**Experiment-1**

Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.

***FIND-S Algorithm***

1. Initialize h to the most specific hypothesis in H

2. **for** each positive training instance x

**for** each attribute constraint ai in h

**if** the constraint ai is satisfied by x **then**

do nothing

**else**

replace ai in h by the next more general constraint that is satisfied by x

3. Output hypothesis h

***Training Examples:***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sky | airtemp | humidity | wind | water | forcast | enjoysport |
| 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 |

***Program:***

import csv

num\_attributes = 6

a = []

print("\n The Given Training Data Set \n")

with open('enjoysport.csv', 'r') as csvfile:

reader = csv.reader(csvfile)

for row in reader:

a.append (row)

print(row)

print("\n The initial value of hypothesis: ")

hypothesis = ['0'] \* num\_attributes

print(hypothesis)

for j in range(0,num\_attributes):

hypothesis[j] = a[0][j];

print("\n Find S: Finding a Maximally Specific Hypothesis\n")

for i in range(0,len(a)):

if a[i][num\_attributes]=='yes':

for j in range(0,num\_attributes):

if a[i][j]!=hypothesis[j]:

hypothesis[j]='?'

else :

hypothesis[j]= a[i][j]

print(" For Training instance No:{0} the hypothesis is ".format(i),hypothesis)

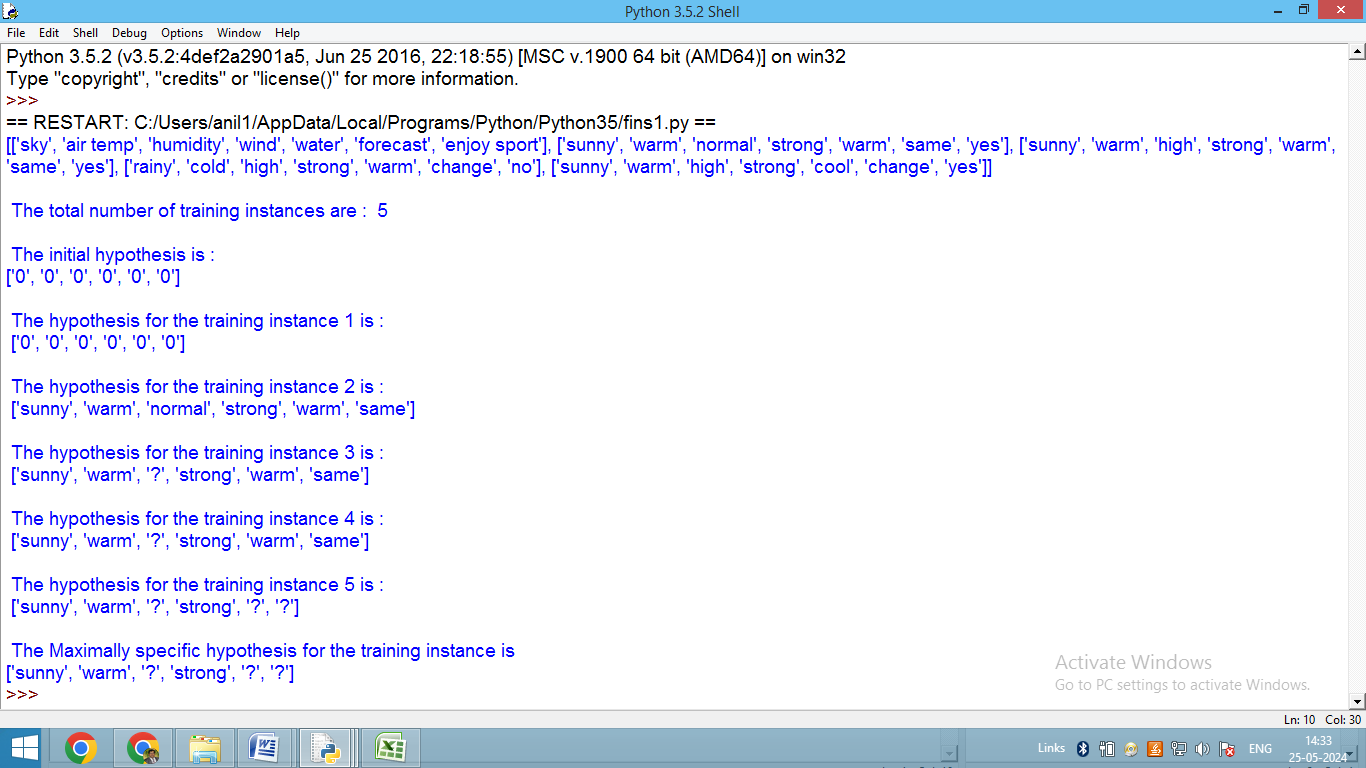
print("\n The Maximally Specific Hypothesis for a given Training Examples :\n")

print(hypothesis)

**Data Set:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |

Output:



The Given Training Data Set

['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']

The initial value of hypothesis:

['0', '0', '0', '0', '0', '0']

Find S: Finding a Maximally Specific Hypothesis

For Training Example No:0 the hypothesis is

['sunny', 'warm', 'normal', 'strong', 'warm', 'same']

For Training Example No:1 the hypothesis is

['sunny', 'warm', '?', 'strong', 'warm', 'same']

For Training Example No:2 the hypothesis is

['sunny', 'warm', '?', 'strong', 'warm', 'same']

For Training Example No:3 the hypothesis is

['sunny', 'warm', '?', 'strong', '?', '?']

The Maximally Specific Hypothesis for a given Training Examples:

**['sunny', 'warm', '?', 'strong', '?', '?']**

**Experiment-2**

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sky | airtemp | humidity | wind | water | forcast | enjoysport |
| 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 |

**Program**

import numpy as np

import pandas as pd

data = pd.read\_csv('enjoysport.csv')

concepts = np.array(data.iloc[:,0:-1])

print(concepts)

target = np.array(data.iloc[:,-1])

print(target)

def learn(concepts, target):

specific\_h = concepts[0].copy()

print("initialization of specific\_h and general\_h")

print(specific\_h)

general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]

print(general\_h)

for i, h in enumerate(concepts):

print("For Loop Starts")

if target[i] == "yes":

print("If instance is Positive ")

for x in range(len(specific\_h)):

if h[x]!= specific\_h[x]:

specific\_h[x] ='?'

general\_h[x][x] ='?'

if target[i] == "no":

print("If instance is Negative ")

for x in range(len(specific\_h)):

if h[x]!= specific\_h[x]:

general\_h[x][x] = specific\_h[x]

else:

general\_h[x][x] = '?'

print(" steps of Candidate Elimination Algorithm",i+1)

print(specific\_h)

print(general\_h)

print("\n")

print("\n")

indices = [i for i, val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]

for i in indices:

general\_h.remove(['?', '?', '?', '?', '?', '?'])

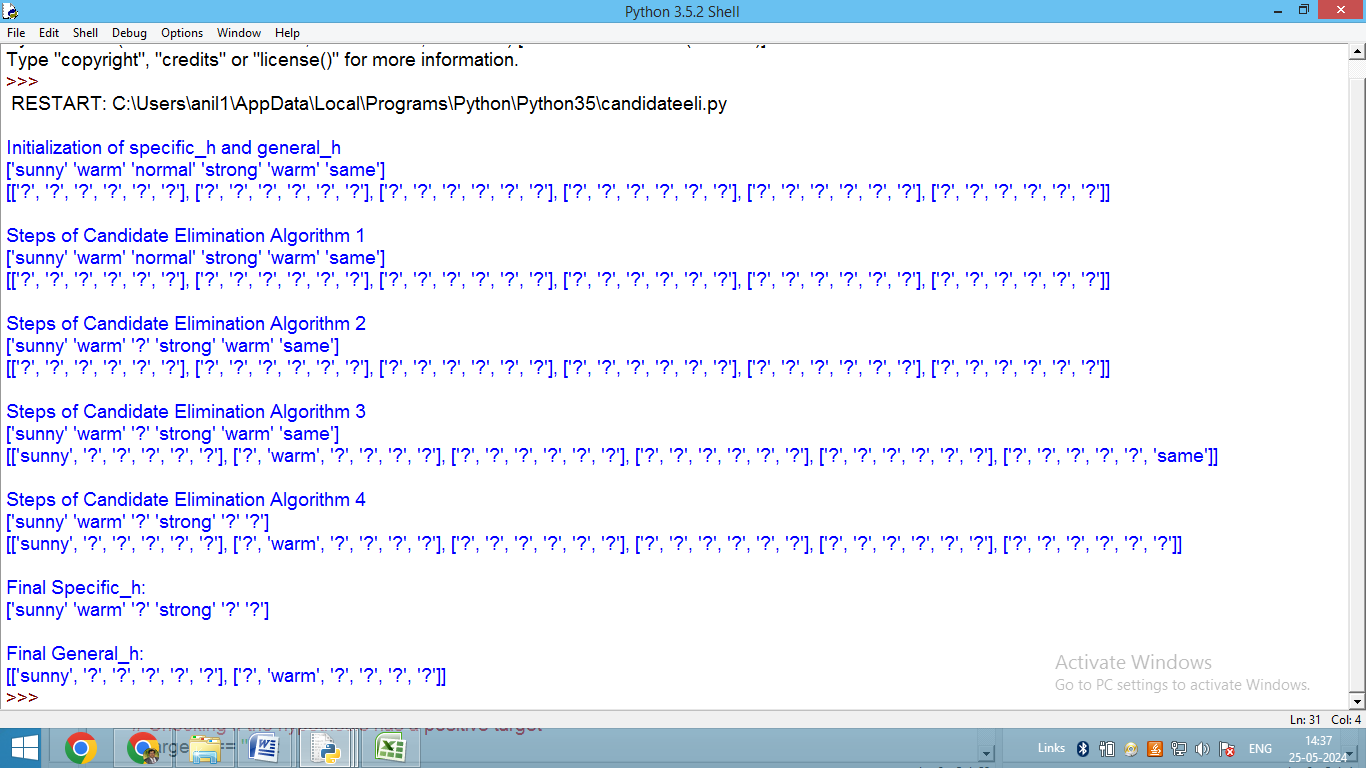
return specific\_h, general\_h

s\_final, g\_final = learn(concepts, target)

print("Final Specific\_h:", s\_final, sep="\n")

print("Final General\_h:", g\_final, sep="\n")

OUTPUT:



Initialization of specific\_h and general\_h

['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 1

['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 2

['sunny' 'warm' '?' 'strong' 'warm' 'same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 3

['sunny' 'warm' '?' 'strong' 'warm' 'same']

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]

Steps of Candidate Elimination Algorithm 4

['sunny' 'warm' '?' 'strong' '?' '?']

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific\_h:

['sunny' 'warm' '?' 'strong' '?' '?']

Final General\_h:

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]

**Experiment-3**

Write a program to 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.

**ID3 Steps**

1. Calculate the Information Gain of each feature.
2. Considering that all rows don’t belong to the same class, split the dataset **S**into subsets using the feature for which the Information Gain is maximum.
3. Make a decision tree node using the feature with the maximum Information gain.
4. If all rows belong to the same class, make the current node as a leaf node with the class as its label.
5. Repeat for the remaining features until we run out of all features, or the decision tree has all leaf nodes.

We denote our dataset as **S,**entropy is calculated as:

**Entropy(S) = - ∑ pᵢ \* log**₂**(pᵢ) ; i = 1 to n**

where,  
**n** is the total number of classes in the target column (in our case n = 2 i.e YES and NO)  
**pᵢ** is the **probability of class ‘i’**or the ratio of “number of rows with class i in the target column” to the “total number of rows” in the dataset.Information Gain for a feature column **A**is calculated as:

**IG(S, A) = Entropy(S) - ∑((|Sᵥ| / |S|) \* Entropy(Sᵥ))**

where **Sᵥ**is the set of rows in **S** for which the feature column **A** has value **v**, |**Sᵥ**| is the number of rows in **Sᵥ**and likewise **|S|** is the number of rows in **S.**

import math

import csv

def load\_csv(filename):

lines=csv.reader(open(filename,"r"));

dataset = list(lines)

headers = dataset.pop(0)

return dataset,headers

class Node:

def \_\_init\_\_(self,attribute):

self.attribute=attribute

self.children=[]

self.answer=""

def subtables(data,col,delete):

dic={}

coldata=[row[col] for row in data]

attr=list(set(coldata))

counts=[0]\*len(attr)

r=len(data)

c=len(data[0])

for x in range(len(attr)):

for y in range(r):

if data[y][col]==attr[x]:

counts[x]+=1

for x in range(len(attr)):

dic[attr[x]]=[[0 for i in range(c)] for j in range(counts[x])]

pos=0

for y in range(r):

if data[y][col]==attr[x]:

if delete:

del data[y][col]

dic[attr[x]][pos]=data[y]

pos+=1

return attr,dic

def entropy(S):

attr=list(set(S))

if len(attr)==1:

return 0

counts=[0,0]

for i in range(2):

counts[i]=sum([1 for x in S if attr[i]==x])/(len(S)\*1.0)

sums=0

for cnt in counts:

sums+=-1\*cnt\*math.log(cnt,2)

return sums

def compute\_gain(data,col):

attr,dic = subtables(data,col,delete=False)

total\_size=len(data)

entropies=[0]\*len(attr)

ratio=[0]\*len(attr)

total\_entropy=entropy([row[-1] for row in data])

for x in range(len(attr)):

ratio[x]=len(dic[attr[x]])/(total\_size\*1.0)

entropies[x]=entropy([row[-1] for row in dic[attr[x]]])

total\_entropy-=ratio[x]\*entropies[x]

return total\_entropy

def build\_tree(data,features):

lastcol=[row[-1] for row in data]

if(len(set(lastcol)))==1:

node=Node("")

node.answer=lastcol[0]

return node

n=len(data[0])-1

gains=[0]\*n

for col in range(n):

gains[col]=compute\_gain(data,col)

split=gains.index(max(gains))

node=Node(features[split])

fea = features[:split]+features[split+1:]

attr,dic=subtables(data,split,delete=True)

for x in range(len(attr)):

child=build\_tree(dic[attr[x]],fea)

node.children.append((attr[x],child))

return node

def print\_tree(node,level):

if node.answer!="":

print(" "\*level,node.answer)

return

print(" "\*level,node.attribute)

for value,n in node.children:

print(" "\*(level+1),value)

print\_tree(n,level+2)

def classify(node,x\_test,features):

if node.answer!="":

print(node.answer)

return

pos=features.index(node.attribute)

for value, n in node.children:

if x\_test[pos]==value:

classify(n,x\_test,features)

'''Main program'''

dataset,features=load\_csv(r'C:\Users\anil1\Desktop\ML KAI601 Notes\ML Lab\id3.csv')

node1=build\_tree(dataset,features)

print("The decision tree for the dataset using ID3 algorithm is")

print\_tree(node1,0)

testdata,features=load\_csv(r'C:\Users\anil1\Desktop\ML KAI601 Notes\ML Lab\id3\_test\_1.csv')

for xtest in testdata:

print("The test instance:",xtest)

print("The label for test instance:",end=" ")

classify(node1,xtest,features)

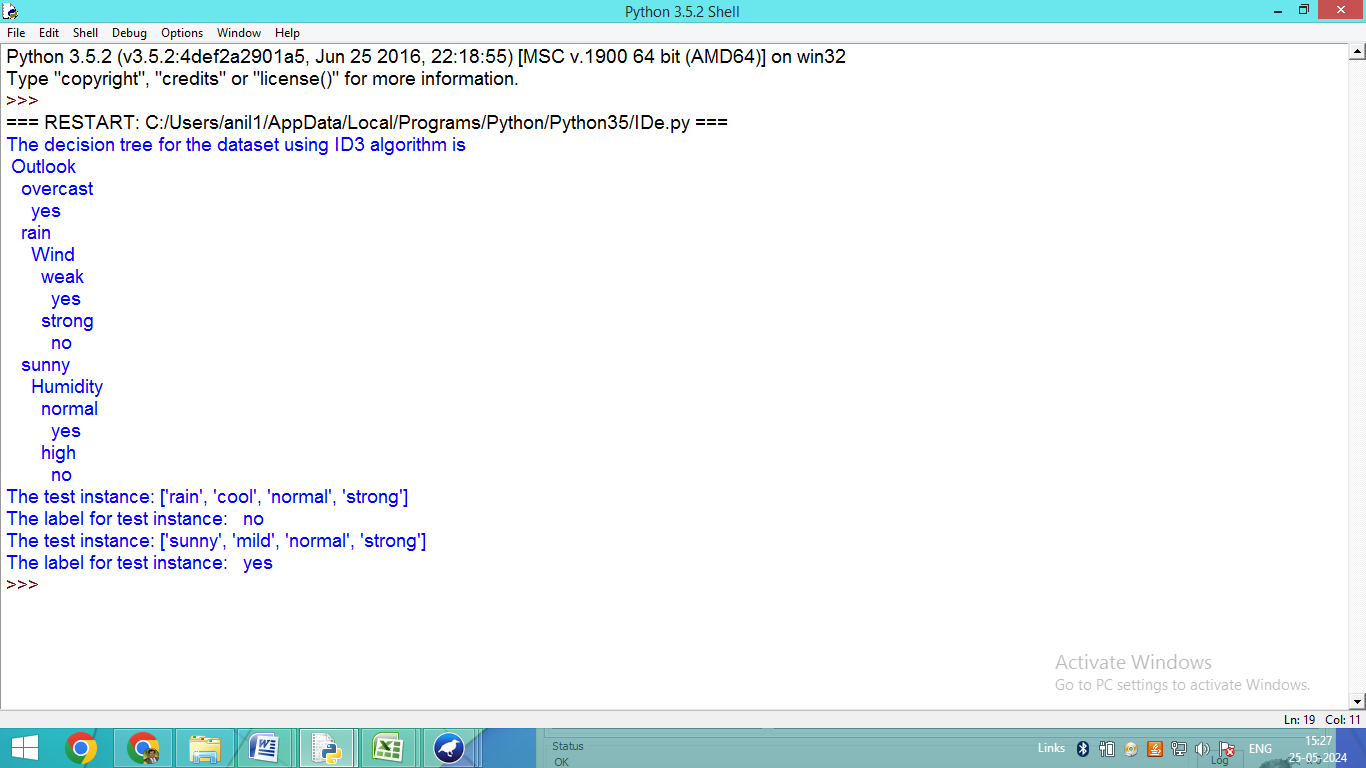
Training Data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Outlook | Temperature | Humidity | Wind | PlayTennis |
| Sunny | hot | high | weak | no |
| Sunny | hot | high | strong | no |
| Overcast | hot | high | weak | yes |
| Rain | mild | high | weak | yes |
| Rain | cool | normal | weak | yes |
| Rain | cool | normal | strong | no |
| Overcast | cool | normal | strong | yes |
| Sunny | mild | high | weak | no |
| Sunny | cool | normal | weak | yes |
| Rain | mild | normal | weak | yes |
| Sunny | mild | normal | strong | yes |
| Overcast | mild | high | strong | yes |
| Overcast | hot | normal | weak | yes |
| Rain | mild | high | strong | no |

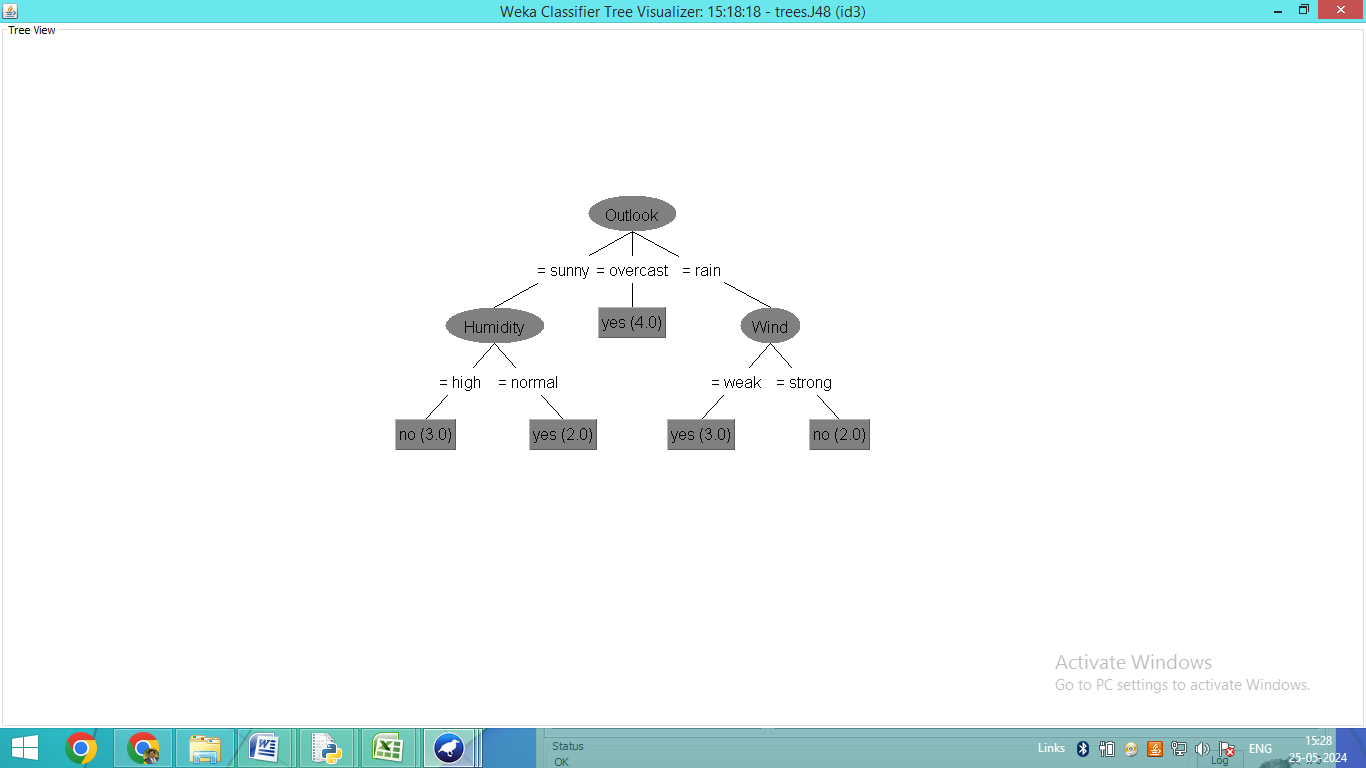
Test Data:

|  |  |  |  |
| --- | --- | --- | --- |
| Outlook | Temperature | Humidity | Wind |
| Rain | cool | normal | strong |
| Sunny | mild | normal | strong |

OUTPUT:



ID3 Tree:



The decision tree for the dataset using ID3 algorithm is

Outlook

overcast

yes

rain

Wind

weak

yes

strong

no

sunny

Humidity

normal

yes

high

no

The test instance: ['rain', 'cool', 'normal', 'strong']

The label for test instance: no

The test instance: ['sunny', 'mild', 'normal', 'strong']

The label for test instance: yes

**Experiment-4**

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

import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float) # two inputs [sleep,study]

y = np.array(([92], [86], [89]), dtype=float) # one output [Expected % in Exams]

X = X/np.amax(X,axis=0) # maximum of X array longitudinally

y = y/100

#Sigmoid Function

def sigmoid (x):

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

#Derivative of Sigmoid Function

def derivatives\_sigmoid(x):

return x \* (1 - x)

#Variable initialization

epoch=5000 #Setting training iterations

lr=0.1 #Setting learning rate

inputlayer\_neurons = 2 #number of features in data set

hiddenlayer\_neurons = 3 #number of hidden layers neurons

output\_neurons = 1 #number of neurons at output layer

#weight and bias initialization

wh=np.random.uniform(size=(inputlayer\_neurons,hiddenlayer\_neurons)) #weight of the link from input node to hidden node

bh=np.random.uniform(size=(1,hiddenlayer\_neurons)) # bias of the link from input node to hidden node

wout=np.random.uniform(size=(hiddenlayer\_neurons,output\_neurons)) #weight of the link from hidden node to output node

bout=np.random.uniform(size=(1,output\_neurons)) #bias of the link from hidden node to output node

#draws a random range of numbers uniformly of dim x\*y

for i in range(epoch):

#Forward Propogation

hinp1=np.dot(X,wh)

hinp=hinp1 + bh

hlayer\_act = sigmoid(hinp)

outinp1=np.dot(hlayer\_act,wout)

outinp= outinp1+ bout

output = sigmoid(outinp)

#Backpropagation

EO = y-output

outgrad = derivatives\_sigmoid(output)

d\_output = EO\* outgrad

EH = d\_output.dot(wout.T)

#how much hidden layer weights contributed to error

hiddengrad = derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer = EH \* hiddengrad

# dotproduct of nextlayererror and currentlayerop

wout += hlayer\_act.T.dot(d\_output) \*lr

wh += X.T.dot(d\_hiddenlayer) \*lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)

Input:

[[0.66666667 1. ]

[0.33333333 0.55555556]

[1. 0.66666667]]

Actual Output:

[[0.92]

[0.86]

[0.89]]

Predicted Output:

[[0.85726252]

[0.83545767]

[0.85611575]]

