ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

◇ INTRODUCTION TO MACHINE LEARNING (ML) & SUPERVISED LEARNING

📌 CHAPTER 1: INTRODUCTION TO MACHINE LEARNING

🔸 1.1 What is Machine Learning?

Machine Learning (ML) is a branch of **artificial intelligence (AI)** that enables computers to learn from data and improve their performance **without being explicitly programmed**. It focuses on building models that can recognize patterns, make predictions, and automate decision-making processes based on historical data. The goal of ML is to develop algorithms that **generalize well to new data**, allowing machines to **self-improve over time**.

ML is widely used in various applications, such as **speech recognition, recommendation systems, fraud detection, self-driving cars, and medical diagnosis**. Instead of following predefined rules, ML models **analyze data, identify patterns, and make predictions** using mathematical algorithms.

Key Characteristics of Machine Learning:

✓ **Data-Driven Approach** – ML models learn from past data and apply their knowledge to make decisions.

✓ **Pattern Recognition** – Identifies complex relationships in data.
✓ **Continuous Improvement** – Performance improves as more data is fed into the model.
✓ **Automation & Adaptability** – Reduces manual intervention and adapts to new information.

📌 **Example:**
Netflix uses **Machine Learning algorithms** to **recommend movies and TV shows** based on a user's viewing history, preferences, and behavior patterns.

💡 **Conclusion:**
Machine Learning is a **core technology behind AI-driven automation**, helping organizations improve efficiency, make data-driven decisions, and enhance user experiences.

🔹 1.2 Types of Machine Learning

ML is broadly classified into **three types** based on how models learn from data:

1. Supervised Learning (Labeled Data)

- The model learns from **labeled data**, where each input has a corresponding output.
- It is used for **classification and regression problems**.

📌 **Example:** Predicting house prices based on historical data (inputs: size, location; output: price).

2. Unsupervised Learning (No Labels)

- The model learns patterns from **unlabeled data** without predefined categories.

- Used for **clustering and association problems**.

📌 **Example:** Customer segmentation for targeted marketing (grouping customers based on shopping habits).

3. Reinforcement Learning (Trial & Error)

- The model learns by **interacting with an environment and receiving rewards or penalties** for its actions.
- Used in robotics, gaming, and self-driving cars.

📌 **Example:** AI in **chess games,** where the model plays against itself and improves through experience.

💡 **Conclusion:**
Each type of ML has its **unique learning approach and applications,** making it essential to choose the **right method based on the problem type**.

📌 Chapter 2: Introduction to Supervised Learning

🔸 2.1 What is Supervised Learning?

Supervised Learning is the **most common type of ML,** where the model is trained using **labeled data**. The dataset consists of **input features (X) and corresponding output labels (Y),** allowing the model to learn the relationship between them and make predictions on new data.

In supervised learning, the **goal is to minimize the difference between the predicted output and the actual output** by adjusting the model's parameters using optimization techniques.

How Supervised Learning Works:

1. **Collect and prepare labeled training data**.
2. **Choose an appropriate model (algorithm)**.
3. **Train the model using historical data**.
4. **Evaluate model performance** using test data.
5. **Deploy the model for real-world predictions**.

📌 **Example:**

- **Email Spam Detection** – The model is trained on **emails labeled as "spam" or "not spam"** and learns to classify future emails automatically.
- **Loan Approval System** – The model learns from past loan applications (**features: income, credit score, debt-to-income ratio**) to predict if a new applicant will **repay the loan or default**.

💡 **Conclusion:**

Supervised learning is highly effective for **structured problems with labeled data**, such as **fraud detection, medical diagnosis, and financial forecasting**.

🔹 **2.2 Types of Supervised Learning**

Supervised learning problems are categorized into **two main types**:

1. Classification (Categorical Output)

- The model predicts **discrete categories or labels**.
- Used when the target variable consists of **finite classes**.

📌 **Examples:**

✓ **Spam detection** – Classifying emails as **Spam or Not Spam**.

✓ **Credit approval** – Predicting **Loan Approved (Yes/No)** based on customer data.
✓ **Disease Diagnosis** – Identifying if a patient has **Diabetes (Yes/No)**.

2. Regression (Continuous Output)

- The model predicts **continuous numerical values**.
- Used when the target variable is **continuous and not categorical**.

📌 **Examples:**
✓ **House price prediction** – Predicting home prices based on **size, location, and amenities**.
✓ **Stock market forecasting** – Estimating future stock prices using historical trends.
✓ **Weather prediction** – Forecasting temperature based on **past weather patterns**.

💡 **Conclusion:**
Choosing **classification or regression** depends on whether the **target variable is categorical (classification) or continuous (regression)**.

🔹 **2.3 Common Supervised Learning Algorithms**

Supervised learning algorithms **analyze patterns in labeled data** and make predictions. The most commonly used algorithms include:

1. Linear Regression (For Regression Problems)

- Used for **predicting continuous values**.

- Finds the **best-fit line** to establish relationships between variables.

📌 **Example:** Predicting house prices based on **size, location, and number of bedrooms**.

2. Logistic Regression (For Classification Problems)

- Used for **binary classification** (Yes/No, 0/1).
- Uses a **sigmoid function** to estimate probabilities.

📌 **Example:** Predicting whether a **customer will buy a product (Yes/No)**.

3. Decision Trees (For Both Classification & Regression)

- Creates a tree-like model to **split data into smaller subsets** based on features.
- Easy to interpret but prone to overfitting.

📌 **Example:** Predicting if a **customer will default on a loan** based on their financial history.

4. Random Forest (Ensemble Method)

- Uses **multiple decision trees** to improve accuracy.
- Reduces **overfitting and enhances model robustness**.

📌 **Example:** Identifying fraudulent transactions in **banking systems**.

5. Support Vector Machines (SVM)

- Finds the **optimal decision boundary** to classify data points.
- Works well in **high-dimensional data** scenarios.

📌 **Example:** Handwriting recognition (e.g., identifying digits in postal codes).

6. Neural Networks (Deep Learning)

- Mimics the **human brain's structure** with multiple layers.
- Best suited for **complex tasks** like **image recognition and NLP (Natural Language Processing)**.

📌 **Example:** Identifying **objects in self-driving cars** (stop signs, pedestrians, traffic lights).

💡 **Conclusion:**
The choice of algorithm depends on **data type, problem complexity, and accuracy requirements**.

📌 Summary & Next Steps

✅ **Key Takeaways:**

✓ **Machine Learning enables computers to learn patterns from data without explicit programming**.
✓ **Supervised Learning** is widely used for structured problems with labeled datasets.
✓ **Classification and Regression** are the two main types of supervised learning.
✓ Popular algorithms include **Linear Regression, Decision Trees, Random Forest, and Neural Networks**.

📌 **Next Steps:**

- 🔹 **Practice ML with Python using Scikit-Learn**.
- 🔹 **Apply supervised learning models to real-world datasets**.

- **Explore advanced ML topics like feature engineering and hyperparameter tuning.**

◇ Regression Techniques: Linear Regression, Polynomial Regression

📌 Chapter 1: Introduction to Regression Analysis

🔸 1.1 What is Regression Analysis?

Regression analysis is a statistical method used to **model relationships between dependent and independent variables**. It helps in **predicting outcomes, understanding data trends, and making informed decisions**. Regression is widely used in **machine learning, finance, healthcare, business analytics, and engineering**.

Why is Regression Important?

✓ **Predicting Outcomes** – Forecasts future trends based on historical data.
✓ **Understanding Relationships** – Identifies the influence of one variable on another.
✓ **Optimizing Business Strategies** – Helps companies make data-driven decisions.
✓ **Feature Selection** – Determines which variables are most significant.

Types of Regression Models:

1. **Linear Regression** – Models relationships using a straight line.
2. **Polynomial Regression** – Fits a curved relationship using higher-degree polynomials.
3. **Multiple Regression** – Includes multiple independent variables.

4. **Logistic Regression** – Used for classification problems (binary outcomes).

📌 **Example:**
A company wants to predict **house prices** based on factors like **square footage, number of bedrooms, and location**. Regression analysis helps find the best-fitting model to make predictions.

💡 **Conclusion:**
Regression analysis is a **powerful statistical tool** that helps **interpret data relationships and make accurate predictions**.

📌 Chapter 2: Linear Regression

🔹 2.1 What is Linear Regression?

Linear Regression is the simplest form of regression analysis. It models the relationship between **one dependent variable (Y)** and **one independent variable (X)** using a straight line. The goal is to find the **best-fitting line** that minimizes errors.

Mathematical Equation of Linear Regression:

$$Y = b_0 + b_1 X + \epsilon$$

where:

- Y = Dependent variable (target variable)
- X = Independent variable (predictor)
- b_0 = Intercept (where the line crosses the Y-axis)
- b_1 = Slope (how much Y changes for each unit change in X)
- ϵ = Error term (accounts for variability in the data)

Key Assumptions of Linear Regression:

✓ **Linearity** – The relationship between X and Y must be linear.
✓ **Independence** – Observations should be independent.
✓ **Homoscedasticity** – Constant variance of residuals (errors).
✓ **Normality of Residuals** – Errors should be normally distributed.

📌 **Example:**
A **restaurant owner** wants to predict **monthly revenue (Y)** based on **advertising spending (X)**. Linear regression helps determine how much revenue increases for every extra dollar spent on ads.

🔹 2.2 Implementing Linear Regression in Python

Step 1: Import Necessary Libraries

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error, r2_score
```

Step 2: Load and Prepare the Dataset

```
# Example dataset (Advertising vs Sales)

data = pd.DataFrame({

  'Advertising': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],

  'Sales': [5, 9, 13, 18, 21, 26, 33, 38, 45, 50]
```

```
})


# Define X (independent variable) and Y (dependent variable)
X = data[['Advertising']]  # Independent variable
Y = data['Sales']  # Dependent variable


# Split data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
```

Step 3: Train the Model

```
# Create and fit the model
model = LinearRegression()
model.fit(X_train, Y_train)


# Make predictions
Y_pred = model.predict(X_test)
```

Step 4: Evaluate the Model

```
print(f"Intercept: {model.intercept_}")
print(f"Coefficient: {model.coef_[0]}")
print(f"Mean Squared Error: {mean_squared_error(Y_test, Y_pred)}")
print(f"R-squared: {r2_score(Y_test, Y_pred)}")
```

Step 5: Visualizing the Regression Line

```
plt.scatter(X, Y, color='blue', label='Actual Data')

plt.plot(X, model.predict(X), color='red', linewidth=2,
label='Regression Line')

plt.xlabel("Advertising Spend ($)")

plt.ylabel("Sales ($)")

plt.legend()

plt.show()
```

📌 **Example:**
If the **coefficient (b_1) = 0.5**, then every **$1 increase in advertising** leads to a **$0.5 increase in sales**.

💡 **Conclusion:**
Linear Regression is **widely used in economics, business forecasting, and machine learning** for making data-driven predictions.

📌 Chapter 3: Polynomial Regression

🔹 3.1 What is Polynomial Regression?

Polynomial Regression is an extension of Linear Regression where the relationship between the independent and dependent variables is **curved (non-linear)**. Instead of fitting a straight line, it fits a **parabolic (or higher-degree) curve** to the data.

Mathematical Equation of Polynomial Regression:

$$Y = b_0 + b_1X + b_2X^2 + b_3X^3 + \cdots + b_nX^n + \epsilon$$

where:

- $X^2, X^3, \ldots X^n$ = Higher-order polynomial terms.
- The degree **(n)** determines the curvature of the model.

When to Use Polynomial Regression?

✓ **When data is non-linear but continuous**.
✓ **When Linear Regression performs poorly**.
✓ **When relationships have turning points** (e.g., profit over time).

📌 **Example:**
A **company** analyzing **sales vs. advertising spend** might notice **diminishing returns** (higher spending doesn't always lead to proportional growth). Polynomial regression can model this trend better.

🔸 3.2 Implementing Polynomial Regression in Python

Step 1: Import Necessary Libraries

```
from sklearn.preprocessing import PolynomialFeatures

from sklearn.pipeline import make_pipeline
```

Step 2: Create Polynomial Features & Train the Model

```
# Define degree of the polynomial

degree = 2  # Quadratic (can increase for more complexity)
```

```
# Create polynomial regression model

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

poly_model.fit(X_train, Y_train)


# Make predictions

Y_poly_pred = poly_model.predict(X_test)
```

Step 3: Visualizing Polynomial Regression

```
X_range = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)  # Generating values for smooth curve

Y_range_pred = poly_model.predict(X_range)


plt.scatter(X, Y, color='blue', label="Actual Data")

plt.plot(X_range, Y_range_pred, color='red', linewidth=2, label="Polynomial Regression Curve")

plt.xlabel("Advertising Spend ($)")

plt.ylabel("Sales ($)")

plt.legend()

plt.show()
```

📌 **Example:**
If **degree = 2,** the model fits a **quadratic curve** (U-shaped or inverted U). If **degree = 3,** it fits an **S-shaped curve**.

💡 **Conclusion:**

Polynomial Regression is **useful for capturing non-linear trends,** but higher-degree models may **overfit** the data.

📌 Summary & Next Steps

✅ **Key Takeaways:**

✓ **Linear Regression** fits a **straight line** to model relationships.

✓ **Polynomial Regression** fits a **curved trend line** for non-linear data.

✓ **Choosing the right regression model** depends on **data distribution and complexity**.

📌 **Next Steps:**

🔹 **Apply regression models on real-world datasets (Kaggle, UCI ML Repository).**

🔹 **Experiment with different polynomial degrees to see overfitting effects.**

🔹 **Learn advanced regression techniques like Ridge & Lasso Regression.** 🚀

◇ Classification Techniques: Logistic Regression, Decision Trees, Random Forest

📌 Chapter 1: Introduction to Classification Techniques

🔹 1.1 What is Classification?

Classification is a **supervised machine learning technique** where the goal is to predict the **category or class** of a given data point. Unlike regression, where the output is continuous, classification deals with **discrete outcomes**.

Key Features of Classification Models:

✓ **Categorical Output** – The target variable belongs to a class (e.g., spam or not spam).
✓ **Supervised Learning** – Requires labeled data for training.
✓ **Probabilistic or Rule-Based Predictions** – Some models use probabilities, while others use decision rules.
✓ **Binary or Multi-Class Classification** – Predicts between two classes (e.g., yes/no) or multiple classes (e.g., cat/dog/rabbit).

📌 Examples:

- **Email Spam Detection** – Classifying emails as **spam or not spam**.
- **Credit Approval** – Determining whether a loan application should be **approved or rejected**.

- **Disease Diagnosis** – Predicting if a patient has **diabetes (yes/no)**.

💡 **Conclusion:**
Classification is a **fundamental problem in data science,** widely applied in finance, healthcare, marketing, and cybersecurity.

🔸 1.2 Types of Classification Algorithms

Classification algorithms can be categorized into **different types based on their methodology**:

1. Linear Classifiers:

- **Logistic Regression** – Best for linearly separable data.
- **Support Vector Machine (SVM)** – Uses hyperplanes for classification.

2. Tree-Based Classifiers:

- **Decision Trees** – Uses hierarchical rules for classification.
- **Random Forest** – An ensemble of decision trees to improve accuracy.

3. Probabilistic Classifiers:

- **Naïve Bayes** – Based on Bayes' Theorem for probabilistic classification.

4. Neural Network-Based Classifiers:

- **Deep Learning Models** – Complex models like Convolutional Neural Networks (CNNs).

💡 **Conclusion:**
Choosing the **right classification model** depends on the **data complexity, interpretability, and performance requirements**.

📌 Chapter 2: Logistic Regression

🔹 2.1 What is Logistic Regression?

Logistic Regression is a **statistical classification algorithm** used for **binary classification problems**. It estimates the **probability** that a given input belongs to a specific class.

Key Characteristics of Logistic Regression:

✓ **Binary Classification** – Predicts outcomes like **yes/no, true/false**.
✓ **Uses Sigmoid Function** – Converts predictions into probabilities.
✓ **Interpretable Model** – Coefficients indicate feature importance.
✓ **Works Well with Linearly Separable Data** – Performs best when classes can be separated by a straight line.

📌 **Example Use Cases:**

- **Predicting whether a customer will buy a product (yes/no).**
- **Diagnosing a patient with a disease (positive/negative).**

💡 **Conclusion:**
Logistic Regression is **simple, effective, and widely used** for binary classification problems.

🔸 2.2 The Logistic Regression Model

Logistic regression **calculates the probability of an event occurring using the sigmoid function**:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(b_0 + b_1 X)}}$$

where:

- b_0 = Intercept
- b_1 = Coefficient of feature X
- e = Euler's number (approx. 2.718)

📌 Example Implementation in Python:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score


# Sample dataset

data = pd.read_csv("heart_disease.csv")

X = data[["age", "cholesterol", "blood_pressure"]]

y = data["has_disease"]
```

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

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

# Predictions
y_pred = model.predict(X_test)

# Model evaluation
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")
```

💡 **Conclusion:**
Logistic Regression is a **lightweight, easy-to-interpret** classifier that performs well on **structured datasets**.

📌 CHAPTER 3: DECISION TREES

🔸 3.1 What is a Decision Tree?

A **Decision Tree** is a classification algorithm that makes predictions based on a **hierarchical series of rules**. It is similar to a **flowchart,** where each internal node represents a decision based on a feature.

Key Features of Decision Trees:

✓ **Non-Linear Decision Boundaries** – Can handle both linear and non-linear data.
✓ **Works Well with Categorical & Numerical Data** – Can handle mixed feature types.
✓ **Interpretable Model** – Provides a clear decision path.
✓ **Prone to Overfitting** – Needs pruning or depth constraints to generalize well.

📌 **Example Use Cases:**

- **Loan Approval** – Determining if a customer qualifies for a loan.
- **Medical Diagnosis** – Predicting disease presence based on symptoms.

💡 **Conclusion:**
Decision Trees are **powerful and interpretable,** but can be **prone to overfitting** if not controlled.

🔹 **3.2 How Decision Trees Work**

A Decision Tree makes predictions by **splitting data** at decision nodes using an algorithm such as **Gini Index or Entropy**.

📌 **Example Implementation in Python:**

```
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn import tree


# Train Decision Tree model
dt_model = DecisionTreeClassifier(max_depth=3)
dt_model.fit(X_train, y_train)


# Visualizing the decision tree
plt.figure(figsize=(12, 8))
tree.plot_tree(dt_model, feature_names=["age", "cholesterol", "blood_pressure"], filled=True)
plt.show()
```

💡 **Conclusion:**
Decision Trees are useful for **explainable AI**, but **pruning techniques** are necessary to **reduce overfitting**.

📌 Chapter 4: Random Forest

🔹 4.1 What is a Random Forest?

Random Forest is an **ensemble learning method** that builds multiple decision trees and combines their predictions to **improve accuracy and reduce overfitting**.

Key Features of Random Forest:

✓ **Uses Multiple Decision Trees** – Reduces the risk of overfitting.
✓ **Robust to Noisy Data** – More stable compared to individual

decision trees.

✓ **Handles Large Datasets Well** – Works efficiently with large feature spaces.

✓ **Provides Feature Importance Scores** – Identifies the most influential variables.

📌 **Example Use Cases:**

- **Fraud Detection** – Identifying fraudulent transactions.
- **Customer Churn Prediction** – Predicting if customers will leave a subscription service.

💡 **Conclusion:**

Random Forest improves **classification accuracy** while maintaining **stability** and **generalization**.

🔸 **4.2 How Random Forest Works**

Random Forest works by:

1. **Bootstrapping** – Randomly selecting subsets of data for training each tree.
2. **Feature Randomization** – Using a subset of features for each split.
3. **Majority Voting** – Aggregating predictions from multiple trees.

📌 **Example Implementation in Python:**

```
from sklearn.ensemble import RandomForestClassifier
```

```
# Train Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predict and evaluate
y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Random Forest Accuracy: {accuracy:.2f}")
```

💡 **Conclusion:**
Random Forest **improves accuracy, reduces overfitting, and works well on complex datasets**.

📌 Summary & Next Steps

✅ **Key Takeaways:**
✓ Logistic Regression – Best for linearly separable binary classification.
✓ Decision Trees – Intuitive but prone to overfitting.
✓ Random Forest – More accurate, reduces overfitting, and generalizes well.

📌 **Next Steps:**
- Experiment with different classification models on datasets.
- Optimize hyperparameters for better model performance.
- Explore deep learning classifiers like Neural Networks! 🚀

◇ Model Evaluation: Accuracy, Precision, Recall, F1-Score

📌 Chapter 1: Introduction to Model Evaluation

🔹 1.1 What is Model Evaluation?

Model evaluation is the process of **assessing the performance of a machine learning model** to determine how well it generalizes to unseen data. The evaluation metrics provide insights into whether the model is **accurate, reliable, and useful** for decision-making.

A well-trained model may still perform poorly if it **overfits the training data** or **fails to generalize**. Therefore, understanding key evaluation metrics such as **Accuracy, Precision, Recall, and F1-Score** is crucial for selecting the best model.

Why is Model Evaluation Important?

✓ Helps in choosing the best model for real-world applications.
✓ Identifies **bias, variance, and overfitting** issues.
✓ Ensures **reliable and interpretable** decision-making.
✓ Assesses **classification and regression model performance**.

📌 Example:
A **fraud detection system** must be evaluated to ensure it correctly detects fraudulent transactions while minimizing false alarms.

💡 Conclusion:
Model evaluation ensures that a machine learning model is **accurate, fair, and effective** before deploying it into production.

🔸 1.2 Key Model Evaluation Metrics

There are different evaluation metrics for **classification and regression models**. This chapter focuses on **classification model evaluation** using metrics such as **Accuracy, Precision, Recall, and F1-Score**.

Metric	Description
Accuracy	Measures overall correctness of the model.
Precision	Measures how many predicted positives are actually correct.
Recall (Sensitivity)	Measures how well the model detects actual positives.
F1-Score	Harmonic mean of Precision and Recall for balanced performance.

📌 Example:

A **medical diagnosis AI model** must be evaluated using **Recall** to ensure it **identifies all disease-positive patients correctly**.

💡 Conclusion:

Each evaluation metric provides unique insights into model performance, helping in **choosing the right metric based on business needs**.

📌 Chapter 2: Understanding Accuracy

🔹 2.1 What is Accuracy?

Accuracy is **the most basic metric** that measures the proportion of correctly predicted instances among the total instances. It is calculated as:

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

Where:

- **TP (True Positives)** = Correctly predicted positive instances.
- **TN (True Negatives)** = Correctly predicted negative instances.
- **FP (False Positives)** = Incorrectly predicted positive instances.
- **FN (False Negatives)** = Incorrectly predicted negative instances.

When to Use Accuracy?

✓ Useful when **classes are balanced** (e.g., spam vs. non-spam classification).
✓ Provides a quick measure of overall performance.

Limitations of Accuracy

✕ Misleading when data is imbalanced.
✕ Fails to distinguish between False Positives and False Negatives.

📌 Example:

A **cancer detection model** has **95% accuracy,** but if **only 5% of cases are cancer-positive,** the model might be predicting **"No Cancer" for all cases,** making it **unreliable**.

💡 **Conclusion:**
Accuracy is **not reliable for imbalanced datasets,** and other metrics like **Precision and Recall** should be used in such cases.

📌 Chapter 3: Understanding Precision

🔹 3.1 What is Precision?

Precision (also called **Positive Predictive Value**) measures how **many predicted positive cases are actually correct**. It is calculated as:

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

When to Use Precision?

✓ Important when **False Positives need to be minimized**.
✓ Used in **spam detection, medical testing, and fraud detection** where a false alarm is costly.

Limitations of Precision

✗ Ignores False Negatives, which may be critical in some applications.

📌 **Example:**
A **fraud detection system** with **high precision** ensures that **non-fraudulent transactions are not mistakenly flagged as fraud**.

💡 **Conclusion:**
Precision is **useful when False Positives need to be minimized,** such as in **spam filters and fraud detection models**.

📌 Chapter 4: Understanding Recall (Sensitivity)

🔹 4.1 What is Recall?

Recall (also called **Sensitivity or True Positive Rate**) measures how **many actual positive cases were correctly identified**. It is calculated as:

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

When to Use Recall?

✓ Important when **False Negatives need to be minimized**.
✓ Used in **medical diagnosis, safety alerts, and crime detection** where missing a real case is risky.

Limitations of Recall

✗ Can be high at the cost of more False Positives.

📌 Example:
A **cancer detection model** with **high recall** ensures that **all cancer patients are correctly identified,** even if some healthy patients are mistakenly flagged.

💡 Conclusion:
Recall is **critical when missing a true case is dangerous,** such as **disease diagnosis or security threat detection**.

📌 Chapter 5: Understanding F1-Score

🔹 5.1 What is F1-Score?

F1-Score is the **harmonic mean of Precision and Recall**, balancing both metrics:

$$F1 - Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

When to Use F1-Score?

✓ Best for **imbalanced datasets** where **both Precision and Recall are important**.
✓ Useful in **fraud detection, search engines, and medical diagnostics**.

📌 **Example:**
A **search engine** ranks pages based on **F1-Score** to ensure that **both relevant and comprehensive pages appear in results**.

💡 **Conclusion:**
F1-Score is **ideal for imbalanced datasets** and provides a **balanced view of Precision and Recall**.

📌 Chapter 6: Choosing the Right Metric

Scenario	Best Metric
Balanced Classes	Accuracy
False Positives Matter	Precision
False Negatives Matter	Recall

Imbalanced Classes	F1-Score

📌 **Example:**
A **hospital AI model** should use **Recall** to **detect all disease cases,** while a **spam filter** should use **Precision** to **minimize false alarms**.

💡 **Conclusion:**
Choosing the right evaluation metric **depends on the problem domain and business needs**.

📌 Chapter 7: Practical Applications of Model Evaluation

🔹 7.1 Real-World Use Cases

✓ **Finance:** Fraud detection models use **Precision** to avoid blocking legitimate transactions.
✓ **Healthcare:** Disease prediction models use **Recall** to ensure no case is missed.
✓ **Marketing:** Ad targeting models use **F1-Score** for balanced predictions.
✓ **E-Commerce:** Product recommendation models use **Precision** to show relevant items.

📌 **Example:**
A **loan approval model** is evaluated using **F1-Score** to balance **approving eligible applicants while minimizing risks**.

💡 **Conclusion:**
Model evaluation is **critical for making AI systems reliable, fair, and effective**.

📌 Summary & Next Steps

✅ Key Takeaways:

✓ Accuracy is simple but misleading in imbalanced datasets.

✓ Precision is useful when False Positives need to be minimized.

✓ Recall is important when False Negatives must be avoided.

✓ F1-Score is best for imbalanced data where both Precision and Recall matter.

📌 Next Steps:

- 🔹 Implement these metrics in Python using Scikit-Learn.
- 🔹 Work on real-world classification problems (Kaggle competitions).
- 🔹 Explore advanced evaluation techniques like AUC-ROC and Confusion Matrices. 🚀

◇ Hyperparameter Tuning & Model Optimization

📌 Chapter 1: Introduction to Hyperparameter Tuning & Model Optimization

🔸 1.1 What is Hyperparameter Tuning?

Hyperparameter tuning is the process of **selecting the best combination of hyperparameters** to improve the performance of a machine learning model. Unlike model parameters (which are learned during training), **hyperparameters are set before training begins** and directly impact model learning.

Difference Between Parameters & Hyperparameters

Feature	Parameters	Hyperparameters
Definition	Learned from data during training.	Set manually before training.
Examples	Weights in neural networks, coefficients in linear regression.	Learning rate, batch size, number of hidden layers.
Impact	Optimized automatically during training.	Needs tuning to improve performance.

📌 Example:

In a **Random Forest model**, the **number of trees (n_estimators)** and **max depth** are hyperparameters, while the **split criteria** for each decision tree node are learned parameters.

💡 **Conclusion:**
Choosing the right hyperparameters significantly impacts a model's **accuracy, generalization ability, and efficiency**.

📌 Chapter 2: Importance of Hyperparameter Tuning in Machine Learning

🔹 2.1 Why Hyperparameter Tuning is Essential?

Hyperparameters control **model complexity, convergence speed, and performance**. Proper tuning can:
✓ Improve **model accuracy**.
✓ Prevent **overfitting (too complex) or underfitting (too simple)**.
✓ Optimize **training time and computational efficiency**.
✓ Enhance **generalization to new data**.

📌 **Example:**
A **deep learning model** with an excessively high learning rate may fail to converge, while a **very small learning rate** might take too long to reach an optimal solution.

💡 **Conclusion:**
Proper hyperparameter tuning helps models **learn efficiently and generalize well** to unseen data.

📌 Chapter 3: Common Hyperparameters in Machine Learning Models

Different machine learning algorithms have **unique hyperparameters** that require optimization.

🔸 3.1 Hyperparameters in Decision Trees & Random Forest

✓ **Max Depth** – Limits how deep the tree can grow (prevents overfitting).
✓ **Min Samples Split** – Minimum samples required to split a node.
✓ **Number of Trees (n_estimators)** – Total number of trees in a Random Forest model.

📌 **Example:**
Setting **max_depth too high** may lead to **overfitting,** while a **very low max_depth** may cause **underfitting**.

🔸 3.2 Hyperparameters in Support Vector Machines (SVM)

✓ **Kernel Type** – Linear, Polynomial, Radial Basis Function (RBF), or Sigmoid.
✓ **C (Regularization Parameter)** – Controls the trade-off between **high accuracy** and **low complexity**.
✓ **Gamma** – Defines influence radius of a training point in RBF kernel.

📌 **Example:**
An **SVM with a high C value** will fit the training data well but might **overfit to noise**.

🔸 3.3 Hyperparameters in Neural Networks & Deep Learning

✓ **Learning Rate** – Controls step size during gradient updates.
✓ **Batch Size** – Number of samples used in one training iteration.
✓ **Number of Hidden Layers & Neurons** – Defines model architecture complexity.

✓ **Dropout Rate** – Prevents overfitting by randomly deactivating neurons during training.

📌 **Example:**
A **learning rate that is too high** may cause the model to oscillate and never converge, while a **low learning rate** may lead to slow convergence.

📌 Chapter 4: Methods for Hyperparameter Tuning

🔸 4.1 Grid Search

Grid Search systematically tests all possible combinations of hyperparameters.

✓ **Pros:** Ensures finding the best parameter combination.
✓ **Cons:** Computationally expensive for large datasets.

📌 **Example (Grid Search in Python using Scikit-Learn):**

```
from sklearn.model_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier


param_grid = {

  'n_estimators': [50, 100, 200],

  'max_depth': [5, 10, None],

  'min_samples_split': [2, 5, 10]

}
```

```
model = RandomForestClassifier()

grid_search = GridSearchCV(model, param_grid, cv=5,
scoring='accuracy')

grid_search.fit(X_train, y_train)


print("Best Parameters:", grid_search.best_params_)
```

💡 **Conclusion:**
Grid Search is **exhaustive but computationally expensive** for complex models.

🔹 4.2 Random Search

Random Search selects random combinations of hyperparameters instead of testing all possibilities.

✓ **Pros:** Faster than Grid Search.
✓ **Cons:** May not always find the absolute best parameters.

📌 **Example (Random Search in Scikit-Learn):**

```
from sklearn.model_selection import RandomizedSearchCV

import numpy as np


param_dist = {

  'n_estimators': np.random.randint(50, 200, 5),

  'max_depth': [None, 5, 10, 20],

  'min_samples_split': [2, 5, 10]
```

```
}


random_search = RandomizedSearchCV(model, param_dist, cv=5, n_iter=10, scoring='accuracy', random_state=42)

random_search.fit(X_train, y_train)


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

💡 **Conclusion:**
Random Search **reduces computation time** while still providing effective hyperparameter tuning.

🔹 4.3 Bayesian Optimization

Bayesian Optimization uses **probability-based methods** to search for optimal hyperparameters efficiently.

✓ **Pros:** Finds optimal solutions **faster** than Grid/Random Search.
✓ **Cons:** More complex to implement.

📌 **Example (Using Optuna for Bayesian Optimization):**

```
import optuna


def objective(trial):

  n_estimators = trial.suggest_int('n_estimators', 50, 200)

  max_depth = trial.suggest_int('max_depth', 3, 20)
```

```
    model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth)
    model.fit(X_train, y_train)

    return model.score(X_val, y_val)

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=20)

print("Best Hyperparameters:", study.best_params)
```

💡 **Conclusion:**
Bayesian Optimization **intelligently narrows down the search space** for hyperparameter tuning.

📌 Chapter 5: Model Optimization Techniques

🔹 5.1 Regularization for Overfitting Prevention

Regularization techniques **prevent overfitting** by adding constraints on model complexity.

✓ **L1 Regularization (Lasso Regression)** – Shrinks coefficients to zero, performing feature selection.
✓ **L2 Regularization (Ridge Regression)** – Penalizes large coefficients to reduce overfitting.
✓ **Elastic Net** – A combination of L1 and L2 regularization.

📌 **Example:**
A **logistic regression model** using L2 regularization prevents overfitting on noisy data.

🔹 5.2 Learning Rate Scheduling for Convergence

✓ **Reduce learning rate over epochs** to improve convergence.
✓ **Use adaptive optimizers (Adam, RMSProp)** for dynamic adjustments.

📌 **Example:**
A **CNN model** trains better when the **learning rate is reduced after every 10 epochs**.

🔹 5.3 Early Stopping for Model Efficiency

✓ Stops training when validation accuracy **stops improving**.
✓ Prevents **overfitting by stopping at the optimal epoch**.

📌 **Example (Using Early Stopping in Keras):**

```
from tensorflow.keras.callbacks import EarlyStopping


early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=50, callbacks=[early_stopping])
```

💡 Conclusion:

Early stopping **prevents unnecessary computation and improves model generalization**.

📌 Summary & Next Steps

✅ Key Takeaways:

✓ Hyperparameter tuning enhances model accuracy and generalization.

✓ Techniques like Grid Search, Random Search, and Bayesian Optimization improve efficiency.

✓ Regularization, early stopping, and adaptive learning rates optimize model performance.

📌 Next Steps:

- 🔹 Experiment with different tuning techniques in real-world datasets.
- 🔹 Explore AutoML for automated hyperparameter optimization.
- 🔹 Stay updated with new advancements in AI model tuning. 🚀

📌 ASSIGNMENT 1:

☑ BUILD A HOUSE PRICE PREDICTION MODEL USING LINEAR REGRESSION.

📌 Solution: House Price Prediction Model using Linear Regression

🎯 Objective:

The goal of this assignment is to **build a house price prediction model using Linear Regression**. We will use **Python and Scikit-Learn** to train the model on a dataset containing house features like **size, number of bedrooms, location, and price**, then evaluate its performance.

🛠 Step 1: Install & Import Required Libraries

Before starting, ensure that you have the required libraries installed. If not, run:

pip install pandas numpy scikit-learn matplotlib seaborn

Now, import the necessary libraries:

```
# Import required 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_absolute_error, mean_squared_error, r2_score
```

🛠 Step 2: Load the House Price Dataset

For this assignment, we will use a real-world **house price dataset** (e.g., from **Kaggle** or **public repositories**). If using a CSV file:

Load the dataset

df = pd.read_csv("house_prices.csv")

Display first five rows

df.head()

📌 Dataset Description:

- **Size (sq ft)** – Size of the house in square feet.
- **Bedrooms** – Number of bedrooms in the house.
- **Bathrooms** – Number of bathrooms in the house.
- **Location** – City/Neighborhood of the house.
- **Price (Target Variable)** – Selling price of the house.

🛠 Step 3: Data Preprocessing & Exploration

🔹 3.1 Check for Missing Values

Before training the model, check for missing values and handle them appropriately.

Check for missing values

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

Fill missing values (if any)

df.fillna(df.median(), inplace=True)

📌 **Why?**
Handling missing values ensures that our model is trained on **complete and reliable data**.

🔹 **3.2 Check for Duplicates**

Check for duplicate rows

print(f"Duplicate rows: {df.duplicated().sum()}")

Remove duplicate records

df.drop_duplicates(inplace=True)

📌 **Why?**
Duplicate records can **bias model predictions** and should be removed.

🔹 **3.3 Data Visualization (Understanding Data Relationships)**

1️⃣ House Size vs. Price

plt.figure(figsize=(8,5))

sns.scatterplot(x=df["Size (sq ft)"], y=df["Price"])

plt.title("House Size vs Price")

plt.xlabel("Size (sq ft)")

plt.ylabel("Price")

plt.show()

📌 **Insight:** Larger houses tend to have **higher prices**.

2️⃣ Number of Bedrooms vs. Price

```
plt.figure(figsize=(8,5))
sns.boxplot(x=df["Bedrooms"], y=df["Price"])
plt.title("Number of Bedrooms vs Price")
plt.show()
```

📌 **Insight:** Houses with **more bedrooms** tend to have **higher prices**, but **the effect is not always linear**.

🛠️ Step 4: Feature Selection & Encoding

🔹 4.1 Select Relevant Features

```
# Select relevant columns
df = df[['Size (sq ft)', 'Bedrooms', 'Bathrooms', 'Price']]
```

📌 **Why?**
We only include features that **directly impact house prices**.

🔹 4.2 Splitting the Dataset into Training & Testing Sets

```
# Define input features (X) and target variable (y)
X = df[['Size (sq ft)', 'Bedrooms', 'Bathrooms']]
y = df['Price']


# Split data into training (80%) and testing (20%)
```

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


# Display shape of training and testing sets

X_train.shape, X_test.shape
```

📌 **Why?**
Splitting the data ensures that we **train the model on one portion and test its performance on unseen data**.

🛠️ Step 5: Train the Linear Regression Model

🔹 5.1 Initialize and Train the Model

```
# Initialize the Linear Regression model

model = LinearRegression()


# Train the model

model.fit(X_train, y_train)
```

📌 **What Happens Here?**

- The **model learns the relationship** between house features and price.
- It **finds the best-fit line** that minimizes the prediction error.

🔹 5.2 Get Model Coefficients & Intercept

```
# Display the coefficients and intercept
```

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

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

📌 **Interpretation:**

- The **intercept** represents the base price when all features are zero.
- The **coefficients** represent the change in house price for **each unit increase** in a feature (e.g., if Bedrooms coefficient is 50,000, then adding 1 bedroom increases price by $50,000).

🛠️ Step 6: Model Evaluation

🔹 6.1 Make Predictions on the Test Data

```
# Make predictions on test data

y_pred = model.predict(X_test)
```

🔹 6.2 Evaluate Model Performance

```
# Calculate Mean Absolute Error (MAE)

mae = mean_absolute_error(y_test, y_pred)


# Calculate Mean Squared Error (MSE)

mse = mean_squared_error(y_test, y_pred)


# Calculate R-squared score

r2 = r2_score(y_test, y_pred)
```

```
# Print model performance metrics
print(f"Mean Absolute Error: {mae}")
print(f"Mean Squared Error: {mse}")
print(f"R-squared Score: {r2}")
```

📌 **Model Performance Insights:**

- **Low MAE & MSE** = Better model accuracy.
- **R-squared Score (~0.7 - 1.0)** = Indicates **how well the model explains price variations**.

🔹 6.3 Visualize Model Predictions vs. Actual Prices

```
plt.figure(figsize=(8,5))
sns.scatterplot(x=y_test, y=y_pred, color='blue')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')
plt.xlabel("Actual Price")
plt.ylabel("Predicted Price")
plt.title("Actual vs Predicted House Prices")
plt.show()
```

📌 **Interpretation:**

- If the points **align well with the red line,** the model is making accurate predictions.
- Large deviations indicate **errors that may require model improvements**.

🛠 Step 7: Improving the Model

🔹 7.1 Feature Engineering (Adding More Predictive Features)

To improve accuracy, we can include additional variables:

- **Location (Convert categorical city/neighborhood data into numerical form).**
- **Property Age (Older houses may be cheaper).**
- **Proximity to amenities (Schools, Hospitals, Parks).**

🔹 7.2 Try Polynomial Regression (For Non-Linear Trends)

```
from sklearn.preprocessing import PolynomialFeatures

# Transform features into polynomial terms
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

# Train a new model
model_poly = LinearRegression()
model_poly.fit(X_train_poly, y_train)

# Evaluate new model
y_pred_poly = model_poly.predict(X_test_poly)
```

```
print("New Model R2 Score:", r2_score(y_test, y_pred_poly))
```

📌 **Why?**

Polynomial Regression captures non-linear relationships between house features and price.

✅ SUMMARY OF STEPS PERFORMED:

1. **Loaded and cleaned the dataset (handled missing values, removed duplicates).**
2. **Visualized key relationships (House size vs. Price, Bedrooms vs. Price).**
3. **Split the data into training & testing sets.**
4. **Trained a Linear Regression model to predict house prices.**
5. **Evaluated model performance using MAE, MSE, and R^2 score.**
6. **Plotted actual vs. predicted prices to assess accuracy.**
7. **Explored improvements (Feature Engineering & Polynomial Regression).**

📌 **Next Steps:**

- 🔹 **Test the model on new house listings.**
- 🔹 **Deploy the model using Flask or Streamlit for real-time predictions.**
- 🔹 **Optimize feature selection for better accuracy.**

📌 ASSIGNMENT 2:

☑ IMPLEMENT A SPAM EMAIL CLASSIFIER USING LOGISTIC REGRESSION.

📌 Solution: Implementing a Spam Email Classifier using Logistic Regression

- **Objective**

The goal of this assignment is to **build a spam email classifier** using **Logistic Regression**. We will use **Natural Language Processing (NLP) techniques** to process the text data and classify emails as **spam or not spam**.

- **Step 1: Import Required Libraries**

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import re

import string


from sklearn.model_selection import train_test_split

from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

from nltk.corpus import stopwords

from nltk.tokenize import word_tokenize

import nltk

nltk.download('stopwords')

nltk.download('punkt')
```

- **Step 2: Load and Explore the Dataset**

For this example, we will use the **SMS Spam Collection Dataset**. If you don't have a dataset, you can download it from **Kaggle** or use a similar dataset.

```
# Load the dataset

df = pd.read_csv("spam.csv", encoding="latin-1")


# Display first few rows

df.head()
```

Understanding the Dataset Structure

The dataset consists of two main columns:
✓ v1 (Label: "ham" for non-spam, "spam" for spam)
✓ v2 (The actual email/message text)

```
# Rename columns for better readability

df = df[['v1', 'v2']]
```

```
df.columns = ['label', 'message']


# Check dataset size
print(f"Dataset contains {df.shape[0]} messages")
```

- **Step 3: Data Preprocessing**

Convert Labels to Binary (Spam = 1, Ham = 0)

```
# Convert label to binary values: 'spam' -> 1, 'ham' -> 0
df['label'] = df['label'].map({'ham': 0, 'spam': 1})


# Check class distribution
df['label'].value_counts()
```

Text Cleaning Function

We need to clean the text before applying machine learning. The cleaning steps include:

✓ Removing punctuation and special characters.
✓ Converting text to lowercase.
✓ Removing stopwords (words like "the", "and", "is").
✓ Tokenizing words.

```
# Define function to clean text data
def clean_text(text):
    text = text.lower()  # Convert to lowercase
```

```
    text = re.sub(r'\d+', '', text)  # Remove numbers
    text = text.translate(str.maketrans("", "", string.punctuation))  # Remove punctuation
    tokens = word_tokenize(text)  # Tokenize words
    text = [word for word in tokens if word not in stopwords.words('english')]  # Remove stopwords
    return " ".join(text)


# Apply text cleaning to messages
df['clean_message'] = df['message'].apply(clean_text)


# Display first few cleaned messages
df[['message', 'clean_message']].head()
```

- **Step 4: Convert Text into Numerical Features (TF-IDF Vectorization)**

Why TF-IDF?

✓ **Term Frequency-Inverse Document Frequency (TF-IDF)** helps identify the most important words in an email.

✓ It transforms raw text into **numerical feature vectors** that machine learning models can understand.

```
# Convert text into numerical features using TF-IDF Vectorization
tfidf_vectorizer = TfidfVectorizer(max_features=3000)  # Limit to 3000 important words
```

```
X = tfidf_vectorizer.fit_transform(df['clean_message'])


# Convert sparse matrix to array

X = X.toarray()

y = df['label'].values  # Labels (spam=1, ham=0)
```

- **Step 5: Split Data into Training & Testing Sets**

We split our dataset into **80% training data and 20% testing data** to evaluate model performance.

```
# Split data into training and test sets (80-20 split)

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


# Print dataset sizes

print(f"Training set size: {X_train.shape[0]} emails")

print(f"Test set size: {X_test.shape[0]} emails")
```

- **Step 6: Train the Logistic Regression Model**

Why Logistic Regression?

✓ Logistic Regression is widely used for **binary classification problems** (spam vs. non-spam).

✓ It provides **probability scores** for predictions.

✓ It is computationally **fast and efficient**.

```
# Initialize Logistic Regression model
model = LogisticRegression()

# Train the model on training data
model.fit(X_train, y_train)
```

- **Step 7: Make Predictions & Evaluate the Model**

Predict on Test Data

```
# Make predictions
y_pred = model.predict(X_test)
```

Evaluate Model Performance

```
# Calculate Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")

# Classification Report
print("Classification Report:\n", classification_report(y_test, y_pred))

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
```

```
# Visualizing Confusion Matrix

plt.figure(figsize=(6,4))

sns.heatmap(conf_matrix, annot=True, cmap="Blues", fmt="d",
xticklabels=['Ham', 'Spam'], yticklabels=['Ham', 'Spam'])

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()
```

Understanding the Confusion Matrix

✓ **True Positives (TP)** – Spam emails correctly classified as spam.
✓ **True Negatives (TN)** – Non-spam emails correctly classified as ham.
✓ **False Positives (FP)** – Non-spam emails incorrectly classified as spam.
✓ **False Negatives (FN)** – Spam emails incorrectly classified as ham.

High accuracy and a low false positive rate indicate a strong classifier.

- **Step 8: Test with New Emails**

Now, let's test the model with some new emails:

```
def predict_spam(email):
    clean_email = clean_text(email)  # Clean the email
```

```
    email_tfidf = tfidf_vectorizer.transform([clean_email])  # Convert to TF-IDF

    prediction = model.predict(email_tfidf)[0]  # Predict

    return "Spam" if prediction == 1 else "Not Spam"


# Test cases

test_email_1 = "Congratulations! You won a $500 gift card. Click the link now!"

test_email_2 = "Hi, please find the attached report for the last quarter."


print(f"Email 1: {predict_spam(test_email_1)}")

print(f"Email 2: {predict_spam(test_email_2)}")
```

📌 **Expected Output:**

✓ Email 1 → Spam

✓ Email 2 → Not Spam

📌 Summary & Next Steps

✅ **Key Takeaways:**

✓ We **cleaned and preprocessed text data** by removing punctuation, stopwords, and tokenizing words.

✓ We **converted text into numerical form** using **TF-IDF vectorization**.

✓ We trained a **Logistic Regression model** for spam detection.

✓ We **evaluated model performance** using accuracy, confusion matrix, and classification report.

✓ We **tested the classifier** with real-world email examples.

📌 **Next Steps:**

🔹 **Improve accuracy using advanced models like Naïve Bayes, SVM, or Deep Learning.**

🔹 **Use Word Embeddings (Word2Vec, BERT) for better text representation.**

🔹 **Deploy the model using Flask or Streamlit for real-time email classification.** 🚀