Machine Learning

Course Notes

Prepared by Abhijit Gatade
iGAP Technologies Pvt Ltd

Contents

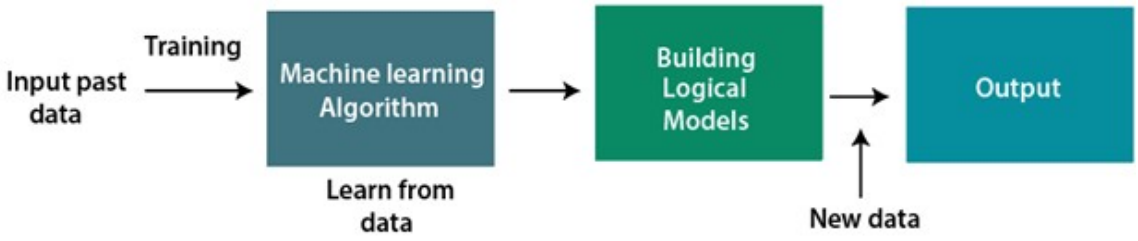
Module 1: Introduction to Machine Learning

What is Machine Learning?

Machine Learning is said as a subset of **artificial intelligence** that is mainly concerned with the development of algorithms which allow a computer to learn from the data and past experiences on their own. The term machine learning was first introduced by **Arthur Samuel** in **1959**.

How does Machine Learning work

A Machine Learning system **learns from historical data, builds the prediction models, and whenever it receives new data, predicts the output for it**. The accuracy of predicted output depends upon the amount of data, as the huge amount of data helps to build a better model which predicts the output more accurately.

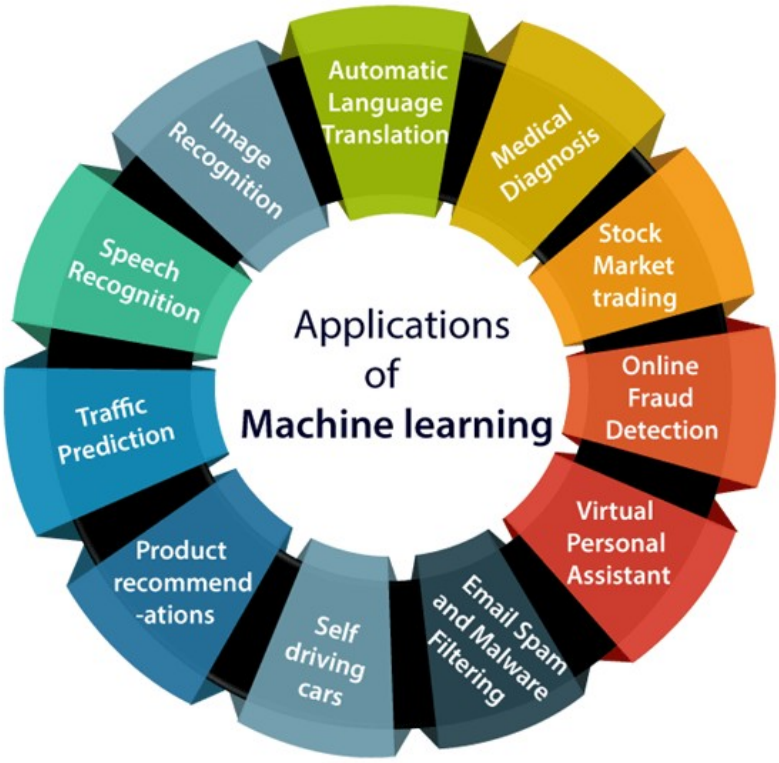


Features of Machine Learning:

- Machine learning uses data to detect various patterns in a given dataset.
- It can learn from past data and improve automatically.
- It is a data-driven technology.
- Machine learning is much similar to data mining as it also deals with the huge amount of the data.

Applications of ML

Machine learning is a buzzword for today's technology, and it is growing very rapidly day by day. We are using machine learning in our daily life even without knowing it such as Google Maps, Google assistant, Alexa, etc. Below are some most trending real-world applications of Machine Learning:



Types of ML Algorithms

Machine Learning algorithms can be broadly classified into three main types:

1. Supervised Learning

- **Definition**: The model learns from a labeled dataset (input and output are provided).
- **Goal**: Predict outputs for new, unseen data.
- **Examples**:
    - **Regression**: Predicting continuous values
        - Simple Linear Regression
        - Multiple Linear Regression
        - Decision Tree Regression
        - Random Forest Regression
        - Support Vector Regression (SVR)
    - **Classification**: Predicting categories or classes
        - Logistic Regression

- K-Nearest Neighbors (kNN)
- Decision Tree Classifier
- Random Forest Classifier
- Support Vector Machine (SVM)
- Naive Bayes
- AdaBoost
- Gradient Boosting, XGBoost

2. Unsupervised Learning

- **Definition**: The model works with **unlabeled data** (only inputs, no outputs).
- **Goal**: Find hidden patterns or groupings in data.
- **Examples**:
  - **Clustering**:
    - K-Means Clustering
    - Hierarchical Clustering
    - DBSCAN
  - **Dimensionality Reduction**:
    - Principal Component Analysis (PCA)
    - t-SNE
    - LDA

3. Reinforcement Learning (Optional for beginners)

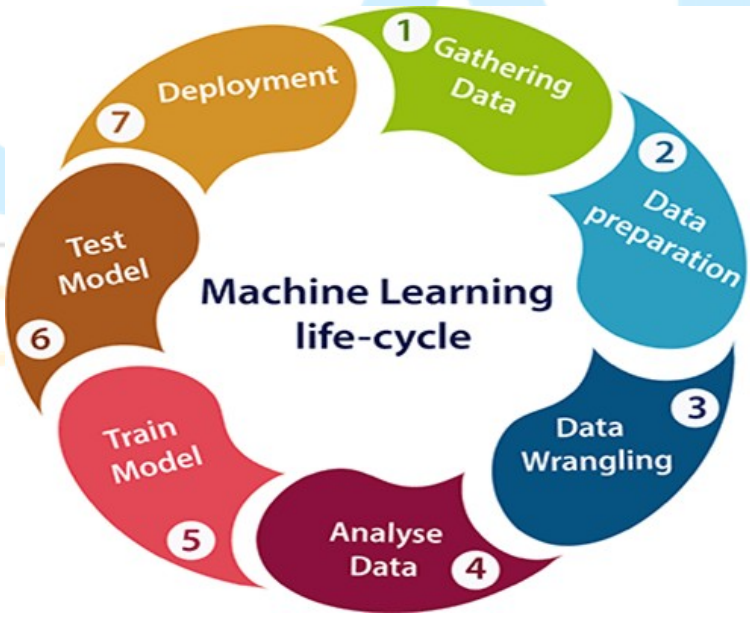
- **Definition**: The model learns by interacting with an environment and receiving **rewards or penalties**.
- **Goal**: Learn a policy for decision-making over time.
- **Examples**:
  - Q-Learning
  - Deep Q-Networks (DQN)

Machine learning Life cycle

Machine learning has given the computer systems the abilities to automatically learn without being explicitly programmed. But how does a machine learning system work? So, it can be described using the life cycle of machine learning. Machine learning life cycle is a cyclic process to build an efficient machine learning project. The main purpose of the life cycle is to find a solution to the problem or project.

Machine learning life cycle involves seven major steps, which are given below:

- **Gathering Data**
- **Data preparation**
- **Data Wrangling**
- **Analyze Data**
- **Train the model**
- **Test the model**
- **Deployment**



Differences between AI, ML, DL, DE, DS and DA

Field	Definition	Goal	Techniques/Tools	Example Use Cases

Artificial Intelligence (AI)	A broader concept of machines being able to carry out tasks in a "smart" way.	Simulate human intelligence in machines.	Expert systems, NLP, robotics, planning.	Chatbots, self-driving cars, fraud detection.
Machine Learning (ML)	A subset of AI that involves training algorithms to learn from data.	Enable systems to learn and improve from experience.	Regression, classification, clustering, reinforcement.	Email spam filters, recommendation systems.
Deep Learning (DL)	A subset of ML using neural networks with multiple layers.	Learn complex patterns using big data and high compute.	CNN, RNN, GANs, Transformers.	Facial recognition, language translation, image tagging.
Data Engineering	Focuses on designing and building data pipelines and systems.	Collect, transform, and store data efficiently.	SQL, Spark, Airflow, ETL tools.	Building data lakes/warehouses, managing big data flow.
Data Science	Combines programming, statistics, and domain knowledge to extract insights.	Solve complex problems with data-driven decisions.	Python, R, ML, statistics, data viz.	Predictive modeling, churn prediction, A/B testing.
Data Analysis	Inspecting, cleaning, and modeling data to discover useful information.	Understand past trends and current patterns.	Excel, SQL, Python (Pandas), Power BI, Tableau.	Sales reports, performance dashboards, KPI monitoring.

Summary:

- **AI** is the umbrella.
- **ML** is a branch of AI.
- **DL** is a deeper branch of ML.
- **Data Engineering** makes data ready and accessible.

- **Data Science** builds models and interprets data for decision-making.
- **Data Analysis** focuses on summarizing the past and present insights from data.

Important libraries for ML

- Numpy
- Pandas
- Matplotlib
- Seaborn
- Scikit-Learn

Module 2: Numpy

Introduction

- NumPy is a Python library used for working with arrays.
- NumPy is short for "Numerical Python".
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

Why Use NumPy?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

Why is NumPy Faster Than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behaviour is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also, it is optimized to work with latest CPU architectures.

Installation of NumPy

pip install numpy

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```

NumPy as np

NumPy is usually imported under the np alias.

Checking NumPy Version

```
import numpy as np
print(np.__version__)
```

NumPy Creating Arrays

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

Use a tuple to create a NumPy array:

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

MultiDimensional Array

0-D Arrays

```
arr = np.array(42)
```

1-D Arrays

```
arr = np.array([1, 2, 3, 4, 5])
```

2-D Arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
```

3-D arrays

```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
```

Check Number of Dimensions?

```
print(a.ndim)
```

Access Array Elements

- Array indexing is the same as accessing an array element.
- You can access an array element by referring to its index number.
- The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Access 2-D Arrays

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

Negative Indexing

Use negative indexing to access an array from the end.

NumPy Array Slicing

- Slicing in python means taking elements from one given index to another given index.
- We pass slice instead of index like this: [*start*:*end*].
- We can also define the step, like this: [*start*:*end*:*step*].
- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension
- If we don't pass step its considered 1

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Negative Slicing

Use the minus operator to refer to an index from the end:

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

STEP

Use the step value to determine the step of the slicing:

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

NumPy Data Types

By default Python have these data types:

- strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- integer - used to represent integer numbers. e.g. -1, -2, -3
- float - used to represent real numbers. e.g. 1.2, 42.42
- boolean - used to represent True or False.
- complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- i - integer
- b - boolean
- u - unsigned integer
- f - float

- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type (void)

Checking the Data Type of an Array

```
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

NumPy Array Shape

The shape of an array is the number of elements in each dimension.

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

NumPy Array Reshaping

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

NumPy Array Iterating

- Iterating means going through elements one by one.
- As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.
- If we iterate on a 1-D array it will go through each element one by one.

```
arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

NumPy Joining Array

- Joining means putting contents of two or more arrays in a single array.

- In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

- We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.concatenate((arr1, arr2))
print(arr)
```

NumPy Splitting Array

- Splitting is reverse operation of Joining.
- Joining merges multiple arrays into one and Splitting breaks one array into multiple.
- We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.

```
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
```

NumPy Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.
To search an array, use the where() method.

```
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

NumPy Sorting Arrays

- Sorting means putting elements in an ordered sequence.

- Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

- The NumPy ndarray object has a function called sort(), that will sort a specified array.

```
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

NumPy Filter Array

- Getting some elements out of an existing array and creating a new array out of them is called filtering.

- In NumPy, you filter an array using a boolean index list.
- A boolean index list is a list of booleans corresponding to indexes in the array.

```
arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]
newarr = arr[x]
print(newarr)
```

The example above will return [41, 43], why?
Because the new array contains only the values where the filter array had the value True, in this case, index 0 and 2.

Creating the Filter Array

In the example above we hard-coded the True and False values, but the common use is to create a filter array based on conditions.

```
arr = np.array([41, 42, 43, 44])
# Create an empty list
filter_arr = []

# go through each element in arr
for element in arr:
 # if the element is higher than 42, set the value to True, otherwise False:
 if element > 42:
   filter_arr.append(True)
 else:
   filter_arr.append(False)

newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

Module 3: Pandas

What is Pandas?

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
-

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Installation of Pandas

pip install pandas

```
import pandas
mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}
myvar = pandas.DataFrame(mydataset)
print(myvar)
```

Pandas as pd

Pandas is usually imported under the pd alias.

Checking Pandas Version

```
import pandas as pd
print(pd.__version__)
```

Pandas Series

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)
```

Pandas DataFrames

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

```
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
```

Locate Row

As you can see from the result above, the DataFrame is like a table with rows and columns.
Pandas use the loc attribute to return one or more specified row(s)

```
print(df.loc[0])
```

Pandas Read CSV

- A simple way to store big data sets is to use CSV files (comma separated files).
- CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

```
import pandas as pd
df = pd.read_csv('data.csv')
```

```
print(df.to_string())
```

Pandas Read JSON

Big data sets are often stored, or extracted as JSON.

JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.

```
import pandas as pd
df = pd.read_json('data.json')
print(df.to_string())
```

Pandas - Analyzing DataFrames

One of the most used method for getting a quick overview of the DataFrame, is the head() method.

The head() method returns the headers and a specified number of rows, starting from the top.

```
print(df.head())
```

The tail() method returns the headers and a specified number of rows, starting from the bottom.

```
print(df.tail())
```

The DataFrames object has a method called info(), that gives you more information about the data set.

```
print(df.info())
```

```
print(df.describe())
```

Pandas - Cleaning Data

Data cleaning means fixing bad data in your data set.

Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

The data set contains some empty cells ("Date" in row 22, and "Calories" in row 18 and 28).

The data set contains wrong format ("Date" in row 26).

The data set contains wrong data ("Duration" in row 7).

The data set contains duplicates (row 11 and 12).

1. Empty Cells

Empty cells can potentially give you a wrong result when you analyze data.

Remove Rows

One way to deal with empty cells is to remove rows that contain empty cells.

This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.

```
df = pd.read_csv('data.csv')
new_df = df.dropna()
print(new_df.to_string())
```

By default, the dropna() method returns a new DataFrame, and will not change the original.

If you want to change the original DataFrame, use the inplace = True argument:

```
df = pd.read_csv('data.csv')
df.dropna(inplace = True)
print(df.to_string())
```

Replace Empty Values

Another way of dealing with empty cells is to insert a new value instead.
This way you do not have to delete entire rows just because of some empty cells.
The fillna() method allows us to replace empty cells with a value:

```
df = pd.read_csv('data.csv')
df.fillna(130, inplace = True)
```

Replace Only For Specified Columns

The example above replaces all empty cells in the whole Data Frame.

To only replace empty values for one column, specify the column name for the DataFrame:

```
df = pd.read_csv('data.csv')
df["Calories"].fillna(130, inplace = True)
```

Replace Using Mean, Median, or Mode

A common way to replace empty cells, is to calculate the mean, median or mode value of the column.

Pandas uses the mean() median() and mode() methods to calculate the respective values for a specified column:

```
df = pd.read_csv('data.csv')
x = df["Calories"].mean()
df["Calories"].fillna(x, inplace = True)

df = pd.read_csv('data.csv')
x = df["Calories"].median()
df["Calories"].fillna(x, inplace = True)

df = pd.read_csv('data.csv')
x = df["Calories"].mode()[0]
df["Calories"].fillna(x, inplace = True)
```

2. Data of Wrong Format

Cells with data of wrong format can make it difficult, or even impossible, to analyze data.

To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.

Convert Into a Correct Format

In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:

Let's try to convert all cells in the 'Date' column into dates.

Pandas has a to_datetime() method for this:

```
df = pd.read_csv('data.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

Removing Rows

The result from the converting in the example above gave us a NaT value, which can be handled as a NULL value, and we can remove the row by using the dropna() method.

```
df.dropna(subset=['Date'], inplace = True)
```

3. Wrong Data

"Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".

Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.

If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.

It doesn't have to be wrong, but taking in consideration that this is the data set of someone's workout sessions, we conclude with the fact that this person did not work out in 450 minutes.

Replacing Values

One way to fix wrong values is to replace them with something else.

In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:

```
df.loc[7, 'Duration'] = 45
```

For small data sets you might be able to replace the wrong data one by one, but not for big data sets.

To replace wrong data for larger data sets you can create some rules, e.g. set some boundaries for legal values, and replace any values that are outside of the boundaries.

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.loc[x, "Duration"] = 120
```

Removing Rows

Another way of handling wrong data is to remove the rows that contains wrong data.

This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.drop(x, inplace = True)
```

4. Removing Duplicates

Discovering Duplicates

Duplicate rows are rows that have been registered more than one time.

```
print(df.duplicated())
```

Removing Duplicates

To remove duplicates, use the drop_duplicates() method.

```
df.drop_duplicates(inplace = True)
```

Pandas - Data Correlations

Finding Relationships
A great aspect of the Pandas module is the corr() method.

The corr() method calculates the relationship between each column in your data set.

```
df.corr()
```

The Result of the corr() method is a table with a lot of numbers that represents how well the relationship is between two columns.

The number varies from -1 to 1.

1 means that there is a 1 to 1 relationship (a perfect correlation), and for this data set, each time a value went up in the first column, the other one went up as well.

0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.

-0.9 would be just as good relationship as 0.9, but if you increase one value, the other will probably go down.

0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

What is a good correlation? It depends on the use, but I think it is safe to say you have to have at least 0.6 (or -0.6) to call it a good correlation.

Perfect Correlation:
We can see that "Duration" and "Duration" got the number 1.000000, which makes sense, each column always has a perfect relationship with itself.

Good Correlation:
"Duration" and "Calories" got a 0.922721 correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

Bad Correlation:
"Duration" and "Maxpulse" got a 0.009403 correlation, which is a very bad correlation, meaning that we can not predict the max pulse by just looking at the duration of the work out, and vice versa.

Module 4: EDA(Exploratory Data Analysis) using Matplotlib and Seaborn Libraries

Introduction

Exploratory Data Analysis (EDA) is an important first step in data science projects. It involves looking at and visualizing data to understand its main features, find patterns, and discover how different parts of the data are connected.

We can present findings in table or visualization format.

3 types of EDA

1. **Univariate** – studying one variable to understand its characteristics like mean, median, mode, variance and standard deviation.
2. **Bivariate** – studying relationship between two variables to find connections, correlations, and dependencies.
3. **Multivariate** - studying relationships between two or more variables in the dataset. It aims to understand how variables interact with one another, which is crucial for most statistical modelling techniques.

To plot findings in chart/graph format we can use Matplotlib or Seaborn library.

Standard graphs and charts

Here's a **comprehensive list of standard graphs and charts** you can plot using **Matplotlib** and **Seaborn**, along with **example Python code** for each.

- **1. Line Plot**

Use: Trend over time or continuous data
Library: Matplotlib, Seaborn

```
import matplotlib.pyplot as plt

import seaborn as sns


x = [1, 2, 3, 4, 5]

y = [10, 12, 8, 15, 20]

plt.plot(x, y)

plt.title("Line Plot")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.show()
```

- **2. Bar Plot**

Use: Comparing categories
Library: Seaborn, Matplotlib

```
import seaborn as sns

import pandas as pd


data = pd.DataFrame({'Category': ['A', 'B', 'C'], 'Values': [10, 20, 15]})

sns.barplot(x='Category', y='Values', data=data)

plt.title("Bar Plot")

plt.show()
```

- **3. Count Plot**

Use: Count of categorical values
Library: Seaborn

```
sns.countplot(x='Category', data=data)

plt.title("Count Plot")

plt.show()
```

- **4. Histogram**

Use: Distribution of numerical data
Library: Matplotlib, Seaborn

```
import numpy as np


data = np.random.randn(100)

plt.hist(data, bins=10)

plt.title("Histogram")

plt.show()
```

```
# Or using seaborn

sns.histplot(data, bins=10, kde=True)

plt.title("Histogram with KDE")

plt.show()
```

- **5. KDE Plot (Kernel Density Estimate)**

Use: Smoothed version of histogram
Library: Seaborn

```
sns.kdeplot(data, shade=True)

plt.title("KDE Plot")

plt.show()
```

- **6. Box Plot**

Use: Distribution + outliers
Library: Seaborn

```
sns.boxplot(y=data)

plt.title("Box Plot")

plt.show()
```

- **7. Violin Plot**

Use: Box plot + KDE
Library: Seaborn

```
sns.violinplot(y=data)

plt.title("Violin Plot")

plt.show()
```

- **8. Scatter Plot**

Use: Relationship between two variables
Library: Matplotlib, Seaborn

```
x = [1, 2, 3, 4, 5]

y = [5, 4, 7, 10, 8]

plt.scatter(x, y)

plt.title("Scatter Plot")

plt.show()


# Seaborn way

sns.scatterplot(x=x, y=y)

plt.title("Scatter Plot")

plt.show()
```

- **9. Pair Plot**

Use: Relationship between all features in a DataFrame
Library: Seaborn

```
df = sns.load_dataset('iris')

sns.pairplot(df, hue='species')

plt.show()
```

- **10. Heatmap**

Use: Correlation or matrix visualizations
Library: Seaborn

```
corr = df.corr()

sns.heatmap(corr, annot=True, cmap='coolwarm')

plt.title("Correlation Heatmap")
```

```
plt.show()
```

- **11. Pie Chart**

Use: Part-to-whole relationships
Library: Matplotlib

```
labels = ['A', 'B', 'C']

sizes = [30, 50, 20]

plt.pie(sizes, labels=labels, autopct='%1.1f%%')

plt.title("Pie Chart")

plt.show()
```

- **12. Swarm Plot**

Use: Categorical scatter plot
Library: Seaborn

```
sns.swarmplot(x='species', y='sepal_length', data=df)

plt.title("Swarm Plot")

plt.show()
```

- **13. Strip Plot**

Use: Jittered scatter plot by category
Library: Seaborn

```
sns.stripplot(x='species', y='sepal_length', data=df, jitter=True)

plt.title("Strip Plot")

plt.show()
```

- **14. Joint Plot**

Use: Combined scatter + histogram
Library: Seaborn

```
sns.jointplot(x='sepal_length', y='petal_length', data=df, kind='scatter')

plt.show()
```

🔹 15. Rug Plot

Use: Small tick marks for univariate data
Library: Seaborn

```
sns.rugplot(data)

plt.title("Rug Plot")

plt.show()
```

Essential code snippets for EDA

Sure! Here's a **collection of essential EDA (Exploratory Data Analysis) code snippets** you can use when analyzing a DataFrame in Python using **Pandas, Seaborn, and Matplotlib**.

🔍 1. Basic Info About DataFrame

```
import pandas as pd

# Load dataset

df = pd.read_csv('your_file.csv')


# Shape of the dataset

print("Shape:", df.shape)


# Column names

print("Columns:", df.columns.tolist())


# Data types
```

```
print(df.dtypes)


# Summary of data

print(df.info())


# First few rows

print(df.head())


# Last few rows

print(df.tail())


# Summary statistics

print(df.describe())
```

🖌️ **2. Missing Value Analysis**

```
# Check for missing values

print(df.isnull().sum())


# Visualize missing data

import seaborn as sns

import matplotlib.pyplot as plt


sns.heatmap(df.isnull(), cbar=False, cmap='viridis')

plt.title("Missing Value Heatmap")

plt.show()
```

🔁 3. Duplicate Records

```
# Find duplicate rows

duplicates = df[df.duplicated()]

print(duplicates)


# Remove duplicates

df = df.drop_duplicates()
```

📊 4. Value Counts for Categorical Columns

```
# Value counts of a categorical column

print(df['column_name'].value_counts())


# Plot it

df['column_name'].value_counts().plot(kind='bar')

plt.title("Value Counts")

plt.show()
```

📈 5. Correlation Between Features

```
# Correlation matrix

corr_matrix = df.corr()


# Heatmap of correlations

sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')

plt.title("Correlation Heatmap")

plt.show()
```

🖇️ 6. Unique Values

```
# Unique values of a column

print(df['column_name'].unique())


# Number of unique values

print(df['column_name'].nunique())
```

📩 7. Outlier Detection (Boxplot)

```
sns.boxplot(x=df['numerical_column'])

plt.title("Boxplot for Outliers")

plt.show()
```

🗓️ 8. Convert Date Columns

```
# Convert to datetime

df['date_column'] = pd.to_datetime(df['date_column'])


# Extract components

df['year'] = df['date_column'].dt.year

df['month'] = df['date_column'].dt.month

df['day'] = df['date_column'].dt.day
```

📊 9. Distribution Plot

```
sns.histplot(df['numerical_column'], kde=True, bins=30)

plt.title("Distribution Plot")

plt.show()
```

🍼 10. Groupby Analysis

```
# Group by category and get mean

print(df.groupby('category_column')['numerical_column'].mean())


# Group by two categories

print(df.groupby(['cat1', 'cat2'])['value'].agg(['mean', 'count']))
```

📌 BONUS: EDA Helper Function

```
def basic_eda(df):
    print("Shape:", df.shape)
    print("\nInfo:")
    print(df.info())
    print("\nMissing values:\n", df.isnull().sum())
    print("\nSummary stats:\n", df.describe())
    print("\nColumns:\n", df.columns)
```

Module 5: Data Preprocessing in Machine Learning

Data preprocessing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model.

When creating a machine learning project, it is not always a case that we come across the clean and formatted data. And while doing any operation with data, it is mandatory to clean it and put in a formatted way. So for this, we use data preprocessing task.

Why do we need Data Preprocessing?

A real-world data generally contains noises, missing values, and maybe in an unusable format which cannot be directly used for machine learning models. Data preprocessing is required tasks for cleaning the data and making it suitable for a machine learning model which also increases the accuracy and efficiency of a machine learning model.

It involves below steps:

- **Getting the dataset**
- **Importing libraries**
- **Importing datasets**
- **Finding Missing Data**
- **Encoding Categorical Data**
- **Splitting dataset into training and test set**
- **Feature scaling**

1. Get the Dataset

To create a machine learning model, the first thing we required is a dataset as a machine learning model completely works on data. The collected data for a particular problem in a proper format is known as the **dataset**.

Dataset may be of different formats for different purposes, such as, if we want to create a machine learning model for business purpose, then dataset will be different with the dataset required for a liver patient. So each dataset is different from another dataset. To use the dataset in our code, we usually put it into a CSV **file**. However, sometimes, we may also need to use an HTML or xlsx file.

2. Importing Libraries

In order to perform data preprocessing using Python, we need to import some predefined Python libraries. These libraries are used to perform some specific jobs.

3. Importing the Datasets

Now we need to import the datasets which we have collected for our machine learning project.

Extracting dependent and independent variables

In machine learning, it is important to distinguish the matrix of features (independent variables) and dependent variables from dataset. In our dataset, there are three independent variables that are **Country, Age**, and **Salary**, and one is a dependent variable which is **Purchased**.

4. Handling Missing data:

The next step of data preprocessing is to handle missing data in the datasets. If our dataset contains some missing data, then it may create a huge problem for our machine learning model. Hence it is necessary to handle missing values present in the dataset.

5. Encoding Categorical data:

Categorical data is data which has some categories such as, in our dataset; there are two categorical variable, **Country**, and **Purchased**.

Since machine learning model completely works on mathematics and numbers, but if our dataset would have a categorical variable, then it may create trouble while building the model. So it is necessary to encode these categorical variables into numbers.

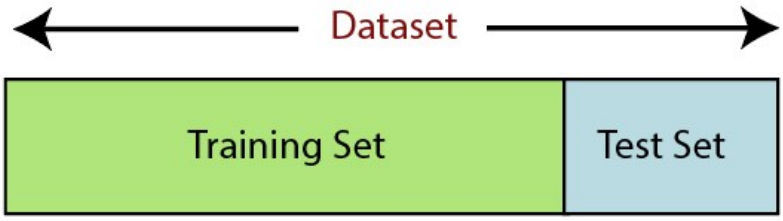
- Label Encoding
- OneHotEncoding
- Ordinal Encoding

6. Splitting the Dataset into the Training set and Test set

In machine learning data preprocessing, we divide our dataset into a training set and test set. This is one of the crucial steps of data preprocessing as by doing this, we can enhance the performance of our machine learning model.

Suppose, if we have given training to our machine learning model by a dataset and we test it by a completely different dataset. Then, it will create difficulties for our model to understand the correlations between the models.

If we train our model very well and its training accuracy is also very high, but we provide a new dataset to it, then it will decrease the performance. So we always try to make a machine learning model which performs well with the training set and also with the test dataset. Here, we can define these datasets as:



Training Set: A subset of dataset to train the machine learning model, and we already know the output.

Test set: A subset of dataset to test the machine learning model, and by using the test set, model predicts the output.

7. Feature Scaling

Feature scaling is the final step of data preprocessing in machine learning. It is a technique to standardize the independent variables of the dataset in a specific range. In feature scaling, we put our variables in the same range and in the same scale so that no any variable dominates the other variable.

- StandardScaler
- MinMaxScaler
- RobustScaler

Module 6: Advanced Data Preparation

1. Outlier Detection

What are Outliers?

- Outliers are data points that are **significantly different** from other observations.
- They can **skew the results** of machine learning models and affect model performance.

Why detect and treat outliers?

- They might be:
    - **Errors in data entry**
    - **Unusual but valid observations** (e.g., very high income)
- They can **distort statistical calculations** like mean, standard deviation, etc.

Methods:

a. IQR Method

- Based on **quartiles** (Q1 and Q3)

- Captures the **middle 50%** of the data.
- Anything below Q1 - 1.5 * IQR or above Q3 + 1.5 * IQR is considered an outlier.

b. Z-Score Method

- Measures how far a data point is from the **mean**, in units of **standard deviation**.
- Typically, values with **|z| > 3** are considered outliers.

```
import pandas as pd

import numpy as np

from scipy import stats

# Sample data

df = pd.DataFrame({'income': [30000, 35000, 40000, 1000000, 38000, 37000]})

# IQR Method

Q1 = df['income'].quantile(0.25)

Q3 = df['income'].quantile(0.75)

IQR = Q3 - Q1

outliers_iqr = df[(df['income'] < Q1 - 1.5*IQR) | (df['income'] > Q3 + 1.5*IQR)]

# Z-score Method

z_scores = stats.zscore(df['income'])

outliers_z = df[np.abs(z_scores) > 3]
```

2. Feature Engineering

What is Feature Engineering?

- It is the process of **creating new input features** from raw data to improve model performance.

Why is it important?

- Helps in uncovering **hidden patterns**
- Converts raw data into **meaningful inputs** for ML models

Examples:

- Creating new columns from dates (year, month, weekday)

- Combining two columns (e.g., total_spent = food + rent)
- Extracting useful information from names, email addresses, etc.

```
# Date feature engineering

df = pd.DataFrame({'purchase_date': pd.to_datetime(['2023-01-15', '2023-03-10'])})

df['year'] = df['purchase_date'].dt.year

df['month'] = df['purchase_date'].dt.month

df['day_of_week'] = df['purchase_date'].dt.dayofweek

# Numeric transformation

df['log_income'] = np.log(df['income'] + 1)  # log transform
```

3. Feature Extraction from Text & Dates

Text Feature Extraction

- Used in **NLP tasks** and recommendation systems.
- Useful metrics:
  - Text length
  - Number of words
  - Word frequency
  - TF-IDF scores

Date Feature Extraction

- Dates are usually not used as raw values.
- We extract:
  - Year
  - Month
  - Day of week
  - Time difference between dates
- These help identify **seasonal trends**, **weekly patterns**, etc.

```
# Text feature extraction
```

```
df = pd.DataFrame({'review': ['Good product', 'Worst experience ever']})

df['word_count'] = df['review'].apply(lambda x: len(x.split()))

df['text_length'] = df['review'].apply(len)

# Date difference

df['today'] = pd.to_datetime('2025-04-10')

df['days_since_purchase'] = (df['today'] - df['purchase_date']).dt.days
```

4. Data Balancing Techniques

Why is Data Balancing important?

- In real-world classification problems, datasets are often **imbalanced**.
    - E.g., 95% 'No fraud' and 5% 'Fraud'
- Models trained on imbalanced data tend to **favor the majority class**, leading to poor performance.

SMOTE (Synthetic Minority Over-sampling Technique)

- Generates **synthetic examples** of the minority class.
- Helps avoid **overfitting** which can happen with simple duplication.

Random Over/Under Sampling

- **Oversampling**: Adds copies of the minority class.
- **Undersampling**: Removes instances from the majority class.

Always apply balancing **after splitting** into train and test to avoid data leakage.

```
from sklearn.datasets import make_classification

from imblearn.over_sampling import SMOTE

from collections import Counter

# Create imbalanced dataset

X, y = make_classification(n_samples=1000, n_features=5, weights=[0.9, 0.1], random_state=42)

print("Before:", Counter(y))

# Apply SMOTE
```

```
smote = SMOTE(random_state=42)

X_res, y_res = smote.fit_resample(X, y)

print("After:", Counter(y_res))
```

Summary Insight

Concept	Why It Matters
Outlier Detection	Prevents skewing of model predictions
Feature Engineering	Boosts model performance by creating richer inputs
Text/Date Extraction	Helps turn unstructured/temporal data into usable form
SMOTE and Sampling	Helps models learn equally from both classes in imbalanced datasets

Module 7: Regression Algorithms

Regression is used when the target variable is **continuous** (e.g., price, salary, temperature).

1. Simple Linear Regression

Introduction

- **Definition**: Predicts the value of a dependent variable (Y) based on one independent variable (X) using a straight line.
- **Equation**: Y = b0 + b1*X
    - b0 is the intercept
    - b1 is the slope of the line
- **Objective**: Minimize the difference between actual and predicted Y values (i.e., minimize Mean Squared Error).

Use Cases

- Predicting salary based on experience
- Predicting height from age

Assumptions

- Linear relationship between X and Y
- Residuals are normally distributed

- Homoscedasticity (constant variance of residuals)
- Independence of residuals

Code Example

```
from sklearn.linear_model import LinearRegression

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error, r2_score

import pandas as pd

import matplotlib.pyplot as plt

# Sample Data

X = df[['YearsExperience']]

y = df['Salary']

# Train Test Split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Model Training

model = LinearRegression()

model.fit(X_train, y_train)

# Prediction

y_pred = model.predict(X_test)

# Evaluation

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

print("R2 Score:", r2_score(y_test, y_pred))
```

2. Multiple Linear Regression

Theory

- **Definition**: Extension of Simple Linear Regression with multiple independent variables.
- **Equation**: Y = b0 + b1*X1 + b2*X2 + ... + bn*Xn

Use Cases

- Predicting house price based on size, location, age

- Predicting student performance based on study hours, attendance, IQ

Code Example

```
X = df[['Size', 'Bedrooms', 'Age']]

y = df['Price']

model = LinearRegression()

model.fit(X, y)

y_pred = model.predict(X)
```

3. Polynomial Regression

Theory

- **Definition**: A regression model where the relationship between X and Y is modeled as an nth degree polynomial.
- **Equation**: Y = b0 + b1*X + b2*X^2 + ... + bn*X^n
- Helps to model non-linear relationships.

Code Example

```
from sklearn.preprocessing import PolynomialFeatures

from sklearn.pipeline import make_pipeline

poly_model = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())

poly_model.fit(X_train, y_train)

poly_pred = poly_model.predict(X_test)
```

4. Decision Tree Regression

Theory

- **Definition**: Splits the dataset into branches at decision nodes based on feature values and predicts by averaging the outputs in a leaf.
- **Non-linear**, **non-parametric** model.

Use Cases

- Predicting prices based on ranges or rules
- Useful for interpretable models

Code Example

```
from sklearn.tree import DecisionTreeRegressor

model = DecisionTreeRegressor(max_depth=3)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)
```

5. Random Forest Regression

Theory

- **Definition**: An ensemble of decision trees. Uses bagging technique (bootstrap + aggregation).
- Reduces overfitting, more robust.

Use Cases

- Large and complex datasets
- Predicting house price, stock prices

Code Example

```
from sklearn.ensemble import RandomForestRegressor

rf_model = RandomForestRegressor(n_estimators=100, random_state=0)

rf_model.fit(X_train, y_train)

y_pred = rf_model.predict(X_test)
```

6. Support Vector Regression (SVR)

Theory

- **Definition**: Tries to fit the best line (or curve with kernel) within a margin of tolerance (epsilon).
- Uses **kernel trick** to model non-linear relationships.

Use Cases

- Financial predictions
- High dimensional data

Code Example

```
from sklearn.svm import SVR
```

```
from sklearn.preprocessing import StandardScaler

sc_X = StandardScaler()

sc_y = StandardScaler()

X_scaled = sc_X.fit_transform(X)

y_scaled = sc_y.fit_transform(y.values.reshape(-1, 1)).ravel()

svr = SVR(kernel='rbf')

svr.fit(X_scaled, y_scaled)

y_pred = svr.predict(sc_X.transform(X_test))
```

Regression Evaluation Metrics

These metrics evaluate how accurately a regression model can predict **continuous numeric values**.

1. Mean Absolute Error (MAE)

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

- Measures the **average absolute difference** between actual and predicted values.
- Easy to understand and not sensitive to outliers.
- **Lower is better**

2. Mean Squared Error (MSE)

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

- Squares the error, so **larger errors are penalized more**.
- Sensitive to outliers.

3. Root Mean Squared Error (RMSE)

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

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

4. R^2 Score (Coefficient of Determination)

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

- Indicates **how much variance** in the dependent variable is explained by the model.
- Ranges between **0 and 1**.
    - **1** = perfect fit
    - **0** = model explains nothing
    - Can also be **negative** (worse than a horizontal line model)

5. Adjusted R^2 Score

$$\text{Adjusted } R^2 = 1 - (1 - R^2)\left(\frac{n-1}{n-p-1}\right)$$

- Adjusts R^2 for the number of predictors used.
- Useful for **Multiple Linear Regression**.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

import numpy as np

# y_test = actual values, y_pred = predicted values

mae = mean_absolute_error(y_test, y_pred)

mse = mean_squared_error(y_test, y_pred)

rmse = np.sqrt(mse)

r2 = r2_score(y_test, y_pred)

print("Mean Absolute Error (MAE):", mae)

print("Mean Squared Error (MSE):", mse)

print("Root Mean Squared Error (RMSE):", rmse)

print("R² Score:", r2)

# For adjusted R² (custom implementation)

n = len(y_test)
```

```
p = X_test.shape[1]

adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)

print("Adjusted R² Score:", adjusted_r2)
```

Metric	Measures	Sensitive to Outliers	Best Value
MAE	Avg. absolute error	❌ No	0
MSE	Avg. squared error	✅ Yes	0
RMSE	Square root of MSE	✅ Yes	0
R^2	Variance explained	❌ No	1
Adj. R^2	R^2 adjusted for feature count	❌ No	1

Overfitting and Underfitting in Regression

Overfitting

- The model learns both the signal **and the noise** in the training data.
- Performs **very well on training data**, but poorly on new/unseen data.
- Often happens when model is too complex (e.g., high-degree polynomial regression, too many features).
- Indicators: High training accuracy, **low test accuracy**.

Underfitting

- The model is too simple to capture the underlying trend in data.
- Performs poorly on both training and test sets.
- Often caused by insufficient model complexity or **not enough training**.

How to Identify

Case	Training Accuracy	Test Accuracy	Cause
Overfitting	High	Low	Too complex model
Underfitting	Low	Low	Too simple model

Good Fit	High	High	Balanced model

Regularization in Regression

Regularization is a technique used to **prevent overfitting** by adding a penalty term to the loss function.

L1 Regularization (Lasso Regression):

- Adds **absolute value** of coefficients as penalty: Loss + λ * |w|
- Can reduce some coefficients to **exactly zero** → automatic **feature selection**.
- Best when you suspect **some features are irrelevant**.
- Used when **sparse models** are desired.

L2 Regularization (Ridge Regression):

- Adds **squared value** of coefficients: Loss + λ * w^2
- Shrinks coefficients **but does not eliminate** them.
- Useful when **multicollinearity** (correlated features) exists.
- Maintains all features but reduces impact of less important ones.

When to Use

Situation	**Use**
Feature selection needed	Lasso (L1)
Multicollinearity or all features useful	Ridge (L2)
Want to combine both benefits	ElasticNet

Both L1 and L2 regularization help to **generalize the model** and reduce overfitting on training data.

Module 8: Classification Algorithms

Classification algorithms are used when the output variable is a **category** or **label**, such as *spam or not spam*, *malignant or benign*, *0 or 1*, etc.

1. Logistic Regression

Concept

- A linear model used for **binary classification**.
- It uses the **logistic (sigmoid)** function to map predicted values between **0 and 1**.

Working

$$\text{Sigmoid Function: } \sigma(z) = \frac{1}{1 + e^{-z}}$$

- If output ≥ 0.5 ⇒ Class 1
- If output < 0.5 ⇒ Class 0

Use Cases

- Email spam detection
- Loan default prediction
- Disease prediction (yes/no)

Pros

- Simple and easy to implement
- Probabilistic output
- Performs well with linearly separable data

Cons

- Not suitable for complex non-linear problems
- Sensitive to outliers

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
# Train model
```

```
model = LogisticRegression()

model.fit(X_train, y_train)

# Predictions

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
```

2. Decision Tree Classifier

Concept

- A tree-like model that splits data into branches based on feature values using **Information Gain** or **Gini Index**.

Use Cases

- Credit scoring
- Medical diagnosis
- Fraud detection

Pros

- Easy to interpret
- Handles both numerical and categorical data
- No need for feature scaling

Cons

- Can overfit on training data
- Sensitive to small variations

```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(criterion='gini')

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
```

3. Random Forest Classifier

Concept

- An ensemble of Decision Trees using **Bagging** (bootstrap + aggregation).
- Final result is based on **majority voting**.

Use Cases

- Image classification
- Text classification
- Banking fraud detection

Pros

- Reduces overfitting
- Handles missing data and outliers well
- High accuracy

Cons

- Slower to train
- Less interpretable than a single tree

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
```

4. K-Nearest Neighbors (kNN)

Concept

- A non-parametric method that classifies data based on the **majority vote of k nearest neighbors**.

Use Cases

- Recommendation engines
- Image recognition
- Pattern recognition

Pros

- Simple and intuitive
- No training phase
- Good for multi-class problems

Cons

- High prediction time on large datasets
- Sensitive to irrelevant features and scaling

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=5)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
```

5. Naive Bayes

Concept

- Based on **Bayes' Theorem** assuming **feature independence**.
- Great for text classification.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Use Cases

- Email spam filtering
- Sentiment analysis
- Document classification

Pros

- Fast and efficient
- Performs well with high-dimensional data

Cons

- Assumes independence between features
- Limited model complexity

```
from sklearn.naive_bayes import GaussianNB

model = GaussianNB()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
```

6. Support Vector Machine (SVM)

Concept

- Finds a **hyperplane** that best separates the classes.

- Uses **kernels** to handle non-linear data.

Use Cases

- Face detection
- Bioinformatics
- Text categorization

Pros

- High accuracy
- Works well for high-dimensional data

Cons

- Not good for large datasets
- Sensitive to parameter tuning

```
from sklearn.svm import SVC

model = SVC(kernel='rbf')  # Try 'linear', 'poly' as well

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
```

Classification Evaluation Metrics

These metrics help us **quantify the performance** of a classification model. They are mainly derived from the **confusion matrix**.

1. Confusion Matrix

A confusion matrix is a table used to evaluate the performance of a classification model. It compares actual vs predicted classes.

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

2. Accuracy

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

- Measures the overall correctness of the model.

- Not ideal for **imbalanced datasets**.

Example: If 95% of emails are non-spam, even a model that predicts all emails as non-spam will be 95% accurate!

3. Precision

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

- Tells **how many of the predicted positives are actually positive**.
- Important in cases like **spam detection, fraud detection**.

4. Recall (Sensitivity or True Positive Rate)

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

- Tells **how many actual positives the model identified correctly**.
- Crucial in **medical diagnosis, safety systems** where missing a positive case can be dangerous.

5. F1 Score

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

- Harmonic mean of precision and recall.
- Used when **class imbalance** exists and you want a **balance** between precision and recall.

6. Specificity (True Negative Rate)

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

- Measures how many actual negatives were correctly identified.
- Useful when **false positives** are costly.

7. ROC Curve (Receiver Operating Characteristic Curve)

- A plot of **True Positive Rate (Recall)** vs **False Positive Rate (1 - Specificity)**.
- Helps compare different models.

◆ **AUC - Area Under the Curve**

- Value ranges from 0 to 1.
- Closer to **1**, better the model.

8. Log Loss / Cross-Entropy Loss

$$\text{Log Loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

- Measures the **probability confidence** of classification.
- Lower log loss is better.

```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_auc_score, classification_report
)
# Sample predictions
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Precision:", precision_score(y_test, y_pred))
print("Recall:", recall_score(y_test, y_pred))
print("F1 Score:", f1_score(y_test, y_pred))

# Confusion Matrix
import seaborn as sns
import matplotlib.pyplot as plt
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d')
plt.title("Confusion Matrix")
plt.show()
# Classification Report
print(classification_report(y_test, y_pred))
# AUC-ROC (if probabilities available)
y_probs = model.predict_proba(X_test)[:, 1]
print("AUC Score:", roc_auc_score(y_test, y_probs))
```

Module 9: Clustering Algoritms

Clustering is an **unsupervised learning** technique used to group **similar data points** together based on features.

- No labeled output.
- Goal is to find **hidden patterns** or **intrinsic groups** in data.

Applications of Clustering

- Customer segmentation
- Market basket analysis
- Document or news article grouping
- Image compression
- Anomaly detection

Types of Clustering Techniques

- **Partitioning Methods** (e.g., K-Means)
- **Hierarchical Clustering**
- **Density-Based Clustering** (e.g., DBSCAN)

1. K-Means Clustering

Concept

- Divides data into **K clusters** by minimizing **intra-cluster variance**.
- Each point belongs to the cluster with the **nearest centroid**.

Algorithm Steps

1. Select the number of clusters K
2. Initialize K centroids randomly
3. Assign each point to the nearest centroid
4. Recompute centroids based on current assignments
5. Repeat steps 3–4 until convergence

Pros

- Simple and fast
- Works well on large datasets

Cons

- Need to specify K in advance
- Sensitive to outliers and initialization

Evaluation Metric

- **Inertia**: Sum of squared distances from each point to its assigned cluster center.
- **Silhouette Score**: How well each point fits in its cluster.

```
from sklearn.cluster import KMeans

from sklearn.metrics import silhouette_score

import matplotlib.pyplot as plt

# Create model

model = KMeans(n_clusters=3, random_state=42)

model.fit(X)

# Cluster labels and centroids

labels = model.labels_

centroids = model.cluster_centers_

# Evaluation

inertia = model.inertia_

score = silhouette_score(X, labels)
```

Elbow Method (to choose optimal K):

```
inertias = []

for k in range(1, 11):

    km = KMeans(n_clusters=k)

    km.fit(X)

    inertias.append(km.inertia_)

plt.plot(range(1, 11), inertias, marker='o')

plt.xlabel('Number of clusters')

plt.ylabel('Inertia')
```

```
plt.title('Elbow Method')

plt.show()
```

2. Hierarchical Clustering

Concept

- Creates a **dendrogram** showing a hierarchy of clusters.
- Two types:
    - **Agglomerative** (bottom-up)
    - **Divisive** (top-down)

Code (Agglomerative)

```
from sklearn.cluster import AgglomerativeClustering

import scipy.cluster.hierarchy as sch

# Dendrogram

dendrogram = sch.dendrogram(sch.linkage(X, method='ward'))

# Model

model = AgglomerativeClustering(n_clusters=3)

labels = model.fit_predict(X)
```

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

Concept

- Clusters based on density of data points.
- Can find arbitrarily shaped clusters.
- Detects outliers as noise.

Code

```
from sklearn.cluster import DBSCAN

model = DBSCAN(eps=0.5, min_samples=5)

labels = model.fit_predict(X)
```

Pros

- No need to specify number of clusters

- Robust to outliers

When to Use Which Clustering Algorithm

Algorithm	Best For	Limitations
K-Means	Spherical, equally sized clusters	Needs K; sensitive to outliers
Hierarchical	Nested grouping and dendrogram visualization	Expensive for large data
DBSCAN	Arbitrary-shaped clusters, outlier detection	Poor with varying density clusters

Module 10: Dimensionality Reduction Algorithms

Dimensionality Reduction refers to the technique of reducing the number of input variables in a dataset. In simpler terms, it's the process of compressing the feature space by **eliminating redundant or less important features**, while preserving as much useful information as possible.

Why Dimensionality Reduction is Important

- **Reduces computation time** (less features = faster training)
- **Removes multicollinearity** and noise from data
- **Improves model accuracy** and generalization
- **Makes data visualizable** in 2D/3D for humans
- **Helps reduce overfitting** in models

Types of Dimensionality Reduction Techniques

➤ **Feature Selection**

- Selecting the most important features based on statistical tests, correlation, or importance scores.
- Examples: SelectKBest, VarianceThreshold, Recursive Feature Elimination (RFE)

➤ **Feature Extraction**

- Creating new features from existing ones.
- Examples: PCA, LDA, t-SNE

Principal Component Analysis (PCA)

PCA is an **unsupervised** linear dimensionality reduction technique that transforms the features into a new set of orthogonal variables (principal components) ordered by the amount of variance they capture from the data.

How It Works

1. Standardize the dataset
2. Calculate the covariance matrix
3. Compute eigenvalues and eigenvectors
4. Select top K eigenvectors based on eigenvalues
5. Transform original data into the new feature space

Benefits

- Reduces dimensionality with **minimal loss of information**
- Helps in **visualization** and model **performance improvement**

```
from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

# Standardize the data

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

# Apply PCA

pca = PCA(n_components=2)

X_pca = pca.fit_transform(X_scaled)

# Explained variance

print(pca.explained_variance_ratio_)
```

Linear Discriminant Analysis (LDA)

LDA is a **supervised** dimensionality reduction technique used for classification tasks. It finds linear combinations of features that **best separate the classes**.

How It Works:

1. Compute the **within-class** and **between-class scatter matrices**
2. Solve the **eigenvalue problem**
3. Project data to lower-dimensional space that maximizes class separability

Use Case

Best suited for **classification problems** with labeled data.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis(n_components=2)

X_lda = lda.fit_transform(X, y)
```

t-SNE (t-distributed Stochastic Neighbor Embedding)

t-SNE is a **non-linear**, unsupervised technique for dimensionality reduction, primarily used for **visualization** of high-dimensional data.

Key Idea

t-SNE tries to maintain the **local structure** of the data — points that are close together in high dimensions stay close in low dimensions.

Important Parameters:

- n_components: Target dimension (usually 2 or 3)
- perplexity: Balance between local and global aspects
- n_iter: Number of iterations for optimization

Note: Not suitable for model input. Use only for visualization.

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=30, n_iter=1000)

X_tsne = tsne.fit_transform(X)
```

Comparison Table

Technique	Type	Supervised	Use-case	Notes
PCA	Linear	❌ No	Dimensionality reduction	Based on variance
LDA	Linear	✅ Yes	Classification problems	Maximizes class separation
t-SNE	Non-linear	❌ No	Visualization (e.g., clustering)	Preserves local relationships

Visualizing with PCA vs t-SNE

```
# PCA Plot

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis')

plt.title('PCA Projection')

plt.xlabel('PC1')
```

```
plt.ylabel('PC2')

plt.colorbar()

plt.show()


# t-SNE Plot

plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='plasma')

plt.title('t-SNE Projection')

plt.xlabel('Dim 1')

plt.ylabel('Dim 2')

plt.colorbar()

plt.show()
```

When to Use What?

- **PCA**: When you want to reduce dimensionality to speed up computation or reduce noise in **unsupervised** learning.
- **LDA**: When you're solving a **classification** problem with **labelled data**.
- **t-SNE**: When your goal is **data visualization**, especially for clustering or high-dimensional datasets.

Module 11: Ensemble Learning Algorithms

Ensemble learning is a machine learning technique where **multiple individual models (often called "weak learners") are combined** to create a **stronger, more accurate predictive model**.

Why use Ensemble Learning?

- To improve **prediction accuracy**
- To increase **model robustness**
- To reduce **overfitting or underfitting**
- To handle **imbalanced** or **noisy datasets**

1. Bagging (Bootstrap Aggregating)

Concept

- Build multiple models using **random subsets** (with replacement) of training data.
- All models are trained **independently**.
- Final prediction is **average (for regression)** or **majority vote (for classification)**.

Advantages

- Reduces **variance**
- Helps in avoiding overfitting

Popular Algorithm: Random Forest

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)
```

2. Boosting

Concept

- Train models **sequentially**, each trying to fix the mistakes of the previous one.
- Assign more weight to misclassified samples.
- Final model is a **weighted sum** of all learners.

Advantages

- Reduces **bias and variance**
- Works well on imbalanced datasets

a. AdaBoost (Adaptive Boosting)

- Focuses on samples that are hard to classify.
- Assigns weights to each instance, increasing weights of misclassified ones.
- Combines multiple **weak learners (typically stumps)** into a strong learner.

```
from sklearn.ensemble import AdaBoostClassifier

model = AdaBoostClassifier(n_estimators=50, learning_rate=1.0, random_state=42)

model.fit(X_train, y_train)
```

b. Gradient Boosting

- Works by building models that **predict residuals (errors)** of previous models.
- Uses **gradient descent** to minimize loss.
- Each new tree improves the total model incrementally.

```
from sklearn.ensemble import GradientBoostingClassifier

model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, random_state=42)

model.fit(X_train, y_train)
```

c. XGBoost (Extreme Gradient Boosting)

- Advanced and efficient version of gradient boosting.
- Adds **regularization (L1 and L2)** to avoid overfitting.
- Very **fast** and handles **missing values**.
- One of the top choices in **Kaggle competitions**.

```
import xgboost as xgb

model = xgb.XGBClassifier(n_estimators=100, learning_rate=0.1, use_label_encoder=False, eval_metric='logloss')

model.fit(X_train, y_train)
```

3. Stacking (Stacked Generalization)

Concept

- Combines predictions from **multiple base learners** using a **meta-model** (e.g., Logistic Regression).

- Base models are trained independently.
- Their predictions are fed to a final model that learns the best combination.

Advantages

- Leverages **diversity** of different models.
- Can outperform individual models or simple ensembles.

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
base_models = [
    ('svc', SVC(probability=True)),
    ('dt', DecisionTreeClassifier())
]
meta_model = LogisticRegression()
model = StackingClassifier(estimators=base_models, final_estimator=meta_model)
model.fit(X_train, y_train)
```

Comparison Table

Technique	Combines	Model Sequence	Reduces	Handles
Bagging	Same models	No	Variance	Overfitting
Boosting	Same models	Yes	Bias & Variance	Underfitting
Stacking	Different models	Yes	Bias	Bias & Variance

When to Use What?

Situation	Use This Technique
Model is overfitting	Bagging (Random Forest)
Model is underfitting	Boosting (Gradient, XGB)

Want to combine strengths of different models	Stacking
Best model for structured/tabular data	XGBoost

Summary

- Ensemble learning is a powerful approach to **improve performance** and **stability** of your models.
- Choose the right strategy based on your **problem**, **data size**, and **model performance**.
- **Boosting** often performs best, but can be prone to overfitting if not tuned properly.
- **Stacking** is more complex but can yield state-of-the-art results when done right.

Module 12: Reinforcement Learning

Reinforcement Learning is a type of Machine Learning where an **agent learns by interacting with an environment**, receiving **rewards or penalties** for actions taken, and learning to **maximize cumulative rewards** over time.

Key Concepts

- **Agent**: The decision-maker (e.g., a robot, AI system)
- **Environment**: The world the agent interacts with
- **State (S)**: Current situation
- **Action (A)**: Choices the agent can take
- **Reward (R)**: Feedback for the action
- **Policy (π)**: Strategy to decide the next action
- **Value Function (V)**: Expected long-term reward from a state

Types of Reinforcement Learning

- **Positive RL**: Encourages behavior by reward
- **Negative RL**: Encourages behavior by avoiding punishment

Popular Algorithms

- **Q-Learning**: Learns optimal actions through a Q-table
- **SARSA**: Learns using the actual action taken (on-policy)
- **Deep Q-Network (DQN)**: Uses neural networks for large state/action spaces
- **Policy Gradient Methods**: Learn the policy directly (used in advanced RL)

Applications

- Game AI (Chess, Go, Atari)
- Robotics (navigation, motion control)
- Finance (portfolio management)
- Healthcare (personalized treatments)
- Self-driving cars (decision-making)

Reinforcement Learning is used when **supervision is not available** and the agent must learn from **experience and feedback**.

Module 13: Recommendation Systems

A **Recommendation System (Recommender System)** suggests relevant items to users by learning from their preferences, behavior, or interactions. Commonly used in:

- E-commerce (Amazon)
- Streaming platforms (Netflix, Spotify)
- Social media (YouTube, Instagram)

1. Content-Based Filtering

- Recommends items similar to those the user liked in the past.
- Uses item attributes (genre, category, keywords).
- Personalized for each user.

Example: If you liked an action movie, the system recommends similar action movies.

Techniques Used:

- TF-IDF
- Cosine Similarity
- NLP-based content profiling

2. Collaborative Filtering

- Based on user-item interactions (ratings, purchases).
- Assumes similar users like similar items.

a) User-Based Collaborative Filtering

- Recommends items liked by similar users.

b) Item-Based Collaborative Filtering

- Recommends items similar to those the user rated/liked.

Techniques Used:

- k-Nearest Neighbors (kNN)
- Cosine similarity

- Pearson correlation

3. Hybrid Systems

- Combine content-based and collaborative methods for better accuracy.

Evaluation Metrics

- **RMSE** (Root Mean Square Error)
- **MAE** (Mean Absolute Error)
- **Precision / Recall / F1-Score**
- **Coverage, Diversity, Novelty**

Libraries for Implementation

- scikit-learn
- surprise (for collaborative filtering)
- lightfm
- implicit
- pandas, numpy

Example – Content-Based Filtering (using Cosine Similarity)

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine_similarity

import pandas as pd

# Sample movie data

data = {

  'title': ['The Matrix', 'Inception', 'Interstellar', 'The Dark Knight'],

  'description': [

    'sci-fi action dystopia',

    'dreams sci-fi thriller',

    'space sci-fi adventure',

    'superhero action drama'
```

```
    ]
}
df = pd.DataFrame(data)
# TF-IDF Vectorization
tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(df['description'])
# Cosine Similarity Matrix
cosine_sim = cosine_similarity(tfidf_matrix)
# Function to recommend similar movies
def recommend_movie(index):
    scores = list(enumerate(cosine_sim[index]))
    scores = sorted(scores, key=lambda x: x[1], reverse=True)
    return [df['title'][i[0]] for i in scores[1:3]]  # Top 2 recommendations
print(recommend_movie(0))  # Recommend for 'The Matrix'
```

Example – Collaborative Filtering (with Surprise Library)

```
from surprise import Dataset, Reader, SVD
from surprise.model_selection import cross_validate
# Sample data
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_builtin('ml-100k')  # MovieLens dataset
# SVD model
model = SVD()
cross_validate(model, data, measures=['RMSE', 'MAE'], cv=3, verbose=True)
```

Challenges in Recommender Systems

- Cold start (new user/item)
- Scalability with large data

- Sparsity in user-item interactions
- Diversity vs. Accuracy trade-off

Summary

Type	Input	Output	Use Case
Content-Based	Item features	Similar items	Netflix: show similar to watched
Collaborative	User behavior	Popular among similar users	Amazon: “People also bought”
Hybrid	Both	Accurate and diverse recos	Spotify, YouTube

Module 14: Model Tuning and Optimization

Model tuning is the process of adjusting model parameters to improve its **performance** and **generalization ability**. It helps in:

- Reducing overfitting or underfitting
- Improving accuracy and reliability
- Selecting the best performing model

Key Concepts - Hyperparameters vs Parameters

Parameters	Hyperparameters
Learned during training	Set manually before training
e.g., weights in LR	e.g., learning rate, n_estimators

Hyperparameter Tuning Techniques

1. Grid Search

- Exhaustive search over a manually specified set of hyperparameter values.
- Time-consuming but effective for small search spaces.

```
from sklearn.model_selection import GridSearchCV

params = {'n_neighbors': [3, 5, 7]}

grid = GridSearchCV(KNeighborsClassifier(), param_grid=params, cv=5)

grid.fit(X_train, y_train)

print(grid.best_params_)
```

2. Randomized Search

- Randomly samples combinations from a hyperparameter space.
- Faster and more efficient for large search spaces.

```
from sklearn.model_selection import RandomizedSearchCV

params = {'n_estimators': [50, 100, 150], 'max_depth': [3, 5, 7]}

random_search = RandomizedSearchCV(RandomForestClassifier(), param_distributions=params, cv=5)

random_search.fit(X_train, y_train)

print(random_search.best_params_)
```

3. Bayesian Optimization (Optional Advanced)

- Uses probability models to find the best set of hyperparameters efficiently.
- Libraries: optuna, hyperopt, scikit-optimize

Cross-Validation

Why Cross-Validation?

- To evaluate model performance on different subsets of data.
- Helps to reduce variance and test robustness.

K-Fold Cross-Validation

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, cv=5)

print(scores.mean())
```

Model Selection

- Use **Evaluation Metrics** (Accuracy, Precision, RMSE, etc.)
- Compare multiple algorithms
- Use **Cross-validation scores** for comparison
- Choose based on:
    - Best performance
    - Generalization
    - Simplicity and interpretability

Bias-Variance Tradeoff

Bias	Variance	Result
High	Low	Underfitting

Low	High	Overfitting
Low	Low	Good model

Regularization (Recap)

- **L1 Regularization (Lasso)**: Shrinks some coefficients to 0 — good for feature selection.
- **L2 Regularization (Ridge)**: Penalizes large weights — keeps all features, reduces multicollinearity.

Pipeline Creation (Best Practice)

Helps to streamline preprocessing + modeling + tuning.

```
from sklearn.pipeline import Pipeline

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression())
])

pipe.fit(X_train, y_train)
```

Model Saving and Loading

```
import joblib

joblib.dump(model, 'model.pkl')

model = joblib.load('model.pkl')
```

```
import pickle as pkl

pkl.dump(model, open("model.pkl", "wb"))

algo = pkl.load(open("SVC.pkl", "rb"))
```

Summary

Task	**Tools**
Tune model hyperparameters	GridSearchCV, RandomizedSearchCV
Validate model	cross_val_score, KFold

Reduce overfitting	Regularization, Ensembling
Deploy best model	Pickle/Joblib + Streamlit/Flask

Module 15: Capstone Projects

Project Guidelines

Each capstone project should:

- Involve **EDA**, **feature engineering**, and **preprocessing**
- Use **multiple ML algorithms** for comparison
- Apply **model evaluation metrics**
- Include **hyperparameter tuning**
- Showcase **model deployment** using Streamlit / Flask / FastAPI
- Be documented with **proper reports and GitHub submission**

Suggested Projects

No	Project Title	Problem Type	Algorithms to Use	Deployment
1	Email Spam Detection	Classification	Naive Bayes, Logistic Regression, SVM	Flask / Streamlit
2	Credit Card Fraud Detection	Classification	Isolation Forest, Random Forest, XGBoost	Streamlit + AWS/GCP
3	Heart Disease Prediction	Classification	Logistic Regression, KNN, Random Forest	Flask + HTML UI
4	Student Performance Classification	Classification	Decision Tree, SVM, XGBoost	Streamlit App
5	Fake News Detection	Classification	TF-IDF + Logistic Regression, PassiveAggressiveClassifier	Flask + Heroku
6	House Price Prediction	Regression	Linear, Ridge, Lasso, Random Forest	Streamlit App

7	Salary Predictor	Regression	Multiple Linear Regression, SVR	Flask + Bootstrap UI
8	Flight Fare Prediction	Regression	XGBoost, Decision Tree, Random Forest	Streamlit + Docker
9	Car Price Prediction	Regression	Ridge, Lasso, Polynomial Regression	Flask
10	Insurance Premium Estimator	Regression	Linear Regression, Decision Tree, XGBoost	Streamlit
11	Customer Segmentation	Clustering	KMeans, DBSCAN, Hierarchical Clustering	Streamlit + Plotly
12	Market Basket Analysis	Clustering	Apriori, KMeans, Association Rules	Flask
13	Website Visitor Segmentation	Clustering	KMeans, PCA	Streamlit Dashboard
14	Animal Grouping	Clustering	Hierarchical, KMeans	Streamlit
15	Movie Recommendation System	Recommendation	Cosine Similarity, Matrix Factorization	Streamlit + TMDb API
16	Book Recommender	Recommendation	Collaborative Filtering, Content-Based Filtering	Flask
17	Music Recommendation Engine	Recommendation	KNN, Cosine Similarity, Clustering	Streamlit
18	E-Commerce Product Recommendation	Recommendation	Collaborative Filtering, Hybrid Filtering	Flask / React frontend
19	Job Recommendation System	Recommendation	NLP + TF-IDF + Cosine Similarity	Streamlit
20	Voting Classifier for Churn Prediction	Ensemble Learning	Logistic, Decision Tree, SVM (VotingClassifier)	Streamlit
21	Heart Disease with Stacking Models	Ensemble Learning	Random Forest + SVM + XGBoost	Flask

22	Loan Risk Prediction	Ensemble Learning	Bagging, Random Forest, AdaBoost	Streamlit + MongoDB
23	Air Quality Index Prediction	Regression	Linear Regression, XGBoost	Streamlit + CSV upload
24	Course Recommendation System	Recommendation	Content-based Filtering, Cosine Similarity	Flask + Streamlit Hybrid App
25	Retail Demand Forecasting	Regression	LSTM, Random Forest, Linear Regression	Flask or Streamlit
26	Stock Price Movement Classification	Classification	Logistic Regression, XGBoost	Streamlit + Plotly
27	Image Classification (Dogs vs Cats)	Classification	CNN, Transfer Learning	Flask + TensorFlow Lite
28	News Category Classifier	Classification	TF-IDF + MultinomialNB	Streamlit
29	Student Dropout Predictor	Classification	Logistic Regression, Random Forest, AdaBoost	Flask

Tools & Technologies to Be Used

- **Python**, **Pandas**, **NumPy**
- **Scikit-learn**, **XGBoost**, **Seaborn**, **Matplotlib**
- **Streamlit / Flask / FastAPI** for deployment
- **Git & GitHub** for version control and project presentation
- **Jupyter Notebook / VS Code**

Deliverables for Each Project

1. **Source Code**
2. **Jupyter Notebook with step-by-step explanation**
3. **Presentation Slides (PPT/PDF)**
4. **Model Report**
5. **Live Deployment (optional)**

6. **GitHub Repository**