**Lab work – 1 (Simple Intelligence Agent)**

For this lab work, you must work with the given code using **matplotlib, random, heapq, numpy** library in Python. Create an instance of the environment**, not having less than 5 columns and 5 rows,** using the **WorldMap()** class. You also need to initiate a single vacuum cleaner agent and run your agent till it completes 25 random moves.

Try to understand the given Python code for the environment class **WorldMap()** and its different methods. Also, see how **matplotlib.pyplot** is used to create a visualization of the **WorldMap()** instance you create. Also, ensure the agent can not alter the obstacles (obs) and only the **dirt\_blocks.**

Return the environment instance after the agent completes its 25 moves. Also, show how you get different results for different iterations.

**Program:**

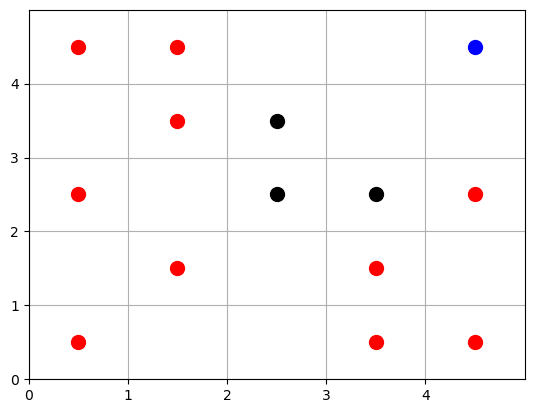
import matplotlib.pyplot as plt  
import random  
import heapq  
import numpy as np

class WorldMap:  
 def \_\_init\_\_(self, rows, cols, num\_dirt\_blocks, num\_obs):  
 self.rows = rows  
 self.cols = cols  
 self.num\_dirt\_blocks = num\_dirt\_blocks  
 self.num\_obs = num\_obs  
 self.world\_map = [['clean' for \_ in range(cols)] for \_ in range(rows)]  
  
 self.agent\_positions = {} # Dictionary to store agent positions  
  
 # Place dirt blocks randomly on the map  
 for \_ in range(num\_dirt\_blocks):  
 row = random.randint(0, rows - 1)  
 col = random.randint(0, cols - 1)  
 while self.world\_map[row][col] == 'dirt' or self.world\_map[row][col] == 'agent':  
 row = random.randint(0, rows - 1)  
 col = random.randint(0, cols - 1)  
 self.world\_map[row][col] = 'dirt'  
  
 # Place obstacles randomly on the map (excluding corners)  
 for \_ in range(num\_obs):  
 row = random.randint(1, rows - 2) # Avoid corners  
 col = random.randint(1, cols - 2) # Avoid corners  
 while self.world\_map[row][col] == 'dirt' or self.world\_map[row][col] == 'agent' or self.world\_map[row][col] == 'obs':  
 row = random.randint(1, rows - 2) # Avoid corners  
 col = random.randint(1, cols - 2) # Avoid corners  
 self.world\_map[row][col] = 'obs'  
  
 def add\_agent(self, agent\_id):  
 while True:  
 row = random.randint(0, self.rows - 1)  
 col = random.randint(0, self.cols - 1)  
 if self.world\_map[row][col] == 'clean':  
 self.world\_map[row][col] = 'agent'  
 self.agent\_positions[agent\_id] = (col,row)  
 break  
  
  
 def getAgentPos(self, agent\_id):  
 if agent\_id in self.agent\_positions:  
 return self.agent\_positions[agent\_id]  
 else:  
 return None # Agent not found  
  
 def move\_agent(self, agent\_id, new\_position):  
 if agent\_id in self.agent\_positions:  
 current\_position = self.agent\_positions[agent\_id]  
 if self.world\_map[current\_position[1]][current\_position[0]] == 'agent':  
 self.world\_map[current\_position[1]][current\_position[0]] = 'clean' # Clear the current cell  
 self.world\_map[new\_position[1]][new\_position[0]] = 'agent' # Place the agent in the new cell  
 self.agent\_positions[agent\_id] = new\_position # Update the agent's position  
  
  
 def display\_map(self):  
 fig,ax = plt.subplots() # Clear the current plot  
 for row in range(self.rows):  
 for col in range(self.cols):  
 if self.world\_map[row][col] == 'dirt':  
 ax.plot(col + 0.5, row + 0.5, 'ro', markersize=10) # Display dirt as red dots  
 elif self.world\_map[row][col] == 'agent':  
 ax.plot(col + 0.5, row + 0.5, 'bo', markersize=10) # Display agents as blue dots  
 elif self.world\_map[row][col] == 'obs':  
 ax.plot(col + 0.5, row + 0.5, 'ko', markersize=10) # Display obstacles as black dots  
  
 ax.set\_xlim(0, self.cols)  
 ax.set\_ylim(0, self.rows)  
  
 ax.set\_xticks(range(self.cols))  
 ax.set\_yticks(range(self.rows))  
 ax.grid()  
  
 def is\_valid\_position(self, row, col):  
 return 0 <= row < self.rows and 0 <= col < self.cols

world = WorldMap(5,5,10,3)  
world.add\_agent('A')

print(world.world\_map)  
world.display\_map()

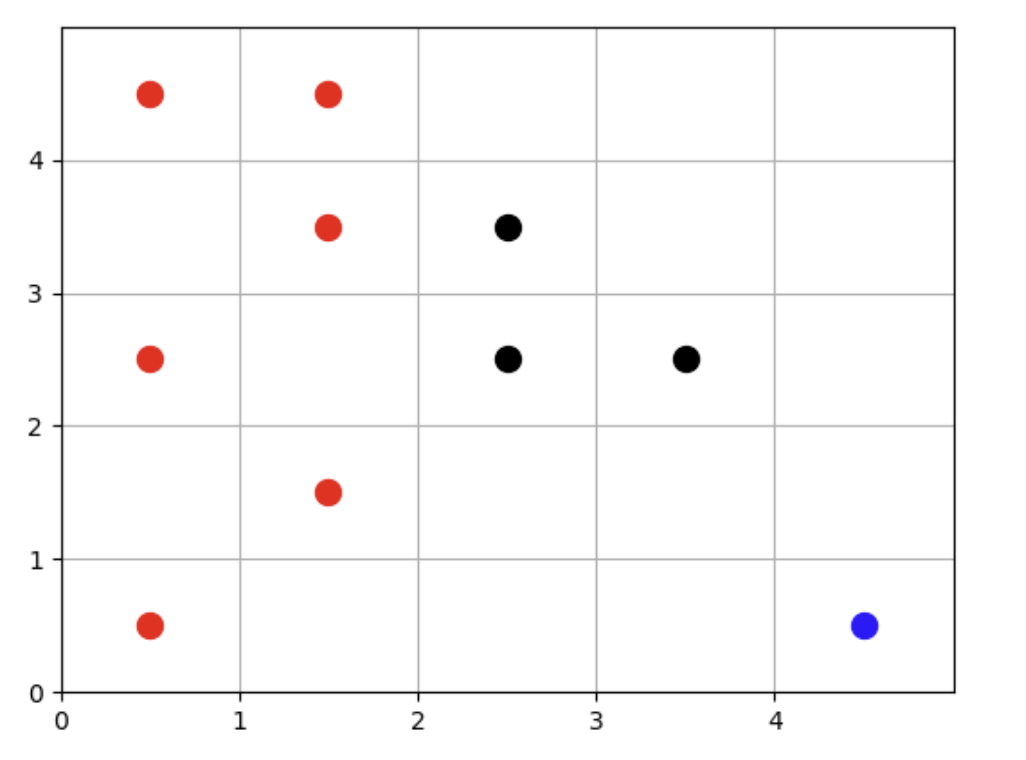
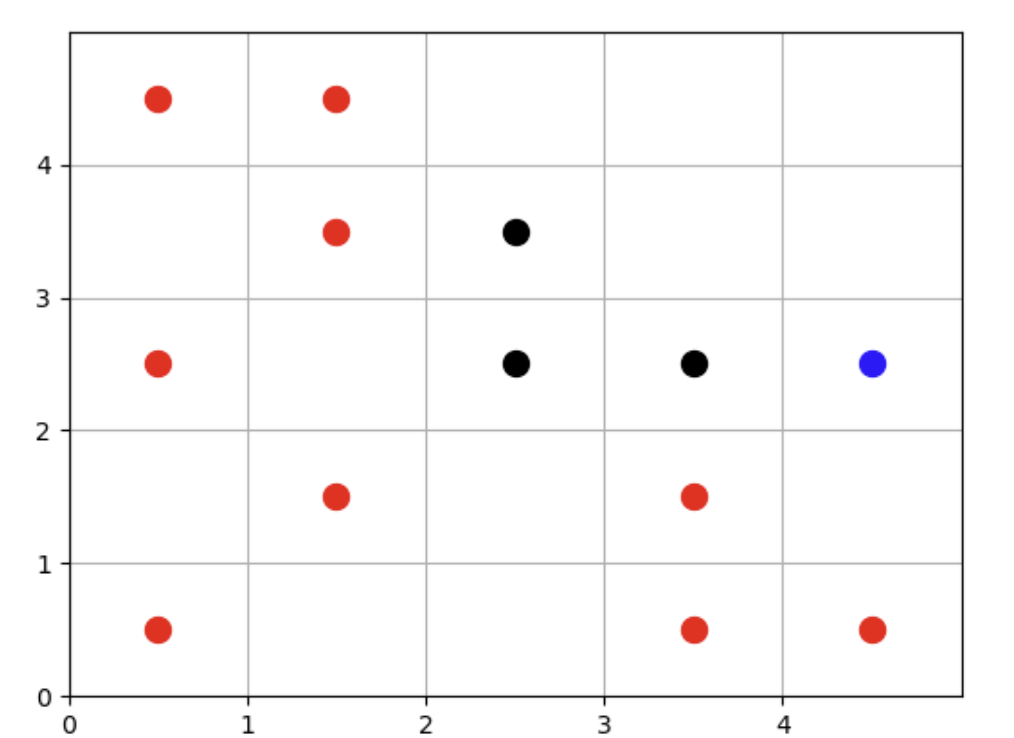
[['dirt', 'clean', 'clean', 'dirt', 'dirt'], ['clean', 'dirt', 'clean', 'dirt', 'clean'], ['dirt', 'clean', 'obs', 'obs', 'dirt'], ['clean', 'dirt', 'obs', 'clean', 'clean'], ['dirt', 'dirt', 'clean', 'clean', 'agent']]



path=[world.getAgentPos('A')]  
count = 0  
while count < 25:  
 a = random.randint(0,4)  
 if a == 0:  
 pos = world.getAgentPos('A')  
 right = (pos[0],pos[1]+1)  
 if world.is\_valid\_position(right[1],right[0]) and world.world\_map[right[1]][right[0]] != 'obs': #and right not in path:  
 world.move\_agent('A',right)  
 path.append(right)  
 count += 1  
 world.display\_map()  
   
 elif a == 1:  
 pos = world.getAgentPos('A')  
 left = (pos[0],pos[1]-1)  
 if world.is\_valid\_position(left[1],left[0]) and world.world\_map[left[1]][left[0]] != 'obs':#and left not in path:  
 world.move\_agent('A',left)  
 path.append(left)  
 count += 1  
 world.display\_map()  
   
 elif a == 2:  
 pos = world.getAgentPos('A')  
 up = (pos[0]+1,pos[1])  
 if world.is\_valid\_position(up[1],up[0]) and world.world\_map[up[1]][up[0]] != 'obs':#and up not in path:  
 world.move\_agent('A',up)  
 path.append(up)  
 count += 1  
 world.display\_map()  
   
 elif a == 3:  
 pos = world.getAgentPos('A')  
 down = (pos[0]-1,pos[1])  
 if world.is\_valid\_position(down[1],down[0]) and world.world\_map[down[1]][down[0]] != 'obs':#and down not in path:  
 world.move\_agent('A',down)  
 path.append(down)  
 count += 1  
 world.display\_map()  
  
print(world.world\_map)  
  
print(path)

**OUTPUT : Path**

[['dirt', 'clean', 'clean', 'clean', 'agent'], ['clean', 'dirt', 'clean', 'clean', 'clean'], ['dirt', 'clean', 'obs', 'obs', 'clean'], ['clean', 'dirt', 'obs', 'clean', 'clean'], ['dirt', 'dirt', 'clean', 'clean', 'clean']]  
[(4, 4), (3, 4), (3, 3), (4, 3), (4, 4), (4, 3), (4, 2), (4, 1), (4, 0), (4, 1), (3, 1), (4, 1), (4, 2), (4, 1), (4, 0), (3, 0), (3, 1), (4, 1), (3, 1), (3, 0), (4, 0)]



* final state:

**Lab work – 2 (DFS and BFS implementation)**

For this lab work, you are required to implement the Breadth First Search (BFS) and Depth First Search (DFS) algorithms to solve the 8-puzzle game. To see how the 8-puzzle game works you can go to this link - http://www.puzzlopia.com/puzzles/puzzle 8/play.  You can refer to the lecture slides for the pseudocode of the BFS and DFS algorithms.

Note that the implementations are similar but the only difference is in the implementation of the **frontier (BFS – Queue (FIFO), DFS – Stack (LIFO)).**

The starter code has already been provided to you. It consists of the following: Queue class – implementation of the queue data structure

Stack class – implementation of the stack data structure

State class – this class models the 8 puzzle game states, an instance of this class can be thought of as a state in the state space or a tree node in the search space.

Comments have been provided to you in the code, so be sure to read them and get familiar with what the code is doing. You will specifically find the get\_children() function helpful. Your task is to fill in the code for the bfs() and dfs() functions respectively. Both functions should return the set of moves required to solve the puzzle as well as the total number of moves required given some initial starting state.Running times will vary depending on your implementation. Improper implementation may result in longer running times.

Also, you will notice the different solutions returned by DFS and BFS. Analyze the time complexity of the solutions and conclude it using the time library.

**CODE:**

import matplotlib.pyplot as plt  
import numpy as np

class Queue():  
 def\_\_init\_\_(self, initial):  
 self.items = [initial]  
 def isEmpty(self):  
 return self.items == []  
 def enqueue(self, item):  
 self.items.insert(0,item)  
 def dequeue(self):  
 return self.items.pop()  
 def size(self):  
 return len(self.items)

class Stack():  
 def\_\_init\_\_(self, initial):  
 self.items = [initial]  
 def isEmpty(self):  
 return self.items == []  
 def push(self, item):  
 self.items.append(item)  
 def pop(self):  
 return self.items.pop()  
 def size(self):  
 return len(self.items)

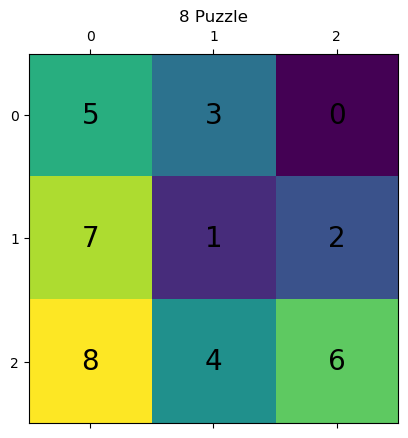
class State():  
 right = {0, 1, 3, 4, 6, 7}  
 left = {1, 2, 4, 5, 7, 8}  
 up = {3, 4, 5, 6, 7, 8}  
 down = {0, 1, 2, 3, 4, 5}  
  
 def\_\_init\_\_(self, board\_config, parent, move):  
 self.board\_config = board\_config # board configuration of the current state in a string  
 self.board\_config\_list = list(map(int,board\_config.split(','))) # board configuration of the current state in a list  
 #print(self.board\_config\_list)  
 self.i = self.board\_config\_list.index(0) # index of empty space in board (index of 0 in this case)  
 self.parent = parent # parent state (node) of the present state  
 self.move = move # the move (Up,Down,Left,Right) made in parent state that results in the present state  
 def get\_children(self):  
 """returns the list of all possible states reachable from the current state,  
 each child in the list is a State object"""  
 children = []  
 if self.i in State.up:  
 new\_board\_config = self.board\_config\_list[:]  
 new\_board\_config[self.i], new\_board\_config[self.i-3] = new\_board\_config[self.i-3], new\_board\_config[self.i]  
 children.append(State(','.join(map(str,new\_board\_config)), self.board\_config,'Up'))  
  
 if self.i in State.down:  
 new\_board\_config = self.board\_config\_list[:]  
 new\_board\_config[self.i], new\_board\_config[self.i+3] = new\_board\_config[self.i+3], new\_board\_config[self.i]  
 children.append(State(','.join(map(str,new\_board\_config)), self.board\_config,'Down'))  
 if self.i in State.left:  
 new\_board\_config = self.board\_config\_list[:]  
 new\_board\_config[self.i], new\_board\_config[self.i-1] = new\_board\_config[self.i-1], new\_board\_config[self.i]  
 children.append(State(','.join(map(str,new\_board\_config)), self.board\_config,'Left'))  
  
 if self.i in State.right:  
 new\_board\_config = self.board\_config\_list[:]   
 new\_board\_config[self.i], new\_board\_config[self.i+1] = new\_board\_config[self.i+1], new\_board\_config[self.i]  
 children.append(State(','.join(map(str,new\_board\_config)), self.board\_config,'Right'))  
 return children  
 def plot\_8\_puzzle(self):  
 board = np.array([int(x) for x in self.board\_config.split(',')]).reshape(3, 3)  
   
 fig, ax = plt.subplots()  
 ax.matshow(board)  
   
 for i in range(3):  
 for j in range(3):  
 ax.text(j, i, str(board[i, j]), va='center', ha='center', fontsize=20, color='black')  
  
 plt.title('8 Puzzle')  
 plt.show()  
   
 def \_\_str\_\_(self):  
 return self.board\_config

def dfs(initial,goal):  
 frontier = Stack(initial)  
 frontier\_set = set()  
 count = 0  
 while not frontier.isEmpty():  
 count +=1  
 state = frontier.pop()  
 frontier\_set.add(state.board\_config)  
 if state.board\_config == goal:  
 return "success"  
 else:  
 for child in state.get\_children():  
 if child.board\_config not in frontier\_set:  
 frontier.push(child)  
 return 'failure'

def bfs(initaial, goal):  
 frontier = Queue(initaial)  
 frontier\_set = set()  
 count = 0  
 while not frontier.isEmpty():  
 count +=1  
 state = frontier.dequeue()  
 frontier\_set.add(state.board\_config)  
 if state.board\_config == goal:  
 return "success"  
 else:  
 for child in state.get\_children():  
 if child.board\_config not in frontier\_set:  
 frontier.enqueue(child)  
 return 'failure'

start = '5,3,0,7,1,2,8,4,6'  
goal = '0,1,2,3,4,5,6,7,8'  
initial\_state = State(start, None, None)

initial\_state.plot\_8\_puzzle()



import time  
s=time.time()  
print(bfs(initial\_state, goal))  
f=time.time()  
e=f-s  
print(e)

**output:**

**success  
12.166506290435791**

import time  
s=time.time()  
print(dfs(initial\_state, goal))  
f=time.time()  
e=f-s  
print(e)

**Output:**

**success  
4.3853373527526855**

**LAB :3 (A\* Search)**

This lab work is a continuation of lab work 1. You are required to implement the A\* search algorithm to solve the 8 puzzle game. To see how the 8 puzzle game works you can go to this link - <http://www.puzzlopia.com/puzzles/puzzle-8/play>. You can refer to the lecture slides for the pseudocode of the algorithms. Note that the implementation of the frontier for A\* is a priority queue The starter code has already been provided to you. It consists of the following: You are required to implement the priority queue for A\* search.

State class – this class models the 8 puzzle game states, an instance of this class can be thought of as a state in the state space or a tree node in the search space. Comments have been provided to you in the code, so be sure to read them and get familiar with what the code is doing. You will specifically find the get\_children() function helpful.

Your task is to fill in code for the ast() functions respectively. All functions should return the set of moves required to solve the puzzle as well as the total number of moves required given some initial starting state. Running times will vary depending on your implementation. Improper implementation may result in longer running times. However, if your code is written fairly well, the program should find the solution for any random starting state in less than 10 seconds.

Also, you will notice the different solutions returned A\*. Analyze the solutions and conclude it. Also, analyze their running times.

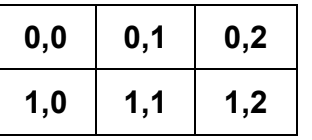
For A\* search to work properly a good heuristic function should be used. As discussed in the lecture, you can use the Manhattan distance as the heuristic which determines how close a current state is to the goal.

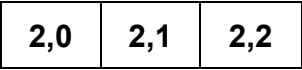
To calculate the Manhattan distance between 2 points, use the following formula:

if a = (x1, y1) and b = (x2, y2),

then the manhattan distance between a and b will be given by

d = |x1 − x2| + |y1 − y2|

You can think of the different positions in the board as different points in a 2d space. So each board position will have the following coordinates (x,y) 



So the Manhattan distance between the board positions 2,2 and 0,1 will be:

x1, y1 = 2,2; x2, y2 = 0,1;

d = |2 − 0| + |2 − 1| = 3

The overall cost function for A\* search will be the following:

f(n) = g(n) + h(n)

g(n) − cost for reaching a node n from the start node

h(n) − heuristic function (in this case,the sum of manhattan distances)

You can use g(n) as simply the number of steps required to reach a node from the start.For calculating this the attribute depth in the state class will be helpful.

**Code:**

import matplotlib.pyplot as plt  
import numpy as np

class PriorityQueue():  
 def \_\_init\_\_(self,item,cost):  
 self.items = {cost:[item]}  
 self.costs = {cost}  
  
 def isEmpty(self):  
 return self.items == {}  
  
 def dequeue(self):  
 least\_cost = sorted(self.costs)[0]  
 item = self.items[least\_cost].pop(0)  
  
 if len(self.items[least\_cost]) == 0:  
 self.costs.remove(least\_cost)  
 del self.items[least\_cost]  
  
 return item  
 def enqueue(self,item,cost):  
 if cost in self.costs:  
 self.items[cost].append(item)  
  
 else:  
 self.items[cost] = [item]  
 self.costs.add(cost)  
  
 def update\_cost(self,item,old\_cost,new\_cost):  
 #print(old\_cost)  
 for i in self.items[old\_cost]:  
 if i.board\_config == item.board\_config:  
 self.items[old\_cost].remove(i)  
 break  
 if len(self.items[old\_cost]) == 0:  
 self.costs.remove(old\_cost)  
 del self.items[old\_cost]  
 if new\_cost in self.costs:  
 self.items[new\_cost].append(item)  
  
 else:  
 self.items[new\_cost] = [item]  
 self.costs.add(new\_cost)

class State():  
 right=(0,1,3,4,6,7)  
 left=(1,2,4,5,7,8)  
 up=(3,4,5,6,7,8,)  
 down=(0,1,2,3,4,5)  
  
 def \_\_init\_\_(self,board\_config,parent,move,depth):  
 self.board\_config = board\_config  
 self.board\_config\_list = list(map(int,board\_config.split(',')))  
 self.i = self.board\_config\_list.index(0)  
 self.parents = parent  
 self.move = move  
 self.depth = depth  
 # print('Constructor called')  
  
 def get\_children(self):  
 children = []  
 if self.i in State.right:  
 new\_config = self.board\_config\_list[:]  
 new\_config[self.i],new\_config[self.i+1]=new\_config[self.i+1], new\_config[self.i]  
 children.append(State(','.join(map(str,new\_config)),self.board\_config,'Right',self.depth+1))  
 if self.i in State.left:  
 new\_config = self.board\_config\_list[:]  
 new\_config[self.i],new\_config[self.i-1]=new\_config[self.i-1], new\_config[self.i]  
 children.append(State(','.join(map(str,new\_config)),self.board\_config,'Left',self.depth+1))  
 if self.i in State.up:  
 new\_config = self.board\_config\_list[:]  
 new\_config[self.i],new\_config[self.i-3]=new\_config[self.i-3], new\_config[self.i]  
 children.append(State(','.join(map(str,new\_config)),self.board\_config,'Up',self.depth+1))  
 if self.i in State.down:  
 new\_config = self.board\_config\_list[:]  
 new\_config[self.i],new\_config[self.i+3]=new\_config[self.i+3], new\_config[self.i]  
 children.append(State(','.join(map(str,new\_config)),self.board\_config,'Down',self.depth+1))  
 # print(children)  
 return children  
 def plot\_8\_puzzle(self):  
 board = np.array([int(x) for x in self.board\_config.split(',')]).reshape(3, 3)  
  
 fig, ax = plt.subplots()  
 ax.matshow(board)  
  
 for i in range(3):  
 for j in range(3):  
 ax.text(j, i, str(board[i, j]), va='center', ha='center', fontsize=20, color='black')  
  
 plt.title('8 Puzzle')  
 plt.show()  
 def \_\_str\_\_(self):  
 return self.board\_config

def manhattan\_dist(x,y):  
 # print('j')  
 return abs(x[0]-y[0])+abs(x[1]-y[1])  
  
indexes = {0:(0,0), 1:(0,1),2:(0,2),3:(1,0),4:(1,1),5:(1,2), 6:(2,0), 7:(2,1),8:(2,2)}

def h(s):  
 s = s.split(',')  
 dist=0  
 # print('k')  
 for each in s :  
 i = s.index(each)  
 x = indexes[i]  
 # print('l')  
 y = indexes[int(each)]  
 dist = dist + manhattan\_dist(x,y)  
 return dist

h("0,5,3,8,2,1,7,4,6")

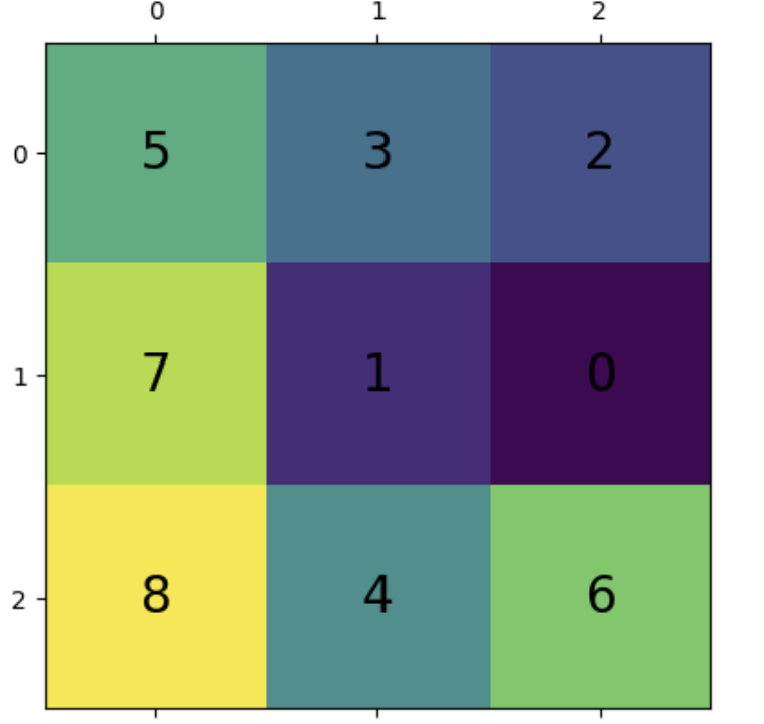
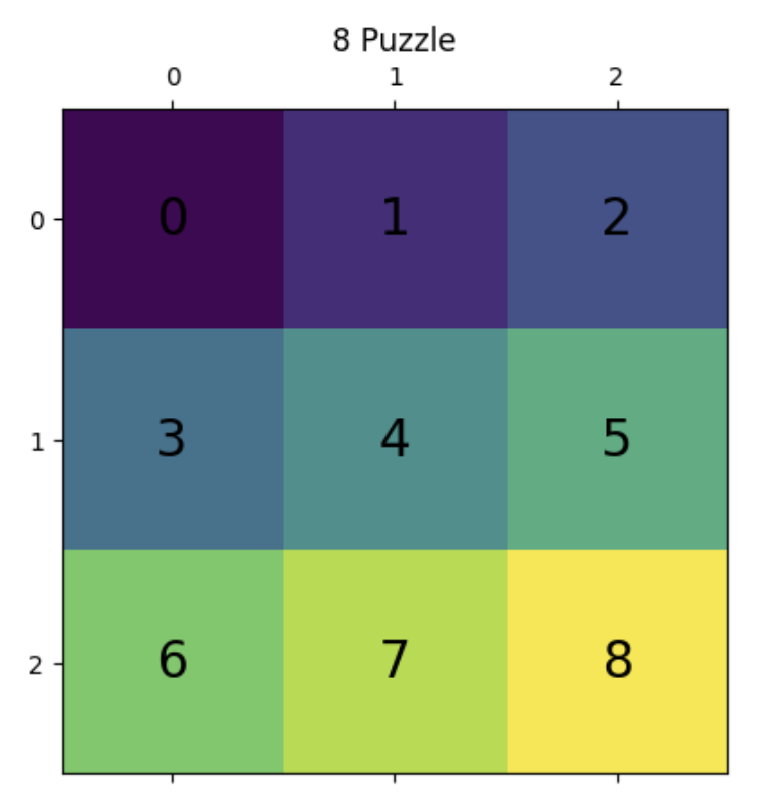
16

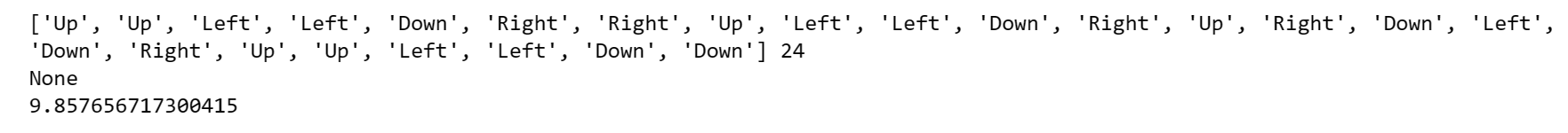
def ast(inp,goal):  
 frontier = PriorityQueue(inp,h(inp.board\_config))  
 frontier\_dict = {inp.board\_config:h(inp.board\_config)}  
 graph = {}  
  
 explored = set()  
  
 while not frontier.isEmpty():  
 state = frontier.dequeue()  
 explored.add(state.board\_config)  
 del frontier\_dict[state.board\_config]  
 graph[state.board\_config] = state  
 if state.board\_config == goal:  
 path = []  
 current\_state = state  
 print(h(goal))  
 while not current\_state.parents == None:  
 current\_state.plot\_8\_puzzle()  
 path.append(current\_state.move)  
 current\_state =graph[current\_state.parents]  
 # print('n')  
 return print (path, len(path))  
  
  
 else:  
 # print('st')  
 a = state.get\_children()  
  
  
 for children in a:  
 # print(children.board\_config)  
 #print (frontier)  
 cost= h(children.board\_config)+ children.depth  
  
 if children.board\_config not in frontier\_dict and children.board\_config not in explored:  
 frontier.enqueue(children,cost)  
 #print('o')  
 frontier\_dict[children.board\_config]= cost  
 #print('v')  
 if children.board\_config in frontier\_dict:  
 # new\_cost = cost  
 #old\_cost = frontier\_dict[children.board\_config]  
 if cost < frontier\_dict[children.board\_config]:  
 frontier.update\_cost(children,frontier\_dict[children.board\_config],cost)  
 # print('p')  
  
 return' failure'

inp= State('5,3,0,7,1,2,8,4,6',None,None,0)

import time  
s=time.time()  
print(ast(inp,'0,1,2,3,4,5,6,7,8'))  
e=time.time()  
t=e-s  
print(t)

**OUTPUT :**

****

****

**LAB :4 (Production\_rule\_based\_system)**

For this lab work, you need to write a program that can use the production rules of the water jug problem and find the solution for any input of the full capacity of jugs and the targeted volume.

Capacity of first jug =X

Capacity of second jug=Y

Targeted Volume = T

The program should return not possible if the given targeted volume can not be achieved using the chosen values of X and Y. Use the following information to make the general solution.

A pair of Jugs can only find targeted volume if

G modulo T = 0 [: where G= Highest Common Factor of X and Y]

**CODE:**

print("Solution for water jug problem")  
x\_capacity = input("Enter Jug 1 capacity:")  
y\_capacity = input("Enter Jug 2 capacity:")  
end = input("Enter target volume:")  
  
def Production\_Rule\_Implementation(start, end, x\_capacity, y\_capacity):  
 path = []  
 front = []  
 front.append(start)  
 visited = []

#visited.append(start)  
 while front:  
 current = front.pop()  
 x = current[0]  
 y = current[1]  
 path.append(current)  
 if x == end or y == end:  
 print("Found!")  
 return path

# rule 1 filling x

if current[0] < x\_capacity and ([x\_capacity, current[1]] not in visited):  
 front.append([x\_capacity, current[1]])  
 visited.append([x\_capacity, current[1]])  
  
 # rule 2 filling y

if current[1]< y\_capacity and ([current[0],y\_capacity] not in visited):  
 front.append([current[0], y\_capacity])  
 visited.append([current[0], y\_capacity])  
  
 # rule 3 emptying x  
 if current[0] > x\_capacity and ([0, current[1]] not in visited):  
 front.append([0, current[1]])  
 visited.append([0, current[1]])  
  
 # rule 4 emptying y  
 if current[1] > y\_capacity and ([x\_capacity, 0] not in visited):  
 front.append([x\_capacity, 0])  
 visited.append([x\_capacity, 0])

# rule 5 pouring y into x, both in case y is empty or y is left with some#(x, y) -> (min(x + y, x\_capacity), max(0, x + y - x\_capacity)) if y > 0

if current[1] > 0 and ([min(x + y, x\_capacity), max(0, x + y - x\_capacity)] not in visited):  
 front.append([min(x + y, x\_capacity), max(0, x + y - x\_capacity)])  
 visited.append([min(x + y, x\_capacity), max(0, x + y - x\_capacity)])  
  
# rule 6 pouring x into y, both in case x is empty or x is left with some  
# (x, y) -> (max(0, x + y - y\_capacity), min(x + y, y\_capacity)) if x > 0

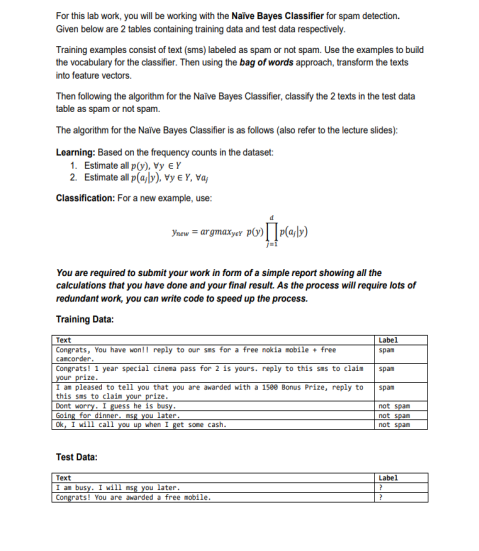
if current[0] > 0 and ([max(0, x + y - y\_capacity), min(x + y, y\_capacity)] not in visited):  
 front.append([max(0, x + y - y\_capacity), min(x + y, y\_capacity)])  
 visited.append([max(0, x + y - y\_capacity), min(x + y, y\_capacity)])  
  
 return "Not found"

def gcd(a, b):  
 if a == 0:  
 return b  
 return gcd(b%a, a)  
  
  
start = [0, 0]  
  
if int(end) % gcd(int(x\_capacity),int(y\_capacity)) == 0:  
 print (Production\_Rule\_Implementation(start, int(end),int( x\_capacity),int( y\_capacity)))  
  
  
else:  
 print ("No solution possible for this combination.")

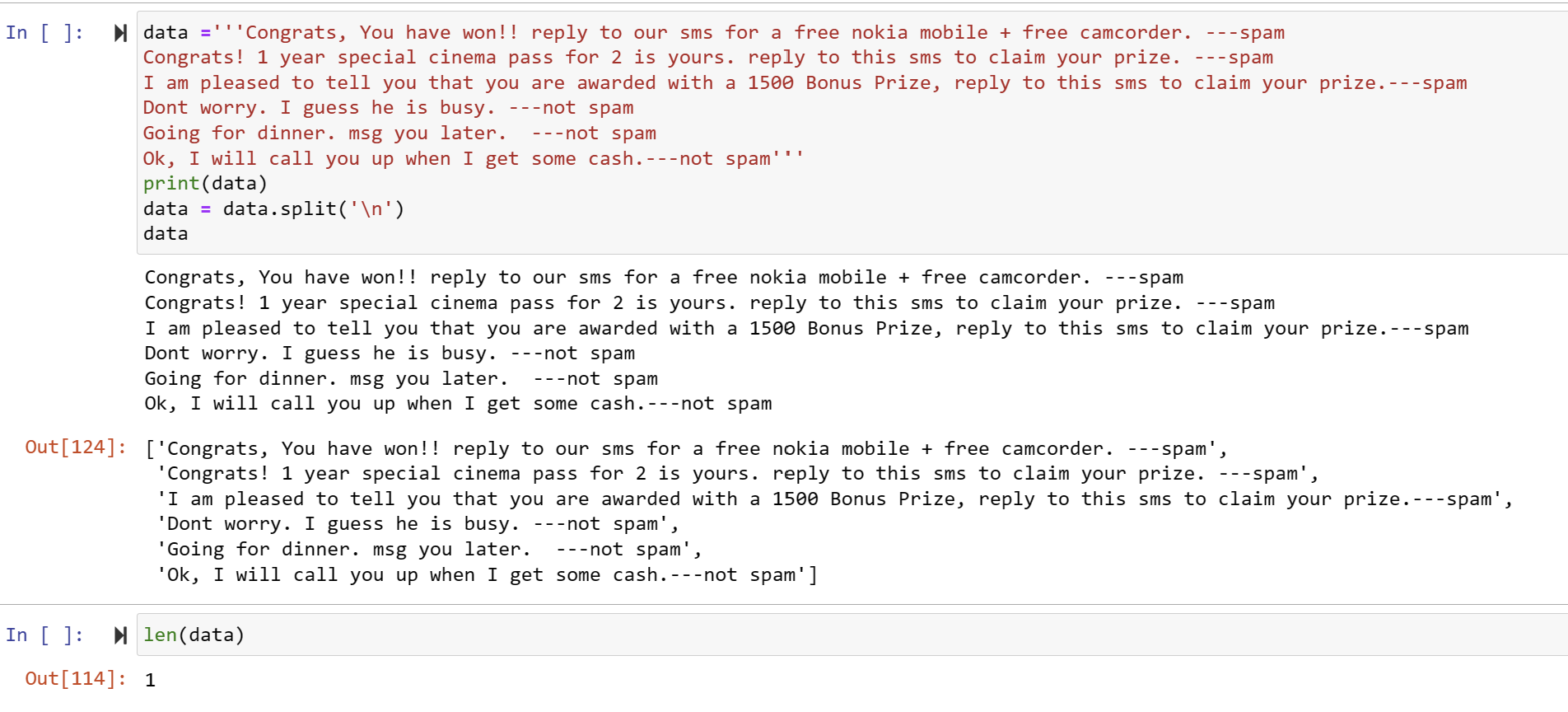
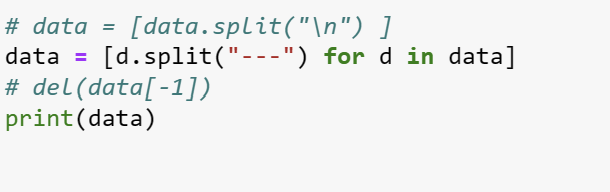
**OUTPUT:**

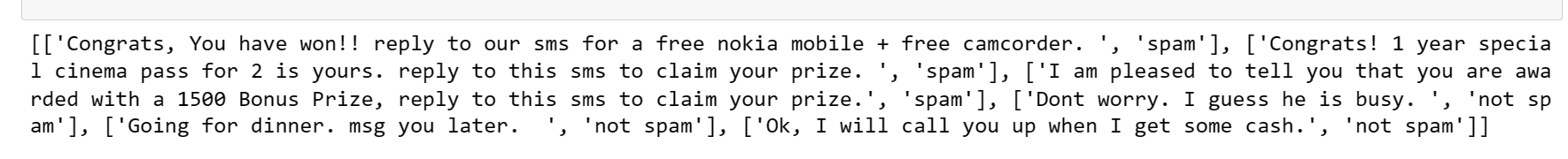
Solution for water jug problem  
Enter Jug 1 capacity:5  
Enter Jug 2 capacity:7  
Enter target volume:3  
Found!  
[[0, 0], [0, 7], [5, 2], [5, 7], [5, 0], [0, 5], [5, 5], [3, 7]]

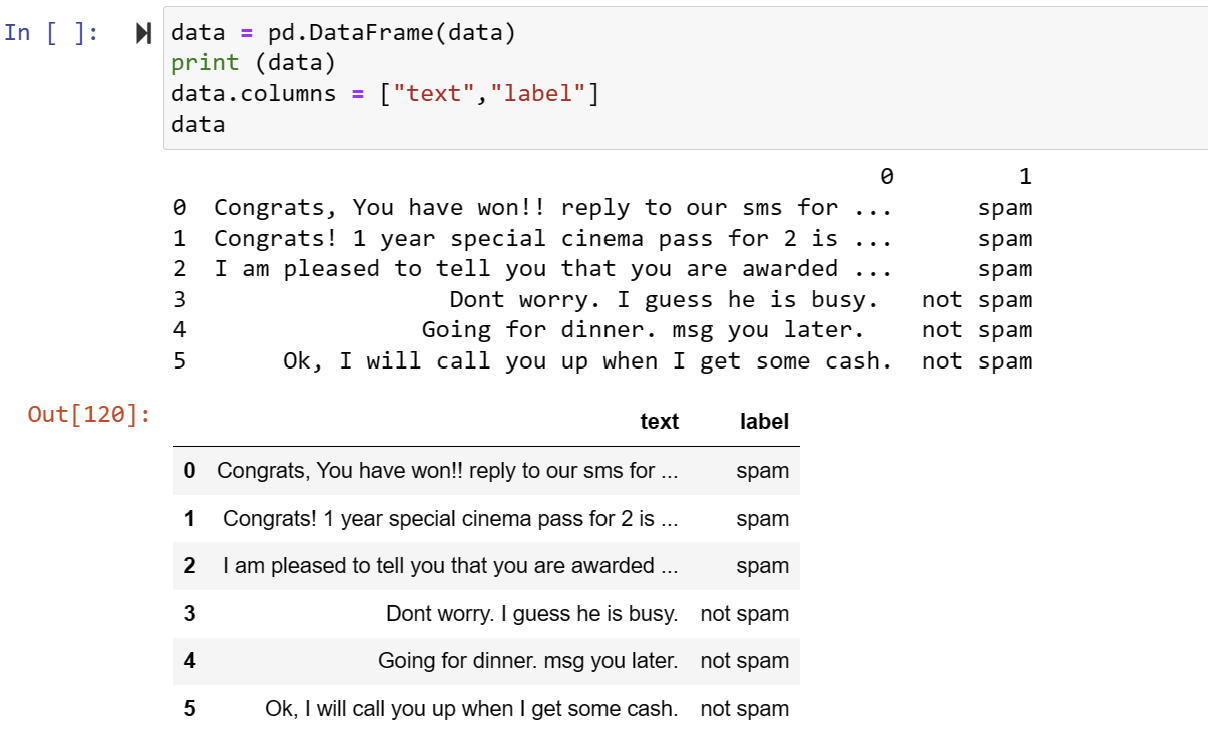
**Lab :5 ( Naïve Bayes Classifier)**

****

**CODE:**





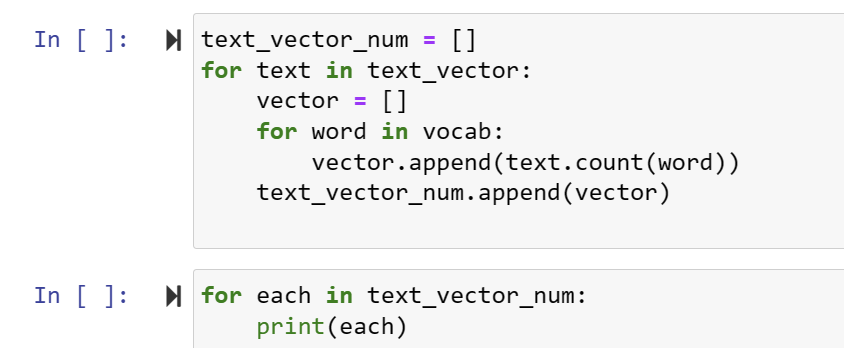
vocab = []  
text\_vector = []  
for each in data["text"]:  
 vocab.extend(each.lower().replace(".","").replace(",","").replace("+","").replace("!","").split())  
 text\_vector.append(each.lower().replace(".","").replace(",","").replace("+","").replace("!","").split())

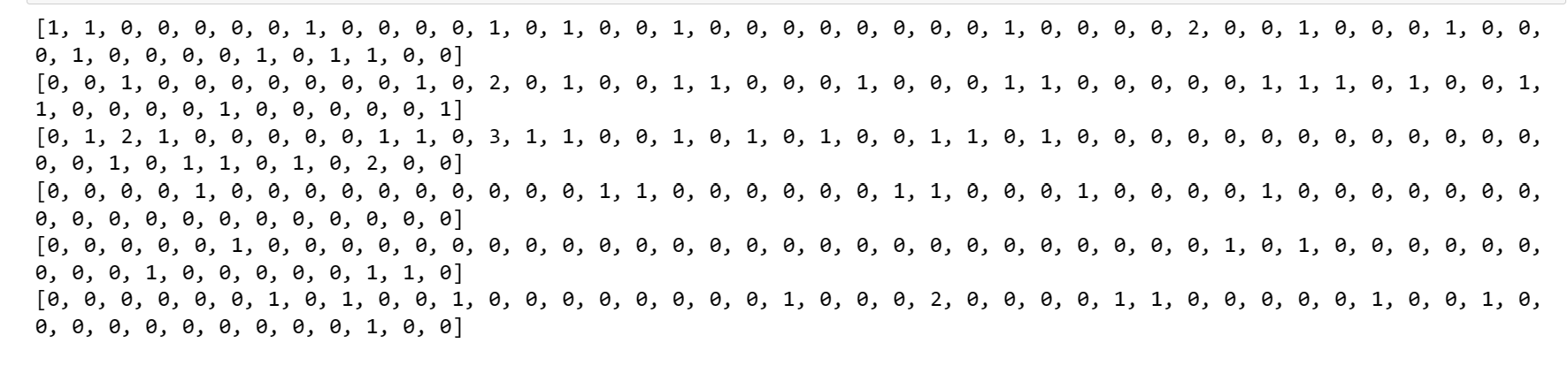
vocab = list(set(vocab))  
print(vocab)

['camcorder', 'a', 'prize', 'pleased', 'busy', 'going', 'up', 'have', 'ok', '1500', 'this', 'will', 'to', 'awarded', 'sms', 'he', 'guess', 'reply', 'cinema', 'am', 'cash', 'bonus', '1', 'dont', 'i', 'that', 'congrats', 'your', 'worry', 'get', 'some', 'free', 'msg', 'is', 'for', 'pass', 'call', 'special', 'won', 'when', 'yours', '2', 'our', 'with', 'later', 'tell', 'claim', 'nokia', 'are', 'mobile', 'you', 'dinner', 'year']

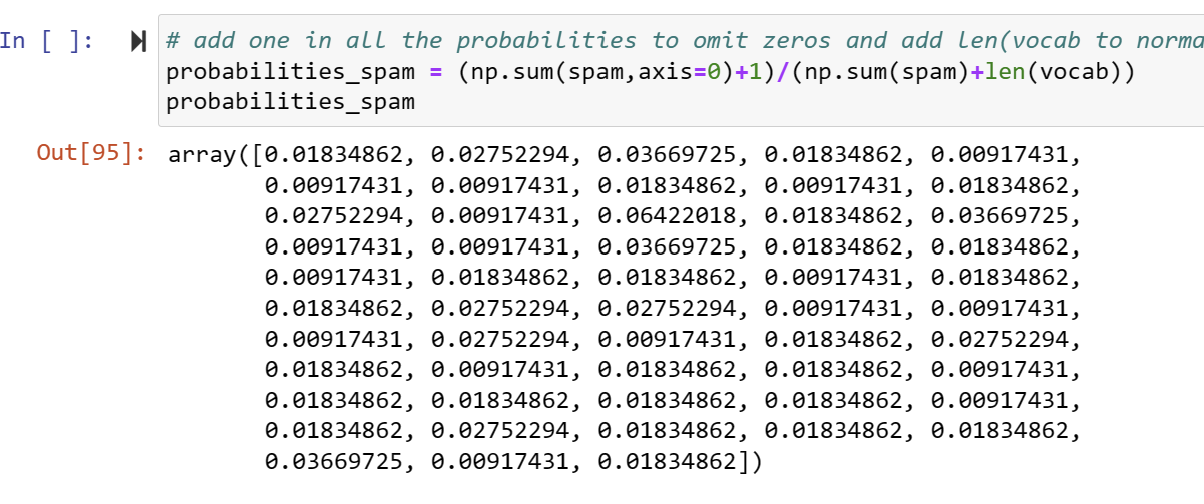
print(text\_vector)

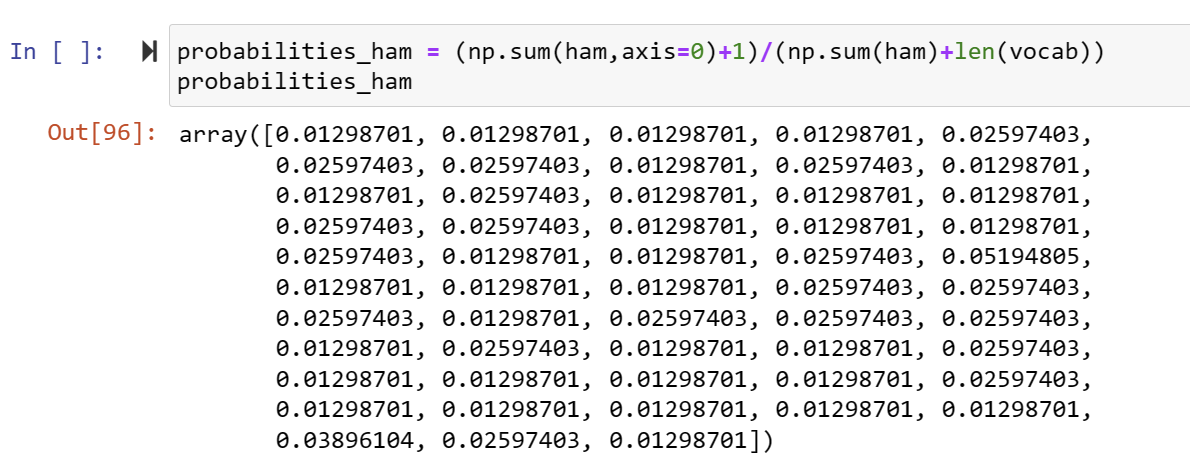
[['congrats', 'you', 'have', 'won', 'reply', 'to', 'our', 'sms', 'for', 'a', 'free', 'nokia', 'mobile', 'free', 'camcorder'], ['congrats', '1', 'year', 'special', 'cinema', 'pass', 'for', '2', 'is', 'yours', 'reply', 'to', 'this', 'sms', 'to', 'claim', 'your', 'prize'], ['i', 'am', 'pleased', 'to', 'tell', 'you', 'that', 'you', 'are', 'awarded', 'with', 'a', '1500', 'bonus', 'prize', 'reply', 'to', 'this', 'sms', 'to', 'claim', 'your', 'prize'], ['dont', 'worry', 'i', 'guess', 'he', 'is', 'busy'], ['going', 'for', 'dinner', 'msg', 'you', 'later'], ['ok', 'i', 'will', 'call', 'you', 'up', 'when', 'i', 'get', 'some', 'cash']]

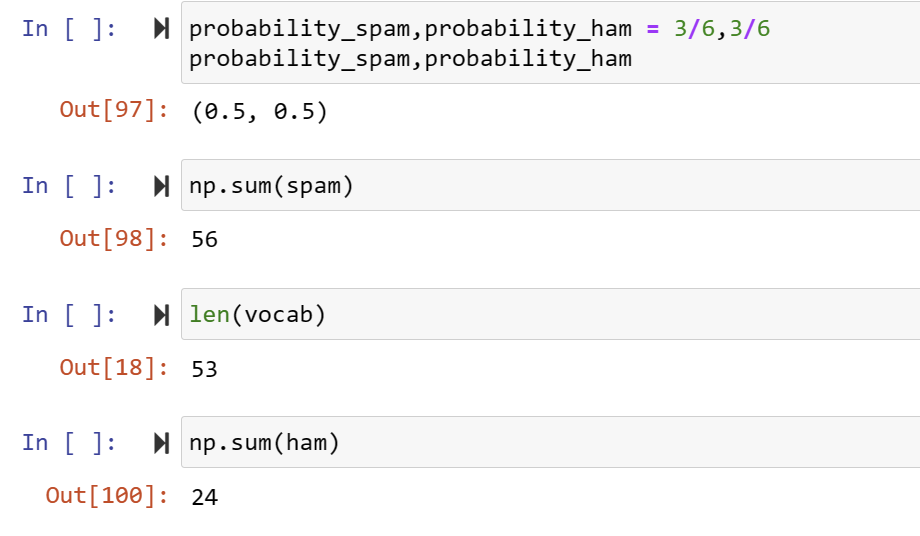




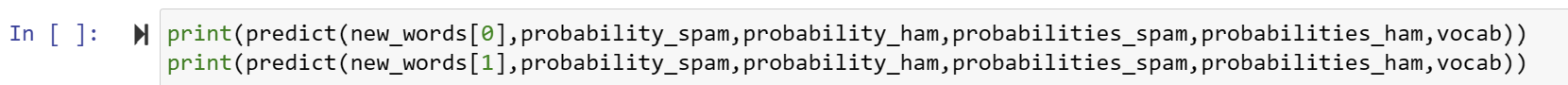
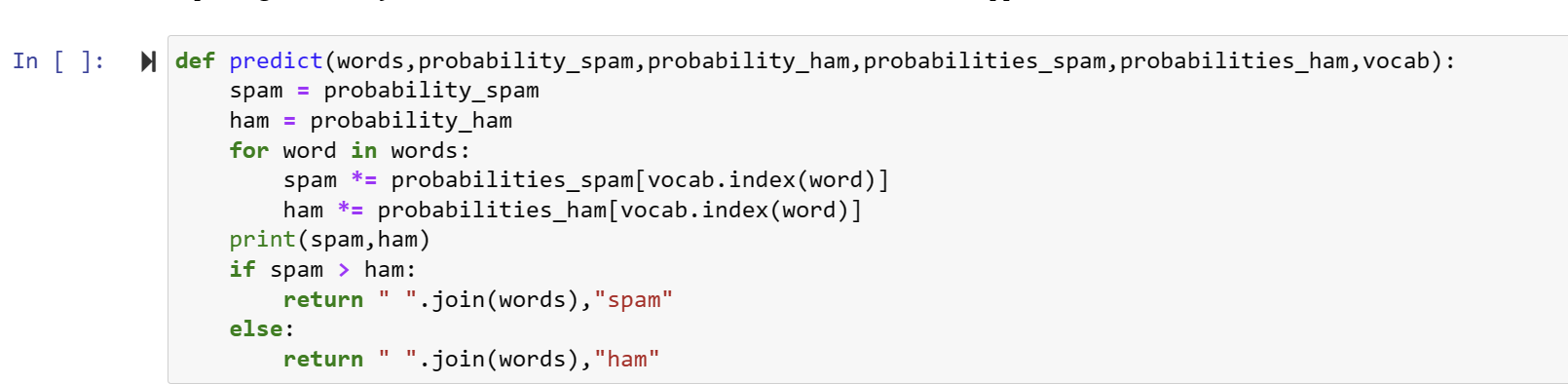










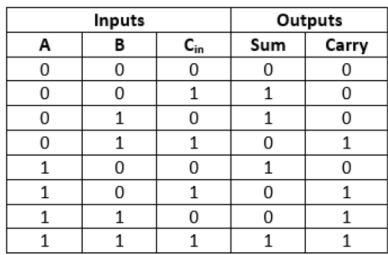


OUTPUT:

8.029860474764295e-16 3.107459112648254e-13  
('i am busy i will msg you later', 'ham')  
2.3631879377231325e-12 9.346654362262329e-14  
('congrats you are awarded a free mobile', 'spam')

**LAB :6 (Neural Network)**

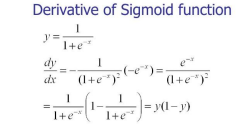
For this lab work, Student need to make a 2 layered artificial neural network model to  replace a full adder circuit.



Sigmoid should be used as activation function.

https://lh7-us.googleusercontent.com/Lm1HfQL-Ob2mTXro7moJ7pzAS2rggP7BkwsPRpegz97M_zAoejro1pAQD2dedtgWZVRtIawtZzyQk9yK8YjpLsc4LBKVFkcrCvQRCPgq494xeC1UTu55esahlrA7L-8GhKDyDaeEYZI_04HRE0ZP78Y

Use backpropagation as the learning method for the model.



**CODE:**

import numpy as np  
X = np.array([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1] ])  
y = np.array([[0,1,1,0,1,0,0,1]]).T  
z = np.array([[0,0,0,1,0,1,1,1]]).T  
syn0 = 2\*np.random.random((3,4)) - 1  
print(syn0)  
syn1 = 2\*np.random.random((4,1)) - 1  
print(syn1)  
syn3 = 2\*np.random.random((3,4)) - 1  
print(syn0)  
syn4 = 2\*np.random.random((4,1)) - 1  
print(syn1)  
#Neural network with Z variable as target  
for j in range(60000):  
 #feed forward with sigmoid activation for first layer  
 l1 = 1/(1+np.exp(-(np.dot(X,syn0))))  
 #feed forward with sigmoid activation for first layer  
 l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))  
 l2\_delta = (y - l2)\*(l2\*(1-l2))  
 l1\_delta = l2\_delta.dot(syn1.T) \* (l1 \* (1-l1))  
 syn1 += l1.T.dot(l2\_delta)  
 syn0 += X.T.dot(l1\_delta)  
# Neural network with Z variable as target  
for k in range(60000):  
 #feed forward with sigmoid activation for first layer  
 l3 = 1/(1+np.exp(-(np.dot(X,syn3))))  
 #feed forward with sigmoid activation for first layer  
 l4 = 1/(1+np.exp(-(np.dot(l3,syn4))))  
 l4\_delta = (z - l4)\*(l4\*(1-l4))  
 l3\_delta = l4\_delta.dot(syn4.T) \* (l3 \* (1-l3))  
 syn4 += l3.T.dot(l4\_delta)  
 syn3 += X.T.dot(l3\_delta)  
  
 #feed forward with sigmoid activation for first layer  
 l1 = 1/(1+np.exp(-(np.dot(X,syn0))))  
 #feed forward with sigmoid activation for Second layer  
 l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))  
 #Backpropagation of error for second layer of weights with derivative of Sigmoid  
 l2\_delta = (y - l2)\*(l2\*(1-l2))  
 #Backpropagation of error for first layer of weights with derivative of Sigmoid and the summation  
 l1\_delta = l2\_delta.dot(syn1.T) \* (l1 \* (1-l1)) #update of weights in both layers with respective errors from back propagation  
 syn1 += l1.T.dot(l2\_delta)  
 syn0 += X.T.dot(l1\_delta)

[[ 0.59748185 0.99880913 -0.86035202 0.11367176]  
 [ 0.2285702 0.58677096 0.94756785 -0.32049195]  
 [ 0.33771263 -0.66553232 -0.07699487 0.72668792]]  
[[ 0.26880237]  
 [-0.10724149]  
 [ 0.86003988]  
 [ 0.64647042]]  
[[ 0.59748185 0.99880913 -0.86035202 0.11367176]  
 [ 0.2285702 0.58677096 0.94756785 -0.32049195]  
 [ 0.33771263 -0.66553232 -0.07699487 0.72668792]]  
[[ 0.26880237]  
 [-0.10724149]  
 [ 0.86003988]  
 [ 0.64647042]]

Z=[]  
for a in range(8):  
 Z.append([y[a],z[a]])  
Z=np.array(Z)  
np.reshape(Z,(8,2))

**Output**

array([[0, 0],  
 [1, 0],  
 [1, 0],  
 [0, 1],  
 [1, 0],  
 [0, 1],  
 [0, 1],  
 [1, 1]])

c = []  
d = []  
for a in X:  
 l5 = 1/(1+np.exp(-(np.dot(a,syn0))))  
 l6 = 1/(1+np.exp(-(np.dot(l5,syn1))))  
 l7 = 1/(1+np.exp(-(np.dot(a,syn3))))  
 l8 = 1/(1+np.exp(-(np.dot(l7,syn4))))  
 c.append(round(list(l6)[0]))  
 d.append(round(list(l8)[0]))  
print(c)  
if c == list(y) and d==list(z):  
 print ('test successful')  
else:  
 print ('failure')

import numpy as np

import numpy as np  
X = np.array([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]])  
y = np.array([[0,1,1,0,1,0,0,1]]).T  
z = np.array([[0,0,0,1,0,1,1,1]]).T  
Z=[]  
for a in range(8):# making the labels  
 Z.append([y[a],z[a]])  
Z=np.reshape(Z,(8,2))  
  
syn0 = 2\*np.random.random((3,4)) - 1  
syn1 = 2\*np.random.random((4,2)) - 1  
  
#Neural network with Z variable as target  
for j in range(60000):  
 #feed forward with sigmoid activation for first layer  
 l1 = 1/(1+np.exp(-(np.dot(X,syn0))))  
 #feed forward with sigmoid activation for first layer  
 l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))  
 l2\_delta = (Z - l2)\*(l2\*(1-l2))  
 l1\_delta = l2\_delta.dot(syn1.T) \* (l1 \* (1-l1))  
 syn1 += l1.T.dot(l2\_delta)  
 syn0 += X.T.dot(l1\_delta)  
# Neural network with Z variable as target

l4 = 1/(1+np.exp(-(np.dot([1,1,0],syn0))))  
l5 = 1/(1+np.exp(-(np.dot(l4,syn1))))  
l5

import numpy as np  
X = np.array([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1] ])  
y = np.array([[0,1,1,0,1,0,0,1]]).T  
z = np.array([[0,0,0,1,0,1,1,1]]).T  
labels=[]  
for a in range(8):# making the labels  
 labels.append([y[a],z[a]])  
labels=np.reshape(labels,(8,2))  
# initializing random neurons  
syn0 = 2\*np.random.random((3,4)) - 1  
syn1 = 2\*np.random.random((4,2)) - 1  
  
#Neural network with Z variable as target  
for j in range(90000):  
 #feed forward with sigmoid activation for first layer  
 b = np.dot(X,syn0)  
 l1 = 1/(1+np.exp(-(b)))  
 #feed forward with sigmoid activation for first layer  
 c = np.dot(l1,syn1)  
 l2 = 1/(1+np.exp(-(c)))  
 sigmoid\_derivatives2 = (l2\*(1-l2))  
 l2\_delta = (labels - l2)\* sigmoid\_derivatives2  
 sigmoid\_derivatives1 =(l1 \* (1-l1))  
 l1\_delta = l2\_delta.dot(syn1.T) \* sigmoid\_derivatives1  
 syn1 += l1.T.dot(l2\_delta)  
 syn0 += X.T.dot(l1\_delta)