M.Sc C.S - II
SEM IV
Journal

Roll No.	054
Name	Raval Palak Rajanikant
Subject	Artificial Intelligence
Subject Code	TCSCPSCSP401

Thakur Educational Trust's (Regd.)

THAKUR COLLEGE OF SCIENCE & COMMERCE

AUTONOMOUS COLLEGE, PERMANENTLY AFFILIATED TO UNIVERSITY OF MUMBAI

NAAC Accredited Grade 'A' (3rd Cycle) & ISO 9001: 2015 (Certified)

Best College Award by University of Mumbai for the Year 2018-2019

CELEBRATING 25 YEARS OF GLORY

CERTIFICATE

This is here to certify that Mr. **Raval Palak Rajanikant** Seat Number **054** of M.Sc. I Computer Science, has satisfactorily completed the required number of experiments prescribed by the **THAKUR COLLEGE OF SCIENCE & COMMERCE AUTONOMOUS COLLEGE, PERMANENTLY AFFILIATED TO UNIVERSITY OF MUMBAI** during the academic year 2021 – 2022.

Date:

Place: Mumbai

Teacher In-Charge Head of Department

External Examiner

INDEX

Practical 1

Aim: *Implement Breadth first search algorithm for Romanian map problem.*

Theory:

Romanian Map problem

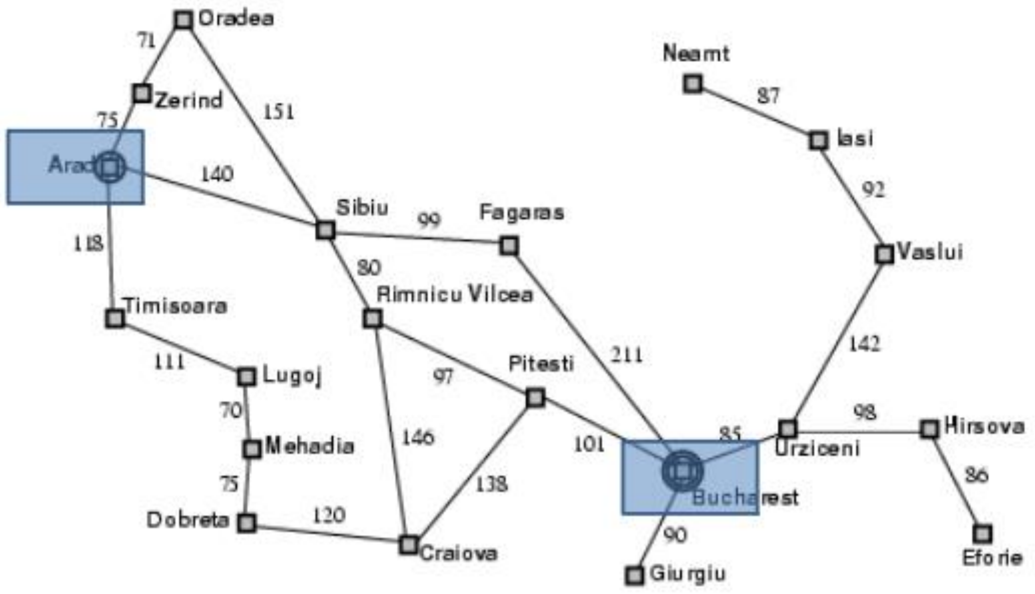
• In the state space view of the world, finding a solution is finding a path through the state space.

• When we (as humans) solve a problem like the 8-puzzle we have some idea of what constitutes the next best move.

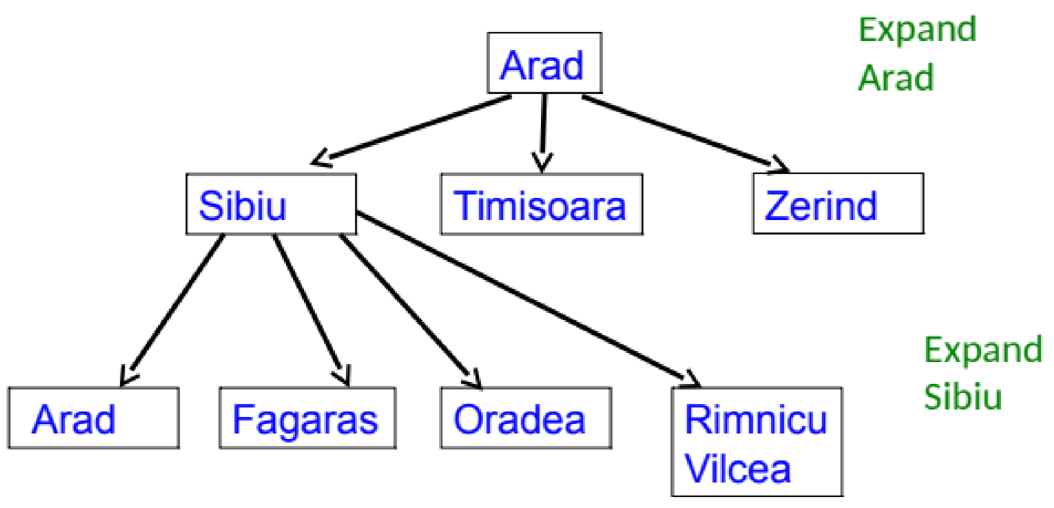
• It is hard to program this kind of approach.

• Instead, we start by programming the kind of repetitive task that computers are good at.

• A brute force approach to problem solving involves exhaustively searching through the space of all possible action sequences to find one that achieves the goal.



The Search Tree



Search strategy: how do we choose which node to expand?

Search Tree Exploration

• The tree is built by taking the initial state and identifying the states that can be obtained by a single application of the operators/actions available.

• These new states become the children of the initial state in the tree.

• These new states are then examined to see if they are the goal state.

• If not, the process is repeated on the new states.

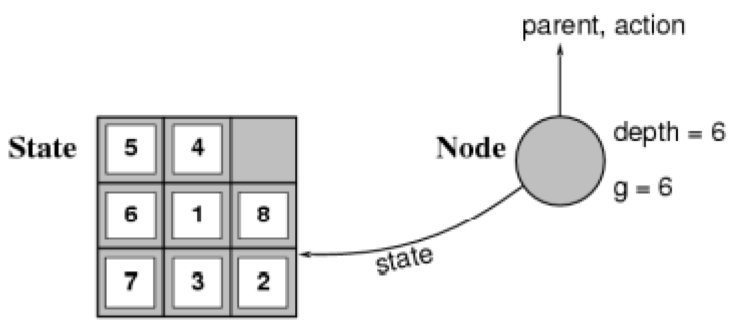
• We can formalise this description by giving an algorithm for it.

• We have different algorithms for different choices of nodes to expand.

Implementation: States vs. Nodes

• A state is a (representation of) a physical configuration.

• A node is a data structure constituting part of a search tree that includes state, parent node, action, path cost g(x), depth.



Expanding the tree creates new nodes, filling in the various fields and creating the corresponding states.

General Algorithm for Search

```
agenda = [initial state];
while agenda not empty do
pick node from agenda;
new nodes = apply operations to state;
if goal state in new nodes then
return solution;
else add new nodes to agenda;
```

• *Question: How to pick states for expansion?*

• *Two obvious strategies:*

  – *depth first search;*

  – ***breadth first search***

Breadth First Search

• Start by expanding initial state - gives tree of depth 1.

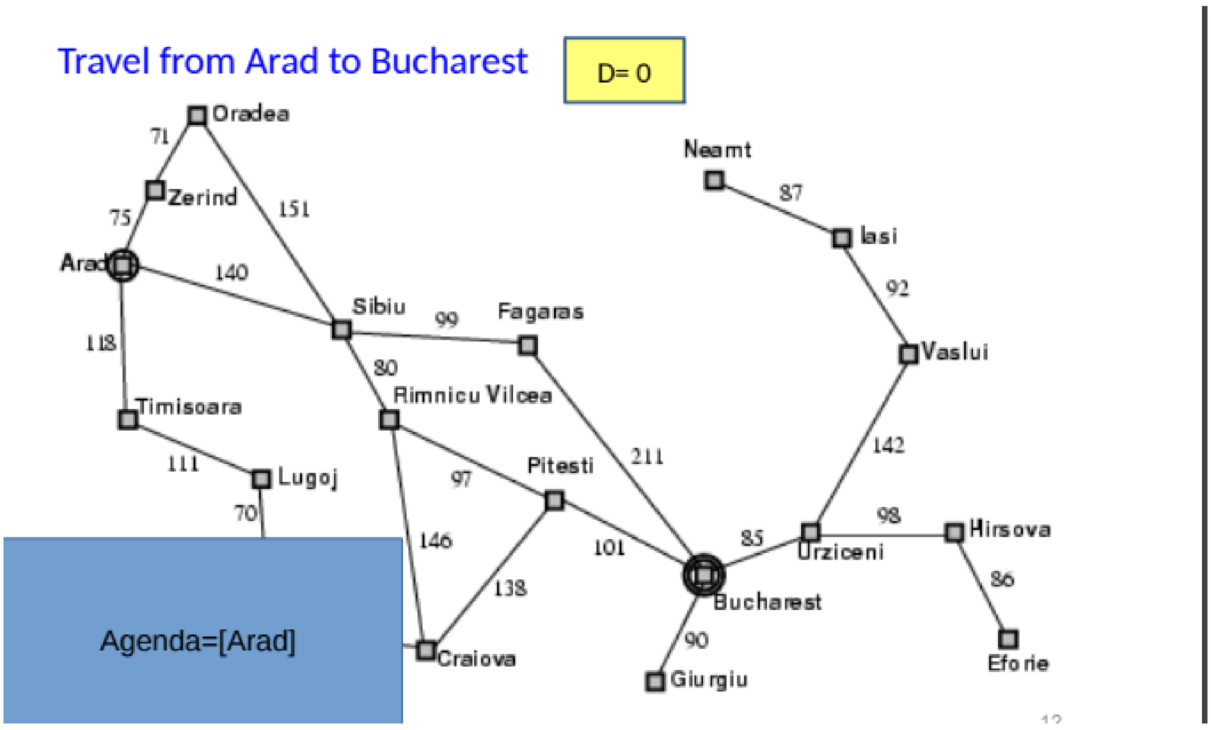
• Then expand all nodes that resulted from previous step gives tree of depth 2.

• Then expand all nodes that resulted from previous step, and so on.

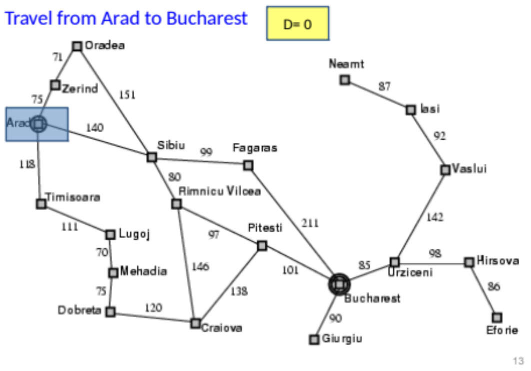
• Expand nodes all at depth n before going to level n + 1.

General Breadth First Search

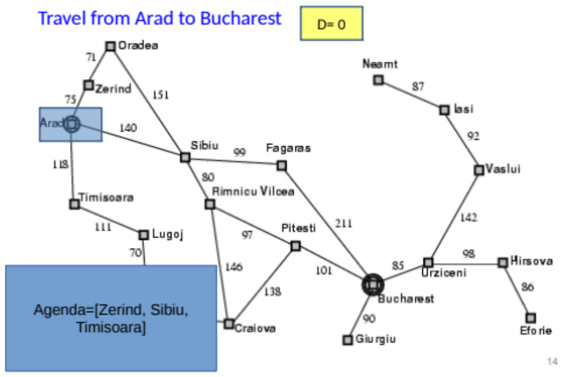
```
/* Breadth first search */
agenda = [initial state];
while agenda not empty do
 pick node from front of agenda;
 new nodes = apply operations to state;
 if goal state in new nodes then
return solution;
else APPEND new nodes to END of agenda
```



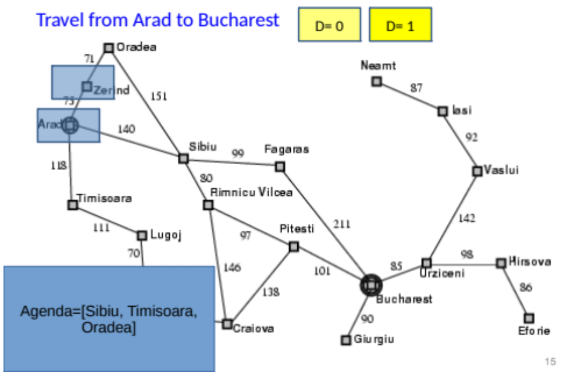
Example: Romania BFS



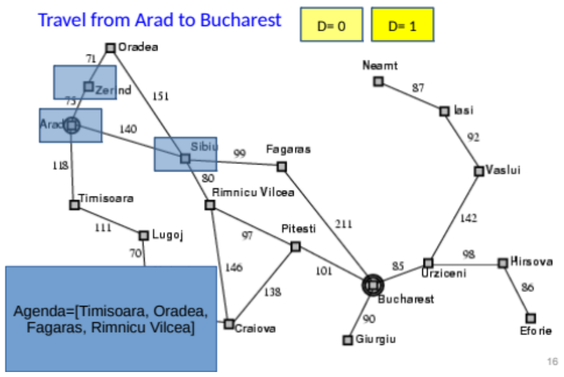
Example: Romania BFS

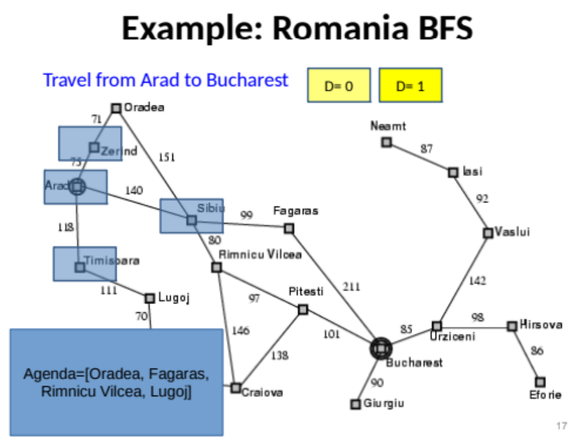

Travel from Arad to Bucharest
D= 0
Agenda=[Zerind, Sibiu, Timisoara]

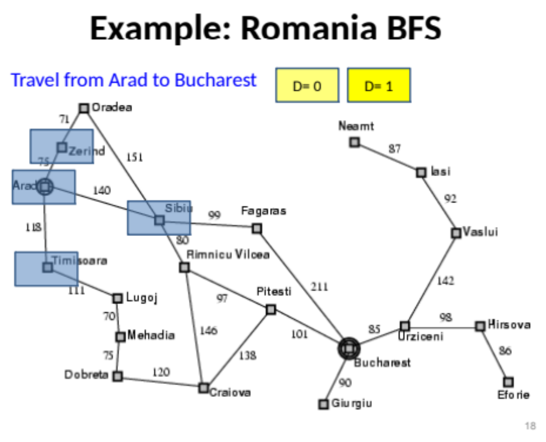
Example: Romania BFS

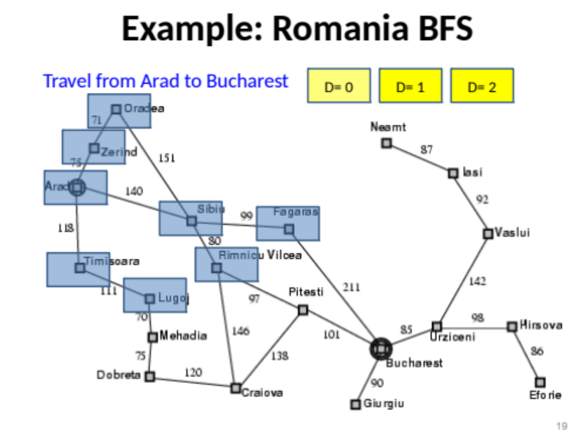

Travel from Arad to Bucharest
D= 0
D= 1
Agenda=[Sibiu, Timisoara, Oradea]

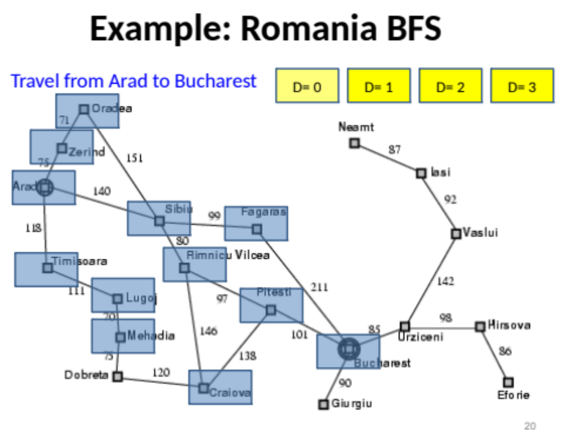
Example: Romania BFS


Travel from Arad to Bucharest
D= 0
D= 1
Agenda=[Timisoara, Oradea, Fagaras, Rimnicu Vilcea]


Example: Romania BFS
Travel from Arad to Bucharest
D= 0
D= 1
Agenda=[Oradea, Fagaras, Rimnicu Vilcea, Lugoj]
17







Properties of Breadth First Search

• *Advantage: guaranteed to reach a solution if one exists.*

• *Finds the shortest (cheapest) solution in terms of the number of operations applied to reach a solution.*

• *Disadvantage: time taken to reach solution.*

– *Let b be branching factor - average number of operations that may be performed from any level.*

– *If solution occurs at depth d, then we will look at*

$b + b^2 + b^3 + \cdots + b^d$

nodes before reaching solution exponential.

– *The memory requirement is* b^d

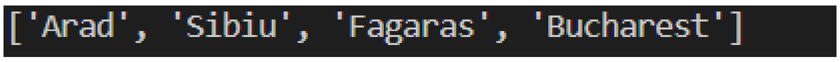
Code

```
from queue import Queue
```

```
romaniaMap = {
    'Arad': ['Sibiu', 'Zerind', 'Timisoara'],
    'Zerind': ['Arad', 'Oradea'],
    'Oradea': ['Zerind', 'Sibiu'],
    'Sibiu': ['Arad', 'Oradea', 'Fagaras', 'Rimnicu'],
    'Timisoara': ['Arad', 'Lugoj'],
    'Lugoj': ['Timisoara', 'Mehadia'],
    'Mehadia': ['Lugoj', 'Drobeta'],
    'Drobeta': ['Mehadia', 'Craiova'],
    'Craiova': ['Drobeta', 'Rimnicu', 'Pitesti'],
    'Rimnicu': ['Sibiu', 'Craiova', 'Pitesti'],
    'Fagaras': ['Sibiu', 'Bucharest'],
    'Pitesti': ['Rimnicu', 'Craiova', 'Bucharest'],
    'Bucharest': ['Fagaras', 'Pitesti', 'Giurgiu', 'Urziceni'],
    'Giurgiu': ['Bucharest'],
    'Urziceni': ['Bucharest', 'Vaslui', 'Hirsova'],
    'Hirsova': ['Urziceni', 'Eforie'],
    'Eforie': ['Hirsova'],
    'Vaslui': ['Iasi', 'Urziceni'],
    'Iasi': ['Vaslui', 'Neamt'],
    'Neamt': ['Iasi']
}

def bfs(startingNode, destinationNode):
    # For keeping track of what we have visited
    visited = {}
    # keep track of distance
    distance = {}
    # parent node of specific graph
    parent = {}

    bfs_traversal_output = []
    # BFS is queue based so using 'Queue' from python built-in
    queue = Queue()

    # travelling the cities in map
    for city in romaniaMap.keys():
        # since intially no city is visited so there will be nothing in visited list
        visited[city] = False
        parent[city] = None
        distance[city] = -1

    # starting from 'Arad'
    startingCity = startingNode
    visited[startingCity] = True
    distance[startingCity] = 0
    queue.put(startingCity)

    while not queue.empty():
        u = queue.get()     # first element of the queue, here it will be 'arad'
        bfs_traversal_output.append(u)

        # explore the adjust cities adj to 'arad'
        for v in romaniaMap[u]:
            if not visited[v]:
                visited[v] = True
                parent[v] = u
                distance[v] = distance[u] + 1
                queue.put(v)

        # reaching our destination city i.e 'bucharest'
    g = destinationNode
    path = []
    while g is not None:
        path.append(g)
        g = parent[g]

    path.reverse()
    # printing the path to our destination city
    print(path)

# Starting City & Destination City
bfs('Arad', 'Bucharest')
```

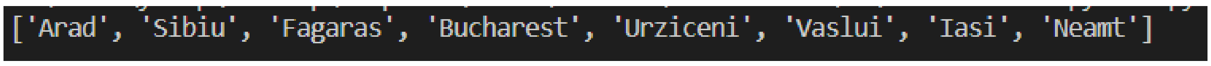
Output:

Arad to Bucharest



```
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
```

Arad to Neamt



```
['Arad', 'Sibiu', 'Fagaras', 'Bucharest', 'Urziceni', 'Vaslui', 'Iasi', 'Neamt']
```

Conclusion:

Implemented Breadth first search algorithm for Romanian map problem.

Practical 2

Aim: *Implement Iterative deep depth first search for Romanian map problem.*

Theory:

There are two common ways to traverse a graph, BFS and DFS. Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to following reasons.

1. **DFS** first traverses nodes going through one adjacent of root, then next adjacent. The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges).
2. **BFS** goes level by level, but requires more space. The space required by DFS is O(d) where d is depth of tree, but space required by BFS is O(n) where n is number of nodes in tree (Why? Note that the last level of tree can have around n/2 nodes and second last level n/4 nodes and in BFS we need to have every level one by one in queue).

IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).

How does IDDFS work?

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

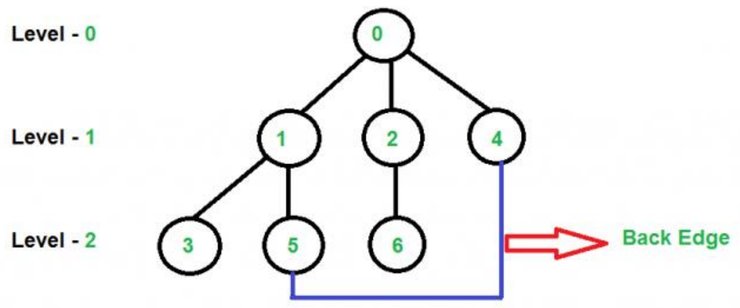
Algorithm:

```
// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
    for limit from 0 to max_depth
       if DLS(src, target, limit) == true
           return true
    return false

bool DLS(src, target, limit)
    if (src == target)
        return true;

    // If reached the maximum depth,
    // stop recursing.
    if (limit <= 0)
        return false;

    foreach adjacent i of src
        if DLS(i, target, limit?1)
            return true

    return false
```

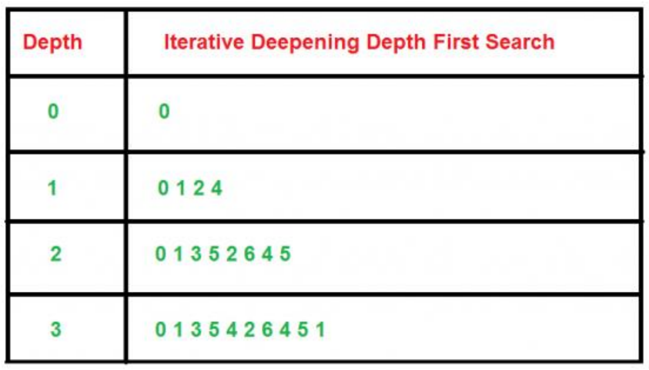
An important thing to note is, we visit top level nodes multiple times. The last (or max depth) level is visited once, second last level is visited twice, and so on. It may seem expensive, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level. So it does not matter much if the upper levels are visited multiple times.

Illustration: There can be two cases:

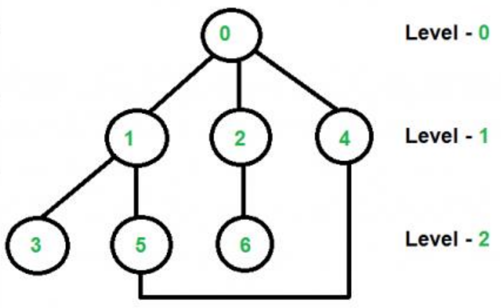
1. ***When the graph has no cycle:*** This case is simple. We can DFS multiple times with different height limits.
2. ***When the graph has cycles.*** This is interesting as there is no visited flag in IDDFS.



Although, at first sight, it may seem that since there are only 3 levels, so we might think that Iterative Deepening Depth First Search of level 3, 4, 5,...and so on will remain same. But, this is not the case. You can see that there is a cycle in the above graph, hence IDDFS will change for level-3,4,5..and so on.



Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Time Complexity: Suppose we have a tree having branching factor ‘b’ (number of children of each node), and its depth ‘d’, i.e., there are **b^d** nodes. In an iterative deepening search, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree, which is expanded d+1 times. So the total number of expansions in an iterative deepening search is-

$(d)b + (d-1)b^2 + ... + 3b^{d-2} + 2b^{d-1} + b^d$

That is,

Summation[$(d + 1 - i)\ b^i$], from $i = 0$ to $i = d$

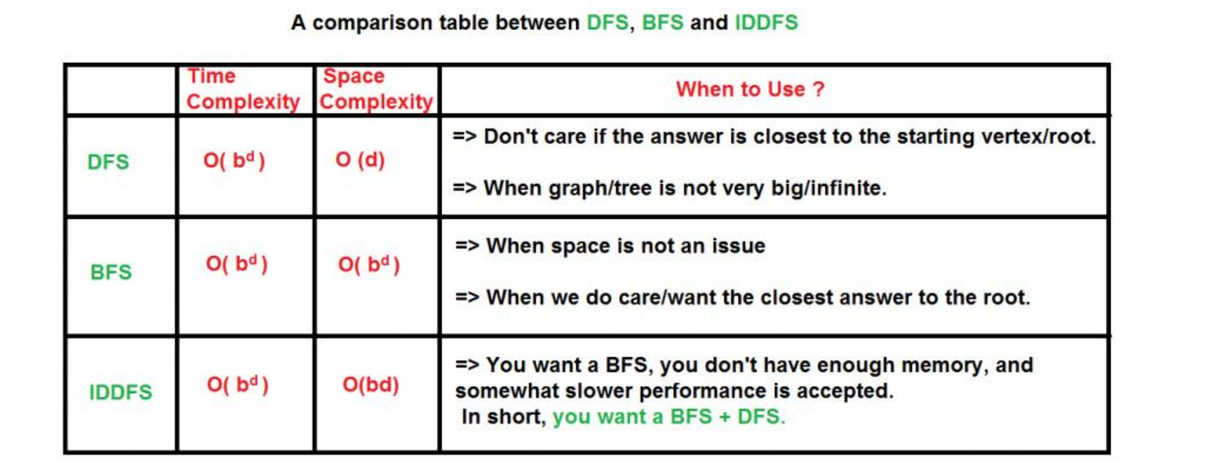
Which is same as $O(b^d)$

After evaluating the above expression, we find that asymptotically IDDFS takes the same time as that of DFS and BFS, but it is indeed slower than both of them as it has a higher constant factor in its time complexity expression. IDDFS is best suited for a complete infinite tree

Complexity:

Time: $O(b^d)$

Space: $O(d)$


A comparison table between DFS, BFS and IDDFS

	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	O (d)	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	O(bd)	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.

Code:

```
dict_graph = {}

# Read the data.txt file
with open('data.txt', 'r') as f:
    for l in f:
        city_a, city_b, p_cost = l.split()
        if city_a not in dict_graph:
            dict_graph[city_a] = {}
        dict_graph[city_a][city_b] = int(p_cost)
        if city_b not in dict_graph:
            dict_graph[city_b] = {}
        dict_graph[city_b][city_a] = int(p_cost)

# Iterative Deepening Search Method
def IterativeDeepening(graph, src, dst):
    level = 0
    count = 0
    stack = [(src, [src], 0)]
    visited = {src}
    while True:
        level += 1
        while stack:
            if count <= level:
                count = 0
                (node, path, cost) = stack.pop()
                for temp in graph[node].keys():
                    if temp == dst:
                        return path + [temp], cost + graph[node][temp]
                    else:
                        if temp not in visited:
                            visited.add(temp)
                            count += 1
                            stack.append((temp, path + [temp], cost + graph[node][temp]))
            else:
                q = stack
                visited_bfs = {src}
                while q:
                    (node, path, cost) = q.pop(0)
                    for temp in graph[node].keys():
                        if temp == dst:
                            return path + [temp], cost + graph[node][temp]
                        else:
                            if temp not in visited_bfs:
                                visited_bfs.add(temp)
                                q.append((temp, path + [temp], cost + graph[node][temp]))
                break

print(dict_graph)
print("-------------------------------------------------")
#src = raw_input("Enter the source:")
#dst = raw_input("Enter the Destination: ")
src = "Oradea"
dst = "Iasi"
print("for ID")
print (IterativeDeepening(dict_graph, src, dst))
```

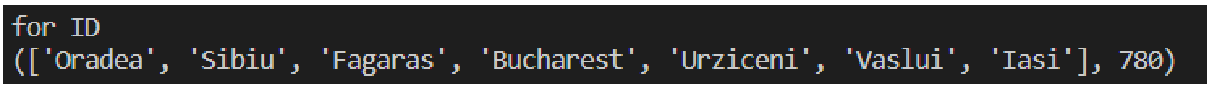
data.txt

```
Oradea Zerind 71
Oradea Sibiu 151
Zerind Arad 75
Arad Sibiu 140
Arad Timisoara 118
Timisoara Lugoj 111
Lugoj Mehadia 70
Mehadia Drobeta 75
Drobeta Craiova 120
Sibiu Rimnicu_Vilcea 80
Sibiu Fagaras 99
Rimnicu_Vilcea Piteshi 97
Rimnicu_Vilcea Craiova 146
Craiova Piteshi 138
Piteshi Bucharest 101
Fagaras Bucharest 211
Bucharest Giurgiu 90
Bucharest Urziceni 85
Urziceni Hirsova 98
Urziceni Vaslui 142
Hirsova Eforie 86
```

```
Vaslui Iasi 92
Neamt Iasi 87
```

Output:

Oradea to Iasi



```
for ID
(['Oradea', 'Sibiu', 'Fagaras', 'Bucharest', 'Urziceni', 'Vaslui', 'Iasi'], 780)
```

Conclusion:

Implemented Iterative deep depth first search for Romanian map problem.

Practical 3

***Aim**: Implement A* search algorithm for Romanian map problem.*

Theory:

What is A* Search Algorithm?

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it picks the node according to a value-‘f’ which is a parameter equal to the sum of two other parameters – ‘g’ and ‘h’. At each step it picks the node/cell having the lowest ‘f’, and process that node/cell.

We define ‘g’ and ‘h’ as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in the later sections.

A* Search Algorithm

```
1. Initialize the open list
2. Initialize the closed list put the starting node on the open  list (you can leave its f at zero)
3. while the open list is not empty
   a) find the node with the least f on the open list, call it "q"
   b) pop q off the open list
   c) generate q 8 successors and set their parents to q
   d) for each successor
       i) if successor is the goal, stop search
       ii) else, compute both g and h for successor
         successor.g = q.g + distance between successor and q
         successor.h = distance from goal to successor (This can be done using many ways, we will discuss
three heuristics- Manhattan, Diagonal and Euclidean  Heuristics)
         successor.f = successor.g + successor.h
       iii) if a node with the same position as  successor is in the OPEN list which has a lower f than
successor, skip this successor
       iV) if a node with the same position as successor  is in the CLOSED list which has  a lower f than
successor, skip this successor otherwise, add  the node to the open list
    end (for loop)
   e) push q on the closed list
   end (while loop)
```

So, suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristic.

Code:

```
import heapq

class priorityQueue:
   def __init__(self):
       self.cities = []

   def push(self, city, cost):
       heapq.heappush(self.cities, (cost, city))

   def pop(self):
       return heapq.heappop(self.cities)[1]

   def isEmpty(self):
       if (self.cities == []):
           return True
       else:
           return False

   def check(self):
       print(self.cities)

class ctNode:
   def __init__(self, city, distance):
       self.city = str(city)
       self.distance = str(distance)

romania = {}

def makedict():
   file = open("romania.txt", 'r')
   for string in file:
       line = string.split(',')
       ct1 = line[0]
       ct2 = line[1]
       dist = int(line[2])
       romania.setdefault(ct1, []).append(ctNode(ct2, dist))
       romania.setdefault(ct2, []).append(ctNode(ct1, dist))

def makehuristikdict():
   h = {}
   with open("romania_sld.txt", 'r') as file:
       for line in file:
           line = line.strip().split(",")
           node = line[0].strip()
           sld = int(line[1].strip())
           h[node] = sld
   return h

def heuristic(node, values):
   return values[node]

def astar(start, end):
   path = {}
   distance = {}
   q = priorityQueue()
   h = makehuristikdict()

   q.push(start, 0)
   distance[start] = 0
   path[start] = None
   expandedList = []

   while (q.isEmpty() == False):
       current = q.pop()
       expandedList.append(current)

       if (current == end):
           break

       for new in romania[current]:
           g_cost = distance[current] + int(new.distance)

           # print(new.city, new.distance, "now : " + str(distance[current]), g_cost)

           if (new.city not in distance or g_cost < distance[new.city]):
               distance[new.city] = g_cost
               f_cost = g_cost + heuristic(new.city, h)
               q.push(new.city, f_cost)
               path[new.city] = current

   printoutput(start, end, path, distance, expandedList)

def printoutput(start, end, path, distance, expandedlist):
   finalpath = []
   i = end
   while (path.get(i) != None):
       finalpath.append(i)
       i = path[i]
   finalpath.append(start)
   finalpath.reverse()
   print("Path From")
   print("\tArad => Bucharest")
   print("=======================================================")
   print("Exploreable cities : " + str(expandedlist))
   print("The number of possible cities: " + str(len(expandedlist)))
   print("=======================================================")
   print("The city that is passed the shortest distance: " + str(finalpath))
   print("Number of cities passed: " + str(len(finalpath)))
   print("Total Distance : " + str(distance[end]))

def main():
   src = "Arad"
   dst = "Bucharest"
   makedict()
   astar(src, dst)

if __name__ == "__main__":
   main()
```

romania_sld.txt

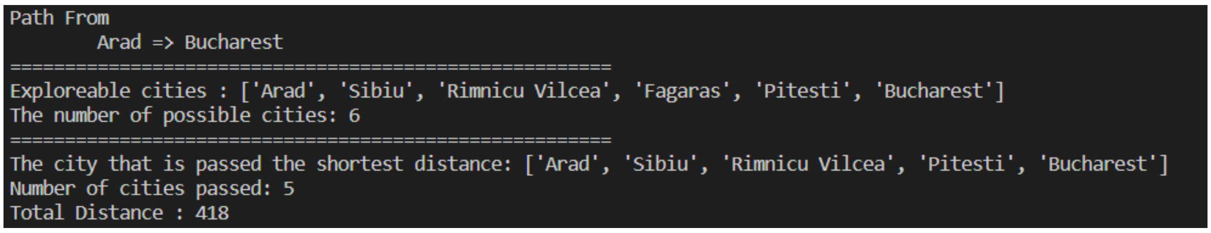
```
Arad, 366
Bucharest, 0
Craiova, 160
Dobreta, 242
Eforie, 161
Fagaras, 176
Giurgiu, 77
Hirsowa, 151
Lasi, 226
Lugoj, 244
Mehadia, 241
Neamt, 234
Oradea, 380
Pitesti, 100
Rimnicu Vilcea, 193
Sibiu, 253
Timisoara, 329
Urziceni, 80
Vaslui, 199
Zerind, 374
```

romania.txt

```
Arad,Zerind, 75
Arad,Sibiu, 140
Arad,Timisoara, 118
Zerind,Oradea, 71
Oradea,Sibiu, 151
```

```
Timisoara,Lugoj, 111
Sibiu,Fagaras, 99
Sibiu,Rimnicu Vilcea, 80
Lugoj,Mehadia, 70
Fagaras,Bucharest, 211
Rimnicu Vilcea,Pitesti, 97
Rimnicu Vilcea,Craiova, 146
Mehadia,Dobreta, 75
Bucharest,Pitesti, 101
Bucharest,Urziceni, 85
Bucharest,Giurglu, 90
Pitesti,Craiova, 138
Craiova,Dobreta, 120
Urziceni,Hirsova, 98
Urziceni,Vaslui, 142
Hirsova,Eforie, 86
Vaslui,Lasi, 92
Lasi,Neamt, 87
```

Output:



```
Path From
        Arad => Bucharest
=======================================================
Exploreable cities : ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Fagaras', 'Pitesti', 'Bucharest']
The number of possible cities: 6
=======================================================
The city that is passed the shortest distance: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']
Number of cities passed: 5
Total Distance : 418
```

Conclusion:

Implemented A search algorithm for Romanian map problem.*

Practical 4

Aim: *Implement recursive best-first search algorithm for Romanian map problem.*

Theory:

It is simple recursive algorithm that resembles the operation of standard best first search but uses only linear space. It is similar to recursive DFS and differs from Recursive DFS as follows,

It keeps track of the f value of the best alternative path available from any ancestor of the current node. Instead of continuing indefinitely down the current path.

Algorithm:

```
{
function RECURSIVE_BEST_FIRST_SEARCH (Problem)
   returns a solution, or failure
   RBFS (problem, MAKE_NODE (INITIAL_STATE problem), ¥)
function RBFS (Problem, Node, f-limit) returns a solution or failure and a new f-cost limit
   if GOAL_TEST [Problem] [State] then return node
Successors ← EXPAND (node, problem)
   if successors is empty then return failure, ¥
for each S in successors do
   f(s) ← max (g(s) + h(s), f[node])
repeat
   best ← the lowest f value node in successors
      if [best] > f-limit then return failure, f[best]
   alternative ← the second lowest f-value among successors
   result, f[best] ← BFS (Problem, best, min (f-limit, alternative)
      if result # failure then return result
}
```

Advantages

- *More efficient than IDA**
- *It is an optimal algorithm if h(n) is admissible*
- *Space complexity is O(bd).*

Disadvantages

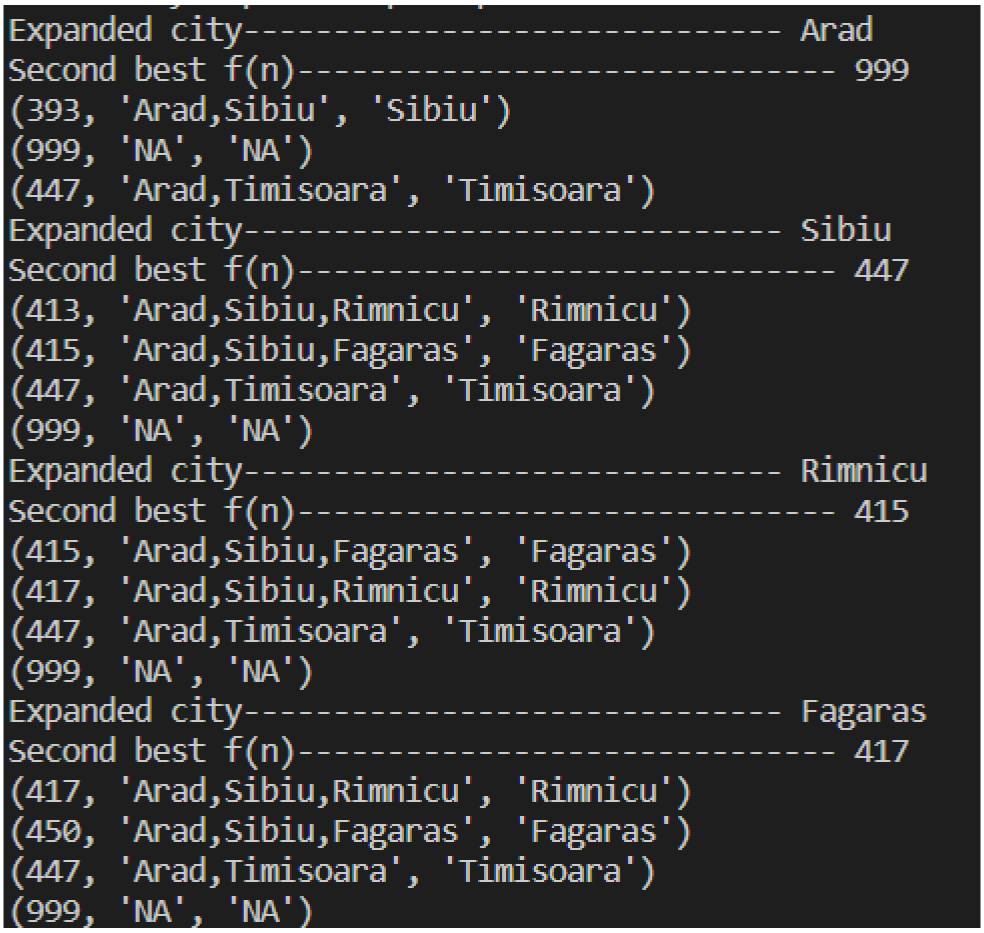
- *It suffers from excessive node regeneration.*
- *Its time complexity is difficult to characterize because it depends on the accuracy of h(n) and how often the best path changes as the nodes are expanded.*

Code:

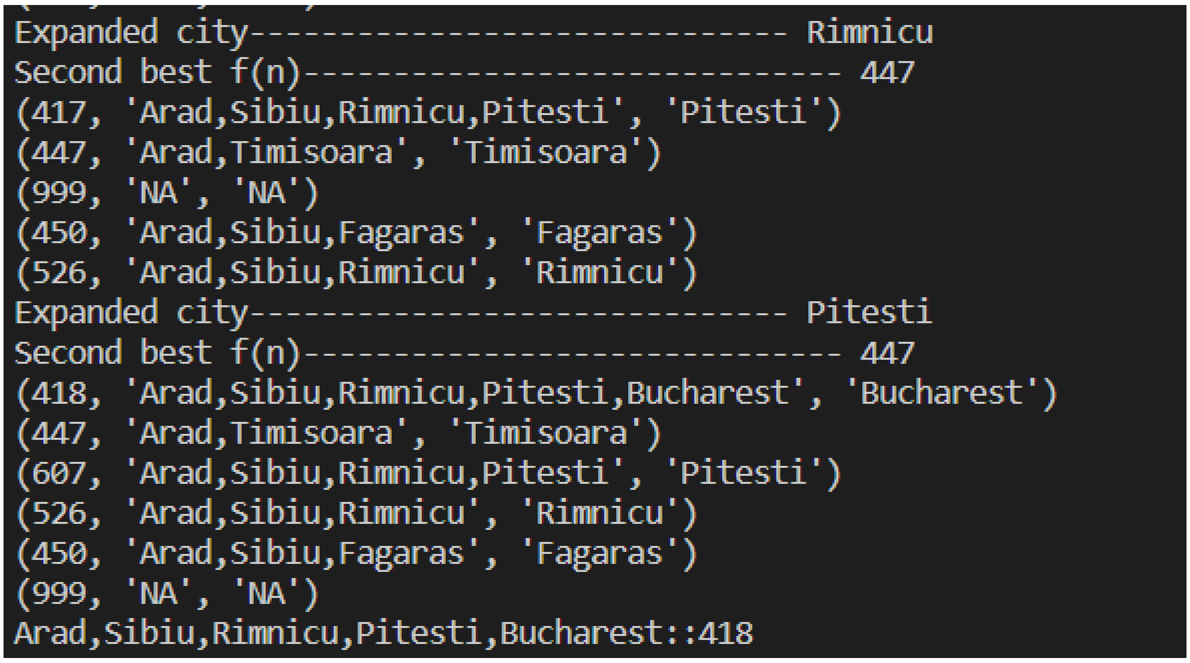
```
dict_hn={'Arad':336,'Bucharest':0,'Craiova':160,'Drobeta':242,'Eforie':161,
        'Fagaras':176,'Giurgiu':77,'Hirsova':151,'Iasi':226,'Lugoj':244,
        'Mehadia':241,'Neamt':234,'Oradea':380,'Pitesti':100,'Rimnicu':193,
        'Sibiu':253,'Timisoara':329,'Urziceni':80,'Vaslui':199,'Zerind':374}
```

```
dict_gn=dict(
Arad=dict(Zerind=75,Timisoara=118,Sibiu=140),
Bucharest=dict(Urziceni=85,Giurgiu=90,Pitesti=101,Fagaras=211),
Craiova=dict(Drobeta=120,Pitesti=138,Rimnicu=146),
Drobeta=dict(Mehadia=75,Craiova=120),
Eforie=dict(Hirsova=86),
Fagaras=dict(Sibiu=99,Bucharest=211),
Giurgiu=dict(Bucharest=90),
Hirsova=dict(Eforie=86,Urziceni=98),
Iasi=dict(Neamt=87,Vaslui=92),
Lugoj=dict(Mehadia=70,Timisoara=111),
Mehadia=dict(Lugoj=70,Drobeta=75),
Neamt=dict(Iasi=87),
Oradea=dict(Zerind=71,Sibiu=151),
Pitesti=dict(Rimnicu=97,Bucharest=101,Craiova=138),
Rimnicu=dict(Sibiu=80,Pitesti=97,Craiova=146),
Sibiu=dict(Rimnicu=80,Fagaras=99,Arad=140,Oradea=151),
Timisoara=dict(Lugoj=111,Arad=118),
Urziceni=dict(Bucharest=85,Hirsova=98,Vaslui=142),
Vaslui=dict(Iasi=92,Urziceni=142),
Zerind=dict(Oradea=71,Arad=75)
)
import queue as Q

start='Arad'
goal='Bucharest'
result=''

def get_fn(citystr):
    cities=citystr.split(',')
    hn=gn=0
    for ctr in range(0,len(cities)-1):
        gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
    hn=dict_hn[cities[len(cities)-1]]
    return(hn+gn)

def printout(cityq):
    for i in range(0,cityq.qsize()):
        print(cityq.queue[i])

def expand(cityq):
    global result
    tot,citystr,thiscity=cityq.get()
    nexttot=999
    if not cityq.empty():
        nexttot,nextcitystr,nextthiscity=cityq.queue[0]
    if thiscity==goal and tot<nexttot:
        result=citystr+'::'+str(tot)
        return
    print("Expanded city------------------------------",thiscity)
    print("Second best f(n)------------------------------",nexttot)
    tempq=Q.PriorityQueue()
    for cty in dict_gn[thiscity]:
            tempq.put((get_fn(citystr+','+cty),citystr+','+cty,cty))
    for ctr in range(1,3):
        ctrtot,ctrcitystr,ctrthiscity=tempq.get()
        if ctrtot<nexttot:
            cityq.put((ctrtot,ctrcitystr,ctrthiscity))
        else:
            cityq.put((ctrtot,citystr,thiscity))
            break
    printout(cityq)
    expand(cityq)
def main():
    cityq=Q.PriorityQueue()
    thiscity=start
    cityq.put((999,"NA","NA"))
    cityq.put((get_fn(start),start,thiscity))
    expand(cityq)
    print(result)
main()
```

Output:



```
Expanded city------------------------------ Arad
Second best f(n)------------------------------ 999
(393, 'Arad,Sibiu', 'Sibiu')
(999, 'NA', 'NA')
(447, 'Arad,Timisoara', 'Timisoara')
Expanded city------------------------------ Sibiu
Second best f(n)------------------------------ 447
(413, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(415, 'Arad,Sibiu,Fagaras', 'Fagaras')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
Expanded city------------------------------ Rimnicu
Second best f(n)------------------------------ 415
(415, 'Arad,Sibiu,Fagaras', 'Fagaras')
(417, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
Expanded city------------------------------ Fagaras
Second best f(n)------------------------------ 417
(417, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(450, 'Arad,Sibiu,Fagaras', 'Fagaras')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
```



```
Expanded city------------------------------ Rimnicu
Second best f(n)------------------------------ 447
(417, 'Arad,Sibiu,Rimnicu,Pitesti', 'Pitesti')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
(450, 'Arad,Sibiu,Fagaras', 'Fagaras')
(526, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
Expanded city------------------------------ Pitesti
Second best f(n)------------------------------ 447
(418, 'Arad,Sibiu,Rimnicu,Pitesti,Bucharest', 'Bucharest')
(447, 'Arad,Timisoara', 'Timisoara')
(607, 'Arad,Sibiu,Rimnicu,Pitesti', 'Pitesti')
(526, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(450, 'Arad,Sibiu,Fagaras', 'Fagaras')
(999, 'NA', 'NA')
Arad,Sibiu,Rimnicu,Pitesti,Bucharest::418
```

Conclusion:

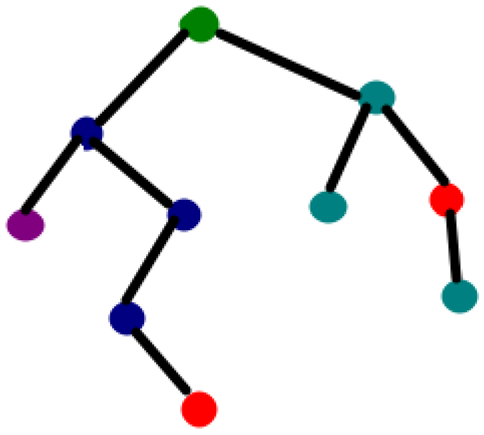
Implemented recursive best-first search algorithm for Romanian map problem.

Practical 5

Aim: *Implement decision tree learning algorithm for the restaurant waiting problem.*

Theory:

Decision Tree is one of the most powerful and popular algorithms. Decision-tree algorithm falls under the category of supervised learning algorithms. It works for both continuous as well as categorical output variables.

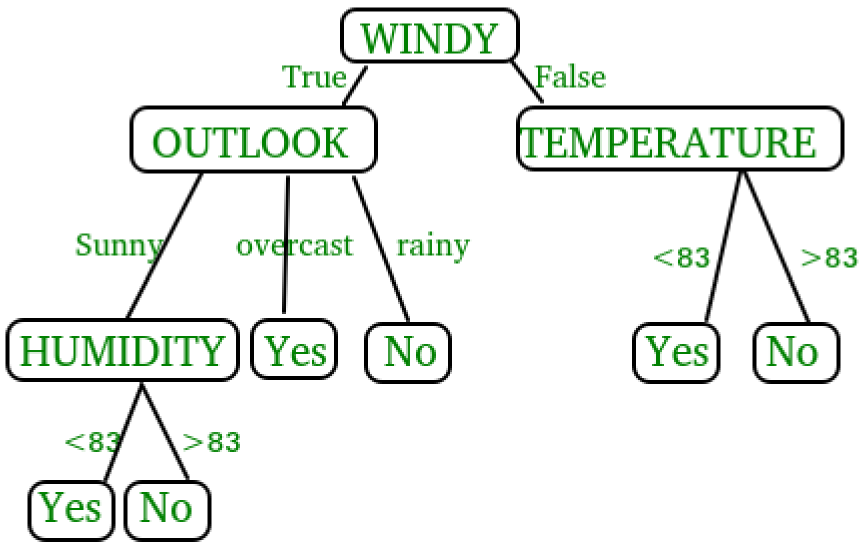


Assumptions we make while using Decision tree :

- *At the beginning, we consider the whole training set as the root.*
- *Attributes are assumed to be categorical for information gain and for gini index, attributes are assumed to be continuous.*
- *On the basis of attribute values records are distributed recursively.*
- *We use statistical methods for ordering attributes as root or internal node.*

Pseudocode :

```
1. Find the best attribute and place it on the root node of the tree.
2. Now, split the training set of the dataset into subsets. While making the subset
   make sure that each subset of training dataset should have the same value for
   an attribute.
3. Find leaf nodes in all branches by repeating 1 and 2 on each subset.
```



While implementing the decision tree we will go through the following two phases:

1. *Building Phase*
   - *Preprocess the dataset.*
   - *Split the dataset from train and test using Python sklearn package.*
   - *Train the classifier.*
2. *Operational Phase*
   - *Make predictions.*
   - *Calculate the accuracy.*

Data Import :

- *To import and manipulate the data we are using the pandas package provided in python.*
- *Here, we are using a URL which is directly fetching the dataset from the UCI site no need to download the dataset. When you try to run this code on your system make sure the system should have an active Internet connection.*
- *As the dataset is separated by ",” so we have to pass the sep parameter's value as ",”.*
- *Another thing is notice is that the dataset doesn't contain the header so we will pass the Header parameter's value as none. If we will not pass the header parameter then it will consider the first line of the dataset as the header.*

Data Slicing :

- *Before training the model we have to split the dataset into the training and testing dataset.*
- *To split the dataset for training and testing we are using the sklearn module train_test_split*
- *First of all we have to separate the target variable from the attributes in the dataset.*

```
X = balance_data.values[:, 1:5]
Y = balance_data.values[:,0]
```

- *Above are the lines from the code which separate the dataset. The variable X contains the attributes while the variable Y contains the target variable of the dataset.*
- *Next step is to split the dataset for training and testing purpose.*

```
X_train, X_test, y_train, y_test = train_test_split(
        X, Y, test_size = 0.3, random_state = 100)
```

- *Above line split the dataset for training and testing. As we are splitting the dataset in a ratio of 70:30 between training and testing so we are pass test_size parameter's value as 0.3.*
- *random_state variable is a pseudo-random number generator state used for random sampling.*

Terms used in code :

Gini index and information gain both of these methods are used to select from the n attributes of the dataset which attribute would be placed at the root node or the internal node.

Gini index:

$$\text{Gini Index } = 1 - \sum_j {}^2_{\text{j}}$$

- *Gini Index is a metric to measure how often a randomly chosen element would be incorrectly identified.*
- *It means an attribute with lower gini index should be preferred.*
- *Sklearn supports "gini" criteria for Gini Index and by default, it takes "gini" value.*

Entropy:

if a random variable x can take N different value , the i'value x_i with probability $p(x_{ii})$,we can associate the following entropy with x :

$$H(x) = -\sum_{i=1}^{N} p(x_i) \log_2 p(x_i)$$

- *Entropy is the measure of uncertainty of a random variable, it characterizes the impurity of an arbitrary collection of examples. The higher the entropy the more the information content.*

Information Gain

$Definition : SupposeSisasetofinstances,Aisanattribute,S_v$ is the subset of s with A = v and Values(A) is the set of all possible of A,then

- *The entropy typically changes when we use a node in a decision tree to partition the training instances into smaller subsets. Information gain is a measure of this change in entropy.*
- *Sklearn supports "entropy" criteria for Information Gain and if we want to use Information Gain method in sklearn then we have to mention it explicitly.*

Accuracy score

- *Accuracy score is used to calculate the accuracy of the trained classifier.*

Confusion Matrix

- *Confusion Matrix is used to understand the trained classifier behavior over the test dataset or validate dataset.*

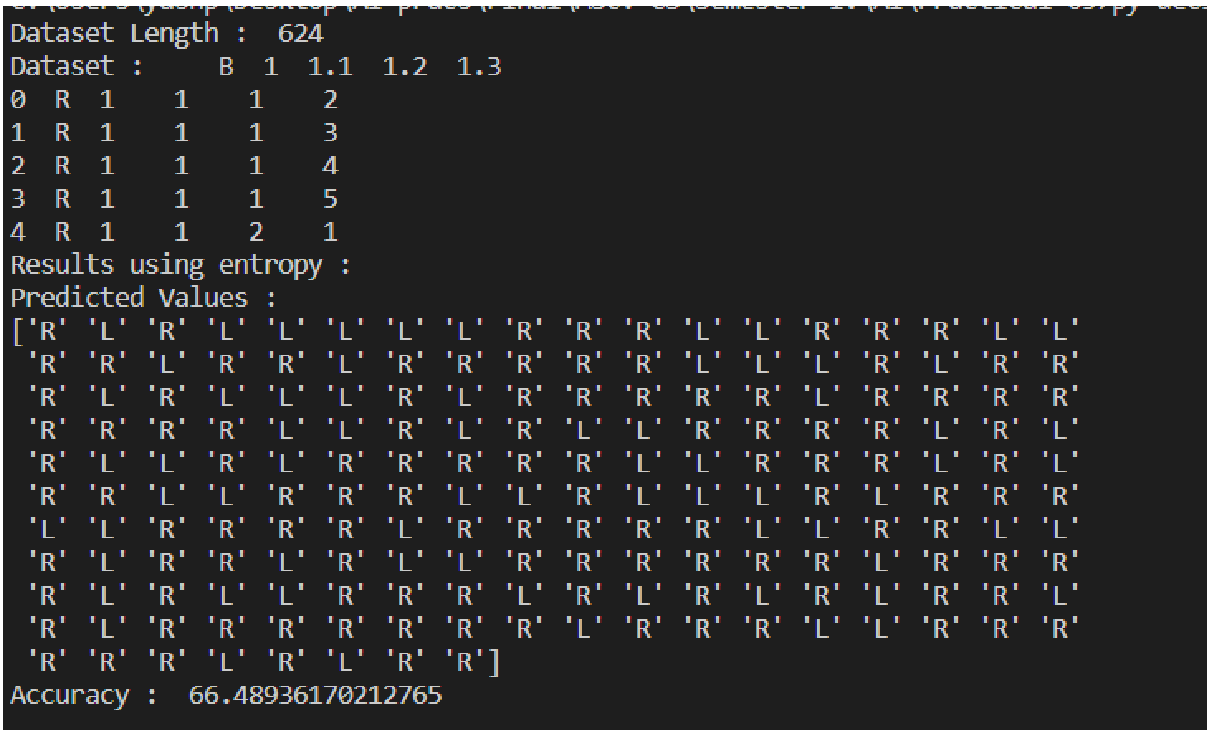
Code:

```
import numpy as np
import pandas as pd
import sklearn as sk
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

#func importing dataset
def importdata():
    balance_data=pd.read_csv("balance-scale.data")

    #print the dataset shape
    print("Dataset Length : ",len(balance_data))

    #printing the dataset observations
    print("Dataset : ",balance_data.head())
    return balance_data

#func to split the dataset
def splitdataset(balance_data):
    #seperating the target variable
    X=balance_data.values[:,1:5]
    Y=balance_data.values[:,0]

    #splitting the dataset into train and test
    X_train,X_test,y_train,y_test=train_test_split(X,Y,test_size=0.3,random_state=100)
    return X,Y,X_train,X_test,y_train,y_test

#function to perform training with entropy
def train_using_entropy(X_train,X_test,y_train,y_test):
    #decision tree with entropy
    clf_entropy=DecisionTreeClassifier(criterion="entropy",random_state=100,max_depth=3,min_sample
s_leaf=5)

    #performing training
    clf_entropy.fit(X_train,y_train)
    return clf_entropy

def prediction(X_test,clf_object):
    y_pred=clf_object.predict(X_test)
    print("Predicted Values : ")
    print(y_pred)
    return y_pred

def cal_accuracy(y_test,y_pred):
    print("Accuracy : ",accuracy_score(y_test,y_pred)*100)

def main():
    data=importdata()
    X,Y,X_train,X_test,y_train,y_test=splitdataset(data)

    clf_entropy=train_using_entropy(X_train,X_test,y_train,y_test)

    print("Results using entropy : ")
    y_pred_entropy=prediction(X_test,clf_entropy)
    cal_accuracy(y_test,y_pred_entropy)

main()
```

balance-sheet.data

```
B,1,1,1,1
R,1,1,1,2
R,1,1,1,3
R,1,1,1,4
R,1,1,1,5
R,1,1,2,1
R,1,1,2,2
.
.
.L,5,5,4,5
L,5,5,5,1
L,5,5,5,2
L,5,5,5,3
L,5,5,5,4
B,5,5,5,5
```

Output:



```
Dataset Length :  624
Dataset :     B  1  1.1  1.2  1.3
0  R  1    1    1    2
1  R  1    1    1    3
2  R  1    1    1    4
3  R  1    1    1    5
4  R  1    1    2    1
Results using entropy :
Predicted Values :
['R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'R' 'R' 'L' 'L'
 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R'
 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R'
 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'L'
 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'R'
 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'R' 'L' 'L'
 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L'
 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'R' 'R'
 'R' 'R' 'R' 'L' 'R' 'L' 'R' 'R']
Accuracy :  66.48936170212765
```

Conclusion:

Implemented decision tree learning algorithm for the restaurant waiting problem.

Practical 6

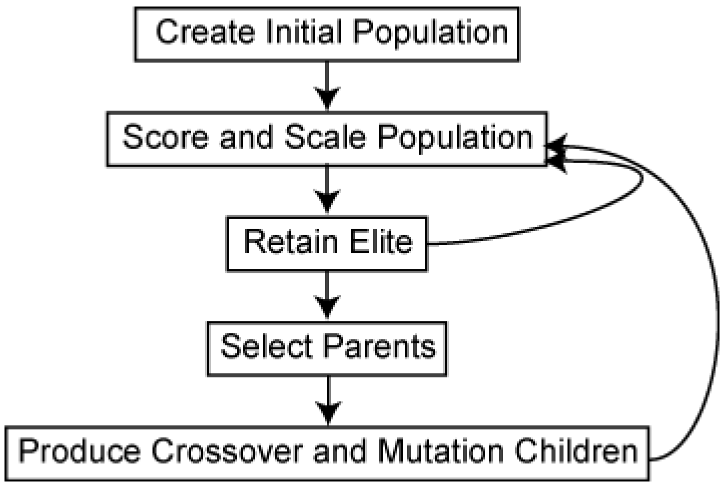
Aim: *Implement Genetic Algorithms for Staff Planning*.

Theory:

What Is the Genetic Algorithm?

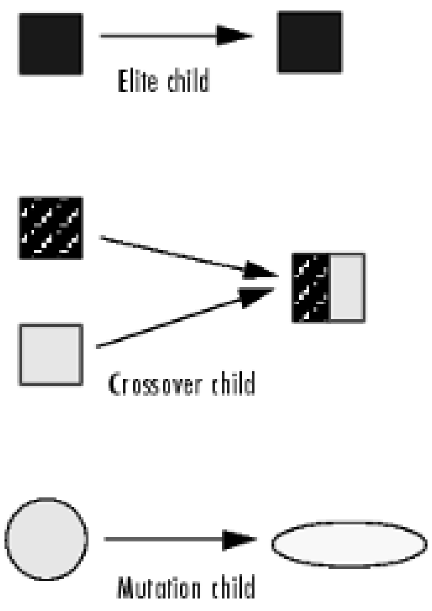
The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear. The genetic algorithm can address problems of *mixed integer programming*, where some components are restricted to be integer-valued.

This flow chart outlines the main algorithmic steps. For details, see How the Genetic Algorithm Works.


Create Initial Population
Score and Scale Population
Retain Elite
Select Parents
Produce Crossover and Mutation Children

The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- *Selection rules* select the individuals, called *parents*, that contribute to the population at the next generation. The selection is generally stochastic, and can depend on the individuals' scores.
- *Crossover rules* combine two parents to form children for the next generation.
- *Mutation rules* apply random changes to individual parents to form children.

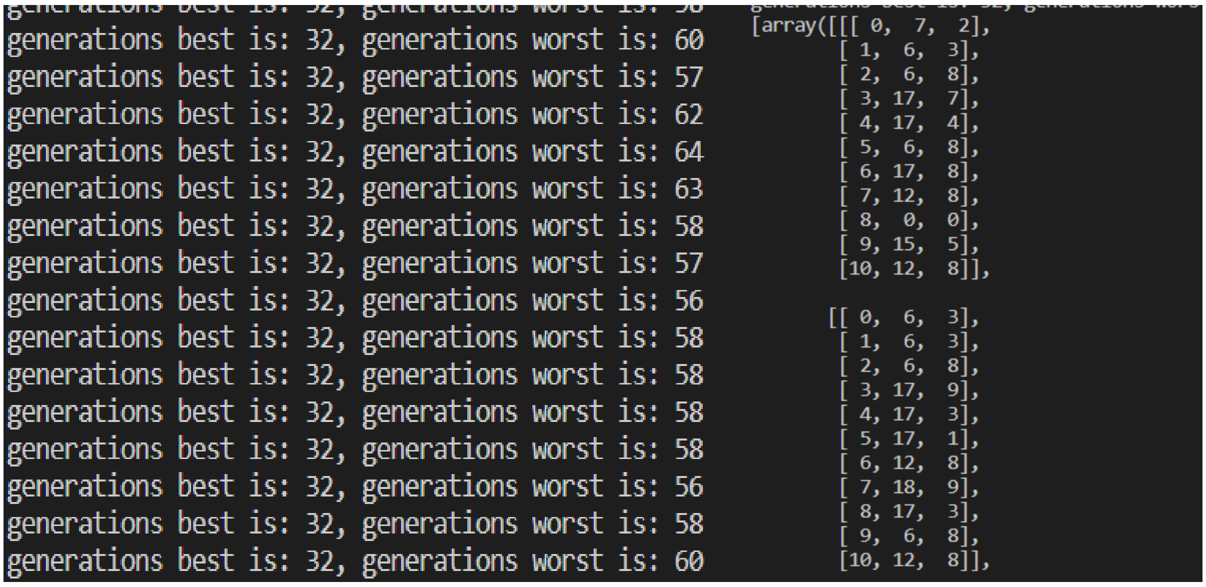


Code:

```
import numpy as np
import pandas as pd
staff_planning = [
   [[0, 0, 10],[1, 0, 10],[2, 0, 10],[3, 0, 10],[4, 0, 10],[5, 0, 10],[6, 0, 10],[7, 0, 10],[8, 0, 10],[9, 0, 10],[10,
0, 10]],
   [[0, 0, 10],[1, 0, 10],[2, 0, 10],[3, 0, 10],[4, 0, 10],[5, 0, 10],[6, 0, 10],[7, 0, 10],[8, 0, 10],[9, 0, 10],[10,
0, 10]],
   [[0, 0, 10],[1, 0, 10],[2, 0, 10],[3, 0, 10],[4, 0, 10],[5, 0, 10],[6, 0, 10],[7, 0, 10],[8, 0, 10],[9, 0, 10],[10,
0, 10]],
   [[0, 0, 10],[1, 0, 10],[2, 0, 10],[3, 0, 10],[4, 0, 10],[5, 0, 10],[6, 0, 10],[7, 0, 10],[8, 0, 10],[9, 0, 10],[10,
0, 10]],
   [[0, 0, 10],[1, 0, 10],[2, 0, 10],[3, 0, 10],[4, 0, 10],[5, 0, 10],[6, 0, 10],[7, 0, 10],[8, 0, 10],[9, 0, 10],[10,
0, 10]]
]

hourlystaff_needed = np.array([
   [0, 0, 0, 0, 0, 0, 4, 4, 4, 2, 2, 2, 6, 6, 2, 2, 2, 6, 6, 6, 2, 2, 2, 2],
   [0, 0, 0, 0, 0, 0, 4, 4, 4, 2, 2, 2, 6, 6, 2, 2, 2, 6, 6, 6, 2, 2, 2, 2],
   [0, 0, 0, 0, 0, 0, 4, 4, 4, 2, 2, 2, 6, 6, 2, 2, 2, 6, 6, 6, 2, 2, 2, 2],
   [0, 0, 0, 0, 0, 0, 4, 4, 4, 2, 2, 2, 6, 6, 2, 2, 2, 6, 6, 6, 2, 2, 2, 2],
   [0, 0, 0, 0, 0, 0, 4, 4, 4, 2, 2, 2, 6, 6, 2, 2, 2, 6, 6, 6, 2, 2, 2, 2]
])

"""
Employee Present: analyse whether the employee is present yes or no on a given time
Based on the employee list of 3 (id, start time, duration)
"""
def employee_present(employee, time):
   employee_start_time = employee[1]
   employee_duration = employee[2]
   employee_end_time = employee_start_time + employee_duration
   if (time >= employee_start_time) and (time < employee_end_time):
      return True
   return False
```

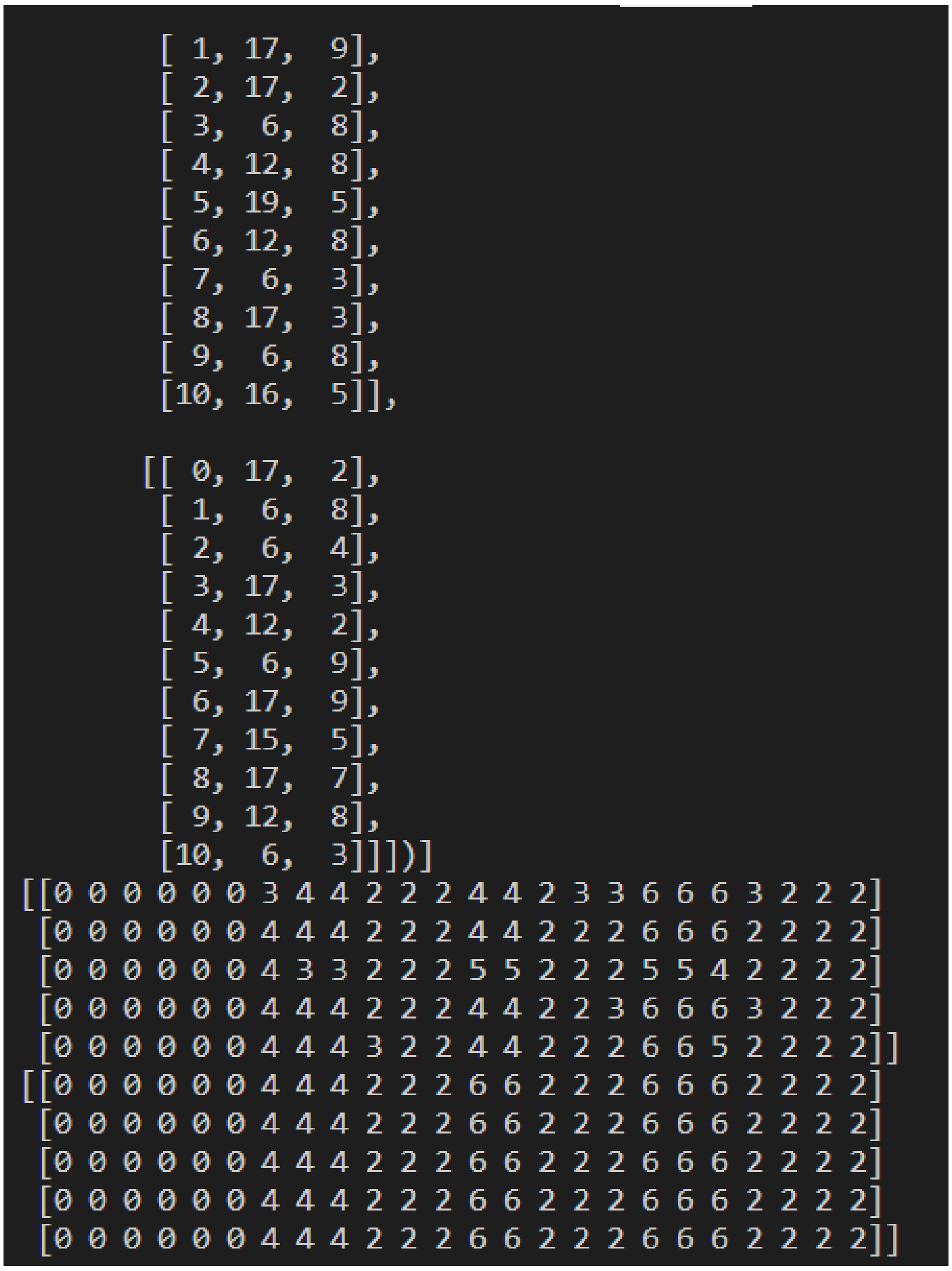
```
"""
convert a staff planning to a staff-needed plannig
The employee planning is organised per employee, the staff-needed planning is the number of employees
working per hour
The staff-needed planning is based on the employee planning and will allow to calculate the difference
with the staff-needed
It doesnt work overnight, but our shop isnt open at night anyway
"""
def staffplanning_to_hourlyplanning(staff_planning):

    hourlystaff_week = []
    for day in staff_planning:

        hourlystaff_day = []
        for employee in day:

            employee_present_hour = []
            for time in range(0, 24):

                employee_present_hour.append(employee_present(employee, time))

            hourlystaff_day.append(employee_present_hour)

        hourlystaff_week.append(hourlystaff_day)

    hourlystaff_week = np.array(hourlystaff_week).sum(axis = 1)
    return hourlystaff_week

"""
the cost is calculated as hours understaffed + hours overstaffed
"""
def cost(hourlystaff, hourlystaff_needed):
    errors = hourlystaff - hourlystaff_needed
    overstaff = abs(errors[errors > 0].sum())
    understaff = abs(errors[errors < 0].sum())

    overstaff_cost = 1
    understaff_cost = 1

    cost = overstaff_cost * overstaff + understaff_cost * understaff
    return cost


"""
generate an entirely random staff planning for a certain number of days
start time is random between 0 and 23; duration is random between 0 and 10
"""
def generate_random_staff_planning(n_days, n_staff):
    period_planning = []
    for day in range(n_days):
        day_planning = []
        for employee_id in range(n_staff):
            start_time = np.random.randint(0, 23)
            duration = np.random.randint(0, 10)
            employee = [employee_id, start_time, duration]
            day_planning.append(employee)
        period_planning.append(day_planning)

    return period_planning

# An example of the code until here

# show the random initialization of the week planning
random_staff_planning = generate_random_staff_planning(n_days = 5, n_staff = 11)
random_staff_planning

# show the cost of this random week planning
cost(staffplanning_to_hourlyplanning(random_staff_planning), hourlystaff_needed)

"""
create a parent generation of n parent plannings
"""
def create_parent_generation(n_parents, n_days = 7, n_staff = 11):
    parents = []
    for i in range(n_parents):
        parent = generate_random_staff_planning(n_days = n_days, n_staff = n_staff)
        parents.append(parent)
    return parents


"""
for each iteration, select randomly two parents and make a random combination of those two parents
by applying a randomly generated yes/no mask to the two selected parents
"""
def random_combine(parents, n_offspring):
    n_parents = len(parents)
    n_periods = len(parents[0])
    n_employees = len(parents[0][0])

    offspring = []
    for i in range(n_offspring):
        random_dad = parents[np.random.randint(low = 0, high = n_parents - 1)]
        random_mom = parents[np.random.randint(low = 0, high = n_parents - 1)]

        dad_mask = np.random.randint(0, 2, size = np.array(random_dad).shape)
        mom_mask = np.logical_not(dad_mask)

        child = np.add(np.multiply(random_dad, dad_mask), np.multiply(random_mom, mom_mask))

        offspring.append(child)
    return offspring

def mutate_parent(parent, n_mutations):
    size1 = parent.shape[0]
    size2 = parent.shape[1]

    for i in range(n_mutations):

        rand1 = np.random.randint(0, size1)
        rand2 = np.random.randint(0, size2)
        rand3 = np.random.randint(1, 2)
        parent[rand1,rand2,rand3] = np.random.randint(0, 10)

    return parent

def mutate_gen(parent_gen, n_mutations):
    mutated_parent_gen = []
    for parent in parent_gen:
        mutated_parent_gen.append(mutate_parent(parent, n_mutations))
    return mutated_parent_gen

def is_acceptable(parent):
    return np.logical_not((np.array(parent)[:,:,2:] > 10).any()) #work more than 10 hours is not ok

def select_acceptable(parent_gen):
    parent_gen = [parent for parent in parent_gen if is_acceptable(parent)]
    return parent_gen

def select_best(parent_gen, hourlystaff_needed, n_best):
    costs = []
    for idx, parent_staff_planning in enumerate(parent_gen):

        parent_hourly_planning = staffplanning_to_hourlyplanning(parent_staff_planning)
        parent_cost = cost(parent_hourly_planning, hourlystaff_needed)
        costs.append([idx, parent_cost])

    print('generations best is: {}, generations worst is: {}'.format(pd.DataFrame(costs)[1].min(),
pd.DataFrame(costs)[1].max()))

    costs_tmp = pd.DataFrame(costs).sort_values(by = 1, ascending = True).reset_index(drop=True)
    selected_parents_idx = list(costs_tmp.iloc[:n_best,0])
    selected_parents = [parent for idx, parent in enumerate(parent_gen) if idx in selected_parents_idx]

    return selected_parents

"""
the overall function
"""
def gen_algo(hourlystaff_needed, n_iterations):

    generation_size = 500

    parent_gen = create_parent_generation(n_parents = generation_size, n_days = 5, n_staff = 11)
    for it in range(n_iterations):
        parent_gen = select_acceptable(parent_gen)
        parent_gen = select_best(parent_gen, hourlystaff_needed, n_best = 100)
        parent_gen = random_combine(parent_gen, n_offspring = generation_size)
        parent_gen = mutate_gen(parent_gen, n_mutations = 1)

    best_child = select_best(parent_gen, hourlystaff_needed, n_best = 1)
    return best_child

best_planning = gen_algo(hourlystaff_needed, n_iterations = 100)
print(best_planning)
print(staffplanning_to_hourlyplanning(best_planning[0]))
print(hourlystaff_needed)
```

Output:



```
generations best is: 32, generations worst is: 58
generations best is: 32, generations worst is: 60
generations best is: 32, generations worst is: 57
generations best is: 32, generations worst is: 62
generations best is: 32, generations worst is: 64
generations best is: 32, generations worst is: 63
generations best is: 32, generations worst is: 58
generations best is: 32, generations worst is: 57
generations best is: 32, generations worst is: 56
generations best is: 32, generations worst is: 58
generations best is: 32, generations worst is: 58
generations best is: 32, generations worst is: 58
generations best is: 32, generations worst is: 58
generations best is: 32, generations worst is: 56
generations best is: 32, generations worst is: 58
generations best is: 32, generations worst is: 60
```

```
generations best is: 32, generations wors
[array([[[ 0,  7,  2],
        [ 1,  6,  3],
        [ 2,  6,  8],
        [ 3, 17,  7],
        [ 4, 17,  4],
        [ 5,  6,  8],
        [ 6, 17,  8],
        [ 7, 12,  8],
        [ 8,  0,  0],
        [ 9, 15,  5],
        [10, 12,  8]],

       [[ 0,  6,  3],
        [ 1,  6,  3],
        [ 2,  6,  8],
        [ 3, 17,  9],
        [ 4, 17,  3],
        [ 5, 17,  1],
        [ 6, 12,  8],
        [ 7, 18,  9],
        [ 8, 17,  3],
        [ 9,  6,  8],
        [10, 12,  8]],
```



```
         [ 1, 17,  9],
         [ 2, 17,  2],
         [ 3,  6,  8],
         [ 4, 12,  8],
         [ 5, 19,  5],
         [ 6, 12,  8],
         [ 7,  6,  3],
         [ 8, 17,  3],
         [ 9,  6,  8],
         [10, 16,  5]],

        [[ 0, 17,  2],
         [ 1,  6,  8],
         [ 2,  6,  4],
         [ 3, 17,  3],
         [ 4, 12,  2],
         [ 5,  6,  9],
         [ 6, 17,  9],
         [ 7, 15,  5],
         [ 8, 17,  7],
         [ 9, 12,  8],
         [10,  6,  3]]])]
[[0 0 0 0 0 0 3 4 4 2 2 2 4 4 2 3 3 6 6 6 3 2 2 2]
 [0 0 0 0 0 0 4 4 4 2 2 2 4 4 2 2 2 6 6 6 2 2 2 2]
 [0 0 0 0 0 0 4 3 3 2 2 2 5 5 2 2 2 5 5 4 2 2 2 2]
 [0 0 0 0 0 0 4 4 4 2 2 2 4 4 2 2 3 6 6 6 3 2 2 2]
 [0 0 0 0 0 0 4 4 4 3 2 2 4 4 2 2 2 6 6 5 2 2 2 2]]
[[0 0 0 0 0 0 4 4 4 2 2 2 6 6 2 2 2 6 6 6 2 2 2 2]
 [0 0 0 0 0 0 4 4 4 2 2 2 6 6 2 2 2 6 6 6 2 2 2 2]
 [0 0 0 0 0 0 4 4 4 2 2 2 6 6 2 2 2 6 6 6 2 2 2 2]
 [0 0 0 0 0 0 4 4 4 2 2 2 6 6 2 2 2 6 6 6 2 2 2 2]
 [0 0 0 0 0 0 4 4 4 2 2 2 6 6 2 2 2 6 6 6 2 2 2 2]]
```

Conclusion:

Implemented Genetic Algorithms for Staff Planning.

Practical 7

Aim: *Implement ANN.*

Theory:

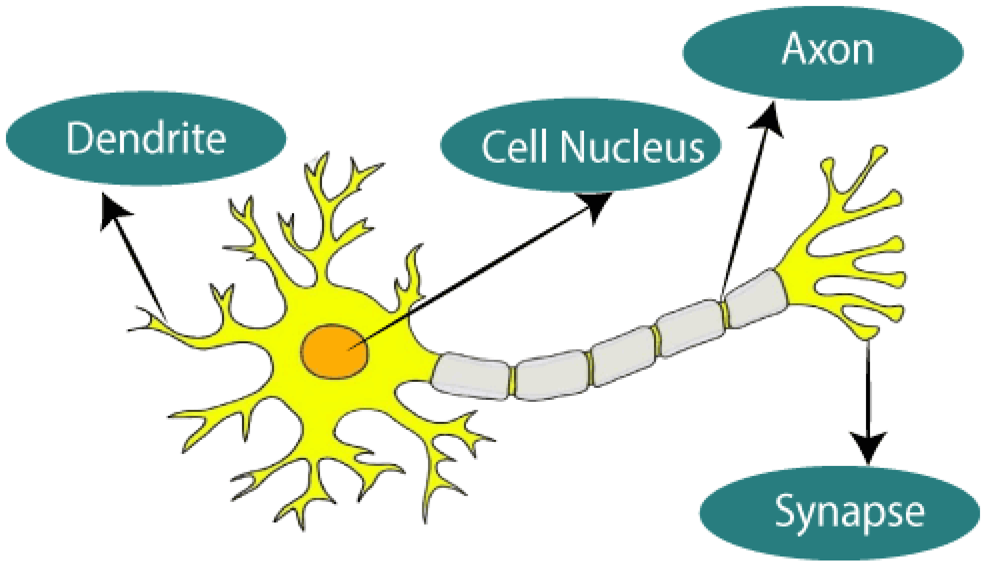
Artificial Neural Network Tutorial provides basic and advanced concepts of ANNs. Our Artificial Neural Network tutorial is developed for beginners as well as professions.

The term "Artificial neural network" refers to a biologically inspired sub-field of artificial intelligence modeled after the brain. An Artificial neural network is usually a computational network based on biological neural networks that construct the structure of the human brain. Similar to a human brain has neurons interconnected to each other, artificial neural networks also have neurons that are linked to each other in various layers of the networks. These neurons are known as nodes.

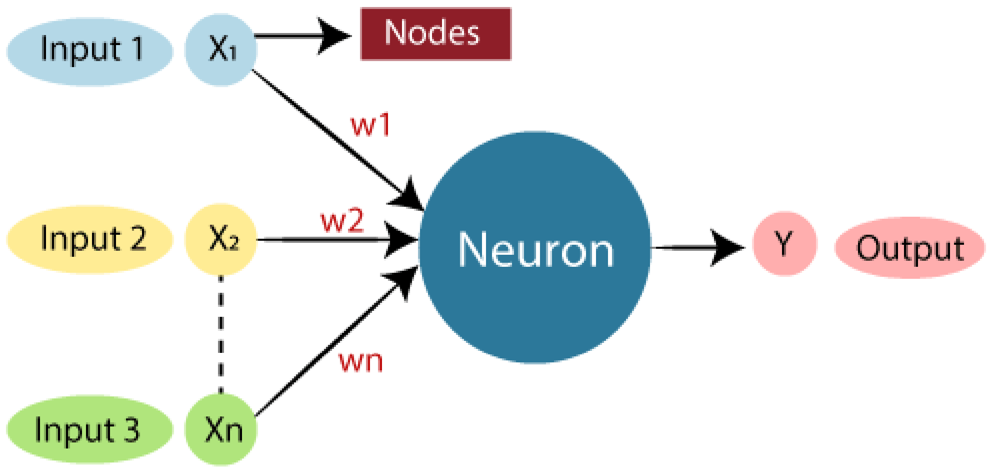
Artificial neural network tutorial covers all the aspects related to the artificial neural network. In this tutorial, we will discuss ANNs, Adaptive resonance theory, Kohonen self-organizing map, Building blocks, unsupervised learning, Genetic algorithm, etc.

What is Artificial Neural Network?

The term "**Artificial Neural Network**" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.



The typical Artificial Neural Network looks something like the given figure.



Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

Relationship between Biological neural network and artificial neural network:

Biological Neural Network	Artificial Neural Network
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

An **Artificial Neural Network** in the field of **Artificial intelligence** where it attempts to mimic the network of neurons makes up a human brain so that computers will have an option to understand things and make decisions in a human-like manner. The artificial neural network is designed by programming computers to behave simply like interconnected brain cells.

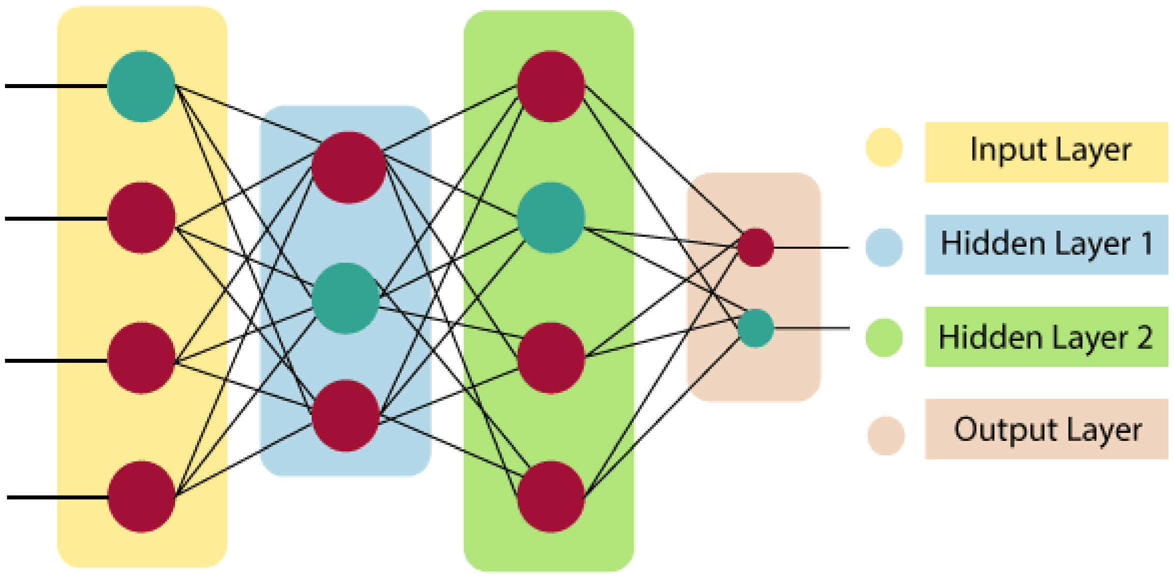
There are around 1000 billion neurons in the human brain. Each neuron has an association point somewhere in the range of 1,000 and 100,000. In the human brain, data is stored in such a manner as to be distributed, and we can extract more than one piece of this data when necessary from our memory parallelly. We can say that the human brain is made up of incredibly amazing parallel processors.

We can understand the artificial neural network with an example, consider an example of a digital logic gate that takes an input and gives an output. "OR" gate, which takes two inputs. If one or both the inputs are "On," then we get "On" in output. If both the inputs are "Off," then we get "Off" in output. Here the output depends upon input. Our brain does not perform the same task. The outputs to inputs relationship keep changing because of the neurons in our brain, which are "learning."

The architecture of an artificial neural network:

To understand the concept of the architecture of an artificial neural network, we have to understand what a neural network consists of. In order to define a neural network that consists of a large number of artificial neurons, which are termed units arranged in a sequence of layers. Lets us look at various types of layers available in an artificial neural network.

Artificial Neural Network primarily consists of three layers:



Input Layer:

As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

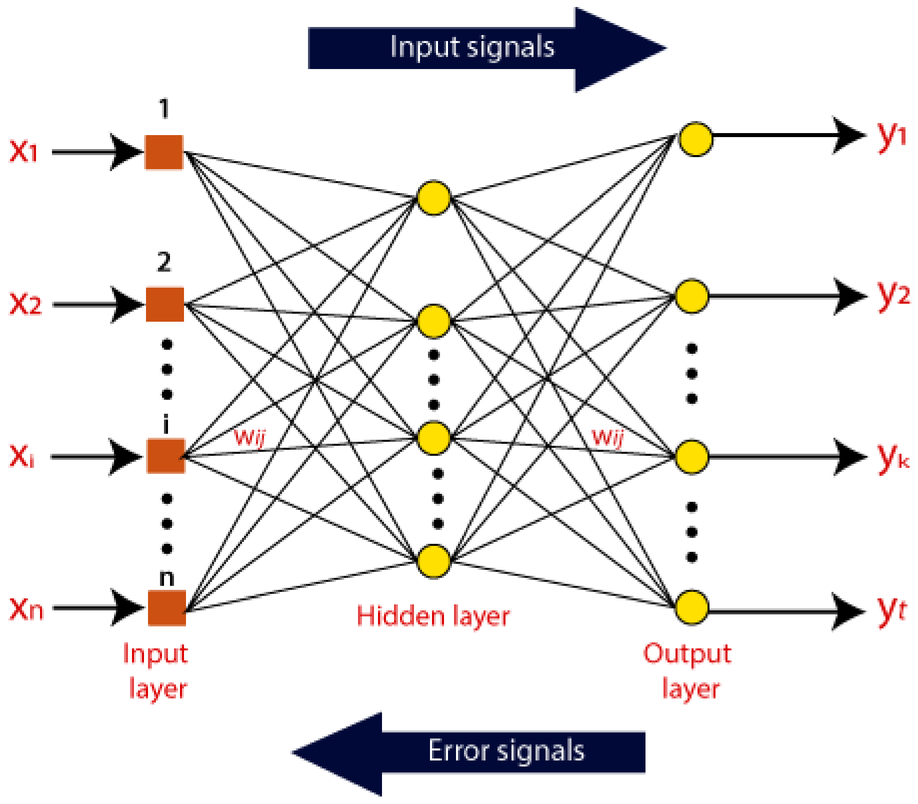
The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

$$\sum_{i=1}^{n} Wi * Xi + b$$

It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

How do artificial neural networks work?

Artificial Neural Network can be best represented as a weighted directed graph, where the artificial neurons form the nodes. The association between the neurons outputs and neuron inputs can be viewed as the directed edges with weights. The Artificial Neural Network receives the input signal from the external source in the form of a pattern and image in the form of a vector. These inputs are then mathematically assigned by the notations x(n) for every n number of inputs.



Afterward, each of the input is multiplied by its corresponding weights (these weights are the details utilized by the artificial neural networks to solve a specific problem). In general terms, these weights normally represent the strength of the interconnection between neurons inside the artificial neural network. All the weighted inputs are summarized inside the computing unit.

If the weighted sum is equal to zero, then bias is added to make the output non-zero or something else to scale up to the system's response. Bias has the same input, and weight equals to 1. Here the total of weighted inputs can be in the range of 0 to positive infinity. Here, to keep the response in the limits of the desired value, a certain maximum value is benchmarked, and the total of weighted inputs is passed through the activation function.

The activation function refers to the set of transfer functions used to achieve the desired output. There is a different kind of the activation function, but primarily either linear or nonlinear sets of functions. Some of the commonly used sets of activation functions are the Binary, linear, and Tan hyperbolic sigmoidal activation functions. Let us take a look at each of them in details:

Binary:

In binary activation function, the output is either a one or a 0. Here, to accomplish this, there is a threshold value set up. If the net weighted input of neurons is more than 1, then the final output of the activation function is returned as one or else the output is returned as 0.

Sigmoidal Hyperbolic:

The Sigmoidal Hyperbola function is generally seen as an "**S**" shaped curve. Here the tan hyperbolic function is used to approximate output from the actual net input. The function is defined as:

F(x) = (1/1 + exp(-????x))

Where ???? is considered the Steepness parameters.

Code:

minst.py -driver code

```
import numpy as np
import torch
import torchvision
from torch.utils.data import DataLoader

import layers
import loss
import optimizers
from model import Model

def get_dataset(batch_size):
    train_loader = DataLoader(
        torchvision.datasets.MNIST('./data/', train=True, download=True,
                              transform=torchvision.transforms.Compose([
                                  torchvision.transforms.ToTensor(),
                              ])), shuffle=True, batch_size=batch_size)

    test_loader = DataLoader(
        torchvision.datasets.MNIST('./data/', train=False, download=True,
                              transform=torchvision.transforms.Compose([
                                  torchvision.transforms.ToTensor(),
                              ])), shuffle=True, batch_size=batch_size)
    return train_loader, test_loader

if __name__ == '__main__':
    torch.random.manual_seed(1234)
    np.random.seed(1234)

    epochs = 10
    lr = 0.01
    batch_size = 32
```

```
optimizer = optimizers.SGD(learning_rate=lr)
criterion = loss.CrossEntropy()
layers = [
    layers.LinearLayer(784, 512),
    layers.ReLU(),
    layers.Dropout(keep_rate=0.8),
    layers.LinearLayer(512, 512),
    layers.ReLU(),
    layers.Dropout(keep_rate=0.8),
    layers.LinearLayer(512, 10)
]
model = Model(layers, optimizer, criterion)

train_loader, test_loader = get_dataset(batch_size)
for epoch_id in range(epochs):
    model.train()
    total = 0
    correct = 0
    for i, (x, y) in enumerate(train_loader):
        x = x.numpy().reshape(y.shape[0], -1, 1)
        y = y.numpy()

        model.optimizer.zero_grad()
        loss, pred, logits = model.forward(x, y)
        model.backward(y, logits)

        correct += np.sum(y == pred.flatten())
        total += y.shape[0]

        if i % 100 == 0:
            print("Loss:", loss.mean())
    print("Accuracy (train) epoch {}: {} %".format(epoch_id + 1, correct / total * 100.0))

    model.eval()
    total = 0
    correct = 0
    for i, (x, y) in enumerate(test_loader):
        x = x.numpy().reshape(y.shape[0], -1, 1)
        y = y.numpy()

        _, pred, _ = model.forward(x, y)

        correct += np.sum(y == pred.flatten())
        total += y.shape[0]

    print("Accuracy (test) epoch {}: {} %".format(epoch_id + 1, correct / total * 100.0))

total = 0
correct = 0
for i, (x, y) in enumerate(train_loader):
    x = x.numpy().reshape(y.shape[0], -1, 1)
    y = y.numpy()

    _, pred, _ = model.forward(x, y)

    correct += np.sum(y == pred.flatten())
```

```
        total += y.shape[0]

    print("Accuracy final (train) epoch {}: {} %".format(epochs, correct / total * 100.0))
```

layers.py

```
from typing import Tuple, List, Dict, Any

import numpy as np

class Param:

    def __init__(self, data: np.ndarray):
        self.data = data

class Layer:

    def __init__(self, train: bool = True):
        """Creates layer.

        :param train: bool deciding whether the layer is in train/eval mode
        """
        self.train = train

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:
        """Forward pass.

        :param x: input of the layer
        :return: output of the layer, *args and **kwargs as a tuple
        Args and kwargs are passed as arguments for backward pass.
        """
        pass

    def backward(self, x: np.ndarray, dy: np.ndarray, *args, **kwargs) -> Tuple[np.ndarray,
List[np.ndarray]]:
        """Backward pass.

        :param x: the layer input
        :param dy: upstream gradient
        :param args: optional args
        :param kwargs: optional kwargs
        :return: tuple of downstream gradient, then list of gradients with respect to parameters (in order)
        defined in weights method
        """
        pass

    def weights(self) -> List[Param]:
        """Learnable parameters of the layer - order must be the same as in backward."""
        return []

def xavier_uniform_init(input_dim, output_dim, gain: float = 1.0):
    r = gain * np.sqrt(6.0 / (input_dim + output_dim))
    return np.random.uniform(-r, r, (input_dim, output_dim))

class LinearLayer(Layer):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.W = Param(xavier_uniform_init(input_dim, output_dim))
        self.b = Param(np.zeros(shape=(1, output_dim)))
        self.output_dim = output_dim

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:
        y = (np.matmul(x.transpose((0, 2, 1)), self.W.data) + self.b.data).transpose((0, 2, 1))
        assert y.shape == (x.shape[0], self.output_dim, 1)
        return y, [], {}

    def backward(self, x: np.ndarray, dy: np.ndarray, *args, **kwargs) -> Tuple[np.ndarray,
List[np.ndarray]]:
        batch_size = x.shape[0]

        dx = dy.transpose((0, 2, 1)).dot(self.W.data.T).transpose((0, 2, 1))
        assert dx.shape == x.shape

        dW = np.matmul(dy, x.transpose((0, 2, 1))).transpose((0, 2, 1))
        assert dW.shape == (batch_size, *self.W.data.shape)

        db = dy.transpose((0, 2, 1))
        assert db.shape == (batch_size, *self.b.data.shape)

        return dx, [dW, db]

    def weights(self) -> List[Param]:
        return [self.W, self.b]

class ReLU(Layer):

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:
        return np.maximum(x, 0), [], {}

    def backward(self, x: np.ndarray, dy: np.ndarray, *args, **kwargs) -> Tuple[np.ndarray,
List[np.ndarray]]:
        dx = np.maximum(x, 0) * dy
        assert dy.shape == dx.shape
        return dx, []

class Dropout(Layer):

    def __init__(self, keep_rate: float):
        super().__init__()
        self.keep_rate = keep_rate

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:
        mask = (np.random.binomial(1, self.keep_rate, size=x.shape) / self.keep_rate
                if self.train else np.ones_like(x))
        return mask * x, [], {"mask": mask}

    def backward(self, x: np.ndarray, dy: np.ndarray, *args, **kwargs) -> Tuple[np.ndarray,
List[np.ndarray]]:
        assert "mask" in kwargs
        return kwargs["mask"] * dy, []
```

loss.py

```
from typing import Tuple

import numpy as np

def softmax(x):
    e = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e / np.sum(e, axis=1, keepdims=True)

class Loss:

    def forward(self, y_true: np.ndarray, logits: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        """Returns loss value and prediction."""
        pass

    def backward(self, y_true: np.ndarray, logits: np.ndarray):
        pass

class CrossEntropy(Loss):

    def forward(self, y_true, logits):
        prob = softmax(logits)
        return - np.log(prob[range(logits.shape[0]), y_true]), np.argmax(prob, axis=1)

    def backward(self, y_true, logits):
        grad = softmax(logits)
        grad[range(logits.shape[0]), y_true] -= 1
        return grad
```

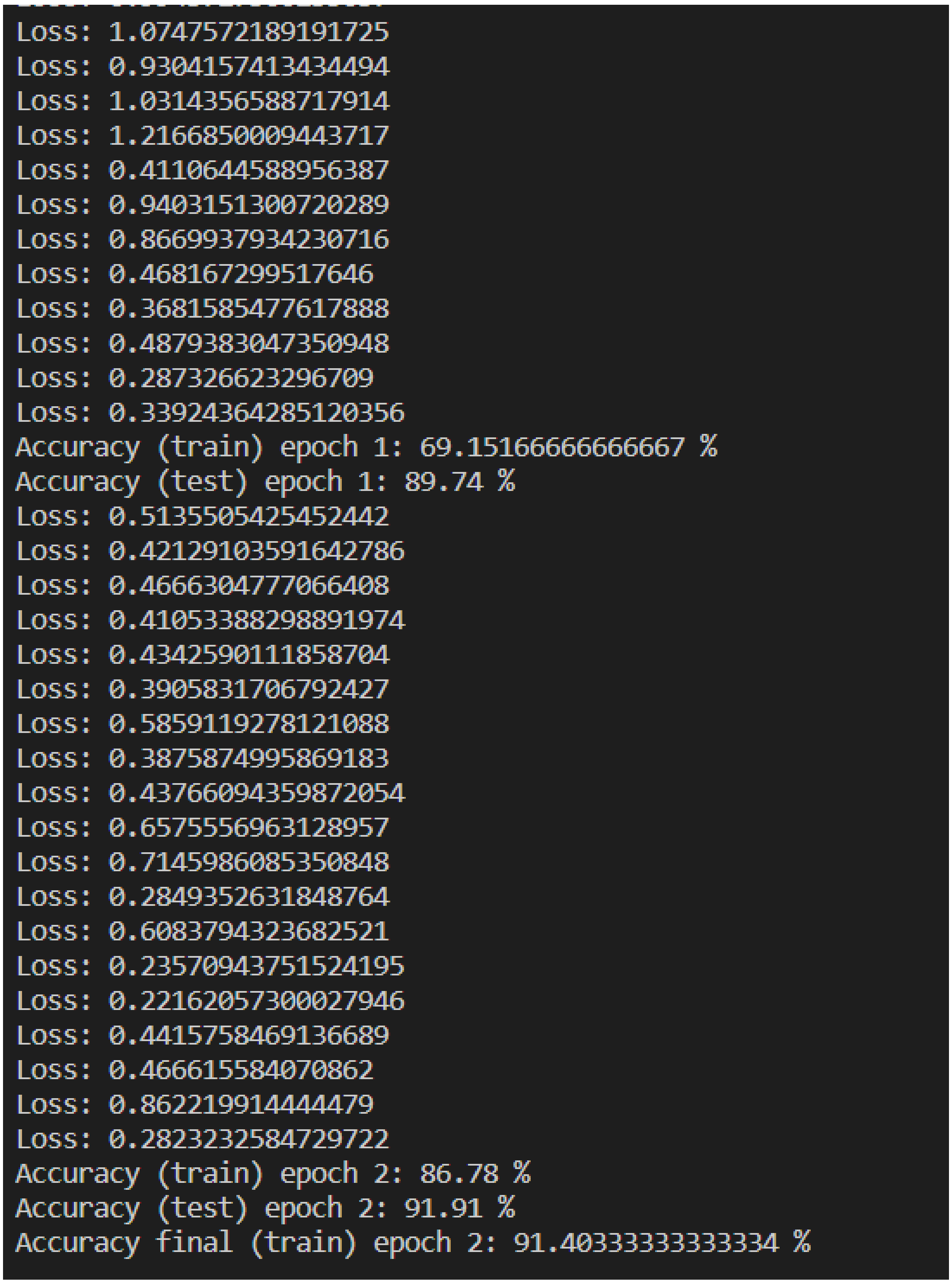
model.py

```
from typing import List

from layers import Layer
from loss import Loss
from optimizers import Optimizer

class Model:

    def __init__(self, layers: List[Layer], optimizer: Optimizer, criterion: Loss):
        self.layers = layers
        self.optimizer = optimizer
        self.criterion = criterion

    def train(self):
        for l in self.layers:
            l.train = True

    def eval(self):
        for l in self.layers:
            l.train = False

    def forward(self, x, y):
        for idx, layer in enumerate(self.layers):
            new_x, args, kwargs = layer.forward(x)
            self.optimizer.save(idx, (x, args, kwargs))
            x = new_x

        logits = x
        loss, pred = self.criterion.forward(y, logits)
        return loss, pred, logits

    def backward(self, y, logits):
        upstream_grad = self.criterion.backward(y, logits)
        for idx, layer in reversed(list(enumerate(self.layers))):
            x, args, kwargs = self.optimizer.load(idx)
            upstream_grad, grad = layer.backward(x, upstream_grad, *args, **kwargs)
            self.optimizer.update_layer(grad, layer)
```

optimizers.py

```
import numpy as np

from layers import Layer, Param

class Optimizer:

    def __init__(self):
        self.cache = {}

    def load(self, idx):
        return self.cache[idx]

    def save(self, idx, x):
        self.cache[idx] = x

    def update_layer(self, grad: np.ndarray, layer: Layer):
        weights = layer.weights()
        assert len(weights) == len(grad)
        for weight, grad in zip(weights, grad):
            self.step(grad, weight)

    def zero_grad(self):
        self.cache = {}

    def step(self, grad: np.ndarray, weight: Param):
        pass

class SGD(Optimizer):

    def __init__(self, learning_rate: float = 0.01):
        super().__init__()
        self.learning_rate = learning_rate

    def step(self, grad: np.ndarray, weight: Param):
        weight.data -= self.learning_rate * grad.mean(axis=0)
```

Output:



```
Loss: 1.0747572189191725
Loss: 0.9304157413434494
Loss: 1.0314356588717914
Loss: 1.2166850009443717
Loss: 0.4110644588956387
Loss: 0.9403151300720289
Loss: 0.8669937934230716
Loss: 0.468167299517646
Loss: 0.3681585477617888
Loss: 0.4879383047350948
Loss: 0.287326623296709
Loss: 0.33924364285120356
Accuracy (train) epoch 1: 69.15166666666667 %
Accuracy (test) epoch 1: 89.74 %
Loss: 0.5135505425452442
Loss: 0.42129103591642786
Loss: 0.4666304777066408
Loss: 0.41053388298891974
Loss: 0.4342590111858704
Loss: 0.3905831706792427
Loss: 0.5859119278121088
Loss: 0.3875874995869183
Loss: 0.43766094359872054
Loss: 0.6575556963128957
Loss: 0.7145986085350848
Loss: 0.2849352631848764
Loss: 0.6083794323682521
Loss: 0.23570943751524195
Loss: 0.22162057300027946
Loss: 0.4415758469136689
Loss: 0.466615584070862
Loss: 0.862219914444479
Loss: 0.2823232584729722
Accuracy (train) epoch 2: 86.78 %
Accuracy (test) epoch 2: 91.91 %
Accuracy final (train) epoch 2: 91.40333333333334 %
```

Conclusion:

Implemented ANN.

Practical 8

Aim: *Implement feed forward back propagation neural network learning algorithm for the restaurant waiting problem.*

Theory:

What is Backpropagation?

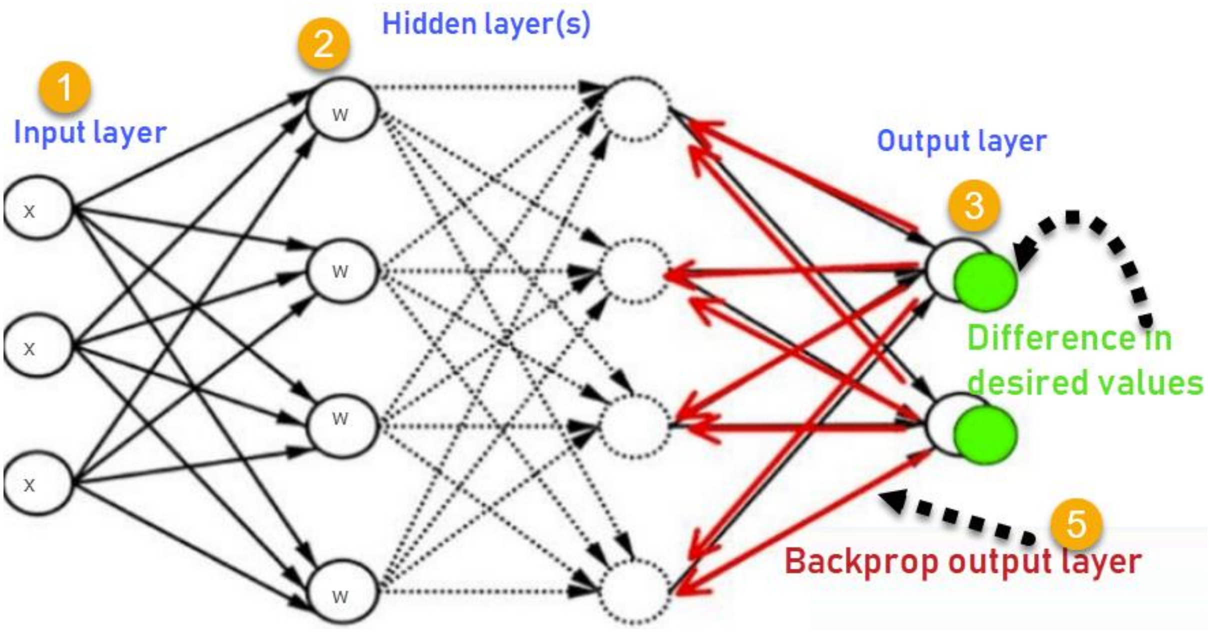
Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization.

Backpropagation in neural network is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

How Backpropagation Algorithm Works

The Back propagation algorithm in neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. It generalizes the computation in the delta rule.

Consider the following Back propagation neural network example diagram to understand:



1. *Inputs X, arrive through the preconnected path*
2. *Input is modelled using real weights W. The weights are usually randomly selected.*

3. *Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.*
4. *Calculate the error in the outputs.*
5. *Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.*

Keep repeating the process until the desired output is achieved

Why We Need Backpropagation?

Most prominent advantages of Backpropagation are:

- *Backpropagation is fast, simple and easy to program*
- *It has no parameters to tune apart from the numbers of input*
- *It is a flexible method as it does not require prior knowledge about the network*
- *It is a standard method that generally works well*
- *It does not need any special mention of the features of the function to be learned.*

What is a Feed Forward Network?

A feedforward neural network is an artificial neural network where the nodes never form a cycle. This kind of neural network has an input layer, hidden layers, and an output layer. It is the first and simplest type of artificial neural network.

Types of Backpropagation Networks

Two Types of Backpropagation Networks are:

- *Static Back-propagation*
- *Recurrent Backpropagation*

Static back-propagation:

It is one kind of backpropagation network which produces a mapping of a static input for static output. It is useful to solve static classification issues like optical character recognition.

Recurrent Backpropagation:

Recurrent Back propagation in data mining is fed forward until a fixed value is achieved. After that, the error is computed and propagated backward.

The main difference between both of these methods is: that the mapping is rapid in static back-propagation while it is non-static in recurrent backpropagation.

Backpropagation Key Points

- Simplifies the network structure by elements weighted links that have the least effect on the trained network
- You need to study a group of input and activation values to develop the relationship between the input and hidden unit layers.
- It helps to assess the impact that a given input variable has on a network output. The knowledge gained from this analysis should be represented in rules.
- Backpropagation is especially useful for deep neural networks working on error-prone projects, such as image or speech recognition.
- Backpropagation takes advantage of the chain and power rules allows backpropagation to function with any number of outputs.

Best practice Backpropagation

Backpropagation in neural network can be explained with the help of “Shoe Lace” analogy

Too little tension =

- Not enough constraining and very loose

Too much tension =

- Too much constraint (overtraining)
- Taking too much time (relatively slow process)
- Higher likelihood of breaking

Pulling one lace more than other =

- Discomfort (bias)

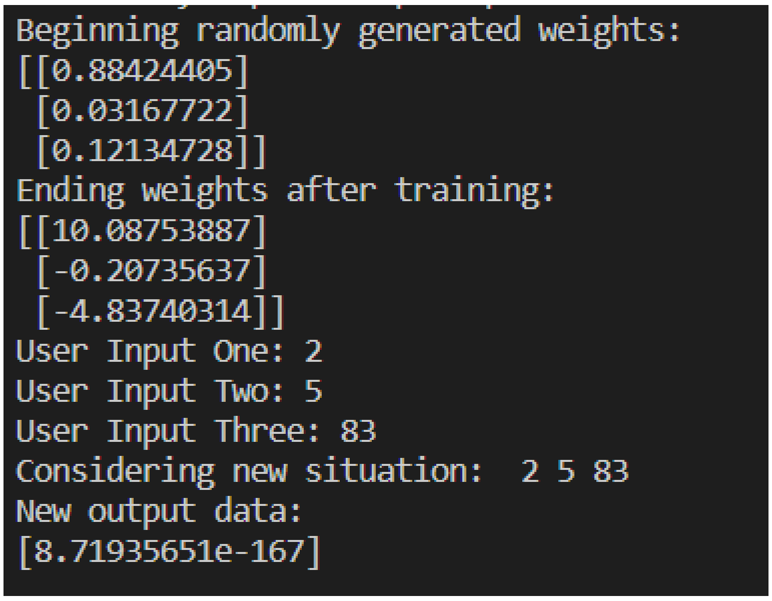
Disadvantages of using Backpropagation

- The actual performance of backpropagation on a specific problem is dependent on the input data.
- Back propagation algorithm in data mining can be quite sensitive to noisy data
- You need to use the matrix-based approach for backpropagation instead of mini-batch.

Code:

```
import numpy as np

class NeuralNetwork():
    def __init__(self):
        #seeding for random number generation
        np.random.seed()

        #converting weights to a 3 by 1 matrix
        self.synaptic_weights=2*np.random.random((3,1))-1
    #x is output variable
    def sigmoid(self, x):
        #applying the sigmoid function
        return 1/(1+np.exp(-x))

    def sigmoid_derivative(self,x):
        #computing derivative to the sigmoid function
        return x*(1-x)

    def train(self,training_inputs,training_outputs,training_iterations):

        #training the model to make accurate predictions while adjusting
        for iteration in range(training_iterations):
            #siphon the training data via the neuron
            output=self.think(training_inputs)

            error=training_outputs-output

            #performing weight adjustments
            adjustments=np.dot(training_inputs.T,error*self.sigmoid_derivative(output))

            self.synaptic_weights+=adjustments

    def think(self,inputs):
        #passing the inputs via the neuron to get output
        #converting values to floats

        inputs=inputs.astype(float)
        output=self.sigmoid(np.dot(inputs,self.synaptic_weights))

        return output

if __name__=="__main__":

    #initializing the neuron class
    neural_network=NeuralNetwork()

    print("Beginning randomly generated weights: ")
    print(neural_network.synaptic_weights)

    #training data consisting of 4 examples--3 inputs & 1 output
    training_inputs=np.array([[0,0,1],[1,1,1],[1,0,1],[0,1,1]])
    training_outputs=np.array([[0,1,1,0]]).T

    #training taking place
    neural_network.train(training_inputs,training_outputs,15000)

    print("Ending weights after training: ")
    print(neural_network.synaptic_weights)

    user_input_one=str(input("User Input One: "))
    user_input_two=str(input("User Input Two: "))
    user_input_three=str(input("User Input Three: "))

    print("Considering new situation: ",user_input_one,user_input_two,user_input_three)
    print("New output data: ")
print(neural_network.think(np.array([user_input_one,user_input_two,user_input_three])))
```

Output:



```
Beginning randomly generated weights:
[[0.88424405]
 [0.03167722]
 [0.12134728]]
Ending weights after training:
[[10.08753887]
 [-0.20735637]
 [-4.83740314]]
User Input One: 2
User Input Two: 5
User Input Three: 83
Considering new situation:  2 5 83
New output data:
[8.71935651e-167]
```

Conclusion:

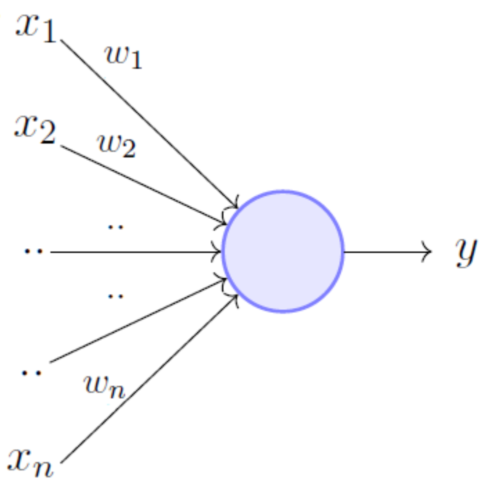
Implemented feed forward back propagation neural network learning algorithm for the restaurant waiting problem.

Practical 9

Aim: *Implement the Perceptron Algorithm.*

Theory:

A perceptron is not the Sigmoid neuron we use in ANNs or any deep learning networks today.

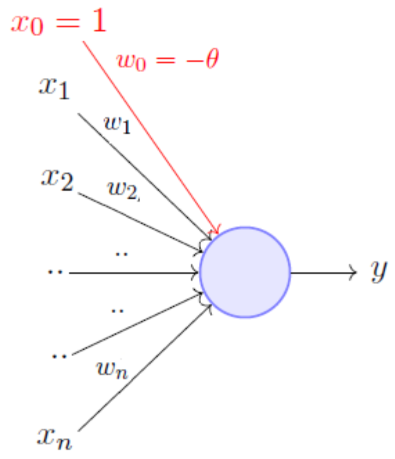


$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

The perceptron model is a more general computational model than McCulloch-Pitts neuron. It takes an input, aggregates it (weighted sum) and returns 1 only if the aggregated sum is more than some threshold else returns 0. Rewriting the threshold as shown above and making it a constant input with a variable weight, we would end up with something like the following:



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

*A single perceptron can only be used to implement **linearly separable** functions. It takes both real and boolean inputs and associates a set of **weights** to them, along with a **bias** (the threshold thing I mentioned above). We learn the weights, we get the function. Let's use a perceptron to learn an OR function.*

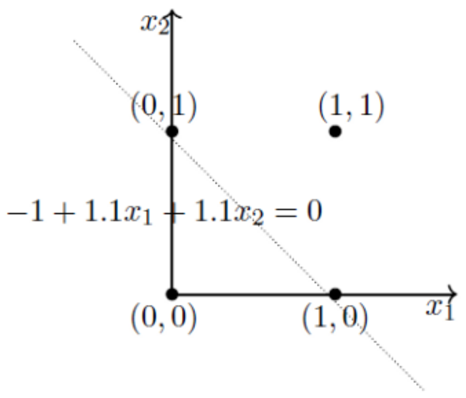
OR Function Using A Perceptron

x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$

$$\begin{aligned}
w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 &\implies w_0 < 0 \\
w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 &\implies w_2 > -w_0 \\
w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 &\implies w_1 > -w_0 \\
w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 &\implies w_1 + w_2 > -w_0
\end{aligned}$$

One possible solution is

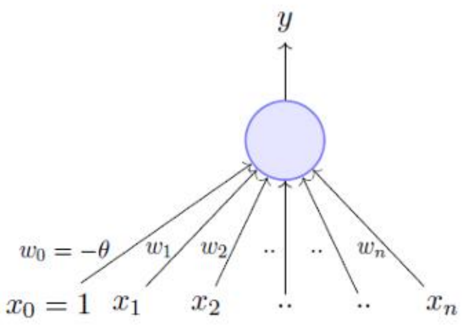
$$w_0 = -1,\ w_1 = 1.1,\ w_2 = 1.1$$



What's going on above is that we defined a few conditions (the weighted sum has to be more than or equal to 0 when the output is 1) based on the OR function output for various sets of inputs, we solved for weights based on those conditions and we got a line that perfectly separates positive inputs from those of negative.

Doesn't make any sense? Maybe now is the time you go through that post I was talking about. Minsky and Papert also proposed a more principled way of learning these weights using a set of examples (data). Mind you that this is NOT a Sigmoid neuron and we're not going to do any Gradient Descent.

Setting Up The Problem



$x_1 = isActorDamon$

$x_2 = isGenreThriller$

$x_3 = isDirectorNolan$

$x_4 = imdbRating(scaled\ to\ 0\ to\ 1)$

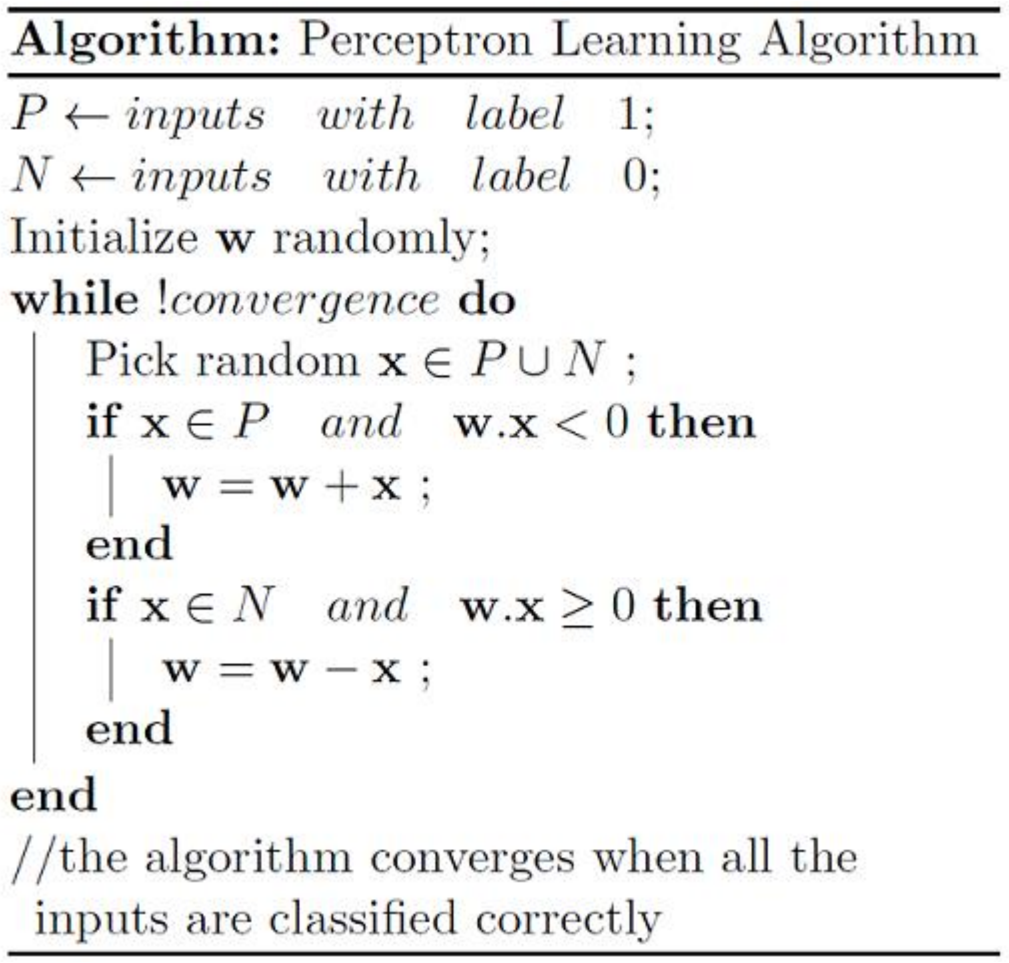
... ...

$x_n = criticsRating(scaled\ to\ 0\ to\ 1)$

We are going to use a perceptron to estimate if I will be watching a movie based on historical data with the above-mentioned inputs. The data has positive and negative examples, positive being the movies I watched i.e., 1. Based on the data, we are going to learn the weights using the perceptron learning algorithm. For visual simplicity, we will only assume two-dimensional input.

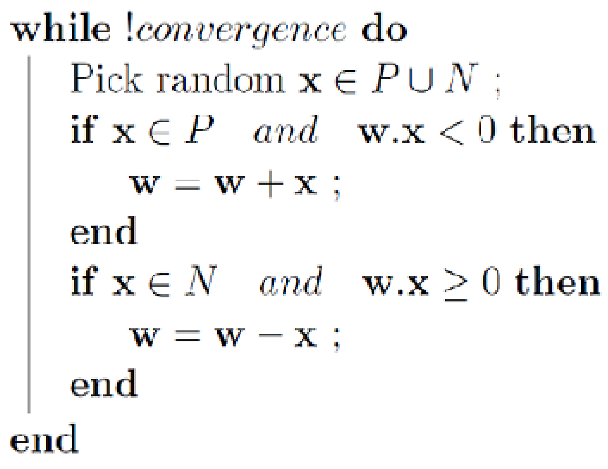
Perceptron Learning Algorithm

Our goal is to find the **w** *vector that can perfectly classify positive inputs and negative inputs in our data. I will get straight to the algorithm. Here goes:*



```
Algorithm: Perceptron Learning Algorithm
$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while $!convergence$ do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
    end
end
//the algorithm converges when all the
  inputs are classified correctly
```

We initialize ***w*** *with some random vector. We then iterate over all the examples in the data, (P U N) both positive and negative examples. Now if an input* ***x*** *belongs to P, ideally what should the dot product* ***w.x*** *be? I'd say greater than or equal to 0 because that's the only thing what our perceptron wants at the end of the day so let's give it that. And if* ***x*** *belongs to N, the dot product MUST be less than 0. So if you look at the if conditions in the while loop:*



```
while $!convergence$ do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
    end
end
```

Case 1: *When* **x** *belongs to P and its dot product* ***w.x*** *< 0*
Case 2: *When* **x** *belongs to N and its dot product* ***w.x*** *≥ 0*

Only for these cases, we are updating our randomly initialized **w**. *Otherwise, we don't touch* **w** *at all because Case 1 and Case 2 are violating the very rule of a perceptron. So we are adding* ***x*** *to* ***w*** *(ahem vector addition ahem) in Case 1 and subtracting* ***x*** *from* ***w*** *in Case 2.*

Why Would The Specified Update Rule Work?

But why would this work? If you get it already why this would work, you've got the entire gist of my post and you can now move on with your life, thanks for reading, bye. But if you are not sure why these seemingly arbitrary operations of **x** *and* **w** *would help you learn that perfect* **w** *that can perfectly classify P and N, stick with me.*

We have already established that when **x** *belongs to P, we want* ***w.x*** *> 0, basic perceptron rule. What we also mean by that is that when* **x** *belongs to P, the angle between* **w** *and* **x** *should be _____ than 90 degrees. Fill in the blank.*

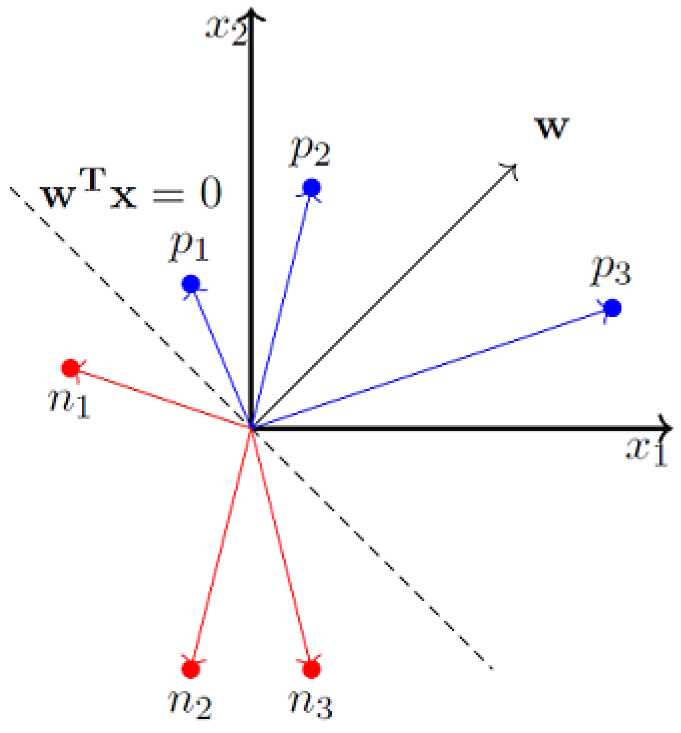
Answer: The angle between **w** *and* **x** *should be less than 90 because the cosine of the angle is proportional to the dot product.*

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||} \quad \Bigg| \quad cos\alpha \propto \mathbf{w}^T\mathbf{x}$$

$$\text{So if } \mathbf{w}^T\mathbf{x} > 0 \quad \Rightarrow cos\alpha > 0 \quad \Rightarrow \alpha < 90$$

$$\text{Similarly, if } \mathbf{w}^T\mathbf{x} < 0 \quad \Rightarrow cos\alpha < 0 \quad \Rightarrow \alpha > 90$$

So whatever the **w** *vector may be, as long as it makes an angle less than 90 degrees with the positive example data vectors (***x** *E P) and an angle more than 90 degrees with the negative example data vectors (***x** *E N), we are cool. So ideally, it should look something like this:*



x_0 is always 1 so we ignore it for now.

*So we now strongly believe that the angle between **w** and **x** should be less than 90 when **x** belongs to P class and the angle between them should be more than 90 when **x** belongs to N class. Pause and convince yourself that the above statements are true and you indeed believe them. Here's why the update works:*

$$(\alpha_{new}) \text{ when } \mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$$

$$\begin{aligned} cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T\mathbf{x} \\ &\propto (\mathbf{w}+\mathbf{x})^T\mathbf{x} \\ &\propto \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\ &\propto cos\alpha + \mathbf{x}^T\mathbf{x} \\ cos(\alpha_{new}) &> cos\alpha \end{aligned}$$

$$(\alpha_{new}) \text{ when } \mathbf{w_{new}} = \mathbf{w} - \mathbf{x}$$

$$\begin{aligned} cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T\mathbf{x} \\ &\propto (\mathbf{w}-\mathbf{x})^T\mathbf{x} \\ &\propto \mathbf{w}^T\mathbf{x} - \mathbf{x}^T\mathbf{x} \\ &\propto cos\alpha - \mathbf{x}^T\mathbf{x} \\ cos(\alpha_{new}) &< cos\alpha \end{aligned}$$

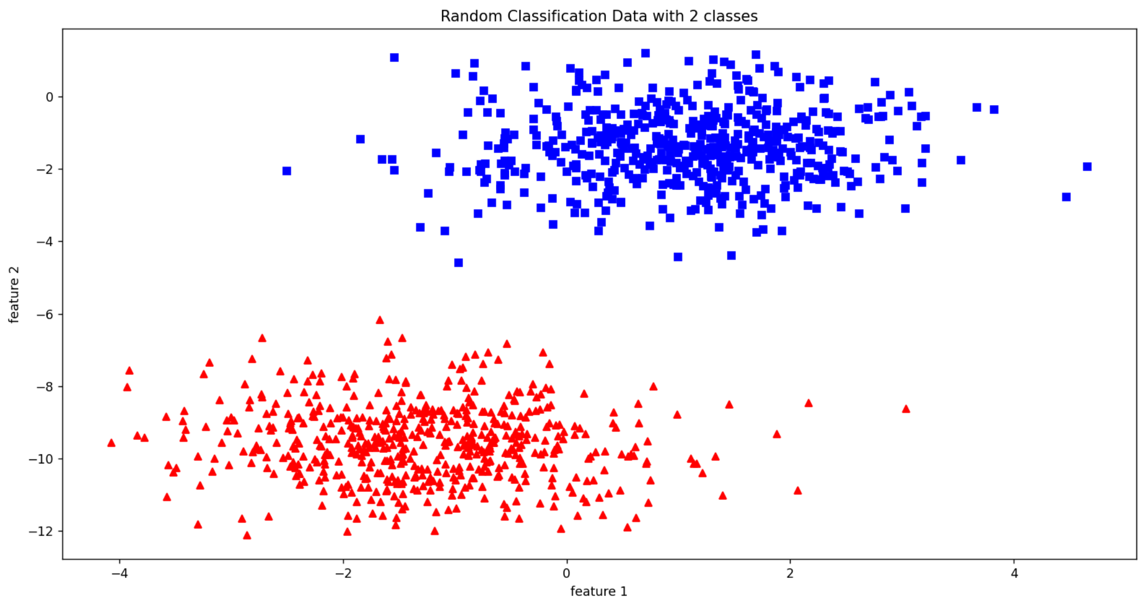
*So when we are adding **x** to **w**, which we do when x belongs to P and **w.x** < 0 (Case 1), we are essentially **increasing the cos(alpha)** value, which means, we are **decreasing the alpha value**, the angle between **w** and **x**, **which is what we desire**. And the similar intuition works for the case when **x** belongs to N and **w.x** ≥ 0 (Case 2).*

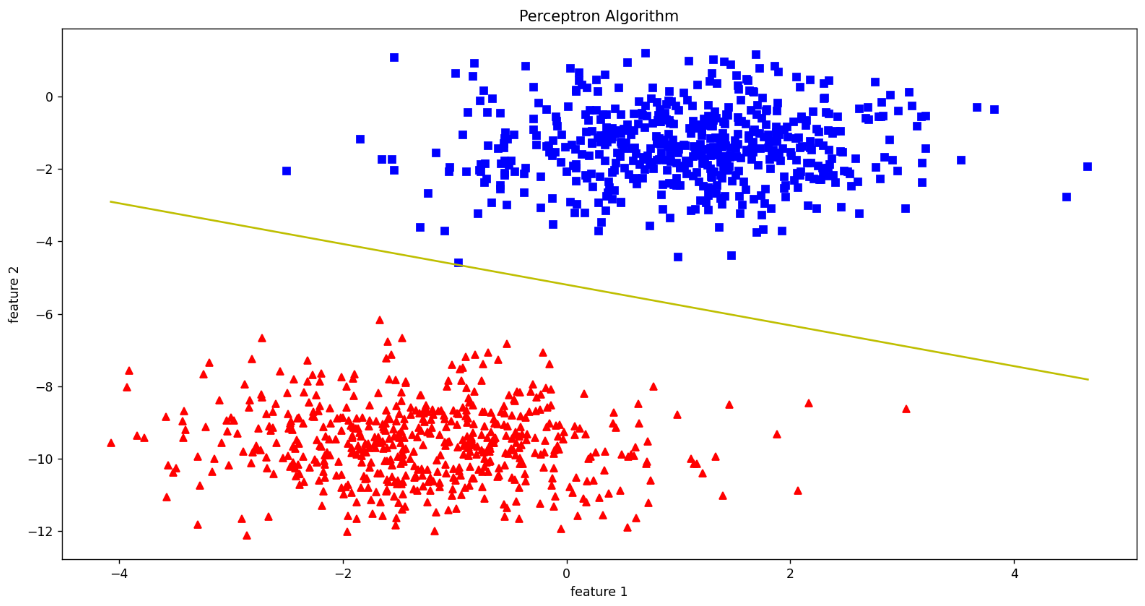
*Here's a toy simulation of how we might up end up learning **w** that makes an angle less than 90 for positive examples and more than 90 for negative examples.*

Code:

```
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt

X, y = datasets.make_blobs(n_samples=1100,n_features=2,centers=2,cluster_std=1.05,random_state=2)
#Plotting
fig = plt.figure(figsize=(10,8))
plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'r^')
plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs')
plt.xlabel("feature 1")
plt.ylabel("feature 2")
plt.title('Random Classification Data with 2 classes')

def step_func(z):
        return 1.0 if (z > 0) else 0.0

def perceptron(X, y, lr, epochs):

    # X --> Inputs.
    # y --> labels/target.
    # lr --> learning rate.
    # epochs --> Number of iterations.

    # m-> number of training examples
    # n-> number of features
    m, n = X.shape

    # Initializing parapeters(theta) to zeros.
    # +1 in n+1 for the bias term.
    theta = np.zeros((n+1,1))

    # Empty list to store how many examples were
    # misclassified at every iteration.
    n_miss_list = []

    # Training.
    for epoch in range(epochs):

        # variable to store #misclassified.
        n_miss = 0

        # looping for every example.
        for idx, x_i in enumerate(X):

            # Insering 1 for bias, X0 = 1.
            x_i = np.insert(x_i, 0, 1).reshape(-1,1)

            # Calculating prediction/hypothesis.
            y_hat = step_func(np.dot(x_i.T, theta))

            # Updating if the example is misclassified.
            if (np.squeeze(y_hat) - y[idx]) != 0:
                theta += lr*((y[idx] - y_hat)*x_i)

                # Incrementing by 1.
            n_miss += 1

        # Appending number of misclassified examples
        # at every iteration.
        n_miss_list.append(n_miss)

    return theta, n_miss_list

def plot_decision_boundary(X, theta):

    # X --> Inputs
    # theta --> parameters

    # The Line is y=mx+c
    # So, Equate mx+c = theta0.X0 + theta1.X1 + theta2.X2
    # Solving we find m and c
    x1 = [min(X[:,0]), max(X[:,0])]
    m = -theta[1]/theta[2]
    c = -theta[0]/theta[2]
    x2 = m*x1 + c

    # Plotting
    fig = plt.figure(figsize=(10,8))
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "r^")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs")
    plt.xlabel("feature 1")
    plt.ylabel("feature 2")
    plt.title("Perceptron Algorithm")
    plt.plot(x1, x2, 'y-')
    plt.show()

theta, miss_l = perceptron(X, y, 0.5, 100)
plot_decision_boundary(X, theta)
```

Output:





Conclusion:

Implemented the Perceptron Algorithm.

Practical 10

Aim: *Implement Fuzzy Inference System.*

Theory:

Fuzzy Inference System is the key unit of a fuzzy logic system having decision making as its primary work. It uses the "IF…THEN" rules along with connectors "OR" or "AND" for drawing essential decision rules.

Characteristics of Fuzzy Inference System

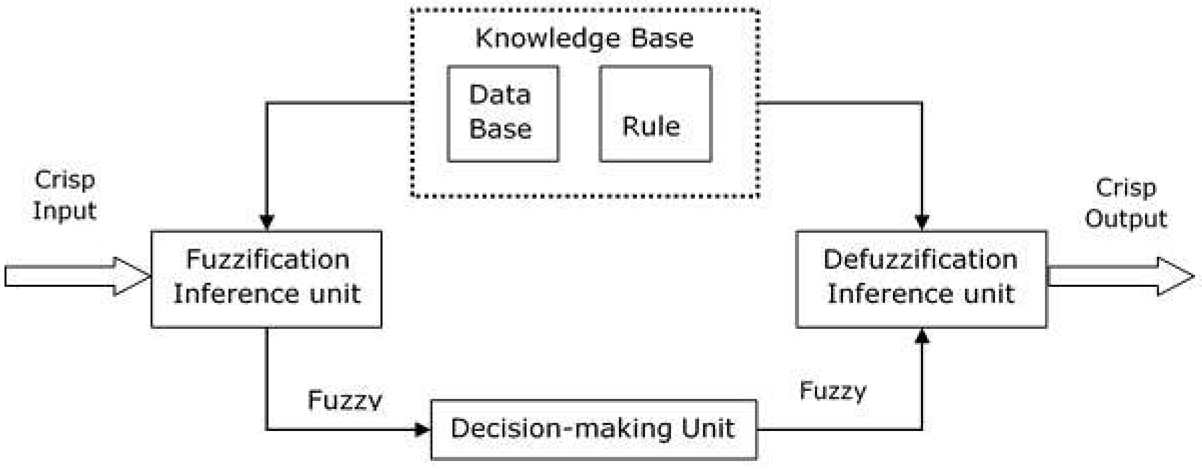
Following are some characteristics of FIS −

- *The output from FIS is always a fuzzy set irrespective of its input which can be fuzzy or crisp.*
- *It is necessary to have fuzzy output when it is used as a controller.*
- *A defuzzification unit would be there with FIS to convert fuzzy variables into crisp variables.*

Functional Blocks of FIS

The following five functional blocks will help you understand the construction of FIS −

- *Rule Base − It contains fuzzy IF-THEN rules.*
- *Database − It defines the membership functions of fuzzy sets used in fuzzy rules.*
- *Decision-making Unit − It performs operation on rules.*
- *Fuzzification Interface Unit − It converts the crisp quantities into fuzzy quantities.*
- *Defuzzification Interface Unit − It converts the fuzzy quantities into crisp quantities. Following is a block diagram of fuzzy interference system.*



Working of FIS

The working of the FIS consists of the following steps −

- *A fuzzification unit supports the application of numerous fuzzification methods, and converts the crisp input into fuzzy input.*
- *A knowledge base - collection of rule base and database is formed upon the conversion of crisp input into fuzzy input.*
- *The defuzzification unit fuzzy input is finally converted into crisp output.*

Methods of FIS

Let us now discuss the different methods of FIS. Following are the two important methods of FIS, having different consequent of fuzzy rules −

- *Mamdani Fuzzy Inference System*
- *Takagi-Sugeno Fuzzy Model (TS Method)*

Mamdani Fuzzy Inference System

This system was proposed in 1975 by Ebhasim Mamdani. Basically, it was anticipated to control a steam engine and boiler combination by synthesizing a set of fuzzy rules obtained from people working on the system.

Steps for Computing the Output

Following steps need to be followed to compute the output from this FIS −

Step 1 − *Set of fuzzy rules need to be determined in this step.*

Step 2 − *In this step, by using input membership function, the input would be made fuzzy.*

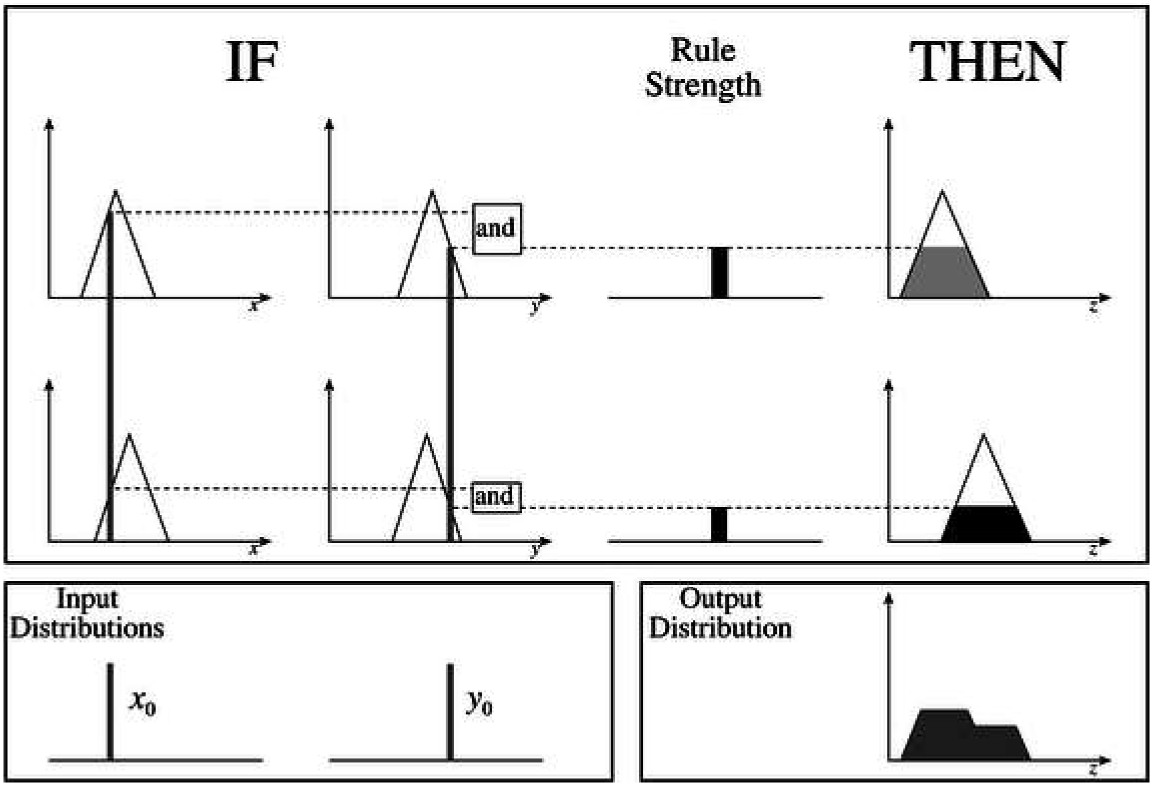
Step 3 − *Now establish the rule strength by combining the fuzzified inputs according to fuzzy rules.*

Step 4 − *In this step, determine the consequent of rule by combining the rule strength and the output membership function.*

Step 5 − *For getting output distribution combine all the consequents.*

Step 6 − *Finally, a defuzzified output distribution is obtained.*

Following is a block diagram of Mamdani Fuzzy Interface System.



Code:

```
fuzzy_clause.py
'''
Fuzzy Clause class. Used in Fuzzy rule
'''
class FuzzyClause():
    '''
    A fuzzy clause of the type 'variable is set'
    used in fuzzy IF ... THEN ... rules
    clauses can be antecedent (if part) or consequent
    (then part)
    '''


    def __init__(self, variable, f_set, degree=1):
        '''
        initialization of the fuzzy clause

        Arguments:
        ----------
        variable -- the clause variable in 'variable is set'
        set -- the clause set in 'variable is set'
        '''


        if f_set is None:
            raise Exception('set none')

        if f_set.name == '':
            raise Exception(str(f_set), 'no set name')
        self._variable = variable
        self._set = f_set

    def __str__(self):
        '''
        string representation of the clause.

        Returns:
        --------
        str: str, string representation of the clause in the form
                    A is x
        '''
        return f'{self._variable.name} is {self._set.name}'

    @property
    def variable_name(self):
        '''
        returns the name of the clause variable

        Returns:
        --------
        variable_name: str, name of variable
        '''
        return self._variable.name

    @property
    def set_name(self):
        '''
        returns the name of the clause variable

        Returns:
        --------
        variable_name: str, name of variable
        '''
        return self._set.name

    def evaluate_antecedent(self):
        '''
        Used when set is antecedent.
        returns the set degree of membership.

        Returns:
        --------
        dom -- number, the set degree of membership given a value for
                that variable. This value is determined at an earlier stage
                and stored in the set
        '''
        return self._set.last_dom_value

    def evaluate_consequent(self, dom):
        '''
        Used when clause is consequent.

        Arguments:
        -----------
        dom -- number, scalar value from the antecedent clauses
        Returns:
        --------
        set -- Type1FuzzySet, a set resulting from min operation with
                the scalar value
        '''
        self._variable.add_rule_contribution(self._set.min_scalar(dom))
```

```
fuzzy_rule.py
from fuzzy_logic.fuzzy_clause import FuzzyClause

class FuzzyRule():
    '''
    A fuzzy rule of the type
    IF [antecedent clauses] THEN [consequent clauses]
    '''

    def __init__(self):
        '''
        initializes the rule. Two data structures are necessary:
            Antecedent clauses list
            consequent clauses list
        '''
        self._antecedent = []
        self._consequent = []

    def __str__(self):
        '''
        string representation of the rule.

        Returns:
        --------
        str: str, string representation of the rule in the form
                    IF [antecedent clauses] THEN [consequent clauses]
        '''
        ante = ' and '.join(map(str, self._antecedent))
        cons = ' and '.join(map(str, self._consequent))
        return f'If {ante} then {cons}'

    def add_antecedent_clause(self, var, f_set):
        '''
        adds an antecedent clause to the rule

        Arguments:
        -----------
            clause -- FuzzyClause, the antecedent clause
        '''
        self._antecedent.append(FuzzyClause(var, f_set))

    def add_consequent_clause(self, var, f_set):
        '''
        adds an consequent clause to the rule

        Arguments:
        -----------
            clause -- FuzzyClause, the consequent clause
        '''
        self._consequent.append(FuzzyClause(var, f_set))
```

```
def evaluate(self):
    '''
    evaluation of the rule.
    the antecedent clauses are executed and the minimum degree of
    membership is retained.
    This is used in teh consequent clauses to min with the consequent
    set
    The values are returned in a dict of the form {variable_name: scalar min set, ...}

    Returns:
    --------
    rule_consequence -- dict, the resulting sets in the form
                    {variable_name: scalar min set, ...}
    '''
    # rule dom initialize to 1 as min operator will be performed
    rule_strength = 1

    # execute all antecedent clauses, keeping the minimum of the
    # returned doms to determine the rule strength
    for ante_clause in self._antecedent:
        rule_strength = min(ante_clause.evaluate_antecedent(), rule_strength)

    # execute consequent clauses, each output variable will update its output_distribution set
    for consequent_clause in self._consequent:
        consequent_clause.evaluate_consequent(rule_strength)

def evaluate_info(self):
    '''
    evaluation of the rule.
    the antecedent clauses are executed and the minimum degree of
    membership is retained.
    This is used in teh consequent clauses to min with the consequent
    set
    The values are returned in a dict of the form {variable_name: scalar min set, ...}

    Returns:
    --------
    rule_consequence -- dict, the resulting sets in the form
                    {variable_name: scalar min set, ...}
    '''
    # rule dom initialize to 1 as min operator will be performed
    rule_strength = 1


    # execute all antecedent clauses, keeping the minimum of the
    # returned doms to determine the rule strength
    for ante_clause in self._antecedent:
        rule_strength = min(ante_clause.evaluate_antecedent(), rule_strength)

    # execute consequent clauses, each output variable will update its output_distribution set
    for consequent_clause in self._consequent:
        consequent_clause.evaluate_consequent(rule_strength)

    return f'{rule_strength} : {self}'
```

```
fuzzy_set.py
import numpy as np
import copy
import matplotlib.pyplot as plt

class FuzzySet:

    _precision: int = 3

    def __init__(self, name, domain_min, domain_max, res):

        self._domain_min = domain_min
        self._domain_max = domain_max
        self._res = res

        self._domain = np.linspace(domain_min, domain_max, res)
        self._dom = np.zeros(self._domain.shape)
        self._name = name
        self._last_dom_value = 0

    def __getitem__(self, x_val):
        return self._dom[np.abs(self._domain-x_val).argmin()]

    def __setitem__(self, x_val, dom):
        self._dom[np.abs(self._domain-x_val).argmin()] = round(dom, self._precision)

    def __str__(self):
        return ' + '.join([str(a) + '/' + str(b) for a,b in zip(self._dom, self._domain)])

    def __get_last_dom_value(self):
        return self._last_dom_value

    def __set_last_dom_value(self, d):
        self._last_dom_value = d

    last_dom_value = property(__get_last_dom_value, __set_last_dom_value)

    @property
    def name(self):
        return self._name

    @property
    def empty(self):
        return np.all(self._dom == 0)

    @property
    def name(self):
        return self._name

    @classmethod
    def create_trapezoidal(cls, name, domain_min, domain_max, res, a, b, c, d):
        t1fs = cls(name, domain_min, domain_max, res)

        a = t1fs._adjust_domain_val(a)
        b = t1fs._adjust_domain_val(b)
        c = t1fs._adjust_domain_val(c)
        d = t1fs._adjust_domain_val(d)
        t1fs._dom = np.round(np.minimum(np.maximum(np.minimum((t1fs._domain-a)/(b-a), (d-
t1fs._domain)/(d-c)), 0), 1), t1fs._precision)
        return t1fs

    @classmethod
    def create_triangular(cls, name, domain_min, domain_max, res, a, b, c):
        t1fs = cls(name, domain_min, domain_max, res)

        a = t1fs._adjust_domain_val(a)
        b = t1fs._adjust_domain_val(b)
        c = t1fs._adjust_domain_val(c)

        if b == a:
            t1fs._dom = np.round(np.maximum((c-t1fs._domain)/(c-b), 0), t1fs._precision)
        elif b == c:
            t1fs._dom = np.round(np.maximum((t1fs._domain-a)/(b-a), 0), t1fs._precision)
        else:
            t1fs._dom = np.round(np.maximum(np.minimum((t1fs._domain-a)/(b-a), (c-t1fs._domain)/(c-b)),
0), t1fs._precision)

        return t1fs

    def _adjust_domain_val(self, x_val):
        return self._domain[np.abs(self._domain-x_val).argmin()]

    def clear_set(self):
        self._dom.fill(0)

    def min_scalar(self, val):

        result = FuzzySet(f'({self._name}) min ({val})', self._domain_min, self._domain_max, self._res)
        result._dom = np.minimum(self._dom, val)

        return result

    def union(self, f_set):

        result = FuzzySet(f'({self._name}) union ({f_set._name})', self._domain_min, self._domain_max,
self._res)
        result._dom = np.maximum(self._dom, f_set._dom)

        return result

    def intersection(self, f_set):

        result = FuzzySet(f'({self._name}) intersection ({f_set._name})', self._domain_min,
self._domain_max, self._res)
        result._dom = np.minimum(self._dom, f_set._dom)

        return result

    def complement(self):

        result = FuzzySet(f'not ({self._name})', self._domain_min, self._domain_max, self._res)
        result._dom = 1 - self._dom

        return result
```

```python
    def cog_defuzzify(self):

        num = np.sum(np.multiply(self._dom, self._domain))
        den = np.sum(self._dom)

        return num/den

    def domain_elements(self):
        return self._domain

    def dom_elements(self):
        return self._dom

    def plot_set(self, ax, col=''):
        ax.plot(self._domain, self._dom, col)
        ax.set_ylim([-0.1,1.1])
        ax.set_title(self._name)
        ax.grid(True, which='both', alpha=0.4)
        ax.set(xlabel='x', ylabel='$\mu(x)$')

if __name__ == "__main__":
    s = FuzzySet.create_trapezoidal('test', 1, 100, 100, 20, 30, 50, 80)

    print(s.empty)

    u = FuzzySet('u', 1, 100, 100)

    print(u.empty)

    t = FuzzySet.create_trapezoidal('test', 1, 100, 100, 30, 50, 90, 100)

    fig, axs = plt.subplots(1, 1)

    s.union(t).complement().intersection(s).min_scalar(0.2).plot_set(axs)

    plt.show()
    print(s.cog_defuzzify())
```

```python
#fuzzy_system.py
from fuzzy_logic.fuzzy_rule import FuzzyRule
from fuzzy_logic.fuzzy_variable_output import FuzzyOutputVariable
from fuzzy_logic.fuzzy_variable_input import FuzzyInputVariable

import matplotlib.pyplot as plt
from matplotlib import rc
import numpy as np

class FuzzySystem:
    '''
    A type-1 fuzzy system based on Mamdani inference system

    Reference:
    ----------
    Mamdani, Ebrahim H., and Sedrak Assilian.
    "An experiment in linguistic synthesis with a
    fuzzy logic controller." Readings in Fuzzy Sets
    for Intelligent Systems. Morgan Kaufmann, 1993. 283-289.
```

```
'''

def __init__(self):
    '''
    initializes fuzzy system.
    data structures required:
        input variables -- dict, having format {variable_name: FuzzyVariable, ...}
        output variables -- dict, having format {variable_name: FuzzyVariable, ...}
        rules -- list of FuzzyRule
        output_distribution -- dict holding fuzzy output for each variable having format
                        {variable_name: FuzzySet, ...}
    '''
    self._input_variables = {}
    self._output_variables = {}
    self._rules = []

def __str__(self):
    '''
    string representation of the system.

    Returns:
    --------
    str: str, string representation of the system in the form
            Input:
            input_variable_name(set_names)...
            Output:
            output_variable_name(set_names)...
            Rules:
            IF [antecedent clauses] THEN [consequent clauses]
    '''

    ret_str = 'Input: \n'
    for n, s in self._input_variables.items():
        ret_str = ret_str + f'{n}: ({s})\n'

    ret_str = ret_str + 'Output: \n'
    for n, s in self._output_variables.items():
        ret_str = ret_str + f'{n}: ({s})\n'

    ret_str = ret_str + 'Rules: \n'
    for rule in self._rules:
        ret_str = ret_str + f'{rule}\n'

    return ret_str

def add_input_variable(self, variable):
    '''
    adds an input variable to the system

    Arguments:
    ----------
    variable -- FuzzyVariable, the input variable
    '''
    self._input_variables[variable.name] = variable

def add_output_variable(self, variable):
    self._output_variables[variable.name] = variable
```

```
def get_input_variable(self, name):
    '''
    get an input variable given the name

    Arguments:
    -----------
    name -- str, name of variable

    Returns:
    --------
    variable -- FuzzyVariable, the input variable
    '''
    return self._input_variables[name]

def get_output_variable(self, name):
    '''
    get an output variable given the name

    Arguments:
    -----------
    name -- str, name of variable

    Returns:
    --------
    variable -- FuzzyVariable, the output variable
    '''

    return self._output_variables[name]

def _clear_output_distributions(self):
    '''
    used for each iteration. The fuzzy result is cleared
    '''
    map(lambda output_var: output_var.clear_output_distribution(), self._output_variables.values())

def add_rule(self, antecedent_clauses, consequent_clauses):
    '''
    adds a new rule to the system.
    TODO: add checks

    Arguments:
    -----------
    antecedent_clauses -- dict, having the form {variable_name:set_name, ...}
    consequent_clauses -- dict, having the form {variable_name:set_name, ...}
    '''
    # create a new rule
    # new_rule = FuzzyRule(antecedent_clauses, consequent_clauses)
    new_rule = FuzzyRule()

    for var_name, set_name in antecedent_clauses.items():
        # get variable by name
        var = self.get_input_variable(var_name)
        # get set by name
        f_set = var.get_set(set_name)
        # add clause
        new_rule.add_antecedent_clause(var, f_set)
        for var_name, set_name in consequent_clauses.items():
            var = self.get_output_variable(var_name)
            f_set = var.get_set(set_name)
            new_rule.add_consequent_clause(var, f_set)

        # add the new rule
        self._rules.append(new_rule)

    def evaluate_output(self, input_values):
        '''
        Executes the fuzzy inference system for a set of inputs

        Arguments:
        -----------
        input_values -- dict, containing the inputs to the systems in the form
                            {input_variable_name: value, ...}

        Returns:
        --------
        output -- dict, containing the outputs from the systems in the form
                    {output_variable_name: value, ...}
        '''
        # clear the fuzzy consequences as we are evaluating a new set of inputs.
        # can be optimized by comparing if the inputs have changes from the previous
        # iteration.
        self._clear_output_distributions()

        # Fuzzify the inputs. The degree of membership will be stored in
        # each set
        for input_name, input_value in input_values.items():
            self._input_variables[input_name].fuzzify(input_value)

        # evaluate rules
        for rule in self._rules:
            rule.evaluate()

        # finally, defuzzify all output distributions to get the crisp outputs
        output = {}
        for output_var_name, output_var in self._output_variables.items():
            output[output_var_name] = output_var.get_crisp_output()

        return output

    def evaluate_output_info(self, input_values):
        '''
        Executes the fuzzy inference system for a set of inputs

        Arguments:
        -----------
        input_values -- dict, containing the inputs to the systems in the form
                            {input_variable_name: value, ...}

        Returns:
        --------
        output -- dict, containing the outputs from the systems in the form
                    {output_variable_name: value, ...}
        '''
        info = {}
        # clear the fuzzy consequences as we are evaluating a new set of inputs.
        # can be optimized by comparing if the inputs have changes from the previous
        # iteration.
        self._clear_output_distributions()

        # Fuzzify the inputs. The degree of membership will be stored in
        # each set
        fuzzification_info = []
        for input_name, input_value in input_values.items():
            fuzzification_info.append(self._input_variables[input_name].fuzzify_info(input_value))

        info['fuzzification'] = '\n'.join(fuzzification_info)

        # evaluate rules
        rule_info = []
        for rule in self._rules:
            rule_info.append(rule.evaluate_info())

        info['rules'] = '\n'.join(rule_info)

        # finally, defuzzify all output distributions to get the crisp outputs
        output = {}
        for output_var_name, output_var in self._output_variables.items():
            output[output_var_name], info = output_var.get_crisp_output_info()
            # info[output_var_name] = info

        return output, info

    def plot_system(self):

        total_var_count = len(self._input_variables) + len(self._output_variables)
        if total_var_count <2:
            total_var_count = 2

        fig, axs = plt.subplots(total_var_count, 1)

        fig.tight_layout(pad=1.0)

        for idx, var_name in enumerate(self._input_variables):
            self._input_variables[var_name].plot_variable(ax=axs[idx], show=False)

        for idx, var_name in enumerate(self._output_variables):
            self._output_variables[var_name].plot_variable(ax=axs[len(self._input_variables)+idx],
show=False)

        plt.show()

if __name__ == "__main__":
    pass
```

```
fuzzy_variable_input.py
from fuzzy_logic.fuzzy_variable import FuzzyVariable

class FuzzyInputVariable(FuzzyVariable):

    def __init__(self, name, min_val, max_val, res):
        super().__init__(name, min_val, max_val, res)

    def fuzzify(self, value):
        '''
        performs fuzzification of the variable. used when the
        variable is an input one

        Arguments:
        ----------
        value -- number, input value for the variable

        '''
        # get dom for each set and store it - it will be required for each rule
        for set_name, f_set in self._sets.items():
            f_set.last_dom_value = f_set[value]

    def fuzzify_info(self, value):
        '''
        performs fuzzification of the variable. used when the
        variable is an input one

        Arguments:
        ----------
        value -- number, input value for the variable

        '''
        # get dom for each set and store it - it will be required for each rule
        for set_name, f_set in self._sets.items():
            f_set.last_dom_value = f_set[value]

        res = []

        res.append(self._name)
        res.append('\n')

        for _, f_set in self._sets.items():
            res.append(f_set.name)
            res.append(str(f_set.last_dom_value))
            res.append('\n')

        return ' '.join(res)

if __name__ == "__main__":
    pass
```

```
#fuzzy_variable_output.py
from fuzzy_logic.fuzzy_variable import FuzzyVariable
from fuzzy_logic.fuzzy_set import FuzzySet

class FuzzyOutputVariable(FuzzyVariable):

    def __init__(self, name, min_val, max_val, res):
        super().__init__(name, min_val, max_val, res)
        self._output_distribution = FuzzySet(name, min_val, max_val, res)

    def clear_output_distribution(self):
        self._output_distribution.clear_set()
    def add_rule_contribution(self, rule_consequence):
        self._output_distribution = self._output_distribution.union(rule_consequence)

    def get_crisp_output(self):
        return self._output_distribution.cog_defuzzify()

    def get_crisp_output_info(self):
        return self._output_distribution.cog_defuzzify(), self._output_distribution

if __name__ == "__main__":
    pass
```

```
#fuzzy_variable.py
from fuzzy_logic.fuzzy_set import FuzzySet
import matplotlib.pyplot as plt
import numpy as np

class FuzzyVariable():
    '''
    A type-1 fuzzy variable that is mage up of a number of type-1 fuzzy sets
    '''
    def __init__(self, name, min_val, max_val, res):
        '''
        creates a new type-1 fuzzy variable (universe)

        Arguments:
        ----------
            min_val -- number, minimum value of variable
            max_val -- number, maximum value of variable
            res -- int, resolution of variable
        '''
        self._sets={}
        self._max_val = max_val
        self._min_val = min_val
        self._res = res
        self._name = name

    def __str__(self):
        return ', '.join(self._sets.keys())

    @property
    def name(self):
        return self._name

    def _add_set(self, name, f_set):
        '''
        adds a fuzzy set to the variable

        Arguments:
        ----------
            name -- string, name of the set
            f_set -- FuzzySet, The set
        '''
        self._sets[name] = f_set

    def get_set(self, name):
        '''
        returns a set given the name
        Arguments:
        ----------
        name -- str, set name

        Returns:
        --------
        set -- FuzzySet, the set
        '''
        return self._sets[name]

    def add_triangular(self, name, low, mid, high):
        new_set = FuzzySet.create_triangular(name, self._min_val, self._max_val, self._res, low, mid, high)
        self._add_set(name, new_set)
        return new_set

    def add_trapezoidal(self, name, a, b, c, d):
        new_set = FuzzySet. create_trapezoidal(name, self._min_val, self._max_val, self._res, a, b, c, d)
        self._add_set(name, new_set)
        return new_set

    def plot_variable(self, ax=None, show=True):
        '''
        plots a graphical representation of the fuzzy variable

        Reference:
        ----------
            https://stackoverflow.com/questions/4700614/how-to-put-the-legend-out-of-the-plot
        '''
        if ax == None:
            ax = plt.subplot(111)

        for n ,s in self._sets.items():
            ax.plot(s.domain_elements(), s.dom_elements(), label=n)

        # Shrink current axis by 20%
        pos = ax.get_position()
        ax.set_position([pos.x0, pos.y0, pos.width * 0.8, pos.height])
        ax.grid(True, which='both', alpha=0.4)
        ax.set_title(self._name)
        ax.set(xlabel='x', ylabel='$\mu (x)$')

        # Put a legend to the right of the current axis
        ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

        if show:
            plt.show()
```

Driver Code:

```
from fuzzy_logic.fuzzy_variable_output import FuzzyOutputVariable
from fuzzy_logic.fuzzy_variable_input import FuzzyInputVariable
# from fuzzy_system.fuzzy_variable import FuzzyVariable
from fuzzy_logic.fuzzy_system import FuzzySystem

temp = FuzzyInputVariable('Temperature', 10, 40, 100)
```

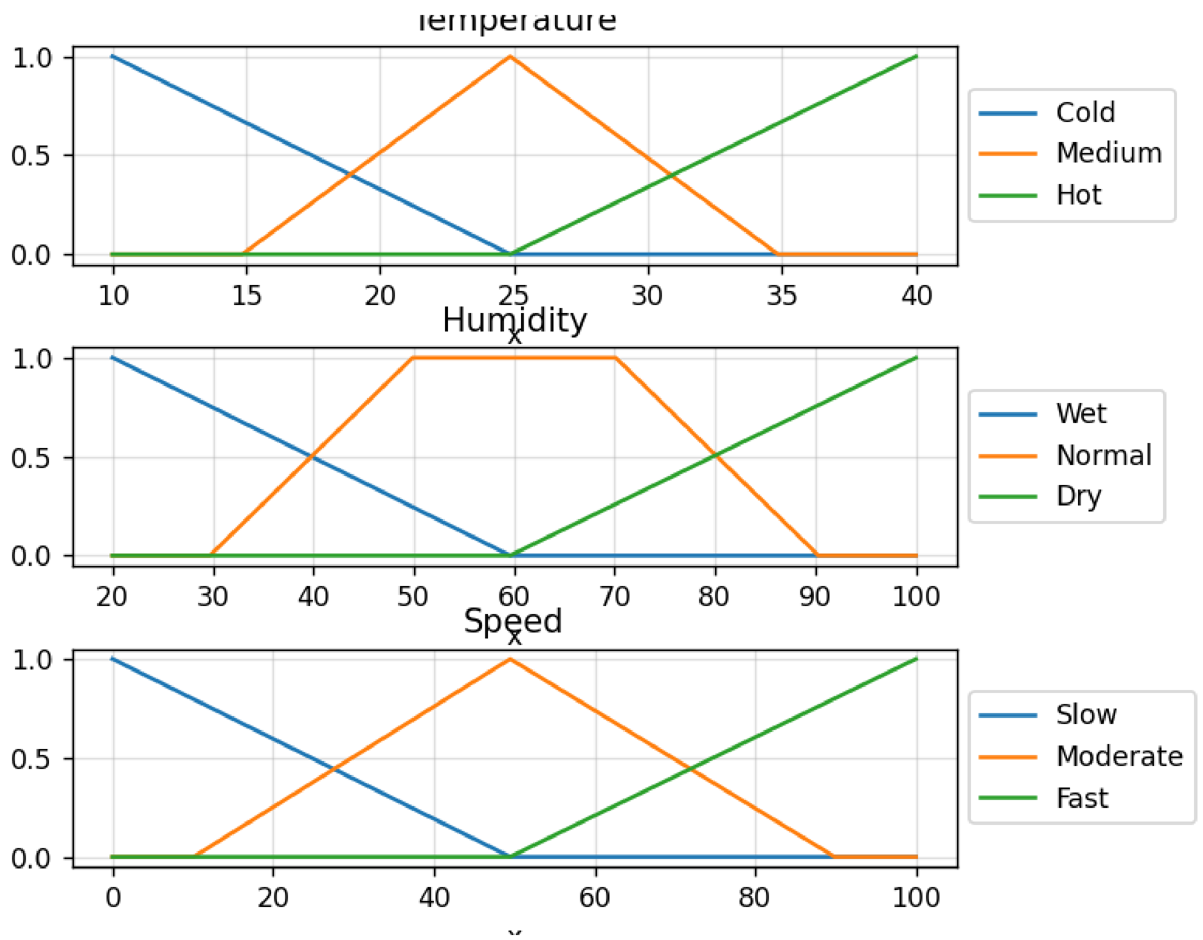
```
temp.add_triangular('Cold', 10, 10, 25)
temp.add_triangular('Medium', 15, 25, 35)
temp.add_triangular('Hot', 25, 40, 40)

humidity = FuzzyInputVariable('Humidity', 20, 100, 100)
humidity.add_triangular('Wet', 20, 20, 60)
humidity.add_trapezoidal('Normal', 30, 50, 70, 90)
humidity.add_triangular('Dry', 60, 100, 100)

motor_speed = FuzzyOutputVariable('Speed', 0, 100, 100)
motor_speed.add_triangular('Slow', 0, 0, 50)
motor_speed.add_triangular('Moderate', 10, 50, 90)
motor_speed.add_triangular('Fast', 50, 100, 100)

system = FuzzySystem()
system.add_input_variable(temp)
system.add_input_variable(humidity)
system.add_output_variable(motor_speed)

system.add_rule(
        { 'Temperature':'Cold',
          'Humidity':'Wet' },
        { 'Speed':'Slow'})

system.add_rule(
        { 'Temperature':'Cold',
          'Humidity':'Normal' },
        { 'Speed':'Slow'})

system.add_rule(
        { 'Temperature':'Medium',
          'Humidity':'Wet' },
        { 'Speed':'Slow'})

system.add_rule(
        { 'Temperature':'Medium',
          'Humidity':'Normal' },
        { 'Speed':'Moderate'})

system.add_rule(
        { 'Temperature':'Cold',
          'Humidity':'Dry' },
        { 'Speed':'Moderate'})

system.add_rule(
        { 'Temperature':'Hot',
          'Humidity':'Wet' },
        { 'Speed':'Moderate'})

system.add_rule(
        { 'Temperature':'Hot',
          'Humidity':'Normal' },
        { 'Speed':'Fast'})

system.add_rule(
        { 'Temperature':'Hot',
          'Humidity':'Dry' },
        { 'Speed':'Fast'})

system.add_rule(
        { 'Temperature':'Medium',
           'Humidity':'Dry' },
        { 'Speed':'Fast'})

output = system.evaluate_output({
                'Temperature':18,
                'Humidity':60
        })

print(output)
# print('fuzzification\n-------------\n', info['fuzzification'])
# print('rules\n-----\n', info['rules'])

system.plot_system()
```

Driver code- 2in 2out

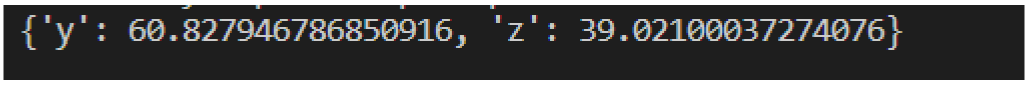
```
from fuzzy_logic.fuzzy_variable_output import FuzzyOutputVariable
from fuzzy_logic.fuzzy_variable_input import FuzzyInputVariable

from fuzzy_logic.fuzzy_system import FuzzySystem

x1 = FuzzyInputVariable('x1', 0, 100, 100)
x1.add_triangular('S', 0, 25, 50)
x1.add_triangular('M', 25, 50, 75)
x1.add_triangular('L', 50, 75, 100)

x2 = FuzzyInputVariable('x2', 0, 100, 100)
x2.add_triangular('S', 0, 25, 50)
x2.add_triangular('M', 25, 50, 75)
x2.add_triangular('L', 50, 75, 100)

y = FuzzyOutputVariable('y', 0, 100, 100)
y.add_triangular('S', 0, 25, 50)
y.add_triangular('M', 25, 50, 75)
y.add_triangular('L', 50, 75, 100)

z = FuzzyOutputVariable('z', 0, 100, 100)
z.add_triangular('S', 0, 25, 50)
z.add_triangular('M', 25, 50, 75)
z.add_triangular('L', 50, 75, 100)

system = FuzzySystem()
system.add_input_variable(x1)
system.add_input_variable(x2)
system.add_output_variable(y)
system.add_output_variable(z)

system.add_rule(
        { 'x1':'S',
           'x2':'S' },
        { 'y':'S',
           'z':'L' })
```

```
system.add_rule(
        { 'x1':'M',
           'x2':'M' },
        { 'y':'M',
           'z':'M' })

system.add_rule(
        { 'x1':'L',
           'x2':'L' },
        { 'y':'L',
           'z':'S' })

system.add_rule(
        { 'x1':'S',
           'x2':'M' },
        { 'y':'S',
           'z':'L' })

system.add_rule(
        { 'x1':'M',
           'x2':'S' },
        { 'y':'S',
           'z':'L' })

system.add_rule(
        { 'x1':'L',
           'x2':'M' },
        { 'y':'L',
           'z':'S' })

system.add_rule(
        { 'x1':'M',
           'x2':'L' },
        { 'y':'L',
           'z':'S' })

system.add_rule(
        { 'x1':'L',
           'x2':'S' },
        { 'y':'M',
           'z':'M' })

system.add_rule(
        { 'x1':'S',
           'x2':'L' },
        { 'y':'M',
           'z':'M' })

output = system.evaluate_output({
           'x1':35,
           'x2':75
        })

print(output)
```

Output:



2in_2out



```
{'y': 60.827946786850916, 'z': 39.02100037274076}
```

Conclusion:

Implemented Fuzzy Inference System.

Practical 11

Aim: *Solve Fuzzy Control Systems: The Tipping Problem.*

Theory:

Fuzzy Control Systems: The Tipping Problem

The 'tipping problem' is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

If you're new to the world of fuzzy control systems, you might want to check out the Fuzzy Control Primer before reading through this worked example.

The Tipping Problem

Let's create a fuzzy control system which models how you might choose to tip at a restaurant. When tipping, you consider the service and food quality, rated between 0 and 10. You use this to leave a tip of between 0 and 25%.

We would formulate this problem as:

- ***Antecednets (Inputs)***
  - ***service***
    - *Universe (ie, crisp value range): How good was the service of the wait staff, on a scale of 0 to 10?*
    - *Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing*
  - ***food quality***
    - *Universe: How tasty was the food, on a scale of 0 to 10?*
    - *Fuzzy set: bad, decent, great*
- ***Consequents (Outputs)***
  - ***tip***
    - *Universe: How much should we tip, on a scale of 0% to 25%*
    - *Fuzzy set: low, medium, high*
- ***Rules***
  - *IF the service was good or the food quality was good, THEN the tip will be high.*
  - *IF the service was average, THEN the tip will be medium.*
  - *IF the service was poor and the food quality was poor THEN the tip will be low.*
- ***Usage***
  - ***If I tell this controller that I rated:***

        - *the service as 9.8, and*
        - *the quality as 6.5,*
    - ***it would recommend I leave:***
        - *a 20.2% tip.*

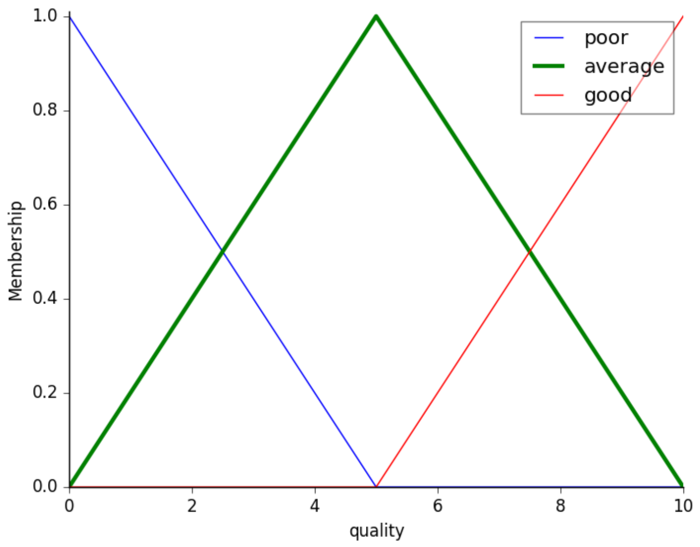
Creating the Tipping Controller Using the skfuzzy control API

We can use the skfuzzy control system API to model this. First, let's define fuzzy variables

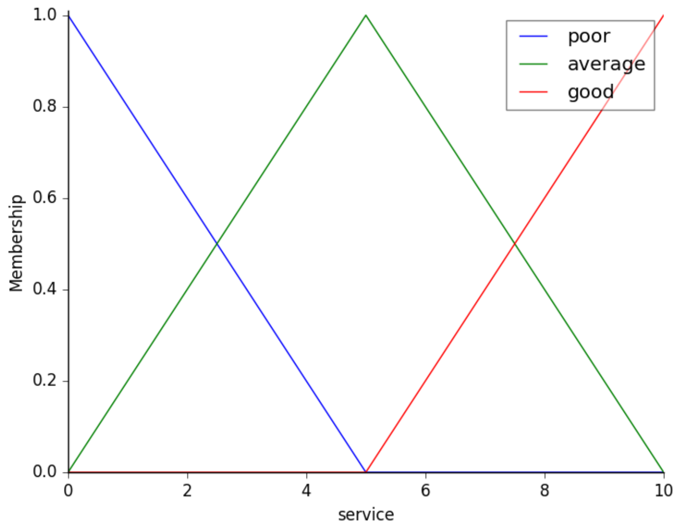
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,
# Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
```

# You can see how these look with .view()

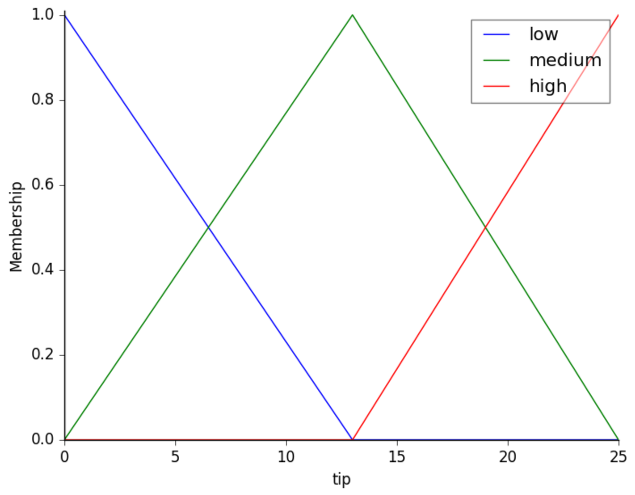
```
quality['average'].view()
```



```
service.view()
```



```
tip.view()
```



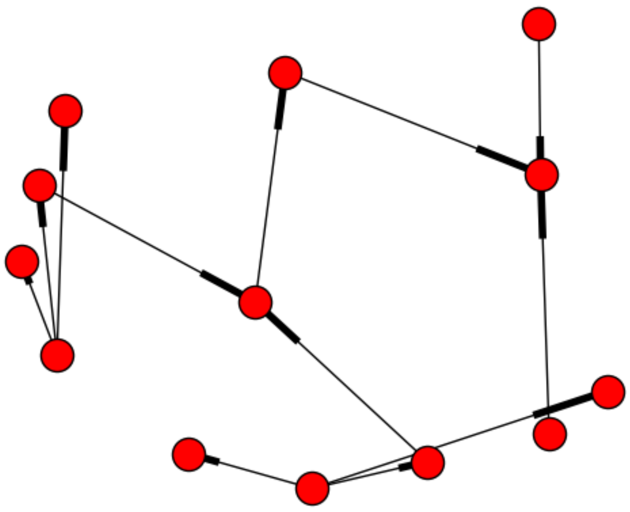
Fuzzy rules

Now, to make these triangles useful, we define the fuzzy relationship between input and output variables. For the purposes of our example, consider three simple rules:

1. *If the food is poor OR the service is poor, then the tip will be low*
2. *If the service is average, then the tip will be medium*
3. *If the food is good OR the service is good, then the tip will be high.*

Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

```
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()
```



Control System Creation and Simulation

Now that we have our rules defined, we can simply create a control system via:

```
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
```

In order to simulate this control system, we will create a ControlSystemSimulation. Think of this object representing our controller applied to a specific set of cirucmstances. For tipping, this might be tipping Sharon at the local brew-pub. We would create another ControlSystemSimulation when we're trying to apply our tipping_ctrl for Travis at the cafe because the inputs would be different.

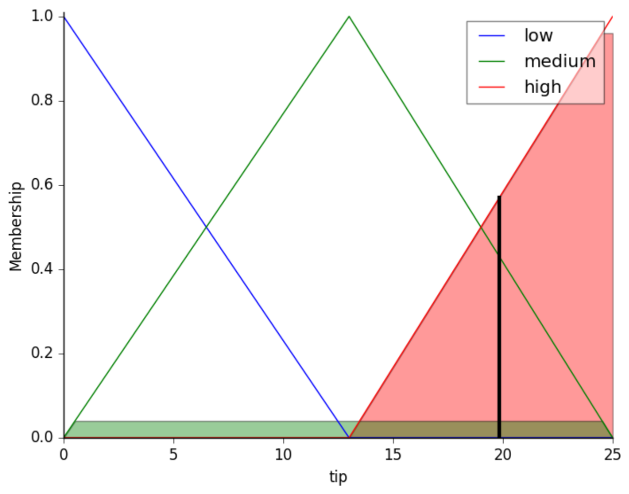
```
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
```

We can now simulate our control system by simply specifying the inputs and calling the compute method. Suppose we rated the quality 6.5 out of 10 and the service 9.8 of 10.

```
# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers
tipping.compute()
```

Once computed, we can view the result as well as visualize it.

```
print tipping.output['tip']
tip.view(sim=tipping)
```



*The resulting suggested tip is **20.24%**.*

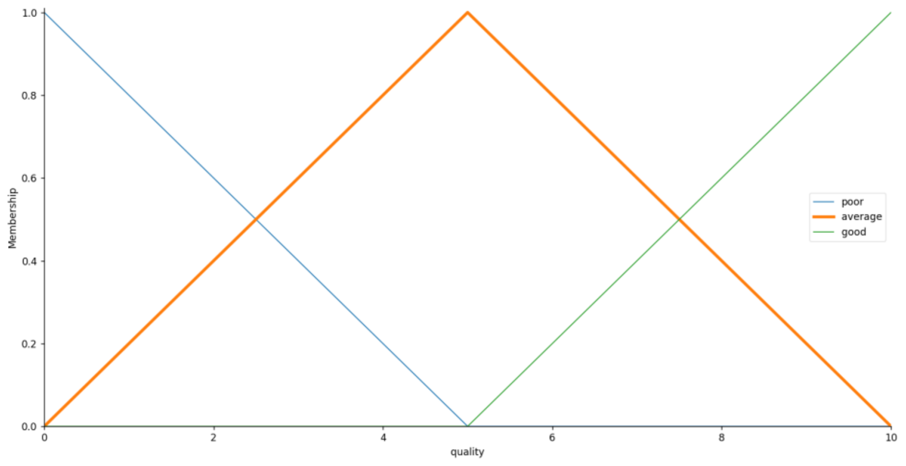
The power of fuzzy systems is allowing complicated, intuitive behavior based on a sparse system of rules with minimal overhead. Note our membership function universes were coarse, only defined at the integers, but fuzz.interp_membership allowed the effective resolution to increase on demand. This system can respond to arbitrarily small changes in inputs, and the processing burden is minimal.

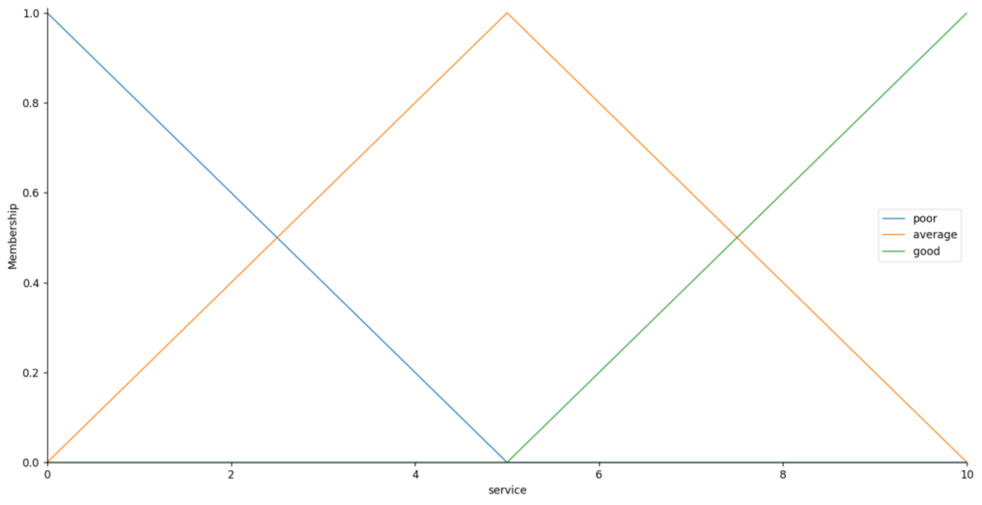
Code:

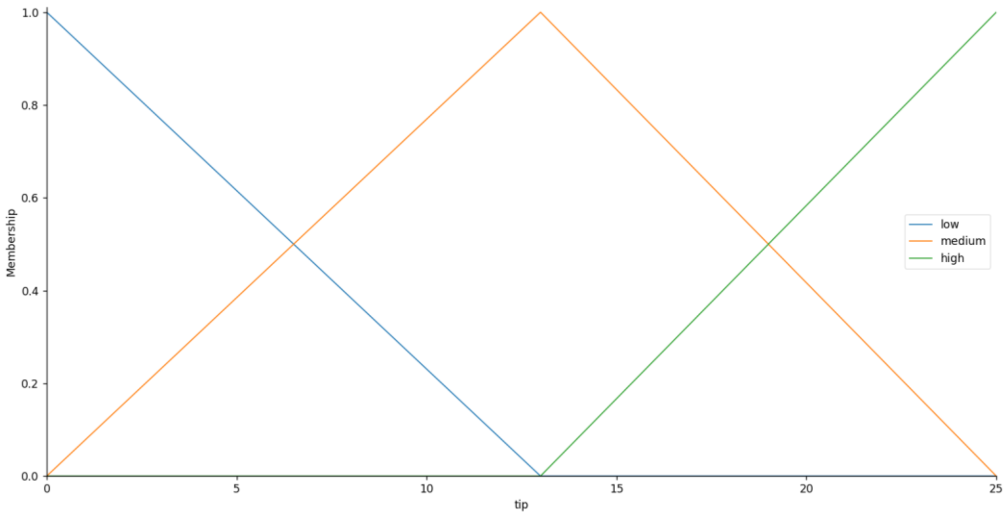
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt
# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,
# Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

# You can see how these look with .view()
quality['average'].view()

service.view()
tip.view()

rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])
```

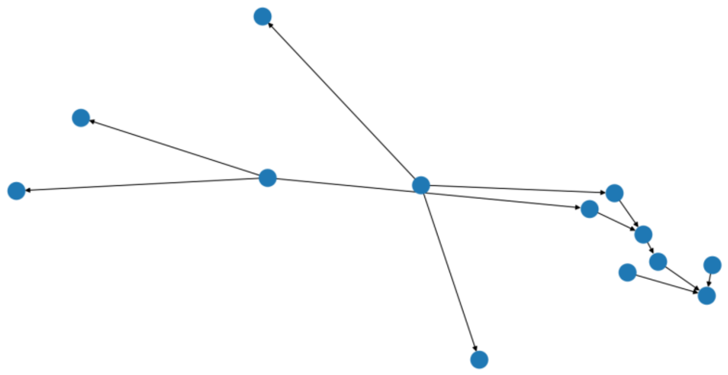
```
rule1.view()
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers
tipping.compute()

print(tipping.output['tip'])
tip.view(sim=tipping)

plt.show()
```

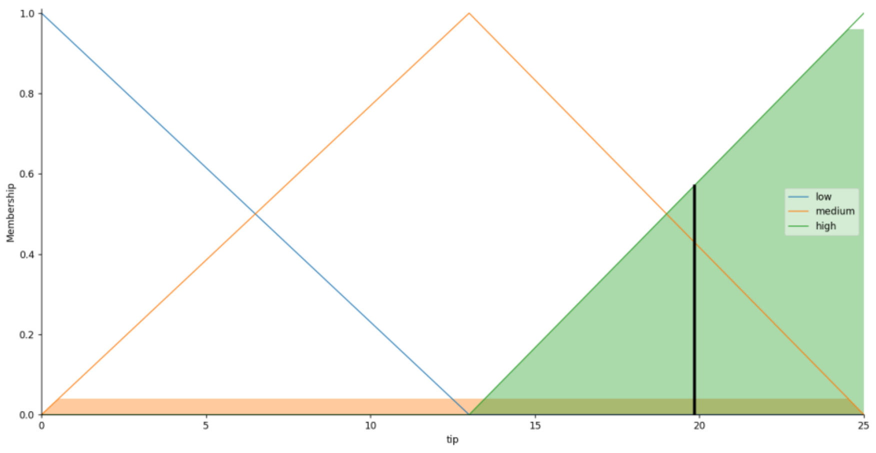
Output:











Conclusion:

Solved Fuzzy Control Systems: The Tipping Problem.

Practical 12

Aim: *Implement Naive Bayes' learning algorithm for the restaurant waiting problem.*

Theory:

Naïve Bayes Classifier Algorithm

- *Naïve Bayes algorithm is a supervised learning algorithm, which is based on* ***Bayes theorem*** *and used for solving classification problems.*
- *It is mainly used in text classification that includes a high-dimensional training dataset.*
- *Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.*
- *It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.*
- *Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.*

Why is it called Naïve Bayes?

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

- ***Naïve****: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.*
- ***Bayes****: It is called Bayes because it depends on the principle of Bayes' Theorem.*

Bayes' Theorem:

- *Bayes' theorem is also known as* ***Bayes' Rule*** *or* ***Bayes' law****, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.*
- *The formula for Bayes' theorem is given as:*

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where,

P(A|B) is Posterior probability*: Probability of hypothesis A on the observed event B.*

***P(B|A) is Likelihood probability**: Probability of the evidence given that the probability of a hypothesis is true.*

***P(A) is Prior Probability**: Probability of hypothesis before observing the evidence.*

***P(B) is Marginal Probability**: Probability of Evidence.*

Working of Naïve Bayes' Classifier:

Working of Naïve Bayes' Classifier can be understood with the help of the below example:

Suppose we have a dataset of **weather conditions** *and corresponding target variable "***Play***". So using this dataset we need to decide that whether we should play or not on a particular day according to the weather conditions. So to solve this problem, we need to follow the below steps:*

1. *Convert the given dataset into frequency tables.*
2. *Generate Likelihood table by finding the probabilities of given features.*
3. *Now, use Bayes theorem to calculate the posterior probability.*

***Problem**: If the weather is sunny, then the Player should play or not?*

***Solution**: To solve this, first consider the below dataset:*

	Outlook	***Play***
0	*Rainy*	*Yes*
1	*Sunny*	*Yes*
2	*Overcast*	*Yes*
3	*Overcast*	*Yes*
4	*Sunny*	*No*
5	*Rainy*	*Yes*
6	*Sunny*	*Yes*
7	*Overcast*	*Yes*
8	*Rainy*	*No*

9	*Sunny*	*No*
10	*Sunny*	*Yes*
11	*Rainy*	*No*
12	*Overcast*	*Yes*
13	*Overcast*	*Yes*

Frequency table for the Weather Conditions:

Weather	*Yes*	*No*
Overcast	*5*	*0*
Rainy	*2*	*2*
Sunny	*3*	*2*
Total	*10*	*5*

Likelihood table weather condition:

Weather	*No*	*Yes*	
Overcast	*0*	*5*	*5/14= 0.35*
Rainy	*2*	*2*	*4/14=0.29*
Sunny	*2*	*3*	*5/14=0.35*
All	*4/14=0.29*	*10/14=0.71*	

Applying Bayes'theorem:

P(Yes|Sunny)= P(Sunny|Yes)*P(Yes)/P(Sunny)

P(Sunny|Yes)= 3/10= 0.3

P(Sunny)= 0.35

P(Yes)=0.71

*So P(Yes|Sunny) = 0.3*0.71/0.35=* ***0.60***

P(No|Sunny)= P(Sunny|No)*P(No)/P(Sunny)

P(Sunny|NO)= 2/4=0.5

P(No)= 0.29

P(Sunny)= 0.35

*So P(No|Sunny)= 0.5*0.29/0.35 =* ***0.41***

So as we can see from the above calculation that ***P(Yes|Sunny)>P(No|Sunny)***

Hence on a Sunny day, Player can play the game.

Advantages of Naïve Bayes Classifier:

- *Naïve Bayes is one of the fast and easy ML algorithms to predict a class of datasets.*
- *It can be used for Binary as well as Multi-class Classifications.*
- *It performs well in Multi-class predictions as compared to the other Algorithms.*
- *It is the most popular choice for* ***text classification problems****.*

Disadvantages of Naïve Bayes Classifier:

- *Naive Bayes assumes that all features are independent or unrelated, so it cannot learn the relationship between features.*

Applications of Naïve Bayes Classifier:

- *It is used for* ***Credit Scoring****.*
- *It is used in* ***medical data classification****.*
- *It can be used in* ***real-time predictions*** *because Naïve Bayes Classifier is an eager learner.*
- *It is used in Text classification such as* ***Spam filtering*** *and* ***Sentiment analysis****.*

Types of Naïve Bayes Model:

There are three types of Naive Bayes Model, which are given below:

- ***Gaussian****: The Gaussian model assumes that features follow a normal distribution. This means if predictors take continuous values instead of discrete, then the model assumes that these values are sampled from the Gaussian distribution.*
- ***Multinomial****: The Multinomial Naïve Bayes classifier is used when the data is multinomial distributed. It is primarily used for document classification problems, it means a particular document belongs to which category such as Sports, Politics, education, etc.*
  The classifier uses the frequency of words for the predictors.

- ***Bernoulli****: The Bernoulli classifier works similar to the Multinomial classifier, but the predictor variables are the independent Booleans variables. Such as if a particular word is present or not in a document. This model is also famous for document classification tasks.*

Code:

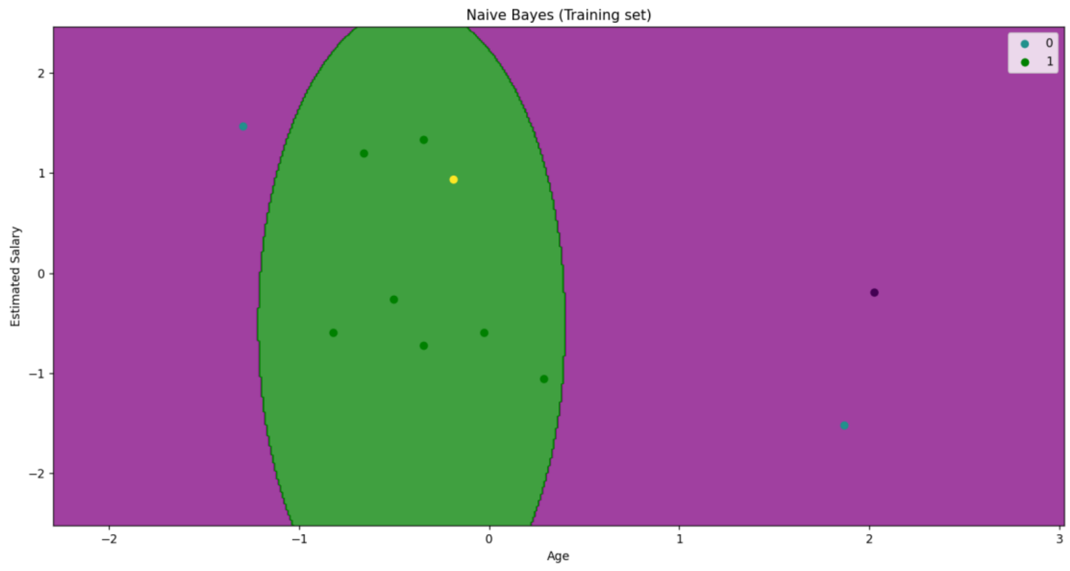
```
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('user_data.csv')
x = dataset.iloc[:, [2, 3]].values
y = dataset.iloc[:, 4].values

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)

from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(x_train, y_train)
# Predicting the Test set results
y_pred = classifier.predict(x_test)

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Visualising the Training set results
from matplotlib.colors import ListedColormap
x_set, y_set = x_train, y_train
X1, X2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step = 0.01),
                     nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
mtp.contourf(X1, X2, classifier.predict(nm.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('purple', 'green')))
mtp.xlim(X1.min(), X1.max())
mtp.ylim(X2.min(), X2.max())
for i, j in enumerate(nm.unique(y_set)):
    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],  c = ListedColormap(('purple', 'green'))(i), label =
j)
mtp.title('Naive Bayes (Training set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
x_set, y_set = x_test, y_test
```

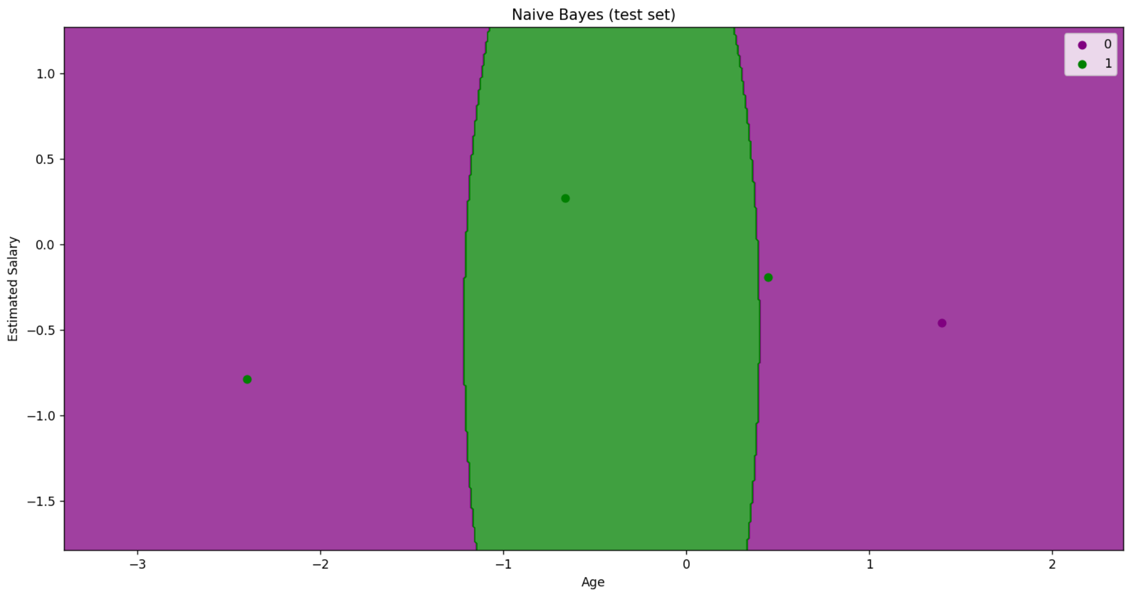
```
X1, X2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step = 0.01),
                     nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
mtp.contourf(X1, X2, classifier.predict(nm.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('purple', 'green')))
mtp.xlim(X1.min(), X1.max())
mtp.ylim(X2.min(), X2.max())
for i, j in enumerate(nm.unique(y_set)):
    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
                c = ListedColormap(('purple', 'green'))(i), label = j)
mtp.title('Naive Bayes (test set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
```

user_data.csv

```
User ID,Gender,Age,Estimated Salary,Purchased
15622,Male,36,15000,1
15669,Female,39,21000,1
15678,Male,31,15000,1
15622,Female,28,46000,0
15641,Female,32,42000,1
15647,Male,35,38000,0
15658,Female,32,28000,1
15650,Female,38,8000,1
15652,Male,21,12000,1
15618,Female,45,17000,0
15682,Female,34,13000,1
15696,Male,34,44000,1
15682,Male,33,20000,1
15650,Female,49,21000,0
15609,Male,48,1000,0
```

Output:





Conclusion:

Implemented Naive Bayes' learning algorithm for the restaurant waiting problem.