**1.Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples.**

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

# Get the number of attributes

num\_attributes = len(training\_data[0]) - 1 # Last column is the target

# Step 1: Initialize hypothesis with most specific values

hypothesis = ['Ø'] \* num\_attributes

# Step 2: Iterate through training data

for example in training\_data:

attributes, label = example[:-1], example[-1]

# Only consider positive examples

if label.lower() == 'yes':

for i in range(num\_attributes):

if hypothesis[i] == 'Ø':

hypothesis[i] = attributes[i]

elif hypothesis[i] != attributes[i]:

hypothesis[i] = '?'

return hypothesis

training\_data = [

['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],

['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],

['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No'],

['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']

]

hypothesis = find\_s\_algorithm(training\_data)

print("Most specific hypothesis:", hypothesis)

**Output:**



**2. For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm in python to output a description of the set of all hypotheses consistent with the training examples**

import pandas as pd

def load\_data(filename):

data = pd.read\_csv(filename)

attributes = data.columns[:-1]

examples = data.values.tolist()

return attributes, examples

def more\_general(h1, h2):

return all((x == '?' or x == y) for x, y in zip(h1, h2))

def candidate\_elimination(examples):

num\_attributes = len(examples[0]) - 1

# Initialize S and G

S = ['Ø'] \* num\_attributes

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

for example in examples:

x, y = example[:-1], example[-1].lower()

if y == 'yes':

# Remove any g in G inconsistent with x

G = [g for g in G if more\_general(g, x)]

# Update S to be more general only where needed

for i in range(num\_attributes):

if S[i] == 'Ø':

S[i] = x[i]

elif S[i] != x[i]:

S[i] = '?'

elif y == 'no':

# Remove any s in S consistent with x

if more\_general(S, x):

S = ['Ø'] \* num\_attributes

new\_G = []

for g in G:

if more\_general(g, x):

for i in range(num\_attributes):

if g[i] == '?':

for val in set(e[i] for e in examples if e[-1].lower() == 'yes'):

if val != x[i]:

new\_hypothesis = g.copy()

new\_hypothesis[i] = val

if more\_general(new\_hypothesis, S):

new\_G.append(new\_hypothesis)

G = new\_G

return S, G

filename = "training\_data.csv"

attributes, data = load\_data(filename)

S\_final, G\_final = candidate\_elimination(data)

print("Final Specific Hypothesis (S):")

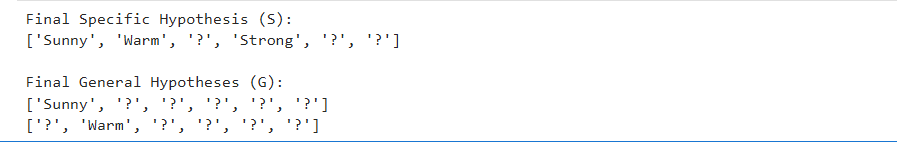
print(S\_final)

print("\nFinal General Hypotheses (G):")

for g in G\_final:

print(g)

**Output:**

****

**3. Demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate  
data set for building the decision tree and apply this knowledge to classify  
a new sample.**

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.preprocessing import LabelEncoder

# Step 1: Create the dataset

data = {

'Outlook': ['Sunny','Sunny','Overcast','Rain','Rain','Rain','Overcast',

'Sunny','Sunny','Rain','Sunny','Overcast','Overcast','Rain'],

'Temperature': ['Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild',

'Cool','Mild','Mild','Mild','Hot','Mild'],

'Humidity': ['High','High','High','High','Normal','Normal','Normal',

'High','Normal','Normal','Normal','High','Normal','High'],

'Wind': ['Weak','Strong','Weak','Weak','Weak','Strong','Strong','Weak',

'Weak','Weak','Strong','Strong','Weak','Strong'],

'PlayTennis': ['No','No','Yes','Yes','Yes','No','Yes','No',

'Yes','Yes','Yes','Yes','Yes','No']

}

df = pd.DataFrame(data)

# Step 2: Encode the categorical features

le\_dict = {}

for col in df.columns:

le = LabelEncoder()

df[col] = le.fit\_transform(df[col])

le\_dict[col] = le

# Step 3: Split into features and target

X = df.drop('PlayTennis', axis=1)

y = df['PlayTennis']

# Step 4: Train Decision Tree using ID3 (entropy)

model = DecisionTreeClassifier(criterion='entropy')

model.fit(X, y)

# Step 5: Show internal decision rules as text

from sklearn.tree import export\_text

tree\_rules = export\_text(model, feature\_names=list(X.columns))

print("ID3 Decision Tree Rules:\n")

print(tree\_rules)

sample = pd.DataFrame([[

le\_dict['Outlook'].transform(['Sunny'])[0],

le\_dict['Temperature'].transform(['Cool'])[0],

le\_dict['Humidity'].transform(['High'])[0],

le\_dict['Wind'].transform(['Strong'])[0]

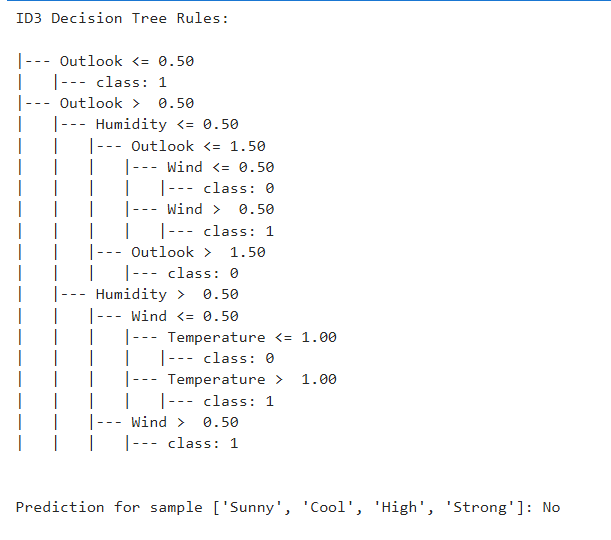
]], columns=X.columns)

prediction = model.predict(sample)

result = le\_dict['PlayTennis'].inverse\_transform(prediction)[0]

print(f"\nPrediction for sample ['Sunny', 'Cool', 'High', 'Strong']: {result}")

**Output:**



**4. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.**

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.preprocessing import OneHotEncoder, StandardScaler

# Load and preprocess the Iris dataset

iris = load\_iris()

X = iris.data

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

# One-hot encode the output

encoder = OneHotEncoder(sparse\_output=False)

y\_encoded = encoder.fit\_transform(y)

# Normalize features

scaler = StandardScaler()

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

# Train/test split

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

# Define the neural network class

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, lr=0.1):

self.lr = lr

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

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

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

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

def sigmoid(self, x):

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

def sigmoid\_derivative(self, x):

return x \* (1 - x)

def forward(self, X):

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

self.a1 = self.sigmoid(self.z1)

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

self.a2 = self.sigmoid(self.z2)

return self.a2

def backward(self, X, y, output):

error = y - output

d\_output = error \* self.sigmoid\_derivative(output)

error\_hidden = d\_output.dot(self.W2.T)

d\_hidden = error\_hidden \* self.sigmoid\_derivative(self.a1)

# Update weights and biases

self.W2 += self.a1.T.dot(d\_output) \* self.lr

self.b2 += np.sum(d\_output, axis=0, keepdims=True) \* self.lr

self.W1 += X.T.dot(d\_hidden) \* self.lr

self.b1 += np.sum(d\_hidden, axis=0, keepdims=True) \* self.lr

def train(self, X, y, epochs=1000):

for epoch in range(epochs):

output = self.forward(X)

self.backward(X, y, output)

if epoch % 100 == 0:

loss = np.mean((y - output) \*\* 2)

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

def predict(self, X):

output = self.forward(X)

return np.argmax(output, axis=1)

# Initialize and train the model

nn = NeuralNetwork(input\_size=4, hidden\_size=6, output\_size=3, lr=0.1)

nn.train(X\_train, y\_train, epochs=1000)

# Test the model

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

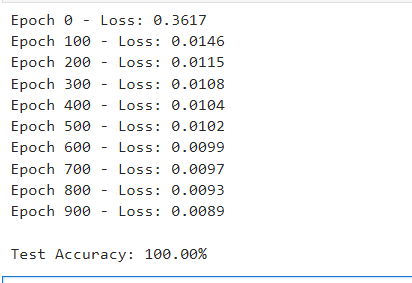
y\_true = np.argmax(y\_test, axis=1)

# Evaluate accuracy

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

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

**Output:**

****