Find-S

import csv

A=[]

with open('enjoysport.csv',newline="") as csvfile:

for row in csv.reader(csvfile):

A.append(row)

S = ['Null' for k in range(len(A[0])-1)]

c=0

for row in A[1:]:

print("h",c," : ",S,sep='')

c+=1

if row[-1]=='No':

continue

for k in range(len(row)-1):

if S[k]=='Null':

S[k]=row[k]

continue

if S[k]=='?':

continue

if S[k]!=row[k]:

S[k]='?'

print("h",c," : ",S,sep='')

print("Most specific hypothesis : ",S)

Text, letter

Description automatically generated

K nearest algorithm to classify the dataset

import numpy

import pandas

import csv

Train = []

Test = []

clstr = []

clstt = []

count = 0

with open('iris.csv',newline="") as csvfile :

for row in csv.reader(csvfile):

if count==0:

count=1

continue

if count%3==0 :

Train.append(row[:-1])

clstr.append(row[-1])

else:

Test.append(row[:-1])

clstr.append(row[-1])

count+=1

def dist(a,b):

import math

n = len(a)

d=0

for j in range(n):

d+=(float(b[j])-float(a[j]))\*\*2

return math.sqrt(d)

def myfunc(a):

return a[0]

testdata = [7.2,3.6,5.1,2.5]

dis = []

for r in range(len(Train)):

h = Train[r]

dis.append([dist(h,testdata),clstr[r]])

dis = sorted(dis,key=myfunc)[:5]

freq = {}

for row in dis:

freq[row[1]]=freq.get(row[1],0)+1

for key in freq.keys():

predicted = key

break

for key in freq.keys():

if freq[predicted]<freq[key]:

predicted = key

print("Prediction : ",predicted)

Breadth First Search :

open=[]

closed=[]

path = []

parent = {}

graph = {'A':['B','C'],

'B':['D','E'],

'C':['F','G'],

'D':[],

'E':[],

'F':[],

'G':[]}

def findpath(tar):

global open,path

while tar!='':

path.append(tar)

tar = parent[tar]

def BFS(tar, ptr):

global open, closed, graph

print('open : ',open[ptr:],'\nclosed : ',closed, "\n")

if ptr >= len(open):

return -1

closed.append(open[ptr])

if open[ptr]==tar:

findpath(tar)

return 1

for child in graph[open[ptr]]:

open.append(child)

parent[child]=open[ptr]

BFS(tar,ptr+1)

open.append('A')

parent['A']=''

tar = input('Enter goal node : ')

BFS(tar,0)

if path==[]:

print("Goal node not found")

else:

path.reverse()

BFSpath = '->'.join(path)

print('Path : ',BFSpath)

Graphical user interface, application, Word

Description automatically generated

DFS :

visited = []

path = []

parent = {}

graph = {'A':['B','C'],

'B':['D','E'],

'C':['F','G'],

'D':[],

'E':[],

'F':[],

'G':[]}

def findpath(goal):

global parent, path

while goal!='':

path.append(goal)

goal = parent[goal]

def DFS(current, goal, lim=float('inf'), depth=0):

print("DFS called on node : ",current)

global graph, parent, visited

visited.append(current)

if current == goal :

findpath(goal)

return 1

if depth==lim:

return

for child in graph[current]:

parent[child] = current

if DFS(child, goal, lim, depth+1)==1:

return 1

parent['A']=''

goal = input("Enter goal node : ")

DFS('A',goal)

print()

if path==[]:

print("Goal node not found")

else:

path.reverse()

DFSpath = '->'.join(path)

print('Path : ',DFSpath)

A screenshot of a computer

Description automatically generated

IDDFS :

visited = []

path = []

parent = {}

graph = {'A':['B','C'],

'B':['D','E'],

'C':['F','G'],

'D':[],

'E':[],

'F':[],

'G':[]}

def findpath(goal):

global parent, path

while goal!='':

path.append(goal)

goal = parent[goal]

def IDDFS(current, goal, lim=float('inf'), depth=0):

print("DFS called on node : ",current)

global graph, parent, visited

visited.append(current)

if current == goal :

findpath(goal)

return 1

if depth==lim:

return

for child in graph[current]:

parent[child] = current

if IDDFS(child, goal, lim, depth+1)==1:

return 1

parent['A']=''

goal = input("Enter goal node : ")

maxdep = int(input("Enter max depth : "))

l=0

while path==[] and l<maxdep:

print("With limit : ",l)

IDDFS('A',goal,lim = l,depth = 0)

l+=1

print()

print()

if path==[]:

print("Goal node not found")

else:

path.reverse()

IDDFSpath = '->'.join(path)

print('Path : ',IDDFSpath)

print("Found when depth was allowed upto ",l)

Graphical user interface, application

Description automatically generated

8-queens :

n = int(input().strip())

board = [[0 for j in range(n+1)] for k in range(n)]

def attack(n,row,col,board):

for j in range(1,n+1):

if j==col:

continue

if board[row][j]==1:

return True

r,c = row-1, col+1

while r>=0 and c<n+1:

if board[r][c]==1:

return True

r-=1

c+=1

r,c = row+1, col+1

while r<n and c<n+1:

if board[r][c]==1:

return True

r+=1

c+=1

return False

def n\_queens(board,n,col):

if col==0:

return True

for k in range(n):

board[k][col]=1

if attack(n,k,col,board):

board[k][col]=0

continue

if n\_queens(board,n,col-1):

return True

else:

board[k][col]=0

return False

n\_queens(board,n,n)

for row in board:

print(row[1:])

PUZZLE :

class Puzzle:

puzzlebox = [[]]

size = ''

steps = []

goal = [[]]

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

self.size = size

self.puzzlebox = [[0 for j in range(self.size)] for k in range(self.size)]

self.boxinput()

self.goal = [[j\*self.size+k for k in range(1,self.size+1)] for j in range(self.size)]

self.goal[-1][-1]=0

def boxinput(self):

print("Enter the value at : ")

print("(Enter 0 at blank) ")

for j in range(1,self.size+1):

for k in range(1,self.size+1):

self.puzzlebox[j-1][k-1] = int(input("Row "+str(j)+" Col "+str(k)+ " : "))

if self.puzzlebox[j-1][k-1]==0:

self.blankpos = [j-1,k-1]

def evaluate(self):

gn = 0

laststate = ''

for row in self.puzzlebox:

print(row)

print()

while self.gethn(self.puzzlebox)!=0 and gn<10:

x,y = self.blankpos[0],self.blankpos[1]

moves = []

if x+1<self.size: moves.append([x+1,y,'R'])

if x-1>-1: moves.append([x-1,y,'L'])

if y-1>-1: moves.append([x,y-1,'U'])

if y+1<self.size: moves.append([x,y+1,'D'])

rem = []

for move in moves:

if [move[0],move[1]]==laststate:

rem = move

if rem!=[]:

moves.remove(rem)

fns = {}

for mo in moves:

fns[mo[-1]]=gn+self.gethn(self.move(mo))

minfn = fns[moves[0][-1]]

nextstate = ''

for key in fns.keys():

if fns[key]<=minfn:

minfn = fns[key]

nextstate = key

self.steps.append(nextstate)

laststate = self.blankpos

if nextstate=='R': self.blankpos = [x+1,y]

elif nextstate=='L': self.blankpos = [x-1,y]

elif nextstate=='U': self.blankpos = [x,y-1]

elif nextstate=='D': self.blankpos = [x,y+1]

self.puzzlebox = self.move(laststate)

gn+=1

for row in self.puzzlebox:

print(row)

print("F(n) : ",minfn,"\n")

print("Sequence of moves for blank are : ")

dirs = {'R':'Down', 'L':'Up', 'U':'Left', 'D':'Right'}

for step in self.steps:

print(dirs[step])

def gethn(self, boxstate):

hn = 0

for row in range(self.size):

for col in range(self.size):

if boxstate[row][col]!=self.goal[row][col]:

hn+=1

return hn

def move(self, mo):

newbox = [[self.puzzlebox[j][k] for k in range(self.size)] for j in range(self.size)]

x = self.blankpos[0]

y = self.blankpos[1]

newbox[x][y], newbox[mo[0]][mo[1]] = newbox[mo[0]][mo[1]], newbox[x][y]

return newbox

if \_\_name\_\_=="\_\_main\_\_":

p = Puzzle(3)

p.evaluate()

Perceptron (and ):

class p:

def input1(self):

x=[[0,0],[0,1],[1,0],[1,1]]

return x

def activation(self,y):

w1=0.5

w2=0.5

z=[]

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

q=w1\*y[i][0] + w2\*y[i][1]

if(q>=1):

z.append(1)

else:

z.append(0)

return z

a=p()

q=a.input1()

b=a.activation(q)

print(b)

output:

[0, 0, 0, 1]

Back propagation :

**Water jug problem :**

from collections import defaultdict

jug1=int(input("enter jug 1 capacity"))

jug2=int(input("enter jug 2 capacity"))

aim=int(input("enter the target capacity"))

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

print(amt1, amt2)

return True

if visited[(amt1, amt2)] == False:

print(amt1, amt2)

visited[(amt1, amt2)] = True

return (waterJugSolver(0, amt2) or

waterJugSolver(amt1, 0) or

waterJugSolver(jug1, amt2) or

waterJugSolver(amt1, jug2) or

waterJugSolver(amt1 + min(amt2, (jug1-amt1)),amt2 - min(amt2, (jug1-amt1))) or

waterJugSolver(amt1 - min(amt1, (jug2-amt2)),amt2 + min(amt1, (jug2-amt2))))

else:

return False

print("Steps: ")

waterJugSolver(0, 0)

