# AI&NN Lab programs

# Program 5

# Number of vertices in the graph

# define 4 4

# check if the colored

# graph is safe or not

def isSafe(graph, color):

    # check for every edge

    for i in range(4):

        for j in range(i + 1, 4):

            if (graph[i][j] and color[j] == color[i]):

                return False

    return True

# /\* This function solves the m Coloring

# problem using recursion. It returns

# false if the m colours cannot be assigned,

# otherwise, return true and prints

# assignments of colours to all vertices.

# Please note that there may be more than

# one solutions, this function prints one

# of the feasible solutions.\*/

def graphColoring(graph, m, i, color):

    # if current index reached end

    if (i == 4):

        # if coloring is safe

        if (isSafe(graph, color)):

            # Print the solution

            printSolution(color)

            return True

        return False

    # Assign each color from 1 to m

    for j in range(1, m + 1):

        color[i] = j

        # Recur of the rest vertices

        if (graphColoring(graph, m, i + 1, color)):

            return True

        color[i] = 0

    return False

# /\* A utility function to prsolution \*/

def printSolution(color):

    print("Solution Exists:" " Following are the assigned colors ")

    for i in range(4):

        print(color[i],end=" ")

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

    # /\* Create following graph and

    # test whether it is 3 colorable

    # (3)---(2)

    # | / |

    # | / |

    # | / |

    # (0)---(1)

    # \*/

    graph = [

        [ 0, 1, 1, 1 ],

        [ 1, 0, 1, 0 ],

        [ 1, 1, 0, 1 ],

        [ 1, 0, 1, 0 ],

    ]

    m = 3 # Number of colors

    # Initialize all color values as 0.

    # This initialization is needed

    # correct functioning of isSafe()

    color = [0 for i in range(4)]

    if (not graphColoring(graph, m, 0, color)):

        print ("Solution does not exist")

# Program6

# Implementation of Perceptron algorithm

import numpy as np

X = np.array([

[-2,4,-1],

[4,1,-1],

[1, 6, -1],

[2, 4, -1],

[6, 2, -1],

])

y = np.array([-1,-1,1,1,1])

def perceptron\_sgd(X, Y):

w = np.zeros(len(X[0]))

eta = 1

epochs = 20

for t in range(epochs):

for i, x in enumerate(X):

if (np.dot(X[i], w)\*Y[i]) <= 0:

w = w + eta\*X[i]\*Y[i]

return w

w = perceptron\_sgd(X,y)

print(w)

import numpy as np

X = np.array([

[-2,4,-1],

[4,1,-1],

[1, 6, -1],

[2, 4, -1],

[6, 2, -1],

])

y = np.array([-1,-1,1,1,1])

def perceptron\_sgd(X, Y):

w = np.zeros(len(X[0]))

eta = 1

epochs = 20

for t in range(epochs):

for i, x in enumerate(X):

if (np.dot(X[i], w)\*Y[i]) <= 0:

w = w + eta\*X[i]\*Y[i]

return w

w = perceptron\_sgd(X,y)

print(w)

import numpy as np

from matplotlib import pyplot as plt

%matplotlib inline

X = np.array([

[-2, 4],

[4, 1],

[1, 6],

[2, 4],

[6, 2]

])

X = np.array([

[-2,4,-1],

[4,1,-1],

[1, 6, -1],

[2, 4, -1],

[6, 2, -1],

])

y = np.array([-1,-1,1,1,1])

for d, sample in enumerate(X):

# Plot the negative samples

if d < 2:

plt.scatter(sample[0], sample[1], s=120, marker='\_', linewidths=2)

# Plot the positive samples

else:

plt.scatter(sample[0], sample[1], s=120, marker='+', linewidths=2)

# Print a possible hyperplane, that is seperating the two classes.

plt.plot([-2,6],[6,0.5])

def perceptron\_sgd(X, Y):

w = np.zeros(len(X[0]))

eta = 1

epochs = 10

for epoch in range(epochs):

for i, x in enumerate(X):

if (np.dot(X[i], w)\*Y[i]) <= 0:

w = w + eta\*X[i]\*Y[i]

return w

def perceptron\_sgd\_plot(X, Y):

'''

train perceptron and plot the total loss in each epoch.

:param X: data samples

:param Y: data labels

:return: weight vector as a numpy array

'''

w = np.zeros(len(X[0]))

eta = 1

n = 30

errors = []

for t in range(n):

total\_error = 0

for i, x in enumerate(X):

if (np.dot(X[i], w)\*Y[i]) <= 0:

total\_error += (np.dot(X[i], w)\*Y[i])

w = w + eta\*X[i]\*Y[i]

errors.append(total\_error\*-1)

plt.plot(errors)

plt.xlabel('Epoch')

plt.ylabel('Total Loss')

return w

print(perceptron\_sgd\_plot(X,y))

# Program2

#Implementation of Two Player Tic-Tac-Toe game in Python.

''' We will make the board using dictionary

in which keys will be the location(i.e : top-left,mid-right,etc.)

and initialliy it's values will be empty space and then after every move

we will change the value according to player's choice of move. '''

theBoard = {'7': ' ' , '8': ' ' , '9': ' ' ,

'4': ' ' , '5': ' ' , '6': ' ' ,

'1': ' ' , '2': ' ' , '3': ' ' }

board\_keys = []

for key in theBoard:

board\_keys.append(key)

''' We will have to print the updated board after every move in the game and

thus we will make a function in which we'll define the printBoard function

so that we can easily print the board everytime by calling this function. '''

def printBoard(board):

print(board['7'] + '|' + board['8'] + '|' + board['9'])

print('-+-+-')

print(board['4'] + '|' + board['5'] + '|' + board['6'])

print('-+-+-')

print(board['1'] + '|' + board['2'] + '|' + board['3'])

# Now we'll write the main function which has all the gameplay functionality.

def game():

turn = 'X'

count = 0

for i in range(10):

printBoard(theBoard)

print("It's your turn," + turn + ".Move to which place?")

move = input()

if theBoard[move] == ' ':

theBoard[move] = turn

count += 1

else:

print("That place is already filled.\nMove to which place?")

continue

# Now we will check if player X or O has won,for every move after 5 moves.

if count >= 5:

if theBoard['7'] == theBoard['8'] == theBoard['9'] != ' ': # across the top

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['4'] == theBoard['5'] == theBoard['6'] != ' ': # across the middle

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['2'] == theBoard['3'] != ' ': # across the bottom

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['4'] == theBoard['7'] != ' ': # down the left side

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['2'] == theBoard['5'] == theBoard['8'] != ' ': # down the middle

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['3'] == theBoard['6'] == theBoard['9'] != ' ': # down the right side

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['7'] == theBoard['5'] == theBoard['3'] != ' ': # diagonal

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['5'] == theBoard['9'] != ' ': # diagonal

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

# If neither X nor O wins and the board is full, we'll declare the result as 'tie'.

if count == 9:

print("\nGame Over.\n")

print("It's a Tie!!")

# Now we have to change the player after every move.

if turn =='X':

turn = 'O'

else:

turn = 'X'

# Now we will ask if player wants to restart the game or not.

restart = input("Do want to play Again?(y/n)")

if restart == "y" or restart == "Y":

for key in board\_keys:

theBoard[key] = " "

game()

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

game()