AI LAB

1) FIND-S

```
import csv
import random
a=[[]]
with open("C:/python55/1602-17-733-103/ws.csv",'r') as CSVFile:
  reader=csv.reader(CSVFile)
  for row in reader:
            a.append(row)
  print(a)
fs=['0']*(len(a[1])-1)
print(fs)
rlen=len(a)
for i in range(0,rlen):
            alen=len(a[i])
            if alen == 0:
            continue
            elif a[i][alen-1] == 'Yes':
            for j in range(0,alen-1):
            if fs[j]=='0':
                        fs[j]=a[i][j]
            elif fs[j]=='?':
                        continue
            elif fs[j] != a[i][j]:
                        fs[j]='?'
            else:
                        fs[j]=a[i][j]
            print(fs)
print(fs)
```

2) Feed Forward

```
#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
```

3) 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)
```

4) 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)
```

5) Locally weighted regression

```
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))) # eye - identity
            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)
            ypred = np.zeros(m)
            for i in range(m):
            ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
            return ypred

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

data = pd.read_csv('tips.csv')
bill = np.array(data.total_bill) # We use only Bill amount and Tips data
tip = np.array(data.tip)
mbill = np.mat(bill) # .mat will convert nd array is converted in 2D array
mtip = np.mat(tip)
m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T)) # 244 rows, 2 cols
ypred = localWeightRegression(X,mtip,3) # increase k to get smooth curves
graphPlot(X,ypred)
```

6) K means

```
from operator import itemgetter
import numpy
import random
from sklearn import datasets
import matplotlib.pyplot as plt

def newcent(clus):
  a=0
  b=0
  n=len(clus)
  for i in range(n):
    a=a+clus[i][0]
    b=b+clus[i][1]
  return [a/n,b/n]


def eucdist(p1,p2):
  return ((p2[0]-p1[0])**2+(p2[1]-p1[1])**2)**(0.5)
```

```
def getmin(a,b,c):
  if(a<=b and a<=c):
    return 1
  elif(b<=a and b<=c):
    return 2
  else:
    return 3

iris=datasets.load_iris()
data=list(iris.data)
target=list(iris.target)
n=len(target)
for i in range(n):
    data[i]=list(data[i])

newattr=[]
for i in range(n):
  newattr.append(data[i][0:2])

k1=random.choice(newattr)
k2=random.choice(newattr)
k3=random.choice(newattr)
#print(k1,k2,k3)

newattr.remove(k1)
newattr.remove(k2)
newattr.remove(k3)
n=len(newattr)
c1=[k1]
c2=[k2]
c3=[k3]
for i in range(n):
  clusno=getmin(eucdist(newattr[i],k1),eucdist(newattr[i],k2),eucdist(newattr[i],k3))
  if(clusno==1):
    c1.append(newattr[i])
    k1=newcent(c1)
  elif(clusno==2):
    c2.append(newattr[i])
    k2=newcent(c2)
  elif(clusno==3):
    c3.append(newattr[i])
    k3=newcent(c3)
#print(k1,k2,k3)
```

```
xcor=[]
ycor=[]
for i in range(len(c1)):
  xcor.append(c1[i][0])
  ycor.append(c1[i][1])


plt.scatter(xcor,ycor,c='g',marker='o')
xcor=[]
ycor=[]
for i in range(len(c2)):
  xcor.append(c2[i][0])
  ycor.append(c2[i][1])

plt.scatter(xcor,ycor,c='r',marker='o')
xcor=[]
ycor=[]
for i in range(len(c3)):
  xcor.append(c3[i][0])
  ycor.append(c3[i][1])

plt.scatter(xcor,ycor,c='y',marker='o'
```

7) EM

```
import matplotlib.pyplot as plt
from sklearn import datasets
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']
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)
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y = gmm.predict(xs)
#print("mean:\n",gmm.means_)
#print('\n')
#print("Covariances\n",gmm.covariances_)
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')
```

8) DFS

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

visited = [] # Array to keep track of visited nodes.

def dfs(visited, graph, node):
  if node not in visited:
      print node,
      visited.append(node)
      for neighbour in graph[node]:
          dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
```

9) BFS

```
graph = {
 'A' : ['B','C'],
 'B' : ['D', 'E'],
 'C' : ['F'],
 'D' : [],
 'E' : ['F'],
 'F' : []
}

visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a 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)

# Driver Code
bfs(visited, graph, 'A')
```

10) IDDFS

```
from collections import defaultdict
class Graph:

        def __init__(self,vertices):

                self.V = vertices
                self.graph = defaultdict(list)

        def addEdge(self,u,v):
                self.graph[u].append(v)

        def DLS(self,src,target,maxDepth):

                if src == target : return True
                if maxDepth <= 0 : return False
                for i in self.graph[src]:
                                if(self.DLS(i,target,maxDepth-1)):
                                        return True
                return False

        def IDDFS(self,src, target, maxDepth):

                for i in range(maxDepth):
                        if (self.DLS(src, target, i)):
                                return True
                return False

g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0
if g.IDDFS(src, target, maxDepth) == True:
        print ("Target is reachable from source " +"within max depth")
else :
        print ("Target is NOT reachable from source " +"within max depth")
```

11) Water jug problem

```
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)
```

12) nQueens

```
class NQueens:
    """Generate all valid solutions for the n queens puzzle"""
    def __init__(self, size):
        # Store the puzzle (problem) size and the number of valid solutions
        self.size = size
        self.solutions = 0
        self.solve()

    def solve(self):
        """Solve the n queens puzzle and print the number of solutions"""
        positions = [-1] * self.size
        self.put_queen(positions, 0)
        print("Found", self.solutions, "solutions.")

    def put_queen(self, positions, target_row):
        """
        Try to place a queen on target_row by checking all N possible
cases.
        If a valid place is found the function calls itself trying to place
a queen
        on the next row until all N queens are placed on the NxN board.
        """
        # Base (stop) case - all N rows are occupied
        if target_row == self.size:
            self.show_full_board(positions)
            # self.show_short_board(positions)
            self.solutions += 1
        else:
            # For all N columns positions try to place a queen
            for column in range(self.size):
                # Reject all invalid positions
                if self.check_place(positions, target_row, column):
                    positions[target_row] = column
                    self.put_queen(positions, target_row + 1)


    def check_place(self, positions, ocuppied_rows, column):
        """
        Check if a given position is under attack from any of
        the previously placed queens (check column and diagonal positions)
        """
        for i in range(ocuppied_rows):
            if positions[i] == column or \
                positions[i] - i == column - ocuppied_rows or \
                positions[i] + i == column + ocuppied_rows:

                return False
        return True

    def show_full_board(self, positions):
        """Show the full NxN board"""
        for row in range(self.size):
            line = ""
            for column in range(self.size):
                if positions[row] == column:
                    line += "Q "
                else:
                    line += ". "
            print(line)
        print("\n")

    def show_short_board(self, positions):
        """
        Show the queens positions on the board in compressed form,
        each number represent the occupied column position in the
corresponding row.
        """
        line = ""
        for i in range(self.size):
            line += str(positions[i]) + " "
        print(line)

n=int(input('Enter size of board:'))
NQueens(n)
```

AlphaBetaPruning :

```
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)
    return ans
```

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()
```

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:])
```

MinMaxTicTacToe

```
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','_','_','_','_','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
```