**## Experiment 1 (WAP to implement BackPropagation ANN)**

**import numpy as np**

**# Activation functions and their derivatives**

**def sigmoid(x):**

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

**def sigmoid\_derivative(x):**

**return x \* (1 - x)**

**# Input data (6 samples, each with 3 features)**

**X = np.array([[0, 0, 1],**

**[0, 1, 1],**

**[1, 0, 1],**

**[1, 1, 1],**

**[0, 1, 0],**

**[1, 0, 0]])**

**# Output data (6 samples, each with 1 target)**

**y = np.array([[0],**

**[1],**

**[1],**

**[0],**

**[1],**

**[0]])**

**# Seed the random number generator for reproducibility**

**np.random.seed(1)**

**# Initialize weights randomly with mean 0**

**weights = 2 \* np.random.random((3, 1)) - 1**

**# Learning rate**

**learning\_rate = 0.1**

**# Number of iterations**

**iterations = 10000**

**# Training process**

**for iteration in range(iterations):**

**# Forward propagation**

**input\_layer = X**

**outputs = sigmoid(np.dot(input\_layer, weights))**

**# Calculate the error**

**error = y - outputs**

**# Backpropagation**

**adjustments = error \* sigmoid\_derivative(outputs)**

**weights += np.dot(input\_layer.T, adjustments) \* learning\_rate**

**# Final weights after training**

**print("Weights after training:")**

**print(weights)**

**# Test the network with a new input**

**test\_input = np.array([1, 0, 0])**

**output = sigmoid(np.dot(test\_input, weights))**

**print("Output for test input [1,0,0]:")**

**print(output)**

**## EXperiment 2 - WAP to implement Naive Bayes Classifier on a given dataset**

**# Deep Learning Experiment 2 import pandas as pd**

**import pandas as pd**

**# Define the Dataset**

**data = {**

**'Age': ['young','young','middle-aged','old','young','middle-aged','old','old','young','middle-aged'],**

**'Income':['low','medium','medium','high','high','low','medium','low','high','medium'],**

**'Purchase':['not-buy','buy','buy','buy','buy','not-buy','buy','not-buy','not-buy','buy']**

**}**

**# Create a dataframe**

**df = pd.DataFrame(data)**

**# Calculate prior probabilities**

**P\_buy = df['Purchase'].value\_counts()['buy'] / len(df)**

**P\_notbuy = df['Purchase'].value\_counts()['not-buy'] / len(df)**

**# Calculate likelihoods**

**P\_middleaged\_buy = len(df[(df['Age'] == 'middle-aged') & (df['Purchase'] == 'buy')]) / len(df[df['Purchase'] == 'buy'])**

**P\_middleaged\_notbuy = len(df[(df['Age'] == 'middle-aged') & (df['Purchase'] == 'not-buy')]) / len(df[df['Purchase'] == 'not-buy'])**

**P\_high\_buy = len(df[(df['Income'] == 'high') & (df['Purchase'] == 'buy')]) / len(df[df['Purchase'] == 'buy'])**

**P\_high\_notbuy = len(df[(df['Income'] == 'high') & (df['Purchase'] == 'not-buy')]) / len(df[df['Purchase'] == 'not-buy'])**

**# Calculate posterior probabilities (unnormalized)**

**posterior\_buy = P\_middleaged\_buy \* P\_high\_buy \* P\_buy**

**posterior\_notbuy = P\_middleaged\_notbuy \* P\_high\_notbuy \* P\_notbuy**

**# Normalize posterior probabilities**

**total\_posterior = posterior\_buy + posterior\_notbuy**

**posterior\_buy\_normalized = posterior\_buy / total\_posterior**

**posterior\_notbuy\_normalized = posterior\_notbuy / total\_posterior**

**# Print Results**

**print(f'P(Purchase = buy| Age = middle-aged, Income = high): {posterior\_buy\_normalized}')**

**print(f'P(Purchase = not-buy| Age = middle-aged, Income = high): {posterior\_notbuy\_normalized}')**

**# Make Prediction**

**if posterior\_buy\_normalized > posterior\_notbuy\_normalized:**

**print('Prediction: Buy, Purchase')**

**else:**

**print('Prediction: No, Do not Purchase')**

**## Experiment 3 - WAP K nearest neighbor algorithm to classify the iris dataset print both correct and wrong predictions**

**import pandas as pd**

**from sklearn.datasets import load\_iris**

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

**from sklearn.neighbors import KNeighborsClassifier**

**from sklearn.metrics import accuracy\_score**

**# Load the Iris dataset**

**iris = load\_iris()**

**X = iris.data**

**y = iris.target**

**target\_names = iris.target\_names**

**# Split the dataset 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)**

**# Initialize the k-NN classifier with k=3**

**knn = KNeighborsClassifier(n\_neighbors=3)**

**# Train the classifier**

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

**# Make predictions**

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

**# Print both correct and wrong predictions**

**correct\_predictions = 0**

**wrong\_predictions = 0**

**for i in range(len(y\_test)):**

**true\_label = target\_names[y\_test[i]]**

**predicted\_label = target\_names[y\_pred[i]]**

**if y\_test[i] == y\_pred[i]:**

**print(f"✅ Correct: True Label = {true\_label}, Predicted Label = {predicted\_label}")**

**correct\_predictions += 1**

**else:**

**print(f"❌ Wrong: True Label = {true\_label}, Predicted Label = {predicted\_label}")**

**wrong\_predictions += 1**

**# Print the number of correct and wrong predictions**

**print(f"\nSummary:")**

**print(f"Correct Predictions: {correct\_predictions}")**

**print(f"Wrong Predictions: {wrong\_predictions}")**

**# Calculate and print the accuracy**

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

**print(f"\nAccuracy: {accuracy \* 100:.2f}%")**

**## Experiment 4 - WAP to implement and demonstrate Locally Weighted Regression Algorithm**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from sklearn.linear\_model import LinearRegression**

**def local\_regression(x0, X, Y, tau):**

**"""Perfrom local regression at a point x0."""**

**#Calculate weights using Gaussian kernel**

**weights = np.exp(- (X-x0)\*2 / (2\*tau\*2)).ravel()**

**# Fit a weighted Linear Model**

**model = LinearRegression()**

**model.fit(X,Y, sample\_weight = weights)**

**#Predict the value at x0**

**return model.predict(np.array([[x0]]))[0]**

**# Sample data**

**np.random.seed(0)**

**X= np.sort(np.random.rand(200)\*10).reshape(-1,1)**

**Y= np.sin(X).ravel() + np.random.normal(0,0.7,200)**

**#Fit the local regression model for each point in X**

**tau = 1.0 #Bandwidth parameter**

**Y\_pred = np.array([local\_regression(x0,X,Y,tau) for x0 in X.ravel()])**

**#Plot the results**

**plt.scatter(X,Y,label = "Data Points", color = "blue")**

**plt.plot(np.sort(X.ravel()), Y\_pred[np.argsort(X.ravel())],label="Locally Weighted Regression", color="red",linewidth=2)**

**plt.title("Locally Weighted Regression")**

**plt.xlabel("X")**

**plt.ylabel("Y")**

**plt.legend()**

**plt.show()**

**## Experiment 6 - WAP to implement K means Clustering in Python**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from sklearn.cluster import KMeans**

**from sklearn.decomposition import PCA**

**# Your custom dataset**

**data = np.array([**

**[2, 10],**

**[2, 5],**

**[8, 4],**

**[5, 8],**

**[7, 5],**

**[6, 4],**

**[1, 2],**

**[4, 9]**

**])**

**# Apply K-Means clustering with 3 clusters**

**kmeans = KMeans(n\_clusters=3, random\_state=42)**

**kmeans.fit(data)**

**labels = kmeans.labels\_**

**centroids = kmeans.cluster\_centers\_**

**# Apply PCA to reduce to 2 components (for visualization)**

**pca = PCA(n\_components=2)**

**data\_pca = pca.fit\_transform(data)**

**centroids\_pca = pca.transform(centroids) # Transform centroids to the PCA space**

**# Plot the clusters**

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

**plt.scatter(data\_pca[:, 0], data\_pca[:, 1], c=labels, cmap='viridis', edgecolor='k', s=100)**

**plt.scatter(centroids\_pca[:, 0], centroids\_pca[:, 1], s=300, c='red', marker='X', label='Centroids')**

**plt.xlabel('PCA Component 1')**

**plt.ylabel('PCA Component 2')**

**plt.title('K-Means Clustering with Centroids on Custom Dataset')**

**plt.legend()**

**plt.show()**

**## Experiemnt 7 - WAP to implement DecisionTreeClassifier and Visualize its decision making process**

**from sklearn.tree import DecisionTreeClassifier**

**from sklearn import tree**

**import numpy as np**

**X= np.array([**

**[50,500], [50,300], [30,300], [20,100], [20,300], [30,500], [50,100],[30,100]**

**])**

**# Target: NO(not buying)=0, Yes(buying) = 1**

**y = np.array([1,1,1,0,0,1,0,0])**

**#create the DecisionTreeClassifier**

**clf = DecisionTreeClassifier(criterion='entropy', max\_depth = 2, random\_state = 42)**

**#Train the classifier with the data**

**clf.fit(X,y)**

**# Make a prediction**

**new\_data = np.array([[55,250]])**

**prediction = clf.predict(new\_data)**

**print(f"Prediction for [Income = 55k, Credit Score = 250]: {'Buy' if prediction[0] == 1 else 'Not Buy'}")**

**#visualize the decision tree**

**import matplotlib.pyplot as plt**

**plt.figure(figsize=(12,8))**

**tree.plot\_tree(clf, feature\_names = ["Income", "Credit Score"], class\_names = ["Not Buy", "Buy"], filled= True)**

**plt.show()**

**## Experiment 8 - WAP to implement Principal Component Analysis (PCA) on any dataset**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Custom dataset**

**data = np.array([**

**[2, 3],**

**[3, 5],**

**[4, 7],**

**[5, 8],**

**[6, 10],**

**[7, 11],**

**[8, 13],**

**[9, 15],**

**[10, 17]**

**])**

**# Centering the data by subtracting the mean**

**mean = np.mean(data, axis=0)**

**data\_centered = data - mean**

**print("Data centered around mean:\n", data\_centered)**

**# Covariance matrix**

**cov\_matrix = np.cov(data\_centered.T)**

**print("Covariance Matrix:\n", cov\_matrix)**

**# Eigen decomposition**

**eigenvalues, eigenvectors = np.linalg.eig(cov\_matrix)**

**print("Eigenvalues:\n", eigenvalues)**

**print("Eigenvectors:\n", eigenvectors)**

**# Sort eigenvalues and eigenvectors**

**sorted\_indices = np.argsort(eigenvalues)[::-1]**

**sorted\_eigenvalues = eigenvalues[sorted\_indices]**

**sorted\_eigenvectors = eigenvectors[:, sorted\_indices]**

**print("Sorted Eigenvalues:\n", sorted\_eigenvalues)**

**print("Sorted Eigenvectors:\n", sorted\_eigenvectors)**

**# Project data onto the first principal component (1D reduction)**

**pca\_data = data\_centered.dot(sorted\_eigenvectors[:, 0])**

**print("Projected Data onto Principal Component:\n", pca\_data)**

**# Plot original data and projection**

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

**plt.scatter(data[:, 0], data[:, 1], color='b', label="Original Data")**

**plt.xlabel('X')**

**plt.ylabel('Y')**

**# Calculate the line representing the first principal component**

**pc1\_line = pca\_data[:, np.newaxis] \* sorted\_eigenvectors[:, 0] + mean**

**plt.plot(pc1\_line[:, 0], pc1\_line[:, 1], color='r', linestyle='--', label="Principal Component 1")**

**plt.legend()**

**plt.title('PCA Step-by-Step on Custom Dataset')**

**plt.show()**

**## Experiment 9 - WAP to implement Confusion Matrix and ROC Curve**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from sklearn.metrics import confusion\_matrix, roc\_curve, auc, ConfusionMatrixDisplay**

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

**from sklearn.linear\_model import LogisticRegression**

**from sklearn.datasets import make\_classification**

**# Generate a synthetic binary classification dataset**

**X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)**

**# Split the data into training and test sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)**

**# Train a logistic regression model**

**model = LogisticRegression()**

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

**# Predict the labels for the test set**

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

**# -------------------**

**# Part 1: Confusion Matrix**

**# -------------------**

**# Calculate the confusion matrix**

**cm = confusion\_matrix(y\_test, y\_pred)**

**print("Confusion Matrix:\n", cm)**

**# Display the confusion matrix**

**disp = ConfusionMatrixDisplay(confusion\_matrix=cm)**

**disp.plot(cmap=plt.cm.Blues)**

**plt.title("Confusion Matrix")**

**plt.show()**

**# -------------------**

**# Part 2: ROC Curve**

**# -------------------**

**# Predict probabilities for the positive class**

**y\_probs = model.predict\_proba(X\_test)[:, 1]**

**# Calculate the ROC curve and AUC**

**fpr, tpr, thresholds = roc\_curve(y\_test, y\_probs)**

**roc\_auc = auc(fpr, tpr)**

**# Plot the ROC curve**

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

**plt.plot(fpr, tpr, color='blue', label=f"ROC Curve (AUC = {roc\_auc:.2f})")**

**plt.plot([0, 1], [0, 1], color='gray', linestyle='--')**

**plt.xlabel('False Positive Rate')**

**plt.ylabel('True Positive Rate')**

**plt.title('ROC Curve')**

**plt.legend()**

**plt.show()**

**## Experiment 10 - WAP to implement SVM on any dataset and analyze the accuracy with logistic regression**

**# Import necessary libraries**

**import numpy as np**

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

**from sklearn.svm import SVC**

**from sklearn.linear\_model import LogisticRegression**

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

**# Create a custom dataset**

**# Features: [feature1, feature2]**

**# Labels: 0 or 1**

**X = np.array([**

**[2, 3], [1, 2], [2, 1], [3, 4],**

**[5, 6], [6, 7], [5, 5], [6, 6],**

**[8, 8], [9, 9], [8, 7], [7, 8],**

**[3, 5], [4, 4], [3, 3], [4, 3]**

**])**

**# Labels**

**y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0])**

**# Split the dataset into training and testing sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)**

**# Create and train the SVM classifier**

**svm\_classifier = SVC(kernel='linear', random\_state=42)**

**svm\_classifier.fit(X\_train, y\_train)**

**# Make predictions with the SVM model**

**y\_pred\_svm = svm\_classifier.predict(X\_test)**

**# Evaluate SVM model**

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

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

**class\_report\_svm = classification\_report(y\_test, y\_pred\_svm)**

**print("SVM Accuracy:", accuracy\_svm)**

**print("SVM Confusion Matrix:\n", conf\_matrix\_svm)**

**print("SVM Classification Report:\n", class\_report\_svm)**

**# Create and train the Logistic Regression classifier**

**logreg\_classifier = LogisticRegression(max\_iter=200, random\_state=42)**

**logreg\_classifier.fit(X\_train, y\_train)**

**# Make predictions with the Logistic Regression model**

**y\_pred\_logreg = logreg\_classifier.predict(X\_test)**

**# Evaluate Logistic Regression model**

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

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

**class\_report\_logreg = classification\_report(y\_test, y\_pred\_logreg)**

**print("\nLogistic Regression Accuracy:", accuracy\_logreg)**

**print("Logistic Regression Confusion Matrix:\n", conf\_matrix\_logreg)**

**print("Logistic Regression Classification Report:\n", class\_report\_logreg)**

**# Compare and print the better model based on accuracy**

**if accuracy\_svm > accuracy\_logreg:**

**print("\nSVM has higher accuracy than Logistic Regression.")**

**elif accuracy\_logreg > accuracy\_svm:**

**print("\nLogistic Regression has higher accuracy than SVM.")**

**else:**

**print("\nBoth models have the same accuracy.")**

**Implement SVM on any dataset**

**import numpy as np**

**from sklearn.svm import SVC**

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

**from sklearn.metrics import accuracy\_score, classification\_report**

**# Data**

**X = np.array([**

**[2, 3], [1, 2], [2, 1], [3, 4],**

**[5, 6], [6, 7], [5, 5], [6, 6],**

**[8, 8], [9, 9], [8, 7], [7, 8],**

**[3, 5], [4, 4], [3, 3], [4, 3]**

**])**

**# Labels**

**y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0])**

**# Split the dataset into training and testing sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)**

**# Initialize the SVM model**

**svm\_model = SVC(kernel='linear')**

**a**

**# Train the model**

**svm\_model.fit(X\_train, y\_train)**

**# Make predictions on the test set**

**y\_pred = svm\_model.predict(X\_test)**

**# Evaluate the model**

**print("Accuracy:", accuracy\_score(y\_test, y\_pred))**

**print("Classification Report:\n", classification\_report(y\_test, y\_pred))**