred, color="red", label="Predicted")
plt.title("KNN Regression (k=5)")
plt.xlabel("Feature")
plt.ylabel("Target")
plt.legend()
plt.show()
```

Accuracy: 0.8



Regression MSE: 522.3194078689484



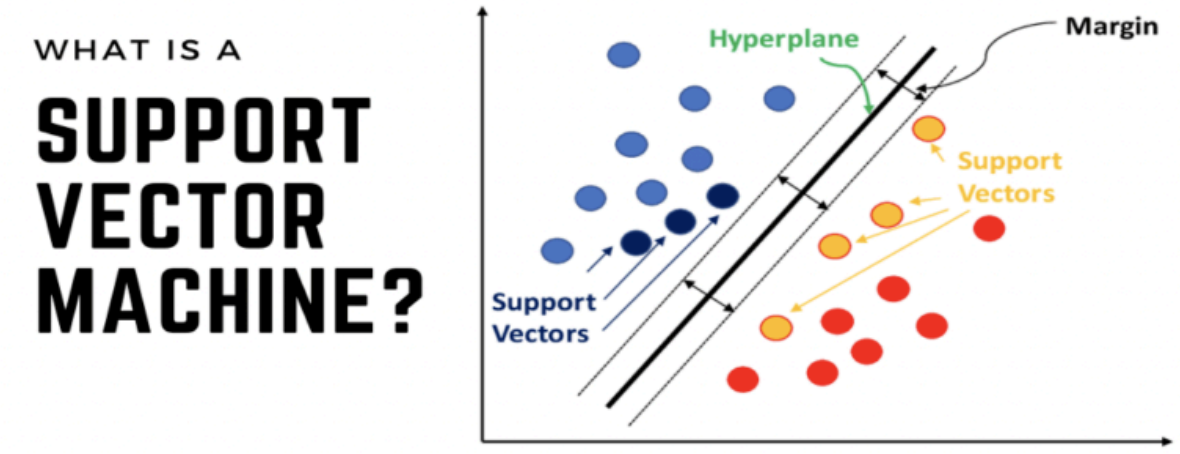
📊 Graphical Intuition

- **Classification** → Decision boundaries become **non-linear** (KNN adapts well to complex data).
- **Regression** → Predictions follow **local averages**, giving a smoother curve than Decision Tree.

📘 Support Vector Machine (SVM) – Notes

🔹 1. Introduction

- **Support Vector Machine (SVM)** is a supervised ML algorithm used for both **classification** and **regression** tasks.
- SVM finds the **best hyperplane** that separates data points of different classes with the **maximum margin**.
- The data points that lie closest to the hyperplane are called **support vectors**.



🔹 2. SVM for Classification

- Goal: Separate two (or more) classes with the widest possible margin.
- Decision Boundary:
    - For linear classification, SVM finds a line (2D), plane (3D), or hyperplane (higher dimensions).
- Formula for decision function:

$$f(x) = w^T x + b$$

Where:

- www → weight vector (defines orientation of hyperplane)
- bbb → bias term
- **Hinge Loss Function** (used in SVM classification):

$$L = \max(0, 1 - y_i(w^T x_i + b))$$

✅ Pros: Works well in **high-dimensional space**.
❌ Cons: Not good when dataset is very large.

🔹 3. SVM for Regression (Support Vector Regression – SVR)

- Instead of finding a hyperplane that separates classes, **SVR fits the data within a margin (ε-tube)**.
- Goal: Predict values within a tolerance (ε).
- Only points outside the margin contribute to the cost function.

SVR tries to solve:

$$\min \frac{1}{2}||w||^2 \quad \text{s.t. } |y_i - (w^T x_i + b)| \leq \epsilon$$

✅ Pros: Works well for **non-linear regression** when kernel trick is applied.

🔹 4. Kernel Trick

- Many datasets are **not linearly separable**.
- SVM uses the **Kernel Trick** to project data into higher dimensions where it becomes separable.

Common Kernels:

1. **Linear Kernel:** $K(x_i, x_j) = x_i^T x_j$
2. **Polynomial Kernel:** $K(x_i, x_j) = (x_i^T x_j + c)^d$
   - Degree d controls complexity (higher → more flexible).
3. **RBF (Radial Basis Function / Gaussian) Kernel:**

$$K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$$

🔹 5. Visual Intuition

- **Linear SVM** → Straight line separation.
- **Polynomial SVM** → Curved decision boundaries, suitable for circular/spiral patterns.
- **RBF SVM** → Flexible, can form complex decision boundaries.
- **SVR** → Fits curve within ε-margin.

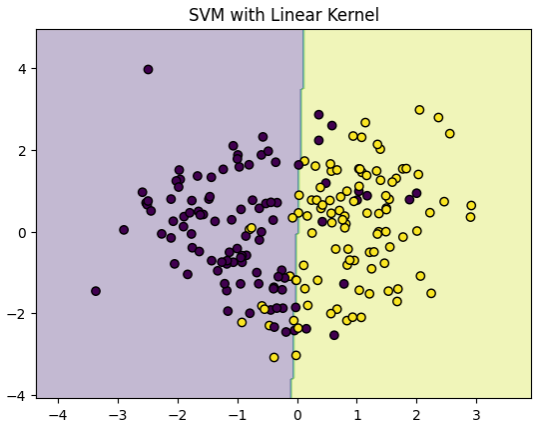
🔹 6. Python Examples

📌 Classification with SVM

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
import matplotlib.pyplot as plt

# Load dataset (fix: set n_informative=2)
```

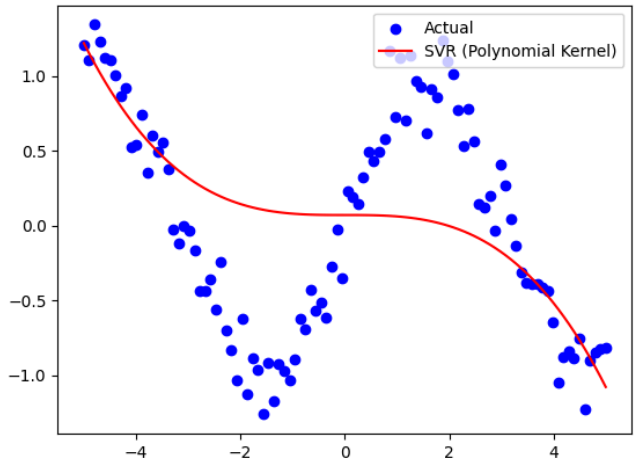
```
X, y = datasets.make_classification(
    n_samples=200,
    n_features=2,
    n_classes=2,
    n_informative=2,
    n_redundant=0,
    random_state=42
)

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

# Train SVM with linear kernel
clf = SVC(kernel='linear', C=1.0)
clf.fit(X_train, y_train)

print("Accuracy:", clf.score(X_test, y_test))

# --- Optional: visualize decision boundary ---
import numpy as np

# Create mesh grid
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                     np.linspace(y_min, y_max, 200))

Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')
plt.title("SVM with Linear Kernel")
plt.show()
```

Accuracy: 0.8166666666666667



📌 Regression with SVR

```
import numpy as np
from sklearn.svm import SVR

# Synthetic dataset
X = np.linspace(-5, 5, 100).reshape(-1, 1)
y = np.sin(X).ravel() + np.random.normal(0, 0.2, X.shape[0])

# Train SVR with polynomial kernel
svr_poly = SVR(kernel='poly', degree=3, C=100, epsilon=0.1)
svr_poly.fit(X, y)

# Predictions
y_pred = svr_poly.predict(X)

plt.scatter(X, y, color='blue', label="Actual")
plt.plot(X, y_pred, color='red', label="SVR (Polynomial Kernel)")
plt.legend()
plt.show()
```



🔹 7. Summary

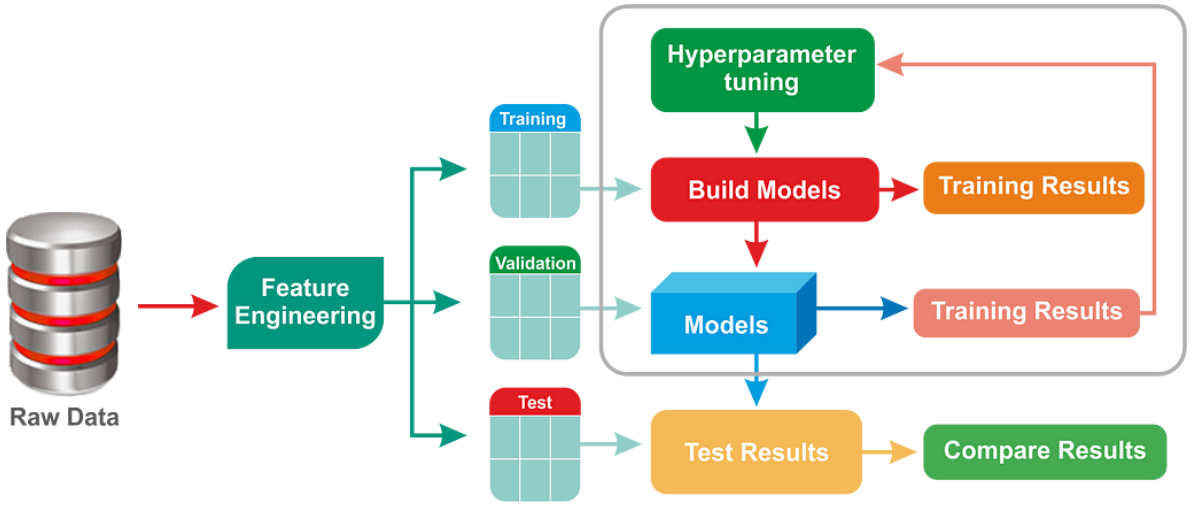
- **SVM Classification** → Finds best hyperplane with max margin.
- **SVM Regression (SVR)** → Fits within ε-tube, robust to outliers.
- **Kernels** → Enable non-linear classification/regression.
- **Polynomial SVM** → Captures curved, complex patterns.

⚡ **Key Takeaway:**
SVM is one of the most powerful algorithms for both **classification & regression**, especially when data is high-dimensional and non-linear.

📘 Hyperparameter Tuning (GridSearchCV & RandomizedSearchCV)

🔹 1. What are Hyperparameters?

- Hyperparameters are the parameters **set before training** a model.
- They are not learned from data but control how the algorithm works.
- Examples:
  - **KNN:** n_neighbors
  - **Decision Tree:** max_depth, min_samples_split
  - **SVM:** C, kernel, gamma
  - **Random Forest:** n_estimators, max_features



🔹 2. Why Tune Hyperparameters?

- Different hyperparameters → different performance.
- Goal = find best combination that gives **highest accuracy (classification)** or **lowest error (regression)**.

🔹 3. GridSearchCV

- **Exhaustive Search**: tries all possible combinations.
- Uses **cross-validation** to evaluate.
- Best when dataset is **small/medium**.

✨ Example: SVM

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.datasets import load_iris

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

# Define model
model = SVC()

# Define parameter grid
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': ['scale', 'auto']
}

# Apply GridSearchCV
grid = GridSearchCV(model, param_grid, cv=5, scoring='accuracy')
grid.fit(X, y)

print("Best Parameters:", grid.best_params_)
print("Best Score:", grid.best_score_)
```

✅ Tests **all combinations** of C, kernel, gamma.

🔹 4. RandomizedSearchCV

- **Random Search**: picks random combinations instead of all.
- Faster for **large parameter space**.
- You can control number of iterations (n_iter).

✨ Example: Random Forest

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
import numpy as np

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

# Define model
rf = RandomForestClassifier()

# Define parameter distribution
param_dist = {
    'n_estimators': np.arange(50, 300, 50),
    'max_depth': [None, 5, 10, 20],
    'max_features': ['sqrt', 'log2']
}

# Apply RandomizedSearchCV
random_search = RandomizedSearchCV(rf, param_dist, n_iter=10, cv=5, scoring='accuracy',
random_state=42)
random_search.fit(X, y)

print("Best Parameters:", random_search.best_params_)
print("Best Score:", random_search.best_score_)
```

✅ Tries **10 random combinations** instead of all.

🔹 5. Difference Between GridSearchCV vs RandomizedSearchCV

Feature	GridSearchCV 🟢	RandomizedSearchCV 🔵
Search Method	Exhaustive (all combos)	Random subset of combos
Time	Slower (large params)	Faster
Best for	Small parameter space	Large parameter space
Example Use Case	SVM tuning	Random Forest tuning

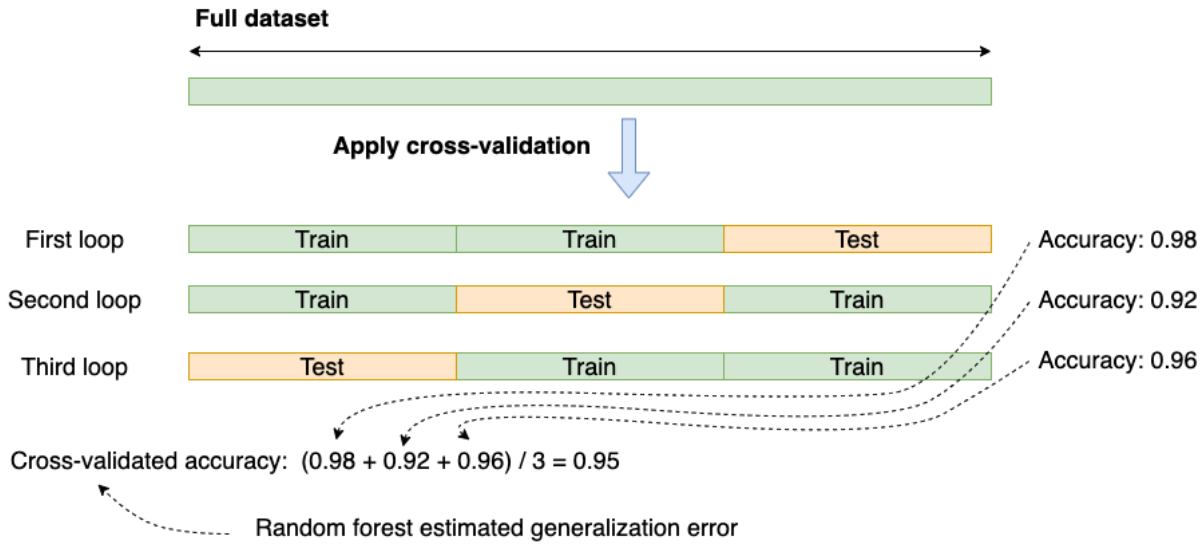
🔹 6. Tips

- Start with **RandomizedSearchCV** (fast).
- If time allows, refine with **GridSearchCV** around best values.
- Always use **cross-validation (cv)** to avoid overfitting.
- Check scoring (accuracy, f1, r2, etc. depending on task).

📘 Cross-Validation (CV) in Machine Learning

🔹 1. What is Cross-Validation?

- A technique to **evaluate model performance** more reliably.
- Instead of training/testing once, dataset is split into multiple **folds** and the model is trained & tested multiple times.
- Helps to avoid **overfitting** and ensures model is **generalizable**.



🔹 2. Why use Cross-Validation?

✅ Uses all data for both training & testing (in different rounds).
✅ Reduces variance in performance score.
✅ More reliable than a single **train-test split**.

🔹 3. Types of Cross-Validation

📍 (a) K-Fold Cross Validation

- Dataset is split into **K equal parts (folds)**.
- Model is trained on (K-1) folds and tested on 1 fold.
- Repeat K times, each fold as test once.
- Final score = **average performance**.

👉 Example (K=5):

Fold1: Train [2,3,4,5] → Test [1]
Fold2: Train [1,3,4,5] → Test [2]
Fold3: Train [1,2,4,5] → Test [3]
Fold4: Train [1,2,3,5] → Test [4]
Fold5: Train [1,2,3,4] → Test [5]

📍 (b) Stratified K-Fold

- Same as K-Fold but ensures **class distribution is balanced** in each fold.
- Useful for **classification problems** (especially imbalanced datasets).

📍 (c) Leave-One-Out CV (LOOCV)

- Special case of K-Fold where **K = N (no. of samples)**.
- Each test set has only 1 observation.
- Very accurate but computationally expensive.

📍 (d) Shuffle Split CV

- Randomly shuffles and splits data into train/test multiple times.
- Flexible: can choose % of train/test size.

🔹 4. Code Example – K-Fold CV

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LogisticRegression

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

# Model
model = LogisticRegression(max_iter=200)

# Define K-Fold
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Apply Cross-Validation
scores = cross_val_score(model, X, y, cv=kf, scoring='accuracy')

print("Scores for each fold:", scores)
print("Mean Accuracy:", scores.mean())
```

🔹 5. When to Use What?

- **K-Fold** → Default choice for general CV.
- **Stratified K-Fold** → For **classification** (imbalanced datasets).
- **LOOCV** → When dataset is **very small**.
- **ShuffleSplit** → When flexibility is needed in train/test sizes.

🔹 6. Key Points

- Typical k=5 or k=10.

- More folds = more reliable but slower.
- CV helps in **model selection + hyperparameter tuning** (used inside GridSearchCV/RandomizedSearchCV).