**Practical No 1**

**AIM: Breadth First Search & Iterative Depth First Search   
• Implement the Breadth First Search algorithm to solve a given problem.   
• Implement the Iterative Depth First Search algorithm to solve the same problem.   
• Compare the performance and efficiency of both algorithms.**

**Program 1: Breadth First Search.**from collections import deque

graph = {

"S": [("A", 2), ("B", 4)],

"A": [("S", 2), ("B", 2), ("C", 3), ("D", 7)],

"B": [("S", 4), ("A", 2), ("C", 1)],

"C": [("A", 3), ("B", 1), ("G", 5)],

"D": [("A", 7), ("G", 2)],

"G": [("C", 5), ("D", 2)]

}

def bfs(start, goal):

queue = deque([(start, [start])])

visited = set()

while queue:

node, path = queue.popleft()

if node == goal:

return path

if node not in visited:

visited.add(node)

for neighbor, \_ in graph[node]:

queue.append((neighbor, path + [neighbor]))

return None

print("BFS Path:", bfs("S", "G"))

**Output:**

****

**Program 2: Iterative Depth First Search.**

from collections import deque

graph = { "S": [("A", 2), ("B", 4)],

"A": [("S", 2), ("B", 2), ("C", 3), ("D", 7)],

"B": [("S", 4), ("A", 2), ("C", 1)],

"C": [("A", 3), ("B", 1), ("G", 5)],

"D": [("A", 7), ("G", 2)],

"G": [("C", 5), ("D", 2)]

}

def dfs\_iter(start, goal):

stack = [(start, [start])]

visited = set()

while stack:

node, path = stack.pop()

if node == goal:

return path

if node not in visited:

visited.add(node)

for neighbor, \_ in graph[node]:

stack.append((neighbor, path + [neighbor]))

return None

print("DFS Iterative Path:", dfs\_iter("S", "G"))

**Output:**

****

**Practical No 2**

**AIM: A\* Search and Recursive Best-First Search.  
• Implement the A\* Search algorithm for solving a pathfinding problem.   
• Implement the Recursive Best-First Search algorithm for the same problem.   
• Compare the performance and effectiveness of both algorithms.**

**Program 1: A\* Search**import heapq

graph = {

"S": [("A", 2), ("B", 4)],

"A": [("S", 2), ("B", 2), ("C", 3), ("D", 7)],

"B": [("S", 4), ("A", 2), ("C", 1)],

"C": [("A", 3), ("B", 1), ("G", 5)],

"D": [("A", 7), ("G", 2)],

"G": [("C", 5), ("D", 2)]

}

heuristic = {

"S": 7, "A": 6, "B": 2, "C": 1, "D": 1, "G": 0

}

def astar(start, goal):

pq = [(heuristic[start], 0, start, [start])] # (f, g, node, path)

visited = set()

while pq:

f, g, node, path = heapq.heappop(pq)

if node == goal:

return path, g

if node not in visited:

visited.add(node)

for neighbor, cost in graph[node]:

new\_g = g + cost

new\_f = new\_g + heuristic[neighbor]

heapq.heappush(pq, (new\_f, new\_g, neighbor, path + [neighbor]))

return None, float("inf")

path, cost = astar("S", "G")

print("A\* Path:", path, "Cost:", cost)

**Output:**

****

**Program 2: Recursive Best-First Search.**

graph = {

"S": [("A", 2), ("B", 4)],

"A": [("S", 2), ("B", 2), ("C", 3), ("D", 7)],

"B": [("S", 4), ("A", 2), ("C", 1)],

"C": [("A", 3), ("B", 1), ("G", 5)],

"D": [("A", 7), ("G", 2)],

"G": [("C", 5), ("D", 2)]

}

heuristic = {

"S": 7, "A": 6, "B": 2, "C": 1, "D": 1, "G": 0

}

def rbfs(node, path, g, f\_limit, goal):

if node == goal:

return path, g

successors = []

for neighbor, cost in graph[node]:

new\_g = g + cost

new\_f = new\_g + heuristic[neighbor]

successors.append([new\_f, neighbor, path + [neighbor], new\_g])

if not successors:

return None, float("inf")

while True:

successors.sort(key=lambda x: x[0])

best = successors[0]

if best[0] > f\_limit:

return None, best[0]

alternative = successors[1][0] if len(successors) > 1 else float("inf")

result, best[0] = rbfs(

best[1], best[2], best[3], min(f\_limit, alternative), goal

)

successors[0] = best

if result is not None:

return result, best[0]

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

path, cost = rbfs(start, [start], 0, float("inf"), goal)

return path, cost

path, cost = recursive\_best\_first\_search("S", "G")

print("RBFS Path:", path, "Cost:", cost)

**Output:**

****

**Practical No 3**

**AIM: Decision Tree Learning-   
• Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.   
• Evaluate the accuracy and effectiveness of the decision tree on test data.   
• Visualize and interpret the generated decision tree.  
Install Libraries:**pip install pandas matplotlib seaborn scikit-learn **Program:***#imports*

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

*# Load dataset*

iris = load\_iris()

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

y = pd.Series(iris.target, name="species")

# Train-Test Split (70% train, 30% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42, stratify=y

)

*# Train Decision Tree (using entropy & max\_depth=3)*

dtree = DecisionTreeClassifier(criterion="entropy", max\_depth=3, random\_state=42)

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

*# Predictions*

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

*# Evaluation*

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

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

*# Confusion Matrix Heatmap*

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

sns.heatmap(

cm, annot=True, fmt="d", cmap="Blues",

xticklabels=iris.target\_names,

yticklabels=iris.target\_names

)

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()

*# Visualize the Decision Tree*

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

plot\_tree(

dtree,

feature\_names=iris.feature\_names,

class\_names=iris.target\_names,

filled=True,

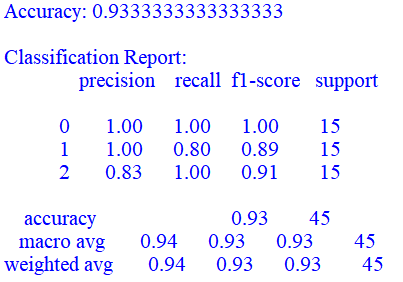
rounded=True,

fontsize=10

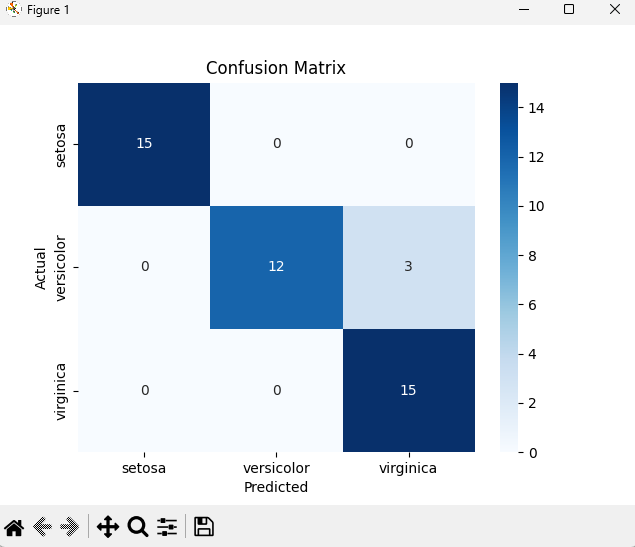
)

plt.show()

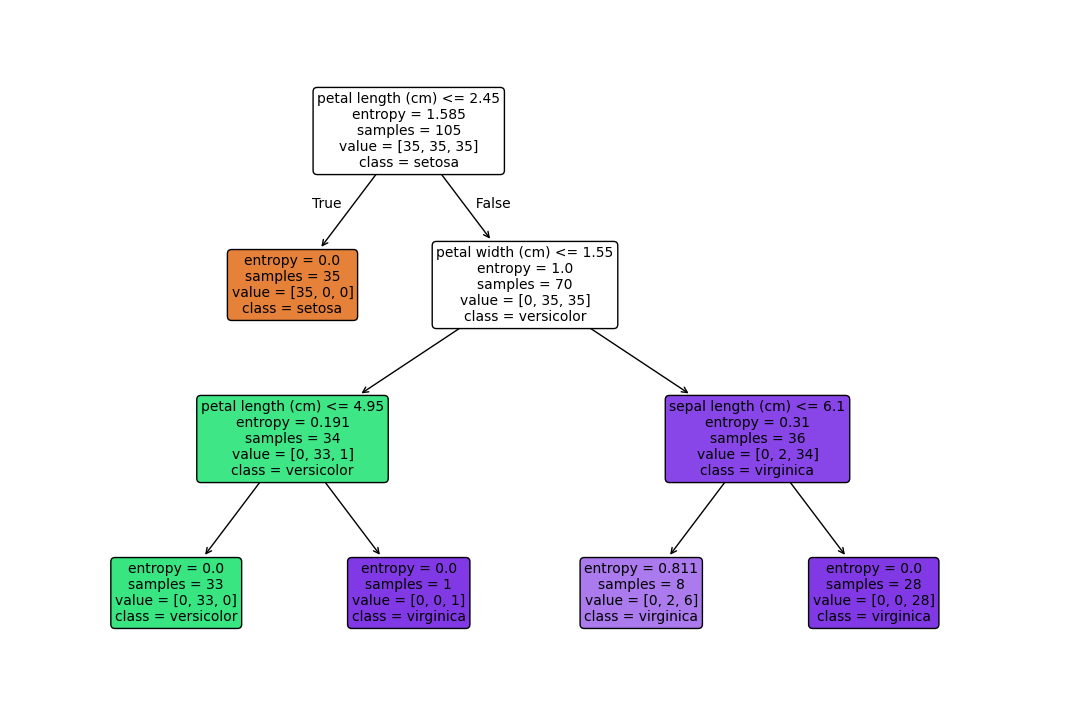
**Output:**

****

**Confusion Matrix-**

****

**Visual Representation-**

****

**Practical No 4**

**AIM: Feed Forward Backpropagation Neural Network   
• Implement the Feed Forward Backpropagation algorithm to train a neural network.   
• Use a given dataset to train the neural network for a specific task.   
• Evaluate the performance of the trained network on test data.  
  
Program:***# Step 1: Import Libraries*

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.preprocessing import OneHotEncoder, StandardScaler

*# Step 2: Load and Preprocess the Dataset*

iris = load\_iris()

X, y = iris.data, iris.target.reshape(-1, 1)

encoder = OneHotEncoder(sparse\_output=False)

y = encoder.fit\_transform(y)

scaler = StandardScaler()

X = scaler.fit\_transform(X)

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

*# Step 3: Define the Neural Network Architecture and Initialize Weights*

input\_size = X\_train.shape[1] *# Number of features (4)*

hidden\_size = 8 *# Number of neurons in the hidden layer*

output\_size = y\_train.shape[1] # Number of output classes (3)

lr = 0.1 # Learning rate for weight updates

epochs = 500 # Number of passes through the entire training dataset

np.random.seed(42)

W1 = np.random.randn(input\_size, hidden\_size)

b1 = np.zeros((1, hidden\_size))

W2 = np.random.randn(hidden\_size, output\_size)

b2 = np.zeros((1, output\_size))

def sigmoid(x):

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

def sigmoid\_deriv(x):

return x \* (1 - x)

def softmax(x):

exp\_x = np.exp(x - np.max(x, axis=1, keepdims=True))

return exp\_x / np.sum(exp\_x, axis=1, keepdims=True)

*# Step 4: Training with Forward and Backward Propagation*

for epoch in range(epochs):

z1 = np.dot(X\_train, W1) + b1

a1 = sigmoid(z1)

z2 = np.dot(a1, W2) + b2

a2 = softmax(z2)

loss = -np.mean(np.sum(y\_train \* np.log(a2 + 1e-9), axis=1))

dz2 = a2 - y\_train

dW2 = np.dot(a1.T, dz2) / X\_train.shape[0]

db2 = np.sum(dz2, axis=0, keepdims=True) / X\_train.shape[0]

dz1 = np.dot(dz2, W2.T) \* sigmoid\_deriv(a1)

dW1 = np.dot(X\_train.T, dz1) / X\_train.shape[0]

db1 = np.sum(dz1, axis=0, keepdims=True) / X\_train.shape[0]

W1 -= lr \* dW1

b1 -= lr \* db1

W2 -= lr \* dW2

b2 -= lr \* db2

if epoch % 100 == 0:

print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Step 5: Evaluate the Model on the Test Set

z1\_test = np.dot(X\_test, W1) + b1

a1\_test = sigmoid(z1\_test)

z2\_test = np.dot(a1\_test, W2) + b2

a2\_test = softmax(z2\_test)

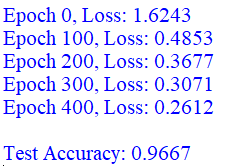
y\_pred = np.argmax(a2\_test, axis=1)

y\_true = np.argmax(y\_test, axis=1)

accuracy = np.mean(y\_pred == y\_true)

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

**Output:**

****

**Practical No 5**

**AIM: Support Vector Machines (SVM)   
• Implement the SVM algorithm for binary classification.   
• Train an SVM model using a given dataset and optimize its parameters.   
• Evaluate the performance of the SVM model on test data and analyze the Results**

**Program:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import make\_classification

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

from sklearn.svm import SVC

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

# Generate synthetic dataset

X, y = make\_classification(n\_samples=300, n\_features=2, n\_redundant=0,

n\_informative=2, n\_clusters\_per\_class=1,

random\_state=42)

# Split into train/test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42, stratify=y

)

print("Training samples:", X\_train.shape[0])

print("Test samples:", X\_test.shape[0])

svm = SVC(kernel="linear", C=1.0, random\_state=42) # Initialize with linear kernel

svm.fit(X\_train, y\_train) # Train

y\_pred = svm.predict(X\_test) # Predict

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

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

# Define parameter grid

param\_grid = {

'C': [0.1, 1, 10],

'kernel': ['linear', 'rbf', 'poly'],

'gamma': ['scale', 'auto']

}

# Grid Search with 5-fold CV

grid = GridSearchCV(SVC(random\_state=42), param\_grid, cv=5, scoring='accuracy')

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

print("Best Parameters:", grid.best\_params\_)

print("Best Cross-Validation Score:", grid.best\_score\_)

# Best model

best\_svm = grid.best\_estimator\_

# Predictions

y\_best\_pred = best\_svm.predict(X\_test)

# Accuracy

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

print("\nClassification Report:\n", classification\_report(y\_test, y\_best\_pred))

# Confusion Matrix

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

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

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()

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

h = 0.02

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

Z = Z.reshape(xx.shape)

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

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, edgecolors='k', cmap=plt.cm.coolwarm)

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

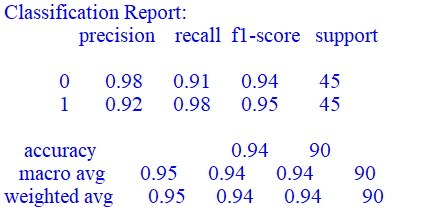
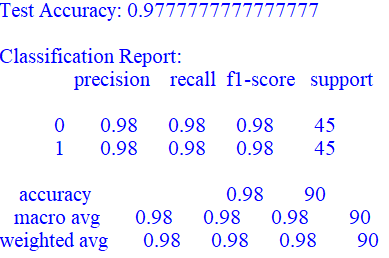
plt.title("SVM Decision Boundary")

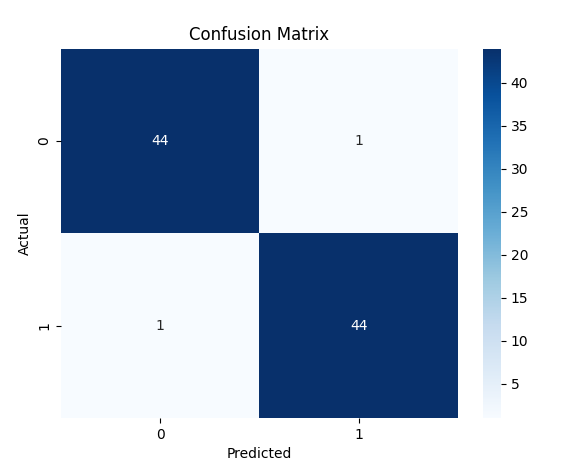
plt.show()

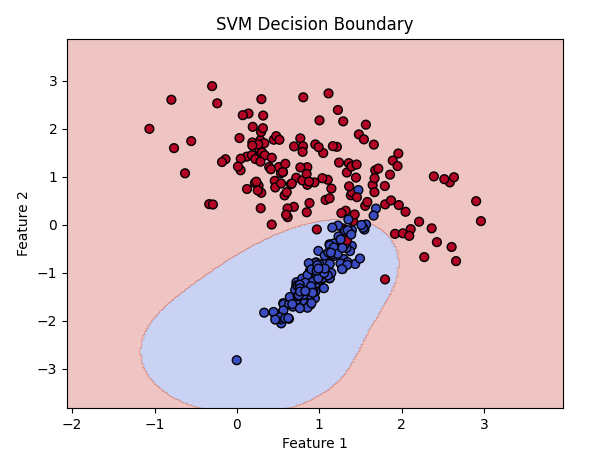
plot\_decision\_boundary(best\_svm, X, y)

**Output:**

** **

** **

****

****

**Practical No 6**

**AIM: Adaboost Ensemble Learning**

**• Implement the Adaboost algorithm to create an ensemble of weak classifiers.**

**• Train the ensemble model on a given dataset and evaluate its performance.**

**• Compare the results with individual weak classifiers.**

**Program:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_classification

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

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

from sklearn.ensemble import AdaBoostClassifier

# 1. Create a toy dataset

X, y = make\_classification(

n\_samples=500, n\_features=2, n\_informative=2, n\_redundant=0,

n\_clusters\_per\_class=1, flip\_y=0.1, class\_sep=1.5, random\_state=42

)

y\_mod = np.where(y == 0, -1, 1) # Convert labels from {0,1} to {-1,1} for AdaBoost from scratch

# Split train/test

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

# 2. Implement AdaBoost from Scratch

class AdaBoostScratch:

def \_\_init\_\_(self, n\_estimators=50):

self.n\_estimators = n\_estimators

self.alphas = []

self.models = []

def fit(self, X, y):

n\_samples, \_ = X.shape

# Initialize weights

w = np.ones(n\_samples) / n\_samples

for \_ in range(self.n\_estimators):

# Train weak classifier (Decision stump)

stump = DecisionTreeClassifier(max\_depth=1, random\_state=42)

stump.fit(X, y, sample\_weight=w)

y\_pred = stump.predict(X)

# Compute weighted error

err = np.sum(w \* (y\_pred != y)) / np.sum(w)

# Compute alpha

alpha = 0.5 \* np.log((1 - err) / (err + 1e-10))

# Update weights

w \*= np.exp(-alpha \* y \* y\_pred)

w /= np.sum(w)

# Save

self.models.append(stump)

self.alphas.append(alpha)

def predict(self, X):

# Weighted vote

final\_pred = np.zeros(X.shape[0])

for alpha, model in zip(self.alphas, self.models):

final\_pred += alpha \* model.predict(X)

return np.sign(final\_pred)

# 3. Train AdaBoost (Scratch)

ada\_scratch = AdaBoostScratch(n\_estimators=50)

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

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

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

print("AdaBoost (Scratch) Accuracy:", acc\_scratch)

# 4. Train AdaBoost (Sklearn)

ada\_sklearn = AdaBoostClassifier(

estimator=DecisionTreeClassifier(max\_depth=1),

n\_estimators=50,

random\_state=42

)

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

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

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

print("AdaBoost (Sklearn) Accuracy:", acc\_sklearn)

# 5. Compare with Weak Classifier Alone

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

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

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

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

print("Weak Classifier Accuracy:", acc\_weak)

# 6. Visualization (Decision Boundaries)

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

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, 200),

np.linspace(y\_min, y\_max, 200))

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

Z = Z.reshape(xx.shape)

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

plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')

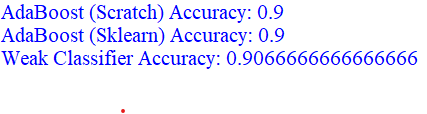
plt.title(title)

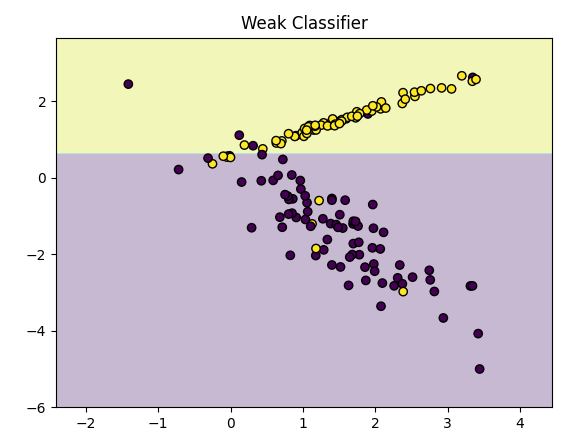
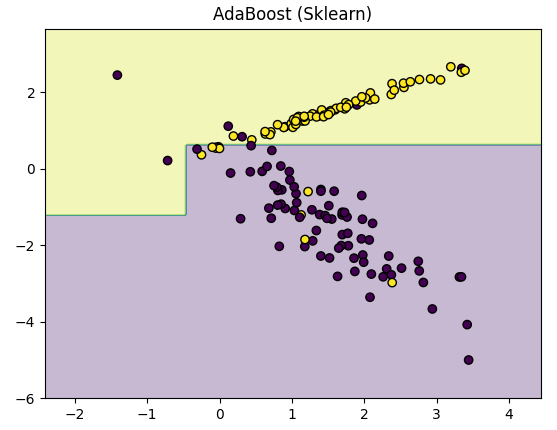
plt.show()

plot\_decision\_boundary(weak\_clf, X\_test, y\_test, "Weak Classifier")

plot\_decision\_boundary(ada\_sklearn, X\_test, y\_test, "AdaBoost (Sklearn)")

**Output:**

****

** **

**Practical No 7**

**AIM: Naive Bayes' Classifier   
• Implement the Naive Bayes' algorithm for classification.   
• Train a Naive Bayes' model using a given dataset and calculate class probabilities.   
• Evaluate the accuracy of the model on test data and analyze the results.**

**Program:**

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.naive\_bayes import GaussianNB

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

# 1. Load Dataset

iris = load\_iris()

X, y = iris.data, iris.target

# Split into train/test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42

)

# 2. Train Naive Bayes Model

nb = GaussianNB()

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

# 3. Predictions & Probabilities

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

y\_prob = nb.predict\_proba(X\_test) # class probabilities

# 4. Evaluate

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

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

print("\nConfusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

# 5. Show some probabilities

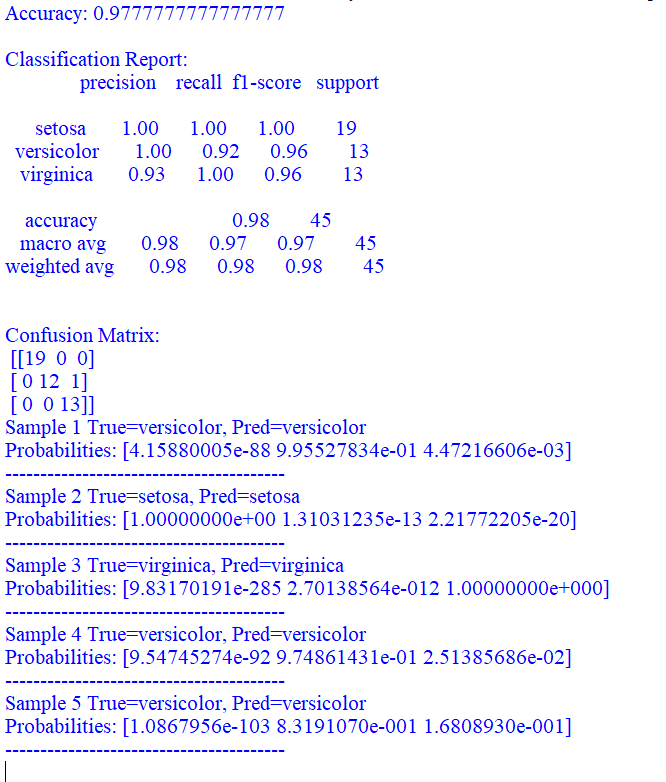
for i in range(5):

print(f"Sample {i+1} True={iris.target\_names[y\_test[i]]}, Pred={iris.target\_names[y\_pred[i]]}")

print("Probabilities:", y\_prob[i])

print("-" \* 40)

**Output:**

****

**Practical No 8**

**AIM: K-Nearest Neighbors (K-NN)   
• Implement the K-NN algorithm for classification or regression.   
• Apply the K-NN algorithm to a given dataset and predict the class or value for test data.   
• Evaluate the accuracy or error of the predictions and analyze the results.**

**Program:**

import numpy as np

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor

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

from sklearn.datasets import load\_iris, fetch\_california\_housing

iris = load\_iris()

X\_cls, y\_cls = iris.data, iris.target

housing = fetch\_california\_housing()

X\_reg, y\_reg = housing.data, housing.target

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

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

scaler\_cls = StandardScaler()

scaler\_reg = StandardScaler()

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

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

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

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

print("--- KNN Classification ---")

knn\_cls = KNeighborsClassifier(n\_neighbors=5)

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

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

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

print(f"Classification Accuracy: {acc:.4f}\n")

print("--- KNN Regression ---")

knn\_reg = KNeighborsRegressor(n\_neighbors=5)

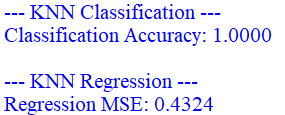
knn\_reg.fit(X\_train\_reg, y\_train\_reg)

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

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

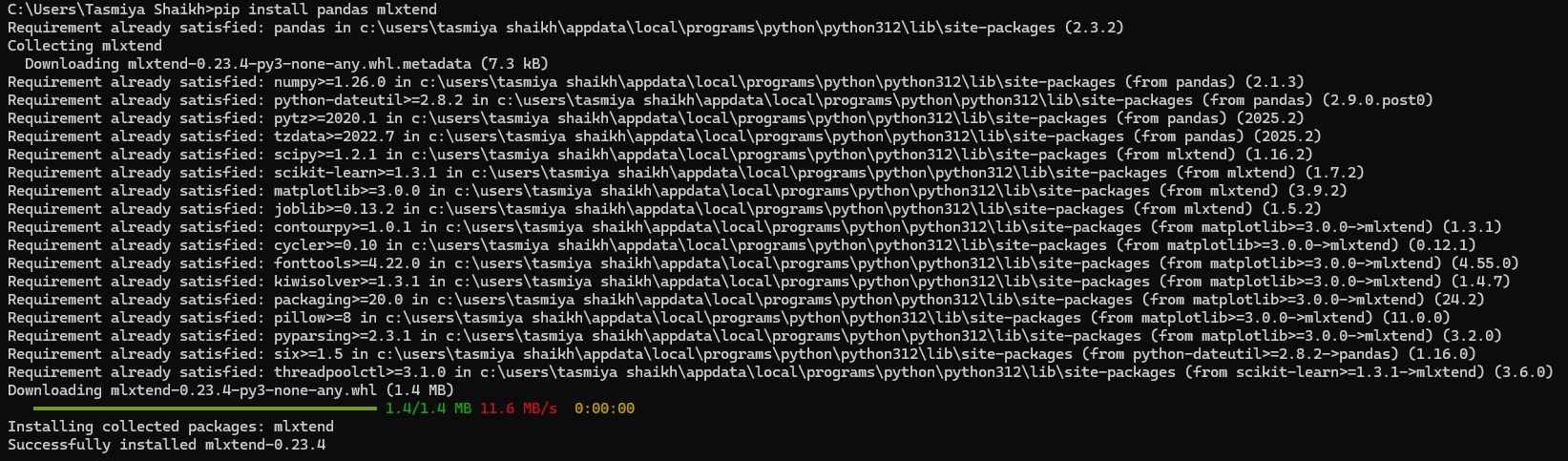
print(f"Regression MSE: {mse:.4f}")

**Output:**

****

**Practical No 9**

**AIM: Association Rule Mining   
• Implement the Association Rule Mining algorithm (e.g., Apriori) to find frequent itemsets.   
• Generate association rules from the frequent itemsets and calculate their support and confidence. • Interpret and analyze the discovered association rules.**

**Install:**pip install pandas mlxtend **Program:**

import pandas as pd

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

from mlxtend.preprocessing import TransactionEncoder

dataset = [

['Milk', 'Bread', 'Eggs'],

['Milk', 'Bread'],

['Milk', 'Eggs'],

['Bread', 'Eggs'],

['Milk', 'Bread', 'Butter'],

['Bread', 'Butter']

]

te = TransactionEncoder()

te\_array = te.fit(dataset).transform(dataset)

df = pd.DataFrame(te\_array, columns=te.columns\_)

print("--- One-Hot Encoded DataFrame ---")

print(df)

print("\n" + "="\*30 + "\n")

frequent\_itemsets = apriori(df, min\_support=0.3, use\_colnames=True)

print("--- Frequent Itemsets (Support >= 0.3) ---")

print(frequent\_itemsets)

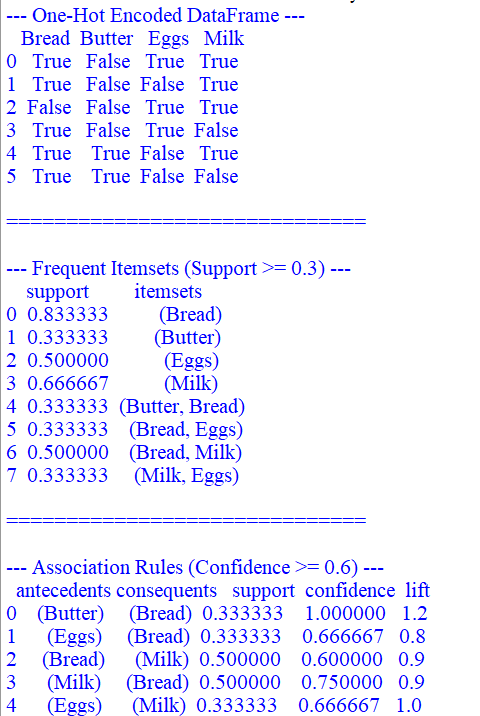
print("\n" + "="\*30 + "\n")

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.6)

print("--- Association Rules (Confidence >= 0.6) ---")

print(rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])

**Output:**

****

**Practical No 10**

**AIM: Demo of OpenAI/TensorFlow Tools:  
• Explore and experiment with OpenAI or TensorFlow tools and libraries.   
• Perform a demonstration or mini-project showcasing the capabilities of the tools.   
• Discuss and present the findings and potential applications.**

**Program 1:**import google.generativeai as genai

import os

from dotenv import load\_dotenv

load\_dotenv()

GOOGLE\_API\_KEY=os.getenv('GOOGLE\_API\_KEY')

print(f"Retrieved API Key: {GOOGLE\_API\_KEY}")

genai.configure(api\_key=GOOGLE\_API\_KEY)

prompt = """Artificial Intelligence is the simulation of human intelligence

in machines that are programmed to think like humans and mimic their actions.

The term may also be applied to any machine that exhibits traits associated

with a human mind such as learning and problem-solving."""

try:

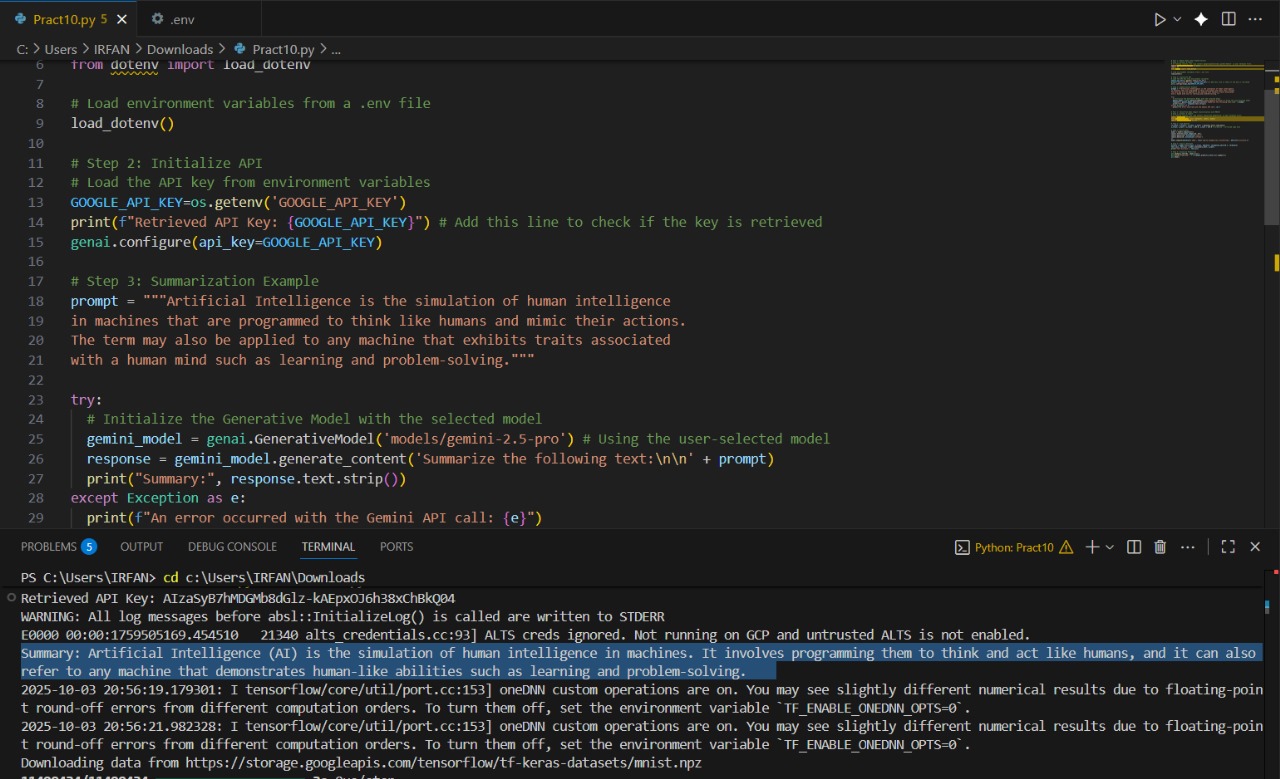
gemini\_model = genai.GenerativeModel('models/gemini-2.5-pro')

response = gemini\_model.generate\_content('Summarize the following text:\n\n' + prompt)

print("Summary:", response.text.strip())

except Exception as e:

print(f"An error occurred with the Gemini API call: {e}")

**Output:**

**Program 2:**

import tensorflow as tf

from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt

import numpy as np

# --- Step 1 & 2: Load and Prepare the MNIST Dataset ---

print("Loading MNIST dataset...")

(x\_train, y\_train), (x\_test, y\_test) = datasets.mnist.load\_data()

# Normalize pixel values to be between 0 and 1

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

print("Dataset loaded and normalized.")

# --- Step 3: Build the Neural Network Model ---

print("Building the model...")

model = models.Sequential([

# Flattens the 28x28 image into a 1D array of 784 pixels

layers.Flatten(input\_shape=(28, 28)),

# A fully connected layer with 128 neurons and ReLU activation

layers.Dense(128, activation='relu'),

# The output layer with 10 neurons (one for each digit 0-9)

# Softmax activation gives a probability distribution for each digit.

layers.Dense(10, activation='softmax')

])

# Compile the model with an optimizer, loss function, and metrics

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

print("Model built and compiled.")

model.summary()

# --- Step 4: Train the Model ---

print("\nTraining the model...")

history = model.fit(x\_train, y\_train,

epochs=5,

validation\_split=0.1,

verbose=2)

print("Training finished.")

# Evaluate the model on the test dataset

print("\nEvaluating the model on the test data...")

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print("\nTest Accuracy:", test\_acc)

# --- Step 5: Visualize a Prediction ---

# Make a prediction on the first image in the test set

predictions = model.predict(x\_test)

predicted\_label = np.argmax(predictions[0])

true\_label = y\_test[0]

# Display the image and the prediction

plt.figure()

plt.imshow(x\_test[0], cmap=plt.cm.binary)

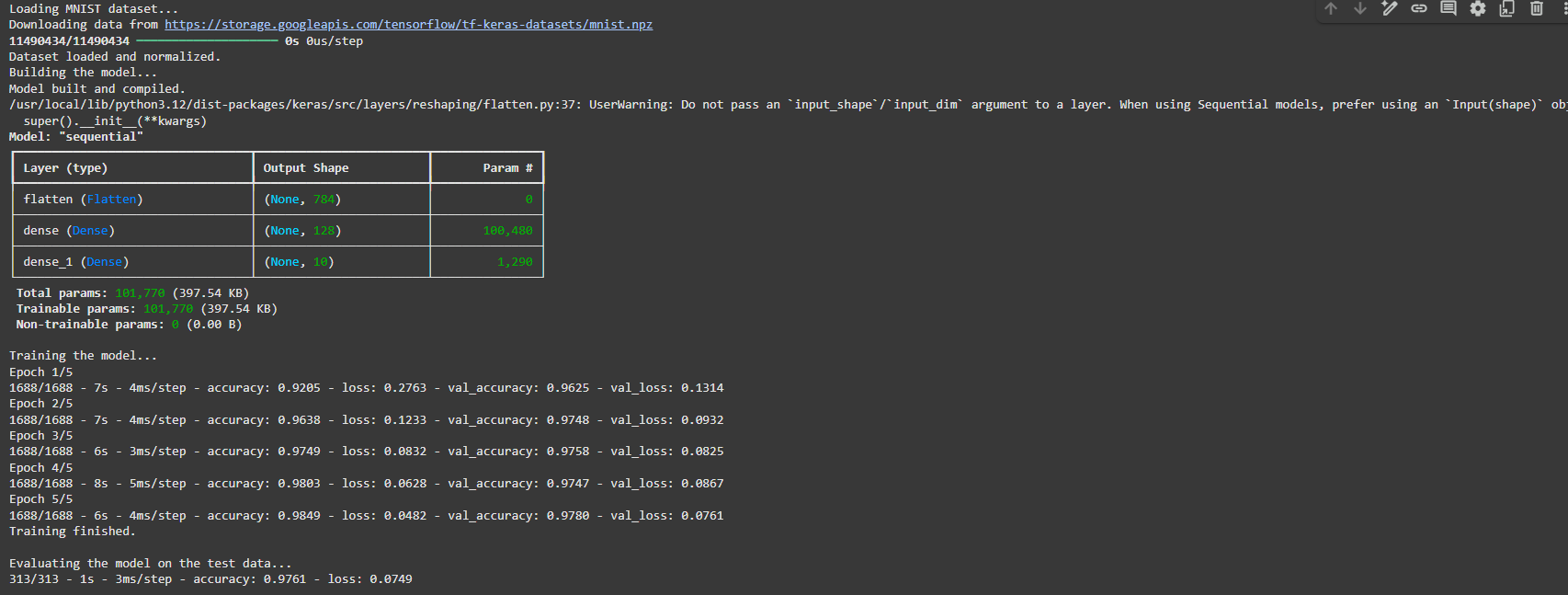
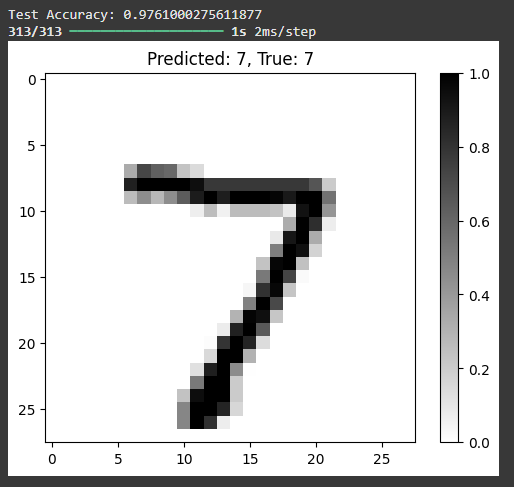
plt.title(f"Predicted: {predicted\_label}, True: {true\_label}")

plt.colorbar()

plt.grid(False)

plt.show()

**Output:**

**  
**