

print("VINAY PANCHAL/27/TYCSB")

from collections import deque,defaultdict class Graph:

def init (self): self.graph=defaultdict(list)

def add\_edge(self,u,v):

#For undirected graph self.graph[u].append(v) self.graph[v].append(u)

def bfs(self,start):

visited=set() queue=deque([start]) traversal=[]

while queue: node=queue.popleft() if node not in visited:

visited.add(node) traversal.append(node)

for neighbor in self.graph[node]: queue.append(neighbor)

return traversal

def dfs\_iterative(self,start): visited=set() stack=[start] traversal=[]

while stack:

node=stack.pop()

if node not in visited: visited.add(node) traversal.append(node)

for neighbor in reversed(self.graph[node]): stack.append(neighbor)

return traversal

g=Graph() edges=[

('A','B'),('A','C'),

('B','D'),('B','E'),

('C','F'),('E','G'), ('F','H')

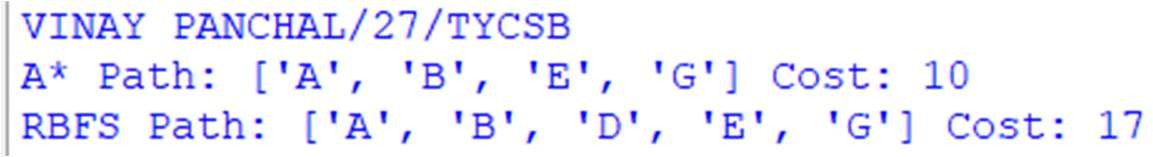
]

#ADDing edges for u,v in edges:

g.add\_edge(u,v) start\_node='A'

print("DFS Traversal: ",g.dfs\_iterative(start\_node)) print("BFS Traversal: ",g.bfs(start\_node))

# Output:



print("VINAY PANCHAL/27/TYCSB")

import heapq

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

open\_list=[] heapq.heappush(open\_list,(h[start],0,start,[start])) visited=set()

while open\_list: f,g,node,path=heapq.heappop(open\_list) if node==goal:

return path,g visited.add(node)

for neighbor,cost in graph[node]:

if neighbor not in visited:

g\_new=g+cost f\_new=g\_new+h[neighbor]

heapq.heappush(open\_list,(f\_new,g\_new,neighbor,path+[neighbor])) return None,float('inf')

def rbfs(graph,h,start,goal):

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

if node==goal:

return path,g successors=[]

for neighbor,cost in graph[node]:

if neighbor not in path:

g\_new=g+cost f=max(g\_new+h[neighbor],f\_limit)

successors.append((f,neighbor,g\_new,path+[neighbor])) if not successors:

return None,float('inf') successors.sort()

while successors:

best=successors[0]

alternative=successors[1][0] if len(successors)>1 else float('inf') result,f\_new=rbfs\_helper(best[1],best[3],best[2],min(f\_limit,alternative)) if result is not None:

return result,f\_new successors[0]=(f\_new,best[1],best[2],best[3]) successors.sort()

return None,float('inf')

return rbfs\_helper(start,[start],0,float('inf')) graph={

'A':[('B',5),('C',10)],

'B':[('A',5),('D',4),('E',3)],

'C':[('A',10),('G',2)],

'D':[('B',4),('E',6)],

'E':[('B',3,),('D',6),('G',2)], 'G':[('C',2),('E',2)]

}

heuristic={ 'A':7,

'B':6,

'C':4,

'D':3,

'E':2,

'G':0

}

start='A' goal='G' #Run A\*

a\_path,a\_cost=a\_star(graph,heuristic,start,goal) print("A\*Path:",a\_path,"Cost",a\_cost)

#Run RBFS

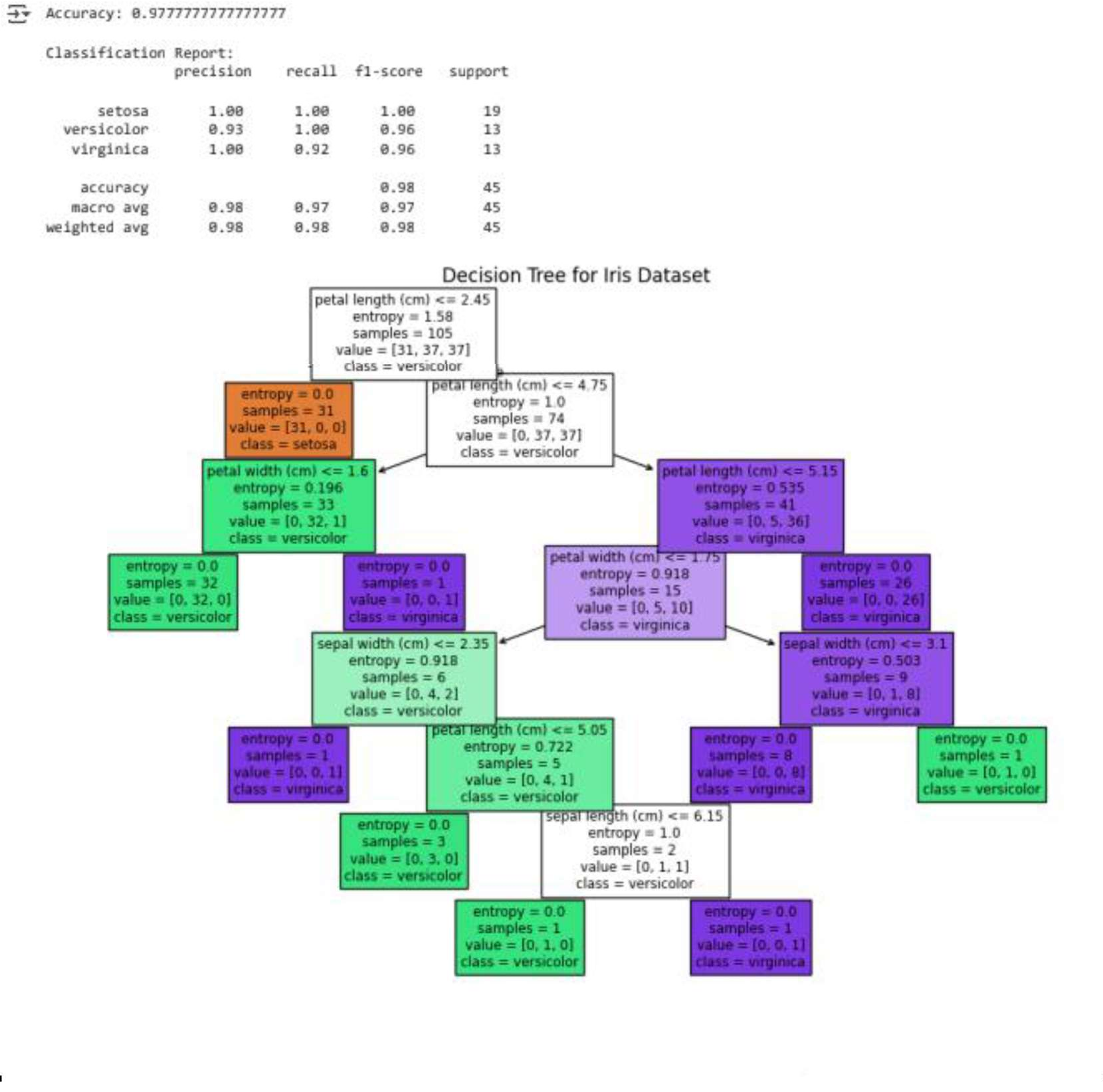
rbfs\_path,rbfs\_cost=rbfs(graph,heuristic,start,goal) print("RBFS Path:",rbfs\_path,"Cost:",rbfs\_cost)

# Output:

**Code:**

from sklearn.datasets import load\_iris

# Practical – 3 Decision Tree Learning



from sklearn.model\_selection import train\_test\_split from sklearn.tree import DecisionTreeClassifier,plot\_tree

from sklearn.metrics import accuracy\_score,classification\_report import matplotlib.pyplot as plt

#load the iris dataset iris = load\_iris()

X = iris.data y = iris.target

feature\_names = iris.feature\_names target\_names = iris.target\_names #split into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42) #craete and train the decision tree classifier

elf = DecisionTreeClassifier(criterion='entropy', random\_state=42) elf.fit(X\_train,y\_train)

#make predictions

y\_pred = elf.predict(X\_test) #evaluate accuracy

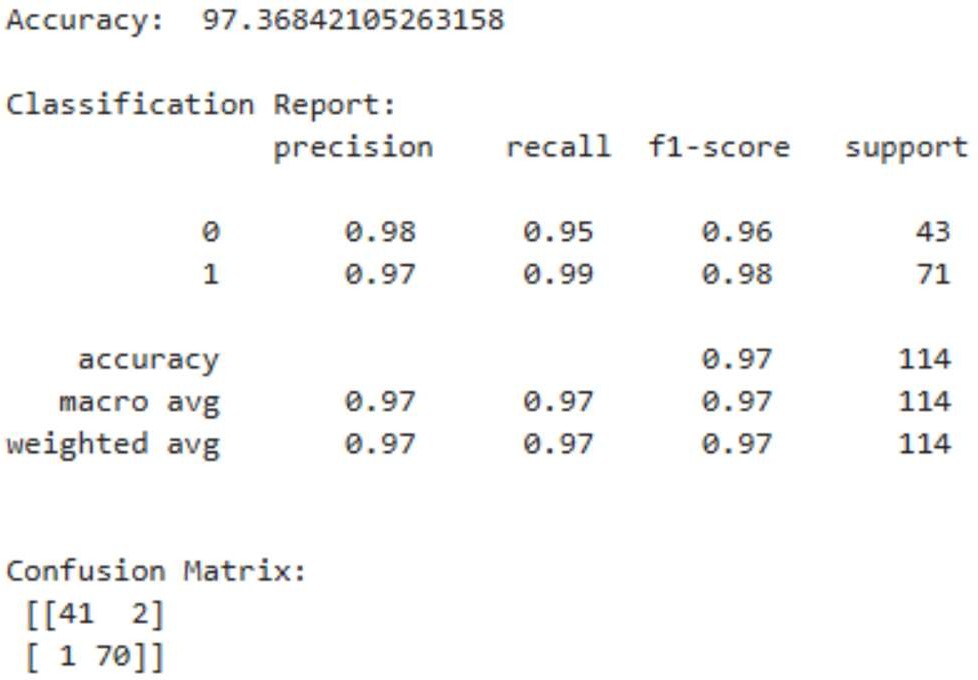
accuracy = accuracy\_score(y\_test, y\_pred) print("Accuracy",accuracy)

print("\nClassification Report",classification\_report(y\_test, y\_pred, target\_names=target\_names)) #Plot the decision tree

plt.figure(figsize=(12,8)) plot\_tree(elf,feature\_names=feature\_names,class\_names=target\_names,filled=True) plt.title('Decision Tree for Classifier')

plt.show()

# Output:



**Code:**

from sklearn.datasets import load\_breast\_cancer

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

from sklearn.preprocessing import StandardScaler

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

#load binary classification dataset data=load\_breast\_cancer()

X=data.data y=data.target#Labels:0=Malignant,1=Benign #Normalized features scalar=StandardScaler() X\_scaled=scalar.fit\_transform(X)

#Split into training and testing data X\_train,X\_test,y\_train,y\_test=train\_test\_split(X\_scaled,y,test\_size=0.2,random\_state=42) #Build SVM model

svm\_model=SVC(kernel='rbf',C=1.0, gamma='scale') svm\_model.fit(X\_train,y\_train)

#MAke Prediction y\_pred=svm\_model.predict(X\_test) #Evaluate Performance

print("Accuracy:",accuracy\_score(y\_test,y\_pred)\*100) print("\nClassifcation Report:\n",classification\_report(y\_test,y\_pred)) print("Confusion Matrix:\n",confusion\_matrix(y\_test,y\_pred)) #Hyperparameter tuning using GridSearchCV

param\_grid={ 'C':[0.1,1,10],

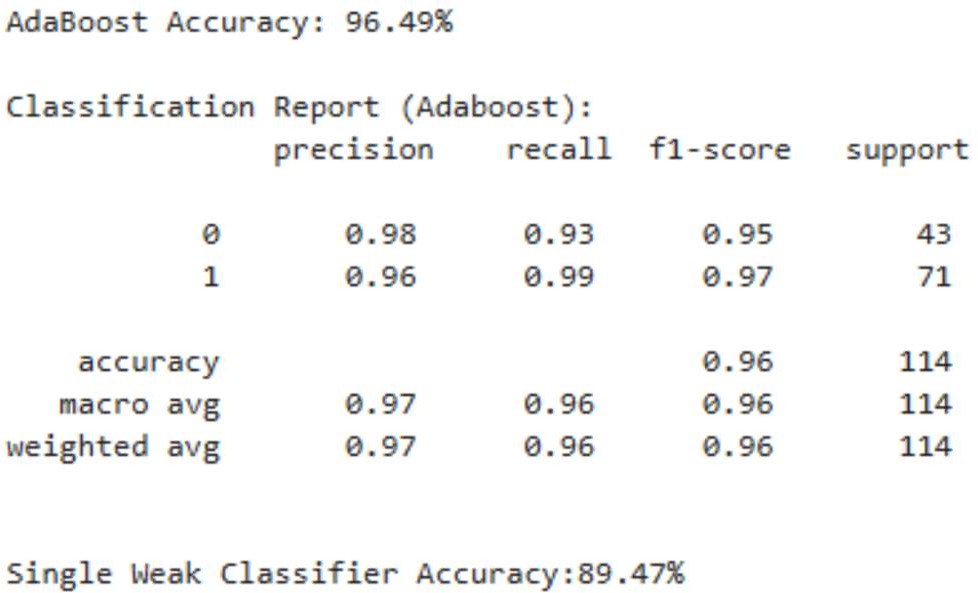
'gamma':['scale',0.1,1], 'kernel':['rbf']

}

grid=GridSearchCV(SVC(),param\_grid,cv=5) grid.fit(X\_train,y\_train)

print("\nBest Parameters:",grid.best\_params\_) print("Best Cross-validation score:",grid.best\_score\_)

# Output:



**Code:**

from sklearn.datasets import load\_breast\_cancer from sklearn.model\_selection import train\_test\_split from sklearn.ensemble import AdaBoostClassifier from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score,classification\_report #Load dataset

data=load\_breast\_cancer()

X=data.data y=data.target #Train-test split

X\_train,X\_test,y\_train,y\_test=train\_test\_split(X,y,test\_size=0.2,random\_state=42) #weak classifier:Decision stump(1-level decision tree) weak\_classifier=DecisionTreeClassifier(max\_depth=5)

#Train Adaboost model with 50 weak classifiers adaboost\_model=AdaBoostClassifier(estimator=weak\_classifier,n\_estimators=50,random\_state=42) adaboost\_model.fit(X\_train,y\_train)

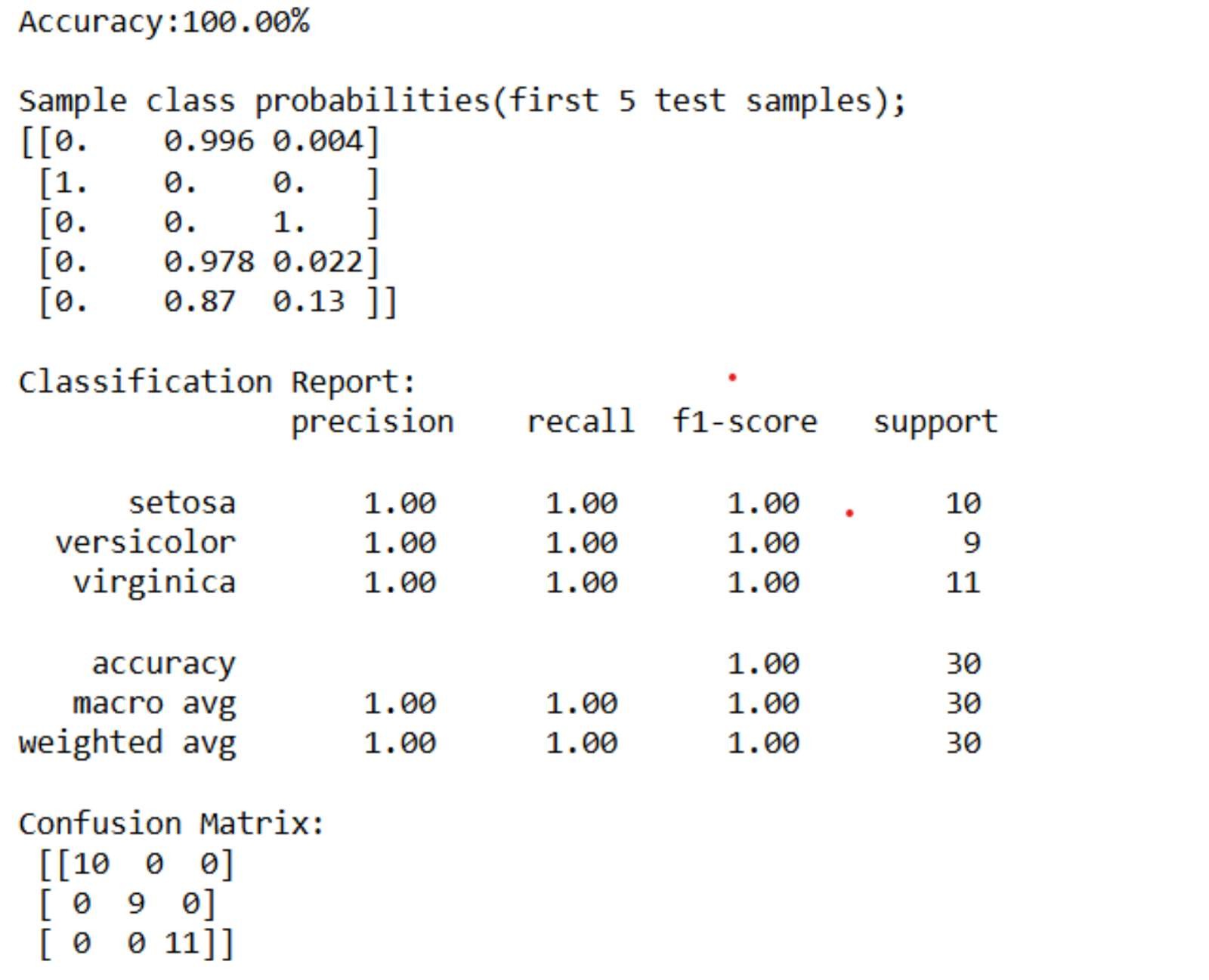
#Predict and evaluate y\_pred=adaboost\_model.predict(X\_test) accuracy\_ada=accuracy\_score(y\_test,y\_pred) print(f"AdaBoost Accuracy: {accuracy\_ada\*100:.2f}%")

print("\nClassification\_report(AdaBoost):\n ",classification\_report(y\_test,y\_pred)) #compare with a single weak classifier

weak\_classifier.fit(X\_train,y\_train) y\_pred\_weak=weak\_classifier.predict(X\_test) accuracy\_weak=accuracy\_score(y\_test,y\_pred\_weak)

print(f"\nSingle weak classifier Accuracy: {accuracy\_weak\*100:.2f}%")

# Output:



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 import numpy as np

#load output data=load\_iris() X=data.data y=data.target

class\_names=data.target\_names #Split into train test

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

model=GaussianNB() model.fit(X\_train,y\_train) #predict y\_pred=model.predict(X\_test) #Evaluate Accuracy

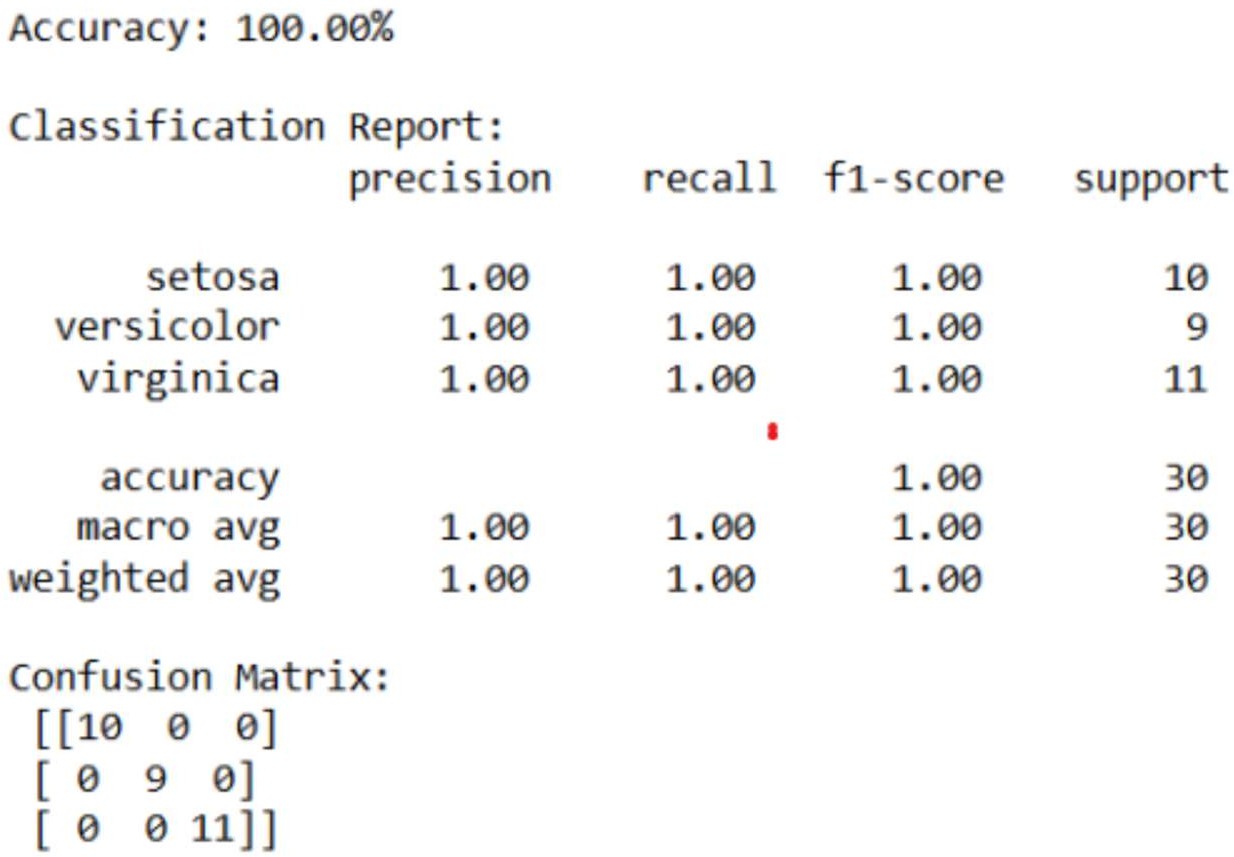
accuracy=accuracy\_score(y\_test,y\_pred) print(f"Accuracy: {accuracy\*100:.2f}%\n") #class probabilities probs=model.predict\_proba(X\_test)

print("Sample class probabilities(first 5test samples):") print(np.round(probs[:5],3))

#detailed evaluation

print("\nClassifier Report:\n",classification\_report(y\_test,y\_pred,target\_names=class\_names)) print("Confusion matrix:\n",confusion\_matrix(y\_test,y\_pred))

# Output:



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, confusion\_matrix, classification\_report

#load dataset data = load\_iris()

X = data.data y = data.target

class\_names = data.target\_names #Train-test split

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

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train) X\_test = scaler.transform(X\_test) #intialize KNN classifier

knn = KNeighborsClassifier(n\_neighbors=3) knn.fit(X\_train, y\_train)

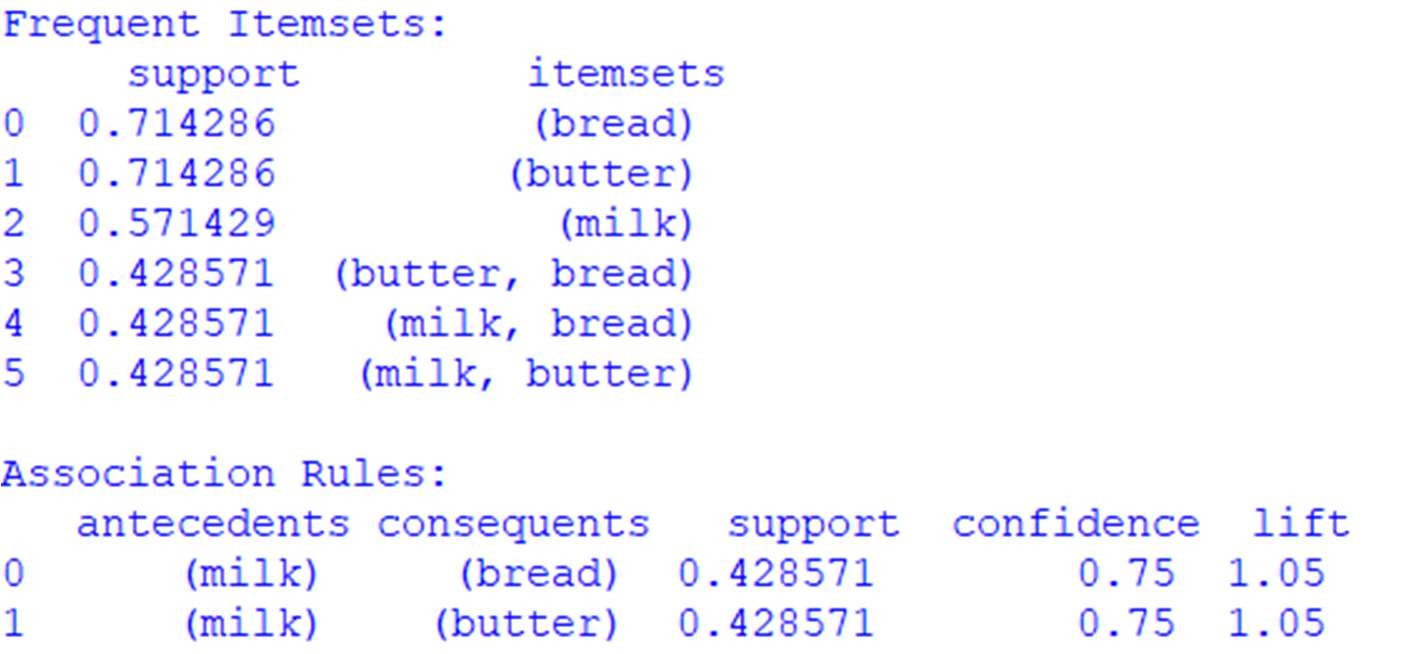
#predict

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

accuracy = accuracy\_score(y\_test, y\_pred) print(f"Accuracy: {accuracy\*100:.2f}%")

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred, target\_names=class\_names)) print("Confusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

# Output:



**Code:**

import pandas as p

from mlxtend.frequent\_patterns import apriori, association\_rules #sample market basket dataset

dataset=[ ['milk','bread','butter'],

['bread','butter'],

['milk','bread'],

['milk','butter'], ['bread'], ['milk','bread','butter'], ['butter']

]

#Convert to one-hot encoded DataFrame

from mlxtend.preprocessing import TransactionEncoder te=TransactionEncoder() te\_ary=te.fit(dataset).transform(dataset)

df = p.DataFrame(te\_ary, columns=te.columns\_) #1.Find frequent itemsets wih min support of 0.3

frequent\_itemsets=apriori(df,min\_support=0.3,use\_colnames=True) #2.generate rule with min confidence of 0.7

rules=association\_rules(frequent\_itemsets,metric="confidence",min\_threshold=0.7) #display results

print("Frequency Itemsets:\n",frequent\_itemsets)

print("\nAssociation Rules:\n",rules[['antecedents','consequents','support','confidence','lift']])

**Output:**