SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

FACULTY OF ENGINEERING & TECHNOLOGY

(Formerly SRM University, Under section 3 of UGC Act, 1956)

S.R.M. NAGAR, KATTANKULATHUR –603 203,
KANCHEEPURAM DISTRICT

SCHOOL OF COMPUTING
DEPARTMENT OF COMPUTING TECHNOLOGIES

Course Code: 18CSE305J

Course Name: Artificial Intelligence

Lab Report

ARYAN ATUL

RA2011003010844

CSE - C2

ARYAN ATUL
RA2011003010844
CSE - C2

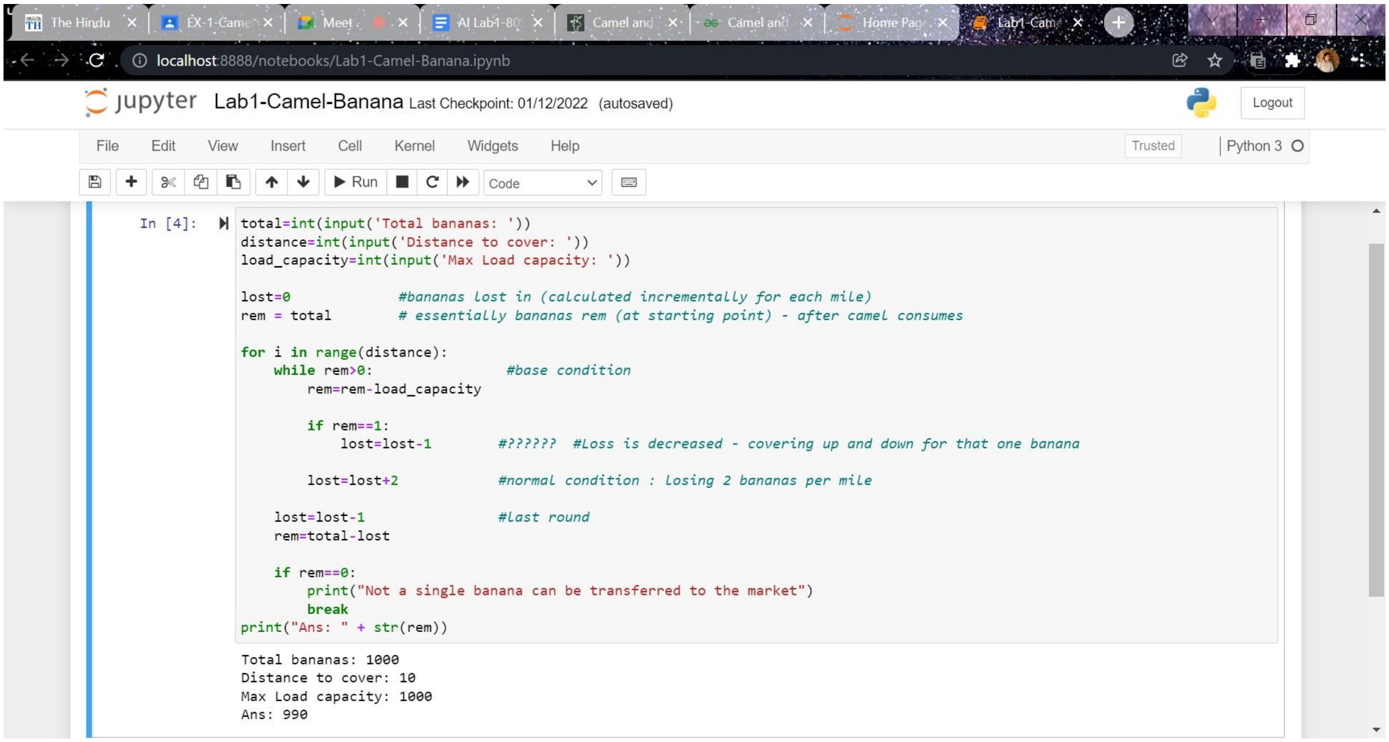
Lab-1 : Camel Banana Problem

Problem : A person wants to transfer bananas over to a destination A km away. He initially has B bananas and a camel. The camel cannot carry more than C bananas at a time and eats a banana every km it travels. Given three integers A, B, and C, the task is to find the maximum number of bananas the person can transfer to the destination using the camel.

Source-Code

```
total=int(input('Total bananas: '))      #taking inputs A, B, C
distance=int(input('Distance to cover: '))
load_capacity=int(input('Max Load capacity: '))

lost=0        #bananas lost in (calculated incrementally for each mile)
rem = total      # essentially bananas rem (at starting point) - after camel consumes

for i in range(distance):
  while rem>0:          #base condition
    rem=rem-load_capacity

    if rem==1:
      lost=lost-1     #Loss is decreased - covering up and down for that one banana

    lost=lost+2       #normal condition : losing 2 bananas per mile

  lost=lost-1         #last round
  rem=total-lost

  if rem==0:
    print("Not a single banana can be transferred to the market")
    break
print("Ans: " + str(rem))
```

Screenshot

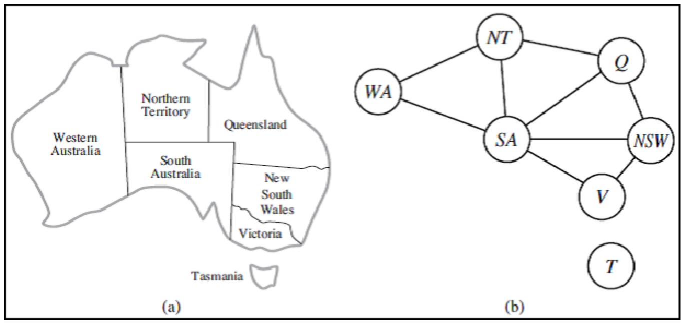


Conclusion : Hence the camel banana problem has been successfully resolved !

ARYAN ATUL
RA2011003010844
CSE - C2

Lab-2 Map-Colouring Problem (CSP)

Problem : To color the regions of the given map such that no two adjacent states have the same color. The states are the variables and the colors are the domains.



We will convert this CSP to a graph coloring problem. Depth first search is apt as the path by which solution should be reached is irrelevant.

Source Code:

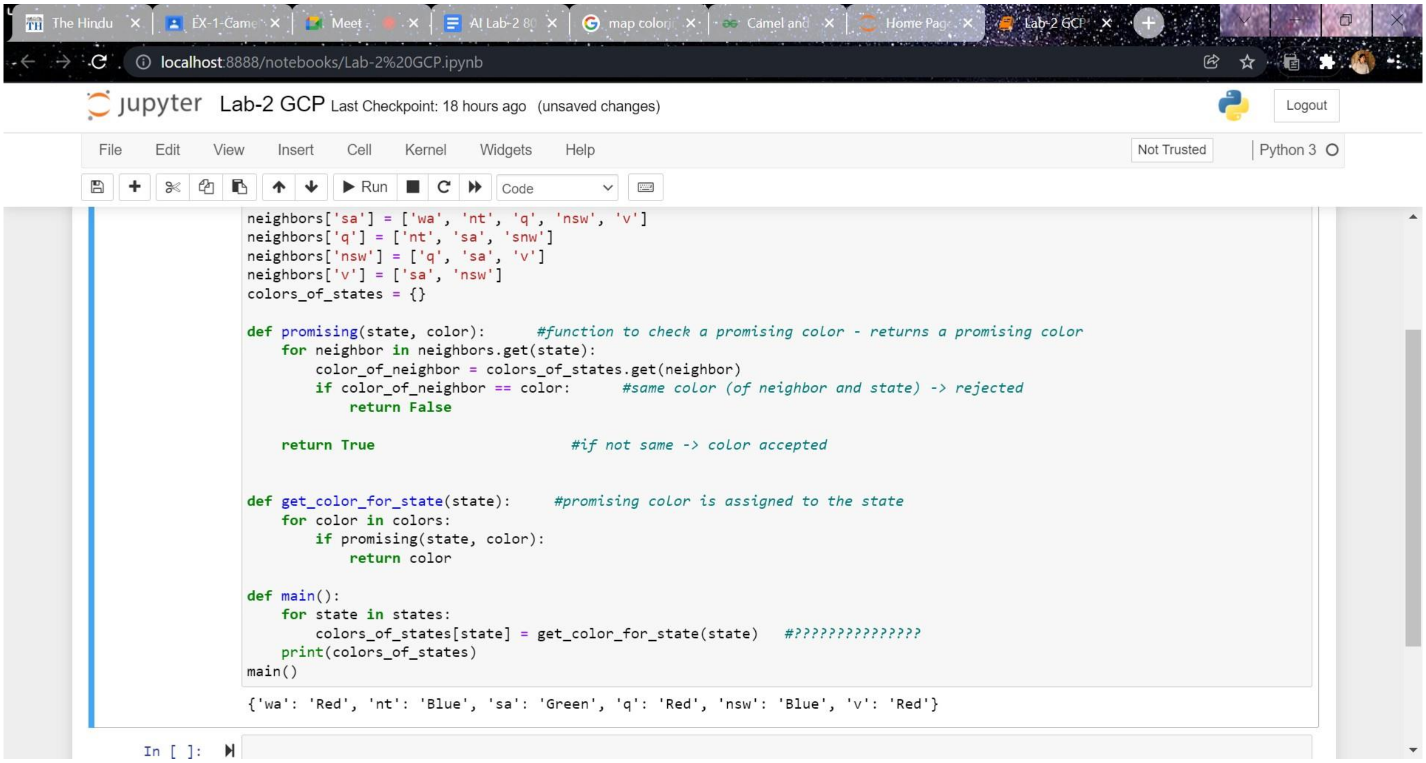
```
colors = ['Red', 'Blue', 'Green']

states = ['wa', 'nt', 'sa', 'q', 'nsw', 'v']

neighbors = {}                    #adjacent pairing neighbors of different states
neighbors['wa'] = ['nt', 'sa']
neighbors['nt'] = ['wa', 'sa', 'q']
neighbors['sa'] = ['wa', 'nt', 'q', 'nsw', 'v']
neighbors['q'] = ['nt', 'sa', 'snw']
neighbors['nsw'] = ['q', 'sa', 'v']
neighbors['v'] = ['sa', 'nsw']


colors_of_states = {}
```

```
def promising(state, color):      #function to check a promising color - returns a promising color
   for neighbor in neighbors.get(state):
      color_of_neighbor = colors_of_states.get(neighbor)
      if color_of_neighbor == color:     #same color (of neighbor and state) -> rejected
         return False

   return True                  #if not same -> color accepted


def   get_color_for_state(state):     #promising color is assigned to the state
   for color in colors:
      if promising(state, color):
         return color

def main():
   for state in states:
      colors_of_states[state]  =  get_color_for_state(state)

   print(colors_of_states)


main()
```

Result :

```
{'wa': 'Red', 'nt': 'Blue', 'sa': 'Green', 'q': 'Red', 'nsw': 'Blue', 'v': 'Red'}
```

Output Screenshot



```
neighbors['sa'] = ['wa', 'nt', 'q', 'nsw', 'v']
neighbors['q'] = ['nt', 'sa', 'snw']
neighbors['nsw'] = ['q', 'sa', 'v']
neighbors['v'] = ['sa', 'nsw']
colors_of_states = {}

def promising(state, color):      #function to check a promising color - returns a promising color
    for neighbor in neighbors.get(state):
        color_of_neighbor = colors_of_states.get(neighbor)
        if color_of_neighbor == color:      #same color (of neighbor and state) -> rejected
            return False

    return True                      #if not same -> color accepted


def get_color_for_state(state):     #promising color is assigned to the state
    for color in colors:
        if promising(state, color):
            return color

def main():
    for state in states:
        colors_of_states[state] = get_color_for_state(state)   #???????????????
    print(colors_of_states)
main()
```

```
{'wa': 'Red', 'nt': 'Blue', 'sa': 'Green', 'q': 'Red', 'nsw': 'Blue', 'v': 'Red'}
```

Conclusion : Hence the graph coloring problem has been successfully resolved.

ARYAN ATUL
RA2011003010844
CSE - C2 1.2.22

Lab-3 Cryptarithmetic Puzzle

Problem : To find the value of the letters between 0-9 given in the operation of addition and under the assumption that a maximum carry of 1 is allowed

Source Code:

```
import re

solved = False

def solve(letters, values, visited, words):
  global solved
  if len(set) == len(values):
    map = {}
    for letter, val in zip(letters,values):
      map[letter] = val

    if map[words[0][0]] == 0 or map[words[1][0]] == 0 or map[words[2][0]] == 0:
      return

    word1, word2, res = "", "", ""
    for c in words[0]:
      word1 += str(map[c])
    for c in words[1]:
      word2 += str(map[c])
    for c in words[2]:
      res += str(map[c])

    if int(word1) + int(word2) == int(res):
      print("{} + {} = {}\t{}".format(word1, word2, res, map))
      solved = True

    return
```

```python
  for i in range(10):
    if not visited[i]:
      visited[i] = True
      values.append(i)
      solve(letters, values, visited, words)
      values.pop()
      visited[i] = False


print("\nCRYPTARITHMETIC PUZZLE SOLVER")
print("WORD1 + WORD2 = RESULT")
word1 = input("Enter WORD1: ").upper()
word2 = input("Enter WORD2: ").upper()
result = input("Enter RESULT: ").upper()

if len(result) > (max(len(word1), len(word2)) + 1):
  print("\n0 Solutions!")
else:
  set = []
  for c in word1:
    if c not in set:
      set.append(c)
  for c in word2:
    if c not in set:
      set.append(c)
  for c in result:
    if c not in set:
      set.append(c)

  if len(set) > 10:
    print("\nNo solutions!")
    exit()

  print("Solutions:")
  solve(set, [], [False for _ in range(10)], [word1, word2, result])

  if not solved:
    print("\n0 solutions!")
```

jupyter Lab3-Cryptarithmetic Problem Last Checkpoint: 4 minutes ago (autosaved)

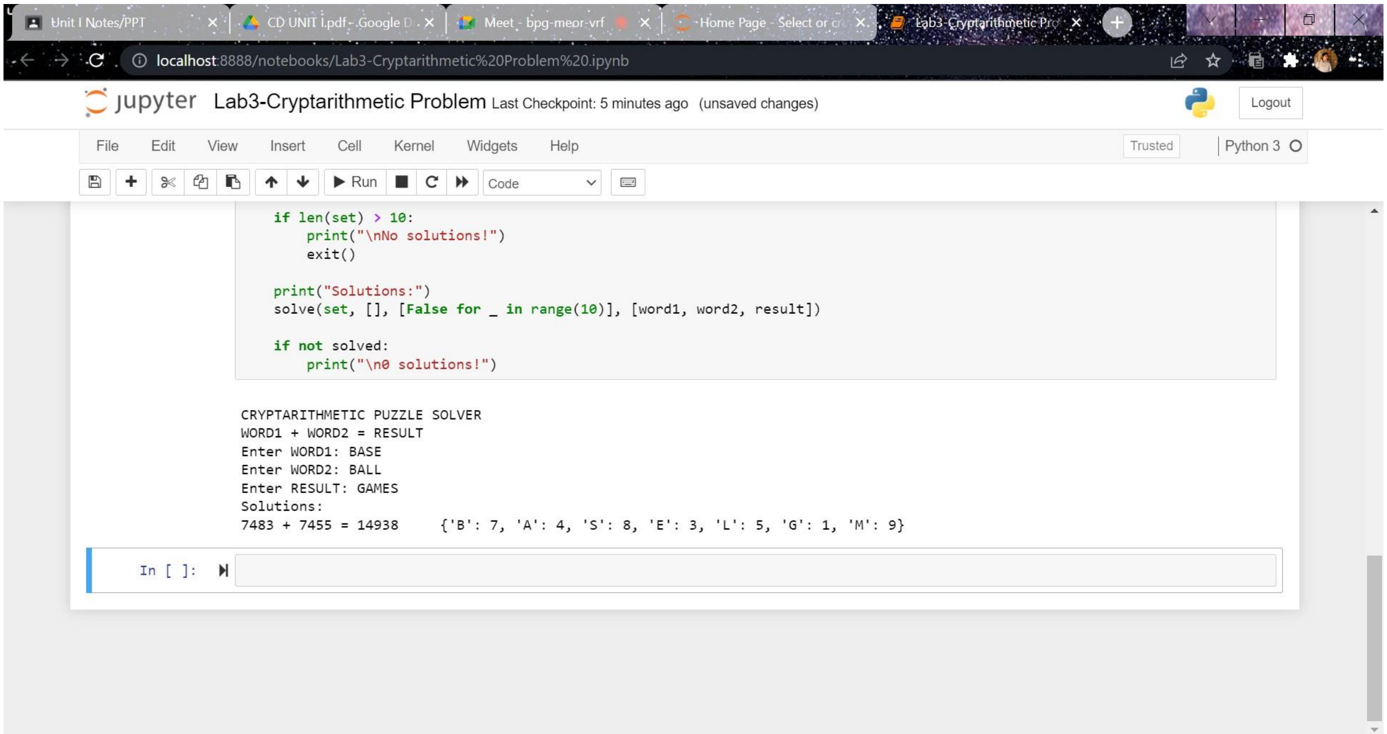
In [1]:

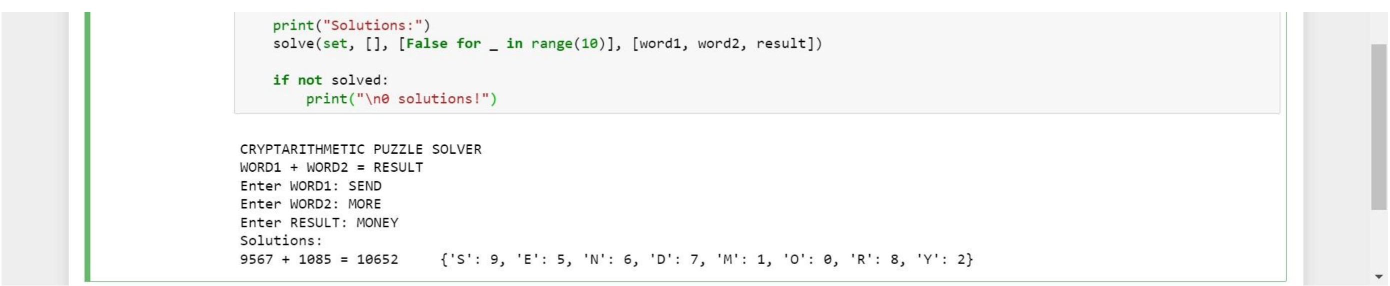
```python
import re

solved = False

def solve(letters, values, visited, words):
    global solved
    if len(set) == len(values):
        map = {}
        for letter, val in zip(letters,values):
            map[letter] = val

        if map[words[0][0]] == 0 or map[words[1][0]] == 0 or map[words[2][0]] == 0:
            return

        word1, word2, res = "", "", ""
        for c in words[0]:
            word1 += str(map[c])
        for c in words[1]:
            word2 += str(map[c])
        for c in words[2]:
            res += str(map[c])

        if int(word1) + int(word2) == int(res):
            print("{} + {} = {}\t{}".format(word1, word2, res, map))
            solved = True

        return

    for i in range(10):
```

jupyter Lab3-Cryptarithmetic Problem Last Checkpoint: 5 minutes ago (autosaved)

```python
    for i in range(10):
        if not visited[i]:
            visited[i] = True
            values.append(i)
            solve(letters, values, visited, words)
            values.pop()
            visited[i] = False


print("\nCRYPTARITHMETIC PUZZLE SOLVER")
print("WORD1 + WORD2 = RESULT")
word1 = input("Enter WORD1: ").upper()
word2 = input("Enter WORD2: ").upper()
result = input("Enter RESULT: ").upper()

if len(result) > (max(len(word1), len(word2)) + 1):
    print("\n0 Solutions!")
else:
    set = []
    for c in word1:
        if c not in set:
            set.append(c)
    for c in word2:
        if c not in set:
            set.append(c)
    for c in result:
        if c not in set:
            set.append(c)
```

Output Screenshot :





Result : The cryptarithmetic problem has been solved using the above code in python

ARYAN ATUL
RA2011003010844
CSE - C2

Lab-4 BFD / DFS

Water Jug problem using BFS

Problem: You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.

Algorithm :

We keep exploring all the different valid cases of the states of water in the jug simultaneously until and unless we reach the required target water.

As provided in the problem statement, at any given state we can do either of the following operations:
1. Fill a jug
2. Empty a jug
3. Transfer water from one jug to another until either of them gets completely filled or empty.

We start at an initial state in the queue where both the jugs are empty.
We then continue to explore all the possible intermediate states derived from the current jug state using the operations provided.
In the BFS, we firstly skip the states which was already visited or if the amount of water in either of the jugs exceeded the jug quantity.
If we continue further, then we firstly mark the current state as visited and check if in this state, if we have obtained the target quantity of water in either of the jugs, we can empty the other jug and return the current state's entire path.
But, if we have not yet found the target quantity, we then derive the intermediate states from the current state of jugs i.e. we derive the valid cases, mentioned in the table above (go through the code once if you have some confusion).
We keep repeating all the above steps until we have found our target or there are no more states left to proceed with.

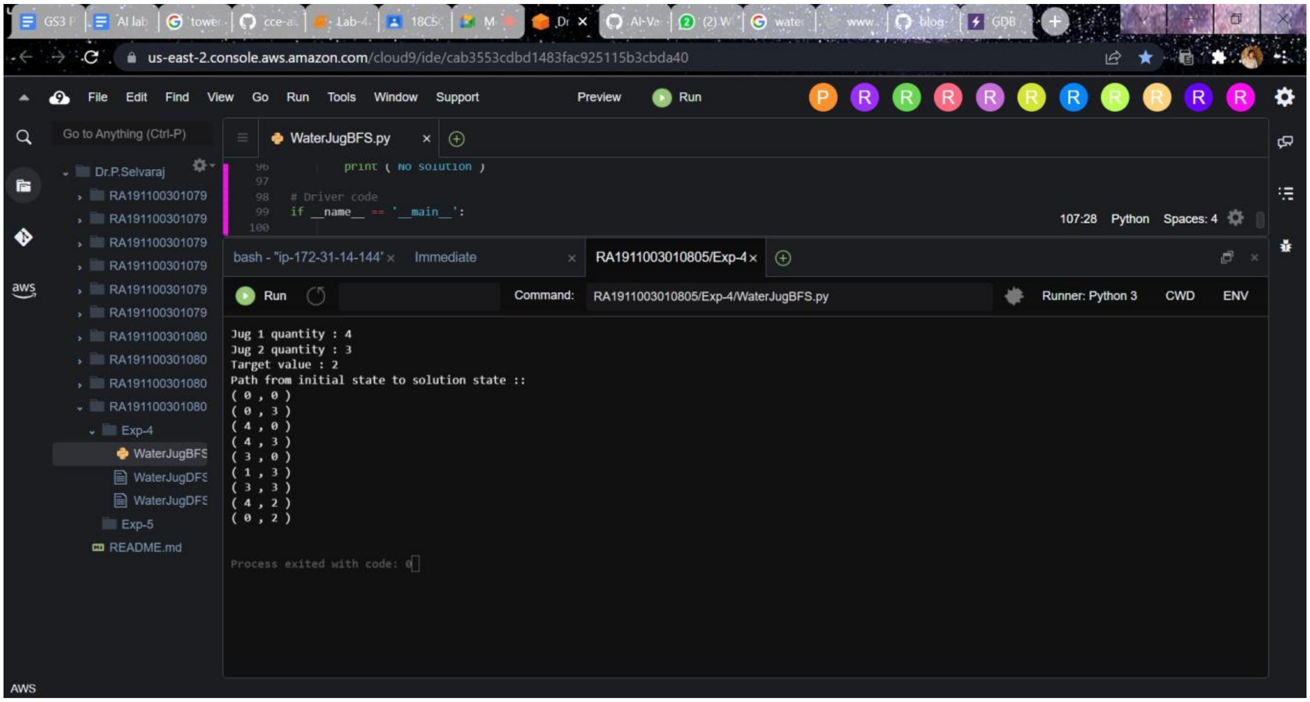
Source Code (in python)

```
from collections import deque

def BFS(a, b, target):

    # Map is used to store the states, every
    # state is hashed to binary value to
    # indicate either that state is visited
    # before or not
    m = {}
    isSolvable = False
    path = []

    # Queue to maintain states
    q = deque()

    # Initialing with initial state
    q.append((0, 0))

    while (len(q) > 0):

        # Current state
        u = q.popleft()

        #q.pop() #pop off used state

        # If this state is already visited
        if ((u[0], u[1]) in m):
            continue

        # Doesn't met jug constraints
        if ((u[0] > a or u[1] > b or
            u[0] < 0 or u[1] < 0)):
            continue

        # Filling the vector for constructing
        # the solution path
        path.append([u[0], u[1]])
```

```
# Marking current state as visited
m[(u[0], u[1])] = 1

# If we reach solution state, put ans=1
if (u[0] == target or u[1] == target):
        isSolvable = True

        if (u[0] == target):
                if (u[1] != 0):

                        # Fill final state
                        path.append([u[0], 0])
        else:
                if (u[0] != 0):

                        # Fill final state
                        path.append([0, u[1]])

        # Print the solution path
        sz = len(path)
        for i in range(sz):
                print("(", path[i][0], ",",
                                path[i][1], ")")
        break

# If we have not reached final state
# then, start developing intermediate
# states to reach solution state
q.append([u[0], b]) # Fill Jug2
q.append([a, u[1]]) # Fill Jug1

for ap in range(max(a, b) + 1):

        # Pour amount ap from Jug2 to Jug1
        c = u[0] + ap
        d = u[1] - ap

        # Check if this state is possible or not
```

```
                if (c == a or (d == 0 and d >= 0)):
                        q.append([c, d])

                # Pour amount ap from Jug 1 to Jug2
                c = u[0] - ap
                d = u[1] + ap

                # Check if this state is possible or not
                if ((c == 0 and c >= 0) or d == b):
                        q.append([c, d])

        # Empty Jug2
        q.append([a, 0])

        # Empty Jug1
        q.append([0, b])

    # No, solution exists if ans=0
    if (not isSolvable):
        print ("No solution")

# Driver code
if __name__ == '__main__':

 Jug1 = int(input("Jug 1 quantity : ")
Jug2 = int(input("Jug 2 quantity : "))
target = int(input("Target value : "))

    print("Path from initial state "
        "to solution state ::")

    BFS(Jug1, Jug2, target)
```

Output Screenshot



[Water Jug problem using DFS]

Source Code in C++ :

```
#include <cstdio>
#include <stack>
#include <map>
#include <algorithm>
using namespace std;

// Representation of a state (x, y)
// x and y are the amounts of water in litres in the two jugs respectively
struct state {
   int x, y;

   // Used by map to efficiently implement lookup of seen states
   bool operator < (const state& that) const {
```

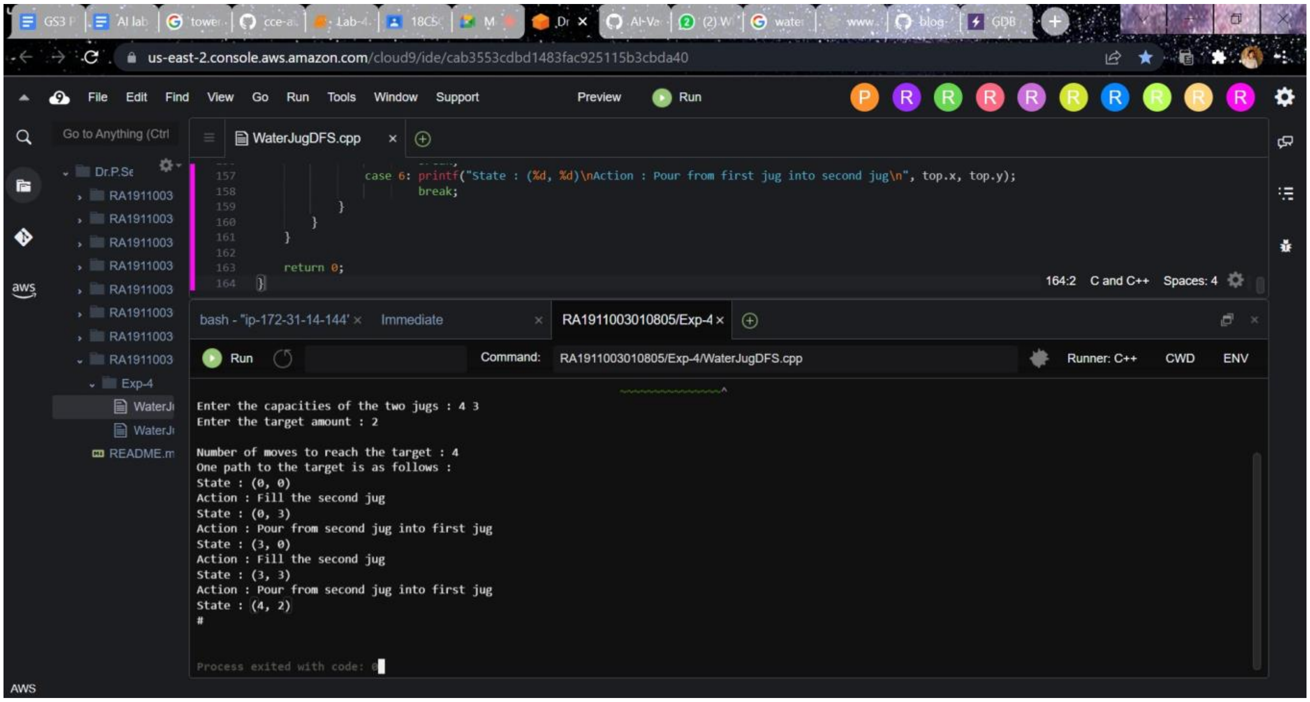
```
        if (x != that.x) return x < that.x;
        return y < that.y;
    }
};

// Capacities of the two jugs respectively and the target amount
int capacity_x, capacity_y, target;

void dfs(state start, stack <pair <state, int> >& path) {
    stack <state> s;
    state goal = (state) {-1, -1};

    // Stores seen states so that they are not revisited and
    // maintains their parent states for finding a path through
    // the state space
    // Mapping from a state to its parent state and rule no. that
    // led to this state
    map <state, pair <state, int> > parentOf;

    s.push(start);
    parentOf[start] = make_pair(start, 0);

    while (!s.empty())    {
        // Get the state at the front of the stack
        state top = s.top();
        s.pop();

        // If the target state has been found, break
        if (top.x == target || top.y == target) {
            goal = top;
            break;
        }

        // Find the successors of this state
        // This step uses production rules to produce successors of the current state
        // while pruning away branches which have been seen before

        // Rule 1: (x, y) -> (capacity_x, y) if x < capacity_x
        // Fill the first jug
```

```
if (top.x < capacity_x) {
   state child = (state) {capacity_x, top.y};
   // Consider this state for visiting only if it has not been visited before
   if (parentOf.find(child) == parentOf.end()) {
      s.push(child);
      parentOf[child] = make_pair(top, 1);
   }
}

// Rule 2: (x, y) -> (x, capacity_y) if y < capacity_y
// Fill the second jug
if (top.y < capacity_y) {
   state child = (state) {top.x, capacity_y};
   if (parentOf.find(child) == parentOf.end()) {
      s.push(child);
      parentOf[child] = make_pair(top, 2);
   }
}

// Rule 3: (x, y) -> (0, y) if x > 0
// Empty the first jug
if (top.x > 0) {
   state child = (state) {0, top.y};
   if (parentOf.find(child) == parentOf.end()) {
      s.push(child);
      parentOf[child] = make_pair(top, 3);
   }
}

// Rule 4: (x, y) -> (x, 0) if y > 0
// Empty the second jug
if (top.y > 0) {
   state child = (state) {top.x, 0};
   if (parentOf.find(child) == parentOf.end()) {
      s.push(child);
      parentOf[child] = make_pair(top, 4);
   }
}
```

```
    // Rule 5: (x, y) -> (min(x + y, capacity_x), max(0, x + y - capacity_x)) if y > 0
    // Pour water from the second jug into the first jug until the first jug is full
    // or the second jug is empty
    if (top.y > 0) {
      state child = (state) {min(top.x + top.y, capacity_x), max(0, top.x + top.y - capacity_x)};
      if (parentOf.find(child) == parentOf.end()) {
        s.push(child);
        parentOf[child] = make_pair(top, 5);
      }
    }

    // Rule 6: (x, y) -> (max(0, x + y - capacity_y), min(x + y, capacity_y)) if x > 0
    // Pour water from the first jug into the second jug until the second jug is full
    // or the first jug is empty
    if (top.x > 0) {
      state child = (state) {max(0, top.x + top.y - capacity_y), min(top.x + top.y, capacity_y)};
      if (parentOf.find(child) == parentOf.end()) {
        s.push(child);
        parentOf[child] = make_pair(top, 6);
      }
    }
  }

  // Target state was not found
  if (goal.x == -1 || goal.y == -1)
    return;

  // backtrack to generate the path through the state space
  path.push(make_pair(goal, 0));
  // remember parentOf[start] = (start, 0)
  while (parentOf[path.top().first].second != 0)
    path.push(parentOf[path.top().first]);
}

int main() {
  stack <pair <state, int> > path;

  printf("Enter the capacities of the two jugs : ");
  scanf("%d %d", &capacity_x, &capacity_y);
```

```
    printf("Enter the target amount : ");
    scanf("%d", &target);

    dfs((state) {0, 0}, path);
    if (path.empty())
      printf("\nTarget cannot be reached.\n");
    else {
      printf("\nNumber of moves to reach the target : %d \nOne path to the target is as follows :\n",
        path.size() - 1);
      while (!path.empty())   {
        state top = path.top().first;
        int rule = path.top().second;
        path.pop();

        switch (rule) {
          case 0: printf("State : (%d, %d)\n#\n", top.x, top.y);
              break;
          case 1: printf("State : (%d, %d)\nAction : Fill the first jug\n", top.x, top.y);
              break;
          case 2: printf("State : (%d, %d)\nAction : Fill the second jug\n", top.x, top.y);
              break;
          case 3: printf("State : (%d, %d)\nAction : Empty the first jug\n", top.x, top.y);
              break;
          case 4: printf("State : (%d, %d)\nAction : Empty the second jug\n", top.x, top.y);
              break;
          case 5: printf("State : (%d, %d)\nAction : Pour from second jug into first jug\n", top.x, top.y);
              break;
          case 6: printf("State : (%d, %d)\nAction : Pour from first jug into second jug\n", top.x, top.y);
              break;
        }
      }
    }

    return 0;
}
```

Output Screenshot :



Result: Hence DFS and BFS algorithms for Water Jug problem has been implemented successfully.

ARYAN ATUL
RA2011003010844
CSE - C2

Lab-5 Best First Search and A* algorithm

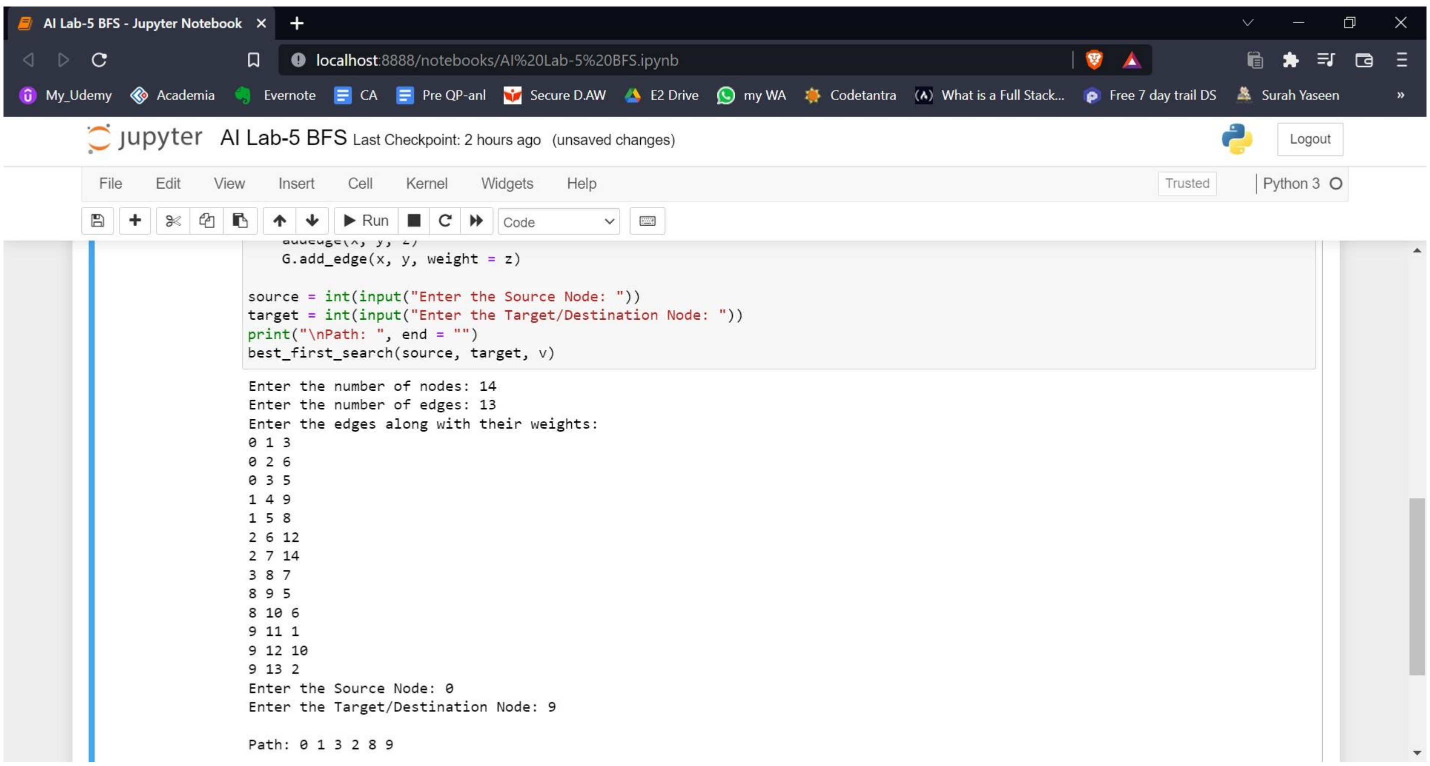
Problem : To implement best first search algorithm and A* algorithm

Source Code for BFS (in Python) : Here we find path with lowest cost

```
from queue import PriorityQueue
import matplotlib.pyplot as plt
import networkx as nx

# for implementing BFS | returns path having lowest cost
def best_first_search(source, target, n):
  visited = [0] * n
  visited[source] = True
  pq = PriorityQueue()
  pq.put((0, source))
  while pq.empty() == False:
    u = pq.get()[1]
    print(u, end=" ") # the path having lowest cost
    if u == target:
      break

    for v, c in graph[u]:
      if visited[v] == False:
        visited[v] = True
        pq.put((c, v))
  print()

# for adding edges to graph
def addedge(x, y, cost):
  graph[x].append((y, cost))
  graph[y].append((x, cost))

G = nx.Graph()
v = int(input("Enter the number of nodes: "))
graph = [[] for i in range(v)] # undirected Graph
e = int(input("Enter the number of edges: "))
print("Enter the edges along with their weights:")
```

```
for i in range(e):
    x, y, z = list(map(int, input().split()))
    addedge(x, y, z)
    G.add_edge(x, y, weight = z)

source = int(input("Enter the Source Node: "))
target = int(input("Enter the Target/Destination Node: "))
print("\nPath: ", end = "")
best_first_search(source, target, v)
```

Output Screenshot (BFS)



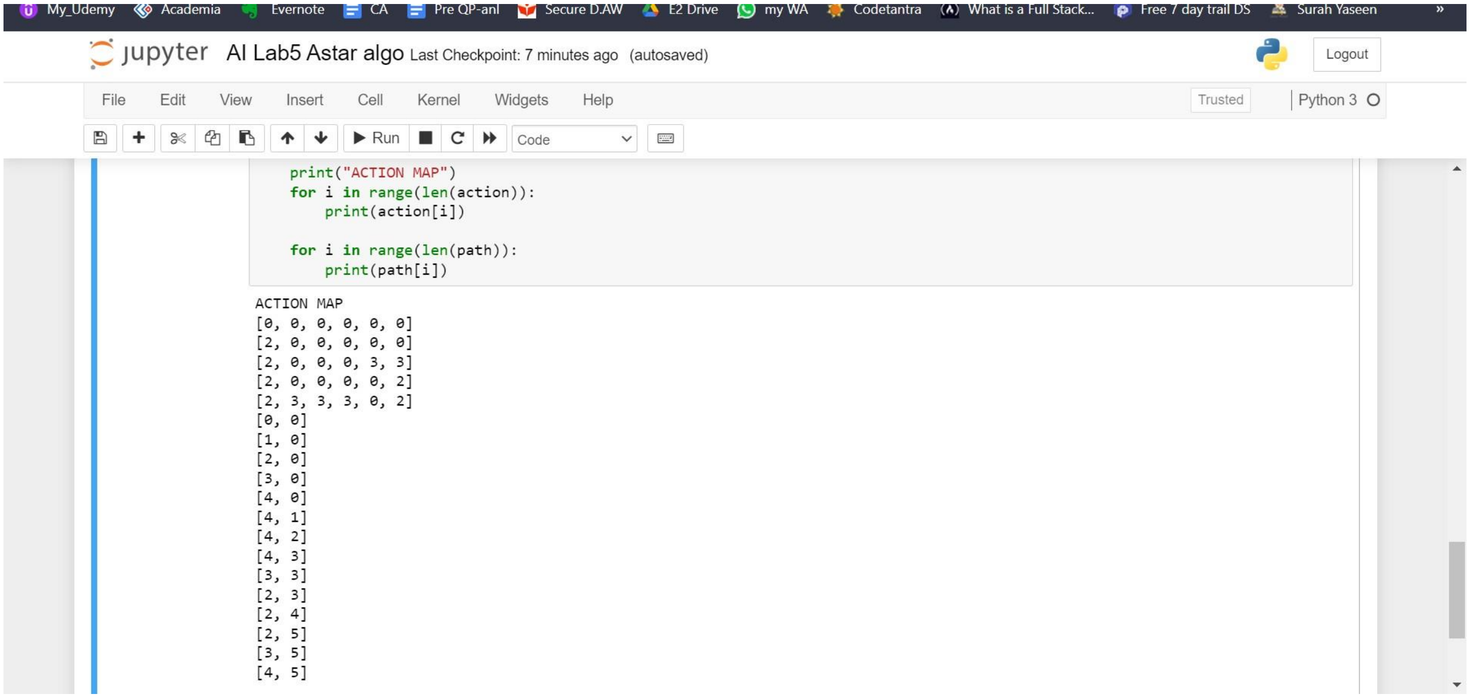
Source Code for A* (in Python) :

Here we find path with lowest cost in the matrix where 1 represent obstacle and o represents free path

```
from__future__import annotations

DIRECTIONS = [
  [-1, 0], # left
  [0, -1], # down
  [1, 0], # right
  [0, 1], # up
]


# function to search the path
def search(
  grid: list[list[int]],
  init: list[int],
  goal: list[int],
  cost: int,
  heuristic: list[list[int]],
) -> tuple[list[list[int]], list[list[int]]]:

  closed = [
    [0 for col in range(len(grid[0]))] for row in range(len(grid))
  ] # the reference grid
  closed[init[0]][init[1]] = 1
  action = [
    [0 for col in range(len(grid[0]))] for row in range(len(grid))
  ] # the action grid

  x = init[0]
  y = init[1]
  g = 0
  f = g + heuristic[x][y] # cost from starting cell to destination cell
  cell = [[f, g, x, y]]

  found = False # flag that is set when search is complete
  resign = False # flag set if we can't find expand
```

```
while not found and not resign:
  if len(cell) == 0:
    raise ValueError("Algorithm is unable to find solution")
  else:  # to choose the least costliest action so as to move closer to the goal
    cell.sort()
    cell.reverse()
    next = cell.pop()
    x = next[2]
    y = next[3]
    g = next[1]

    if x == goal[0] and y == goal[1]:
      found = True
    else:
      for i in range(len(DIRECTIONS)): # to try out different valid actions
        x2 = x + DIRECTIONS[i][0]
        y2 = y + DIRECTIONS[i][1]
        if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 < len(grid[0]):
          if closed[x2][y2] == 0 and grid[x2][y2] == 0:
            g2 = g + cost
            f2 = g2 + heuristic[x2][y2]
            cell.append([f2, g2, x2, y2])
            closed[x2][y2] = 1
            action[x2][y2] = i
invpath = []
x = goal[0]
y = goal[1]
invpath.append([x, y]) # we get the reverse path from here
while x != init[0] or y != init[1]:
  x2 = x - DIRECTIONS[action[x][y]][0]
  y2 = y - DIRECTIONS[action[x][y]][1]
  x = x2
  y          =          y2
  invpath.append([x,  y])

path = []
for i in range(len(invpath)):
  path.append(invpath[len(invpath) - 1 - i])
return path, action
```

```python
if__name__== "__main__":
  grid = [
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],  # 0 are free path whereas 1's are obstacles
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 0],
  ]

  init = [0, 0]
  # all coordinates are given in format [y,x]
  goal = [len(grid) - 1, len(grid[0]) - 1]
  cost = 1

  # the cost map which pushes the path closer to the goal
  heuristic = [[0 for row in range(len(grid[0]))] for col in range(len(grid))]
  for i in range(len(grid)):
    for j in range(len(grid[0])):
      heuristic[i][j] = abs(i - goal[0]) + abs(j - goal[1])
      if grid[i][j] == 1:
        # added extra penalty in the heuristic map
        heuristic[i][j] = 99

  path, action = search(grid, init, goal, cost, heuristic)

  print("ACTION MAP")
  for i in range(len(action)):
    print(action[i])

  for i in range(len(path)):
    print(path[i])
```




```
    print("ACTION MAP")
    for i in range(len(action)):
        print(action[i])

    for i in range(len(path)):
        print(path[i])
```

```
ACTION MAP
[0, 0, 0, 0, 0, 0]
[2, 0, 0, 0, 0, 0]
[2, 0, 0, 0, 3, 3]
[2, 0, 0, 0, 0, 2]
[2, 3, 3, 3, 0, 2]
[0, 0]
[1, 0]
[2, 0]
[3, 0]
[4, 0]
[4, 1]
[4, 2]
[4, 3]
[3, 3]
[2, 3]
[2, 4]
[2, 5]
[3, 5]
[4, 5]
```

Result: Hence BFS and A* algorithms have been implemented

ARYAN ATUL

RA2011003010844

CSE - C2

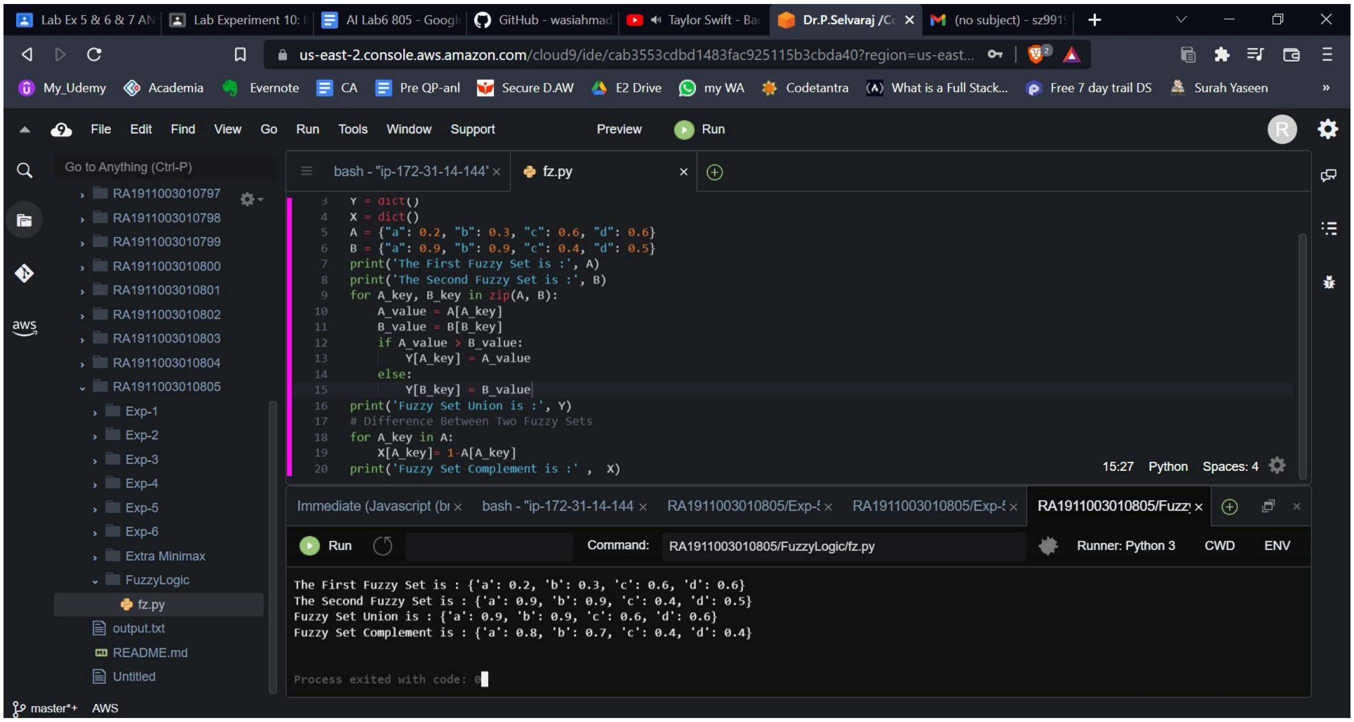
Lab 6 - DST / Fuzzy Logic

Aim: To implement uncertain methods of fuzzy loic

Source Code:

```
A = dict()
B = dict()
Y = dict()
X = dict()
A = {&quot;a&quot;: 0.2, &quot;b&quot;: 0.3, &quot;c&quot;: 0.6, &quot;d&quot;: 0.6}
B = {&quot;a&quot;: 0.9, &quot;b&quot;: 0.9, &quot;c&quot;: 0.4, &quot;d&quot;: 0.5}
print(&#39;The First Fuzzy Set is :&#39;, A)
print(&#39;The Second Fuzzy Set is :&#39;, B)
for A_key, B_key in zip(A, B):
A_value = A[A_key]
B_value = B[B_key]
if A_value &gt; B_value:
Y[A_key] = A_value
else:
Y[B_key] = B_value
print(&#39;Fuzzy Set Union is :&#39;, Y)
# Difference Between Two Fuzzy Sets
for A_key in A:
X[A_key]=  1-A[A_key]
print(&#39;Fuzzy Set Complement is :&#39;, X)
```

Output Screenshot :



Result: Fuzzy logic successfully implemented

ARYAN ATUL
RA2011003010844
CSE - C2

Lab-7 Unification and Resolution

Problem: To implement unification and resolution for a given set of statements

Source Code

1. For unification

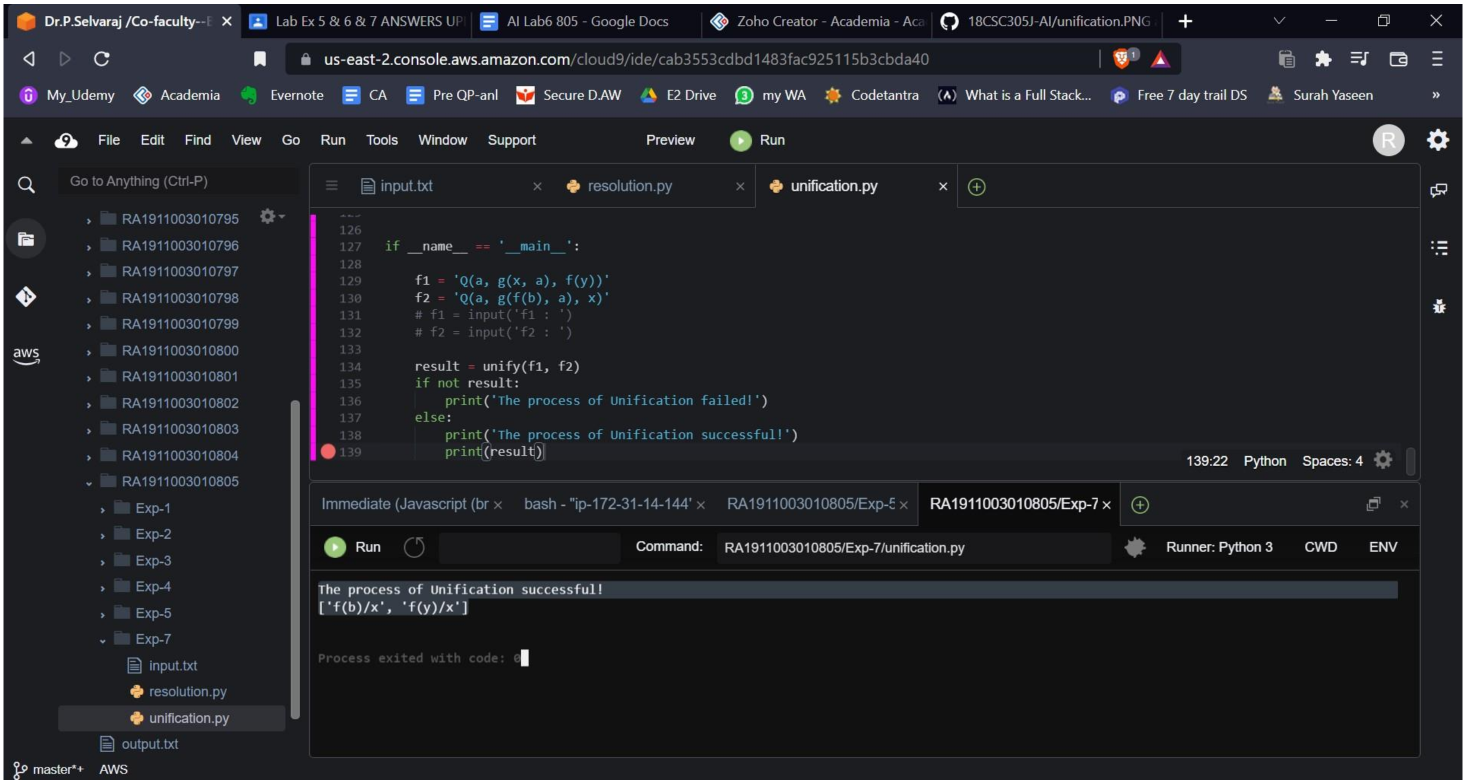
```
def get_index_comma(string):
  index_list = list()
  par_count = 0

  for i in range(len(string)):
    if string[i] == ',' and par_count == 0:
      index_list.append(i)
    elif string[i] == '(':
      par_count += 1
    elif string[i] == ')':
      par_count -= 1

  return index_list


def is_variable(expr):
  for i in expr:
    if i == '(' or i == ')':
      return False

  return True


def process_expression(expr):
  expr = expr.replace(' ', '')
  index = None
  for i in range(len(expr)):
    if expr[i] == '(':
```

```
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list


def get_arg_list(expr):
    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)

    return arg_list


def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
```

```
  if var in arg_list:
    return True

  return False


def unify(expr1, expr2):

  if is_variable(expr1) and is_variable(expr2):
    if expr1 == expr2:
      return 'Null'
    else:
      return False
  elif is_variable(expr1) and not is_variable(expr2):
    if check_occurs(expr1, expr2):
      return False
    else:
      tmp = str(expr2) + '/' + str(expr1)
      return tmp
  elif not is_variable(expr1) and is_variable(expr2):
    if check_occurs(expr2, expr1):
      return False
    else:
      tmp = str(expr1) + '/' + str(expr2)
      return tmp
  else:
    predicate_symbol_1, arg_list_1 = process_expression(expr1)
    predicate_symbol_2, arg_list_2 = process_expression(expr2)

    # Step 2
    if predicate_symbol_1 != predicate_symbol_2:
      return False
    # Step 3
    elif len(arg_list_1) != len(arg_list_2):
      return False
    else:
      # Step 4: Create substitution list
      sub_list = list()

      # Step 5:
```

```
        for i in range(len(arg_list_1)):
            tmp = unify(arg_list_1[i], arg_list_2[i])

            if not tmp:
                return  False
            elif tmp == 'Null':
                pass
            else:
                if type(tmp) == list:
                    for j in tmp:
                        sub_list.append(j)
                else:
                    sub_list.append(tmp)

        # Step 6
        return sub_list


if __name__ == '__main__':

    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')

    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)
```

Output Screenshot



2. For Resolution

```
import copy
import time


class Parameter:
  variable_count = 1

  def__init__(self, name=None):
    if name:
      self.type = "Constant"
      self.name = name
    else:
      self.type = "Variable"
      self.name = "v" + str(Parameter.variable_count)
```

```
        Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name


class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params

    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)


class Sentence:
    sentence_count = 0

    def __init__(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
        self.predicates = []
        self.variable_map = {}
        local = {}
```

```
  for predicate in string.split("|"):
    name = predicate[:predicate.find("(")]
    params = []

    for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
      if param[0].islower():
        if param not in local:  # Variable
          local[param] = Parameter()
          self.variable_map[local[param].name] = local[param]
        new_param = local[param]
      else:
        new_param = Parameter(param)
        self.variable_map[param] = new_param

      params.append(new_param)

    self.predicates.append(Predicate(name, params))

def getPredicates(self):
  return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):
  return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate):
  self.predicates.remove(predicate)
  for key, val in self.variable_map.items():
    if not val:
      self.variable_map.pop(key)

def containsVariable(self):
  return any(not param.isConstant() for param in self.variable_map.values())

def __eq__(self, other):
  if len(self.predicates) == 1 and self.predicates[0] == other:
    return True
  return False

def __str__(self):
```

```
    return "".join([str(predicate) for predicate in self.predicates])


class KB:
  def__init_(self, inputSentences):
    self.inputSentences = [x.replace(" ", "") for x in inputSentences]
    self.sentences = []
    self.sentence_map = {}

  def prepareKB(self):
    self.convertSentencesToCNF()
    for sentence_string in self.inputSentences:
      sentence  =  Sentence(sentence_string)
      for predicate in sentence.getPredicates():
        self.sentence_map[predicate] = self.sentence_map.get(
          predicate, []) + [sentence]

  def  convertSentencesToCNF(self):
    for  sentenceIdx  in  range(len(self.inputSentences)):
      # Do negation of the Premise and add them as literal
      if "=>" in self.inputSentences[sentenceIdx]:
        self.inputSentences[sentenceIdx] = negateAntecedent(
          self.inputSentences[sentenceIdx])

  def askQueries(self,  queryList):
    results = []

    for query in queryList:
      negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
      negatedPredicate = negatedQuery.predicates[0]
      prev_sentence_map = copy.deepcopy(self.sentence_map)
      self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
        negatedPredicate.name, []) + [negatedQuery]
      self.timeLimit = time.time() + 40

      try:
        result = self.resolve([negatedPredicate], [
                    False]*(len(self.inputSentences) + 1))
      except:
        result = False
```

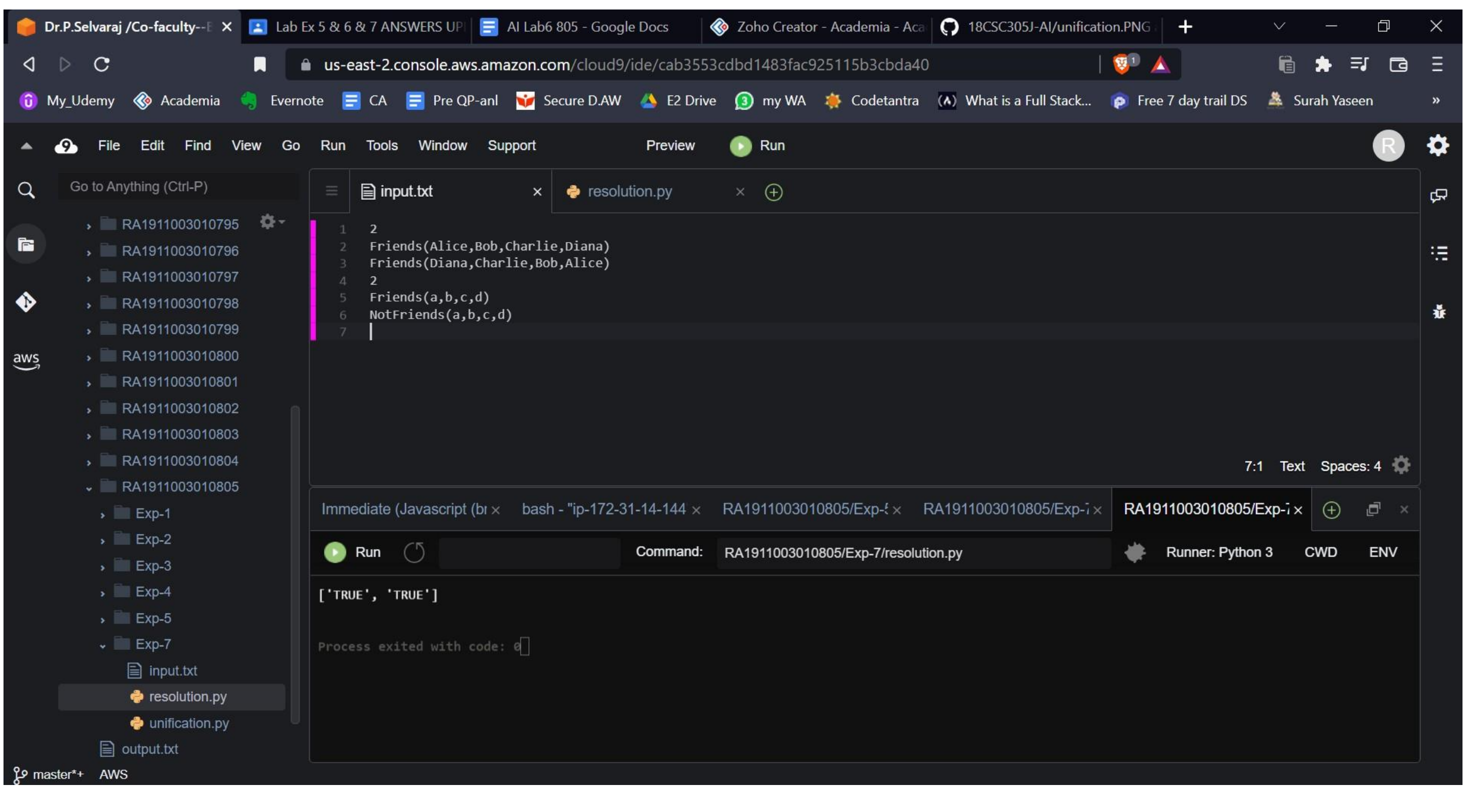
```
        self.sentence_map = prev_sentence_map

        if result:
            results.append("TRUE")
        else:
            results.append("FALSE")

    return results

def resolve(self, queryStack, visited, depth=0):
    if time.time() > self.timeLimit:
        raise Exception
    if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()
        queryPredicateName = negatedQuery.name
        if queryPredicateName not in self.sentence_map:
            return False
        else:
            queryPredicate = negatedQuery
            for kb_sentence in self.sentence_map[queryPredicateName]:
                if not visited[kb_sentence.sentence_index]:
                    for kbPredicate in kb_sentence.findPredicates(queryPredicateName):

                        canUnify, substitution = performUnification(
                            copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

                        if canUnify:
                            newSentence = copy.deepcopy(kb_sentence)
                            newSentence.removePredicate(kbPredicate)
                            newQueryStack = copy.deepcopy(queryStack)

                            if substitution:
                                for old, new in substitution.items():
                                    if old in newSentence.variable_map:
                                        parameter = newSentence.variable_map[old]
                                        newSentence.variable_map.pop(old)
                                        parameter.unify(
                                            "Variable" if new[0].islower() else "Constant", new)
```

```
                        newSentence.variable_map[new] = parameter

                    for predicate in newQueryStack:
                        for index, param in enumerate(predicate.params):
                            if param.name in substitution:
                                new = substitution[param.name]
                                predicate.params[index].unify(
                                    "Variable" if new[0].islower() else "Constant", new)

                for predicate in newSentence.predicates:
                    newQueryStack.append(predicate)

                new_visited = copy.deepcopy(visited)
                if kb_sentence.containsVariable() and len(kb_sentence.predicates) > 1:
                    new_visited[kb_sentence.sentence_index] = True

                if self.resolve(newQueryStack, new_visited, depth + 1):
                    return True
        return False
    return True


def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
                    return False, {}
            else:
```

```
            if not query.isConstant():
                if kb.name not in substitution:
                    substitution[kb.name] = query.name
                elif substitution[kb.name] != query.name:
                    return False, {}
                kb.unify("Variable", query.name)
            else:
                if kb.name not in substitution:
                    substitution[kb.name] = query.name
                elif substitution[kb.name] != query.name:
                    return False, {}
    return True, substitution


def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate


def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)


def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip()
                          for _ in range(noOfSentences)]
        return inputQueries, inputSentences


def printOutput(filename, results):
```

```
  print(results)
  with open(filename, "w") as file:
    for line in results:
      file.write(line)
      file.write("\n")
  file.close()


if__name__=='__main__':
  inputQueries_, inputSentences_ = getInput('RA2011003010844/Exp-6/input.txt')
  knowledgeBase = KB(inputSentences_)
  knowledgeBase.prepareKB()
  results_ = knowledgeBase.askQueries(inputQueries_)
  printOutput("output.txt", results_)
```

Output Screenshot



Result: Hence unification and resolution have been implemented successfully

ARYAN ATUL
RA2011003010844
CSE - C2

AI Lab-8

Aim: To implement a learning algorithm for an application

The learning models used are

1) Logistic Regression
2) SVM
3) Naive Bayes

1) Logistic Regression

Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, True or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1

Source Code

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(X_train,Y_train)
Y_pred_lr = lr.predict(X_test)
In [45]:
Y_pred_lr.shape
```

V. Model Fitting

```
In [43]: from sklearn.metrics import accuracy_score
```

Logistic Regression

```
In [44]: from sklearn.linear_model import LogisticRegression

         lr = LogisticRegression()

         lr.fit(X_train,Y_train)

         Y_pred_lr = lr.predict(X_test)

In [45]: Y_pred_lr.shape

Out[45]: (61,)

In [46]: score_lr = round(accuracy_score(Y_pred_lr,Y_test)*100,2)

         print("The accuracy score achieved using Logistic Regression is: "+str(score_lr)+" %")

         The accuracy score achieved using Logistic Regression is: 85.25 %
```

2) SVM (Support Vector Machine)

- Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

Source Code

```
from sklearn import svm
sv = svm.SVC(kernel='linear')
sv.fit(X_train, Y_train)
Y_pred_svm = sv.predict(X_test)
In [51]:
Y_pred_svm.shape
```

SVM

```
In [50]: from sklearn import svm
         sv = svm.SVC(kernel='linear')
         sv.fit(X_train, Y_train)
         Y_pred_svm = sv.predict(X_test)

In [51]: Y_pred_svm.shape
Out[51]: (61,)

In [52]: score_svm = round(accuracy_score(Y_pred_svm,Y_test)*100,2)
         print("The accuracy score achieved using Linear SVM is: "+str(score_svm)+" %")

         The accuracy score achieved using Linear SVM is: 81.97 %
```

3) Naive Bayes

- 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.

Source Code

```
from sklearn.naive_bayes import GaussianNB
nb = GaussianNB()
nb.fit(X_train,Y_train)
Y_pred_nb = nb.predict(X_test)
In [48]:
Y_pred_nb.shape
```

Naive Bayes

In [47]:
```
from sklearn.naive_bayes import GaussianNB

nb = GaussianNB()

nb.fit(X_train,Y_train)

Y_pred_nb = nb.predict(X_test)
```

In [48]:
```
Y_pred_nb.shape
```

Out[48]:
```
(61,)
```

In [49]:
```
score_nb = round(accuracy_score(Y_pred_nb,Y_test)*100,2)

print("The accuracy score achieved using Naive Bayes is: "+str(score_nb)+" %")
```

```
The accuracy score achieved using Naive Bayes is: 85.25 %
```

Result: Hence the above three learning algorithms have been implemented successfully

ARYAN ATUL
RA2011003010844
CSE - C2

AI Lab-9

Aim: To implement an NLP program

Natural language processing (NLP) refers to the branch of computer science—and more specifically, the branch of artificial intelligence or AI—concerned with giving computers the ability to understand the text and spoken words in much the same way human beings can.

NLP combines computational linguistics—rule-based modeling of human language—with statistical, machine learning, and deep learning models. Together, these technologies enable computers to process human language in the form of text or voice data and to ‘understand’ its full meaning, complete with the speaker or writer’s intent and sentiment.

Source Code:

```
import nltk
from nltk.book import *
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

```
text1
<Text: Moby Dick by Herman Melville 1851>
sents()
```

sent1: Call me Ishmael .
sent2: The family of Dashwood had long been settled in Sussex .
sent3: In the beginning God created the heaven and the earth .
sent4: Fellow - Citizens of the Senate and of the House of Representatives :
sent5: I have a problem with people PMing me to lol JOIN
sent6: SCENE 1 : [wind] [clop clop clop] KING ARTHUR : Whoa there !
sent7: Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .
sent8: 25 SEXY MALE , seeks attrac older single lady , for discreet encounters .
sent9: THE suburb of Saffron Park lay on the sunset side of London , as red and ragged as a cloud of sunset .

```
sent1
```

['Call', 'me', 'Ishmael', '.']

```
print(text7, len(text7))
```

<Text: Wall Street Journal> 100676

```
print(sent7, len(sent7))
```

['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']
18

```
list(set(text7))[:10]
```

['Foster',
'tallies',
'rejected',
'budding',
'Ratings',
'earns',
'Raton',
'8.70',
'Carnival',
'Driscoll']

```
# Frequency of words
dist = FreqDist(text7)
len(dist)
```

12408

```
vocab1 = list(dist.keys())
vocab1[:10]
```

['Pierre', 'Vinken', ',', '61', 'years', 'old', 'will', 'join', 'the', 'board']

```
freqwords = [w for w in vocab1 if len(w) > 5 and dist[w] > 100]
freqwords
```

['billion',
'company',
'president',
'because',
'market',
'million',

```
 'shares',
 'trading',
 'program']
# different forms of the same "word"
input1 = 'List listed lists listing listings'
words1 = input1.lower().split(' ')
words1
['list', 'listed', 'lists', 'listing', 'listings']
porter = nltk.PorterStemmer()
[porter.stem(t) for t in words1]
['list', 'list', 'list', 'list', 'list']
# tokenization
text11 = "Children shouldn't drink a sugary drink before bed."
text11.split(' ')
['Children', "shouldn't", 'drink', 'a', 'sugary', 'drink', 'before', 'bed.']
nltk.word_tokenize(text11)
['Children',
 'should',
 "n't",
 'drink',
 'a',
 'sugary',
 'drink',
 'before',
 'bed',
 '.']
# sentence splitting
text12 = 'This is the first sentence. A gallon of milk in the U.S. costs $2.99. Is this the third sentence? Yes, it is!'
sentences = nltk.sent_tokenize(text12)
len(sentences)
sentences
['This is the first sentence.',
 'A gallon of milk in the U.S. costs $2.99.',
 'Is this the third sentence?',
 'Yes, it is!'
```

Screenshot

```
In [28]: # different forms of the same "word"
         input1 = 'List listed lists listing listings'
         words1 = input1.lower().split(' ')
         words1

Out[28]: ['list', 'listed', 'lists', 'listing', 'listings']

In [29]: porter = nltk.PorterStemmer()
         [porter.stem(t) for t in words1]

Out[29]: ['list', 'list', 'list', 'list', 'list']

In [34]: # tokenization
         text11 = "Children shouldn't drink a sugary drink before bed."
         text11.split(' ')

Out[34]: ['Children', "shouldn't", 'drink', 'a', 'sugary', 'drink', 'before', 'bed.']

In [35]: nltk.word_tokenize(text11)

Out[35]: ['Children',
          'should',
          "n't",
          'drink',
          'a',
          'sugary',
          'drink',
          'before',
          'bed',
          '.']

In [36]: # sentence splitting
         text12 = 'This is the first sentence. A gallon of milk in the U.S. costs $2.99. Is this the third sentence? Yes, it is!'
         sentences = nltk.sent_tokenize(text12)
         len(sentences)

Out[36]: 4

In [37]: sentences

Out[37]: ['This is the first sentence.',
          'A gallon of milk in the U.S. costs $2.99.',
          'Is this the third sentence?',
          'Yes, it is!']
```

Result : Hence NLP has been successfully implemented.

ARYAN ATUL
RA2011003010844
CSE - C2

AI Lab-10

Aim: To apply deep learning to solve a problem

Source Codes

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train,Y_train)
Y_pred_knn=knn.predict(X_test)
Y_pred_knn.shape
score_knn = round(accuracy_score(Y_pred_knn,Y_test)*100,2)

print("The accuracy score achieved using KNN is: "+str(score_knn)+" %")
```

K Nearest Neighbors

```
In [53]: from sklearn.neighbors import KNeighborsClassifier

         knn = KNeighborsClassifier(n_neighbors=7)
         knn.fit(X_train,Y_train)
         Y_pred_knn=knn.predict(X_test)

In [54]: Y_pred_knn.shape

Out[54]: (61,)

In [55]: score_knn = round(accuracy_score(Y_pred_knn,Y_test)*100,2)

         print("The accuracy score achieved using KNN is: "+str(score_knn)+" %")

         The accuracy score achieved using KNN is: 67.21 %
```

```
max_accuracy = 0


for x in range(200):
  dt = DecisionTreeClassifier(random_state=x)
  dt.fit(X_train,Y_train)
```

```
  Y_pred_dt = dt.predict(X_test)
  current_accuracy = round(accuracy_score(Y_pred_dt,Y_test)*100,2)
  if(current_accuracy>max_accuracy):
    max_accuracy = current_accuracy
    best_x = x

#print(max_accuracy)
#print(best_x)


dt = DecisionTreeClassifier(random_state=best_x)
dt.fit(X_train,Y_train)
Y_pred_dt = dt.predict(X_test)
print(Y_pred_dt.shape)
score_dt = round(accuracy_score(Y_pred_dt,Y_test)*100,2)

print("The accuracy score achieved using Decision Tree is: "+str(score_dt)+" %")
```

```
The accuracy score achieved using KNN is: 67.21 %
```

Decision Tree

```
In [56]: from sklearn.tree import DecisionTreeClassifier

         max_accuracy = 0


         for x in range(200):
             dt = DecisionTreeClassifier(random_state=x)
             dt.fit(X_train,Y_train)
             Y_pred_dt = dt.predict(X_test)
             current_accuracy = round(accuracy_score(Y_pred_dt,Y_test)*100,2)
             if(current_accuracy>max_accuracy):
                 max_accuracy = current_accuracy
                 best_x = x

         #print(max_accuracy)
         #print(best_x)


         dt = DecisionTreeClassifier(random_state=best_x)
         dt.fit(X_train,Y_train)
         Y_pred_dt = dt.predict(X_test)

In [57]: print(Y_pred_dt.shape)

         (61,)

In [58]: score_dt = round(accuracy_score(Y_pred_dt,Y_test)*100,2)
```

```python
from sklearn.ensemble import RandomForestClassifier

max_accuracy = 0


for x in range(2000):
  rf = RandomForestClassifier(random_state=x)
  rf.fit(X_train,Y_train)
  Y_pred_rf = rf.predict(X_test)
  current_accuracy = round(accuracy_score(Y_pred_rf,Y_test)*100,2)
  if(current_accuracy>max_accuracy):
    max_accuracy = current_accuracy
    best_x = x

#print(max_accuracy)
#print(best_x)

rf = RandomForestClassifier(random_state=best_x)
rf.fit(X_train,Y_train)
Y_pred_rf = rf.predict(X_test)
Y_pred_rf.shape
score_rf = round(accuracy_score(Y_pred_rf,Y_test)*100,2)

print("The accuracy score achieved using Decision Tree is: "+str(score_rf)+" %")
```

```
print("The accuracy score achieved using Decision Tree is: "+str(score_dt)+" %")
```

```
The accuracy score achieved using Decision Tree is: 81.97 %
```

Random Forest

In [59]:

```
from sklearn.ensemble import RandomForestClassifier

max_accuracy = 0


for x in range(2000):
    rf = RandomForestClassifier(random_state=x)
    rf.fit(X_train,Y_train)
    Y_pred_rf = rf.predict(X_test)
    current_accuracy = round(accuracy_score(Y_pred_rf,Y_test)*100,2)
    if(current_accuracy>max_accuracy):
        max_accuracy = current_accuracy
        best_x = x

#print(max_accuracy)
#print(best_x)

rf = RandomForestClassifier(random_state=best_x)
rf.fit(X_train,Y_train)
Y_pred_rf = rf.predict(X_test)
```

In [60]:

```
Y_pred_rf.shape
```

Out[60]:

```
(61,)
```

Result : Hence deep learning methods have been implemented and compared to to find the best for precision decision making