3. 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.

**ALGORITHM**

For each training example d, do:

If d is positive example

Remove from G any hypothesis h inconsistent with d

For each hypothesis s in S not consistent with d:

Remove s from S

Add to S all minimal generalizations of s consistent with d and having a generalization in G

Remove from S any hypothesis with a more specific h in S

If d is negative example

Remove from S any hypothesis h inconsistent with d

For each hypothesis g in G not consistent with d:

Remove g from G

Add to G all minimal specializations of g consistent with d and having a specialization in S

Remove from G any hypothesis having a more general hypothesis in G

**PROGRAM**

```
import numpy as np

import pandas as pd

data = pd.read_csv(path+'/enjoysport.csv')

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

print("\nInstances are:\n",concepts)

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

```
print("\nTarget Values are: ",target)
def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("\nInitialization of specific_h and genearal_h")
    print("\nSpecific Boundary: ", specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print("\nGeneric Boundary: ",general_h)
    for i, h in enumerate(concepts):
        print("\nInstance", i+1 , "is ", h)
        if target[i] == "yes":
            print("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("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("Specific Bundary after ", i+1, "Instance is ", specific_h)
        print("Generic Boundary after ", i+1, "Instance is ", general_h)
        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")
```

**DATASET**

| 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 |

**OUTPUT**

```
Instances are:
 [['sunny' 'warm' 'normal' 'strong' 'warm' 'same']
 ['sunny' 'warm' 'high' 'strong' 'warm' 'same']
 ['rainy' 'cold' 'high' 'strong' 'warm' 'change']
 ['sunny' 'warm' 'high' 'strong' 'cool' 'change']]
Target Values are:  ['yes' 'yes' 'no' 'yes']
Initialization of specific_h and genearal_h
Specific Boundary:  ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']
Generic Boundary:  [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'],
['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
```

Instance 1 is ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

Instance is Positive

Specific Bundary after 1 Instance is ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

Generic Boundary after 1 Instance is [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Instance 2 is ['sunny' 'warm' 'high' 'strong' 'warm' 'same']

Instance is Positive

Specific Bundary after 2 Instance is ['sunny' 'warm' '?' 'strong' 'warm' 'same']

Generic Boundary after 2 Instance is [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Instance 3 is ['rainy' 'cold' 'high' 'strong' 'warm' 'change']

Instance is Negative

Specific Bundary after 3 Instance is ['sunny' 'warm' '?' 'strong' 'warm' 'same']

Generic Boundary after 3 Instance is [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]

Instance 4 is ['sunny' 'warm' 'high' 'strong' 'cool' 'change']

Instance is Positive

Specific Bundary after 4 Instance is ['sunny' 'warm' '?' 'strong' '?' '?']

Generic Boundary after 4 Instance is [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific_h:

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

Final General_h:

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

**4.** 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.

**ALGORITHM**

ID3(Examples, Target_attribute, Attributes)

Create a Root node for the tree

If all Examples are positive, Return the single-node tree Root, with label = +

If all Examples are negative, Return the single-node tree Root, with label = -

If Attributes is empty, Return the single-node tree Root,

with label = most common value of Target_attribute in Examples

Otherwise Begin

  A ← the attribute from Attributes that best* classifies Examples

  The decision attribute for Root ← A

  For each possible value, vi, of A,

    Add a new tree branch below Root, corresponding to the test A = vi

    Let Examples vi, be the subset of Examples that have value vi for A

    If Examples vi , is empty

      Then below this new branch add a leaf node with

      label = most common value of Target_attribute in Examples

    Else

      below this new branch add the subtree

      ID3(Examples vi, Targe_tattribute, Attributes – {A}))

End

Return Root

**PROGRAM**

```
import pandas as pd

import math

import numpy as np

data = pd.read_csv("3-dataset.csv")

features = [feat for feat in data]

features.remove("answer")

class Node:

    def __init__(self):

        self.children = []

        self.value = ""

        self.isLeaf = False

        self.pred = ""

def entropy(examples):

    pos = 0.0

    neg = 0.0

    for _, row in examples.iterrows():

        if row["answer"] == "yes":

            pos += 1

        else:

            neg += 1

    if pos == 0.0 or neg == 0.0:

        return 0.0

    else:

        p = pos / (pos + neg)
```

```
        n = neg / (pos + neg)
        return -(p * math.log(p, 2) + n * math.log(n, 2))
def info_gain(examples, attr):
    uniq = np.unique(examples[attr])
    gain = entropy(examples)
    for u in uniq:
        subdata = examples[examples[attr] == u]
        sub_e = entropy(subdata)
        gain -= (float(len(subdata)) / float(len(examples))) * sub_e
    return gain
def ID3(examples, attrs):
    root = Node()
    max_gain = 0
    max_feat = ""
    for feature in attrs:
        gain = info_gain(examples, feature)
        if gain > max_gain:
            max_gain = gain
            max_feat = feature
    root.value = max_feat
    uniq = np.unique(examples[max_feat])
    for u in uniq:
        subdata = examples[examples[max_feat] == u]
        if entropy(subdata) == 0.0:
            newNode = Node()
```

```
                newNode.isLeaf = True
                newNode.value = u
                newNode.pred = np.unique(subdata["answer"])
                root.children.append(newNode)
            else:
                dummyNode = Node()
                dummyNode.value = u
                new_attrs = attrs.copy()
                new_attrs.remove(max_feat)
                child = ID3(subdata, new_attrs)
                dummyNode.children.append(child)
                root.children.append(dummyNode)
    return root
def printTree(root: Node, depth=0):
    for i in range(depth):
        print("\t", end="")
    print(root.value, end="")
    if root.isLeaf:
        print(" -> ", root.pred)
    print()
    for child in root.children:
        printTree(child, depth + 1)
root = ID3(data, features)
printTree(root)
```

**DATASET**

| **Day** | ***Outlook*** | ***Temperature*** | ***Humidity*** | ***Wind*** | ***PlayTennis*** |
|---|---|---|---|---|---|
| **D1** | Sunny | Hot | High | Weak | No |
| **D2** | Sunny | Hot | High | Strong | No |
| **D3** | Overcast | Hot | High | Weak | Yes |
| **D4** | Rain | Mild | High | Weak | Yes |
| **D5** | Rain | Cool | Normal | Weak | Yes |
| **D6** | Rain | Cool | Normal | Strong | No |
| **D7** | Overcast | Cool | Normal | Strong | Yes |
| **D8** | Sunny | Mild | High | Weak | No |
| **D9** | Sunny | Cool | Normal | Weak | Yes |
| **D10** | Rain | Mild | Normal | Weak | Yes |
| **D11** | Sunny | Mild | Normal | Strong | Yes |
| **D12** | Overcast | Mild | High | Strong | Yes |
| **D13** | Overcast | Hot | Normal | Weak | Yes |
| **D14** | Rain | Mild | High | Strong | No |

**OUTPUT**

```
Outlook
        Overcast ->  ['yes']
        Rainy
                Wind
                        Strong ->  ['No']
                        Weak ->  ['yes']
        Sunny
                Humidity
                        High ->  ['No']
                        Normal ->  ['yes']
```

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

**ALGORITHM**

BACKPROPAGATION (*training_example, $\eta$, $n_{in}$, $n_{out}$, $n_{hidden}$*)

*Each training example is a pair of the form $(\vec{x}, \vec{t})$, where $(\vec{x})$ is the vector of network input values, $(\vec{t})$ and is the vector of target network output values.*
*$\eta$ is the learning rate (e.g., .05). $n_i$, is the number of network inputs, $n_{hidden}$ the number of units in the hidden layer, and $n_{out}$ the number of output units.*
*The input from unit i into unit j is denoted $x_{ji}$, and the weight from unit i to unit j is denoted $w_{ji}$*

- Create a feed-forward network with $n_i$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do
  - For each $(\vec{x}, \vec{t})$, in training examples, Do

    *Propagate the input forward through the network:*
    1. Input the instance $\vec{x}$, to the network and compute the output $o_u$ of every unit u in the network.

    *Propagate the errors backward through the network:*

    2. For each network output unit k, calculate its error term $\delta_k$

    $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

    3. For each hidden unit *h*, calculate its error term $\delta_h$

    $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

    4. Update each network weight $w_{ji}$

    $$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

    Where

    $$\Delta w_{ji} = \eta \delta_j x_{i,j}$$

**PROGRAM**

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
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=5 #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))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))

#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)
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    print ("-----------Epoch-", i+1, "Starts----------")
    print("Input: \n" + str(X))
    print("Actual Output: \n" + str(y))
    print("Predicted Output: \n" ,output)
    print ("-----------Epoch-", i+1, "Ends----------\n")
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

**OUTPUT**

```
-----------Epoch- 1 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.71669304]
 [0.70551416]
 [0.72402119]]
-----------Epoch- 1 Ends----------
-----------Epoch- 2 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
```

```
Predicted Output:
 [[0.72021397]
 [0.70884396]
 [0.72759384]]
-----------Epoch- 2 Ends----------


-----------Epoch- 3 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.72361682]
 [0.71206417]
 [0.7310446 ]]
-----------Epoch- 3 Ends----------


-----------Epoch- 4 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
```

```
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.72690698]
 [0.71517975]
 [0.73437912]]
-----------Epoch- 4 Ends----------


-----------Epoch- 5 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.73008956]
 [0.71819539]
 [0.73760274]]
-----------Epoch- 5 Ends----------
```

Input:

```
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
```

Actual Output:

```
[[0.92]
 [0.86]
 [0.89]]
```

Predicted Output:

```
 [[0.73008956]
 [0.71819539]
 [0.73760274]]
```