EXP-1 FIND S ALGORITHM

# FIND-S Algorithm Implementation

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

# Step 1: Initialize with the most specific hypothesis

hypothesis = ['0'] \* len(training\_data[0][:-1]) # Exclude target column

for example in training\_data:

# If the example is positive

if example[-1] == "Yes":

# Update the hypothesis

for i in range(len(hypothesis)):

if hypothesis[i] == '0': # Initialize to first positive example

hypothesis[i] = example[i]

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

hypothesis[i] = '?' # Generalize the differing attributes

return hypothesis

# Sample Training Data

training\_data = [

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

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

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

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

]

# Apply FIND-S Algorithm

final\_hypothesis = find\_s\_algorithm(training\_data)

# Print the result

print("Most Specific Hypothesis:", final\_hypothesis)

OUTPUT:

Most Specific Hypothesis:

['Sunny', 'Warm', '?', 'Strong']

**EXP2**: Candidate-Elimination algorithm

import pandas as pd

# Load dataset

data = pd.read\_csv(r"C:\Users\bhadr\OneDrive\Documents\can.csv")

# Extract attributes and examples

attributes = list(data.columns)[:-1]

examples = data.values

# Initialize boundaries

S = ['0'] \* len(attributes) # Most specific hypothesis

G = [['?'] \* len(attributes)] # Most general hypothesis

# Candidate-Elimination Algorithm

for example in examples:

if example[-1] == 'Yes': # Positive example

# Update S: Generalize to cover the example

for i in range(len(S)):

if S[i] == '0':

S[i] = example[i]

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

S[i] = '?'

# Update G: Remove inconsistent hypotheses

G = [g for g in G if all(g[i] == '?' or g[i] == example[i] for i in range(len(g)))]

else: # Negative example

# Update G: Specialize to exclude the example

new\_G = []

for g in G:

for i in range(len(g)):

if g[i] == '?': # Specialization only on '?'

new\_h = g.copy()

new\_h[i] = example[i]

# Keep only hypotheses consistent with S

if all(new\_h[j] == '?' or new\_h[j] == S[j] for j in range(len(new\_h))):

new\_G.append(new\_h)

G = new\_G

# Output results

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

print("Final General Hypotheses (G):", G)

print(data.head())

**EXP3**: Decision tree

import pandas as pd

import math

from collections import Counter

# Load dataset

data = pd.DataFrame({

"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"]

})

# Calculate entropy

def entropy(data):

target = data["PlayTennis"]

counts = Counter(target)

total = len(target)

return -sum((count / total) \* math.log2(count / total) for count in counts.values())

# Calculate information gain

def information\_gain(data, attribute):

total\_entropy = entropy(data)

values = data[attribute].unique()

subset\_entropy = 0

total = len(data)

for value in values:

subset = data[data[attribute] == value]

subset\_entropy += (len(subset) / total) \* entropy(subset)

return total\_entropy - subset\_entropy

# ID3 Algorithm

def id3(data, attributes, depth=0):

# Base cases

if len(data["PlayTennis"].unique()) == 1: # Pure dataset

return data["PlayTennis"].iloc[0]

if not attributes: # No attributes left

return data["PlayTennis"].mode()[0]

# Find best attribute to split

gains = {attr: information\_gain(data, attr) for attr in attributes}

best\_attr = max(gains, key=gains.get)

# Build tree

tree = {best\_attr: {}}

remaining\_attributes = [attr for attr in attributes if attr != best\_attr]

for value in data[best\_attr].unique():

subset = data[data[best\_attr] == value]

if subset.empty:

tree[best\_attr][value] = data["PlayTennis"].mode()[0]

else:

tree[best\_attr][value] = id3(subset, remaining\_attributes, depth + 1)

return tree

# Build tree

attributes = list(data.columns[:-1])

decision\_tree = id3(data, attributes)

# Display decision tree

import pprint

pprint.pprint(decision\_tree)

# Classify a new sample

def classify(tree, sample):

if not isinstance(tree, dict): # Leaf node

return tree

attr = next(iter(tree))

value = sample[attr]

return classify(tree[attr][value], sample)

# Example: Classify a new sample

sample = {"Outlook": "Sunny", "Temperature": "Cool", "Humidity": "High", "Wind": "Strong"}

result = classify(decision\_tree, sample)

print("\nClassified as:", result)

OUTPUT:

{'Outlook': {'Overcast': 'Yes',

'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},

'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}

Classified as: No

**EXP4**: Artificial Neural Network

import numpy as np

# Initialize dataset (XOR problem)

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

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

# Set the seed for reproducibility

np.random.seed(42)

# Initialize parameters

input\_layer\_neurons = 2

hidden\_layer\_neurons = 2

output\_neurons = 1

# Weights and biases initialization

weights\_input\_hidden = np.random.uniform(size=(input\_layer\_neurons, hidden\_layer\_neurons))

bias\_hidden = np.random.uniform(size=(1, hidden\_layer\_neurons))

weights\_hidden\_output = np.random.uniform(size=(hidden\_layer\_neurons, output\_neurons))

bias\_output = np.random.uniform(size=(1, output\_neurons))

# Learning rate

learning\_rate = 0.5

# Training the neural network

epochs = 10000

for epoch in range(epochs):

# Forward propagation

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

hidden\_layer\_output = 1 / (1 + np.exp(-hidden\_layer\_input)) # Sigmoid activation

output\_layer\_input = np.dot(hidden\_layer\_output, weights\_hidden\_output) + bias\_output

predicted\_output = 1 / (1 + np.exp(-output\_layer\_input)) # Sigmoid activation

# Calculate error

error = y - predicted\_output

# Backpropagation

d\_predicted\_output = error \* (predicted\_output \* (1 - predicted\_output))

error\_hidden\_layer = d\_predicted\_output.dot(weights\_hidden\_output.T)

d\_hidden\_layer = error\_hidden\_layer \* (hidden\_layer\_output \* (1 - hidden\_layer\_output))

# Update weights and biases

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

bias\_output += np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learning\_rate

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

bias\_hidden += np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* learning\_rate

# Output the final trained results

print("Final hidden weights: ", weights\_input\_hidden)

print("Final hidden biases: ", bias\_hidden)

print("Final output weights: ", weights\_hidden\_output)

print("Final output biases: ", bias\_output)

print("\nOutput from neural network after 10,000 epochs: \n", predicted\_output)

OUTPUT:

Final hidden weights: [[4.59244504 6.47246975]

[4.5971031 6.49153682]]

Final hidden biases: [[-7.05239171 -2.8842842 ]]

Final output weights: [[-10.32676834]

[ 9.62121009]]

Final output biases: [[-4.44969307]]

Output from neural network after 10,000 epochs:

[[0.01890475]

[0.98371361]

[0.98369334]

[0.01686123]]

**EXP5**: K-Nearest Neighbours

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,confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

iris = load\_iris()

X = iris.data

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, test\_size=0.25, random\_state = 0)

model = KNeighborsClassifier(n\_neighbors = 5)

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

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

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

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

print("Confusion Matrix: ", confusion)

print("Accuracy:" ,accuracy)

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

sns.heatmap(confusion, annot=True, fmt='d', cmap='Blues',

xticklabels=iris.target\_names, yticklabels=iris.target\_names)

plt.ylabel("Actual")

plt.xlabel("Confusion Matrix")

plt.show()

OUTPUT:

Confusion Matrix: [[13 0 0]

[ 0 15 1]

[ 0 0 9]]

Accuracy: 0.9736842105263158