**Practical 1**

**Cmd installation:**

**pip install folium**

**pip install networkx**

**pip install matplotlib**

**Code:**

import networkx as nx

import matplotlib.pyplot as plt

from collections import deque

from matplotlib.lines import Line2D

# Graph represented as an adjacency list (city connections)

graph = {

'Mumbai': ['Pune', 'Delhi', 'Bangalore'],

'Pune': ['Mumbai', 'Nagpur'],

'Delhi': ['Mumbai', 'Chandigarh'],

'Bangalore': ['Mumbai', 'Hyderabad'],

'Nagpur': ['Pune'],

'Chandigarh': ['Delhi'],

'Hyderabad': ['Bangalore']

}

# Breadth First Search (BFS) with limit

def bfs(graph, start\_node, limit=None):

visited = set()

queue = deque([start\_node])

bfs\_traversal = []

while queue:

node = queue.popleft()

if node not in visited:

bfs\_traversal.append(node)

visited.add(node)

queue.extend(graph[node])

if limit and len(bfs\_traversal) >= limit:

break

return bfs\_traversal

# Iterative Depth First Search (IDFS) with limit

def idfs(graph, start\_node, limit=None):

visited = set()

stack = [(start\_node, 0)]

idfs\_traversal = []

while stack:

node, depth = stack.pop()

if node not in visited:

idfs\_traversal.append(node)

visited.add(node)

stack.extend((neighbor, depth + 1) for neighbor in graph[node])

if limit and len(idfs\_traversal) >= limit:

break

return idfs\_traversal

# Plotting the graph and traversal

def plot\_graph(graph, traversal=None, title="City Connections"):

G = nx.Graph()

# Add edges between cities

for city, neighbors in graph.items():

for neighbor in neighbors:

G.add\_edge(city, neighbor)

pos = nx.spring\_layout(G) # Layout for the nodes

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

nx.draw(G, pos, with\_labels=True, node\_color='skyblue', node\_size=2000, edge\_color='black', font\_size=10, font\_color='black', font\_weight='bold')

# Highlight the traversal path if provided

if traversal:

edges\_in\_path = [(traversal[i], traversal[i+1]) for i in range(len(traversal)-1)]

nx.draw\_networkx\_edges(G, pos, edgelist=edges\_in\_path, width=4, edge\_color='orange')

nx.draw\_networkx\_nodes(G, pos, nodelist=traversal, node\_color='orange', node\_size=2000)

plt.title(title)

plt.show()

# Plot both BFS and IDFS in one graph

def plot\_comparison(graph, bfs\_traversal, idfs\_traversal):

G = nx.Graph()

for city, neighbors in graph.items():

for neighbor in neighbors:

G.add\_edge(city, neighbor)

pos = nx.spring\_layout(G)

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

# Plot BFS edges in orange

edges\_in\_bfs = [(bfs\_traversal[i], bfs\_traversal[i+1]) for i in range(len(bfs\_traversal)-1)]

nx.draw(G, pos, with\_labels=True, node\_color='skyblue', node\_size=2000, edge\_color='black', font\_size=10, font\_color='black', font\_weight='bold')

nx.draw\_networkx\_edges(G, pos, edgelist=edges\_in\_bfs, width=4, edge\_color='orange')

nx.draw\_networkx\_nodes(G, pos, nodelist=bfs\_traversal, node\_color='orange', node\_size=2000)

# Plot IDFS edges in green

edges\_in\_idfs = [(idfs\_traversal[i], idfs\_traversal[i+1]) for i in range(len(idfs\_traversal)-1)]

nx.draw\_networkx\_edges(G, pos, edgelist=edges\_in\_idfs, width=4, edge\_color='green')

nx.draw\_networkx\_nodes(G, pos, nodelist=idfs\_traversal, node\_color='green', node\_size=2000)

# Create custom legend with colors

legend\_elements = [Line2D([0], [0], color='orange', lw=4, label='BFS Path'),

Line2D([0], [0], color='green', lw=4, label='IDFS Path')]

plt.legend(handles=legend\_elements, loc="upper right")

plt.title("Comparison of BFS and IDFS Traversals (Limited to 4 cities)")

plt.show()

# Step 1: BFS Traversal (with limit)

bfs\_result = bfs(graph, 'Mumbai', limit=4)

print("BFS Traversal: ", bfs\_result)

# Step 2: IDFS Traversal (with limit)

idfs\_result = idfs(graph, 'Mumbai', limit=4)

print("IDFS Traversal: ", idfs\_result)

# Step 3: Plot the graph for BFS

plot\_graph(graph, bfs\_result, "BFS Traversal (Limited to 4 cities)")

# Step 4: Plot the graph for IDFS

plot\_graph(graph, idfs\_result, "IDFS Traversal (Limited to 4 cities)")

# Step 6: Plot comparison between BFS and IDFS traversals

plot\_comparison(graph, bfs\_result, idfs\_result)

**Practical 2**

**Cmd installation:**

**pip install folium**

**pip install networkx**

**pip install matplotlib**

**Code:**

import folium

import networkx as nx

import heapq

import time

import matplotlib.pyplot as plt

# Define city connections in adjacency list format

city\_connections = {

'Mumbai': ['Pune', 'Nashik', 'Goa'],

'Pune': ['Mumbai', 'Aurangabad', 'Satara'],

'Nagpur': ['Aurangabad'],

'Nashik': ['Mumbai', 'Aurangabad'],

'Aurangabad': ['Pune', 'Nagpur', 'Nashik'],

'Goa': ['Mumbai', 'Kolhapur'],

'Satara': ['Pune', 'Kolhapur'],

'Kolhapur': ['Satara', 'Goa'],

}

# Create the graph

G = nx.Graph()

for city, connections in city\_connections.items():

for neighbor in connections:

G.add\_edge(city, neighbor)

# A\* Search function

def a\_star\_search(graph, start, goal):

queue = []

heapq.heappush(queue, (0, start, [start]))

visited = set()

while queue:

\_, current, path = heapq.heappop(queue)

if current == goal:

return path

visited.add(current)

for neighbor in graph.neighbors(current):

if neighbor not in visited:

heapq.heappush(queue, (0, neighbor, path + [neighbor]))

return None

# Recursive Best-First Search function

def rbfs(graph, current, goal, f\_limit, path):

if current == goal:

return path

successors = []

for neighbor in graph.neighbors(current):

if neighbor not in path:

successors.append(neighbor)

if not successors:

return None

while successors:

best = successors[0]

result = rbfs(graph, best, goal, f\_limit, path + [best])

if result:

return result

successors.pop(0)

return None

# Helper function for RBFS

def recursive\_best\_first\_search(graph, start, goal):

return rbfs(graph, start, goal, float('inf'), [start])

# Execute A\* Search

start = 'Mumbai'

goal = 'Kolhapur'

start\_time = time.time()

a\_star\_result = a\_star\_search(G, start, goal)

a\_star\_time = time.time() - start\_time

print("Path from Mumbai to Kolhapur using A\* Search:", a\_star\_result)

print("A\* Search Time:", a\_star\_time, "seconds")

# Execute RBFS

start\_time = time.time()

rbfs\_result = recursive\_best\_first\_search(G, start, goal)

rbfs\_time = time.time() - start\_time

print("Path from Mumbai to Kolhapur using RBFS:", rbfs\_result)

print("RBFS Time:", rbfs\_time, "seconds")

# Create the map with Folium

m = folium.Map(location=[19.0760, 72.8777], zoom\_start=6)

# Add city markers

for city in city\_connections.keys():

folium.Marker(

location=[19.0760, 72.8777], # You can set a default location for markers

popup=city,

).add\_to(m)

# Add paths to the map

for path in [a\_star\_result, rbfs\_result]:

if path:

for i in range(len(path) - 1):

folium.PolyLine(

locations=[[19.0760, 72.8777], [19.0760, 72.8777]], # Dummy coordinates for illustration

color='blue' if path == a\_star\_result else 'green',

weight=2.5,

opacity=0.7

).add\_to(m)

# Function to plot a path in the graph

def plot\_path(graph, path, title, color):

plt.figure(figsize=(10, 8))

pos = nx.spring\_layout(graph)

nx.draw(graph, pos, with\_labels=True, node\_color='skyblue', node\_size=2000, edge\_color='gray', font\_size=10, font\_color='black', font\_weight='bold')

# Highlight the path

if path:

path\_edges = list(zip(path[:-1], path[1:]))

nx.draw\_networkx\_edges(graph, pos, edgelist=path\_edges, edge\_color=color, width=3)

plt.title(title)

plt.show()

# Plotting graphs

plot\_path(G, a\_star\_result, "A\* Search Path from Mumbai to Kolhapur", 'blue')

plot\_path(G, rbfs\_result, "RBFS Path from Mumbai to Kolhapur", 'green')

# Comparison graph for both A\* Search and RBFS

plt.figure(figsize=(10, 8))

pos = nx.spring\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_color='skyblue', node\_size=2000, edge\_color='gray', font\_size=10, font\_color='black', font\_weight='bold')

# Highlight A\* Search path

if a\_star\_result:

a\_star\_edges = list(zip(a\_star\_result[:-1], a\_star\_result[1:]))

nx.draw\_networkx\_edges(G, pos, edgelist=a\_star\_edges, edge\_color='blue', width=3, label='A\* Path')

# Highlight RBFS path

if rbfs\_result:

rbfs\_edges = list(zip(rbfs\_result[:-1], rbfs\_result[1:]))

nx.draw\_networkx\_edges(G, pos, edgelist=rbfs\_edges, edge\_color='green', width=3, label='RBFS Path')

plt.title("Comparison of A\* Search and RBFS Paths")

plt.legend()

plt.show()

**Practical 3**

**Cmd installation:**

**pip install pandas**

**pip install scikit-learn**

**pip install matplotlib**

**pip install numpy pandas matplotlib seaborn scikit-learn**

**Code:**

# Import necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import load\_iris

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

from sklearn.tree import DecisionTreeClassifier, plot\_tree

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

# Step 1: Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Convert to DataFrame for better visualization

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

df['target'] = y

print("First 5 rows of the dataset:")

print(df.head())

print("\nShape of the dataset:", df.shape)

# Step 2: 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)

# Step 3: Build the Original Decision Tree Classifier

original\_decision\_tree = DecisionTreeClassifier(random\_state=42)

original\_decision\_tree.fit(X\_train, y\_train)

# Step 4: Make predictions on the test set

y\_pred\_original = original\_decision\_tree.predict(X\_test)

# Step 5: Evaluate the original model

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

print(f"\nAccuracy of Original Decision Tree Classifier: {accuracy\_original \* 100:.2f}%")

# Confusion Matrix for the original model

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

print("\nConfusion Matrix (Original):")

print(conf\_matrix\_original)

# Classification Report for the original model

print("\nClassification Report (Original):")

print(classification\_report(y\_test, y\_pred\_original, target\_names=iris.target\_names))

# Step 6: Build the Pruned Decision Tree Classifier

max\_depth = 3 # Set the maximum depth for pruning

pruned\_decision\_tree = DecisionTreeClassifier(max\_depth=max\_depth, random\_state=42)

pruned\_decision\_tree.fit(X\_train, y\_train)

# Step 7: Make predictions on the test set for the pruned tree

y\_pred\_pruned = pruned\_decision\_tree.predict(X\_test)

# Step 8: Evaluate the pruned model

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

print(f"\nAccuracy of Pruned Decision Tree Classifier: {accuracy\_pruned \* 100:.2f}%")

# Confusion Matrix for the pruned model

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

print("\nConfusion Matrix (Pruned):")

print(conf\_matrix\_pruned)

# Classification Report for the pruned model

print("\nClassification Report (Pruned):")

print(classification\_report(y\_test, y\_pred\_pruned, target\_names=iris.target\_names))

# Step 9: Visualize the Original Decision Tree

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

plt.title("Original Decision Tree Visual Representation", fontsize=16) # Ensure title is clear

plot\_tree(original\_decision\_tree, feature\_names=iris.feature\_names, class\_names=iris.target\_names, filled=True)

plt.suptitle("Original Decision Tree", fontsize=20) # Add a main title

plt.tight\_layout() # Adjust layout

plt.show()

# Step 10: Visualize the Pruned Decision Tree

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

plt.title("Pruned Decision Tree Visual Representation", fontsize=16) # Ensure title is clear

plot\_tree(pruned\_decision\_tree, feature\_names=iris.feature\_names, class\_names=iris.target\_names, filled=True)

plt.suptitle("Pruned Decision Tree", fontsize=20) # Add a main title

plt.tight\_layout() # Adjust layout

plt.show()

**Practical 4**

**Cmd installation:**

**pip install numpy**

**pip install pandas**

**pip install matplotlib**

**pip install seaborn**

**pip install scikit-learn**

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import load\_wine

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

from sklearn.preprocessing import StandardScaler

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

# Step 1: Load and Prepare the Data

wine = load\_wine()

X = wine.data

y = wine.target

# Convert to DataFrame for visualization

data = pd.DataFrame(X, columns=wine.feature\_names)

data['target'] = y

print(data.describe())

print()

print("Shape of the dataset is:", data.shape)

print()

# 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)

# Normalize the data

scaler = StandardScaler()

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

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

# Step 2: Build and Train the Feed Forward Neural Network

# Activation function and its derivative

def sigmoid(x):

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

def sigmoid\_derivative(x):

return x \* (1 - x)

# Neural Network parameters

input\_layer\_size = X\_train.shape[1]

hidden\_layer\_size = 10 # Number of neurons in hidden layer

output\_layer\_size = 3 # Three classes in the wine dataset

learning\_rate = 0.01

epochs = 10000

# Initialize weights

np.random.seed(42)

weights\_input\_hidden = np.random.rand(input\_layer\_size, hidden\_layer\_size)

weights\_hidden\_output = np.random.rand(hidden\_layer\_size, output\_layer\_size)

# Training the Neural Network

mse\_values = []

for epoch in range(epochs):

# Feedforward

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

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

final\_layer\_input = np.dot(hidden\_layer\_output, weights\_hidden\_output)

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

# One-hot encode the output

y\_train\_onehot = np.zeros((y\_train.size, y\_train.max() + 1))

y\_train\_onehot[np.arange(y\_train.size), y\_train] = 1

# Compute the error

error = y\_train\_onehot - final\_output

# Backpropagation

d\_final\_output = error \* sigmoid\_derivative(final\_output)

error\_hidden\_layer = np.dot(d\_final\_output, weights\_hidden\_output.T)

d\_hidden\_layer = error\_hidden\_layer \* sigmoid\_derivative(hidden\_layer\_output)

# Update weights

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

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

mse = np.mean(error\*\*2)

mse\_values.append(mse)

if epoch % 1000 == 0:

print(f'Epoch {epoch}/{epochs}, MSE: {mse}')

# Visualize the training process

plt.figure(figsize=(10, 6))

plt.plot(mse\_values, label="MSE during Training")

plt.xlabel("Epochs")

plt.ylabel("Mean Squared Error")

plt.title("Training Progress of the Neural Network")

plt.legend()

plt.show()

# Step 3: Evaluate the Neural Network

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

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

final\_layer\_input\_test = np.dot(hidden\_layer\_output\_test, weights\_hidden\_output)

final\_output\_test = sigmoid(final\_layer\_input\_test)

# Convert final output to class predictions

predictions = np.argmax(final\_output\_test, axis=1)

# Evaluate performance

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

print(f'Accuracy: {accuracy \* 100:.2f}%')

# Confusion Matrix

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

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

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

yticklabels=wine.target\_names)

plt.xlabel("Predicted Labels")

plt.ylabel("True Labels")

plt.title("Confusion Matrix of the Neural Network Predictions")

plt.show()

**Practical 5**

**Cmd installation:**

**pip install numpy**

**pip install pandas**

**pip install matplotlib**

**pip install seaborn**

**pip install scikit-learn**

**Code:**

# Import necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn import svm

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

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score, f1\_score, precision\_score

# Load the breast cancer dataset

from sklearn.datasets import load\_breast\_cancer

data = load\_breast\_cancer()

X = data.data

y = data.target

# Standardize the features

scaler = StandardScaler()

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

# Perform PCA to reduce to 2D for visualization purposes

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X\_scaled)

# Split the dataset into training and testing sets

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

# Function to plot the decision boundary for SVM

def plot\_decision\_boundary(clf, X, y, title):

h = .02 # step size in the mesh

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.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

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

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')

plt.title(title)

plt.xlabel('PCA Component 1')

plt.ylabel('PCA Component 2')

plt.show()

# Function to print evaluation metrics (confusion matrix and classification report)

def evaluate\_model(clf, X\_test, y\_test, y\_pred, title):

print(f"Evaluation for {title}:\n")

# Accuracy, F1 Score, Precision

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

f1 = f1\_score(y\_test, y\_pred)

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

print(f"Accuracy: {accuracy:.4f}")

print(f"F1 Score: {f1:.4f}")

print(f"Precision: {precision:.4f}")

# Confusion matrix

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

sns.heatmap(cm, annot=True, fmt="d", cmap='Blues')

plt.title(f'Confusion Matrix - {title}')

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.show()

# Classification report

print(f"Classification Report for {title}:\n")

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

# Applying Linear SVM

linear\_svm = svm.SVC(kernel='linear')

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

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

# Plot Linear SVM

plot\_decision\_boundary(linear\_svm, X\_test, y\_test, 'Linear SVM (After Classification)')

evaluate\_model(linear\_svm, X\_test, y\_test, y\_pred\_linear, 'Linear SVM')

# Applying Non-linear SVM with RBF Kernel

rbf\_svm = svm.SVC(kernel='rbf')

rbf\_svm.fit(X\_train, y\_train)

y\_pred\_rbf = rbf\_svm.predict(X\_test)

# Plot RBF Kernel SVM

plot\_decision\_boundary(rbf\_svm, X\_test, y\_test, 'Non-linear SVM with RBF Kernel (After Classification)')

evaluate\_model(rbf\_svm, X\_test, y\_test, y\_pred\_rbf, 'RBF Kernel SVM')

**Practical 6**

**Cmd installation:**

**pip install numpy pandas matplotlib seaborn scikit-learn**

**pip install scikit-learn --upgrade**

**Code:**

# Import necessary libraries for visualization

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.datasets import fetch\_openml

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

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

from sklearn.metrics import accuracy\_score, classification\_report, roc\_curve, auc, confusion\_matrix, roc\_auc\_score

import warnings

# Suppress warnings (optional)

warnings.filterwarnings("ignore", category=FutureWarning)

# Load Spambase dataset from UCI (via fetch\_openml)

spam\_data = fetch\_openml(name='spambase', version=1)

# Convert to DataFrame

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

df['target'] = spam\_data.target.astype(int)

# Print the first 5 rows of the dataset

print("First 5 rows of the dataset:")

print(df.head())

# Distribution of target (spam vs not spam)

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

sns.countplot(x='target', data=df)

plt.title('Distribution of Target (Spam vs Not Spam)')

plt.xlabel('Target (0 = Not Spam, 1 = Spam)')

plt.ylabel('Count')

plt.show()

# Features (X) and target (y)

X = df.drop(columns=['target'])

y = df['target']

# Split the dataset into training and testing sets (70% training, 30% testing)

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

# Initialize a weak classifier (decision stump)

weak\_classifier = DecisionTreeClassifier(max\_depth=1)

# Initialize the AdaBoost model using the weak classifier with the SAMME algorithm

ada\_boost = AdaBoostClassifier(estimator=weak\_classifier, n\_estimators=50, random\_state=42, algorithm='SAMME')

# Train the AdaBoost model

ada\_boost.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = ada\_boost.predict(X\_test)

# Evaluate the performance of the AdaBoost model

print("Classification Report for AdaBoost:")

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

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

print("Accuracy (AdaBoost):", ada\_accuracy)

# Confusion Matrix

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

# Plot Confusion Matrix Heatmap

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

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])

plt.title('Confusion Matrix - AdaBoost Classifier')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# Calculate ROC AUC Score for AdaBoost

y\_pred\_proba\_ada = ada\_boost.predict\_proba(X\_test)[:, 1]

roc\_auc\_ada = roc\_auc\_score(y\_test, y\_pred\_proba\_ada)

print("ROC AUC Score for AdaBoost:", roc\_auc\_ada)

# Compare with weak classifier (decision stump)

weak\_classifier.fit(X\_train, y\_train)

y\_pred\_weak = weak\_classifier.predict(X\_test)

y\_pred\_proba\_weak = weak\_classifier.predict\_proba(X\_test)[:, 1]

# Evaluate weak classifier

print("Classification Report for Weak Classifier (Decision Stump):")

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

weak\_accuracy = accuracy\_score(y\_test, y\_pred\_weak)

print("Accuracy (Weak Classifier):", weak\_accuracy)

# Confusion Matrix for Weak Classifier

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

# Plot Confusion Matrix Heatmap for Weak Classifier

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

sns.heatmap(conf\_matrix\_weak, annot=True, fmt='d', cmap='Greens', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])

plt.title('Confusion Matrix - Weak Classifier (Decision Stump)')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# Calculate ROC AUC Score for Weak Classifier

roc\_auc\_weak = roc\_auc\_score(y\_test, y\_pred\_proba\_weak)

print("ROC AUC Score for Weak Classifier:", roc\_auc\_weak)

# Plot ROC Curves for both AdaBoost and Weak Classifier together

fpr\_ada, tpr\_ada, \_ = roc\_curve(y\_test, y\_pred\_proba\_ada)

fpr\_weak, tpr\_weak, \_ = roc\_curve(y\_test, y\_pred\_proba\_weak)

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

plt.plot(fpr\_ada, tpr\_ada, color='blue', label=f'AdaBoost (AUC = {roc\_auc\_ada:.2f})')

plt.plot(fpr\_weak, tpr\_weak, color='green', label=f'Weak Classifier (AUC = {roc\_auc\_weak:.2f})')

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

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curves - AdaBoost vs Weak Classifier')

plt.legend(loc="lower right")

plt.show()

# Plot a bar graph comparing the accuracy of AdaBoost and Weak Classifier

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

models = ['AdaBoost', 'Weak Classifier']

accuracies = [ada\_accuracy, weak\_accuracy]

sns.barplot(x=models, y=accuracies, hue=models, palette='Set1', legend=False) # Added hue parameter

plt.title('Accuracy Comparison: AdaBoost vs Weak Classifier')

plt.ylabel('Accuracy')

plt.show()

**Practical 7**

**Cmd installation:**

**pip install numpy**

**pip install matplotlib**

**pip install pandas**

**pip install scikit-learn**

**pip install seaborn**

**Code:**

# Importing the libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB, BernoulliNB

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

from sklearn.datasets import load\_iris

import seaborn as sns

# Importing the dataset from sklearn

iris = load\_iris()

X = iris.data[:, [0, 3]] # Selecting features (sepal length and petal width)

y = iris.target # Selecting target variable (species)

# Splitting the dataset into the Training set and Test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=0)

# Feature Scaling (only for GaussianNB)

sc = StandardScaler()

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

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

# Training the Gaussian Naive Bayes model on the Training set

gaussian\_classifier = GaussianNB()

gaussian\_classifier.fit(X\_train\_scaled, y\_train)

# Predicting the Test set results for Gaussian

y\_pred\_gaussian = gaussian\_classifier.predict(X\_test\_scaled)

# Evaluating the Gaussian Naive Bayes model

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

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

# Training the Bernoulli Naive Bayes model on the Training set (no scaling)

bernoulli\_classifier = BernoulliNB()

bernoulli\_classifier.fit(X\_train, y\_train)

# Predicting the Test set results for Bernoulli

y\_pred\_bernoulli = bernoulli\_classifier.predict(X\_test)

# Evaluating the Bernoulli Naive Bayes model

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

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

# Print the results

print("Gaussian Naive Bayes:")

print("Predicted Test Results: ", y\_pred\_gaussian)

print("Model Accuracy: ", accuracy\_gaussian \* 100, "%")

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

print("~" \* 20)

print("Bernoulli Naive Bayes:")

print("Predicted Test Results: ", y\_pred\_bernoulli)

print("Model Accuracy: ", accuracy\_bernoulli \* 100, "%")

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

print("~" \* 20)

# Visualization of Accuracy Comparison

labels = ['Gaussian', 'Bernoulli']

accuracies = [accuracy\_gaussian \* 100, accuracy\_bernoulli \* 100]

# Create a colorful bar plot for accuracy

colors = ['#FF9999', '#99FF99']

plt.figure(figsize=(10, 6))

bars = plt.bar(labels, accuracies, color=colors)

plt.title('Accuracy Comparison of Naive Bayes Classifiers')

plt.xlabel('Classifier')

plt.ylabel('Accuracy (%)')

plt.ylim(0, 100)

# Adding data labels to bars

for bar in bars:

yval = bar.get\_height()

plt.text(bar.get\_x() + bar.get\_width()/2, yval + 1, f"{yval:.2f}%", ha='center')

plt.show()

# Function to plot confusion matrix

def plot\_confusion\_matrix(cm, title):

plt.figure(figsize=(6, 5))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.title(title)

plt.ylabel('Actual')

plt.xlabel('Predicted')

plt.xticks(ticks=[0.5, 1.5, 2.5], labels=iris.target\_names)

plt.yticks(ticks=[0.5, 1.5, 2.5], labels=iris.target\_names, rotation=0)

plt.show()

# Plot confusion matrices for each classifier

plot\_confusion\_matrix(cm\_gaussian, 'Gaussian Naive Bayes Confusion Matrix')

plot\_confusion\_matrix(cm\_bernoulli, 'Bernoulli Naive Bayes Confusion Matrix')

**Practical 8**

**Cmd installation:**

**pip install numpy**

**pip install pandas**

**pip install scikit-learn**

**pip install matplotlib**

**pip install seaborn**

**Code:**

import numpy as np

import pandas as pd

from sklearn.datasets import load\_breast\_cancer

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

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

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

import seaborn as sns

# Load the Breast Cancer dataset

cancer = load\_breast\_cancer()

X = cancer.data

y = cancer.target

# Convert to DataFrame for better visualization

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

# Display the first 8 rows of the dataset

print("\nFirst 8 rows of the dataset:")

print(df\_cancer.head(8))

# Display the first 8 rows of the target column

print("\nFirst 8 rows of the target column:")

print(pd.Series(cancer.target).head(8))

# Data Preprocessing: Standardize the features

scaler = StandardScaler()

X = scaler.fit\_transform(X)

# Visualizing the dataset before classification using PCA

pca = PCA(n\_components=2) # Reducing to 2 components

X\_pca = pca.fit\_transform(X)

# Create a DataFrame for visualization

df\_pca = pd.DataFrame(X\_pca, columns=['PCA1', 'PCA2'])

df\_pca['target'] = y

# Plot the dataset before classification

plt.figure(figsize=(10, 6))

sns.scatterplot(x='PCA1', y='PCA2', hue='target', data=df\_pca, palette='Set1')

plt.title('Breast Cancer Dataset Visualization (Before Classification)')

plt.show()

# 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)

# Transform the data using PCA for visualization

X\_train\_pca = pca.fit\_transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

# Implement the K-NN algorithm for different k-values

k\_values = [2, 5, 7, 9]

for k in k\_values:

knn = KNeighborsClassifier(n\_neighbors=k)

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

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

# Evaluate the accuracy of the predictions

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

print(f'\nAccuracy for k={k}: {accuracy:.2f}')

# Display the classification report

print(f"\nClassification Report for k={k}:")

print(classification\_report(y\_test, y\_pred, target\_names=cancer.target\_names))

# Confusion matrix

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

# Visualize the confusion matrix

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

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

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.title(f'Confusion Matrix for k={k}')

plt.show()

# Create a mesh grid for plotting decision boundaries

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

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

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02),

np.arange(y\_min, y\_max, 0.02))

Z = knn.predict(pca.inverse\_transform(np.c\_[xx.ravel(), yy.ravel()]))

Z = Z.reshape(xx.shape)

# Plot decision boundaries

plt.figure(figsize=(10, 6))

plt.contourf(xx, yy, Z, alpha=0.3, cmap='Set1')

plt.scatter(X\_train\_pca[:, 0], X\_train\_pca[:, 1], c=y\_train, edgecolor='k', cmap='Set1')

plt.xlabel('PCA Component 1')

plt.ylabel('PCA Component 2')

plt.title(f'Decision Boundaries for k={k}')

plt.show()

# Cross-validation to find the optimal number of neighbors

neighbors = np.arange(1, 21)

cv\_scores = []

# Perform 10-fold cross-validation

for k in neighbors:

knn\_cv = KNeighborsClassifier(n\_neighbors=k)

scores = cross\_val\_score(knn\_cv, X\_train, y\_train, cv=10, scoring='accuracy')

cv\_scores.append(scores.mean())

# Plotting error rate vs. k value

plt.figure(figsize=(10, 6))

plt.plot(neighbors, 1 - np.array(cv\_scores), marker='o', linestyle='--', color='b')

plt.xlabel('Number of Neighbors K')

plt.ylabel('Error Rate')

plt.title('Error Rate vs. K Value')

plt.show()

# Define k values for KNN and store accuracy scores

k\_range = range(1, 21)

accuracy\_scores = []

for k in k\_range:

knn = KNeighborsClassifier(n\_neighbors=k)

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

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

# Evaluate the accuracy of the predictions

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

accuracy\_scores.append(accuracy)

# Print the k values and their corresponding accuracy scores

print("\nK values and their accuracy:")

for k, acc in zip(k\_range, accuracy\_scores):

print(f'k={k}: Accuracy={acc:.2f}')

# Plotting accuracy vs. k value

plt.figure(figsize=(10, 6))

plt.plot(k\_range, accuracy\_scores, marker='o', linestyle='-', color='b', label='Accuracy')

plt.xlabel('Number of Neighbors (k)')

plt.ylabel('Accuracy')

plt.title('Accuracy vs. Number of Neighbors (k)')

plt.xticks(k\_range)

plt.grid(True)

plt.legend()

plt.show()

**Practical 9**

**Cmd installation:**

**pip install pandas numpy mlxtend matplotlib seaborn**

**Code:**

# Import necessary libraries

import pandas as pd

import numpy as np

from mlxtend.frequent\_patterns import apriori, association\_rules

import matplotlib.pyplot as plt

import seaborn as sns

# Set random seed for reproducibility

np.random.seed(42)

# Step 1: Create a random dataset

num\_samples = 1000

data = {

'id': range(num\_samples),

'full\_name': [f'Person {i}' for i in range(num\_samples)],

'age': np.random.randint(15, 100, size=num\_samples),

'bmi': np.random.uniform(15, 40, size=num\_samples),

'blood\_pressure': np.random.randint(60, 180, size=num\_samples),

'glucose\_levels': np.random.randint(70, 300, size=num\_samples),

'gender': np.random.choice(['Male', 'Female'], size=num\_samples),

'smoking\_status': np.random.choice(['Smoker', 'Non-smoker'], size=num\_samples),

'condition': np.random.choice(['Diabetic', 'Hypertensive', 'Healthy'], size=num\_samples)

}

df = pd.DataFrame(data)

# Step 2: Data Preprocessing

# Drop unnecessary columns

df = df.drop(columns=['id', 'full\_name'])

# Bin continuous variables into categories

df['age\_bin'] = pd.cut(df['age'], bins=[0, 20, 40, 60, 80, 100], labels=['0-20', '21-40', '41-60', '61-80', '81-100'])

df['bmi\_bin'] = pd.cut(df['bmi'], bins=[0, 18.5, 25, 30, 40], labels=['Underweight', 'Normal', 'Overweight', 'Obese'])

df['blood\_pressure\_bin'] = pd.cut(df['blood\_pressure'], bins=[0, 80, 120, 140, 200], labels=['Low', 'Normal', 'Prehypertension', 'Hypertension'])

df['glucose\_levels\_bin'] = pd.cut(df['glucose\_levels'], bins=[0, 90, 140, 200, 300], labels=['Normal', 'Pre-diabetic', 'Diabetic', 'High Diabetic'])

# Convert categorical columns to one-hot encoded variables

df = pd.get\_dummies(df, columns=['gender', 'smoking\_status', 'condition', 'age\_bin', 'bmi\_bin', 'blood\_pressure\_bin', 'glucose\_levels\_bin'])

# Drop original continuous columns

df.drop(columns=['age', 'bmi', 'blood\_pressure', 'glucose\_levels'], inplace=True)

# Print dataset after preprocessing

print("\nPreprocessed Data: \n", df.head())

# Set display options to avoid squeezing the output

pd.set\_option('display.max\_columns', None)

pd.set\_option('display.width', 1000)

# Step 3: Apply the Apriori Algorithm to find frequent itemsets with minimum support of 0.1 (10%)

frequent\_itemsets = apriori(df, min\_support=0.1, use\_colnames=True)

# Limit the frequent itemsets display to the first 10 rows

print("\nFrequent Itemsets (First 10 rows): \n", frequent\_itemsets.head(10))

# Step 4: Generate Association Rules with minimum confidence of 0.5

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.5)

# Display only the first 10 association rules

print("\nAssociation Rules (First 10 rows): \n", rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']].head(10))

# Evaluate and interpret the rules

# Top 10 rules based on confidence

top\_rules = rules.sort\_values(by='confidence', ascending=False).head(10)

print("\nTop 10 Association Rules: \n", top\_rules)

# --- Visualization of Support and Confidence ---

# Plot support vs confidence

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

sns.scatterplot(x="support", y="confidence", size="lift", hue="lift", data=rules, palette="coolwarm", sizes=(40, 200))

plt.title("Association Rules - Support vs Confidence")

plt.xlabel("Support")

plt.ylabel("Confidence")

plt.legend(loc='upper right')

plt.show()

# Plot lift for top 10 rules (modifying to remove the warning)

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

sns.barplot(x=top\_rules['lift'], y=top\_rules.index, hue=top\_rules['lift'], palette='viridis', legend=False)

plt.title("Top 10 Rules - Lift")

plt.xlabel("Lift")

plt.ylabel("Rule Index")

plt.show()

# --- New Addition: Plot top 10 rules by support ---

top\_rules\_by\_support = rules.sort\_values(by='support', ascending=False).head(10)

# Plot support for top 10 rules

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

sns.barplot(x=top\_rules\_by\_support['support'], y=top\_rules\_by\_support.index, hue=top\_rules\_by\_support['support'], palette='Blues', legend=False)

plt.title("Top 10 Rules - Support")

plt.xlabel("Support")

plt.ylabel("Rule Index")

plt.show()

**Practical 10**

**Cmd installation:**

**Pip install tensorflow**

**Pip install tensorflow matplotlib**

**pip install opencv-python numpy**

**Code:**

**First we have to run this code in idle file:**

import os

import cv2

import numpy as np

# Create directories for dataset

os.makedirs('random\_digits/train/0', exist\_ok=True)

os.makedirs('random\_digits/train/1', exist\_ok=True)

os.makedirs('random\_digits/train/2', exist\_ok=True)

os.makedirs('random\_digits/train/3', exist\_ok=True)

os.makedirs('random\_digits/train/4', exist\_ok=True)

os.makedirs('random\_digits/train/5', exist\_ok=True)

os.makedirs('random\_digits/train/6', exist\_ok=True)

os.makedirs('random\_digits/train/7', exist\_ok=True)

os.makedirs('random\_digits/train/8', exist\_ok=True)

os.makedirs('random\_digits/train/9', exist\_ok=True)

# Function to generate random digit images

def generate\_random\_digit\_images(num\_images=1000):

for \_ in range(num\_images):

digit = np.random.randint(0, 10) # Random digit from 0 to 9

# Create a blank image

img = np.ones((28, 28), dtype=np.uint8) \* 255

# Set the font

font = cv2.FONT\_HERSHEY\_SIMPLEX

# Draw the digit on the image

cv2.putText(img, str(digit), (5, 20), font, 1, (0, 0, 0), 2, cv2.LINE\_AA)

# Save the image in the corresponding folder

cv2.imwrite(f'random\_digits/train/{digit}/{\_}.png', img)

# Generate random digit images

generate\_random\_digit\_images()

**Then after running above code cut the code and**

**Then run the following code in the same idle file:**

import warnings

import tensorflow as tf

import matplotlib.pyplot as plt

import os

# Suppress warnings

warnings.filterwarnings("ignore")

# Image parameters

img\_width, img\_height = 28, 28

batch\_size = 32

epochs = 5

# Directory for the dataset

train\_data\_dir = 'random\_digits/train'

# Load the dataset using ImageDataGenerator

train\_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0/255.0)

train\_generator = train\_datagen.flow\_from\_directory(

train\_data\_dir,

target\_size=(img\_width, img\_height),

color\_mode='grayscale',

batch\_size=batch\_size,

class\_mode='sparse'

)

# Build the model

model = tf.keras.models.Sequential([

tf.keras.layers.Flatten(input\_shape=(img\_width, img\_height)), # Flatten the 28x28 images

tf.keras.layers.Dense(128, activation='relu'), # First hidden layer

tf.keras.layers.Dropout(0.2), # Dropout layer to reduce overfitting

tf.keras.layers.Dense(10) # Output layer with 10 classes (digits 0-9)

])

# Compile the model

model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

metrics=['accuracy'])

# Train the model

model.fit(train\_generator, epochs=epochs)

# Visualize some predictions

def display\_predictions(generator, n=10):

images, labels = next(generator)

predictions = model.predict(images)

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

for i in range(n):

plt.subplot(2, 5, i + 1)

plt.imshow(images[i].squeeze(), cmap='gray')

plt.title(f'Pred: {predictions[i].argmax()}')

plt.axis('off')

plt.tight\_layout()

plt.show()

# Display predictions for the first 10 generated images

display\_predictions(train\_generator)