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)
```

DFS

```
graph = {
'A' : ['B','C'],  'B' : ['D', 'E'],  'C' : ['F'],  'D' : [],  'E' : ['F'],  'F' : []
}
visited = []
def dfs(visited, graph, node):
        if node not in visited:
         print node,
        visited.append(node)
        for neighbour in graph[node]:
                dfs(visited, graph, neighbour)
 dfs(visited, graph, 'A')
```

BFS

```
graph = {  'A' : ['B','C'], 'B' : ['D', 'E'], 'C' : ['F'], 'D' : [], 'E' : ['F'], 'F' : [] }
visited = []
 queue = []
 def bfs(visited, graph, node):
visited. append(node)
queue.  append(node)
while queue:
s = queue.  pop(0)
print (s, end = " ")
for neighbour in graph[s]:
            if neighbour not in visited:
                      visited. append(neighbour)
                      queue. append(neighbour)
bfs(visited, graph, 'A')
```

Water jug

```
j1=int(input('capacity of small jug:'))
j2=int(input('capacity of big jug:'))
x=0
y=0
print('enter the final capacities')
d=int(input())
def transfer(x,y,d,j1,j2):
      print(x,'\t',y)
      if y==d:
      return
      elif y==j2:
      transfer(0,x,d,j1,j2)
      elif x!=0 and y==0:
      transfer(0,x,d,j1,j2)
      elif x==d:
      transfer(x,0,d,j1,j2)
      elif x<j1:
      transfer(j1,y,d,j1,j2)
      elif x<(j2-y):
      transfer(0,(x+y),d,j1,j2)
      else:
      transfer(x-(j2-y),(j2-y)+y,d,j1,j2)
print('jar1 \t jar2')
transfer(0,0,d,j1,j2)
```

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 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 : ")
maxdep = int(input("Enter max depth : "))
l=0
while path==[] and l<maxdep:
  print("With limit : ",l)
  DFS('A',goal,lim = l,depth = 0)
  l+=1
  print()
print()
if path==[]:
  print("Goal node not found")
else:
  path.reverse()
  BFSpath = '->'.join(path)
  print('Path : ',BFSpath)
  print("Found when depth was allowed upto ",l)
```

A star

```
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()
```

N-Queen

```
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:])
```

ALPHA-BETA

```
def alphabetapruning(alpha, beta, p, tree, node):
    if isinstance(node,int):
        print('Visited node ',node)
        return node
    ans = ''
    player = ''
    if p==-1:
        for c in range(len(tree[node])):
            child = tree[node][c]
            beta = min(beta,alphabetapruning(alpha,beta,p*-
1,tree,child))
            if beta<=alpha and tree[node][c+1:]!=[]:
                print('Pruning : ',tree[node][c+1:])
                break
        ans = beta
        player = 'Min'
    elif p==1:
        for c in range(len(tree[node])):
            child = tree[node][c]
            alpha = max(alpha,alphabetapruning(alpha,beta,p*-
1,tree,child))
            if alpha >=beta and tree[node][c+1:]!=[]:
                print('Pruning : ',tree[node][c+1:])
                break
        ans = alpha
        player = 'Max'
    print('Visited node '+node+' as '+player+' and returning ',ans," ",
 (alpha, beta))
    return ans
tree = {
    'A' : ['B','C'],
    'B' : ['D','E'],
    'C' : ['F','G'],
    'D' : [3,5],
    'E' : [6,9],
    'F' : [1,2],
    'G' : [0,-1]
}
alphabetapruning(-float('inf'), float('inf'), 1, tree, 'A')
```

tic-tac-toe MAXMIN

```
def prntbox(s):
    print(s[:3])
    print(s[3:6])
    print(s[6:9])
def evalu(s):
    X = 0
    O = 0
    if 'O' not in s[:3]: X+=1
    if 'O' not in s[3:6]: X+=1
    if 'O' not in s[6:9]: X+=1
    if 'O' not in [s[0],s[3],s[6]] : X+=1
    if 'O' not in [s[1],s[4],s[7]] : X+=1
    if 'O' not in [s[2],s[5],s[8]] : X+=1
    if 'O' not in [s[0],s[4],s[8]] : X+=1
    if 'O' not in [s[2],s[4],s[6]] : X+=1
    if 'X' not in s[:3]: O+=1
    if 'X' not in s[3:6]: O+=1
    if 'X' not in s[6:9]: O+=1
    if 'X' not in [s[0],s[3],s[6]] : O+=1
    if 'X' not in [s[1],s[4],s[7]] : O+=1
    if 'X' not in [s[2],s[5],s[8]] : O+=1
    if 'X' not in [s[0],s[4],s[8]] : O+=1
    if 'X' not in [s[2],s[4],s[6]] : O+=1
    return X-O
def checkwin(s):
    rows = [[0,1,2],[3,4,5],[6,7,8]]
    cols = [[0,3,6],[1,4,7],[2,5,8]]
    diag = [[0,4,8],[2,4,6]]
    for r in rows:
        if s[r[0]]==s[r[1]] and s[r[1]]==s[r[2]]:
            if s[r[0]]=='X':
                print('Max player won')
                return 1
            elif s[r[0]]=='O':
                print("Min player won")
                return 1
    for r in cols:
        if s[r[0]]==s[r[1]] and s[r[1]]==s[r[2]]:
            if s[r[0]]=='X':
                print('Max player won')
                return 1
            elif s[r[0]]=='O':
                print("Min player won")
                return 1
      for r in diag:
        if s[r[0]]==s[r[1]] and s[r[1]]==s[r[2]]:
            if s[r[0]]=='X':
                print('Max player won')
                return 1
            elif s[r[0]]=='O':
                print("Min player won")
                return 1
    return -1
p = 1
s = ('_','_','X','O','_','O','_','X','_')
print("Initial state :")
prntbox(s)
print()
while '_' in s:
    d = {}
    if p==1:
        for pl in range(9):
            if s[pl] == '_':
                newb = list(s).copy()
                newb[pl] = 'X'
                d[tuple(newb)] = evalu(newb)
        nxt = ()
        score = -float('inf')
        for k in d.keys():
            if d[k]>score:
                nxt = k
                score = d[k]
        print("Max made move :")
        prntbox(nxt)
        print(d[nxt])
        print()
        s = nxt
      if p==-1:
        for pl in range(9):
            if s[pl] == '_':
                newb = list(s).copy()
                newb[pl] = 'O'
                d[tuple(newb)] = evalu(newb)
        score = float('inf')
        for k in d.keys():
            if d[k]<score:
                nxt = k
                score = d[k]
        print("Min made move :")
        prntbox(nxt)
        print(d[nxt])
        print()
        s = nxt
    if checkwin(s)==1:
       prntbox(s)
        break
    p*=-1
```

Missionaries and cannibals

```
b1 = {'M':3, 'C':3}
b2 = {'M':0, 'C':0}
def move(b1, b2, loc, visited):
  locs = ['src','des']
  if (b1['C']>b1['M'] and b1['M']!=0) or (b2['C']>b2['M'] and b2['M']!=
0):
    return 0
  if (list(b1.values()).copy(), list(b2.values()).copy()) in visited:
    return 0
  if b1['M']==0 and b1['C']==0:
    return 1
  visited.append((list(b1.values()).copy(), list(b2.values()).copy()))
  steps = {
      (0,2),
      (2,0),
      (1,1),
      (0,1),
      (1,0)
  }
if loc==1:
    for step in steps:
      if step[0]>b1['M'] or step[1]>b1['C']:
        continue
      b1['M'] -= step[0]
      b1['C'] -= step[1]
      b2['M'] += step[0]
      b2['C'] += step[1]
      if move(b1, b2, 2, visited)==1:
        b1['M'] += step[0]
        b1['C'] += step[1]
        b2['M'] -= step[0]
        b2['C'] -= step[1]
        print(step,'from src to des','  ', b2,' ', b1)
        return 1
      if (list(b1.values()).copy(), list(b2.values()).copy()) in visite
d:
        visited.remove((list(b1.values()).copy(), list(b2.values()).cop
y()))
      b1['M'] += step[0]
      b1['C'] += step[1]
      b2['M'] -= step[0]
      b2['C'] -= step[1]
elif loc==2:
    for step in steps:
      if step[0]>b2['M'] or step[1]>b2['C']:
        continue
      b2['M'] -= step[0]
      b2['C'] -= step[1]
      b1['M'] += step[0]
      b1['C'] += step[1]
      if move(b1, b2, 1, visited)==1:
        b2['M'] += step[0]
        b2['C'] += step[1]
        b1['M'] -= step[0]
        b1['C'] -= step[1]
        print(step,'from des to src','  ', b2,' ', b1)
        return 1
      if (list(b1.values()).copy(), list(b2.values()).copy()) in visite
d:
        visited.remove((list(b1.values()).copy(), list(b2.values()).cop
y()))
      b2['M'] += step[0]
      b2['C'] += step[1]
      b1['M'] -= step[0]
      b1['C'] -= step[1]

print('Step :bank 1 bank 2')
move(b1,b2,1,[])
```

Locally weighted Regressio

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
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):
  wt = kernel(point, xmat, k)
  W = (X.T * (wt*X)).I * (X.T * wt * ymat.T)
  return W
def localWeightedRegression(xmat, ymat, k):
  m,n = np.shape(xmat)
  ypred = np.zeros(m)
  for i in range(m):
    ypred[i] = xmat[i] * localWeight(xmat[i], xmat, ymat, k)
  return ypred
df = pd.read_csv('tips.csv')
colA = np.array(df.total_bill)
colB = np.array(df.tip)
mcolA = np.mat(colA)
mcolB = np.mat(colB)
m = np.shape(mcolB)[1]
one = np.ones((1, m), dtype = int)
X = np.hstack((one.T, mcolA.T))
print(X.shape)
ypred = localWeightedRegression(X, mcolB, 0.8)
xsort = X.copy()
xsort.sort(axis = 0)
plt.scatter(colA, colB, color = 'blue')
plt.plot(xsort[:,1], ypred[X[:,1].argsort(0)], color = 'yellow', linewi
dth=5)
plt.xlabel('Total Bill')
plt.ylabel('Tip')
plt.show()
```

K MEANS

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
df = pd.read_csv('iris.csv')
kmeans = KMeans(n_clusters=3)
coltypes = df.dtypes
cols = len(coltypes)
numcols = []
for d in range(cols):
  if 'int' in str(coltypes[d]) or 'float' in str(coltypes[d]):
    numcols.append(df.columns[d])
for numcol in numcols :
  X = []
  for k in range(df.shape[0]):
    X.append([k,df.iloc[k][numcol]])
  X = np.array(X)
  print("True position w.r.t. attribute :"+str(numcol))
  plt.scatter(X[:,0],X[:,1], label='True Position')
  plt.show()
  print()
print("Clustered points w.r.t. attribute :"+str(numcol))
  kmeans.fit(X)
  plt.scatter(X[:,0],X[:,1], c=kmeans.labels_, cmap='rainbow')
  plt.scatter(kmeans.cluster_centers_[:,0] ,kmeans.cluster_centers_[:,1
], color='black')
  plt.show()
  print("\n\n\n\n\n")
```

EM ALGORITHM

```
import matplotlib.pyplot as plt
from sklearn import datasets
import pandas as pd i
mport 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']
colormap = np.array(['red', 'lime', 'black'])
plt.figure(figsize=(7,10))
plt.subplot(2, 1, 1)
plt.scatter(X.Sepal_Length, X.Sepal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Clusters')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns) f
rom sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y = gmm.predict(xs)
 plt.subplot(2, 1, 2)
plt.scatter(X.Sepal_Length, X.Sepal_Width, c=colormap[gmm_y], s=40)
plt.title('GMM Clustering using EM')
 plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
```

FEED FORWARD

```
import math
import numpy
def derivefunc(x):
        return activation(x)*(1-activation(x))
def activation(x):
        x=0-x
        return 1/(1+math.exp(x))
inp = []
wgt = [[]]
n = int(input('Enter no. of inputs :'))
for i in range(0,n):
        x=int(input('Enter input value :'))
        inp.append(x)
hn=int(input('No. of nodes in hidden layer'))
hw=[0]*hn
hd=[0]*hn
for i in range(0,hn):
        hw.append(int(input('Enter hidden weight : ')))
hw.pop(0)
for i in range(0,hn):
        ex=[]
        for j in range(0,n):
                hf=int(input('Enter value of weights :'))
                ex.append(hf)
        wgt.append(ex)
wgt.pop(0)
def feedf():
        for i in range(0,hn):
                val=0
                for j in range(0,n):
                        val= val+wgt[j][i]*inp[j]
                b=int(input('Enter bias'))
                val=val+b
                hd[i]=activation(val)
        val=0
        for i in range(0,hn):
                val=val+hd[i]*hw[i]
                b=int(input('Enter bias :'))
                val=val+b
                val=activation(val)
                print(val)
                return val
```

BACK PROPAGATION

```
#import numpy
import math
def derivefunc(x):
        return activation(x)*(1-activation(x))
def activation(x):
        x=0-x
        return 1/(1+math.exp(x))
inp = []
wgt = [[]]
n = int(input('Enter no. of inputs :'))
for i in range(0,n):
        x=int(input('Enter input value :'))
        inp.append(x)
hn=int(input('No. of nodes in hidden layer'))
hw=[0]*hn
hd=[0]*hn
for i in range(0,hn):
        hw.append(int(input('Enter hidden weight : ')))
hw.pop(0)
for i in range(0,hn):
        ex=[]
        for j in range(0,n):
                hf=int(input('Enter value of weights :'))
                ex.append(hf)
        wgt.append(ex)
wgt.pop(0)
def feedf():
        for i in range(0,hn):
                val=0
                for j in range(0,n):
                val= val+wgt[j][i]*inp[j]
        b=int(input('Enter bias'))
        val=val+b
        hd[i]=activation(val)
val=0
for i in range(0,hn):
        val=val+hd[i]*hw[i]
b=int(input('Enter bias :'))
val=val+b
val=activation(val)
print(val)
return val
val=feedf()
for i in range(0,hn):
        hw[i]=hw[i]+0.1*derivefunc(val)*hd[i]
        print(hw[i])
for i in range(0,hn):
        for j in range(0,n):
        wgt[j][i]=wgt[j][i]+0.1*derivefunc(hd[i])*inp[j]
        print(wgt[j][i])
abc=feedf()
print(abc)
```

KNN

```
from sklearn import datasets
import random
import math
iris=datasets.load_iris()
arr=list(iris.data)
tr=[[]]
for row in arr:
	tr.append(list(row))
res=list(iris.target)
tr.pop(0)
c=0
for i in tr:
	if res[c]==0:
		i.extend([0])
	if res[c]==1:
		i.extend([1])
	if res[c]==2:
		i.extend([2])
	c=c+1
tr_data=[[]]
ts_data=[[]]
random.shuffle(tr)
for i in range(0,int(2*len(tr)/3)):
	tr_data.append(tr[i])
for i in range(int(2*len(tr)/3),len(tr)):
	ts_data.append(tr[i])
tr_data.pop(0)
ts_data.pop(0)
#print("Training Set :- ")
#print(tr_data)
#print("Testing Set :- ")
#print(ts_data)
def euclidean_distance(row1, row2):
	distance = 0.0
	for i in range(len(row1)-1):
```

```
                distance += (row1[i] - row2[i])**2
        return math.sqrt(distance)
def get_neighbors(train, test_row, num_neighbors):
        distances = list()
        for train_row in train:
                dist = euclidean_distance(test_row, train_row)
                distances.append((train_row, dist))
distances.sort(key=lambda tup: tup[1])
neighbors = list()
for i in range(num_neighbors):
        neighbors.append(distances[i][0])
return neighbors

def predict_classification(train, test_row, num_neighbors):
        neighbors = get_neighbors(train, test_row, num_neighbors)
        output_values = [row[-1] for row in neighbors]
        prediction = max(set(output_values), key=output_values.count)
        return prediction
i=0
cor=0
for i in range(len(ts_data)):
        pred=predict_classification(tr_data,ts_data[i],3)
        print(pred)
        if ts_data[i][4] == pred:
                cor=cor+1
print('Correctly predicted : ',cor)
print('Total Tests : ',i)
print('Total Percentage : ',(cor/i)*100)
```

WEEK-2

```
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])
      clstt.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)
```