# ITA0658-192212469L

**\*\*1.Find S-Algorithm\*\***

import pandas as pd

def find\_s\_algorithm(training\_data):

# Initialize the most specific hypothesis

hypothesis = ["ϕ"] \* (len(training\_data.columns) - 1) # Exclude target column

for index, row in training\_data.iterrows():

if row.iloc[-1] == "Yes": # Fix: Use .iloc[-1] instead of row[-1]

for i in range(len(hypothesis)):

if hypothesis[i] == "ϕ":

hypothesis[i] = row.iloc[i] # Fix: Use .iloc[i] instead of row[i]

elif hypothesis[i] != row.iloc[i]:

hypothesis[i] = "?" # Generalize when attributes differ

return hypothesis

# Sample dataset (Outlook, Temperature, Humidity, Wind, PlayTennis)

data = [

["Sunny", "Warm", "Normal", "Strong", "Yes"],

["Sunny", "Warm", "High", "Strong", "Yes"],

["Rainy", "Cold", "High", "Strong", "No"],

["Sunny", "Warm", "High", "Weak", "Yes"]

]

# Convert data to DataFrame

columns = ["Outlook", "Temperature", "Humidity", "Wind", "PlayTennis"]

training\_data = pd.DataFrame(data, columns=columns)

# Apply Find-S Algorithm

final\_hypothesis = find\_s\_algorithm(training\_data)

print("Most Specific Hypothesis Found:")

print(final\_hypothesis)

**\*\*2.Candidate Elimination Algorithm\*\***

import pandas as pd

def candidate\_elimination(training\_data):

num\_attributes = len(training\_data.columns) - 1 # Exclude target column

# Initialize S-hypothesis with the first positive example

S = None

for index, row in training\_data.iterrows():

if row.iloc[-1] == "Yes":

S = list(row.iloc[:-1])

break

# If no positive examples exist, return empty hypotheses

if S is None:

return None, None

# Initialize G-hypothesis with the most general hypothesis

G = [["?"] \* num\_attributes]

# Process each example in the dataset

for index, row in training\_data.iterrows():

instance = list(row.iloc[:-1])

label = row.iloc[-1]

if label == "Yes": # Positive example

for i in range(num\_attributes):

if S[i] != instance[i]:

S[i] = "?" # Generalize S to be consistent with this positive example

# Remove inconsistent hypotheses from G

G = [g for g in G if all(g[i] == "?" or g[i] == instance[i] for i in range(num\_attributes))]

elif label == "No": # Negative example

new\_G = []

for g in G:

for i in range(num\_attributes):

if g[i] == "?":

for value in set(training\_data.iloc[:, i]): # Consider all values for that attribute

if value != instance[i]:

new\_hypothesis = g[:]

new\_hypothesis[i] = value

new\_G.append(new\_hypothesis)

G = new\_G

G = [g for g in G if all(g[i] == "?" or g[i] != instance[i] for i in range(num\_attributes))]

return S, G

# Sample dataset (Outlook, Temperature, Humidity, Wind, PlayTennis)

data = [

["Sunny", "Warm", "Normal", "Strong", "Yes"],

["Sunny", "Warm", "High", "Strong", "Yes"],

["Rainy", "Cold", "High", "Strong", "No"],

["Sunny", "Warm", "High", "Weak", "Yes"]

]

# Convert data to DataFrame

columns = ["Outlook", "Temperature", "Humidity", "Wind", "PlayTennis"]

training\_data = pd.DataFrame(data, columns=columns)

# Apply Candidate Elimination Algorithm

S\_final, G\_final = candidate\_elimination(training\_data)

print("\nMost Specific Hypothesis (S):", S\_final)

print("Most General Hypothesis (G):", G\_final)

**\*\*3.DT ID3 Algorithm\*\***

import pandas as pd

import numpy as np

import math

# Function to calculate entropy

def entropy(data):

target = data.keys()[-1] # Get target column

values = data[target].unique()

entropy\_value = 0

for value in values:

fraction = data[target].value\_counts()[value] / len(data[target])

entropy\_value -= fraction \* math.log2(fraction)

return entropy\_value

# Function to calculate information gain

def info\_gain(data, attribute):

total\_entropy = entropy(data)

values = data[attribute].unique()

weighted\_entropy = 0

for value in values:

subset = data[data[attribute] == value]

weighted\_entropy += (len(subset) / len(data)) \* entropy(subset)

return total\_entropy - weighted\_entropy

# Function to find the best attribute

def best\_attribute(data):

attributes = data.keys()[:-1] # Exclude target column

gains = {attr: info\_gain(data, attr) for attr in attributes}

return max(gains, key=gains.get) # Attribute with highest information gain

# Function to build decision tree

def id3(data, tree=None):

target = data.keys()[-1] # Target column

# If only one class remains in target, return that class

if len(data[target].unique()) == 1:

return data[target].unique()[0]

# If no more attributes, return the most common class

if len(data.columns) == 1:

return data[target].mode()[0]

# Find the best attribute to split on

best = best\_attribute(data)

# Create tree with best attribute

if tree is None:

tree = {}

tree[best] = {}

# Split dataset and build subtrees

for value in data[best].unique():

subset = data[data[best] == value].drop(columns=[best]) # Remove split attribute

subtree = id3(subset)

tree[best][value] = subtree

return tree

# Sample dataset (PlayTennis)

data = {

"Outlook": ["Sunny", "Sunny", "Overcast", "Rainy", "Rainy", "Rainy", "Overcast", "Sunny", "Sunny", "Rainy", "Sunny", "Overcast", "Overcast", "Rainy"],

"Temperature": ["Hot", "Hot", "Hot", "Mild", "Cool", "Cool", "Cool", "Mild", "Cool", "Mild", "Mild", "Mild", "Hot", "Mild"],

"Humidity": ["High", "High", "High", "High", "Normal", "Normal", "Normal", "High", "Normal", "Normal", "Normal", "High", "Normal", "High"],

"Wind": ["Weak", "Strong", "Weak", "Weak", "Weak", "Strong", "Strong", "Weak", "Weak", "Weak", "Strong", "Strong", "Weak", "Strong"],

"PlayTennis": ["No", "No", "Yes", "Yes", "Yes", "No", "Yes", "No", "Yes", "Yes", "Yes", "Yes", "Yes", "No"]

}

# Convert data to DataFrame

df = pd.DataFrame(data)

# Build decision tree using ID3

decision\_tree = id3(df)

# Print the decision tree

import pprint

pprint.pprint(decision\_tree)

**\*\*4.Artificial Neural Network \*\***

import numpy as np

# Sigmoid activation function and its derivative

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

return x \* (1 - x)

# Initialize dataset (X: inputs, y: outputs)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Input data (XOR problem)

y = np.array([[0], [1], [1], [0]]) # Expected output

# Initialize neural network parameters

input\_neurons = 2 # Input layer neurons

hidden\_neurons = 3 # Hidden layer neurons

output\_neurons = 1 # Output layer neurons

# Randomly initialize weights and biases

np.random.seed(42)

weights\_input\_hidden = np.random.uniform(size=(input\_neurons, hidden\_neurons))

bias\_hidden = np.random.uniform(size=(1, hidden\_neurons))

weights\_hidden\_output = np.random.uniform(size=(hidden\_neurons, output\_neurons))

bias\_output = np.random.uniform(size=(1, output\_neurons))

# Training parameters

epochs = 10000

learning\_rate = 0.5

# Training the neural network

for epoch in range(epochs):

# Forward propagation

hidden\_layer\_input = np.dot(X, weights\_input\_hidden) + bias\_hidden

hidden\_layer\_output = sigmoid(hidden\_layer\_input)

output\_layer\_input = np.dot(hidden\_layer\_output, weights\_hidden\_output) + bias\_output

predicted\_output = sigmoid(output\_layer\_input)

# Compute error

error = y - predicted\_output

# Backpropagation

d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output)

d\_hidden\_layer = d\_predicted\_output.dot(weights\_hidden\_output.T) \* sigmoid\_derivative(hidden\_layer\_output)

# Update weights and biases

weights\_hidden\_output += hidden\_layer\_output.T.dot(d\_predicted\_output) \* learning\_rate

bias\_output += np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learning\_rate

weights\_input\_hidden += X.T.dot(d\_hidden\_layer) \* learning\_rate

bias\_hidden += np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* learning\_rate

# Print error at intervals

if epoch % 1000 == 0:

print(f"Epoch {epoch}, Error: {np.mean(np.abs(error))}")

# Final output after training

print("\nFinal Output After Training:")

print(predicted\_output)

**\*\*5.K-NN Algorithm\*\***

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

# Load the Iris dataset

iris = load\_iris()

X = iris.data # Features (4 features)

y = iris.target # Labels (3 classes)

# Select only the first two features for visualization

X = X[:, :2] # Taking only the first two features

# Split dataset into training and testing sets (80% train, 20% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Initialize KNN classifier with K=5

k = 5

knn = KNeighborsClassifier(n\_neighbors=k)

# Train the model using only two features

knn.fit(X\_train, y\_train)

# Predict on the test set

y\_pred = knn.predict(X\_test)

# Compute accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"KNN Model Accuracy: {accuracy \* 100:.2f}%")

# Plot decision boundaries

def plot\_decision\_boundary(X, y, model, k):

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

# Predict class for each point in the meshgrid

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)

scatter = plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors="k", cmap=plt.cm.Paired)

plt.title(f"KNN Decision Boundary (K={k})")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.colorbar(scatter)

plt.show()

# Call the function to plot the decision boundary

plot\_decision\_boundary(X\_train, y\_train, knn, k)

**\*\*6.Logistic Regression Algorithm\*\***

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# Load the Iris dataset

iris = load\_iris()

X = iris.data[:, :2] # Using only first two features for visualization

y = iris.target # Labels (0, 1, or 2)

# Convert multi-class to binary classification (only classes 0 and 1 for visualization)

X = X[y < 2]

y = y[y < 2]

# Split dataset into training and testing sets (80% train, 20% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize features (important for Logistic Regression)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Initialize and train Logistic Regression model

log\_reg = LogisticRegression()

log\_reg.fit(X\_train, y\_train)

# Predict on test set

y\_pred = log\_reg.predict(X\_test)

# Compute accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Logistic Regression Model Accuracy: {accuracy \* 100:.2f}%")

# Function to plot decision boundary

def plot\_decision\_boundary(X, y, model):

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

# Predict class for each point in the meshgrid

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)

scatter = plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors="k", cmap=plt.cm.Paired)

plt.title("Logistic Regression Decision Boundary")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.colorbar(scatter)

plt.show()

# Plot the decision boundary

plot\_decision\_boundary(X\_train, y\_train, log\_reg)

**\*\*7.Linear regression Algorithm\*\***

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.datasets import make\_regression

# Generate a synthetic dataset (100 samples, 1 feature, with noise)

X, y = make\_regression(n\_samples=100, n\_features=1, noise=10, random\_state=42)

# Split dataset into training (80%) and testing (20%) sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train the Linear Regression model

lin\_reg = LinearRegression()

lin\_reg.fit(X\_train, y\_train)

# Predict on the test set

y\_pred = lin\_reg.predict(X\_test)

# Model evaluation

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Mean Squared Error: {mse:.2f}")

print(f"R² Score: {r2:.2f}")

# Visualization of the Regression Line

plt.scatter(X\_test, y\_test, color='blue', label="Actual Data")

plt.plot(X\_test, y\_pred, color='red', linewidth=2, label="Regression Line")

plt.xlabel("Feature Value")

plt.ylabel("Target Value")

plt.title("Linear Regression Model")

plt.legend()

plt.show()

**\*\*8.Linear and polynomial regression Algorithm\*\***

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.preprocessing import PolynomialFeatures

from sklearn.metrics import mean\_squared\_error, r2\_score

# Generate synthetic dataset (change to real dataset if needed)

np.random.seed(42)

X = np.linspace(0, 10, 100).reshape(-1, 1)

y = 3 \* X\*\*3 + 2 \* X\*\*2 + 7 \* X + 5 + np.random.normal(0, 100, (100, 1)) # Polynomial relation with noise

# Split dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, shuffle=True)

# --- LINEAR REGRESSION ---

linear\_reg = LinearRegression()

linear\_reg.fit(X\_train, y\_train)

y\_pred\_linear = linear\_reg.predict(X\_test)

# --- POLYNOMIAL REGRESSION (Degree = 3) ---

poly = PolynomialFeatures(degree=3)

X\_poly\_train = poly.fit\_transform(X\_train)

X\_poly\_test = poly.transform(X\_test)

poly\_reg = LinearRegression()

poly\_reg.fit(X\_poly\_train, y\_train)

y\_pred\_poly = poly\_reg.predict(X\_poly\_test)

# --- MODEL PERFORMANCE ---

mse\_linear = mean\_squared\_error(y\_test, y\_pred\_linear)

r2\_linear = r2\_score(y\_test, y\_pred\_linear)

mse\_poly = mean\_squared\_error(y\_test, y\_pred\_poly)

r2\_poly = r2\_score(y\_test, y\_pred\_poly)

print(f"Linear Regression - MSE: {mse\_linear:.2f}, R² Score: {r2\_linear:.2f}")

print(f"Polynomial Regression (Degree 3) - MSE: {mse\_poly:.2f}, R² Score: {r2\_poly:.2f}")

# --- VISUALIZATION ---

plt.scatter(X\_test, y\_test, color='blue', label="Actual Data")

plt.plot(X\_test, y\_pred\_linear, color='red', linewidth=2, label="Linear Regression")

# For smooth polynomial curve

X\_curve = np.linspace(0, 10, 100).reshape(-1, 1)

X\_curve\_poly = poly.transform(X\_curve)

y\_curve\_poly = poly\_reg.predict(X\_curve\_poly)

plt.plot(X\_curve, y\_curve\_poly, color='green', linewidth=2, label="Polynomial Regression (Degree 3)")

plt.xlabel("Feature Value")

plt.ylabel("Target Value")

plt.title("Linear vs Polynomial Regression")

plt.legend()

plt.show()

**\*\*9.Estimation and Maximation Algorithm\*\***

import numpy as np

import matplotlib.pyplot as plt

from sklearn.mixture import GaussianMixture

from sklearn.datasets import make\_blobs

# Generate synthetic dataset

np.random.seed(42)

X, \_ = make\_blobs(n\_samples=300, centers=3, cluster\_std=1.0, random\_state=42)

# Fit Gaussian Mixture Model (GMM) using Expectation-Maximization

gmm = GaussianMixture(n\_components=3, max\_iter=100, random\_state=42)

gmm.fit(X)

# Predict cluster assignments

labels = gmm.predict(X)

# Get cluster centers

centers = gmm.means\_

# Plot clusters

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', alpha=0.6)

plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x', s=200, label="Centers")

plt.title("Gaussian Mixture Model (EM Algorithm)")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()

# Print GMM parameters

print("Estimated Means:\n", gmm.means\_)

print("Estimated Variances:\n", gmm.covariances\_)

print("Estimated Weights:\n", gmm.weights\_)

**\*\*10. program for the task of credit score classification\*\***

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.ensemble import RandomForestClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

# 1️⃣ Load synthetic dataset (Replace with real credit score dataset)

data = pd.DataFrame({

'Age': np.random.randint(20, 70, 1000),

'Income': np.random.randint(20000, 150000, 1000),

'Debt': np.random.randint(1000, 50000, 1000),

'Credit\_History': np.random.choice([0, 1], 1000),

'Loan\_Amount': np.random.randint(5000, 50000, 1000),

'Credit\_Score': np.random.choice([0, 1], 1000) # 1 = Good, 0 = Bad

})

# 2️⃣ Preprocessing: No missing values, normalize numerical features

X = data.drop("Credit\_Score", axis=1)

y = data["Credit\_Score"]

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# 3️⃣ Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

# 4️⃣ Train models: Logistic Regression & Random Forest

log\_reg = LogisticRegression()

log\_reg.fit(X\_train, y\_train)

rf\_clf = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_clf.fit(X\_train, y\_train)

# 5️⃣ Predictions

y\_pred\_log = log\_reg.predict(X\_test)

y\_pred\_rf = rf\_clf.predict(X\_test)

# 6️⃣ Model Evaluation

print("Logistic Regression Accuracy:", accuracy\_score(y\_test, y\_pred\_log))

print(classification\_report(y\_test, y\_pred\_log))

print("Random Forest Accuracy:", accuracy\_score(y\_test, y\_pred\_rf))

print(classification\_report(y\_test, y\_pred\_rf))

# 7️⃣ Confusion Matrix

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

sns.heatmap(confusion\_matrix(y\_test, y\_pred\_log), annot=True, fmt='d', cmap="Blues")

plt.title("Logistic Regression - Confusion Matrix")

plt.subplot(1, 2, 2)

sns.heatmap(confusion\_matrix(y\_test, y\_pred\_rf), annot=True, fmt='d', cmap="Greens")

plt.title("Random Forest - Confusion Matrix")

plt.show()

**\*\*11.Iris flower classification using KNN\*\***

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

# 1️⃣ Load the Iris dataset

iris = datasets.load\_iris()

X = iris.data # Features: sepal length, sepal width, petal length, petal width

y = iris.target # Labels: 0 - Setosa, 1 - Versicolor, 2 - Virginica

labels = iris.target\_names

# 2️⃣ Preprocessing: Normalize features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# 3️⃣ Train-Test Split (80% Training, 20% Testing)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

# 4️⃣ Train KNN Classifier (Choose k=5)

k = 5

knn = KNeighborsClassifier(n\_neighbors=k)

knn.fit(X\_train, y\_train)

# 5️⃣ Predictions

y\_pred = knn.predict(X\_test)

# 6️⃣ Evaluate Model Performance

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"KNN Model Accuracy: {accuracy \* 100:.2f}%")

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred, target\_names=labels))

# 7️⃣ Confusion Matrix

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

sns.heatmap(confusion\_matrix(y\_test, y\_pred), annot=True, cmap="Blues", xticklabels=labels, yticklabels=labels)

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix for KNN on Iris Dataset")

plt.show()

**\*\*12.Car price prediction using Python\*\***

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

# 1️⃣ Load Dataset (Ensure file is present in the directory)

file\_path = "car\_data.csv" # Change this to your correct file path

try:

df = pd.read\_csv(file\_path)

print("✅ Dataset loaded successfully!")

except FileNotFoundError:

print("❌ File not found: car\_data.csv. Please check the file path and try again.")

exit()

# 2️⃣ View Data

print(df.head()) # Display first few rows

# 3️⃣ Data Preprocessing: Handle Missing Values

df.dropna(inplace=True)

# 🔹 Select Features and Target Variable

X = df[['year', 'mileage', 'engine\_size']] # Choose relevant numerical features

y = df['price'] # Target variable

# 4️⃣ 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)

# 5️⃣ Feature Scaling (Standardization)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# 6️⃣ Train Linear Regression Model

model = LinearRegression()

model.fit(X\_train\_scaled, y\_train)

# 7️⃣ Make Predictions

y\_pred = model.predict(X\_test\_scaled)

# 8️⃣ Evaluate Model Performance

mae = mean\_absolute\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"\n📊 Model Performance:")

print(f"Mean Absolute Error (MAE): {mae:.2f}")

print(f"Mean Squared Error (MSE): {mse:.2f}")

print(f"R² Score: {r2:.2f}")

# 9️⃣ Visualizing Predictions vs. Actual

plt.figure(figsize=(8, 6))

sns.scatterplot(x=y\_test, y=y\_pred, alpha=0.6)

plt.xlabel("Actual Prices")

plt.ylabel("Predicted Prices")

plt.title("Car Price Prediction: Actual vs Predicted")

plt.show()

**\*\*16.house price prediction\*\***

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.preprocessing import StandardScaler

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

# 1️⃣ 🔹 Create a Synthetic Dataset (No External File Required)

np.random.seed(42)

num\_samples = 500

# Generate random data for features

square\_feet = np.random.randint(800, 4000, num\_samples)

num\_bedrooms = np.random.randint(1, 6, num\_samples)

num\_bathrooms = np.random.randint(1, 4, num\_samples)

location\_score = np.random.uniform(1, 10, num\_samples) # Location rating (1-10)

# Generate house prices with some randomness

price = (square\_feet \* 150) + (num\_bedrooms \* 50000) + (num\_bathrooms \* 30000) + (location\_score \* 10000) + np.random.randint(-50000, 50000, num\_samples)

# Create DataFrame

df = pd.DataFrame({

'square\_feet': square\_feet,

'num\_bedrooms': num\_bedrooms,

'num\_bathrooms': num\_bathrooms,

'location\_score': location\_score,

'price': price

})

print("✅ Dataset Created Successfully!")

print(df.head())

# 2️⃣ 🔹 Select Features and Target

X = df[['square\_feet', 'num\_bedrooms', 'num\_bathrooms', 'location\_score']]

y = df['price']

# 3️⃣ 🔹 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)

# 4️⃣ 🔹 Feature Scaling

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# 5️⃣ 🔹 Train Linear Regression Model

model = LinearRegression()

model.fit(X\_train\_scaled, y\_train)

# 6️⃣ 🔹 Make Predictions

y\_pred = model.predict(X\_test\_scaled)

# 7️⃣ 🔹 Evaluate Model Performance

mae = mean\_absolute\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("\n📊 Model Performance:")

print(f"Mean Absolute Error (MAE): {mae:.2f}")

print(f"Mean Squared Error (MSE): {mse:.2f}")

print(f"R² Score: {r2:.2f}")

# 8️⃣ 🔹 Visualizing Predictions vs. Actual Prices

plt.figure(figsize=(8, 6))

sns.scatterplot(x=y\_test, y=y\_pred, alpha=0.6)

plt.xlabel("Actual Prices")

plt.ylabel("Predicted Prices")

plt.title("House Price Prediction: Actual vs Predicted")

plt.show()

\*\*17.Navie Bayes Algorithm\*\*

# 1️⃣ Import Required Libraries

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

from sklearn.datasets import load\_iris

# 2️⃣ Load the Iris Dataset

iris = load\_iris()

df = pd.DataFrame(iris.data, columns=iris.feature\_names)

df['species'] = iris.target # Add target labels

species\_map = {0: 'Setosa', 1: 'Versicolor', 2: 'Virginica'}

df['species'] = df['species'].map(species\_map)

print("✅ Dataset Loaded Successfully!")

print(df.head())

# 3️⃣ Define Features and Target

X = df.iloc[:, :-1] # All columns except 'species'

y = df.iloc[:, -1] # Target column 'species'

# 4️⃣ Split the Data (80% Train, 20% Test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 5️⃣ Feature Scaling (Optional for Naïve Bayes)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# 6️⃣ Train the Naïve Bayes Model

model = GaussianNB()

model.fit(X\_train\_scaled, y\_train)

# 7️⃣ Make Predictions

y\_pred = model.predict(X\_test\_scaled)

# 8️⃣ Evaluate Model Performance

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"\n📊 Model Accuracy: {accuracy \* 100:.2f}%")

print("\n🔹 Classification Report:")

print(classification\_report(y\_test, y\_pred))

# 9️⃣ Confusion Matrix Visualization

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

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

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=species\_map.values(), yticklabels=species\_map.values())

plt.xlabel("Predicted Label")

plt.ylabel("Actual Label")

plt.title("Confusion Matrix - Naïve Bayes")

plt.show()

**\*\*18.Mobile price classification\*\***

# 1️⃣ Import Required Libraries

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

# 2️⃣ Load the Dataset (Ensure the file is in the correct directory)

try:

df = pd.read\_csv("mobile\_price.csv") # Replace with actual dataset path

print("✅ Dataset Loaded Successfully!")

except FileNotFoundError:

print("❌ File not found: mobile\_price.csv. Please check the file path and try again.")

exit()

# 3️⃣ Display Dataset Information

print(df.head()) # View the first 5 rows

print(df.info()) # Dataset summary

print(df.describe()) # Statistical details

# 4️⃣ Define Features (X) and Target (y)

X = df.drop(columns=["price\_range"]) # Features: Remove target column

y = df["price\_range"] # Target: Classifying price range (0,1,2,3)

# 5️⃣ Split Data into Training (80%) & Testing (20%)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 6️⃣ Feature Scaling (Standardization)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# 7️⃣ Train the Naïve Bayes Model

model = GaussianNB()

model.fit(X\_train\_scaled, y\_train)

# 8️⃣ Make Predictions

y\_pred = model.predict(X\_test\_scaled)

# 9️⃣ Evaluate Model Performance

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"\n📊 Model Accuracy: {accuracy \* 100:.2f}%")

print("\n🔹 Classification Report:")

print(classification\_report(y\_test, y\_pred))

# 🔟 Confusion Matrix Visualization

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

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

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=[0,1,2,3], yticklabels=[0,1,2,3])

plt.xlabel("Predicted Label")

plt.ylabel("Actual Label")

plt.title("Confusion Matrix - Mobile Price Classification")

plt.show()

**\*\*19.Navie bayes in python\*\***

# 1️⃣ Import Required Libraries

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

from sklearn.datasets import load\_iris

# 2️⃣ Load the Iris Dataset

iris = load\_iris()

X = iris.data # Features

y = iris.target # Target labels

# 3️⃣ Split Data into Training (80%) & Testing (20%)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 4️⃣ Feature Scaling (Standardization)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# 5️⃣ Train the Naïve Bayes Model

model = GaussianNB()

model.fit(X\_train\_scaled, y\_train)

# 6️⃣ Make Predictions

y\_pred = model.predict(X\_test\_scaled)

# 7️⃣ Evaluate Model Performance

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"\n📊 Model Accuracy: {accuracy \* 100:.2f}%")

print("\n🔹 Classification Report:")

print(classification\_report(y\_test, y\_pred))

# 8️⃣ Confusion Matrix Visualization

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

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

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=iris.target\_names, yticklabels=iris.target\_names)

plt.xlabel("Predicted Label")

plt.ylabel("Actual Label")

plt.title("Confusion Matrix - Naïve Bayes")

plt.show()

**\*\*20.Future sales prediction\*\***

# 1️⃣ 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\_squared\_error, r2\_score

# 2️⃣ Generate Sample Sales Data (Avoids FileNotFoundError)

data = {

"Date": pd.date\_range(start="2023-01-01", periods=100, freq='D'),

"Sales": np.random.randint(500, 5000, 100) # Random sales data

}

df = pd.DataFrame(data)

# 3️⃣ Convert 'Date' Column to Datetime Format

df['Date'] = pd.to\_datetime(df['Date'])

df = df.sort\_values(by='Date')

# 4️⃣ Extract Year, Month, and Day as Features

df['Year'] = df['Date'].dt.year

df['Month'] = df['Date'].dt.month

df['Day'] = df['Date'].dt.day

# 5️⃣ Define Features (X) and Target (y)

X = df[['Year', 'Month', 'Day']]

y = df['Sales']

# 6️⃣ Split Data into Training (80%) and Testing (20%) Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 7️⃣ Train a Linear Regression Model

model = LinearRegression()

model.fit(X\_train, y\_train)

# 8️⃣ Make Predictions

y\_pred = model.predict(X\_test)

# 9️⃣ Evaluate the Model

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"\n📊 Model Performance:")

print(f"✅ Mean Squared Error (MSE): {mse:.2f}")

print(f"✅ R² Score: {r2:.2f}")

# 🔟 Future Sales Prediction

future\_dates = pd.DataFrame({'Year': [2025], 'Month': [12], 'Day': [1]}) # Example Future Date

future\_sales\_pred = model.predict(future\_dates)

print(f"\n📈 Predicted Sales for 2025-12-01: {future\_sales\_pred[0]:.2f}")

# 🔹 Plot Actual vs Predicted Sales

plt.figure(figsize=(10,5))

plt.scatter(y\_test, y\_pred, color='blue', label="Predicted Sales")

plt.plot(y\_test, y\_test, color='red', label="Perfect Fit")

plt.xlabel("Actual Sales")

plt.ylabel("Predicted Sales")

plt.title("Actual vs Predicted Sales")

plt.legend()

plt.show()