Machine Learning

🧠 What is Machine Learning (ML)?

Definition:
Machine Learning is a branch of **Artificial Intelligence (AI)** that enables computers to **learn from data** and make decisions or predictions **without being explicitly programmed**.

🔁 Simple Definition:

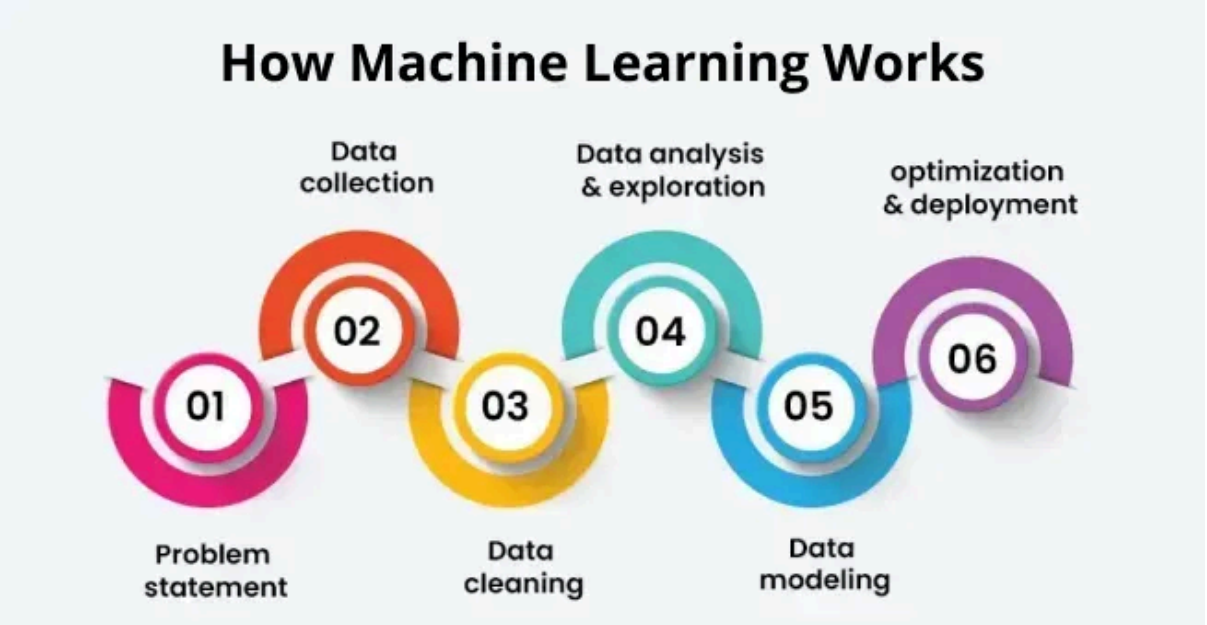
> **"Machine Learning is teaching computers to learn from examples, not rules."**

Machine Learning

🛠️ How ML Works (Step-by-step)

1. **Data Collection**
    - Gather relevant data (CSV files, databases, sensors, etc.)
2. **Data Preprocessing**
    - Clean the data: handle missing values, normalize, encode, etc.
3. **Split the Data**
    - Train data (e.g., 80%) to learn
    - Test data (e.g., 20%) to evaluate

4. **Choose a Model**
    - e.g., Linear Regression, Decision Tree, SVM, etc.
5. **Train the Model**
    - Feed data into the model so it learns patterns
6. **Evaluate the Model**
    - Test accuracy, precision, recall, etc.
7. **Predict New Data**
    - Use the trained model to predict on new, unseen data



🧮 Example: Predict House Price

If you have features like area, number of rooms, and location, ML can learn from old data and predict prices of new houses.

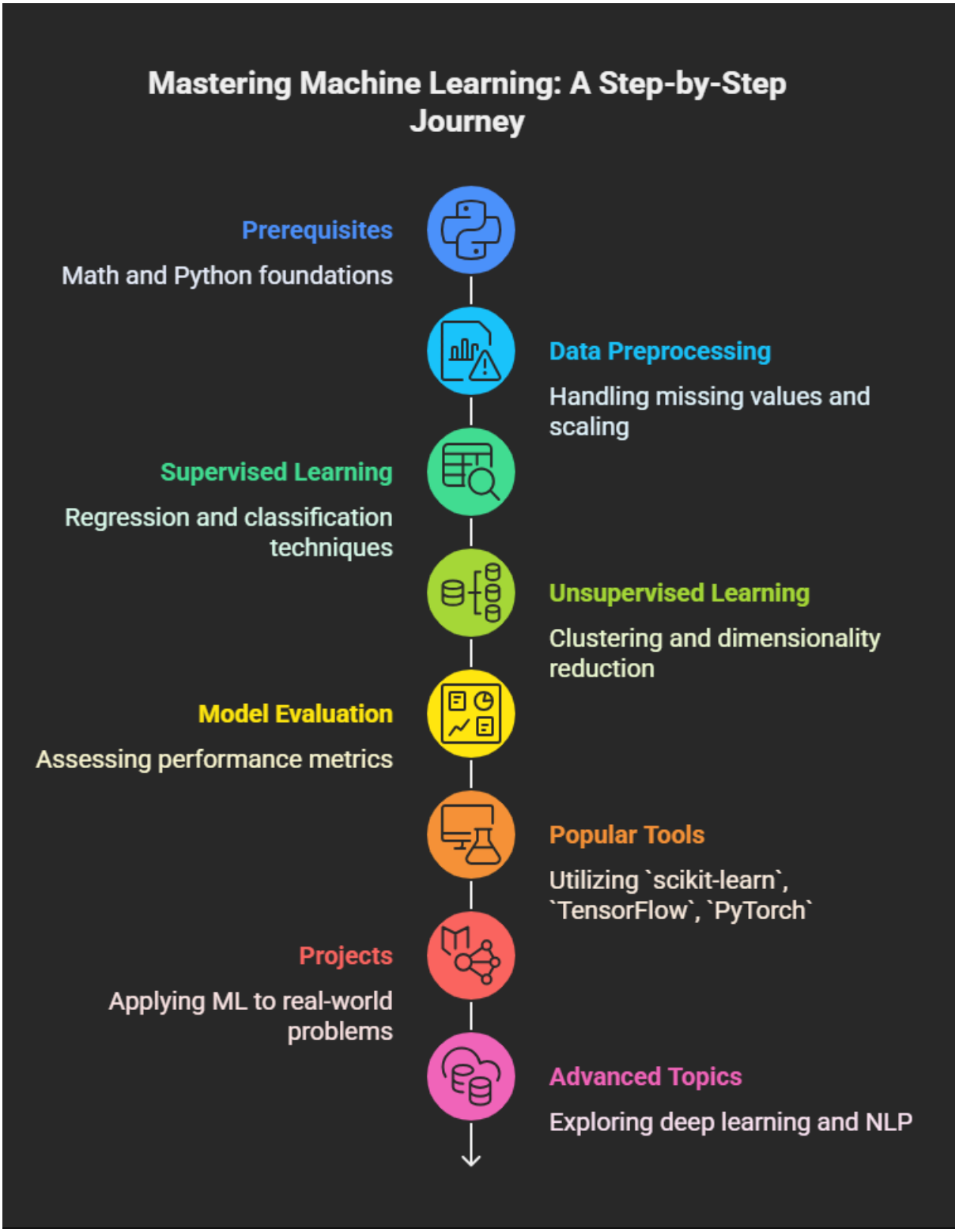
🔁 Traditional Programming vs Machine Learning

Aspect	Traditional Programming	Machine Learning
You Provide	Rules + Data	Data + Output (examples)
System Produces	Output	**Rules/Model** (learned by system)
Best For	Clear logic-based problems	Pattern recognition, prediction
Examples	Calculator, ATM software	Spam detection, ChatGPT, Netflix recommendations

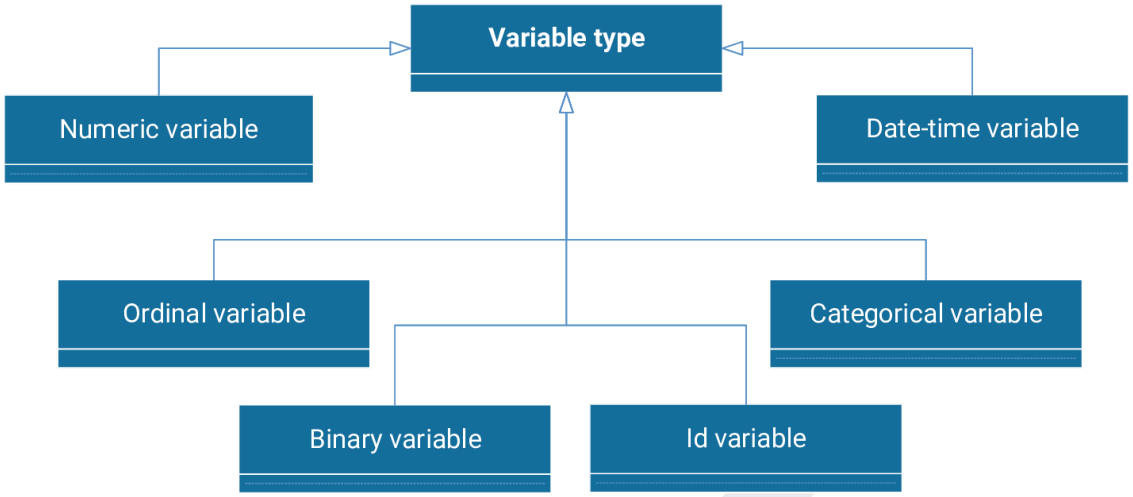
---

🚀 Use of Machine Learning in Real Life

- 📧 **Spam detection** (Gmail filters)
- 📈 **Stock prediction**
- 📷 **Image recognition** (Face unlock, OCR)
- 🛒 **Recommendation systems** (Amazon, Netflix)
- 🤖 **Chatbots & voice assistants** (Siri, Alexa)
- 🚗 **Self-driving cars**
- 🧬 **Healthcare** (Disease prediction)


Mastering Machine Learning: A Step-by-Step Journey
Prerequisites
Math and Python foundations
Data Preprocessing
Handling missing values and scaling
Supervised Learning
Regression and classification techniques
Unsupervised Learning
Clustering and dimensionality reduction
Model Evaluation
Assessing performance metrics
Popular Tools
Utilizing `scikit-learn`, `TensorFlow`, `PyTorch`
Projects
Applying ML to real-world problems
Advanced Topics
Exploring deep learning and NLP

🧠 Types of Variables in Machine Learning



1️⃣ Numerical Variables (Quantitative)

Used for measurable quantities.

- **Continuous**
  - Infinite possible values
  - *Example:* Height, Weight, Temperature
- **Discrete**
  - Countable values
  - *Example:* Number of students, Items sold

2️⃣ Categorical Variables (Qualitative)

Used for names or labels.

- **Nominal**

- No specific order
- *Example:* Gender, Country, Color

🔹 **Ordinal**

- Ordered categories
- *Example:* Education level (High < Medium < High)

3️⃣ Datetime Variables

Store date or time-related values.

- *Example:* Date of Birth, Purchase Date, Login Time
- Useful for extracting:
    - Year, Month, Day
    - Time since event
    - Seasonality, trend (in Time Series)

4️⃣ Mixed-Type Variables

Contain more than one type (usually text + number)

- *Example:* "Product123", "Room No. 5", "Batch2025"
- Require special preprocessing:
    - Splitting numbers and letters
    - Feature engineering

✅ Summary Table

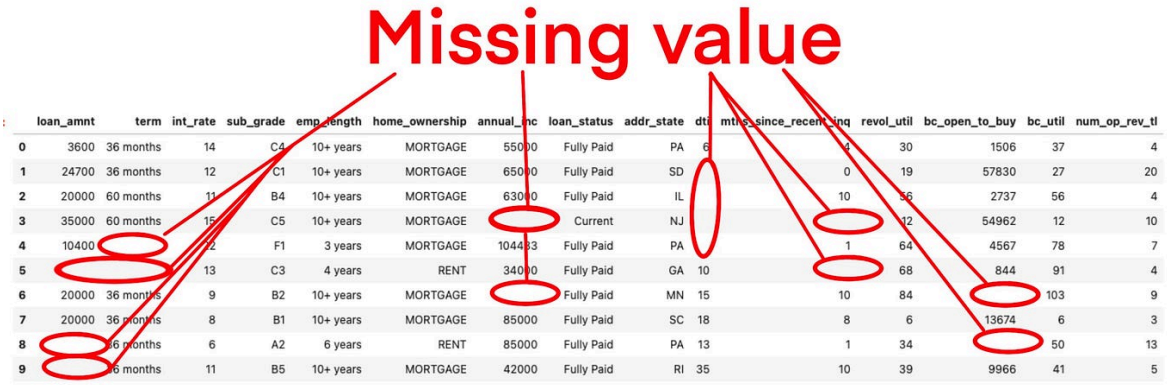
Type	Subtypes	Example
Numerical	Continuous	Weight = 72.5 kg
	Discrete	Age = 18, Children = 3
Categorical	Nominal	Gender = Male
	Ordinal	Level = High
Datetime	-	Date = 2023-10-01
Mixed	-	Code = "Room5", ID = "P123"

Data Cleaning in Machine Learning

Finding Missing Values

🔍 What Are Missing Values?

Missing values are entries in your dataset where data is not available (e.g., NaN or None). Detecting and handling them is a crucial preprocessing step in any machine learning workflow.



	loan_amnt	term	int_rate	sub_grade	emp_length	home_ownership	annual_inc	loan_status	addr_state	dti	mths_since_recent_inq	revol_util	bc_open_to_buy	bc_util	num_op_rev_tl
0	3600	36 months	14	C4	10+ years	MORTGAGE	55000	Fully Paid	PA	6	4	30	1506	37	4
1	24700	36 months	12	C1	10+ years	MORTGAGE	65000	Fully Paid	SD		0	19	57830	27	20
2	20000	60 months	11	B4	10+ years	MORTGAGE	63000	Fully Paid	IL		10	56	2737	56	4
3	35000	60 months	15	C5	10+ years	MORTGAGE		Current	NJ			12	54962	12	10
4	10400		12	F1	3 years	MORTGAGE	104433	Fully Paid	PA		1	64	4567	78	7
5			13	C3	4 years	RENT	34000	Fully Paid	GA	10		68	844	91	4
6	20000	36 months	9	B2	10+ years	MORTGAGE		Fully Paid	MN	15	10	84		103	9
7	20000	36 months	8	B1	10+ years	MORTGAGE	85000	Fully Paid	SC	18	8	6	13674	6	3
8		36 months	6	A2	6 years	RENT	85000	Fully Paid	PA	13	1	34		50	13
9		36 months	11	B5	10+ years	MORTGAGE	42000	Fully Paid	RI	35	10	39	9966	41	5

🛠️ Common Methods to Detect Missing Values

1. dataset.isnull()

- **Description**: Returns a DataFrame of the same shape as dataset, showing True for missing (null) entries and False otherwise.

Example:

```
dataset.isnull()
```

---

2. dataset.isnull().sum()

- **Description**: Returns the total count of missing values **per column**.
- **Use Case**: Helps identify which columns have missing values and how many.

Example:

```
dataset.isnull().sum()
```

---

3. dataset.isnull().sum().sum()

- **Description**: Returns the **total number** of missing values in the entire dataset.
- **Use Case**: Provides a quick overview of how severe the missing value problem is.

Example:

```
dataset.isnull().sum().sum()
```

---

4. (dataset.isnull().sum() / dataset.shape[0]) * 100

- **Description**: Calculates the **percentage of missing values per column**.
- **Use Case**: Helps decide whether to drop or impute columns based on the missing data ratio.

Example:

```
(dataset.isnull().sum() / dataset.shape[0]) * 100
```

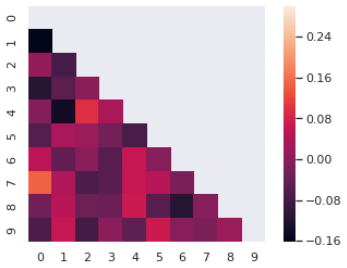
📊 Visualizing Missing Data

5. sns.heatmap(dataset.isnull()) + plt.show()

- **Description**: Shows a heatmap where missing values are visualized.
- **Use Case**: Easy to see patterns (e.g., if a whole row or column is missing values).

Code Example:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.heatmap(dataset.isnull(), cbar=False, cmap='viridis')
plt.show()
```



Dropping a Specific Column with Missing Values

✅ Code:

```
dataset.drop(columns=["Credit_History"], inplace=True)
dataset.isnull().sum()
```

🔍 Explanation:

1. dataset.drop(columns=["Credit_History"], inplace=True)

- **Purpose**: Permanently deletes the column named "Credit_History" from the dataset.
- **Reason**: This is often done when:
    - The column has too many missing values.
    - It is not useful or relevant for the analysis.
    - You prefer to remove rather than impute.

✅ inplace=True means the changes will directly modify the dataset without needing reassignment.

---

2. dataset.isnull().sum()

- **Purpose**: Re-checks the dataset to see how many missing values remain **after dropping the column**.
- **Output**: A series showing missing value counts for each remaining column.

📝 Best Practice

Before dropping:

```
print(dataset["Credit_History"].isnull().sum())
```

After dropping:

```
print(dataset.columns)
print(dataset.isnull().sum())
```

dataset.dropna()

- **Description**: Removes rows with **any missing values**.
- **Example**:

```
cleaned_dataset = dataset.dropna()
```

	A	B	C	D
1	**No.**	**Name**	**Date**	**salary**
2	00001	ana varela	6/1/2016	20930
3	00002	Patricia King	6/1/2016	5410
4	00003	Charles Monaghan	6/1/2016	11350
5	00004		6/1/2016	
6	00005	John Botts	6/1/2016	25390
7	00008	Matthew Martin	6/1/2016	29520
8	00009	JP VAN BEUZEKOM	6/1/2016	
9	00010	Christian Faust	6/1/2016	23060
10	00011	Ricky Kwon	6/1/2016	20060
11	00012	Alfonso Gonzalez Pedregal	6/1/2016	28070
12	00013		6/1/2016	20710
13	00014	Simone Williams	6/1/2016	25020
14	00015	Michael Naidu	6/1/2016	12790
15	00016	Bill Waits	6/1/2016	30620
16	00017	Steffen Helmschrott	6/1/2016	
17	00018	Robert Lanza	6/1/2016	27240
18	00019	Mauro Claudio Coelho	6/1/2016	17560
19	00020	Wiebe Geldenhuys	6/1/2016	600
20	00021		6/1/2016	16280

	A	B	C	D
1	**No.**	**Name**	**Date**	**salary**
2	00001	ana varela	6/1/2016	20930
3	00002	Patricia King	6/1/2016	5410
4	00003	Charles Monaghan	6/1/2016	11350
5	00005	John Botts	6/1/2016	25390
6	00008	Matthew Martin	6/1/2016	29520
7	00010	Christian Faust	6/1/2016	23060
8	00011	Ricky Kwon	6/1/2016	20060
9	00012	Alfonso Gonzalez Pedregal	6/1/2016	28070
10	00014	Simone Williams	6/1/2016	25020
11	00015	Michael Naidu	6/1/2016	12790
12	00016	Bill Waits	6/1/2016	30620
13	00018	Robert Lanza	6/1/2016	27240
14	00019	Mauro Claudio Coelho	6/1/2016	17560
15	00020	Wiebe Geldenhuys	6/1/2016	600

Filling Missing Value

✅ Clean & Proper Way to Fill Missing Values in All Categorical Columns Using Mode:

```
# Fill missing values in all categorical columns with their mode
for col in dataset.select_dtypes(include="object").columns:
    mode_value = dataset[col].mode()[0]
    dataset[col].fillna(mode_value, inplace=True)
```

🔍 Explanation:

- dataset.select_dtypes(include="object"): Selects all categorical (non-numeric) columns.
- .columns: Gets the column names.
- mode()[0]: Returns the most frequent value in that column.
- fillna(..., inplace=True): Fills the missing values directly in the original dataset.

📘 Optional: Add a Print to Confirm What Was Filled

If you want to **see what value was used** to fill each column:

```
for col in dataset.select_dtypes(include="object").columns:
    mode_value = dataset[col].mode()[0]
    dataset[col].fillna(mode_value, inplace=True)
    print(f"Filled missing values in '{col}' with: {mode_value}")
```

🧪 Handling Missing Values Using Scikit-Learn

📦 Import:

```
from sklearn.impute import SimpleImputer
```

🧠 What is SimpleImputer?

SimpleImputer is a part of Scikit-learn's preprocessing module.
It is used to **automatically handle missing values** in numeric and categorical columns using strategies like:

- "mean" (default)
- "median"
- "most_frequent" (mode)
- "constant" (custom value)

✨ Use Case 1: Filling Numeric Columns with Mean

```
import pandas as pd
from sklearn.impute import SimpleImputer

# Create imputer object
imputer = SimpleImputer(strategy='mean')

# Select numeric columns only
numeric_cols = dataset.select_dtypes(include=['int64', 'float64']).columns

# Apply the imputer
dataset[numeric_cols] = imputer.fit_transform(dataset[numeric_cols])
```

✨ Use Case 2: Filling Categorical Columns with Most Frequent (Mode)

```
# Create imputer object for categorical data
cat_imputer = SimpleImputer(strategy='most_frequent')

# Select categorical columns
cat_cols = dataset.select_dtypes(include=['object']).columns

# Apply imputer
dataset[cat_cols] = cat_imputer.fit_transform(dataset[cat_cols])
```

✨ Use Case 3: Filling with a Constant Value

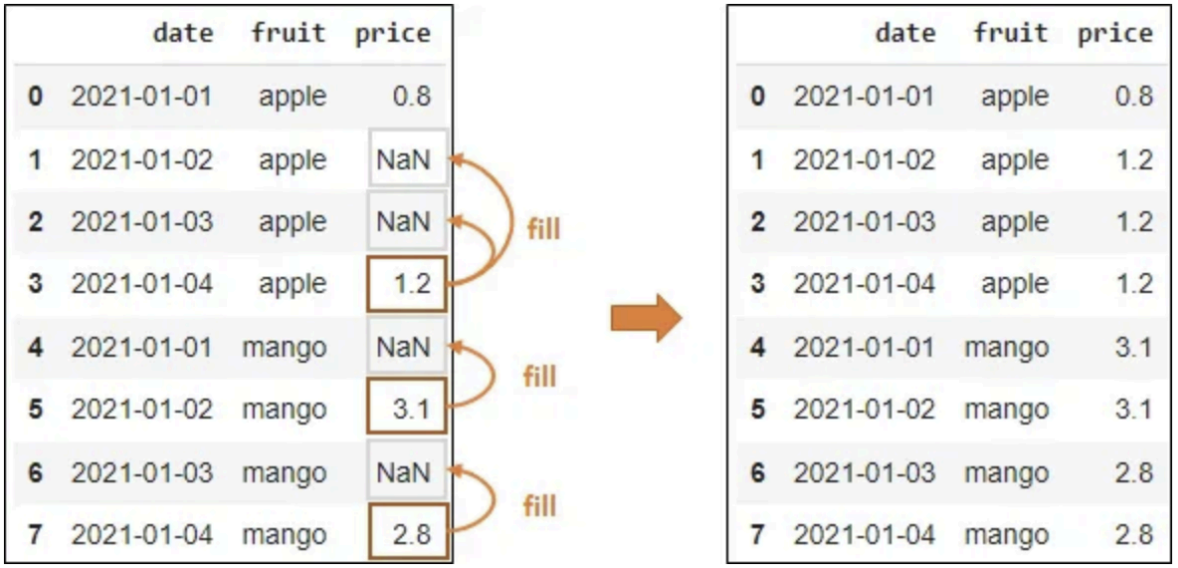
```
# Fill missing with a constant like 'Unknown' or 0
const_imputer = SimpleImputer(strategy='constant', fill_value='Unknown')

dataset[cat_cols] = const_imputer.fit_transform(dataset[cat_cols])
```

🔁 Summary Table

Strategy	Description	Best For
'mean'	Replaces with column mean	Numerical data
'median'	Replaces with column median	Skewed numeric data
'most_frequent'	Replaces with mode (most common)	Categorical data
'constant'	Replaces with a custom constant value	Missing flags, etc.

🧪 Example Dataset Workflow

```
from sklearn.impute import SimpleImputer

num_imputer = SimpleImputer(strategy="mean")
cat_imputer = SimpleImputer(strategy="most_frequent")

dataset[numeric_cols] = num_imputer.fit_transform(dataset[numeric_cols])
dataset[cat_cols] = cat_imputer.fit_transform(dataset[cat_cols])
```



	date	fruit	price
0	2021-01-01	apple	0.8
1	2021-01-02	apple	NaN
2	2021-01-03	apple	NaN
3	2021-01-04	apple	1.2
4	2021-01-01	mango	NaN
5	2021-01-02	mango	3.1
6	2021-01-03	mango	NaN
7	2021-01-04	mango	2.8

	date	fruit	price
0	2021-01-01	apple	0.8
1	2021-01-02	apple	1.2
2	2021-01-03	apple	1.2
3	2021-01-04	apple	1.2
4	2021-01-01	mango	3.1
5	2021-01-02	mango	3.1
6	2021-01-03	mango	2.8
7	2021-01-04	mango	2.8

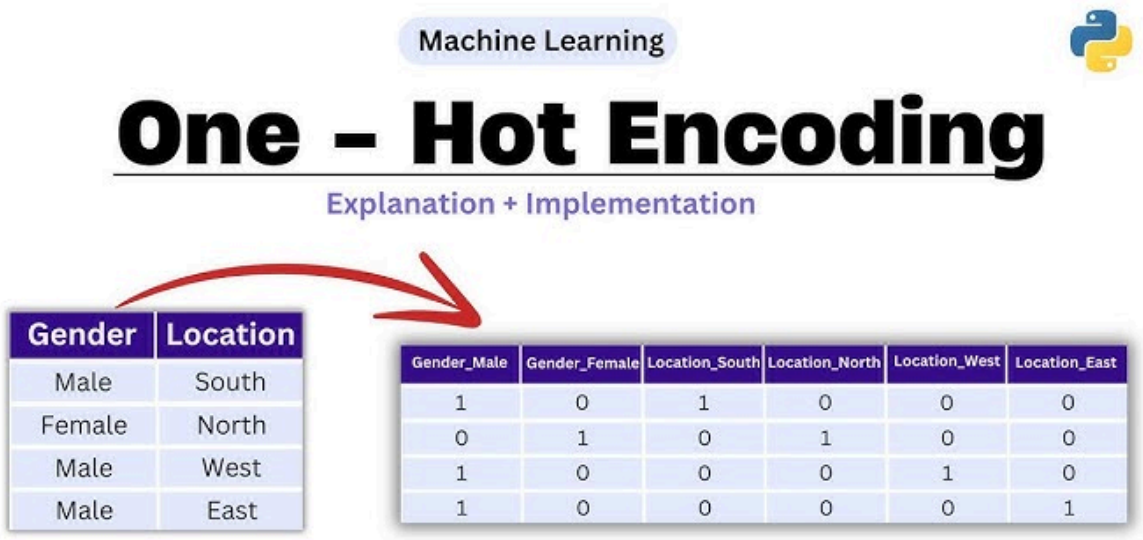
Full Source Code :- Github

🧠 One-Hot Encoding in Machine Learning

🔍 What is One-Hot Encoding?

One-Hot Encoding is a technique used to convert **categorical variables** into a form that can be provided to ML algorithms to do a better job in prediction.

It creates **binary (0 or 1)** columns for each unique category in the feature.



🎯 Why Use One-Hot Encoding?

Many ML models (like Linear Regression, Logistic Regression, etc.) **cannot handle categorical values directly**.
They require all features to be **numerical**.

📘 Example

Original Categorical Data:

Country
India

USA

USA

Canada

After One-Hot Encoding:

Country_Canada	Country_India	Country_USA
0	1	0
0	0	1
0	0	1
1	0	0

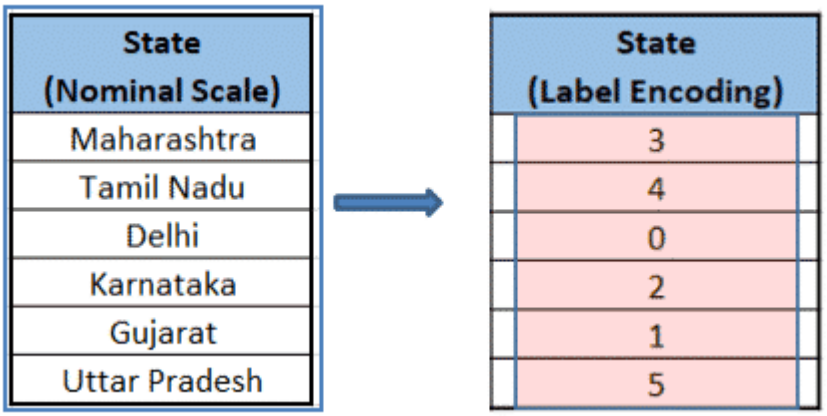
🔁 Using Scikit-Learn: OneHotEncoder

```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

# Sample data
data = pd.DataFrame({'Gender': ['Male', 'Female', 'Female', 'Male']})

# Create encoder object
encoder = OneHotEncoder(sparse=False, drop=None)

# Fit and transform
encoded = encoder.fit_transform(data[['Gender']])

# Convert to DataFrame
encoded_df = pd.DataFrame(encoded, columns=encoder.get_feature_names_out(['Gender']))
print(encoded_df)
```

🔢 Label Encoding in Machine Learning

🔍 What is Label Encoding?

Label Encoding is the process of converting **categorical (text) values into numeric labels**.
Each category is assigned an integer value starting from 0.



State (Nominal Scale)
Maharashtra
Tamil Nadu
Delhi
Karnataka
Gujarat
Uttar Pradesh

State (Label Encoding)
3
4
0
2
1
5

📘 Example

Original Categorical Data:

Gender
Male
Female
Female
Male

After Label Encoding:

Gender	Gender_Encoded
Male	1
Female	0
Female	0

Male 1

🛠️ Using Scikit-Learn: LabelEncoder

```
from sklearn.preprocessing import LabelEncoder
import pandas as pd

# Sample data
data = pd.DataFrame({'Gender': ['Male', 'Female', 'Female', 'Male']})

# Create encoder object
le = LabelEncoder()

# Fit and transform
data['Gender_Encoded'] = le.fit_transform(data['Gender'])
```

🔁 How It Works

- le.fit(data['Gender']): Learns the mapping from categories to integers.
- le.transform(...): Converts values to numerical labels.

You can check the mapping:

```
print(le.classes_)  # Output: ['Female' 'Male']
```

✅ When to Use Label Encoding

Use Case	Use Label Encoding?
Ordinal Data (with order)	✅ Yes
Nominal Data (no order)	❌ No (Use One-Hot)
Tree-based models (e.g. DecisionTree, RandomForest)	✅ Yes

⚠️ Warning:

Label Encoding **introduces an ordinal relationship** (e.g., 0 < 1 < 2), which can be **misleading for nominal data** like "Red", "Green", "Blue".

> ❌ Don't use for linear models unless the data has a natural order.

🔄 Decode Labels (Optional)

You can convert encoded values back to original labels:

```
data['Original'] = le.inverse_transform(data['Gender_Encoded'])
```

🧪 Full Example:

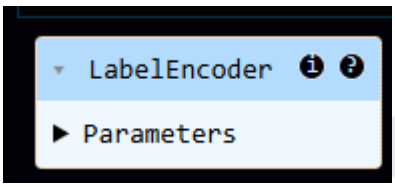
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

data = pd.DataFrame({'City': ['Delhi', 'Mumbai', 'Kolkata', 'Delhi']})

le = LabelEncoder()
data['City_Label'] = le.fit_transform(data['City'])

print(data)
```

Output:

```
      City  City_Label
0    Delhi           0
1   Mumbai           2
2  Kolkata           1
3    Delhi           0
```

🔁 Alternative: Use OrdinalEncoder for multi-column categorical data

```
from sklearn.preprocessing import OrdinalEncoder

encoder = OrdinalEncoder()
data[['City']] = encoder.fit_transform(data[['City']])
```

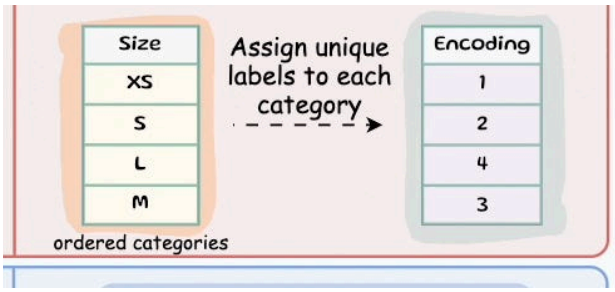


🧮 Ordinal Encoding in Machine Learning

🔍 What is Ordinal Encoding?

Ordinal Encoding converts **categorical features** into integer values **based on the rank/order** of the categories.

Unlike One-Hot or Label Encoding, **Ordinal Encoding is appropriate only when the categories have a meaningful order** (e.g., Low < Medium < High).



🧠 When to Use Ordinal Encoding?

Use Case	Ordinal Encoding?
Categorical with **order**	✅ Yes
Categorical with **no order**	❌ No (use One-Hot)
Examples: Education Level, Rank, Rating	✅ Yes

📘 Example:

Original Data:

Education
High
Medium
Low
Medium

After Ordinal Encoding:

Education	Encoded
High	2
Medium	1
Low	0
Medium	1

🛠️ Using Scikit-Learn: OrdinalEncoder

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

# Sample data
data = pd.DataFrame({'Education': ['High', 'Medium', 'Low', 'Medium']})

# Define custom order
categories = [['Low', 'Medium', 'High']]  # Must be nested list per column

# Create encoder
encoder = OrdinalEncoder(categories=categories)

# Transform
data['Education_Encoded'] = encoder.fit_transform(data[['Education']])

print(data)
```

```
   Education  Education_Encoded
0       High                2.0
1     Medium                1.0
2        Low                0.0
3     Medium                1.0
```

⚠️ Important Notes:

- **Always define the order explicitly** using categories=[['Low', 'Medium', 'High']]
- If you **don't define order**, it will default to alphabetical: High=0, Low=1, etc. — which can be **incorrect**
- Returns **float** by default; you can cast to int if needed

```
data['Education_Encoded'] = data['Education_Encoded'].astype(int)
```

🔁 Decode Back (if needed)

```
decoded = encoder.inverse_transform(data[['Education_Encoded']])
print(decoded)
```

🧪 Full Example:

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

df = pd.DataFrame({'Size': ['Small', 'Medium', 'Large', 'Medium']})

# Specify order
categories = [['Small', 'Medium', 'Large']]

encoder = OrdinalEncoder(categories=categories)
df['Size_Encoded'] = encoder.fit_transform(df[['Size']])

print(df)
```

🗒️ Summary: Comparison with Other Encodings

Encoding Type	Preserves Order	Increases Dimensionality	Suitable For

Label Encoding	❌ No	❌ No	Nominal (Tree models)
One-Hot Encoding	❌ No	✅ Yes	Nominal
Ordinal Encoding	✅ Yes	❌ No	Ordinal

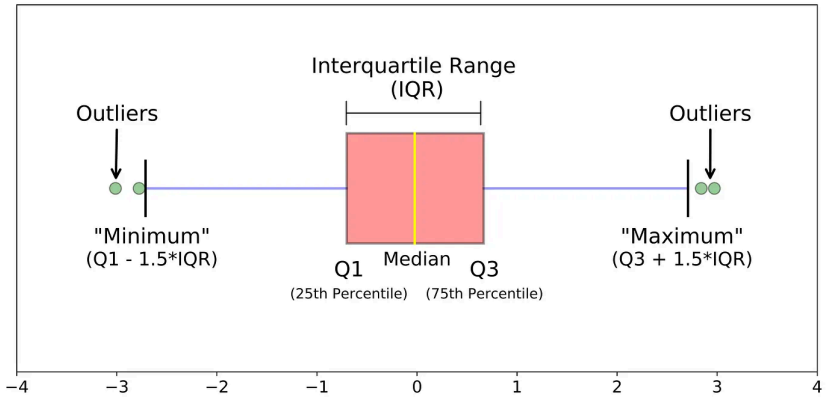
Full Source Code Link :- Github

📌 Detecting and Removing Outliers in Python

📘 What are Outliers?

Outliers are data points that **differ significantly** from other observations in a dataset.
They can:

- Skew your model performance
- Mislead interpretation
- Affect statistics like mean and standard deviation



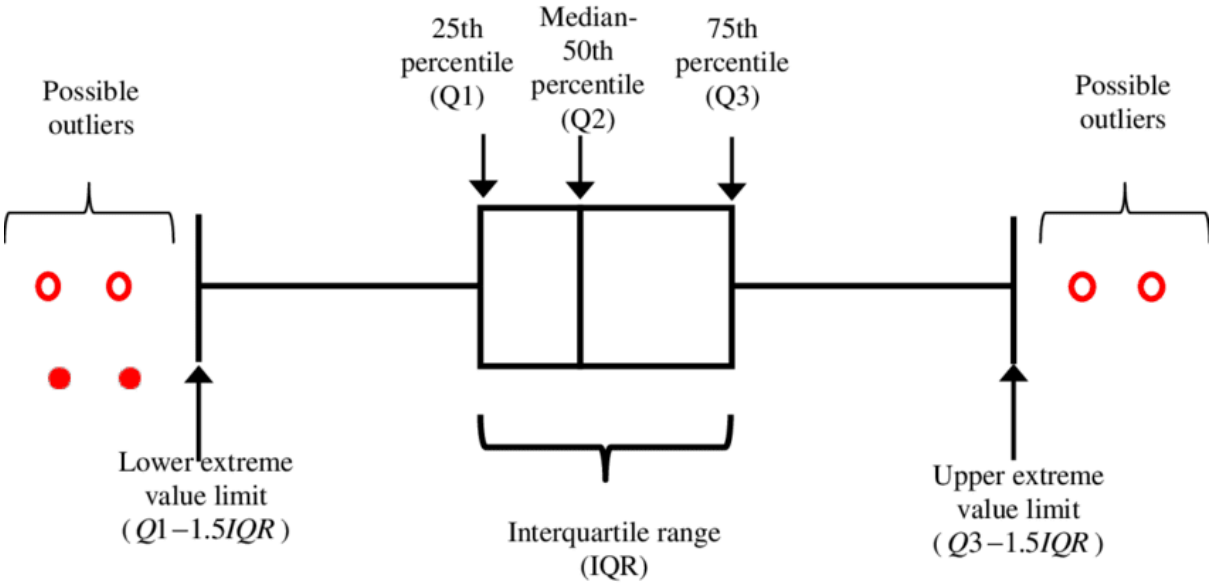
📂 Sample Dataset

```
import pandas as pd

data = {
   'Name': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
   'Salary': [25000, 27000, 30000, 28000, 26000, 29000, 27500, 29500, 50000, 1000000]
}

df = pd.DataFrame(data)
print(df)
```

	Name	Salary
0	A	25000
1	B	27000
2	C	30000
3	D	28000
4	E	26000
5	F	29000
6	G	27500
7	H	29500
8	I	50000
9	J	1000000

🔍 Step 1: Visual Detection Using Boxplot

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.boxplot(x=df['Salary'])
plt.title('Boxplot of Salary')
plt.show()
```



Points far outside the boxplot whiskers are considered outliers.

🧮 Step 2: Detect Outliers Using IQR Method

```
Q1 = df['Salary'].quantile(0.25)
Q3 = df['Salary'].quantile(0.75)
IQR = Q3 - Q1

lower_limit = Q1 - 1.5 * IQR
upper_limit = Q3 + 1.5 * IQR

print("Lower Limit:", lower_limit)
print("Upper Limit:", upper_limit)

# Detecting Outliers
outliers_iqr = df[(df['Salary'] < lower_limit) | (df['Salary'] > upper_limit)]
print("Outliers Detected (IQR):\n", outliers_iqr)
```

```
Lower Limit: 23000.0
Upper Limit: 34000.0
Outliers Detected (IQR):
   Name   Salary
8    I    50000
9    J  1000000
```

✅ Step 3: Remove Outliers (IQR)

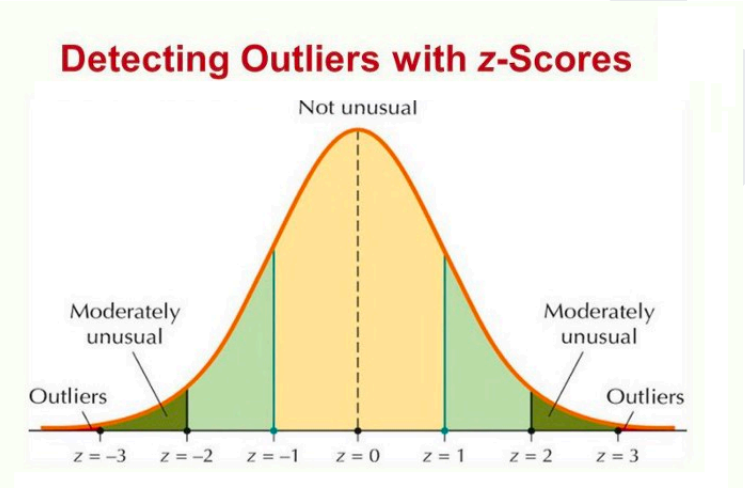
```
df_iqr_cleaned = df[(df['Salary'] >= lower_limit) & (df['Salary'] <= upper_limit)]
print("Cleaned Dataset (IQR):\n", df_iqr_cleaned)
```

```
Cleaned Dataset (IQR):
   Name  Salary
0    A   25000
1    B   27000
2    C   30000
3    D   28000
4    E   26000
5    F   29000
6    G   27500
7    H   29500
```

🧠 Step 4: Detect Outliers Using Z-Score

```
from scipy.stats import zscore

df['z_score'] = zscore(df['Salary'])

# Threshold typically used: ±3
df_z_cleaned = df[(df['z_score'] > -3) & (df['z_score'] < 3)]

# Drop z_score column if not needed
df_z_cleaned.drop(columns=['z_score'], inplace=True)

print("Cleaned Dataset (Z-Score):\n", df_z_cleaned)
```

```
Cleaned Dataset (Z-Score):
   Name   Salary
0    A    25000
1    B    27000
2    C    30000
3    D    28000
4    E    26000
5    F    29000
6    G    27500
7    H    29500
8    I    50000
9    J  1000000
```



🗒️ Summary Table

Method	Good For	Threshold	Strength
Boxplot	Visualization	--	Easy to interpret
IQR	Skewed numeric data	1.5 * IQR	No assumption of normality
Z-Score	Normally distributed	±3	Best for Gaussian data

⚠️ When NOT to Remove Outliers:

- When they are **valid** observations (e.g., CEO salary)
- When you're building **robust models** (e.g., tree-based models)
- When the outliers indicate **important phenomena** (e.g., fraud detection)

Full Code:- Github

⚖️ Feature Scaling in Machine Learning

📌 Why Scale Features?

Machine learning algorithms (especially distance-based ones like KNN, K-Means, SVM, and Gradient Descent–based models) are **sensitive to the scale of features**.

✅ Feature Scaling ensures:

- All features contribute equally to the model
- Faster convergence in gradient-based models
- Better performance and accuracy

🧭 Types of Feature Scaling

1. Normalization (Min-Max Scaling)

- Scales data between **0 and 1**
- Formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- Good when data **doesn't follow a normal distribution**

📘 **Code (Using MinMaxScaler):**

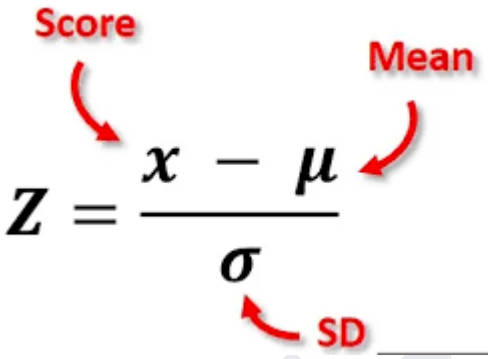
```
from sklearn.preprocessing import MinMaxScaler
import pandas as pd

# Sample Data
data = pd.DataFrame({'Salary': [20000, 25000, 40000, 50000, 100000]})
```

```
scaler = MinMaxScaler()
data['Salary_Normalized'] = scaler.fit_transform(data[['Salary']])

print(data)
```

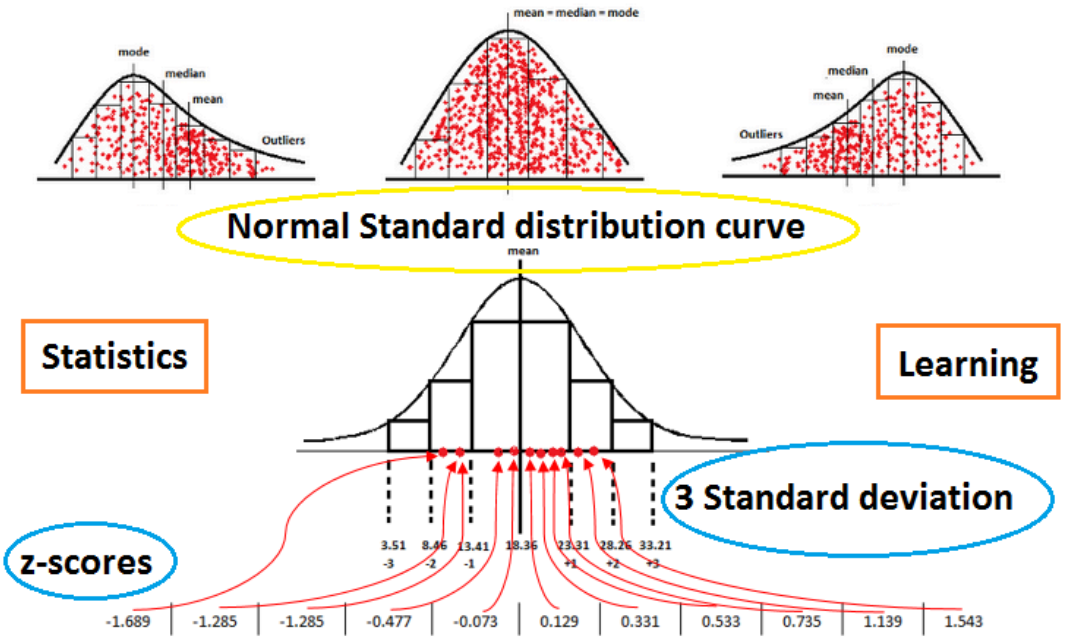
---

2. Standardization (Z-Score Scaling)

- Scales data to have **mean = 0** and **standard deviation = 1**
- Formula:



- Works well with **normally distributed** data



📘 **Code (Using StandardScaler):**

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
data['Salary_Standardized'] = scaler.fit_transform(data[['Salary']])

print(data)
```

📊 Comparison Table

Technique	Scales To	Preserves Outliers?	Use When
Normalization	0 to 1	❌ No	When features are not Gaussian
Standardization	Mean = 0, SD = 1	✅ Yes	When data is normally distributed
Robust Scaling	Median-based	✅ Yes	When outliers are present

---

3. RobustScaler (Bonus)

- Scales using **median and IQR**
- Good for **datasets with outliers**

```
from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()
data['Salary_Robust'] = scaler.fit_transform(data[['Salary']])
```

🧪 Full Example: All Scalers

```python
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler
import pandas as pd

df = pd.DataFrame({'Age': [20, 30, 40, 50, 60, 100]})

df['MinMax'] = MinMaxScaler().fit_transform(df[['Age']])
df['Standard'] = StandardScaler().fit_transform(df[['Age']])
df['Robust'] = RobustScaler().fit_transform(df[['Age']])

print(df)
```

⚠️ Important Notes

Always **fit scalers only on training data**, then transform both train and test sets.

```python
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- **Don't scale categorical variables** directly — encode them first (One-Hot, Label).

Source Code:- Github

🔁 Handling Duplicate Values in a Dataset

🧠 Why Remove Duplicates?

Duplicate rows can:

- Mislead data analysis and model performance
- Inflate the importance of certain patterns
- Affect accuracy, especially in classification/regression tasks

📊 Step-by-Step Guide

🔍 1. Check for Duplicate Rows

```
# Returns True/False for each row if it is duplicated
dataset.duplicated()
```

✅ **See the count of duplicates:**

```
dataset.duplicated().sum()
```

📌 This tells you how many **duplicate rows** exist (excluding the first occurrence).

🔎 2. View Duplicate Rows

```
# Display only duplicate rows
duplicates = dataset[dataset.duplicated()]
print(duplicates)
```

❌ 3. Remove Duplicate Rows

```
# Drop all duplicate rows and keep the first occurrence
dataset.drop_duplicates(inplace=True)
```

⚙️ Optional Parameters:

Parameter	Description
keep='first'	(default) keeps the first occurrence
keep='last'	keeps the last occurrence
keep=False	removes **all** duplicates (no exceptions)

```
dataset.drop_duplicates(keep=False, inplace=True)
```

🧪 4. Remove Duplicates Based on Specific Columns

```
# Drop duplicates considering only selected columns
dataset.drop_duplicates(subset=['Name', 'Email'], inplace=True)
```

🔁 5. Reset Index (Optional)

After removing duplicates, you might want to reset the index:

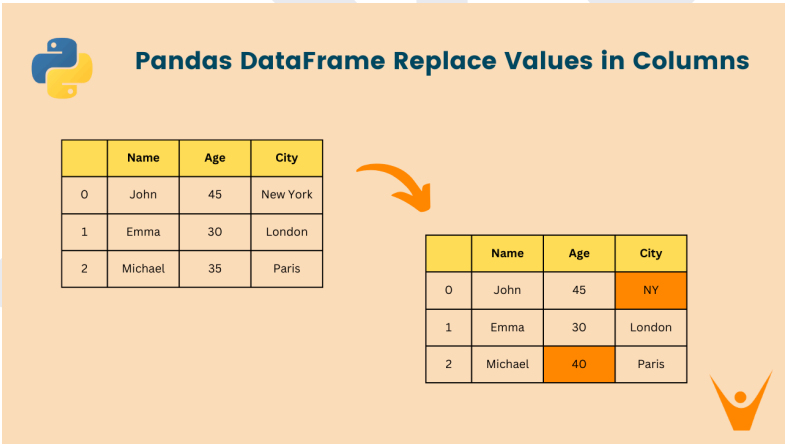
```
dataset.reset_index(drop=True, inplace=True)
```

Source Code:- Github

🔄 Replacing Values & Changing Data Types in Pandas

📌 1. Replacing Values in a Dataset

🧠 Why Replace Values?

- To clean dirty data (e.g., “Male ” → “Male”)
- To encode categorical strings as numbers (e.g., “Yes” → 1)
- To unify formats (e.g., “N/A” → np.nan)
- 

✅ Replace Single Value

```
dataset['Gender'].replace('M', 'Male', inplace=True)
```

✅ Replace Multiple Values

```
dataset['Gender'].replace({'M': 'Male', 'F': 'Female'}, inplace=True)
```

✅ Replace Missing/Custom Strings with NaN

```
import numpy as np

dataset.replace(['N/A', 'na', 'unknown'], np.nan, inplace=True)
```

✅ Replace in Entire DataFrame

```
dataset.replace({'Yes': 1, 'No': 0}, inplace=True)
```

🧪 Example:

```
data = {'Gender': ['M', 'F', 'F', 'M', 'M'],
        'Status': ['Yes', 'No', 'Yes', 'No', 'Yes']}

import pandas as pd
df = pd.DataFrame(data)

df.replace({'M': 'Male', 'F': 'Female', 'Yes': 1, 'No': 0}, inplace=True)
```

🔀 2. Changing Data Types

🔎 Check Data Types

```
print(dataset.dtypes)
```

🔁 Convert Column to Integer

```
dataset['Age'] = dataset['Age'].astype(int)
```

🔁 Convert to Float

```
dataset['Salary'] = dataset['Salary'].astype(float)
```

🔁 Convert to String

```
dataset['ID'] = dataset['ID'].astype(str)
```

🔁 Convert to Datetime

```
dataset['Join_Date'] = pd.to_datetime(dataset['Join_Date'])
```

⚠️ Handle Conversion Errors

```
# If some values can't be converted, force errors to NaT (missing datetime)
dataset['Join_Date'] = pd.to_datetime(dataset['Join_Date'], errors='coerce')
```

Source Code:- Github

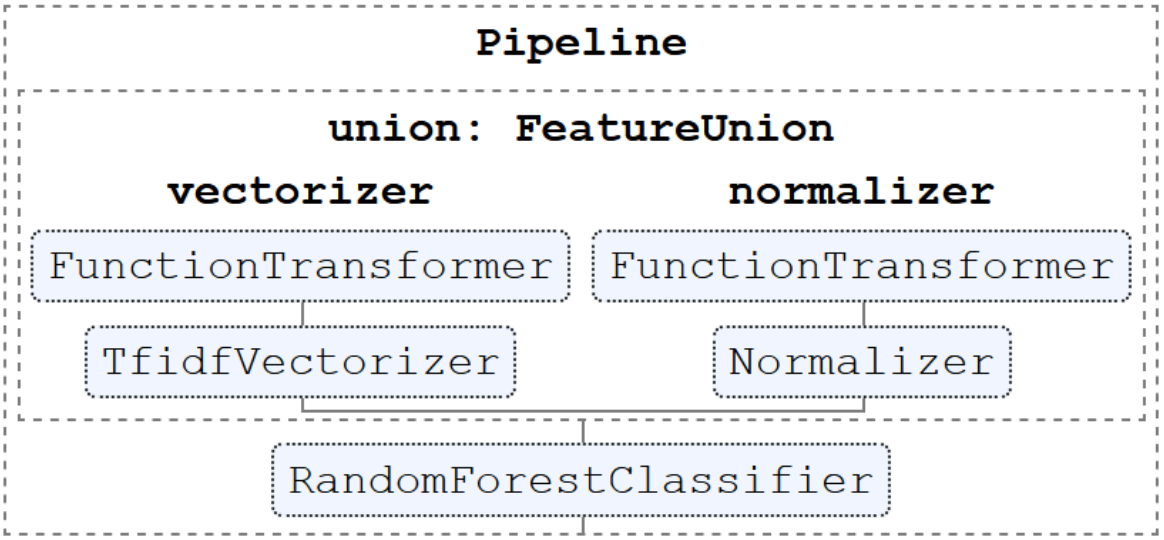
Vikas

🔧 FunctionTransformer in Scikit-Learn

📌 What is FunctionTransformer?

FunctionTransformer is a utility in **Scikit-learn** that allows you to wrap a custom or existing function (like log, sqrt, etc.) and apply it consistently in preprocessing pipelines.

> It is especially useful in pipelines where transformations need to be **repeatable**, **fitted**, and **applied consistently** on train/test splits.



⚙️ Syntax

```
from sklearn.preprocessing import FunctionTransformer

transformer = FunctionTransformer(func, inverse_func=None, validate=True)
```

Parameter	Description
func	Function to apply (e.g., np.log1p)
inverse_func	Function to inverse-transform (optional)

validate	Validates input shape (ensure 2D)

🎯 Why Use It?

- Compatible with **Pipeline** and **ColumnTransformer**
- Useful for **log**, **square root**, **power** transformations
- Allows **custom preprocessing** steps

🧪 Example 1: Apply log1p to a Column

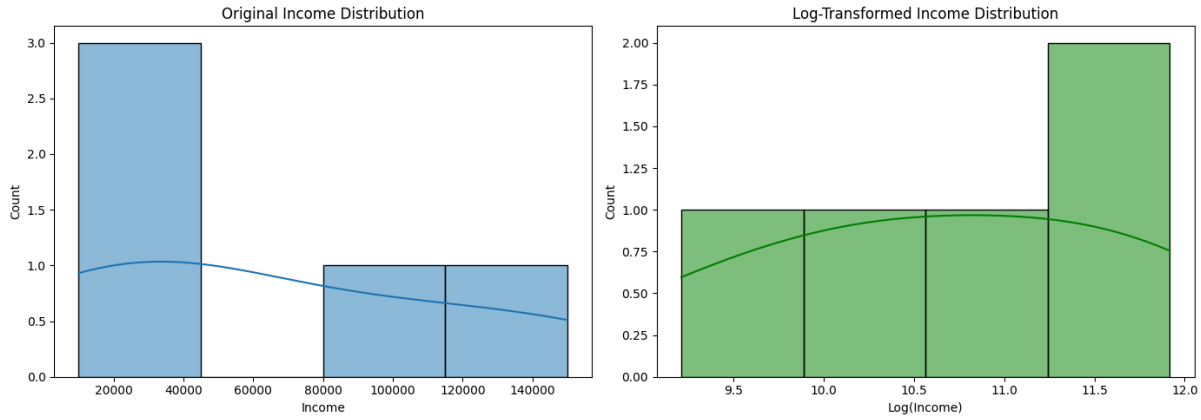
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import FunctionTransformer

data = pd.DataFrame({'Income': [10000, 25000, 40000, 100000, 150000]})

# Apply log transformation: log(1 + x)
log_transformer = FunctionTransformer(np.log1p, validate=True)

data['Income_Log'] = log_transformer.fit_transform(data[['Income']])
print(data)
```

```
   Income  Income_Log
0   10000    9.210440
1   25000   10.126671
2   40000   10.596660
3  100000   11.512935
4  150000   11.918397
```

📈 Visualizing Before & After (Optional)

```
import seaborn as sns
```

```
import matplotlib.pyplot as plt

# Create subplots: 1 row, 2 columns
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot original distribution
sns.histplot(data['Income'], kde=True, ax=axes[0])
axes[0].set_title("Original Income Distribution")
axes[0].set_xlabel("Income")

# Plot log-transformed distribution
sns.histplot(data['Income_Log'], kde=True, color='green', ax=axes[1])
axes[1].set_title("Log-Transformed Income Distribution")
axes[1].set_xlabel("Log(Income)")

# Improve layout
plt.tight_layout()
plt.show()
```



🧪 Example 2: Custom Transformation (e.g., Square Root)

🧠 Use Case in a Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression

pipeline = Pipeline(steps=[
    ('log_transform', FunctionTransformer(np.log1p)),
    ('model', LinearRegression())
])
```

🔁 Inverse Transformation

You can use inverse_func to recover the original values:

```
log_trans = FunctionTransformer(np.log1p, inverse_func=np.expm1)
X_log = log_trans.transform(data[['Income']])
X_original = log_trans.inverse_transform(X_log)
```

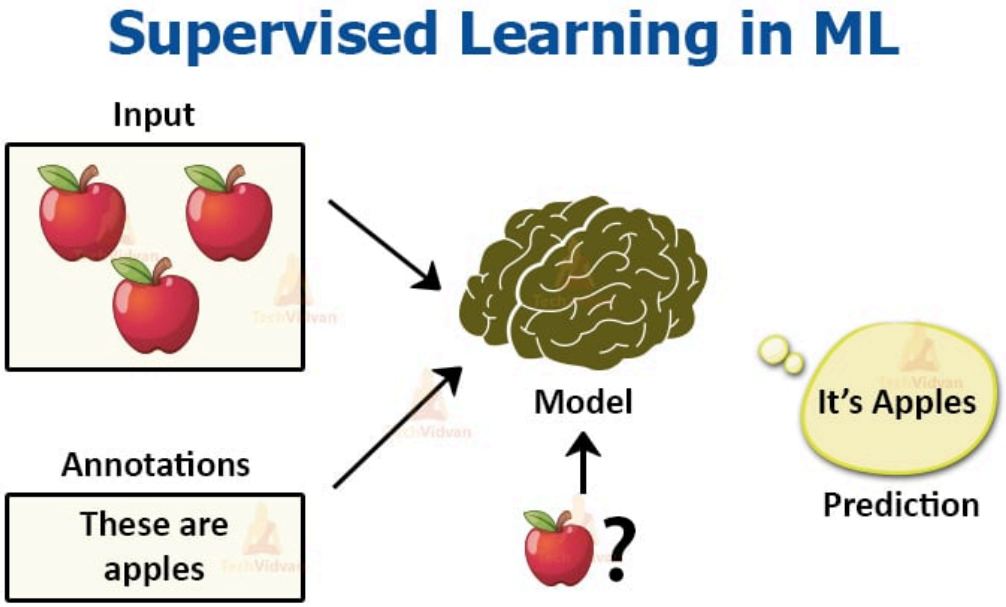
Source Code :- Github

📘 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

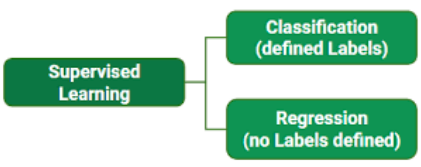
- Email spam detection
- Loan approval
- Disease prediction
- Stock price prediction
- House price estimation

🧩 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:

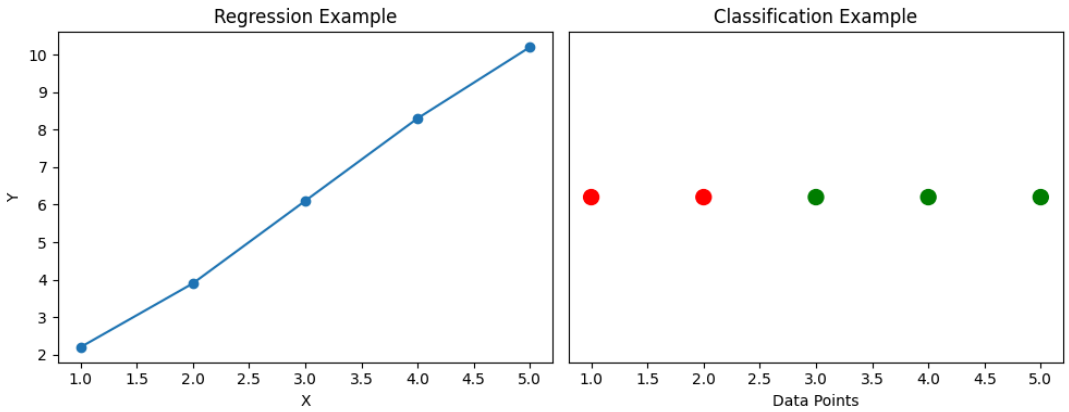
- Logistic Regression
- K-Nearest Neighbors (KNN)
- Decision Tree Classifier
- Random Forest Classifier
- Support Vector Machine (SVM)
- Naive Bayes

📊 **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:

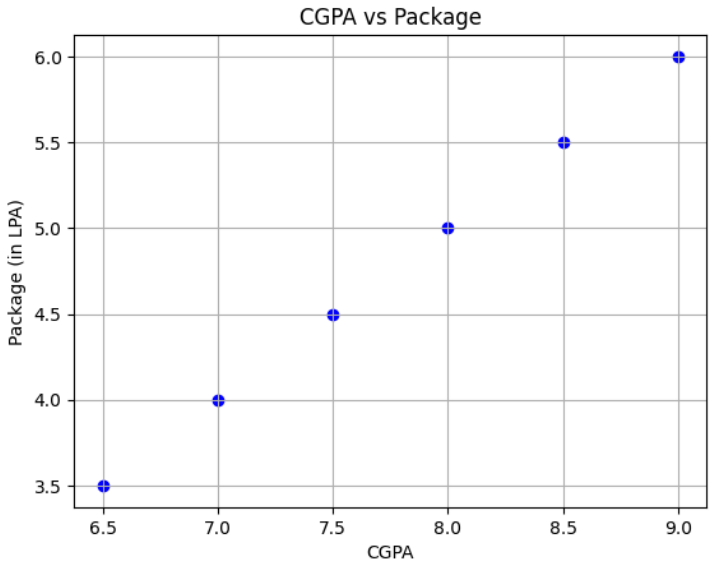
Predicting a student's **salary package** (Y) based on their **CGPA** (X).

📦 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

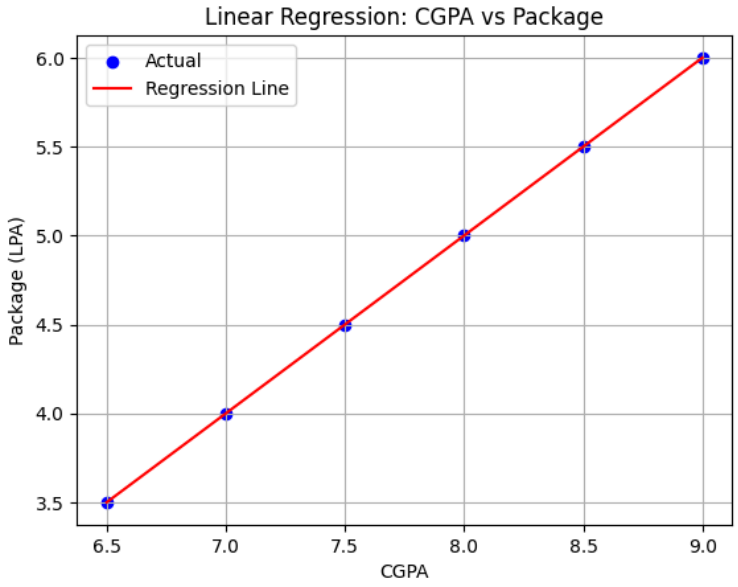
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# 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

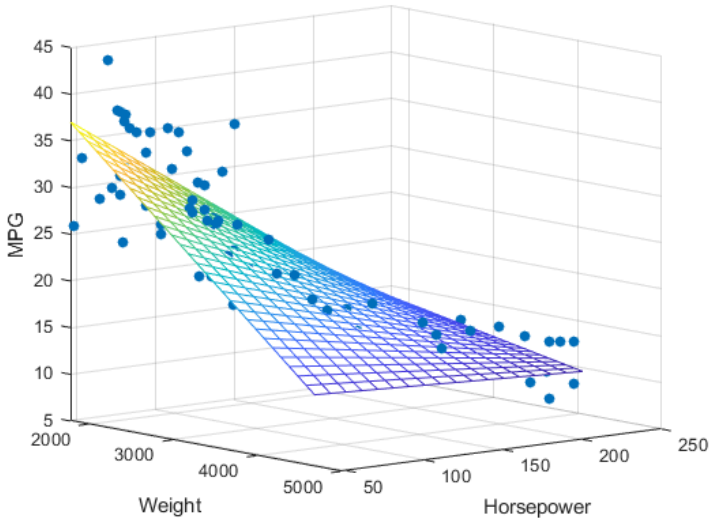
What is Multiple Linear Regression?

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=b0+b1X1+b2X2+...+bnXn+ϵY = b_0 + b_1X_1 + b_2X_2 + ... + b_nX_n + \epsilon

- **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

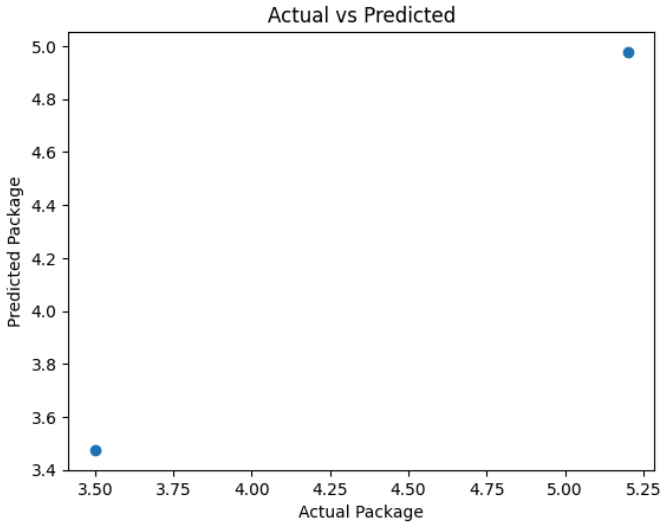
```
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
print("R-squared Score:", r2_score(y_test, y_pred))
```

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()
```



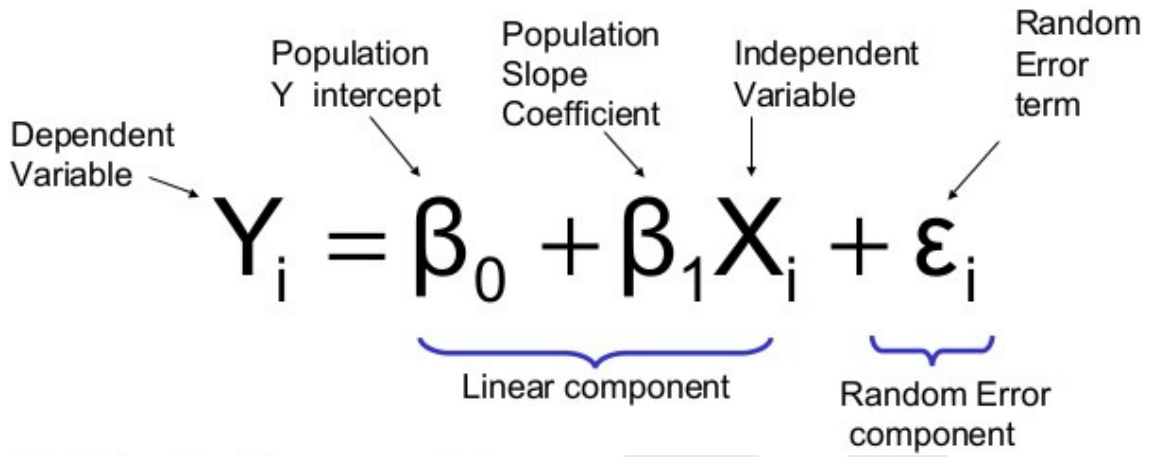
Applications:

- 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?

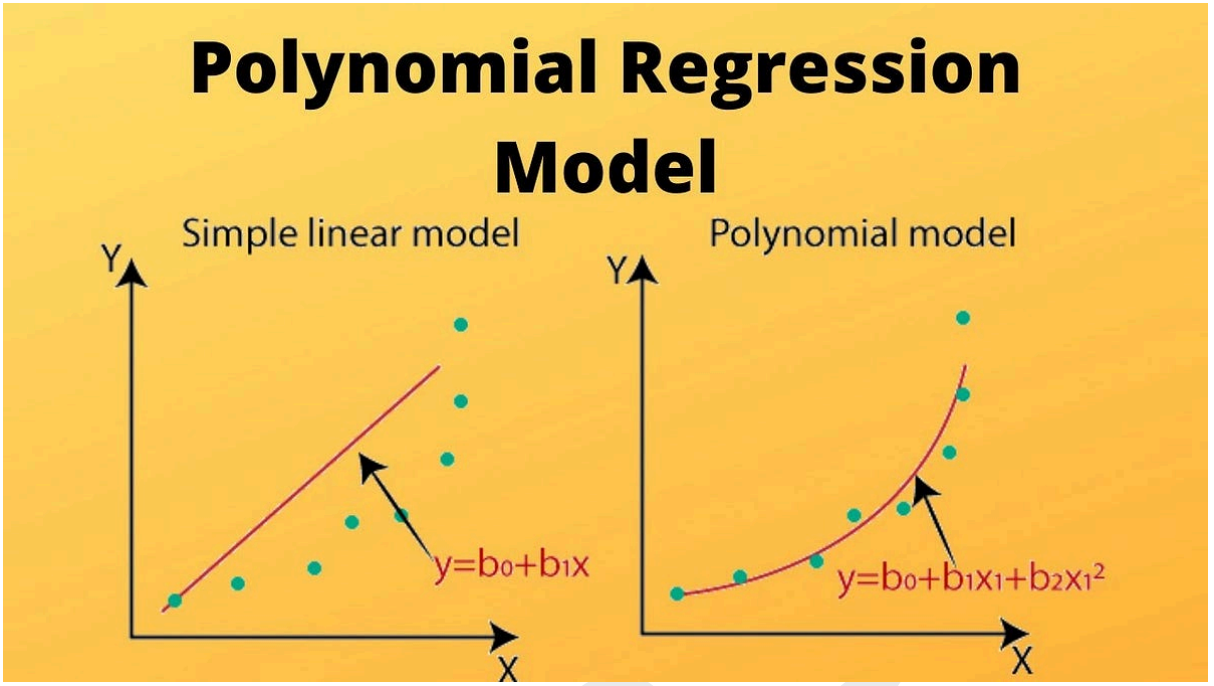
- Linear regression fails to capture curves in data.
- Polynomial regression adds **powers of features** to model non-linearity.
- It still uses **Linear Regression under the hood**, but on **transformed polynomial features**.

---

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

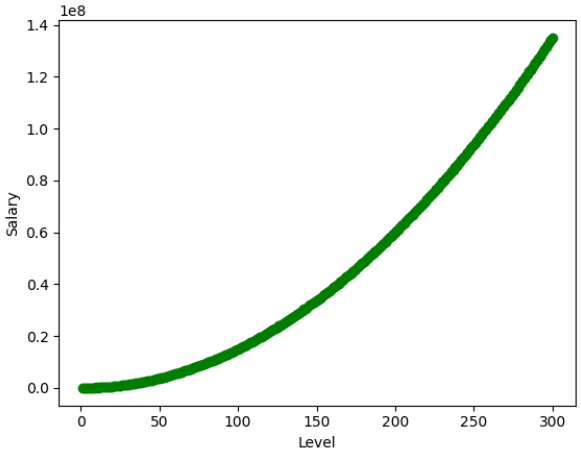


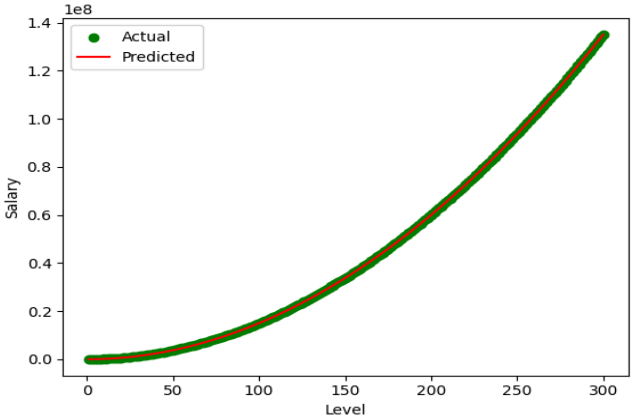
---

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

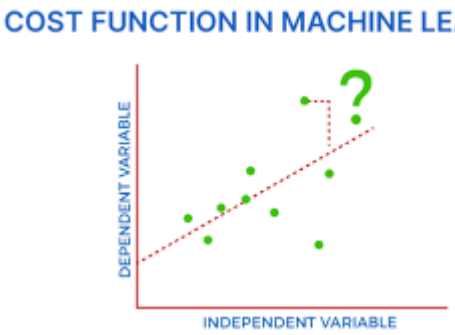
- **Degree selection:** Higher degree → more flexibility, but risk of overfitting.
- **Overfitting:** Too many polynomial terms can make the model fit training data too perfectly but fail in new data.
- **Use cross-validation** to choose the right degree.

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

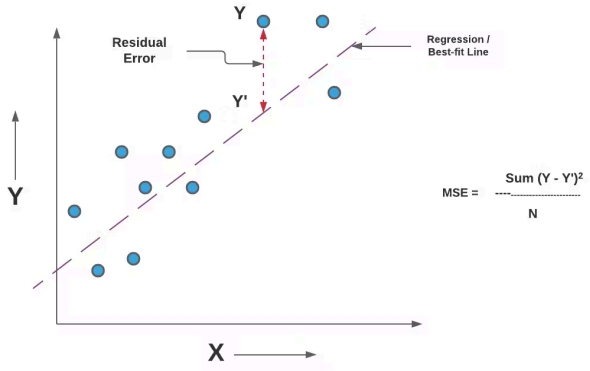
Types of Cost Functions

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

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

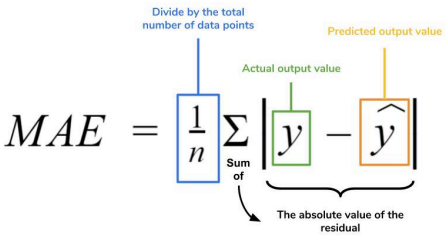
Mean Error Squared

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



---

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



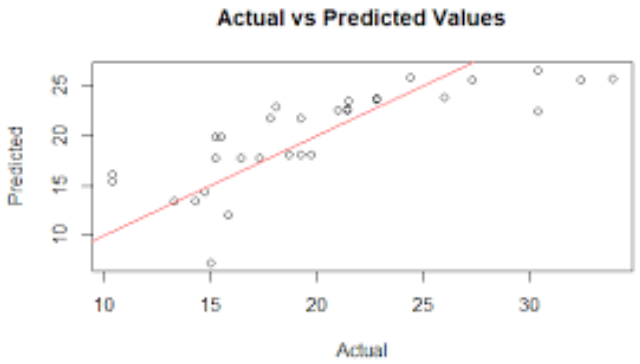
- Takes the **absolute difference** between predicted and actual values.
- Less sensitive to outliers compared to MSE.

---

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

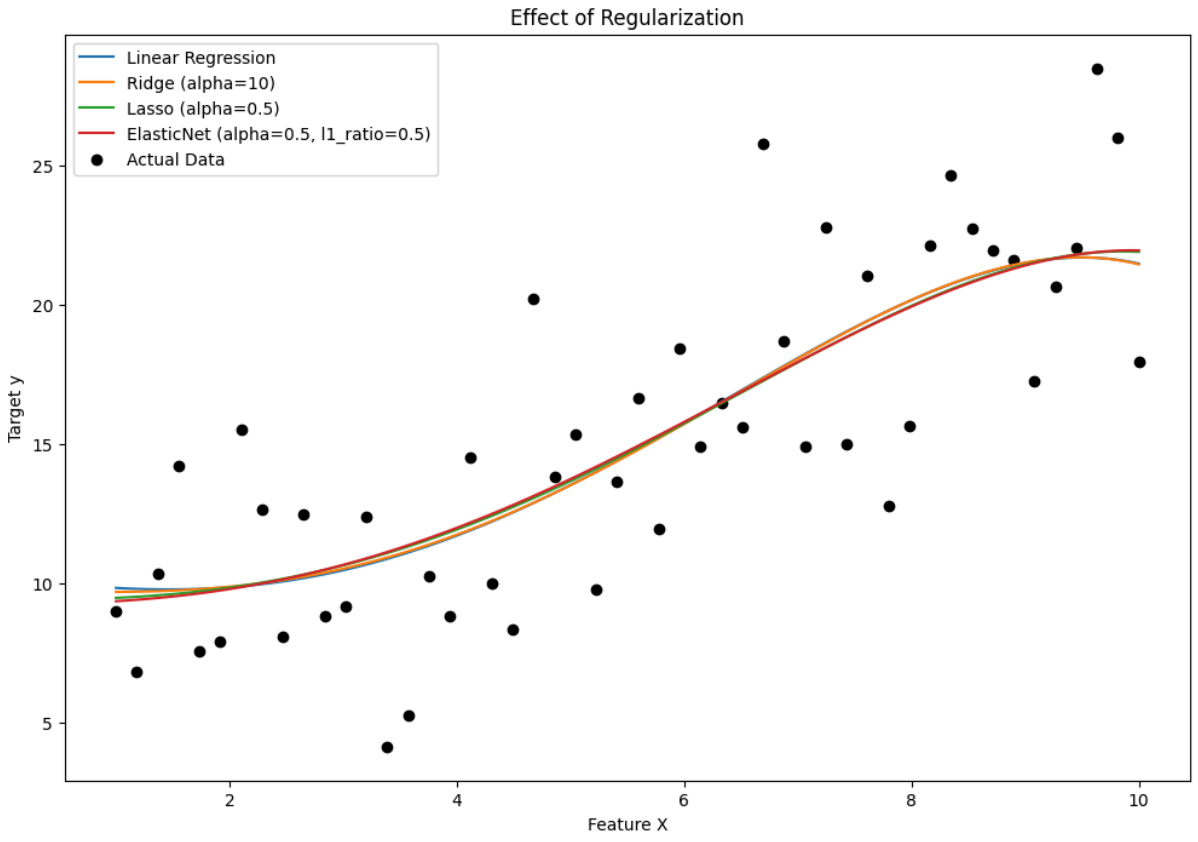
- With regularization: Model is smoother → Better generalization

---

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.

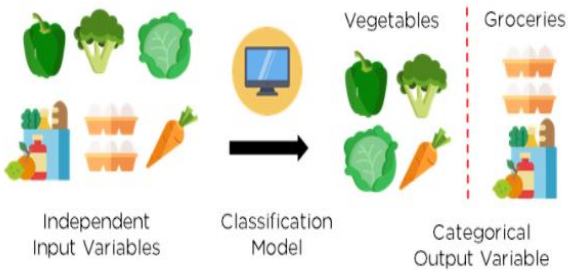
Classification Analysis

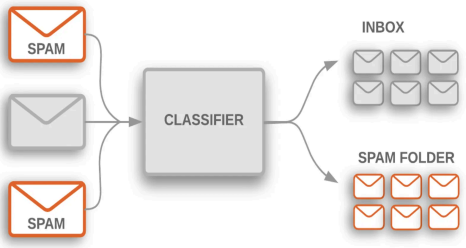
1. What is Classification Analysis?

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)

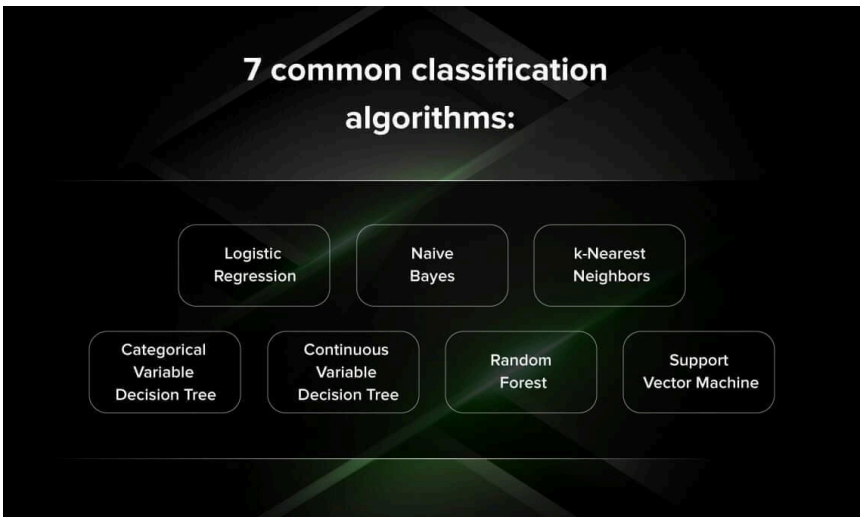
B. Multi-Class Classification

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:**

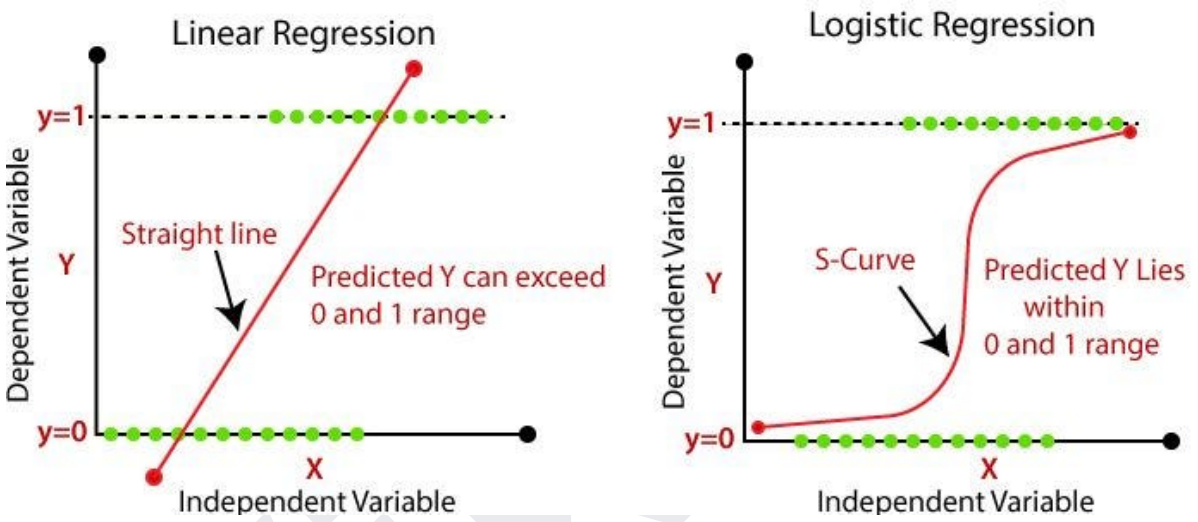
$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

- **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)

---

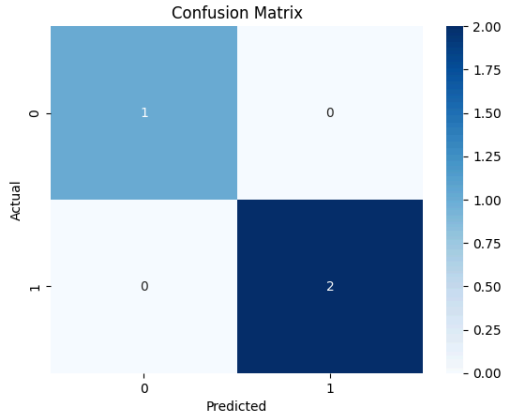
7. Example – Logistic Regression in Python

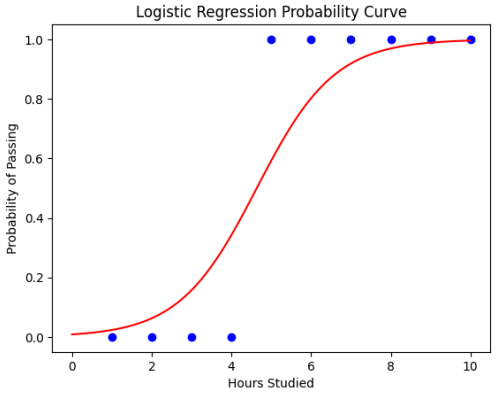
Let's create a **custom dataset** for "Exam pass prediction based on hours studied."

```
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

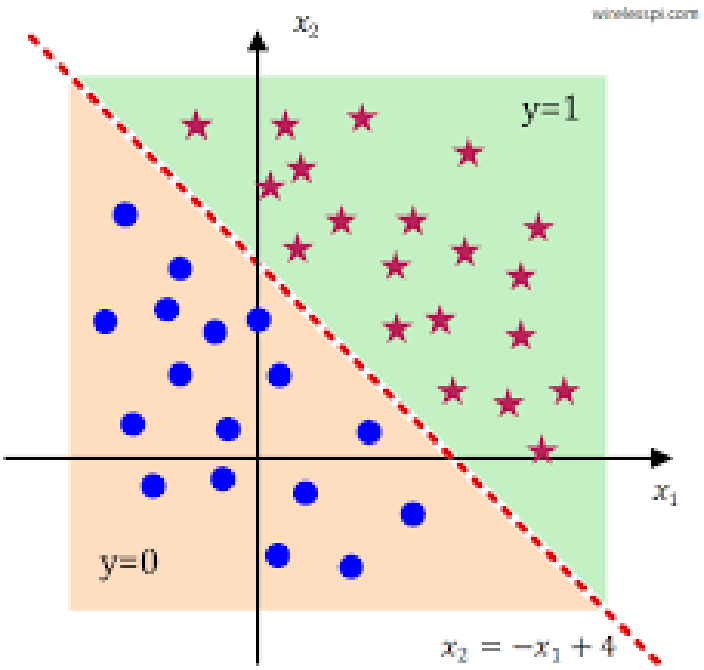
- **Output:** Probability of belonging to a class.
- **Evaluation metrics:** Accuracy, Precision, Recall, F1-score, ROC-AUC.
- **Best for:** Binary classification problems.
- **Limitations:** Assumes linear relationship between independent variables and log-odds.

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

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

---

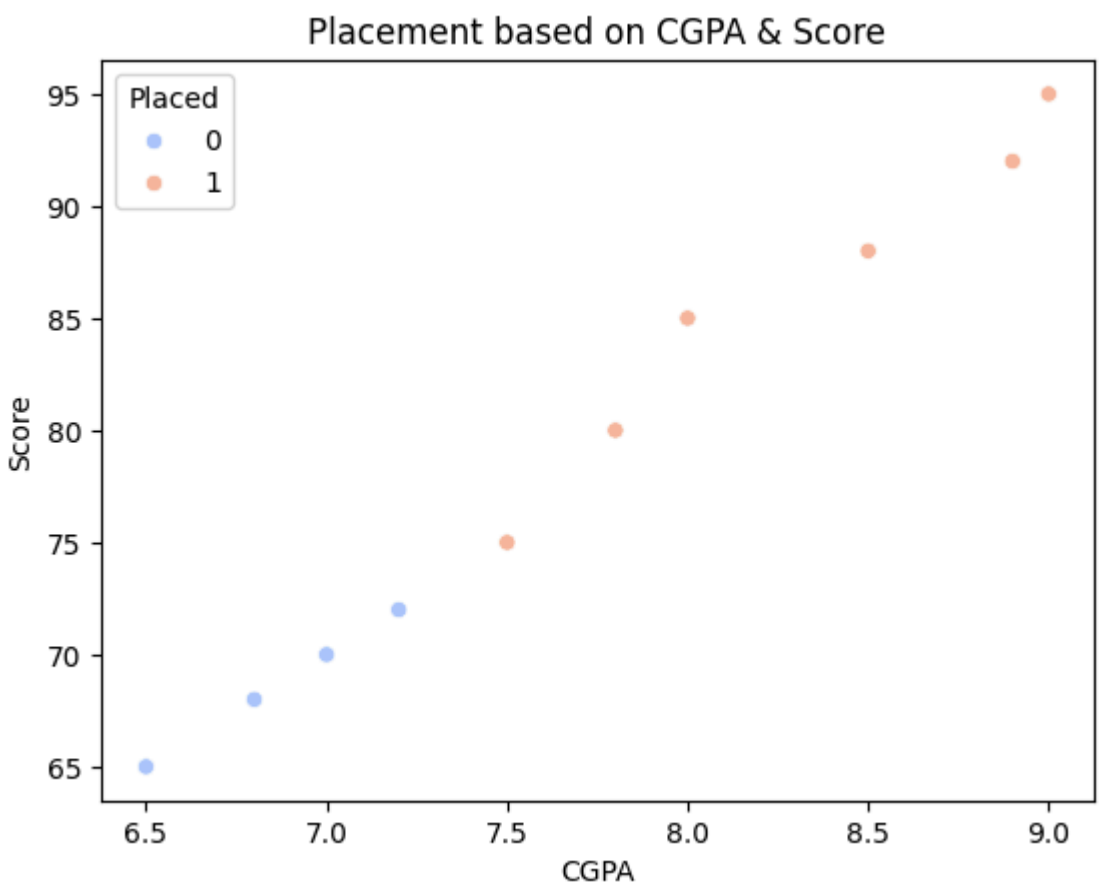
3. Implementation Steps

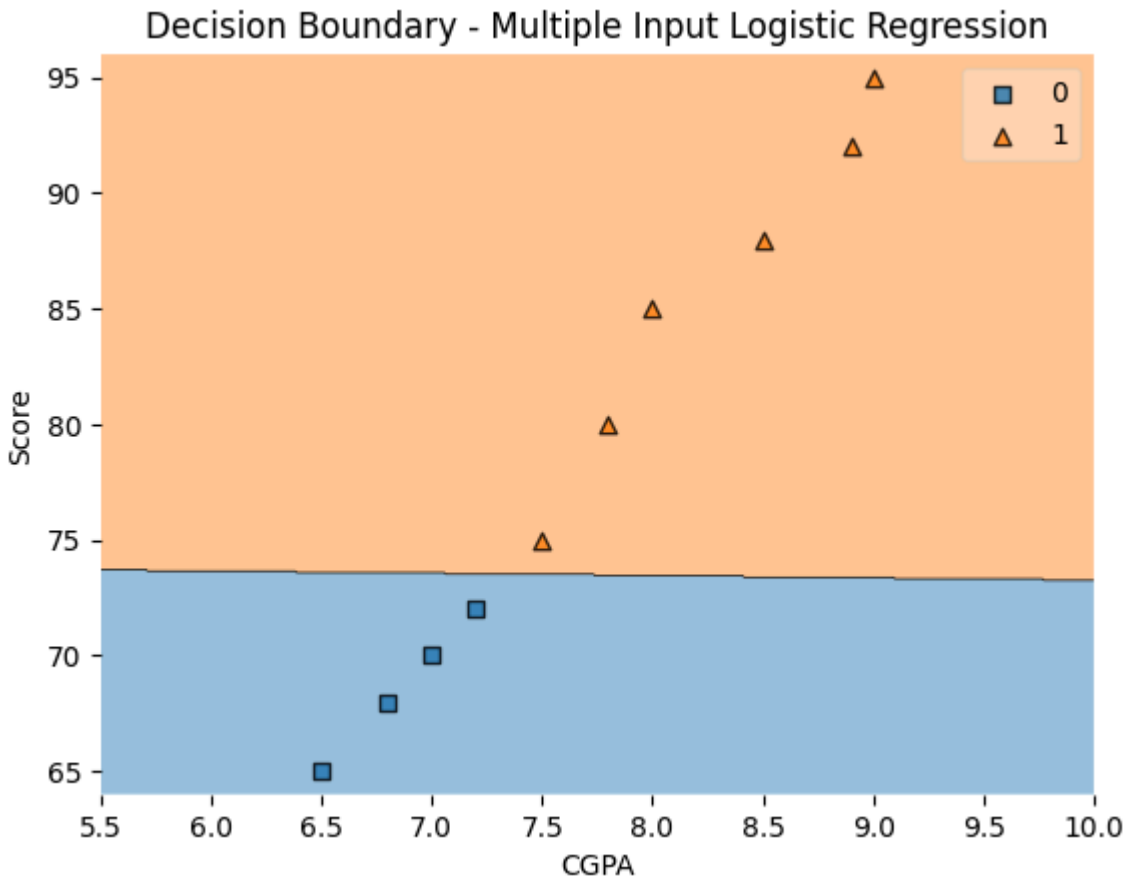
```
# Step 1: Import Libraries
```

```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

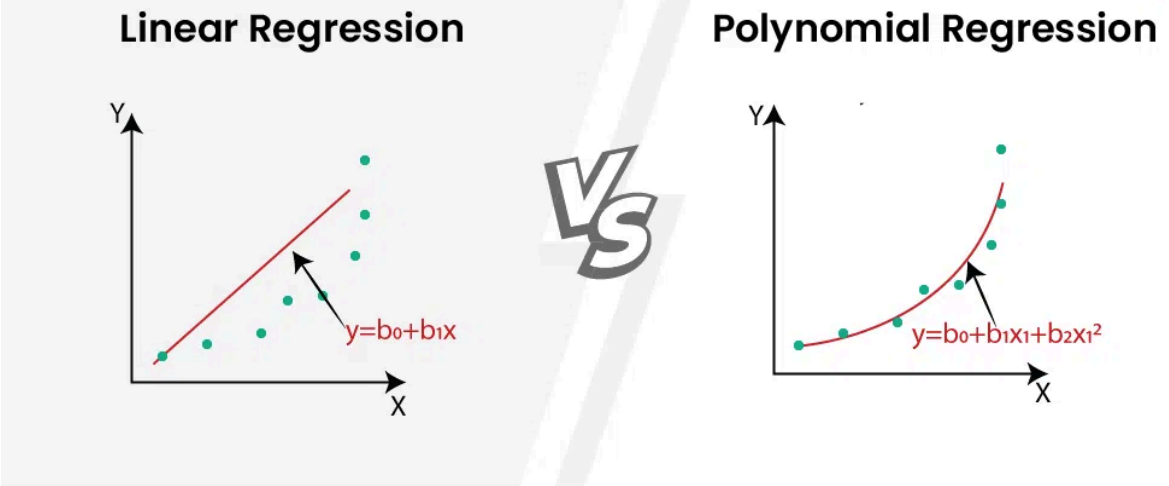
- **Multiple features** improve classification power compared to single-feature logistic regression.
- Decision boundaries can be visualized using plot_decision_regions from mlxtend.
- Works best when the relationship between **log-odds** and inputs is **linear**.

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):

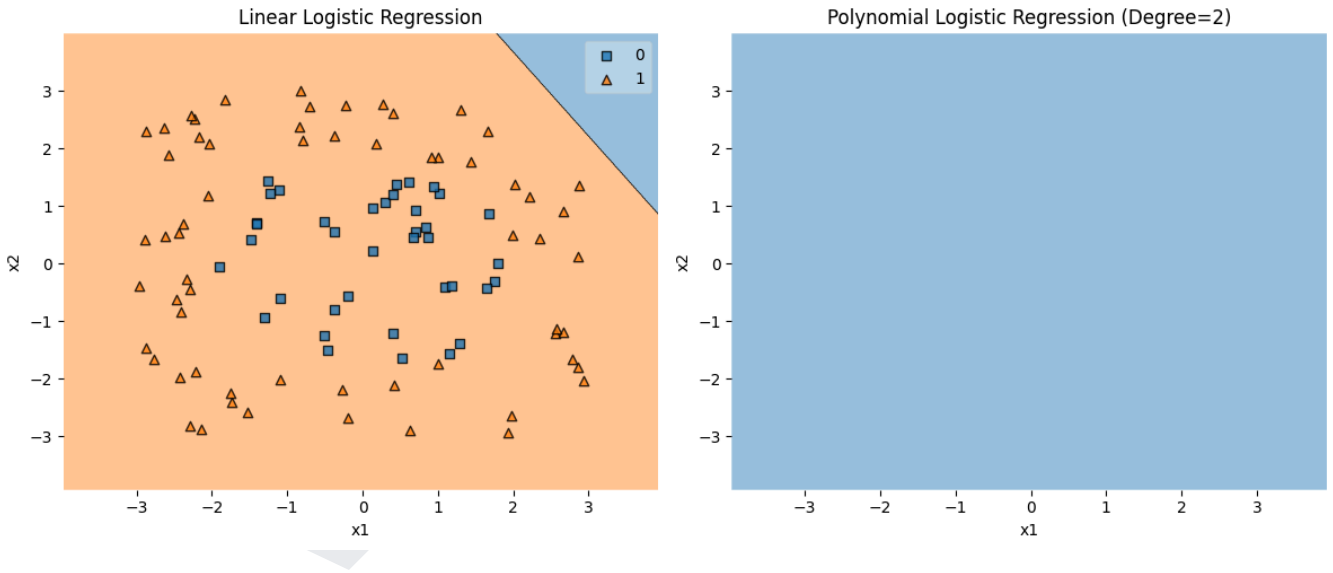
$$P(Y=1) = \frac{1}{1+e^{-(b_0+b_1X_1+b_2X_2+b_3X_1^2+b_4X_2^2+b_5X_1X_2)}}$$

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

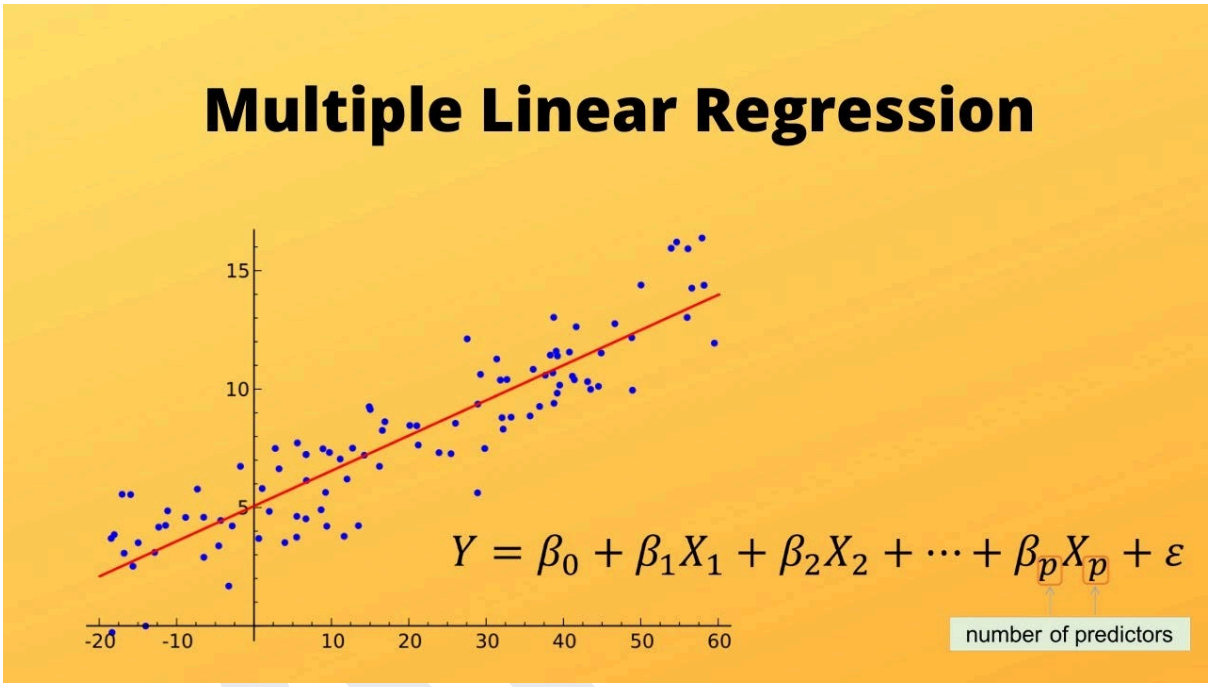
Fit to Data	Works well for linearly separable data	Works for **non-linear patterns**
Complexity	Simple, easy to interpret	More complex (higher features)
Overfitting	Less prone	Higher risk if degree is too high
Use Case	Simple binary classification	Circular, spiral, or complex boundaries

---

📘 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 🌸)

```
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**:

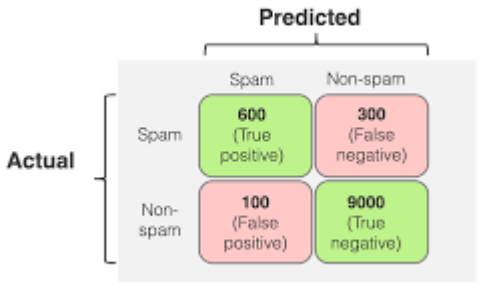
- **OvR** = simple, fast, but less accurate for correlated classes.
- **Multinomial (Softmax)** = slower, but usually better accuracy & probability calibration.

📊 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.

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

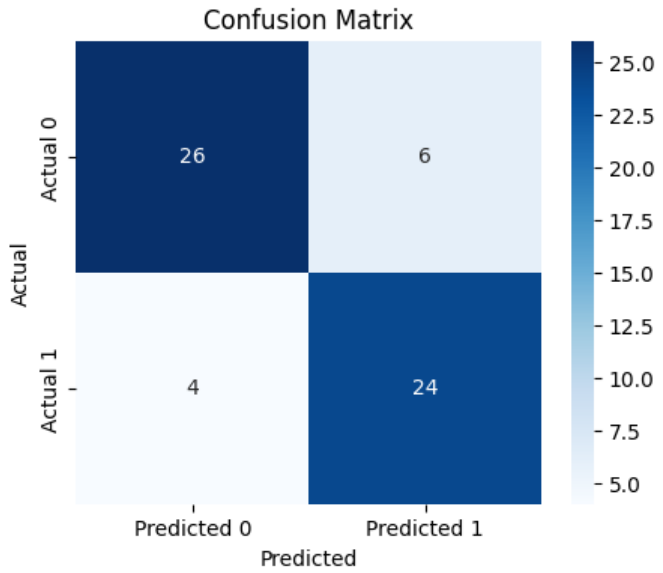
🔹 Why is Confusion Matrix Important?

✔ 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

```
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))
```



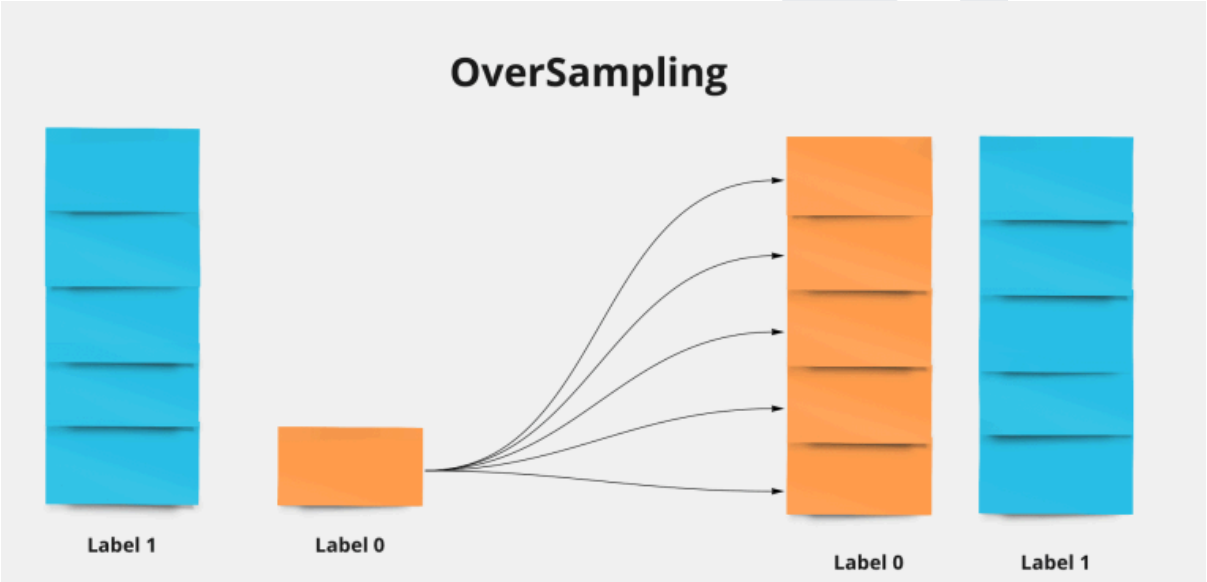

```
              precision    recall  f1-score   support

           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.

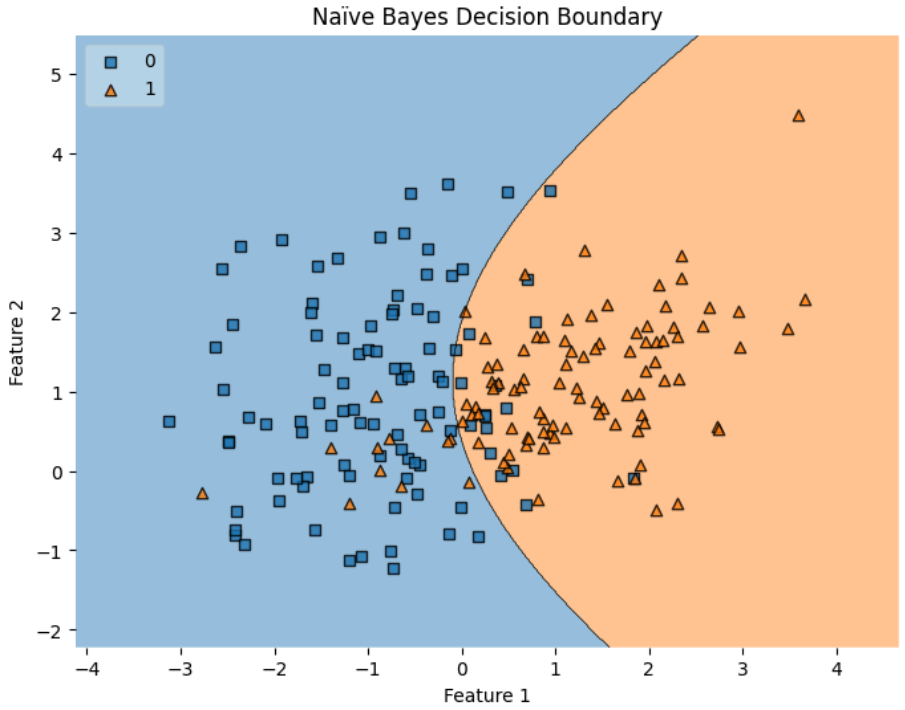
🔹 4. Types of Naïve Bayes

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)
```

```python
# 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

---

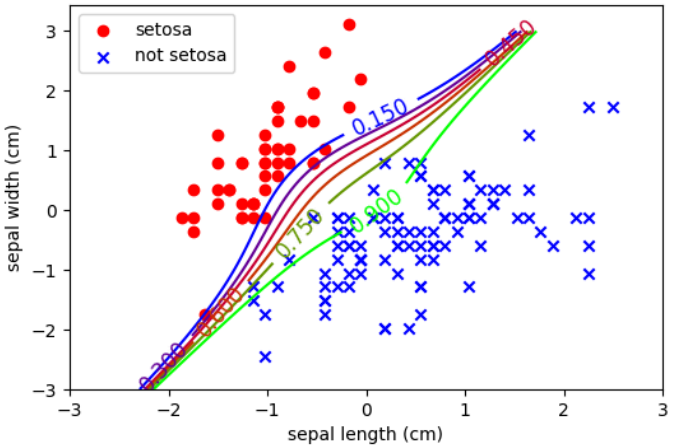
🔹 8. Applications

- 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**.



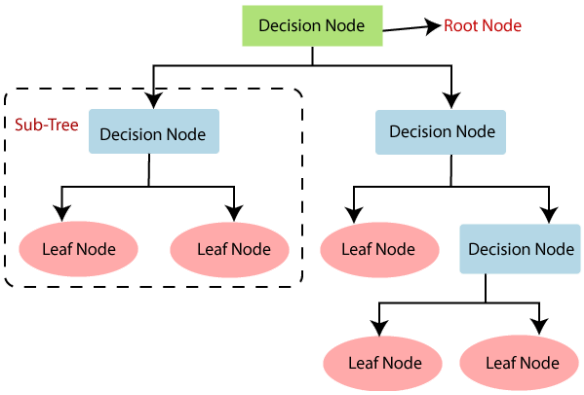
👉 In short:

- 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:

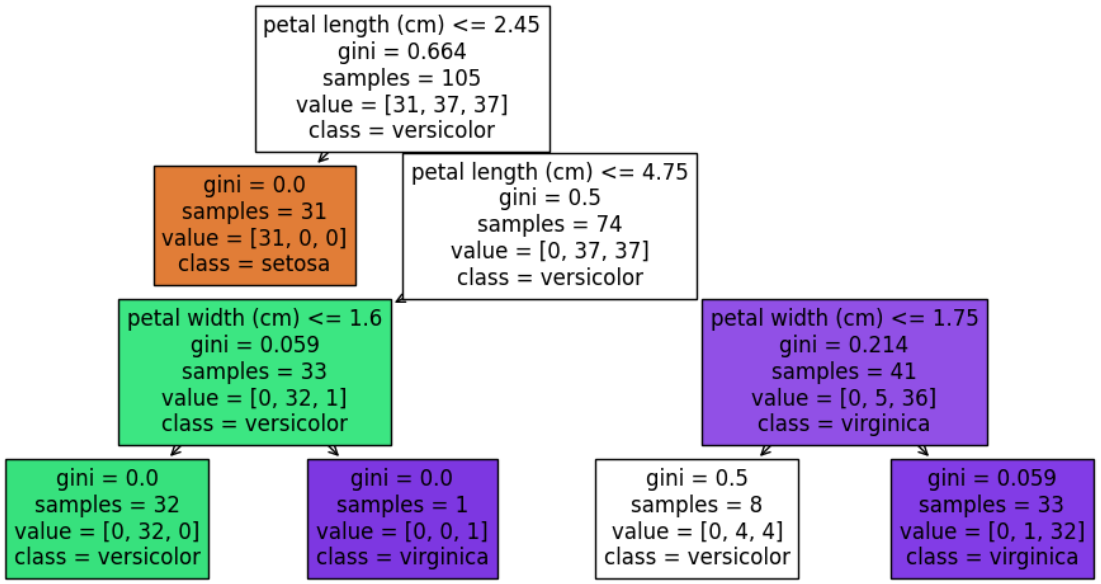
- If Age > 30 → Yes
- Else If Salary > 50k → Yes
- Else → No

🔷 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)
```

```
# 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



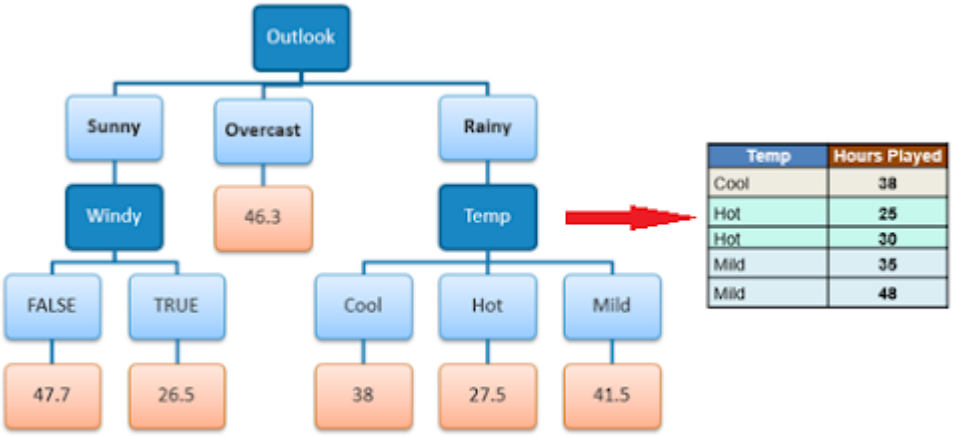
🔹 Output:

✅ 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

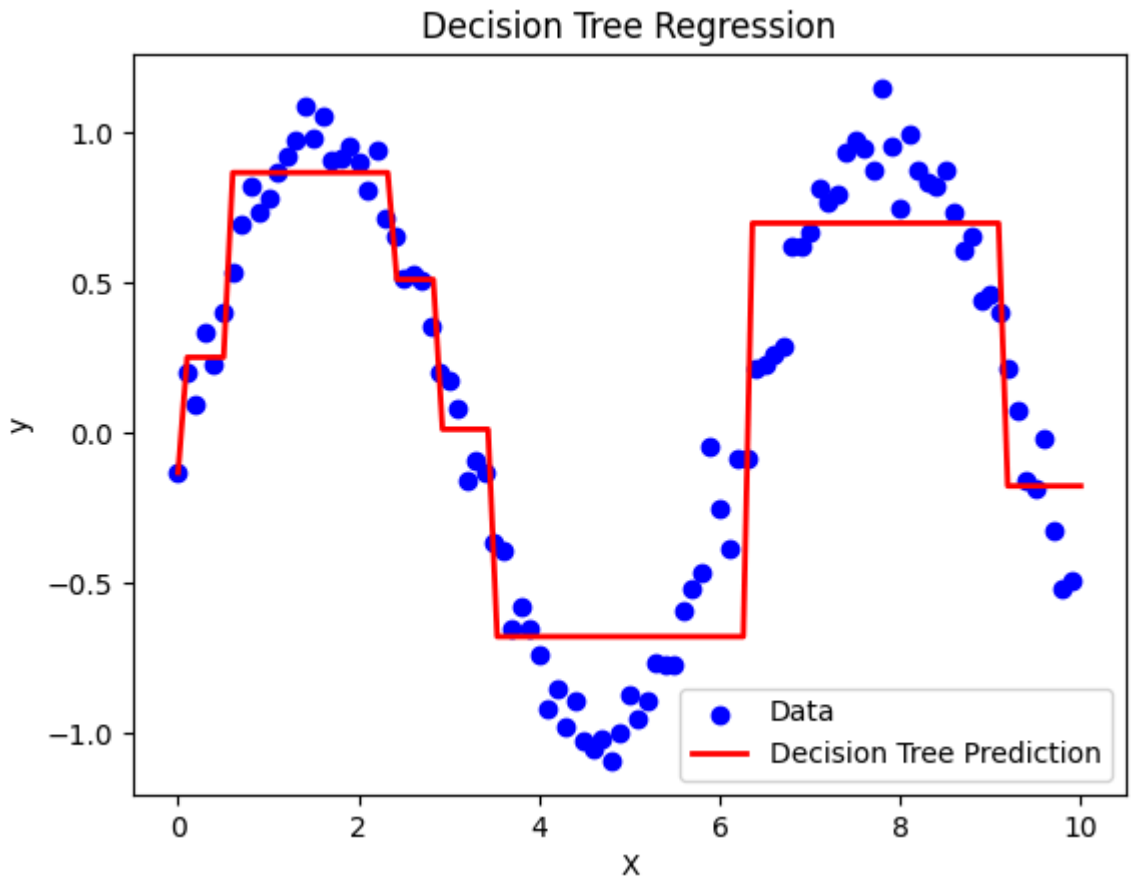
✅ Handles **non-linear** and complex relationships.
✅ Easy to visualize and interpret.
✅ No need for feature scaling.
✅ Can automatically capture feature interactions.

🔹 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).

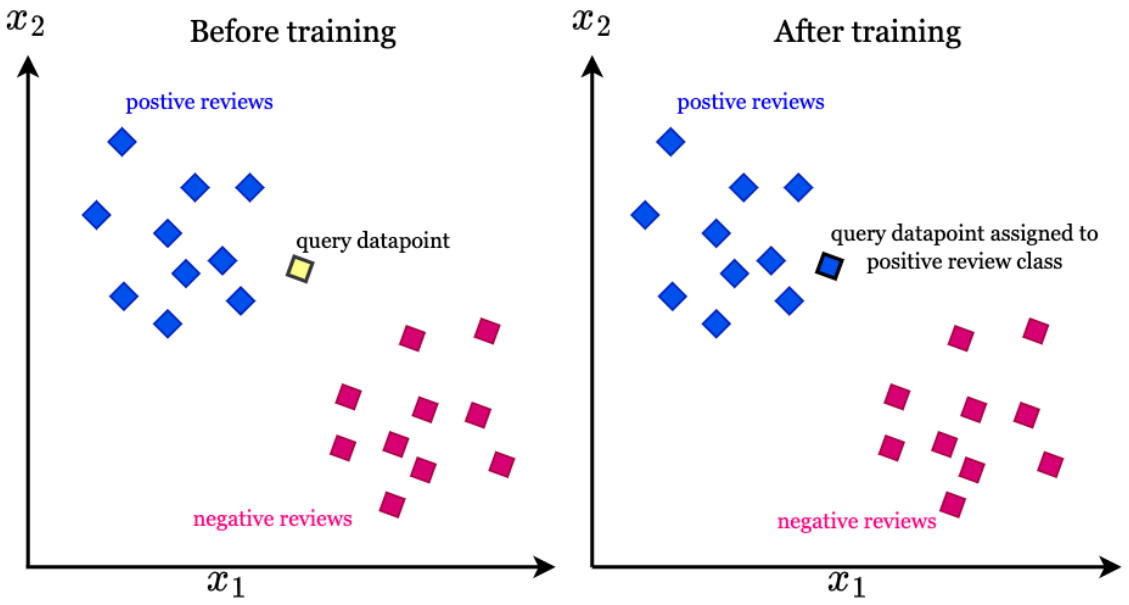
🔹 K-Nearest Neighbors (KNN)

📌 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

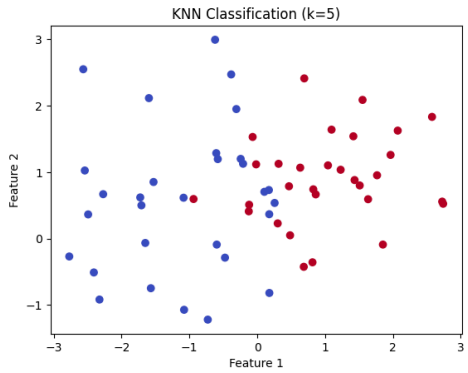
❌ Computationally expensive for large datasets (needs to compute distance for all points).
❌ Sensitive to irrelevant features → needs **feature scaling (Normalization/Standardization)**.
❌ Choosing optimal k is tricky.

🐍 Python Code – Classification & Regression

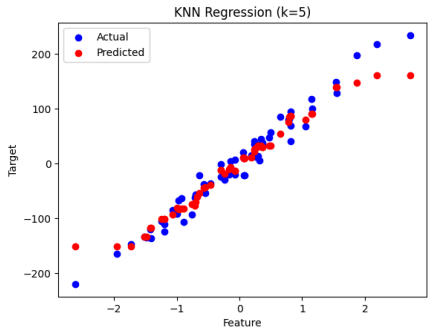
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification, make_regression
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.metrics import accuracy_score, mean_squared_error

# ======================
# 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



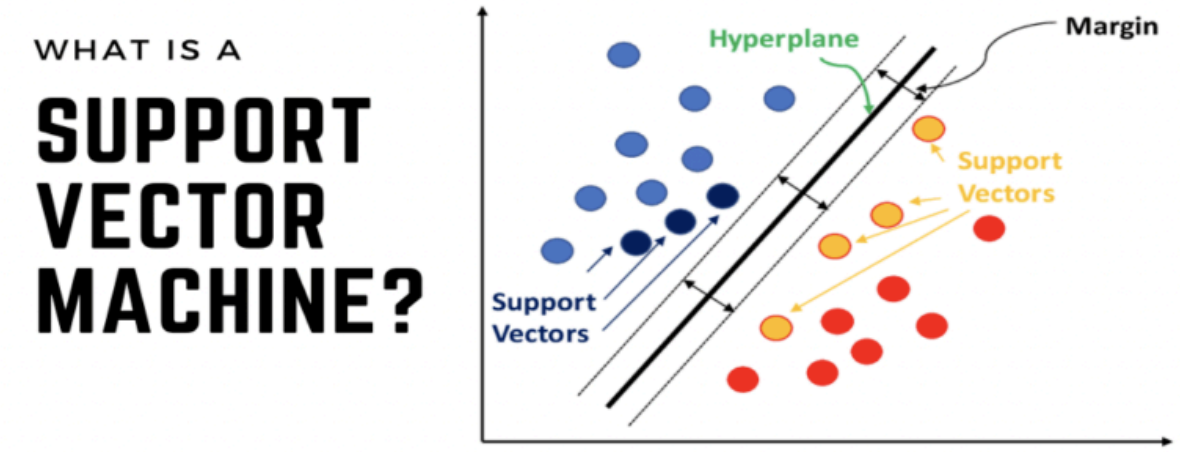
📊 Graphical Intuition

- **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.

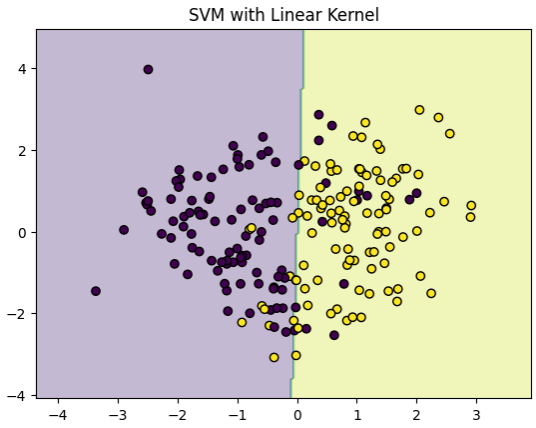
🔹 6. Python Examples

📌 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)
```

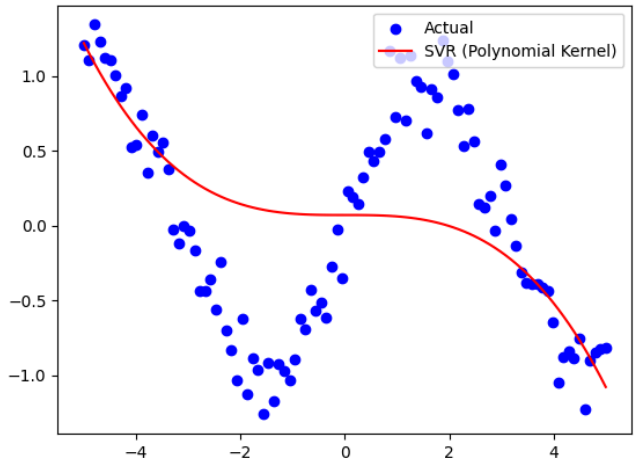
```
X, y = datasets.make_classification(
    n_samples=200,
    n_features=2,
    n_classes=2,
    n_informative=2,   # important fix
    n_redundant=0,
    random_state=42
)

# 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

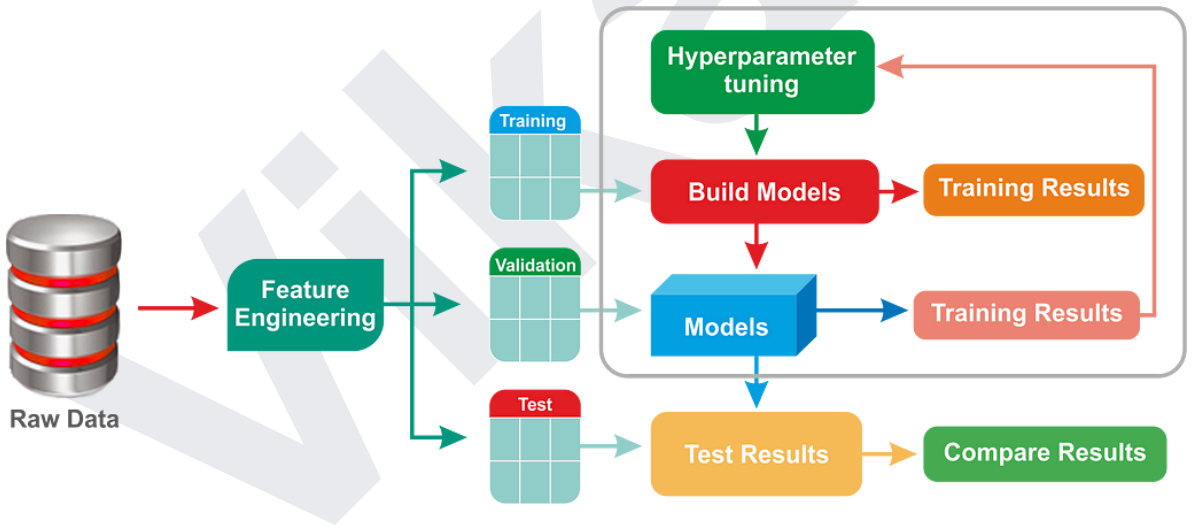
- **SVM Classification** → Finds best hyperplane with max margin.
- **SVM Regression (SVR)** → Fits within ε-tube, robust to outliers.
- **Kernels** → Enable non-linear classification/regression.
- **Polynomial SVM** → Captures curved, complex patterns.

⚡ **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

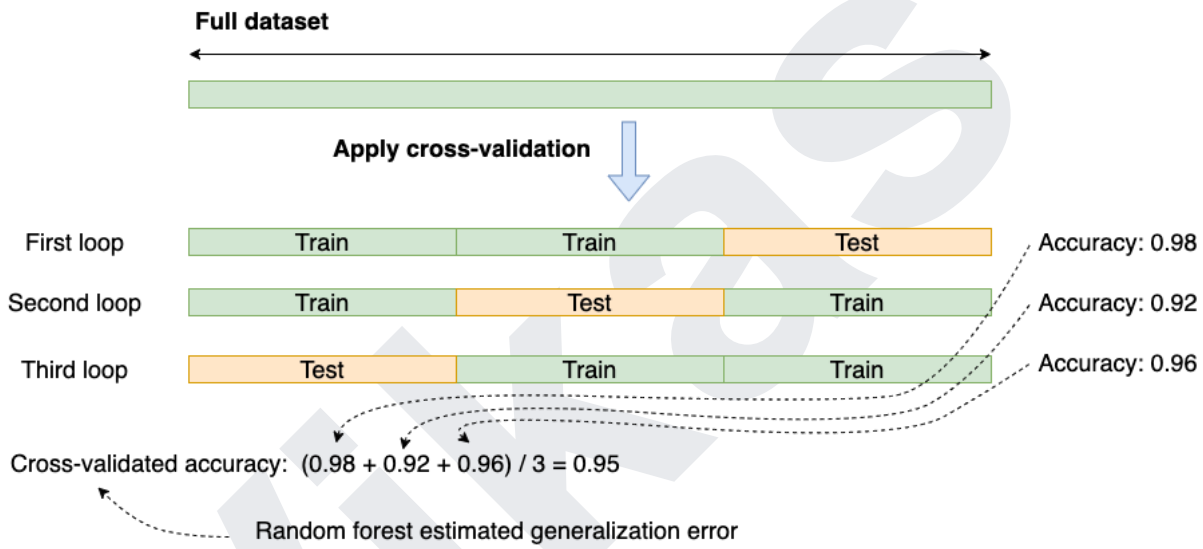
🔹 6. Tips

- 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

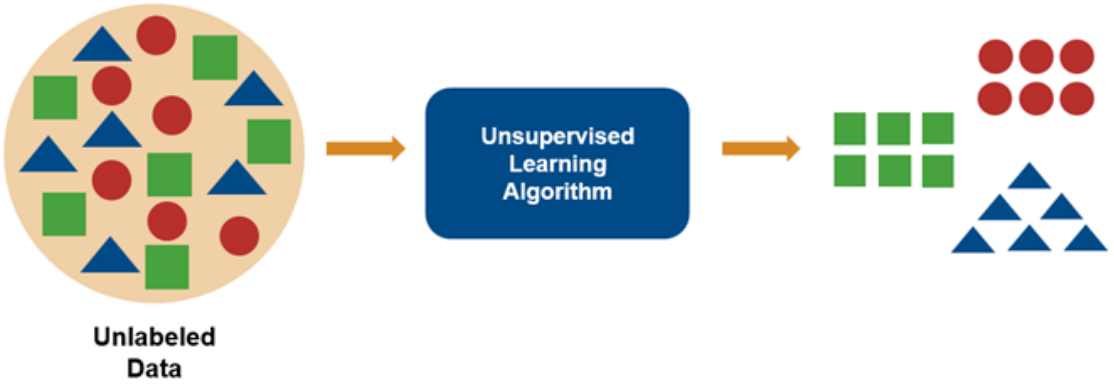
- Typical k=5 or k=10.

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

Unsupervised Machine Learning

What is Unsupervised Learning?

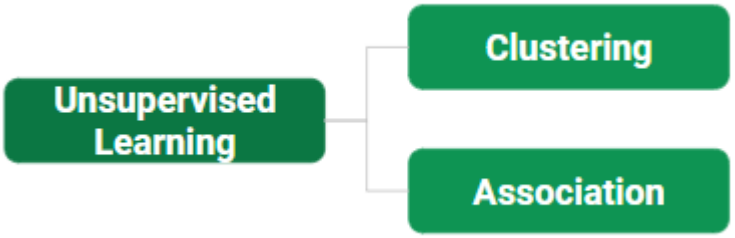
- Unsupervised Learning is a type of **Machine Learning** where the model is trained on **unlabeled data**.
- The system tries to find **patterns, structures, or groupings** without predefined output.
- It is mainly used for **exploratory data analysis, clustering, dimensionality reduction, and anomaly detection**.



Key Characteristics

- Works with **unlabeled data**
- Helps in **discovering hidden patterns**
- Often used as a **preprocessing step** for supervised learning
- Less accurate than supervised learning (since no labels for guidance)

Types of Unsupervised Learning



1. Clustering

- Groups data points into clusters based on similarity.
- Common algorithms:
    - **K-Means Clustering**
    - **Hierarchical Clustering**
    - **DBSCAN (Density-Based Clustering)**

📊 Example: Segmenting customers into groups based on buying behavior.

---

2. Association Rule Learning

- Finds relationships between variables in large datasets.
- Common algorithms:
    - **Apriori Algorithm**
    - **Eclat Algorithm**

📊 Example: Market Basket Analysis (e.g., people who buy bread often buy butter).

🔹 Applications of Unsupervised Learning

✅ Customer Segmentation
✅ Market Basket Analysis
✅ Anomaly & Fraud Detection

✅ Dimensionality Reduction for Visualization
✅ Recommender Systems

✨ **In short:**
Unsupervised ML = **No labels, just patterns** 🔎
Types = **Clustering, Association, Dimensionality Reduction, Anomaly Detection**

📘 K-Means Clustering

🔹 What is K-Means?

- **K-Means** is one of the most popular **unsupervised learning algorithms** used for **clustering**.
- It divides a dataset into **K clusters (groups)** where each data point belongs to the nearest cluster center (centroid).

🔹 How K-Means Works (Steps)

1. Choose the number of clusters **K**.
2. Randomly initialize **K centroids**.
3. Assign each data point to the **nearest centroid**.
4. Recalculate centroids by taking the **mean** of points in each cluster.
5. Repeat steps 3–4 until centroids don't change much (convergence).

🔹 Key Points

- Distance metric used: **Euclidean Distance**
- Requires user to specify **K (number of clusters)**
- Sensitive to outliers and initial centroid placement
- Works well for spherical-shaped clusters

🔹 Applications

✅ Customer Segmentation
✅ Market Basket Analysis
✅ Image Compression
✅ Document Clustering

📘 K-Means Clustering

🔹 What is K-Means?

- **K-Means** is one of the most popular **unsupervised learning algorithms** used for **clustering**.
- It divides a dataset into **K clusters (groups)** where each data point belongs to the nearest cluster center (centroid).

🔹 How K-Means Works (Steps)

1. Choose the number of clusters **K**.
2. Randomly initialize **K centroids**.
3. Assign each data point to the **nearest centroid**.
4. Recalculate centroids by taking the **mean** of points in each cluster.
5. Repeat steps 3–4 until centroids don't change much (convergence).

🔹 Key Points

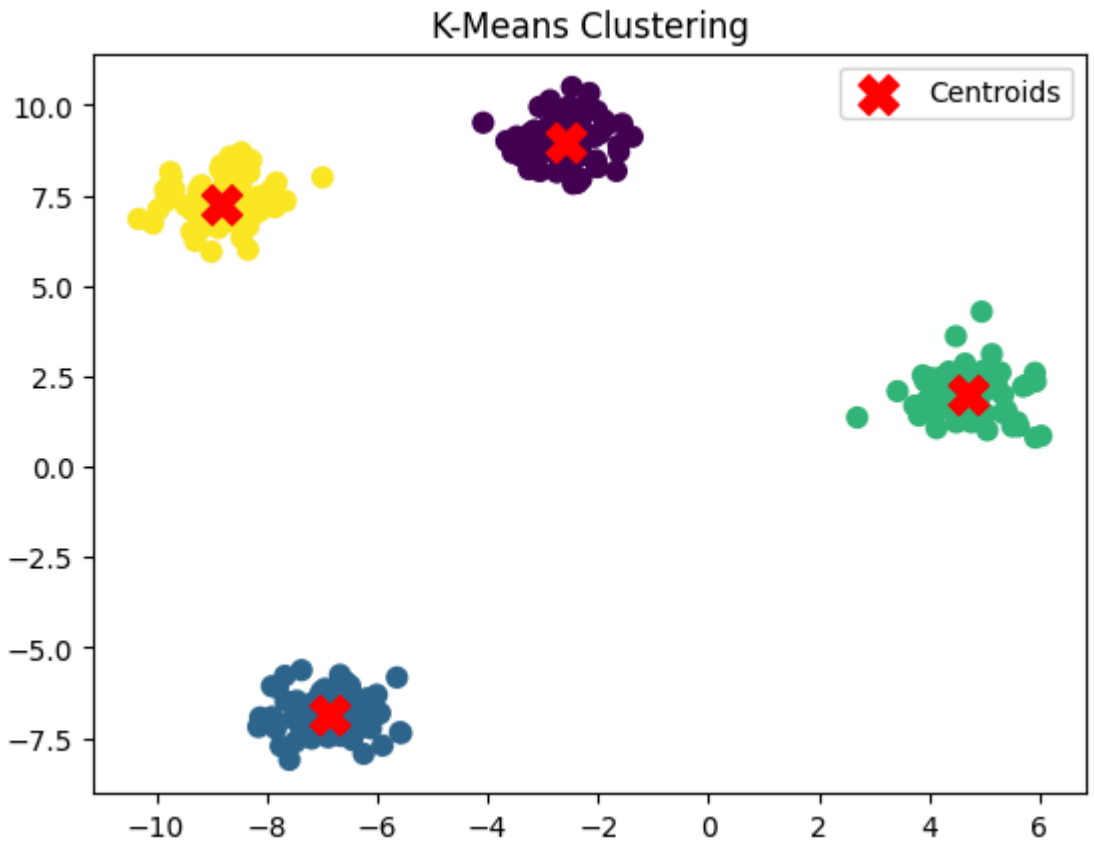
- Distance metric used: **Euclidean Distance**
- Requires user to specify **K (number of clusters)**
- Sensitive to outliers and initial centroid placement
- Works well for spherical-shaped clusters

🔹 Applications

✅ Customer Segmentation
✅ Market Basket Analysis
✅ Image Compression
✅ Document Clustering

🔹 Python Code Example

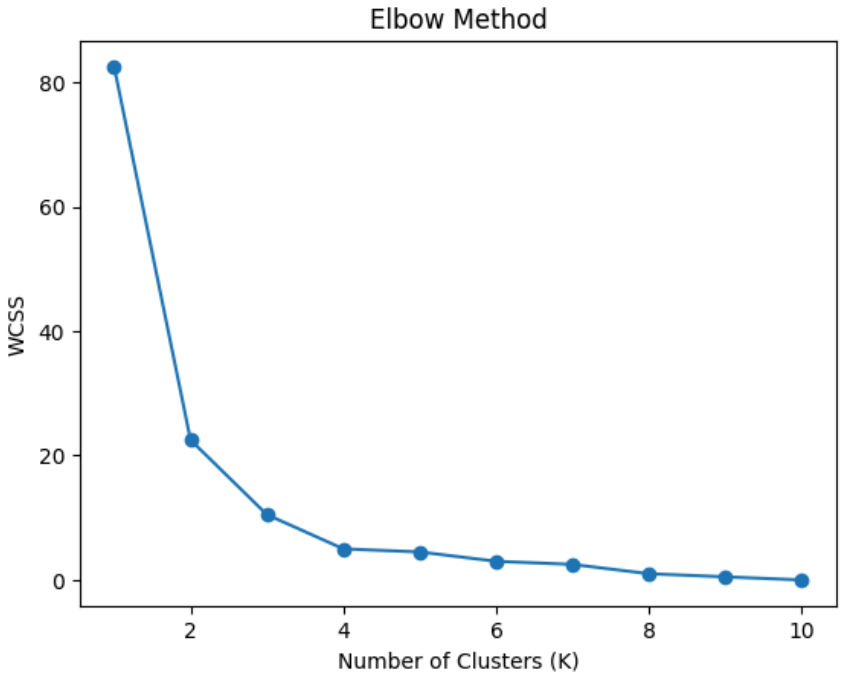
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Generate synthetic dataset
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.6, random_state=42)

# Apply KMeans
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Plot clusters
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis', s=50)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            s=200, c='red', marker='X', label="Centroids")
plt.title("K-Means Clustering")
plt.legend()
plt.show()
```



🔹 Elbow Method (Finding Best K)

- To choose the best **K**, plot **Within-Cluster Sum of Squares (WCSS)** vs. **K**.
- The “elbow” point in the curve shows the optimal number of clusters.

```
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss, marker='o')
plt.title("Elbow Method")
plt.xlabel("Number of Clusters (K)")
plt.ylabel("WCSS")
plt.show()
```



📘 Hierarchical Clustering

🔹 What is Hierarchical Clustering?

- **Unsupervised ML algorithm** for **clustering**.
- Unlike K-Means, it does **not require you to predefine the number of clusters**.
- Builds a hierarchy (tree-like structure) of clusters.
- Can be **Agglomerative (bottom-up)** or **Divisive (top-down)**.

🔹 Types of Hierarchical Clustering

1. **Agglomerative Clustering (Most Common)**
    - Start with each point as its own cluster.
    - Iteratively merge the two closest clusters.
    - Continue until all points are in one big cluster.
2. **Divisive Clustering**
    - Start with one big cluster.
    - Iteratively split clusters into smaller ones.
    - Continue until each point is its own cluster.

🔹 Key Concepts

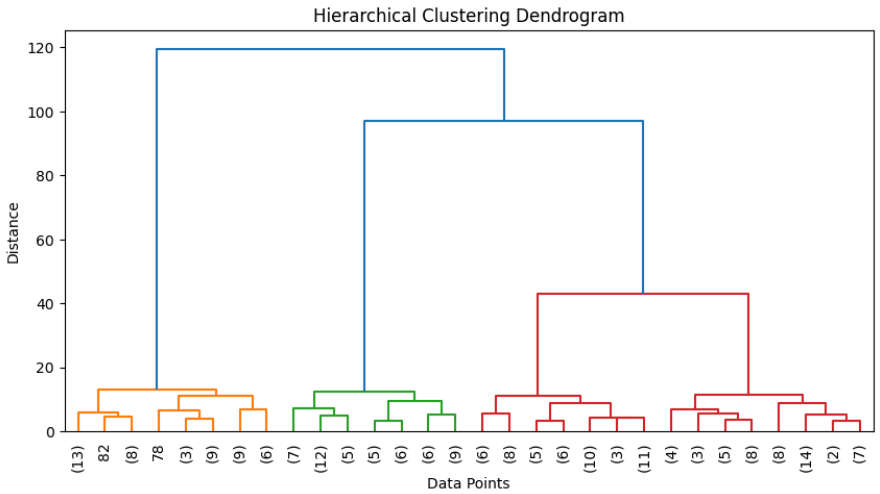
- **Linkage Criteria** → how distance between clusters is calculated:
    - **Single Linkage** → Minimum distance between points of two clusters.

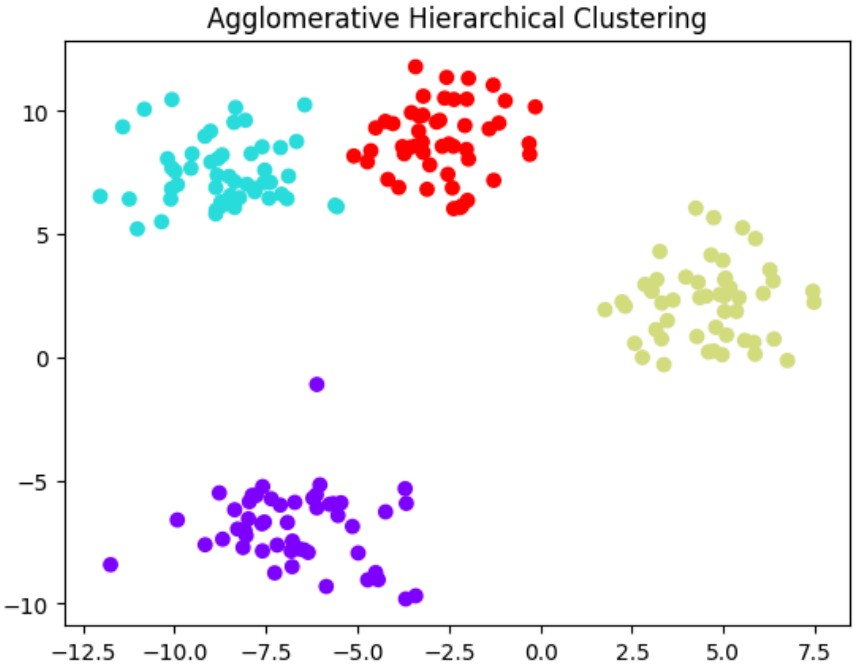
- **Complete Linkage** → Maximum distance between points of two clusters.
- **Average Linkage** → Average distance between points of two clusters.
- **Ward's Method** → Minimizes variance within clusters.

- **Dendrogram** → A tree-like diagram that shows merging/splitting of clusters.

🔹 Python Code Example

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

# Generate dataset
X, y = make_blobs(n_samples=200, centers=4, random_state=42, cluster_std=1.5)

# ----------------------
# 1. Plot Dendrogram
# ----------------------
linked = linkage(X, method='ward')  # Ward's linkage
plt.figure(figsize=(10, 5))
dendrogram(linked, truncate_mode='lastp', p=30, leaf_rotation=90., leaf_font_size=10.)
plt.title("Hierarchical Clustering Dendrogram")
plt.xlabel("Data Points")
plt.ylabel("Distance")
plt.show()

# ----------------------
# 2. Apply Agglomerative Clustering
# ----------------------
hc = AgglomerativeClustering(n_clusters=4, metric='euclidean', linkage='ward')
y_hc = hc.fit_predict(X)

# Plot Clusters
plt.scatter(X[:, 0], X[:, 1], c=y_hc, cmap='rainbow')
plt.title("Agglomerative Hierarchical Clustering")
plt.show()
```





🔹 Pros & Cons

✅ Advantages

- No need to specify K (can cut dendrogram later).
- Gives full hierarchy (better visualization).

❌ **Disadvantages**

- Computationally expensive for large datasets.
- Once merged/split, cannot be undone.

✨ **In short:**

- Hierarchical clustering builds a **tree of clusters**.
- Dendrogram helps **decide number of clusters** by cutting at a chosen distance.

📊 Silhouette Score in Clustering

🔹 What is Silhouette Score?

Silhouette Score is a metric used to **evaluate the quality of clusters** formed in unsupervised learning algorithms like **K-Means, Hierarchical, DBSCAN**, etc.

It tells how well each data point lies within its cluster compared to other clusters.

- The score ranges between **-1 and +1**:
    - **+1** → Perfectly matched to its own cluster & far from others (good clustering).
    - **0** → On or very close to the boundary between two clusters.
    - **-1** → Wrongly assigned to the cluster (bad clustering).

🔹 Formula

For a given data point iii:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

- **a(i):** Average distance of point iii from other points in the **same cluster**.
- **b(i):** Minimum average distance of point iii from all points in **other clusters**.

🔹 Interpretation

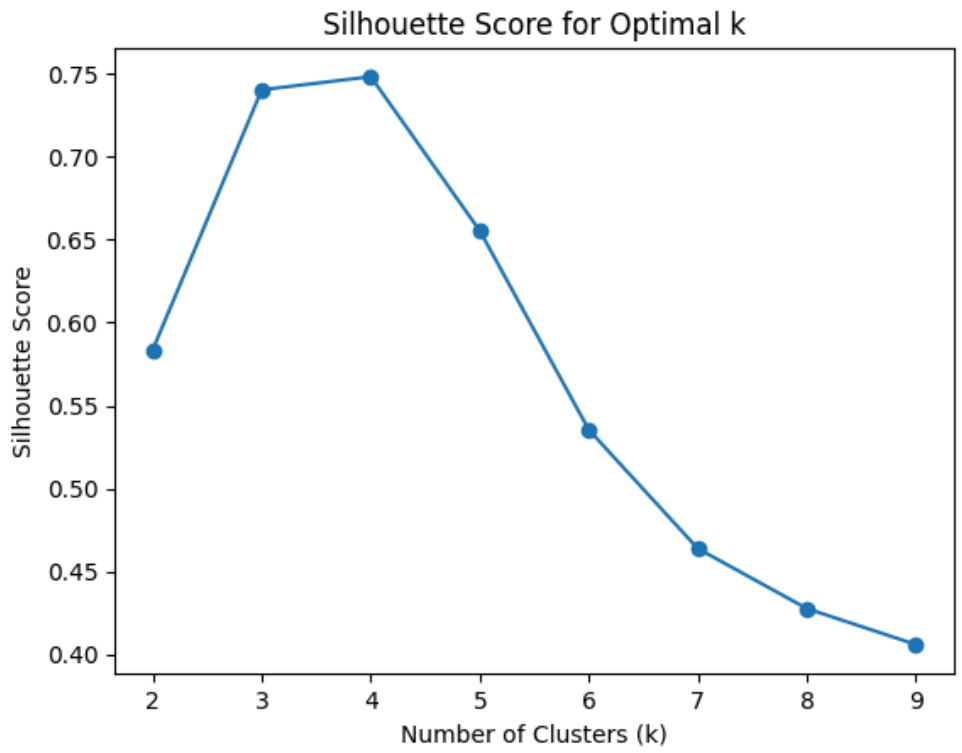
- **0.71 – 1.00** → Strong structure (excellent clustering)
- **0.51 – 0.70** → Reasonable structure

- **0.26 – 0.50** → Weak structure, might need improvement
- **< 0.25** → No substantial cluster structure

🔹 Python Example (with K-Means)

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Generate dataset
X, y = make_blobs(n_samples=500, centers=4, cluster_std=1.2, random_state=42)

# Try different k values
scores = []
for k in range(2, 10):
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(X)
    score = silhouette_score(X, labels)
    scores.append(score)
    print(f"k={k}, Silhouette Score={score:.3f}")

# Plot silhouette scores
plt.plot(range(2, 10), scores, marker='o')
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Silhouette Score")
plt.title("Silhouette Score for Optimal k")
plt.show()
```

```
k=2, Silhouette Score=0.584
k=3, Silhouette Score=0.740
k=4, Silhouette Score=0.748
k=5, Silhouette Score=0.655
k=6, Silhouette Score=0.535
k=7, Silhouette Score=0.464
k=8, Silhouette Score=0.428
k=9, Silhouette Score=0.406
```



🔹 Applications

- Helps determine the **optimal number of clusters**.
- Evaluates clustering performance in **K-Means, Hierarchical, DBSCAN, GMM** etc.
- Useful for **model selection** in unsupervised learning.

🛒 Association Learning – Apriori Algorithm

🔹 What is Association Learning?

Association Learning is an **unsupervised machine learning technique** used to discover **hidden relationships or patterns** between items in large datasets.
It is mostly used in **Market Basket Analysis** (e.g., "Customers who buy bread also buy butter").

🔹 Apriori Algorithm

Apriori is the most popular algorithm for **Association Rule Mining**.

It is based on the **Apriori Property**:
👉 *"If an itemset is frequent, then all its subsets must also be frequent."*

🔹 Key Terms

1. **Itemset** → A collection of one or more items.
   Example: {Milk, Bread}

2. **Support** → How often an itemset appears in the dataset.
   Support(A)=Transactions containing ATotal Transactions\text{Support(A)} = \frac{\text{Transactions containing A}}{\text{Total Transactions}}Support(A)=Total TransactionsTransactions containing A
3. **Confidence** → How often item B is purchased when item A is purchased.
   Confidence(A→B)=Support(A ∪ B)Support(A)\text{Confidence(A→B)} = \frac{\text{Support(A ∪ B)}}{\text{Support(A)}}Confidence(A→B)=Support(A)Support(A ∪ B)
4. **Lift** → How much more likely A and B occur together compared to being independent.
   Lift(A→B)=Confidence(A→B)Support(B)\text{Lift(A→B)} = \frac{\text{Confidence(A→B)}}{\text{Support(B)}}Lift(A→B)=Support(B)Confidence(A→B)
   - Lift > 1 → Positive association

- Lift = 1 → No relation
- Lift < 1 → Negative association

🔹 Steps of Apriori Algorithm

1. Set minimum **support** and **confidence** thresholds.
2. Generate **frequent itemsets** (those meeting support threshold).
3. Generate **strong rules** from frequent itemsets using confidence and lift.

🔹 Python Example – Apriori

```
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules

# Example dataset (Market Basket)
dataset = [
    ['Milk', 'Bread', 'Eggs'],
    ['Milk', 'Bread'],
    ['Milk', 'Eggs'],
    ['Bread', 'Eggs'],
    ['Milk', 'Bread', 'Butter'],
    ['Bread', 'Butter']
]

# Convert into DataFrame
from mlxtend.preprocessing import TransactionEncoder
te = TransactionEncoder()
te_data = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_data, columns=te.columns_)

# Apply Apriori
frequent_itemsets = apriori(df, min_support=0.3, use_colnames=True)
print(frequent_itemsets)
```

```python
# Generate Association Rules
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
print(rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])
```

```
     support         itemsets
0   0.833333          (Bread)
1   0.333333         (Butter)
2   0.500000           (Eggs)
3   0.666667           (Milk)
4   0.333333  (Butter, Bread)
5   0.333333    (Bread, Eggs)
6   0.500000    (Bread, Milk)
7   0.333333     (Eggs, Milk)
  antecedents consequents   support  confidence  lift
0    (Butter)     (Bread)  0.333333    1.000000   1.2
1     (Bread)    (Butter)  0.333333    0.400000   1.2
2      (Eggs)      (Milk)  0.333333    0.666667   1.0
3      (Milk)      (Eggs)  0.333333    0.500000   1.0
```

🔹 Applications

- 🛒 **Market Basket Analysis** (Retail, E-commerce)
- 🎬 **Movie Recommendation Systems**
- 🧬 **Bioinformatics** (gene associations)
- 📈 **Cross-selling & Upselling Strategies**

🌳 FP-Growth Algorithm (Frequent Pattern Growth)

🔹 What is FP-Growth?

FP-Growth is an **association rule mining algorithm** used to find frequent itemsets in large datasets.
It is considered **faster and more efficient** than Apriori because it **avoids generating candidate itemsets** repeatedly.

Instead, FP-Growth uses a **compact data structure** called the **FP-Tree (Frequent Pattern Tree)**.

🔹 How it Works?

1. **Build FP-Tree**
    - Scan dataset once to count item frequencies.
    - Discard infrequent items (below min support).
    - Construct FP-Tree (items arranged in descending frequency).
2. **Mine Frequent Patterns**
    - Traverse FP-Tree to extract frequent itemsets.
    - Use recursive "conditional FP-Tree" to find longer patterns.

🔹 Advantages over Apriori

✅ Much faster for large datasets.
✅ Requires fewer database scans (Apriori scans multiple times).
✅ More memory efficient.

🔹 Key Terms

- **Support**: Fraction of transactions containing an itemset.
- **Confidence**: Likelihood of consequent given antecedent.
- **Lift**: Strength of rule compared to random chance.

🔹 Python Example – FP-Growth

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import fpgrowth, association_rules

# Example dataset (Market Basket)
dataset = [
    ['Milk', 'Bread', 'Eggs'],
    ['Milk', 'Bread'],
    ['Milk', 'Eggs'],
    ['Bread', 'Eggs'],
    ['Milk', 'Bread', 'Butter'],
    ['Bread', 'Butter']
]

# Convert into DataFrame
te = TransactionEncoder()
te_data = te.fit(dataset).transform(dataset)
```

```
df = pd.DataFrame(te_data, columns=te.columns_)



# Apply FP-Growth

frequent_itemsets = fpgrowth(df, min_support=0.3, use_colnames=True)

print(frequent_itemsets)



# Generate Association Rules

rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)

print(rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])
```

```
    support         itemsets
0  0.833333          (Bread)
1  0.666667           (Milk)
2  0.500000           (Eggs)
3  0.333333         (Butter)
4  0.500000    (Bread, Milk)
5  0.333333     (Eggs, Milk)
6  0.333333    (Bread, Eggs)
7  0.333333  (Butter, Bread)
  antecedents consequents   support  confidence  lift
0      (Eggs)      (Milk)  0.333333    0.666667   1.0
1      (Milk)      (Eggs)  0.333333    0.500000   1.0
2    (Butter)     (Bread)  0.333333    1.000000   1.2
3     (Bread)    (Butter)  0.333333    0.400000   1.2
```

🔹 Applications

- 🛒 **Retail & E-commerce** – Market Basket Analysis
- 🎵 **Music/Movie Recommendation** – Suggest related content
- 🧬 **Healthcare** – Find disease-symptom relations

- 📊 **Finance** – Customer spending patterns

⚡ **Comparison with Apriori**

- Apriori → Candidate generation + pruning (slow for big data).
- FP-Growth → FP-Tree mining (faster & memory efficient).

🌟 Ensemble Learning – Voting

🔹 What is Ensemble Learning?

Ensemble Learning is a technique where **multiple models (weak learners/base models) are combined** to improve overall performance.
Instead of relying on a single model, ensemble methods aggregate predictions → **better accuracy, stability, and robustness**.

🔹 Voting Classifier (for Classification)

- Combines predictions of multiple classifiers.
- Types of Voting:
    1. **Hard Voting** → takes **majority class label** (like majority vote).
    2. **Soft Voting** → averages **predicted probabilities** and selects the class with the highest probability.

✅ Works best when base models are diverse (e.g., Logistic Regression, Decision Tree, SVM, etc.)

🔹 Voting Regressor (for Regression)

- Combines predictions of multiple regression models.
- Final prediction = **average of all base model outputs**.

✅ Helps reduce variance and improve generalization.

🔹 Python Example – Voting Classifier

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score

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

# Base models
clf1 = LogisticRegression(max_iter=200)
clf2 = DecisionTreeClassifier(max_depth=4)
clf3 = KNeighborsClassifier(n_neighbors=5)

# Voting Classifier (Hard Voting)
voting_clf = VotingClassifier(estimators=[
    ('lr', clf1), ('dt', clf2), ('knn', clf3)], voting='hard')

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

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

Accuracy: 1.0

🔹 Python Example – Voting Regressor

```
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor
from sklearn.metrics import mean_squared_error

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

```python
# Base models
reg1 = LinearRegression()
reg2 = DecisionTreeRegressor(max_depth=5)

# Voting Regressor
voting_reg = VotingRegressor(estimators=[('lr', reg1), ('dt', reg2)])
voting_reg.fit(X_train, y_train)

y_pred = voting_reg.predict(X_test)
print("MSE:", mean_squared_error(y_test, y_pred))
```

MSE: 2941.5119690786482

🔹 Key Points

- Voting is a **simple yet powerful** ensemble method.
- Works better when base models are **diverse** and not highly correlated.
- Useful as a **baseline ensemble** before trying advanced methods (Bagging, Boosting, Stacking).

🌟 Bagging (Bootstrap Aggregating)

🔹 What is Bagging?

- **Bagging** = **Bootstrap Aggregating**.
- It is an **ensemble method** that trains multiple models on different random subsets of the dataset (with replacement = bootstrap sampling).
- Final prediction = **majority vote (classification)** or **average (regression)**.

👉 Goal: Reduce **variance** and avoid **overfitting**.

🔹 How Bagging Works

1. Take **random bootstrap samples** (same size as dataset but sampled with replacement).
2. Train a **base model** on each sample.
3. Aggregate predictions:
    - Classification → Majority Voting
    - Regression → Averaging

🔹 Example: Bagging Classifier

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

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

```
# Base model: Decision Tree
base_clf = DecisionTreeClassifier()

# Bagging Classifier
bag_clf = BaggingClassifier(
    estimator=base_clf,
    n_estimators=50,       # number of trees
    max_samples=0.8,       # 80% data per tree
    bootstrap=True,        # with replacement
    random_state=42
)

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

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

Bagging Accuracy: 1.0

🔹 Example: Bagging Regressor

```
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor
from sklearn.metrics import mean_squared_error

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

# Base model: Decision Tree
base_reg = DecisionTreeRegressor()

# Bagging Regressor
bag_reg = BaggingRegressor(
    estimator=base_reg,
    n_estimators=50,
```

```
    max_samples=0.8,
    bootstrap=True,
    random_state=42
)

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

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

Bagging MSE: 2831.034279699248

🔹 Key Points

- Bagging **reduces variance** → prevents overfitting.
- Works well with **unstable models** (like Decision Trees).
- Famous Example → **Random Forest = Bagging of Decision Trees**.
- Best when dataset has **high variance**.

All Code + Projects On Github