AIML Lab

18CSL76

Table of contents:

1. A* Algo
2. AO* Algo
3. Candidate elimination
4. ID3 Algo
5. Pro5 : Backpropagation
6. Pro6 : Naive Bayes
7. Pro7 : EM Algo
8. Pro8 : KNN Algo
9. Pro9 : LWR
10. Find-S (Not for Exam)

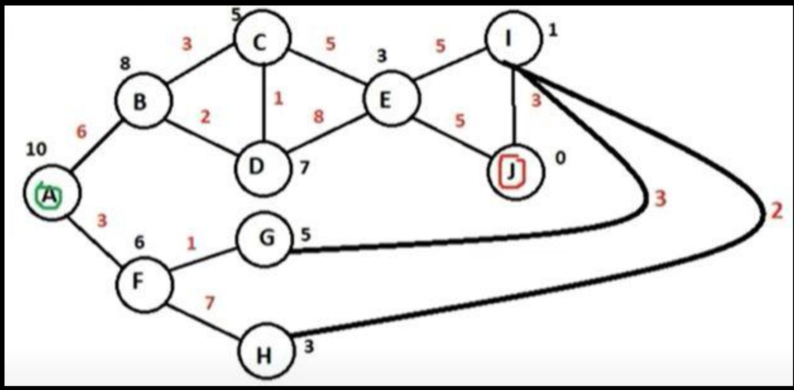
Pro 1 : A* algo

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    #print(set(start_node))
    closed_set = set()

    g = {}               # store distance from starting node
    parents = {}         # parents contains an adjacency map of all nodes

    #distance of starting node from itself is zero
    g[start_node] = 0

    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set, are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #for each node m,compare its distance from start i.e g(m) to the
                #from start through n node
                else:
```

```
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n
                        #if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path

        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)

    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None


```

```python
#for simplicity we’ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {'A':10,
              'B':8,
              'C':5,
              'D':7,
              'E':3,
              'F':6,
              'G':5,
              'H':3,
              'I':1,
              'J':0 }


    return H_dist[n]

Graph_nodes = {
    'A':[('B',6),('F',3)],
    'B':[('C',3),('D',2)],
    'C':[('D',1),('E',5)],
    'D':[('C',1),('E',8)],
    'E':[('I',5),('J',5)],
    'F':[('G',1),('H',7)],
    'G':[('I',3)],
    'H':[('I',2)],
    'I':[('E',5),('J',3)]
}


aStarAlgo('A', 'J')
```

Path found: ['A', 'F', 'G', 'I', 'J']
['A', 'F', 'G', 'I', 'J']



Algorithm:

```
1) Initialize the open and closed lists
2) Add start node to the open list
3) For all the neighbouring nodes, find the least cost F node
4) Switch to the closed list
    For nodes adjacent to the current node
        If the node is not reachable, ignore it.
        Else
        If the node is not on the open list, move it to the open list and calculate f,
g, h.
        If the node is on the open list, check if the path it offers is less than the
current path and change to it if does so.
5) Stop working when
    You find the destination
    You cannot find the destination going through all possible points
```

Same program, but different example 👇

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    #print(set(start_node))
    closed_set = set()
    g = {}               #store distance from starting node
    parents = {}         # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
```

```python
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #for each node m,compare its distance from start i.e g(m) to the
                #from start through n node
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n
                        #if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
```

```
            return path
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {'A': 11,'B': 6,'C':99, 'D':1, 'E':7, 'G':0}
    return H_dist[n]
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('A',2),('C', 1), ('G',9)],
    'C': [('B',1)],
    'D': [('E',6),('G',1)],
    'E': [('A',3),('D',6)],
    'G': [('B',9),('D',1)]}
aStarAlgo('A', 'G')
```

Path found: ['A', 'E', 'D', 'G']
['A', 'E', 'D', 'G']

Pro 2 : AO* star

```
#Algorithm
1. Let GRAPH consist only of the node representing the initial state. Call this node
INIT, Compute h' (INIT).
2. Until INIT is labelled SOLVED or until INIT’s h' value becomes greater than
FUTILITY, repeat the following procedure:
    a. Trace the labelled arcs from INIT and select for expansion one of the as yet
unexpanded nodes that occurs on this path. Call the selected node NODE.
    b. Generate the successors of NODE. If there are none, then assign FUTILITY as the
h' value of NODE. This is equivalent to saying that NODE is not solvable. If there are
successors, then for each one (called SUCCESSOR) do the following:
        i) Add SUCCESSOR to GRAPH.
        ii) If SUCCESSOR is a terminal node, label it SOLVED & assign it an h' value
to 0.
        iii) If SUCCESSOR is not a terminal node, compute its h' value.
    c. Propagate the newly discovered information up the graph by doing the following:
Let S be a set of nodes that have been labelled SOLVED or whose h' values have been
changed and so need to have values propagated back to their parents. Initialize S to
NODE. Until S is empty, repeat the following procedure:
        i) If possible, select from S a node none of whose descendants in GRAPH occurs
in S. If there is no such node, select any node from S. Call this node CURRENT, and
remove it from S.
        ii) Compute the cost of each of the arcs emerging from CURRENT. The cost of
each arc is equal to the sum of the h' values of each of the nodes at the end of the
arc plus whatever the cost of the arc itself is. Assign as CURRENT’S new h' value the
minimum of the costs just computed for the arcs emerging from it.
        iii) Mark the best path out of CURRENT by making the arc that had the minimum
cost as compared in the previous step.
        iv) Mark CURRENT SOLVED if all the nodes connected to it through the new
labelled arc have been labelled SOLVED.
        v) If CURRENT has been labelled SOLVED or if the cost of CURRENT was just
changed, then its new status must be propagated back up the graph. So add all of the
ancestors of CURRENT to S.
```

```python
# Recursive implementation of AO* aglorithm

class Graph:
    #instantiate graph object with graph topology, heuristic values, start node
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self):         # starts a recursive AO* algorithm
        self.aoStar(self.start, False)

    def getNeighbors(self, v):     # gets the Neighbors of a given node
        return self.graph.get(v,'')

    def getStatus(self,v):         # return the status of a given node
        return self.status.get(v,0)

    def setStatus(self,v, val):    # set the status of a given node
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0)     # always return the heuristic value of a given node

    def setHeuristicNodeValue(self, n, value):
        self.H[n]=value            # set the revised heuristic value of a given node


    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")
    # Computes the Minimum Cost of child nodes of a given node v
    def computeMinimumCostChildNodes(self, v):
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
```

```
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v):  # iterate over all the set of child node/s
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)
            # initialize Minimum Cost with the cost of first set of child node/s
            if flag==True:
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s
                flag=False
            else:   # checking the Minimum Cost nodes with the current Minimum Cost
                if minimumCost>cost:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s
            # return Minimum Cost and Minimum Cost child node/s
        return minimumCost, costToChildNodeListDict[minimumCost]

    # AO* algorithm for a start node and backTracking status flag
    def aoStar(self, v, backTracking):
        print("HEURISTIC VALUES  :", self.H)
        print("SOLUTION GRAPH    :", self.solutionGraph)
        print("PROCESSING NODE   :", v)
        print("-----------------------------------------------------------------------------------------")
        if self.getStatus(v) >= 0:        # if status node v >= 0, compute Minimum Cost nodes of v
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v,len(childNodeList))
            solved=True                   # check the Minimum Cost nodes of v are solved
            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode)!=-1:
                    solved=solved & False
            # if the Minimum Cost nodes of v are solved, set the current node status as solved(-1)
```

```python
            if solved==True:
                self.setStatus(v,-1)
                # update the solution graph with the solved nodes which may be a part of solution
                self.solutionGraph[v]=childNodeList
            # check the current node is the start node for backtracking the current node value
            if v!=self.start:
                # backtracking the current node value with backtracking status set to true
                self.aoStar(self.parent[v], True)
            if backTracking==False:     # check the current call is not for backtracking
                for childNode in childNodeList: # for each Minimum Cost child node
                    self.setStatus(childNode,0) # set the status of child node to 0(needs exploration)
                    # Minimum Cost child node is further explored with backtracking status as false
                    self.aoStar(childNode, False)


h1 = {'A': 9, 'B': 3, 'C': 4, 'D': 5, 'E': 5, 'F': 7, 'G': 4, 'H': 4}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('E', 1)], [('F', 1)]],
    'D': [[('G', 1), ('H', 1)]]

}
G1= Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()
```

```
HEURISTIC VALUES : {'A': 9, 'B': 3, 'C': 4, 'D': 5, 'E': 5, 'F': 7, 'G': 4, 'H': 4}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 6, 'B': 3, 'C': 4, 'D': 5, 'E': 5, 'F': 7, 'G': 4, 'H': 4}
SOLUTION GRAPH : {}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 6, 'B': 3, 'C': 4, 'D': 10, 'E': 5, 'F': 7, 'G': 4, 'H': 4}
SOLUTION GRAPH : {}
```

```
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 9, 'B': 3, 'C': 4, 'D': 10, 'E': 5, 'F': 7, 'G': 4, 'H': 4}
SOLUTION GRAPH : {}
PROCESSING NODE : G
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 9, 'B': 3, 'C': 4, 'D': 10, 'E': 5, 'F': 7, 'G': 0, 'H': 4}
SOLUTION GRAPH : {'G': []}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 9, 'B': 3, 'C': 4, 'D': 6, 'E': 5, 'F': 7, 'G': 0, 'H': 4}
SOLUTION GRAPH : {'G': []}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 7, 'B': 3, 'C': 4, 'D': 6, 'E': 5, 'F': 7, 'G': 0, 'H': 4}
...
FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A
------------------------------------------------------------ {'G': [], 'H': [], 'D': ['G',
'H'], 'A': ['D']}
```

Pro 3 : Candidate elimination

```
# 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 hypothesis consistant with the training
examples
```

```python
import numpy as np
import pandas as pd
data = pd.read_csv('enjoysport.csv')
```

enjoysport.csv 👇

	Sky	Airtemp	Humidity	Wind	Water	Forecast	Enjoysport
0	Sunny	Warm	Normal	Strong	Warm	Same	Yes
1	Sunny	Warm	High	Strong	Warm	Same	Yes
2	Rainy	Cold	High	Strong	Warm	Change	No
3	Sunny	Warm	High	Strong	Cool	Change	Yes

```python
concepts = np.array(data)[:,:-1]
concepts
```

```
array([['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'],
['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same'],
['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change'],
['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change']],
 dtype=object)
```

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

array(['Yes', 'Yes', 'No', 'Yes'], dtype=object)
```

```python
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):
        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("Iteration["+str(i+1)+ "]")
        print("Specific: "+str(specific_h))
        print("General: "+str(general_h)+"\n\n")
    general_h=[general_h[i] for i,h in enumerate(general_h) if h!=["?" for x in range(len(specific_h))]]
    return specific_h,general_h
```

```
specific,general=learn(concepts,target)
```

```
initialization of specific_h and general_h
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same'] [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

If instance is Positive
Iteration[1]
Specific: ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same'] General: [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

If instance is Positive
Iteration[2]
Specific: ['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same'] General: [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

If instance is Negative
```

```
Iteration[3]
Specific: ['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same'] General: [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

If instance is Positive
Iteration[4]
Specific: ['Sunny' 'Warm' '?' 'Strong' '?' '?'] General: [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
```

```
print("Final hypothesis: ")
print("Specific: "+str(specific))
print("General: "+str(general))
```

```
Final hypothesis: Specific: ['Sunny' 'Warm' '?' 'Strong' '?' '?'] General: [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]
```

Algorithm:

G ← maximally general hypotheses in H
S ← maximally specific hypotheses in H
For each training example d=<x,c(x)>
Case 1 : If d is a positive example

Remove from G any hypothesis that is inconsistent with d
For each hypothesis s in S that is not consistent with d

- *Remove s from S.*
- *Add to S all minimal generalizations h of s such that*
  - *h consistent with d*
  - *Some member of G is more general than h*
- *Remove from S any hypothesis that is more general than another hypothesis in S*

Case 2: If d is a negative example

Remove from S any hypothesis that is inconsistent with d
For each hypothesis g in G that is not consistent with d

- *Remove g from G.*
- *Add to G all minimal specializations h of g such that*
  - *h consistent with d*
  - *Some member of S is more specific than h*
- *Remove from G any hypothesis that is less general than another hypothesis in G*

Pro 4 : **ID3 Agorithm**

Decision tree based on ID3

data3_test.csv

Outlook	Temperature	Humidity	Wind
rain	cool	normal	strong
sunny	mild	normal	strong

data3.csv

Outlook	Temperature	Humidity	Wind	Answer
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

```python
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 = "" # NULL indicates children exists.
                         # Not Null indicates this is a Leaf Node

def subtables(data, col, delete):
    dic = {}
    coldata = [ row[col] for row in data]
    attr = list(set(coldata)) # All values of attribute retrieved
    for k in attr:
        dic[k] = []
    for y in range(len(data)):
        key = data[y][col]
        if delete:
            del data[y][col]
        dic[key].append(data[y])
    return attr, dic

def entropy(S):
    attr = list(set(S))
    if len(attr) == 1: #if all are +ve/-ve then entropy = 0
        return 0
    counts = [0,0] # Only two values possible 'yes' or 'no'
    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):
    attValues, dic = subtables(data, col, delete=False)
    total_entropy = entropy([row[-1] for row in data])

    for x in range(len(attValues)):
        ratio = len(dic[attValues[x]]) / ( len(data) * 1.0)
        entro = entropy([row[-1] for row in dic[attValues[x]]])
        total_entropy -= ratio*entro

    return total_entropy

def build_tree(data, features):
    lastcol = [row[-1] for row in data]
```

```python
    if (len(set(lastcol))) == 1: # If all samples have same labels return that label
        node=Node("")
        node.answer = lastcol[0]
        return node

    n = len(data[0])-1

    gains = [compute_gain(data, col) for col in range(n) ]
    split = gains.index(max(gains)) # Find max gains and returns index
    node = Node(features[split]) # 'node' stores attribute selected
    #del (features[split])
    fea = features[:split]+features[split+1:]
    attr, dic = subtables(data, split, delete=True) # Data will be spilt in subtables

    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) # Displays leaf node yes/no
        return

    print(" "*level, node.attribute) # Displays attribute Name

    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)
```

```python
''' Main program '''
dataset, features = load_csv("data3.csv") # Read Tennis data
node = build_tree(dataset, features) # Build decision tree
print("The decision tree for the dataset using ID3 algorithm is ")
print_tree(node, 0)
testdata, features = load_csv("data3_test.csv")
for xtest in testdata:
    print("The test instance : ",xtest)
    print("The predicted label : ", end="")
    classify(node,xtest,features)
```

```
The decision tree for the dataset using ID3 algorithm is
Outlook rain Wind weak yes strong no sunny Humidity normal yes high no
overcast yes
The test instance : ['rain', 'cool', 'normal', 'strong']
The predicted label : no
The test instance : ['sunny', 'mild', 'normal', 'strong']
The predicted label : yes
```

ID3 - Algorithm

ID3(*Examples, TargetAttribute, 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 *TargetAttribute* 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 *TargetAttribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{vi}$, *TargetAttribute*, *Attributes* – {A})
- End
- Return *Root*

Pro 5 : Backpropagation

```
# Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.
```

```
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)
y = y/100

def sigmoid (x):
    return 1/(1 + np.exp(-x))
def derivatives_sigmoid(x):
    return x * (1 - x)

epoch=1000
lr=0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1
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))

for i in range(epoch):
    h_ip=np.dot(X,wh)+bh
    h_act = sigmoid(h_ip)
    o_ip=np.dot(h_act,wout)
    output = sigmoid(o_ip)
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(h_act)
    d_hidden = EH * hiddengrad
    wout += h_act.T.dot(d_output) *lr
    wh += X.T.dot(d_hidden) *lr
```

```python
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: [[ 0.89626316] [ 0.87318862] [ 0.89928505]]
```

function BackProp $(D, \eta, n_{\text{in}}, n_{\text{hidden}}, n_{\text{out}})$

- D is the training set consists of m pairs: $\{(x_i, y_i)^m\}$
- η is the learning rate as an example (0.1)
- n_{in}, n_{hidden} e n_{out} are the numbero of imput hidden and output unit of neural network

Make a feed-forward network with n_{in}, n_{hidden} e n_{out} units

Initialize all the weight to short randomly number (es. [-0.05 0.05])

Repeat until termination condition are verifyed:

For any sample in D:

Forward propagate the network computing the output o_u of every unit u of the network

Back propagate the errors onto the network:

– For every output unit k, compute the error δ_k: $\delta_k = o_k(1-o_k)(t_k - o_k)$

– For every hidden unit h compute the error δ_h: $\delta_h = o_h(1-o_h) \sum_{k \in outputs} w_{kh}\delta_k$

– Update the network weight w_{ji}: $w_{ji} = w_{ji} + \Delta w_{ji}, \quad where \;\; \Delta w_{ji} = \eta \delta_j x_{ji}$

(x_{ji} is the input of unit j from coming from unit i)

Pro 6 : Naives Baiyes classifier

```
# Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.
```

```
import pandas as pd
data=pd.read_csv('diabetes.csv')

from sklearn.model_selection import train_test_split
x=data.drop('Outcome',axis=1)
y=data[['Outcome']]

x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.30,random_state=1)


from sklearn.naive_bayes import GaussianNB
model=GaussianNB()
model.fit(x_train,y_train)
y_pred=model.predict(x_test)


    -> /home/mllab/anaconda3/lib/python3.8/site-packages/sklearn/utils/validation.py:72: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
      return f(**kwargs)

from sklearn import metrics
print('accuracy:',metrics.accuracy_score(y_test,y_pred))

    -> accuracy: 0.7835497835497836
```

Bayes Theorem

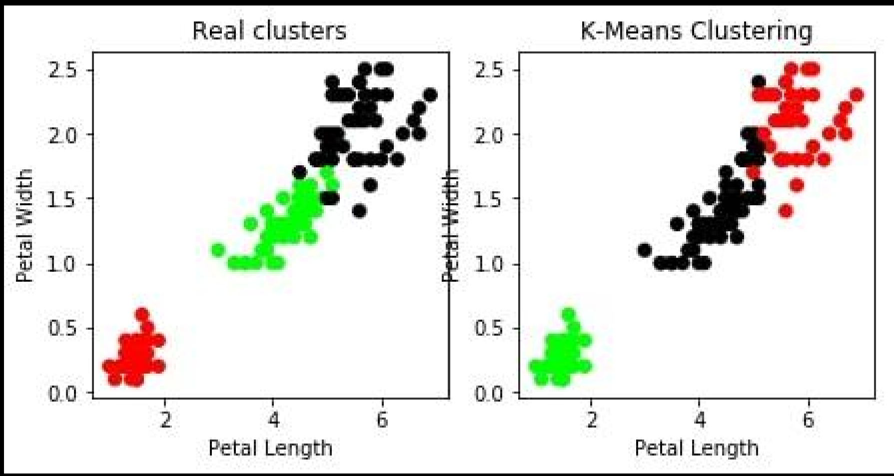
$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- $P(h)$ = prior probability of hypothesis h
- $P(D)$ = prior probability of training data D
- $P(h|D)$ = probability of h given D
- $P(D|h)$ = probability of D given h

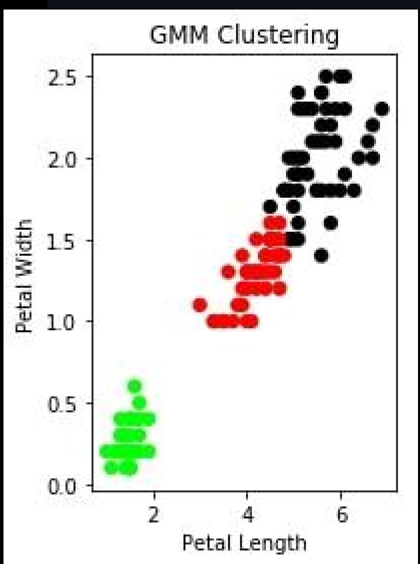
Pro 7 : EM Algo

```
# Apply EM algorithm to cluster a set of data stored in a .CSV file. Use
the same data set for clustering using k-Means algorithm. Compare the
results of these two algorithms and comment on the quality of clustering.
You can add Java/Python ML library classes/API in the program.
```

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import pandas as pd
import numpy as np

iris = datasets.load_iris()
X = pd.DataFrame(iris.data)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

model = KMeans(n_clusters=3)
model.fit(X)
plt.figure(figsize=(7,7))
colormap = np.array(['red', 'lime', 'black'])
plt.subplot(2, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.subplot(2, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()
```




```python
from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y= gmm.predict(xs)
plt.subplot(1, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[gmm_y], s=40)
plt.title('GMM Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()
print('Observation: The GMM using EM algorithm based clustering matched the true labels more closely than KMeans')
```



Observation: The GMM using EM algorithm based clustering matched the true labels more closely than KMeans

Pro 8 : KNN Algo

```
# Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem
```

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

```python
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets

iris = datasets.load_iris()

print("size of training data and its label",x_train.shape, y_train.shape)
print("size of testing data and its label",x_test.shape, y_test.shape)

```

a. -> size of training data and its label (135, 4) (135,)
b. size of testing data and its label (15, 4) (15,)

```python

for i in range(len(iris.target_names)):
    print("label",i, "-",str(iris.target_names[i]))

```

a. -> label 0 - setosa
b. label 1 - versicolor
c. label 2 - virginica

```python
classifier=KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train,y_train)
y_pred=classifier.predict(x_test)


print("Results of classification using K-nn with k=1")
for r in range(0,len(x_test)):
    print("Sample:",str(x_test[r]), "Actual-label:",str(y_test[r]),
"Predicted-label:",str(y_pred[r]))
print("classification accuracy:", classifier.score(x_test,y_test))

```

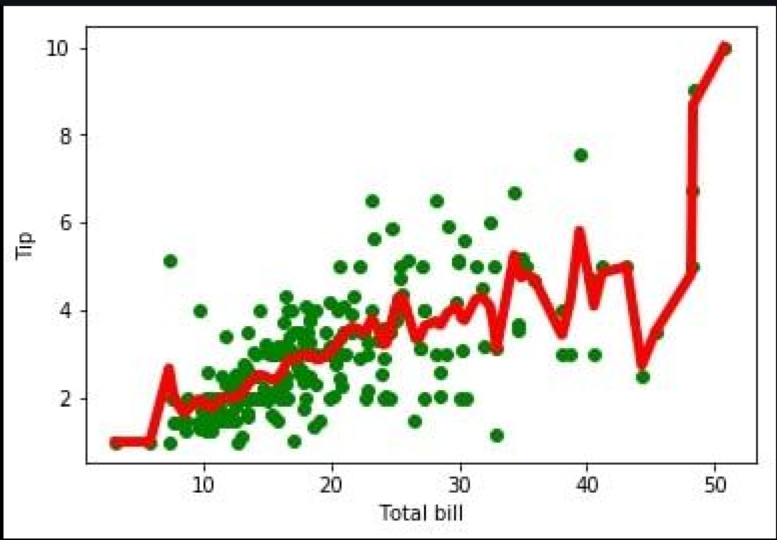
```
-> Results of classification using K-nn with k=1
Sample: [ 5.   2.3  3.3  1. ] Actual-label: 1 Predicted-label: 1
Sample: [ 5.2  3.4  1.4  0.2] Actual-label: 0 Predicted-label: 0
Sample: [ 5.7  4.4  1.5  0.4] Actual-label: 0 Predicted-label: 0
Sample: [ 6.3  2.5  5.   1.9] Actual-label: 2 Predicted-label: 2
Sample: [ 6.1  3.   4.9  1.8] Actual-label: 2 Predicted-label: 2
Sample: [ 6.3  3.3  6.   2.5] Actual-label: 2 Predicted-label: 2
Sample: [ 5.1  3.4  1.5  0.2] Actual-label: 0 Predicted-label: 0
Sample: [ 5.9  3.   5.1  1.8] Actual-label: 2 Predicted-label: 2
Sample: [ 6.4  3.1  5.5  1.8] Actual-label: 2 Predicted-label: 2
Sample: [ 6.2  3.4  5.4  2.3] Actual-label: 2 Predicted-label: 2
Sample: [ 5.7  3.   4.2  1.2] Actual-label: 1 Predicted-label: 1
Sample: [ 5.1  3.8  1.9  0.4] Actual-label: 0 Predicted-label: 0
Sample: [ 4.9  3.1  1.5  0.1] Actual-label: 0 Predicted-label: 0
Sample: [ 5.8  4.   1.2  0.2] Actual-label: 0 Predicted-label: 0
Sample: [ 6.1  2.9  4.7  1.4] Actual-label: 1 Predicted-label: 1
classification accuracy: 1.0
```

Pro 9 : LWR

Implement non-parametric locally weighted regression algorithm in order to fit data points.

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

def kernel(point, xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights


def localWeight(point, xmat, ymat, k):
    wei = kernel(point,xmat,k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W


def localWeightRegression(xmat, ymat, k):
    m,n = np.shape(xmat)
    y_pred = np.zeros(m)
    for i in range(m):
        y_pred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
    return y_pred

def graphPlot(X,y_pred):
    sortindex = X[:,1].argsort(0)
    xsort = X[sortindex][:,0]
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.scatter(bill,tip, color='green')
    ax.plot(xsort[:,1],y_pred[sortindex], color = 'red', linewidth=5)
    plt.xlabel('Total bill')
    plt.ylabel('Tip')
    plt.show()

```

```

# load data points
data = pd.read_csv('bill.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)
mbill = np.mat(bill)
mtip = np.mat(tip)
m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T))
y_pred = localWeightRegression(X,mtip,0.5)
graphPlot(X,y_pred)
```



Pro 10: Find-S Algo

```
# Implement and demonstratethe 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.


import random
import csv
```