PROGRAM:

```
theBoard = {'7': ' ' , '8': ' ' , '9': ' ' ,
            '4': ' ' , '5': ' ' , '6': ' ' ,
            '1': ' ' , '2': ' ' , '3': ' ' }
board_keys = []
for key in theBoard:
    board_keys.append(key)

''' We will have to print the updated board after every move in the game and
    thus we will make a function in which we'll define the printBoard function
    so that we can easily print the board everytime by calling this function. '''

def printBoard(board):
    print(board['7'] + '|' + board['8'] + '|' + board['9'])
    print('-+-+-')
    print(board['4'] + '|' + board['5'] + '|' + board['6'])
    print('-+-+-')
    print(board['1'] + '|' + board['2'] + '|' + board['3'])

# Now we'll write the main function which has all the gameplay functionality.
def game():

    turn = 'X'
    count = 0
for i in range(10):
        printBoard(theBoard)
        print("It's your turn," + turn + ".Move to which place?")
```

```
move = input()

if theBoard[move] == ' ':
    theBoard[move] = turn
    count += 1
else:
    print("That place is already filled.\nMove to which place?")
    continue
```

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

```
if count >= 5:
    if theBoard['7'] == theBoard['8'] == theBoard['9'] != ' ': # across the top
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
    elif theBoard['4'] == theBoard['5'] == theBoard['6'] != ' ': # across the middle
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
    elif theBoard['1'] == theBoard['2'] == theBoard['3'] != ' ': # across the bottom
        printBoard(theBoard)
        print("\nGame Over.\n")
        print(" **** " +turn + " won. ****")
        break
    elif theBoard['1'] == theBoard['4'] == theBoard['7'] != ' ': # down the left side
```

```
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break
        elif theBoard['2'] == theBoard['5'] == theBoard['8'] != ' ': # down the middle
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break
        elif theBoard['3'] == theBoard['6'] == theBoard['9'] != ' ': # down the right side
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break
        elif theBoard['7'] == theBoard['5'] == theBoard['3'] != ' ': # diagonal
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break
        elif theBoard['1'] == theBoard['5'] == theBoard['9'] != ' ': # diagonal
            printBoard(theBoard)
            print("\nGame Over.\n")
            print(" **** " +turn + " won. ****")
            break
```

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

```
        if count == 9:
            print("\nGame Over.\n")
            print("It's a Tie!!")

        # Now we have to change the player after every move.
        if turn =='X':
            turn = 'O'
        else:
            turn = 'X'

    # Now we will ask if player wants to restart the game or not.
    restart = input("Do want to play Again?(y/n)")
    if restart == "y" or restart == "Y":
        for key in board_keys:
            theBoard[key] = " "
        game()

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

OUTPUT:

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============= RESTART: C:/Users/91824/Desktop/exp 1 Tic Tac Toe.py =============
 | |
-+-+-
 | |
-+-+-
 | |
It's your turn,X.Move to which place?
7
X| |
-+-+-
 | |
-+-+-
 | |
It's your turn,O.Move to which place?
5
X| |
-+-+-
 |O|
-+-+-
 | |
It's your turn,X.Move to which place?
3
X| |
-+-+-
 |O|
-+-+-
 | |X
It's your turn,O.Move to which place?
5
That place is already filled.
Move to which place?
X| |
-+-+-
 |O|
-+-+-
 | |X
```

PROGRAM:

 # Solves a randomized 8-puzzle using A* algorithm with plug-in heuristics

```
import random
import math

_goal_state = [[1,2,3],
               [4,5,6],
               [7,8,0]]

def index(item, seq):
    """Helper function that returns -1 for non-found index value of a seq"""

    if item in seq:
        return seq.index(item)
    else:
        return -1

class EightPuzzle:

    def __init__(self):
        # heuristic value
        self._hval = 0
        # search depth of current instance
        self._depth = 0
        # parent node in search path
        self._parent = None
        self.adj_matrix = []
        for i in range(3):
            self.adj_matrix.append(_goal_state[i][:])

    def __eq__(self, other):
        if self.__class__ != other.__class__:
            return False
        else:
            return self.adj_matrix == other.adj_matrix

    def __str__(self):
        res = ''
        for row in range(3):
            res += ' '.join(map(str, self.adj_matrix[row]))
            res += '\r\n'
        return res
```

```
def _clone(self):
    p = EightPuzzle()
    for i in range(3):
        p.adj_matrix[i] = self.adj_matrix[i][:]
    return p

def _get_legal_moves(self):

    """Returns list of tuples with which the free space may be swapped"""
    # get row and column of the empty piece
    row, col = self.find(0)
    free = []

    # find which pieces can move there
    if row > 0:
        free.append((row - 1, col))
    if col > 0:
        free.append((row, col - 1))
    if row < 2:
        free.append((row + 1, col))
    if col < 2:
        free.append((row, col + 1))

    return free

def _generate_moves(self):
    free = self._get_legal_moves()
    zero = self.find(0)

    def swap_and_clone(a, b):
        p = self._clone()
        p.swap(a,b)
        p._depth = self._depth + 1
        p._parent = self
        return p

    return map(lambda pair: swap_and_clone(zero, pair), free)

def _generate_solution_path(self, path):
    if self._parent == None:
        return path
    else:
```

```
        path.append(self)
        return self._parent._generate_solution_path(path)

    def solve(self, h):
        """Performs A* search for goal state.h(puzzle) - heuristic function, returns
an   integer """

        def is_solved(puzzle):
            return puzzle.adj_matrix == _goal_state

        openl = [self]
        closedl = []
        move_count = 0
        while len(openl) > 0:
            x = openl.pop(0)
            move_count += 1
            if (is_solved(x)):
                if len(closedl) > 0:
                    return x._generate_solution_path([]), move_count
                else:
                    return [x]

            succ = x._generate_moves()
            idx_open = idx_closed = -1
            for move in succ:
                # have we already seen this node?
                idx_open = index(move, openl)
                idx_closed = index(move, closedl)
                hval = h(move)
                fval = hval + move._depth

                if idx_closed == -1 and idx_open == -1:
                    move._hval = hval
                    openl.append(move)
                elif idx_open > -1:
                    copy = openl[idx_open]
                    if fval < copy._hval + copy._depth:
                        # copy move's values over existing
                        copy._hval = hval
                        copy._parent = move._parent
                        copy._depth = move._depth
                elif idx_closed > -1:
                    copy = closedl[idx_closed]
```

```
            if fval < copy._hval + copy._depth:
                move._hval = hval
                closedl.remove(copy)
                openl.append(move)

        closedl.append(x)
        openl = sorted(openl, key=lambda p: p._hval + p._depth)

    # if finished state not found, return failure
    return [], 0

def shuffle(self, step_count):
    for i in range(step_count):
        row, col = self.find(0)
        free = self._get_legal_moves()
        target = random.choice(free)
        self.swap((row, col), target)
        row, col = target

def find(self, value):
    """returns the row, col coordinates of the specified value in the graph"""
    if value < 0 or value > 8:
        raise Exception("value out of range")

    for row in range(3):
        for col in range(3):
            if self.adj_matrix[row][col] == value:
                return row, col

def peek(self, row, col):
    """returns the value at the specified row and column"""
    return self.adj_matrix[row][col]

def poke(self, row, col, value):
    """sets the value at the specified row and column"""
    self.adj_matrix[row][col] = value

def swap(self, pos_a, pos_b):
    """swaps values at the specified coordinates"""
    temp = self.peek(*pos_a)
    self.poke(pos_a[0], pos_a[1], self.peek(*pos_b))
    self.poke(pos_b[0], pos_b[1], temp)
```

```
def heur(puzzle, item_total_calc, total_calc):
    """
    Heuristic template that provides the current and target position for each
number and the
    total function.

    Parameters:
    puzzle - the puzzle
    item_total_calc - takes 4 parameters: current row, target row, current col,
target col.
    Returns int.
    total_calc - takes 1 parameter, the sum of item_total_calc over all entries, and
returns int.
    This is the value of the heuristic function
    """
    t = 0
    for row in range(3):
        for col in range(3):
            val = puzzle.peek(row, col) - 1
            target_col = val % 3
            target_row = val / 3

            # account for 0 as blank
            if target_row < 0:
                target_row = 2

            t += item_total_calc(row, target_row, col, target_col)

    return total_calc(t)

#some heuristic functions, the best being the standard manhattan distance in this
case, as it comes
#closest to maximizing the estimated distance while still being admissible.

def h_manhattan(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: abs(tr - r) + abs(tc - c),
                lambda t : t)

def h_manhattan_lsq(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: (abs(tr - r) + abs(tc - c))**2,
                lambda t: math.sqrt(t))
```

```
def h_linear(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: math.sqrt(math.sqrt((tr - r)**2 + (tc - c)**2)),
                lambda t: t)

def h_linear_lsq(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: (tr - r)**2 + (tc - c)**2,
                lambda t: math.sqrt(t))

def h_default(puzzle):
    return 0

def main():
    p = EightPuzzle()
    p.shuffle(20)
    print p

    path, count = p.solve(h_manhattan)
    path.reverse()
    for i in path:
        print i

    print "Solved with Manhattan distance exploring", count, "states"
    path, count = p.solve(h_manhattan_lsq)
    print "Solved with Manhattan least squares exploring", count, "states"
    path, count = p.solve(h_linear)
    print "Solved with linear distance exploring", count, "states"
    path, count = p.solve(h_linear_lsq)
    print "Solved with linear least squares exploring", count, "states"

#    path, count = p.solve(heur_default)
#    print "Solved with BFS-equivalent in", count, "moves"

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

OUTPUT:

```
Shell
2 3 5

1 7 8

0 4 6


2 3 5

1 7 8

4 0 6


2 3 5

1 0 8

4 7 6


2 3 5

1 8 0

4 7 6
```

PROGRAM:

```
// CPP program for solving cryptographic puzzles
#include <bits/stdc++.h>
using namespace std;

// vector stores 1 corresponding to index
// number which is already assigned
// to any char, otherwise stores 0

vector<int> use(10);

// structure to store char and its corresponding integer

struct node
{
	char c;
	int v;
};

// function check for correct solution

int check(node* nodeArr, const int count, string s1, string s2, string s3)
{
	int val1 = 0, val2 = 0, val3 = 0, m = 1, j, i;
	// calculate number corresponding to first string
	for (i = s1.length() - 1; i >= 0; i--)
	{
		char ch = s1[i];
		for (j = 0; j < count; j++)
			if (nodeArr[j].c == ch)
				break;
		val1 += m * nodeArr[j].v;
		m *= 10;
```

```
}
m = 1;

// calculate number corresponding to second string
for (i = s2.length() - 1; i >= 0; i--)
{
        char ch = s2[i];
        for (j = 0; j < count; j++)
                if (nodeArr[j].c == ch)
                        break;

        val2 += m * nodeArr[j].v;
        m *= 10;
}
m = 1;

// calculate number corresponding to third string
for (i = s3.length() - 1; i >= 0; i--)
{
        char ch = s3[i];
        for (j = 0; j < count; j++)
                if (nodeArr[j].c == ch)
                        break;
        val3 += m * nodeArr[j].v;
        m *= 10;
}
```

```
	// sum of first two number equal to third return true
	if (val3 == (val1 + val2))
		return 1;

	// else return false
	return 0;
}

// Recursive function to check solution for all permutations
bool permutation(const int count, node* nodeArr, int n,
				string s1, string s2, string s3)
{
	// Base case
	if (n == count - 1)
	{
		// check for all numbers not used yet
		for (int i = 0; i < 10; i++)
		{

			// if not used
			if (use[i] == 0)
			{
				// assign char at index n integer i
				nodeArr[n].v = i;
				// if solution found
				if (check(nodeArr, count, s1, s2, s3) == 1)
				{
```

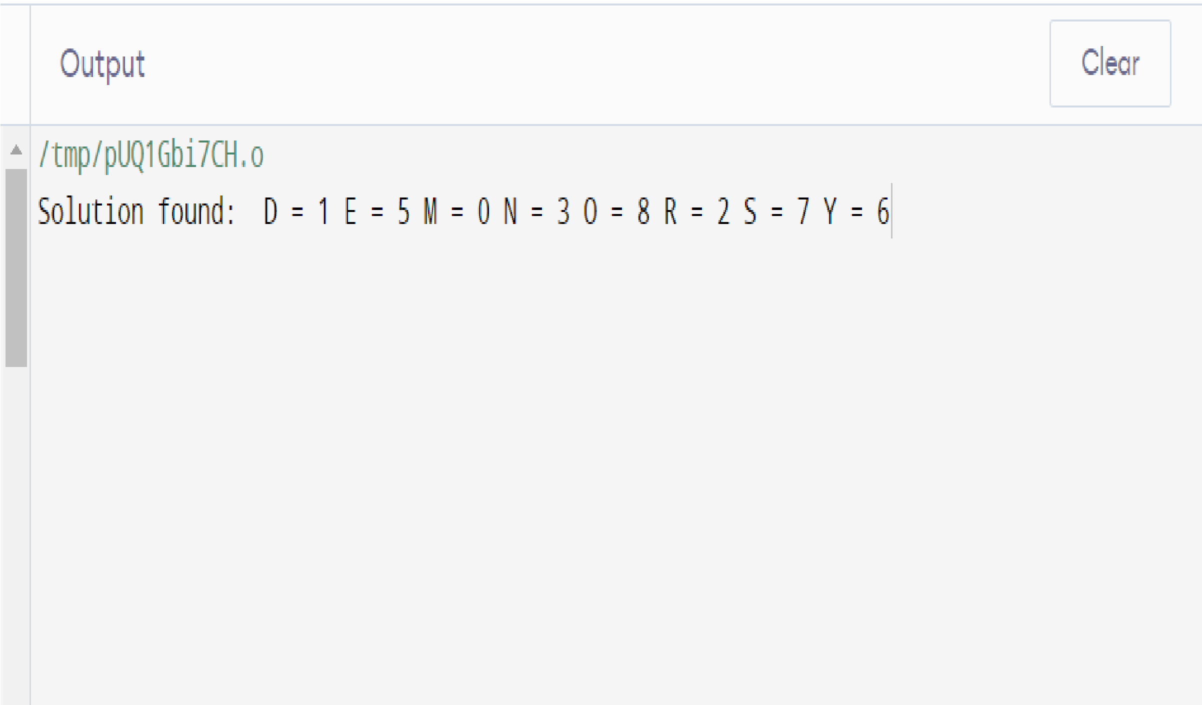
```
}

                cout << "\nSolution found: ";
                for (int j = 0; j < count; j++)
                    cout << " " << nodeArr[j].c << " = "
                        << nodeArr[j].v;
                return true;
            }
        }
    }
    return false;
}

for (int i = 0; i < 10; i++)
{
    // if ith integer not used yet
    if (use[i] == 0)
    {
        // assign char at index n integer i
        nodeArr[n].v = i;
        // mark it as not available for other char
        use[i] = 1;
        // call recursive function
        if (permutation(count, nodeArr, n + 1, s1, s2, s3))
            return true;
        // backtrack for all other possible solutions
        use[i] = 0;
    }
}
```

```
        return false;
}
bool solveCryptographic(string s1, string s2,       string s3)
{
        // count to store number of unique char
        int count = 0;
        // Length of all three strings
        int l1 = s1.length();
        int l2 = s2.length();
        int l3 = s3.length();

        // vector to store frequency of each char
        vector<int> freq(26);
        for (int i = 0; i < l1; i++)
                ++freq[s1[i] - 'A'];
        for (int i = 0; i < l2; i++)
                ++freq[s2[i] - 'A'];
        for (int i = 0; i < l3; i++)
                ++freq[s3[i] - 'A'];

        // count number of unique char
        for (int i = 0; i < 26; i++)
                if (freq[i] > 0)
                        count++;

        // solution not possible for count greater than 10
        if (count > 10)
```

```
    {
        cout << "Invalid strings";
        return 0;
    }

    // array of nodes
    node nodeArr[count];

    // store all unique char in nodeArr
    for (int i = 0, j = 0; i < 26; i++)
    {
        if (freq[i] > 0)
        {
            nodeArr[j].c = char(i + 'A');
            j++;
        }
    }
    return permutation(count, nodeArr, 0, s1, s2, s3);
}

// Driver function
int main()
{
    string s1 = "SEND";
    string s2 = "MORE";
    string s3 = "MONEY";
```

```
	if (solveCryptographic(s1, s2, s3) == false)
		cout << "No solution";
	return 0;
}
```

<u>OUTPUT:</u>

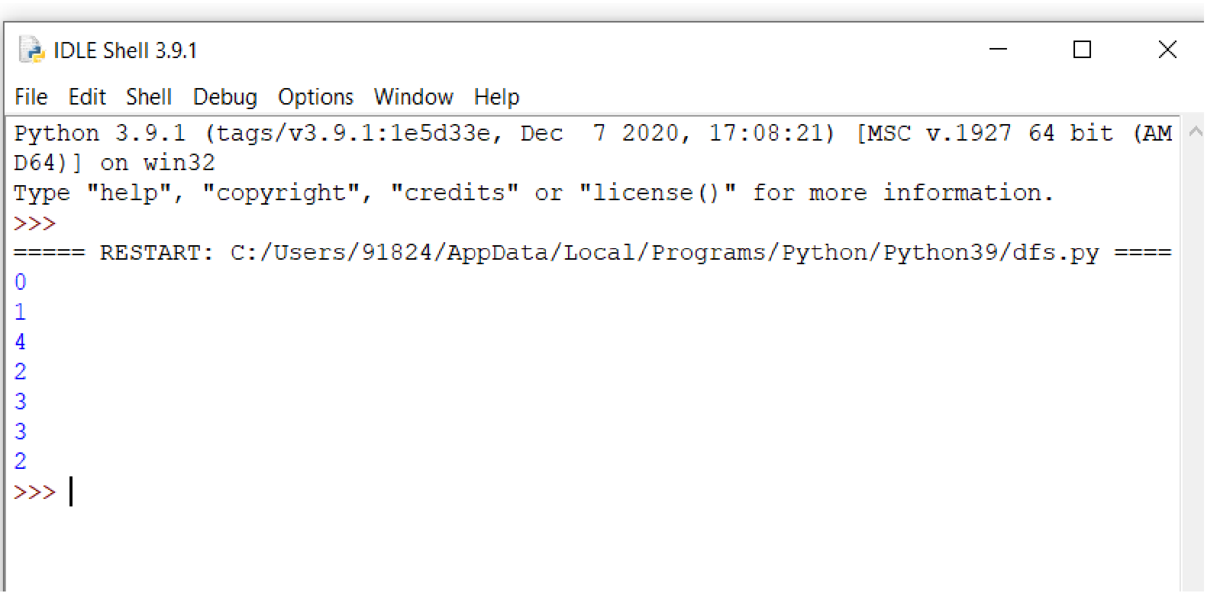


PROGRAM:

 # DFS algorithm in Python
 # DFS algorithm

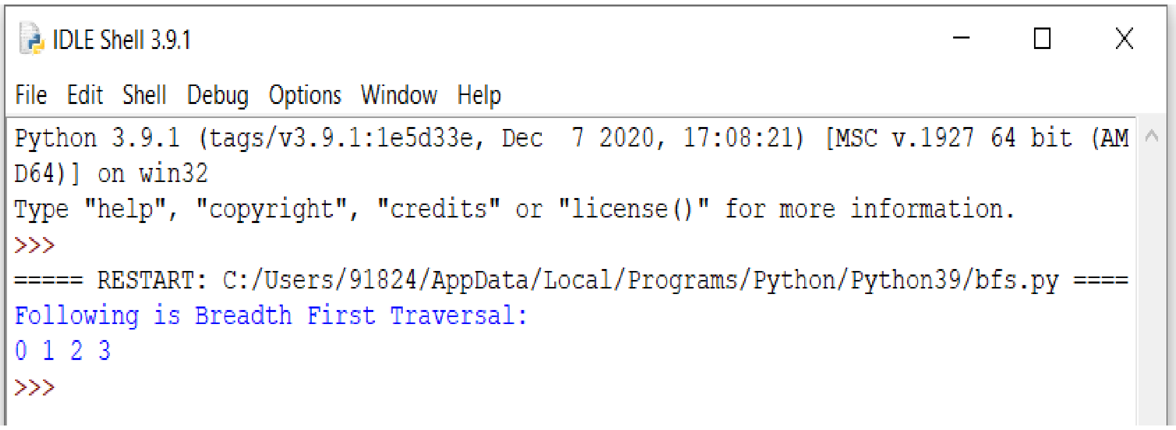
```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited

graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

OUTPUT:



```
# BFS algorithm in Python
# BFS algorithm

import collections
def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)

    while queue:
        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)

          if __name__ == '__main__':
          graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
          print("Following is Breadth First Traversal: ")
          bfs(graph, 0)
```

OUTPUT:


IDLE Shell 3.9.1
File Edit Shell Debug Options Window Help
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/91824/AppData/Local/Programs/Python/Python39/bfs.py ====
Following is Breadth First Traversal:
0 1 2 3
>>>

PROGRAM:

```
class Node():
    """A node class for A* Pathfinding"""
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.position == other.position
def astar(maze, start, end):

    """Returns a list of tuples as a path from the given start to the given end in
the given maze"""
    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
```

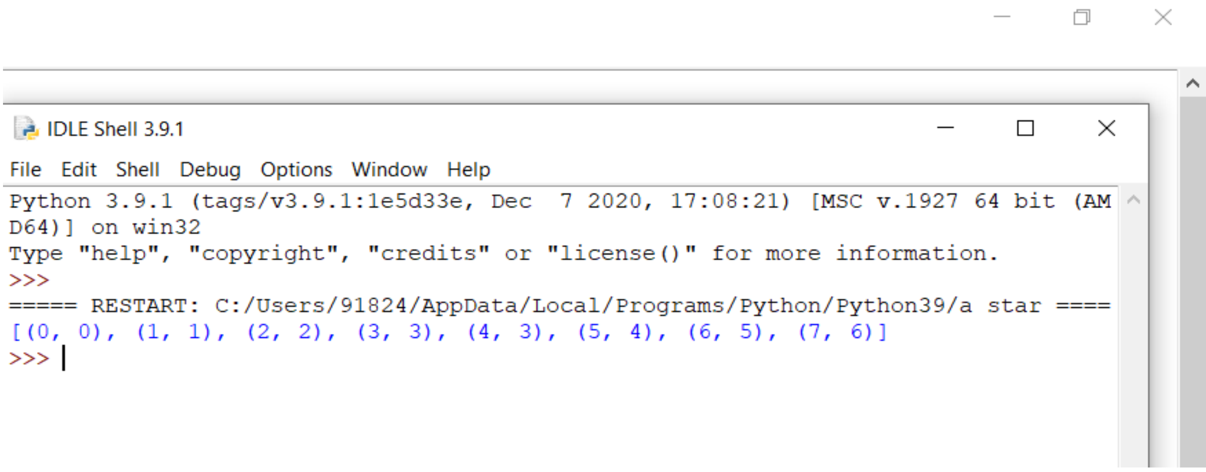
```
closed_list = []
# Add the start node
open_list.append(start_node)

# Loop until you find the end
while len(open_list) > 0:

# Get the current node
    current_node = open_list[0]
    current_index = 0
    for index, item in enumerate(open_list):
        if item.f < current_node.f:
            current_node = item
            current_index = index

    # Pop current off open list, add to closed list
    open_list.pop(current_index)
    closed_list.append(current_node)

    # Found the goal
    if current_node == end_node:
        path = []
        current = current_node
```

```
        while current is not None:
            path.append(current.position)
            current = current.parent
        return path[::-1] # Return reversed path


    # Generate children
    children = []
    for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares


        # Get node position
        node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])


        # Make sure within range
        if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or node_position[1] < 0:
            continue
        # Make sure walkable terrain
        if maze[node_position[0]][node_position[1]] != 0:
            continue
        # Create new node
        new_node = Node(current_node, node_position)
        # Append
```

```
            children.append(new_node)

        # Loop through children

        for child in children:

            # Child is on the closed list

            for closed_child in closed_list:

                if child == closed_child:

                    continue

            # Create the f, g, and h values

            child.g = current_node.g + 1

            child.h = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1]
- end_node.position[1]) ** 2)

            child.f = child.g + child.h

            # Child is already in the open list

            for open_node in open_list:

                if child == open_node and child.g > open_node.g:

                    continue

            # Add the child to the open list

            open_list.append(child)

def main():

    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
```

```
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
    start = (0, 0)
    end = (7, 6)
    path = astar(maze, start, end)
    print(path)
if __name__ == '__main__':
    main()
```

OUTPUT:



PROGRAM:

```
# Import libraries
import sys
import random
# This class represent a tic tac to game
class TicTacToeGame:
    # Create a new game
    def __init__(self, rows:int, columns:int, goal:int, max_depth:int=4):


        # Create the game state
        self.state = []
        self.tiles = {}
        self.inverted_tiles = {}
        tile = 0
        for y in range(rows):
            row = []
            for x in range(columns):
                row += '.'
                tile += 1
                self.tiles[tile] = (y, x)
                self.inverted_tiles[(y, x)] = tile
            self.state.append(row)
     # Set the number of noughts and crosses in a row that is needed to win the game
        self.goal = goal
     # Create vectors
        self.vectors = [(1,0), (0,1), (1,1), (-1,1)] n
```

```
# Set lengths
        self.rows = rows
        self.columns = columns
        self.max_row_index = rows - 1
        self.max_columns_index = columns - 1
        self.max_depth = max_depth
 # Heuristics for cutoff
        self.winning_positions = []
        self.get_winning_positions()
        # Set the starting player at random
        #self.player = 'O'
        self.player = random.choice(['X', 'O'])
  # Get winning positions
  def get_winning_positions(self):
  # Loop the board
        for y in range(self.rows):
            for x in range(self.columns):
    # Loop vectors
                for vector in self.vectors:

                    # Get the start position
                    sy, sx = (y, x)
                    # Get vector deltas
                    dy, dx = vector
                    # Create a counter
                    counter = 0
                    # Loop until we are outside the board
```

```
            positions = []
            while True:
                # Add the position
                positions.append(self.inverted_tiles.get((sy, sx)))
                # Check if we have a winning position
                if (len(positions) == self.goal):
                    # Add winning positions
                    self.winning_positions.append(positions)
                    # Break out from the loop
                    break
                # Update the position
                sy += dy
                sx += dx

                # Check if the loop should terminate
                if(sy < 0 or abs(sy) > self.max_row_index or sx < 0 or abs(sx) >
self.max_columns_index):
                    break
    # Play the game
    def play(self):
        # Variables
        result = None
        # Create an infinite loop
        print('Starting board')
        while True:
            # Draw the state
            self.print_state()
```

```
# Get a move from a player
if (self.player == 'X'): # AI
    # Print AI move
    print('Player X moving (AI) ...')
    # Get the best move
    max, py, px, depth = self.max(-sys.maxsize, sys.maxsize)
    # Get a heuristic move at cutoff
    print('Depth: {0}'.format(depth))
    if(depth > self.max_depth):
        py, px = self.get_best_move()
    # Make a move
    self.state[py][px] = 'X'
    # Check if the game has ended, break out from the loop in that case
    result = self.game_ended()
    if(result != None):
        break
    # Change turn
    self.player = 'O'
elif (self.player == 'O'): # Human player

    # Print turn
    print('Player O moving (Human) ...')
    # Get a recommended move
    min, py, px, depth = self.min(-sys.maxsize, sys.maxsize)
    # Get a heuristic move at cutoff
    print('Depth: {0}'.format(depth))
    if(depth > self.max_depth):
```

```
            py, px = self.get_best_move()
        # Print a recommendation
        print('Recommendation: {0}'.format(self.inverted_tiles.get((py, px))))
        # Get input
        number = int(input('Make a move (tile number): '))
        tile = self.tiles.get(number)
        # Check if the move is legal
        if(tile != None):
            # Make a move
            py, px = tile
            self.state[py][px] = 'O'
            # Check if the game has ended, break out from the loop in that case
            result = self.game_ended()
            if(result != None):
                break
            # Change turn
            self.player = 'X'
        else:
            print('Move is not legal, try again.')
    # Print result
    self.print_state()
    print('Winner is player: {0}'.format(result))
# An evaluation function to get the best move based on heuristics
def get_best_move(self):
    # Create an heuristic dictionary
    heuristics = {}
    # Get all empty cells
```

```
empty_cells = []
for y in range(self.rows):
    for x in range(self.columns):
        if (self.state[y][x] == '.'):
            empty_cells.append((y, x))
# Loop empty positions
for empty in empty_cells:
    # Get numbered position
    number = self.inverted_tiles.get(empty)
    # Loop winning positions
    for win in self.winning_positions:
        # Check if number is in a winning position
        if(number in win):
            # Calculate the number of X:s and O:s in the winning position
            player_x = 0
            player_o = 0
            start_score = 1
            for box in win:
                # Get the position
                y, x = self.tiles[box]
                # Count X:s and O:s
                if(self.state[y][x] == 'X'):
                    player_x += start_score if self.player == 'X' else start_score * 2
                    start_score *= 10
                elif (self.state[y][x] == 'O'):
                    player_o += start_score if self.player == 'O' else start_score * 2
                    start_score *= 10
```

```
                # Save heuristic
                if(player_x == 0 or player_o == 0):
                    # Calculate a score
                    score = max(player_x, player_o) + start_score
                    # Update the score
                    if(heuristics.get(number) != None):
                        heuristics[number] += score
                    else:
                        heuristics[number] = score
        # Get the best move from the heuristic dictionary
        best_move = random.choice(empty_cells)
        best_count = -sys.maxsize
        for key, value in heuristics.items():
            if(value > best_count):
                best_move = self.tiles.get(key)
                best_count = value
        # Return the best move
        return best_move
    # Check if the game has ended
    def game_ended(self) -> str:
        # Check if a player has won
        result = self.player_has_won()
        if(result != None):
            return result
        # Check if the board is full
        for y in range(self.rows):
            for x in range(self.columns):
```

```
            if (self.state[y][x] == '.'):
                return None
        # Return a tie
        return 'It is a tie!'

    # Check if a player has won
    def player_has_won(self) -> str:

        # Loop the board
        for y in range(self.rows):
            for x in range(self.columns):
                # Loop vectors
                for vector in self.vectors:
                    # Get the start position
                    sy, sx = (y, x)
                    # Get vector deltas
                    dy, dx = vector
                    # Create counters
                    steps = 0
                    player_x = 0
                    player_o = 0
                    # Loop until we are outside the board or have moved the number of
steps in the goal
                    while steps < self.goal:
                        # Add steps
                        steps += 1
                        # Check if a player has a piece in the tile
```

```
                if(self.state[sy][sx] == 'X'):
                    player_x += 1
                elif(self.state[sy][sx] == 'O'):
                    player_o += 1
                # Update the position
                sy += dy
                sx += dx

                # Check if the loop should terminate
                if(sy < 0 or abs(sy) > self.max_row_index or sx < 0 or abs(sx) >
self.max_columns_index):
                    break
            # Check if we have a winner
            if(player_x >= self.goal):
                return 'X'
            elif(player_o >= self.goal):
                return 'O'
        # Return None if no winner is found
        return None
    # Get a min value (O)
    def min(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):

        # Variables
        min_value = sys.maxsize
        by = None
        bx = None
```

```
# Check if the game has ended
result = self.game_ended()
if(result != None):
    if result == 'X':
        return 1, 0, 0, depth
    elif result == 'O':
        return -1, 0, 0, depth
    elif result == 'It is a tie!':
        return 0, 0, 0, depth
elif(depth > self.max_depth):
    return 0, 0, 0, depth
# Loop the board
for y in range(self.rows):
    for x in range(self.columns):
        # Check if the tile is empty
        if (self.state[y][x] == '.'):
            # Make a move
            self.state[y][x] = 'O'
            # Get max value
            max, max_y, max_x, depth = self.max(alpha, beta, depth + 1)

            # Set min value to max value if it is lower than curren min value
            if (max < min_value):
                min_value = max
                by = y
                bx = x
```

```
            # Reset the tile
            self.state[y][x] = '.'
            # Do an alpha test
            if (min_value <= alpha):
                return min_value, bx, by, depth
            # Do a beta test
            if (min_value < beta):
                beta = min_value
    # Return min value
    return min_value, by, bx, depth
# Get max value (X)
def max(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):
    # Variables
    max_value = -sys.maxsize
    by = None
    bx = None
    # Check if the game has ended
    result = self.game_ended()
    if(result != None):
        if result == 'X':
            return 1, 0, 0, depth
        elif result == 'O':
            return -1, 0, 0, depth
        elif result == 'It is a tie!':
            return 0, 0, 0, depth
    elif(depth > self.max_depth):
        return 0, 0, 0, depth
```

```
        # Loop the board
        for y in range(self.rows):
            for x in range(self.columns):
                # Check if the current tile is empty
                if (self.state[y][x] == '.'):

                    # Add a piece to the board
                    self.state[y][x] = 'X'
                    # Set max value to min value if min value is greater than current max value
                    min, min_y, min_x, depth = self.min(alpha, beta, depth + 1)
                    # Adjust the max value
                    if (min > max_value):
                        max_value = min
                        by = y
                        bx = x
                    # Reset the tile
                    self.state[y][x] = '.'
                    # Do a beta test
                    if (max_value >= beta):
                        return max_value, bx, by, depth
                    # Do an alpha test
                    if (max_value > alpha):
                        alpha = max_value
        # Return max value
        return max_value, by, bx, depth
    # Print the current game state
```

```
    def print_state(self):
        for y in range(self.rows):
            print('| ', end='')
            for x in range(self.columns):
                if (self.state[y][x] != '.'):
                    print(' {0}  | '.format(self.state[y][x]), end='')
                else:
                    digit = str(self.inverted_tiles.get((y,x))) if len(str(self.inverted_tiles.get((y,x)))) > 1 else ' ' + str(self.inverted_tiles.get((y,x)))
                    print('{0}  | '.format(digit), end='')
            print()
        print()
# The main entry point for this module
def main():
    # Create a game
    #game = TicTacToeGame(7, 6, 4, 1000)
    game = TicTacToeGame(3, 3, 3, 1000)
    # Play the game
    game.play()
# Tell python to run main method
if __name__ == "__main__": main()
```

OUTPUT:

```
*IDLE Shell 3.9.1*
File  Edit  Shell  Debug  Options  Window  Help
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:/Users/91824/AppData/Local/Programs/Python/Python39/minimax.py ==
Starting board
|   1   |   2   |   3   |
|   4   |   5   |   6   |
|   7   |   8   |   9   |

Player O moving (Human) ...
Depth: 1029
Recommendation: 5
Make a move (tile number): 3
|   1   |   2   |   O   |
|   4   |   5   |   6   |
|   7   |   8   |   9   |

Player X moving (AI) ...
Depth: 1012
|   1   |   2   |   O   |
|   4   |   X   |   6   |
|   7   |   8   |   9   |

Player O moving (Human) ...
Depth: 1006
Recommendation: 1
Make a move (tile number): 6
|   1   |   2   |   O   |
|   4   |   X   |   O   |
|   7   |   8   |   9   |

Player X moving (AI) ...
Depth: 322
|   1   |   2   |   O   |
|   4   |   X   |   O   |
|   7   |   8   |   X   |

Player O moving (Human) ...
Depth: 29
Recommendation: 1
Make a move (tile number):
```

PROGRAM:

```
def get_index_comma(string):
	"""
	Return index of commas in string
	"""

	index_list = list()
	# Count open parentheses
	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):
	"""
	Check if expression is variable
	"""

	for i in expr:
```

```
        if i == '(':
            return False

    return True



def process_expression(expr):
    """
    input:  - expression:
            'Q(a, g(x, b), f(y))'
    return: - predicate symbol:
            Q
            - list of arguments
            ['a', 'g(x, b)', 'f(y)']
    """
    # Remove space in expression
    expr = expr.replace(' ', '')

    # Find the first index == '('
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break

    # Return predicate symbol and remove predicate symbol in expression
    predicate_symbol = expr[:index]
```

```
    expr = expr.replace(predicate_symbol, '')

    # Remove '(' in the first index and ')' in the last index
    expr = expr[1:len(expr) - 1]

    # List of arguments
    arg_list = list()

    # Split string with commas, return list of arguments
    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):
    """
    input:  expression:
            'Q(a, g(x, b), f(y))'
    return: full list of arguments:
```

```
        ['a', 'x', 'b', 'y']
    """

    _, 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):
    """
    Check if var occurs in expr
    """

    arg_list = get_arg_list(expr)
```

```
        if var in arg_list:
            return True

        return False



def unify(expr1, expr2):
        """
```

Unification Algorithm

Step 1: If Ψ1 or Ψ2 is a variable or constant, then:

a, If Ψ1 or Ψ2 are identical, then return NULL.

b, Else if Ψ1 is a variable:

- then if Ψ1 occurs in Ψ2, then return False

- Else return (Ψ2 / Ψ1)

c, Else if Ψ2 is a variable:

- then if Ψ2 occurs in Ψ1, then return False

- Else return (Ψ1 / Ψ2)

d, Else return False

Step 2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return False.

Step 3: IF Ψ1 and Ψ2 have a different number of arguments, then return False.

Step 4: Create Substitution list.

Step 5: For i=1 to the number of elements in Ψ1

a, Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.

b, If S = False then returns False

c, If S ≠ Null then append to Substitution list

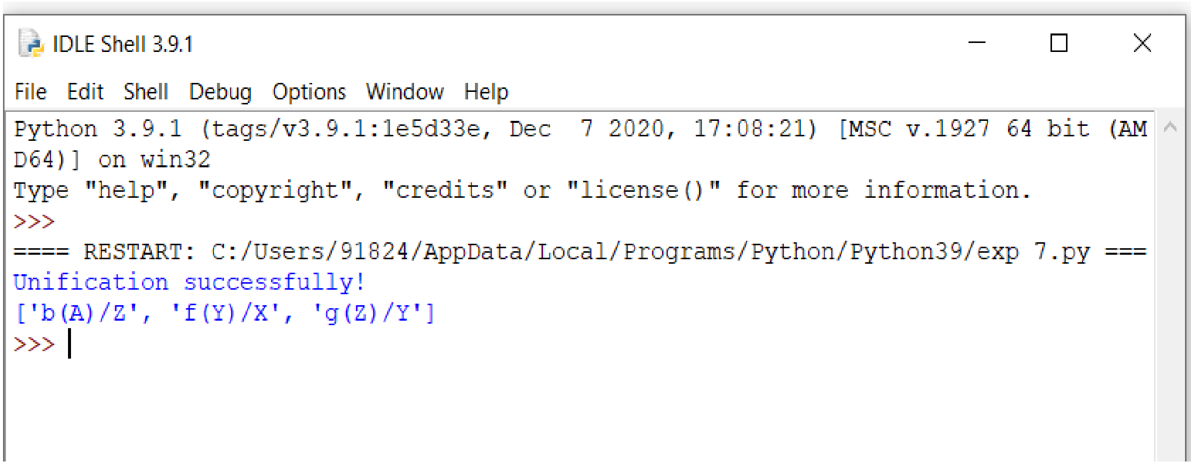
Step 6: Return Substitution list.

```
"""

# Step 1:
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__':
    # Data 1
    f1 = 'p(b(A), X, f(g(Z)))'
    f2 = 'p(Z, f(Y), f(Y))'

    # Data 2
    # f1 = 'Q(a, g(x, a), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    # Data 3
    # f1 = 'Q(a, g(x, a, d), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    result = unify(f1, f2)
    if not result:
        print('Unification failed!')
    else:
        print('Unification successfully!')
        print(result)
```

<u>OUTPUT:</u>



IDLE Shell 3.9.1
File Edit Shell Debug Options Window Help
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/91824/AppData/Local/Programs/Python/Python39/exp 7.py ===
Unification successfully!
['b(A)/Z', 'f(Y)/X', 'g(Z)/Y']
>>>

PROGRAM:

```
#include <stdio.h>
#define SIZE 9

//sudoku problem
int matrix[9][9] = {
    {5,3,0,0,7,0,0,0,0},
    {6,0,0,1,9,5,0,0,0},
    {0,9,8,0,0,0,0,6,0},
    {8,0,0,0,6,0,0,0,3},
    {4,0,0,8,0,3,0,0,1},
    {7,0,0,0,2,0,0,0,6},
    {0,6,0,0,0,0,2,8,0},
    {0,0,0,4,1,9,0,0,5},
    {0,0,0,0,8,0,0,7,9}
};


//function to print sudoku
void print_sudoku()
{
    int i,j;
    for(i=0;i<SIZE;i++)
    {
        for(j=0;j<SIZE;j++)
        {
            printf("%d\t",matrix[i][j]);
        }
```

```
        printf("\n\n");
    }
}

//function to check if all cells are assigned or not
//if there is any unassigned cell
//then this function will change the values of
//row and col accordingly
int number_unassigned(int *row, int *col)
{
    int num_unassign = 0;
    int i,j;
    for(i=0;i<SIZE;i++)
    {
        for(j=0;j<SIZE;j++)
        {
            //cell is unassigned
            if(matrix[i][j] == 0)
            {
                //changing the values of row and col
                *row = i;
                *col = j;
                //there is one or more unassigned cells
                num_unassign = 1;
                return num_unassign;
            }
        }
```

```
    }
    return num_unassign;
}

//function to check if we can put a
//value in a paticular cell or not
int is_safe(int n, int r, int c)
{
    int i,j;
    //checking in row
    for(i=0;i<SIZE;i++)
    {
        //there is a cell with same value
        if(matrix[r][i] == n)
            return 0;
    }
    //checking column
    for(i=0;i<SIZE;i++)
    {
        //there is a cell with the value equal to i
        if(matrix[i][c] == n)
            return 0;
    }
    //checking sub matrix
    int row_start = (r/3)*3;
    int col_start = (c/3)*3;
    for(i=row_start;i<row_start+3;i++)
```

```
    {
        for(j=col_start;j<col_start+3;j++)
        {
            if(matrix[i][j]==n)
                return 0;
        }
    }
    return 1;
}

//function to solve sudoku
//using backtracking
int solve_sudoku()
{
    int row;
    int col;
    //if all cells are assigned then the sudoku is already solved
    //pass by reference because number_unassigned will change the values of row
and col
    if(number_unassigned(&row, &col) == 0)
        return 1;
    int n,i;
    //number between 1 to 9
    for(i=1;i<=SIZE;i++)
    {
        //if we can assign i to the cell or not
        //the cell is matrix[row][col]
```

```
        if(is_safe(i, row, col))
        {
            matrix[row][col] = i;
            //backtracking
            if(solve_sudoku())
                return 1;
            //if we can't proceed with this solution
            //reassign the cell
            matrix[row][col]=0;
        }
    }
    return 0;
}

int main()
{
    if (solve_sudoku())
        print_sudoku();
    else
        printf("No solution\n");
    return 0;
}
```

OUTPUT:

Output Clear

```
/tmp/itehq6fMxt.o
5   3   4   6   7   8   9   1   2
6   7   2   1   9   5   3   4   8
1   9   8   3   4   2   5   6   7
8   5   9   7   6   1   4   2   3
4   2   6   8   5   3   7   9   1

7   1   3   9   2   4   8   5   6

9   6   1   5   3   7   2   8   4

2   8   7   4   1   9   6   3   5

3   4   5   2   8   6   1   7   9
```

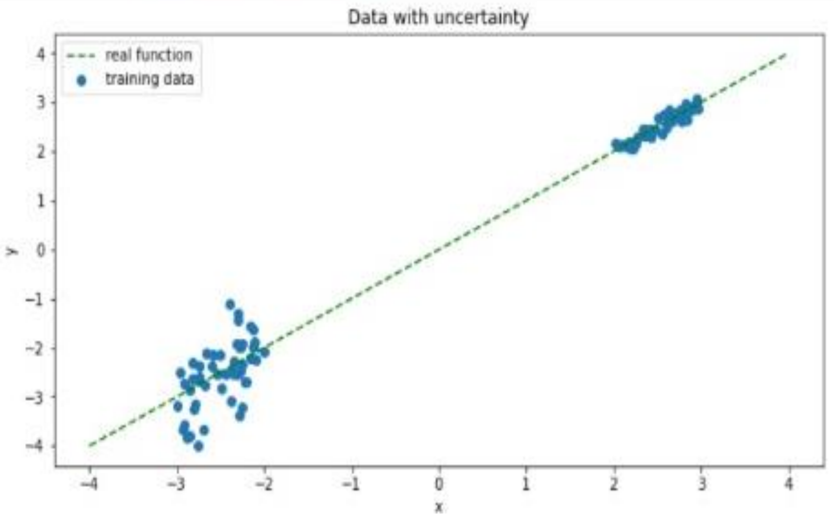
PROGRAM:

 # prediction
y_pred_without_dropout = model_without_dropout.predict(x_test)
y_pred_with_dropout = model_with_dropout.predict(x_test)

 # plotting
fig, ax = plt.subplots(1,1,figsize=(10,5))
ax.scatter(x_train, y_train, s=10, label='train data')
ax.plot(x_test, x_test, ls='--', label='test data', color='green')
ax.plot(x_test, y_pred_without_dropout, label='predicted ANN - R2 {:.2f}'.format(r2_score(x_test, y_pred_without_dropout)), color='red')
ax.plot(x_test, y_pred_with_dropout, label='predicted ANN Dropout - R2 {:.2f}'.format(r2_score(x_test, y_pred_with_dropout)), color='black')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.set_title('test data');

OUTPUT:

```
x_train = np.concatenate([x_train, np.random.uniform(2, 3, 50)])
y_train = np.concatenate([y_train, x_train[50:] + np.random.randn(*x_train[50:].shape)*0.1])
x_test = np.linspace(-10,10,100)

fig, ax = plt.subplots(1,1,figsize=(10,5))
ax.scatter(x_train, y_train, label='training data')
ax.plot(x_func, y_func, ls='--', label='real function', color='green')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.set_title('Data with uncertainty');
```



PROGRAM:

#Base Classes

#PREDICATE - ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY

```
class PREDICATE:
  def __str__(self):
    pass
  def __repr__(self):
    pass
  def __eq__(self, other) :
    pass
  def __hash__(self):
    pass
  def get_action(self, world_state):
    pass
```

#OPERATIONS - Stack, Unstack, Pickup, Putdown

```
class Operation:
  def __str__(self):
    pass
  def __repr__(self):
    pass
  def __eq__(self, other) :
    pass
  def precondition(self):
    pass
  def delete(self):
```

```
    pass
  def add(self):
    pass

class ON(PREDICATE):
  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "ON({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
      return hash(str(self))

  def get_action(self, world_state):
    return StackOp(self.X,self.Y)

class ONTABLE(PREDICATE):
  def __init__(self, X):
    self.X = X
```

```
  def __str__(self):
    return "ONTABLE({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
      return hash(str(self))

  def get_action(self, world_state):
    return PutdownOp(self.X)

class CLEAR(PREDICATE):
  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "CLEAR({X})".format(X=self.X)
    self.X = X

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
```

```
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
    return hash(str(self))

  def get_action(self, world_state):
    for predicate in world_state:
      #If Block is on another block, unstack
      if isinstance(predicate,ON) and predicate.Y==self.X:
        return UnstackOp(predicate.X, predicate.Y)
    return None

class HOLDING(PREDICATE):
  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "HOLDING({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
    return hash(str(self))
```

```
  def get_action(self, world_state):
    X = self.X
    #If block is on table, pick up
    if ONTABLE(X) in world_state:
      return PickupOp(X)
    #If block is on another block, unstack
    else:
      for predicate in world_state:
        if isinstance(predicate,ON) and predicate.X==X:
          return UnstackOp(X,predicate.Y)

class ARMEMPTY(PREDICATE):
  def __init__(self):
    pass

  def __str__(self):
    return "ARMEMPTY"

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
return hash(str(self))
```

```
  def get_action(self, world_state=[]):
    for predicate in world_state:
      if isinstance(predicate,HOLDING):
        return PutdownOp(predicate.X)
    return None

class StackOp(Operation):
  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "STACK({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def delete(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]
  def add(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) ]
```

```
class UnstackOp(Operation):
  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "UNSTACK({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) , CLEAR(self.X) ]

  def delete(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) ]

  def add(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]


class PickupOp(Operation):
  def __init__(self, X):
```

```
    self.X = X

  def __str__(self):
    return "PICKUP({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
    return [ CLEAR(self.X) , ONTABLE(self.X) , ARMEMPTY() ]

  def delete(self):
    return [ ARMEMPTY() , ONTABLE(self.X) ]

  def add(self):
    return [ HOLDING(self.X) ]

class PutdownOp(Operation):
  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "PUTDOWN({X})".format(X=self.X)
```

```
 def __repr__(self):
   return self.__str__()

  def __eq__(self, other) :
   return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
   return [ HOLDING(self.X) ]

  def delete(self):
   return [ HOLDING(self.X) ]

  def add(self):
   return [ ARMEMPTY() , ONTABLE(self.X) ]

def isPredicate(obj):
  predicates = [ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY]
  for predicate in predicates:
    if isinstance(obj,predicate):
      return True
  return False

def isOperation(obj):
  operations = [StackOp, UnstackOp, PickupOp, PutdownOp]
  for operation in operations:
    if isinstance(obj,operation):
      return True
```

```
  return False

def arm_status(world_state):
  for predicate in world_state:
    if isinstance(predicate, HOLDING):
      return predicate
  return ARMEMPTY()

class GoalStackPlanner:
  def __init__(self, initial_state, goal_state):
    self.initial_state = initial_state
    self.goal_state = goal_state

  def get_steps(self):
    #Store Steps
    steps = []
    #Program Stack
    stack = []
    #World State/Knowledge Base
    world_state = self.initial_state.copy()

    #Initially push the goal_state as compound goal onto the stack
    stack.append(self.goal_state.copy())



    #Repeat until the stack is empty
    while len(stack)!=0:
```

```
#Get the top of the stack
stack_top = stack[-1]

#If Stack Top is Compound Goal, push its unsatisfied goals onto stack
if type(stack_top) is list:
  compound_goal = stack.pop()
  for goal in compound_goal:
    if goal not in world_state:
      stack.append(goal)

#If Stack Top is an action
elif isOperation(stack_top):

  #Peek the operation
  operation = stack[-1]

  all_preconditions_satisfied = True

  #Check if any precondition is unsatisfied and push it onto program stack
  for predicate in operation.delete():
    if predicate not in world_state:
      all_preconditions_satisfied = False
      stack.append(predicate)
  #If all preconditions are satisfied, pop operation from stack and execute it
  if all_preconditions_satisfied:
```

```
    stack.pop()
    steps.append(operation)

    for predicate in operation.delete():
      world_state.remove(predicate)
    for predicate in operation.add():
      world_state.append(predicate)



#If Stack Top is a single satisfied goal
elif stack_top in world_state:
  stack.pop()

#If Stack Top is a single unsatisfied goal
else:
  unsatisfied_goal = stack.pop()

  #Replace Unsatisfied Goal with an action that can complete it
  action = unsatisfied_goal.get_action(world_state)

  stack.append(action)

  #Push Precondition on the stack
  for predicate in action.precondition():
    if predicate not in world_state:
      stack.append(predicate)
```

```
        return steps

if __name__ == '__main__':
  initial_state = [
    ON('B','A'),
    ONTABLE('A'),ONTABLE('C'),ONTABLE('D'),
    CLEAR('B'),CLEAR('C'),CLEAR('D'),
    ARMEMPTY()
  ]

  goal_state = [
    ON('B','D'),ON('C','A'),
    ONTABLE('D'),ONTABLE('A'),
    CLEAR('B'),CLEAR('C'),
    ARMEMPTY()
  ]

  goal_stack = GoalStackPlanner(initial_state=initial_state,
goal_state=goal_state)
  steps = goal_stack.get_steps()
  print(steps)
```

OUTPUT:

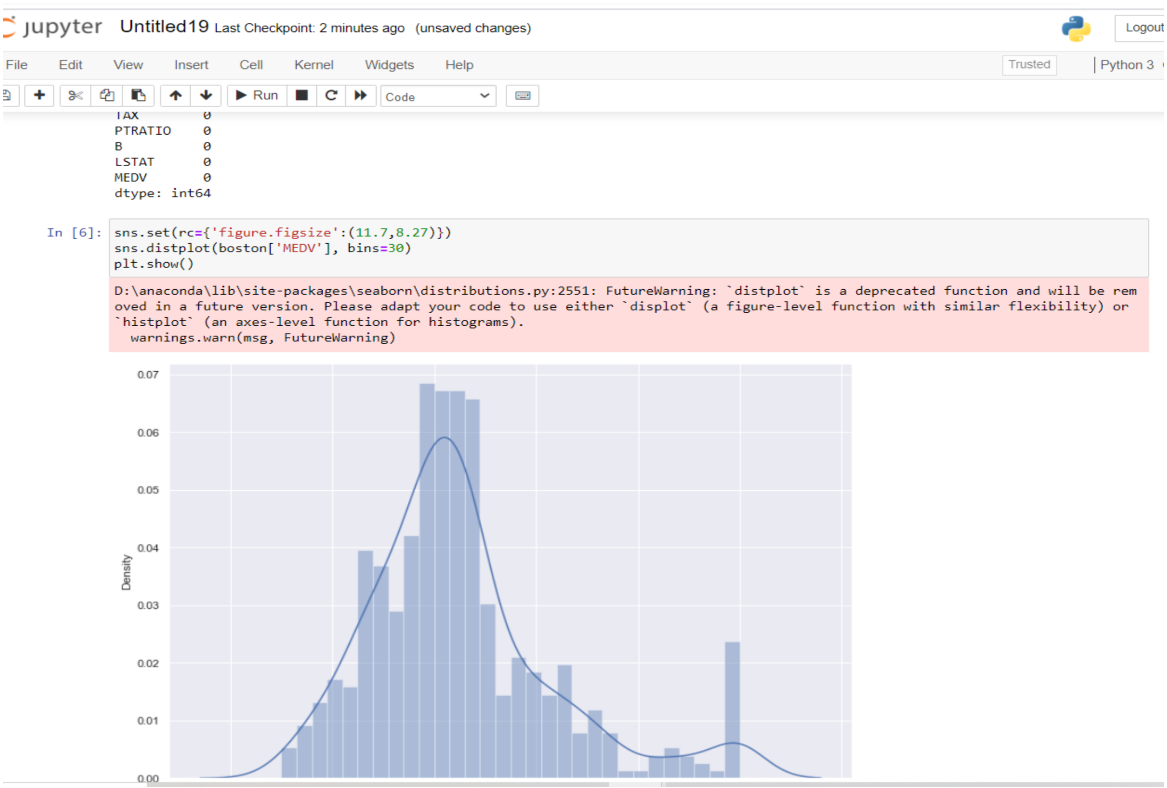
```
Shell                                                                Clear
[PICKUP(C), PUTDOWN(C), UNSTACK(B,A), PUTDOWN(B), PICKUP(C), STACK(C,A), PICKUP(B),
    STACK(B,D)]
>
```

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

import pandas as pd
import seaborn as sns

%matplotlib inline
from sklearn.datasets import load_boston
boston_dataset = load_boston()
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston.head()
boston['MEDV'] = boston_dataset.target
boston.isnull().sum()
sns.set(rc={'figure.figsize':(11.7,8.27)})
sns.distplot(boston['MEDV'], bins=30)
plt.show()
correlation_matrix = boston.corr().round(2)
# annot = True to print the values inside the square
sns.heatmap(data=correlation_matrix, annot=True)
plt.figure(figsize=(20, 5))

features = ['LSTAT', 'RM']
target = boston['MEDV']

for i, col in enumerate(features):
    plt.subplot(1, len(features) , i+1)
```

```
x = boston[col]
y = target
plt.scatter(x, y, marker='o')
plt.title(col)
plt.xlabel(col)
plt.ylabel('MEDV')
```

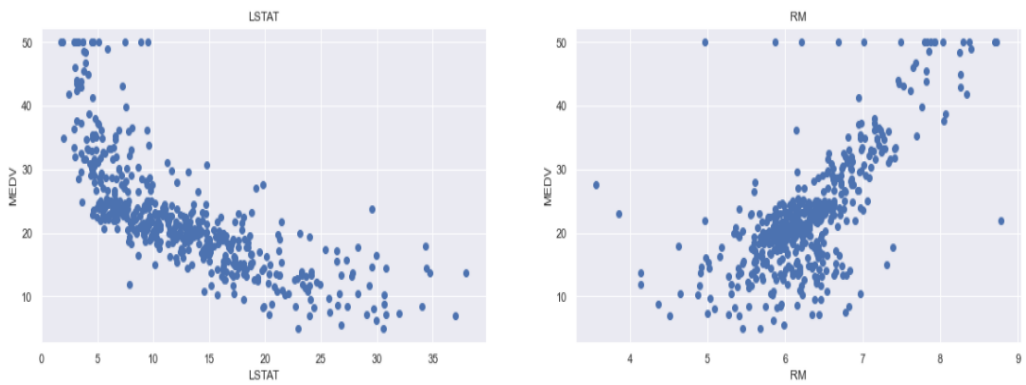
OUTPUT:



```
[7]: correlation_matrix = boston.corr().round(2)
     # annot = True to print the values inside the square
     sns.heatmap(data=correlation_matrix, annot=True)
```

t[7]: <AxesSubplot:>


```
In [8]: plt.figure(figsize=(20, 5))

        features = ['LSTAT', 'RM']
        target = boston['MEDV']

        for i, col in enumerate(features):
            plt.subplot(1, len(features) , i+1)
            x = boston[col]
            y = target
            plt.scatter(x, y, marker='o')
            plt.title(col)
            plt.xlabel(col)
            plt.ylabel('MEDV')
```




```
In [ ]:
```

PROGRAM:

 # Load libraries

from sklearn.ensemble import AdaBoostClassifier

from sklearn import datasets

 # Import train_test_split function

from sklearn.model_selection import train_test_split

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

 # Load data

iris = datasets.load_iris()

X = iris.data

y = iris.target

 # Split dataset into training set and test set

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training and 30% test

 # Create adaboost classifer object

abc = AdaBoostClassifier(n_estimators=50,

learning_rate=1)

 # Train Adaboost Classifer

model = abc.fit(X_train, y_train)

#Predict the response for test dataset

y_pred = model.predict(X_test)

 # Model Accuracy, how often is the classifier correct?

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

 # Load libraries

```
from sklearn.ensemble import AdaBoostClassifier
# Import Support Vector Classifier
from sklearn.svm import SVC
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
svc=SVC(probability=True, kernel='linear')

# Create adaboost classifer object
abc =AdaBoostClassifier(n_estimators=50, base_estimator=svc,learning_rate=1)

# Train Adaboost Classifer
model = abc.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = model.predict(X_test)
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

<u>OUTPUT:</u>

Edit View Insert Cell Kernel Widgets Help Trusted

Run Code

In [3]:
```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training and 30% test
```

In [4]:
```
# Create adaboost classifer object
abc = AdaBoostClassifier(n_estimators=50,
                         learning_rate=1)
# Train Adaboost Classifer
model = abc.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = model.predict(X_test)
```

In [5]:
```
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9777777777777777
```

In [6]:
```
# Load libraries
from sklearn.ensemble import AdaBoostClassifier

# Import Support Vector Classifier
from sklearn.svm import SVC
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
svc=SVC(probability=True, kernel='linear')

# Create adaboost classifer object
abc =AdaBoostClassifier(n_estimators=50, base_estimator=svc,learning_rate=1)

# Train Adaboost Classifer
model = abc.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = model.predict(X_test)


# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9777777777777777
```

PROGRAM:

```
# import the necessary libraries
import nltk
import string
import re
```

1. Text Lowercase:

```
def text_lowercase(text):
        return text.lower()
input_str = "Hey, did you know that the summer break is coming? Amazing right !!
It's only 5 more days !!"
text_lowercase (input_str)
```

OUTPUT:

Input: “Hey, did you know that the summer break is coming? Amazing right!! It’s only 5 more days!!”

Output: “hey, did you know that the summer break is coming? amazing right!! it’s only 5 more days!!”

2. Remove numbers:

```
# Remove numbers
def remove_numbers(text):
        result = re.sub(r'\d+', '', text)
        return result
input_str = "There are 3 balls in this bag, and 12 in the other one."
remove_numbers(input_str)
```

Input: “There are 3 balls in this bag, and 12 in the other one.”

Output: ‘There are balls in this bag, and in the other one.’

3. Convert numbers into words

```
# import the inflect library
import inflect
```

```
p = inflect.engine()

# convert number into words
def convert_number(text):
    # split string into list of words
    temp_str = text.split()
    # initialise empty list
    new_string = []

    for word in temp_str:
            # if word is a digit, convert the digit
            # to numbers and append into the new_string list
            if word.isdigit():
                    temp = p.number_to_words(word)
                    new_string.append(temp)

            # append the word as it is
            else:
                    new_string.append(word)

    # join the words of new_string to form a string
    temp_str = ' '.join(new_string)
    return temp_str

input_str = 'There are 3 balls in this bag, and 12 in the other one.'
convert_number(input_str)
```

Input: “There are 3 balls in this bag, and 12 in the other one.”

Output: “There are three balls in this bag, and twelve in the other one.”

4. **Remove punctuation:**

```
# remove punctuation
def remove_punctuation(text):
    translator = str.maketrans('', '', string.punctuation)
    return text.translate(translator)

input_str = "Hey, did you know that the summer break is coming? Amazing right
!! It's only 5 more days !!"
```

```
remove_punctuation(input_str)
```

Input: “Hey, did you know that the summer break is coming? Amazing right!! It’s only 5 more days!!”
Output: “Hey did you know that the summer break is coming Amazing right Its only 5 more days”

5. **Remove whitespaces:**

```
# remove whitespace from text
def remove_whitespace(text):
    return " ".join(text.split())

input_str = " we don't need the given questions"
remove_whitespace(input_str)
```

Input: "   we don't need   the given questions"
Output: "we don't need the given questions"

6. **Remove default stopwords:**

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# remove stopwords function
def remove_stopwords(text):
    stop_words = set(stopwords.words("english"))
    word_tokens = word_tokenize(text)
    filtered_text = [word for word in word_tokens if word not in stop_words]
    return filtered_text

example_text = "This is a sample sentence and we are going to remove the stopwords from this."
remove_stopwords(example_text)
```

The NLTK library has a set of stopwords and we can use these to remove stopwords from our text and return a list of word tokens.

```
LIST OF ENGLISH STOPWORDS IN NLTK:

their, few, wasn't, has, m, or, did, isn, very, themselves, you've, you'd, do,
between, other, t, shan, yourself, does, ours, i, it, should, what, himself, so
me, itself, there, weren, most, her, mustn, hers, doesn, won, doesn't, hasn, s,
y, wouldn't, didn't, him, couldn, after, a, will, ain, than, for, being, which,
during, ll, my, isn't, its, any, hadn't, his, then, don, of, shouldn't, out, ou
r, have, such, o, nor, too, re, should've, needn't, same, she's, but, weren't,
all, against, down, don't, can, you, under, where, wouldn, only, been, aren't,
haven, that, doing, if, up, d, needn, ma, yours, shan't, wasn, because, about,
those, he, are, was, at, hasn't, over, until, had, with, you're, below, have
n't, mightn, here, own, off, both, whom, while, as, ourselves, they, further, m
ightn't, these, from, to, them, she, who, were, more, am, why, your, aren, had
n, in, won't, yourselves, no, me, didn, an, so, before, is, on, now, each, how,
be, theirs, shouldn, mustn't, above, herself, just, you'll, the, through, agai
n, once, having, by, when, myself, we, it's, this, that'll, couldn't, ve, and,
into, not,
```

Input: “This is a sample sentence and we are going to remove the stopwords from this”
Output: [‘This’, ‘sample’, ‘sentence’, ‘going’, ‘remove’, ‘stopwords’]

7. **Stemming:**
Stemming is the process of getting the root form of a word. Stem or root is the part to which inflectional affixes (-ed, -ize, -de, -s, etc.) are added. The stem of a word is created by removing the prefix or suffix of a word. So, stemming a word may not result in actual words.
Example:
books ---> book
looked ---> look
denied ---> deni
flies ---> fli

	words	stemmed words
0	connect	connect
1	connected	connect
2	connection	connect
3	connections	connect
4	connects	connect

	words	stemmed words
0	friend	friend
1	friends	friend
2	friended	friend
3	friendly	friendli

```
from nltk.stem.porter import PorterStemmer
from nltk.tokenize import word_tokenize
stemmer = PorterStemmer()

# stem words in the list of tokenized words
def stem_words(text):
    word_tokens = word_tokenize(text)
    stems = [stemmer.stem(word) for word in word_tokens]
    return stems

text = 'data science uses scientific methods algorithms and many types of processes'
stem_words(text)
```

Input: ‘data science uses scientific methods algorithms and many types of processes’
Output: [‘data’, ‘scienc’, ‘use’, ‘scientif’, ‘method’, ‘algorithm’, ‘and’, ‘mani’, ‘type’, ‘of’, ‘process’]

8. Lemmatization:

Lemmatization also converts a word to its root form. The only difference is that lemmatization ensures that the root word belongs to the language.

```
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
lemmatizer = WordNetLemmatizer()

# lemmatize string
def lemmatize_word(text):
    word_tokens = word_tokenize(text)
    # provide context i.e. part-of-speech
    lemmas = [lemmatizer.lemmatize(word, pos ='v') for word in word_tokens]
    return lemmas

text = 'data science uses scientific methods algorithms and many types of processes'
lemmatize_word(text)
```

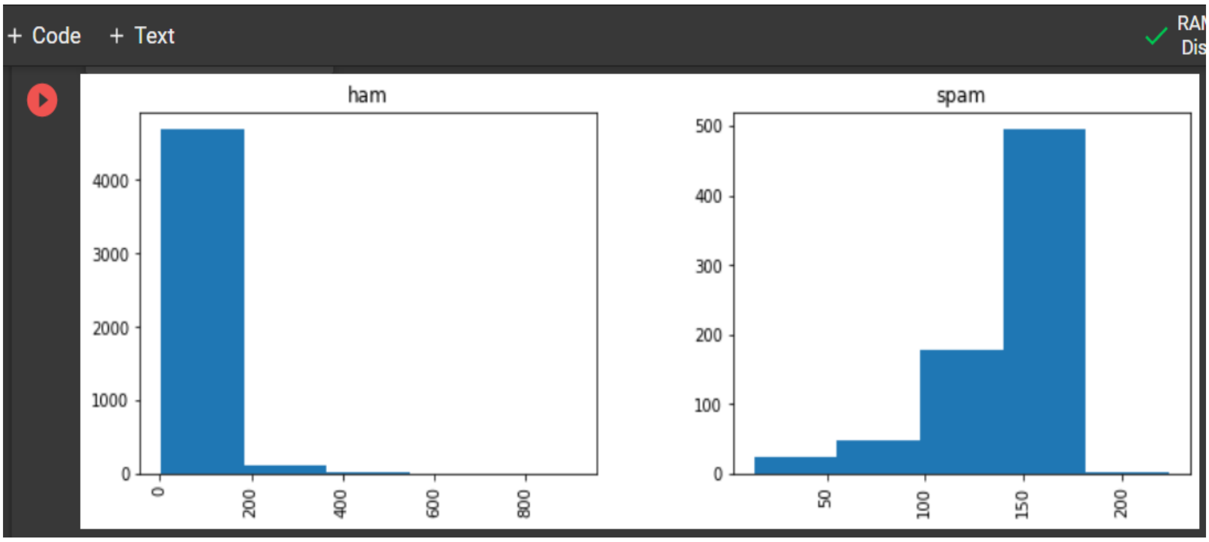
Input: ‘data science uses scientific methods algorithms and many types of processes’
Output: [‘data’, ‘science’, ‘use’, ‘scientific’, ‘methods’, ‘algorithms’, ‘and’, ‘many’, ‘type’, ‘of’, ‘process’]

PROGRAM:

```
import pandas as pd
import numpy as np
import string
import seaborn as sns
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from collections import Counter
from sklearn.metrics import classification_report,confusion_matrix
from sklearn.model_selection import GridSearchCV
%matplotlib inline

# Load data
data = pd.read_excel('data.xlsx')

# Rename names columns
data.columns = ['label', 'messages']
data["length"] = data["messages"].apply(len)
data.sort_values(by='length', ascending=False).head(10)
data.hist(column = 'length', by ='label',figsize=(12,4), bins = 5)
def transform_message(message):
message_not_punc = []

# Message withoutpunctuation
for punctuation in message:
    if punctuation not in string.punctuation:
        message_not_punc.append(punctuation)

# Join words again to form the string.
     message_not_punc =
   ''.join(message_not_punc)

# Remove any stopwords for message_not_punc, but first we should
# to transform this into the list.
message_clean = list(message_not_punc.split(" "))
while i <= len(message_clean):
for mess in message_clean:

if mess.lower() in stopwords.words('english'):  message_clean.remove(mess)
   i =i +1
```

```
    return   message_clean
    vectorization = CountVectorizer(analyzer = transform_message )
    X =
vectorization.fit(data['
messages'])
X_transform = X.transform([data['messages']])



tfidf_transformer = TfidfTransformer().fit(X_transform)

X_tfidf = tfidf_transformer.transform(X_transform)

print(X_tfidf.shape)

# Classification Model


X_train, X_test, y_train, y_test = train_test_split(X_tfidf, data['messages'],
test_size=0.30,

random_state = 50)

clf = SVC(kernel='linear').fit(X_train, y_train)

# Test model


predictions = clf.predict(X_test)

print('predicted', predictions)

# Is our model reliable?


print (classification_report(y_test, predictions))

print(confusion_matrix(y_test,predictions))
```

OUTPUT:


+ Code + Text
ham
spam
4000
3000
2000
1000
0
0
200
400
600
800
500
400
300
200
100
0
50
100
150
200

PROGRAM:

```
import argparse
import json

import cv2
import editdistance
from path import Path

from DataLoaderIAM import DataLoaderIAM, Batch
from Model import Model, DecoderType
from SamplePreprocessor import preprocess


class FilePaths:
"filenames and paths to data"
fnCharList = '../model/charList.txt'
fnSummary = '../model/summary.json'
fnInfer = '../data/test.png'
fnCorpus = '../data/corpus.txt'


def write_summary(charErrorRates, wordAccuracies):
with open(FilePaths.fnSummary, 'w') as f:
json.dump({'charErrorRates': charErrorRates, 'wordAccuracies': wordAccuracies}, f)


def train(model, loader):
"train NN"
epoch = 0

# number of training epochs since start
summaryCharErrorRates = []
summaryWordAccuracies = []
bestCharErrorRate = float('inf') # best valdiation character error rate
noImprovementSince = 0 # number of epochs no improvement of character error rate
occured
earlyStopping = 25 # stop training after this number of epochs without improvement
```

```
while True:
epoch += 1
print('Epoch:', epoch)


# train
print('Train NN')
loader.trainSet()
while loader.hasNext():
iterInfo = loader.getIteratorInfo()
batch = loader.getNext()
loss = model.trainBatch(batch)

# validate
    charErrorRate, wordAccuracy =
    validate(model, loader)
# write summary
summaryCharErrorRates.append(charErrorRate)
summaryWordAccuracies.append(wordAccuracy)
write_summary(summaryCharErrorRates,
summaryWordAccuracies)
# if best validation accuracy so far, save model parameters
if charErrorRate < bestCharErrorRate:
print('Character error rate improved, save model')
bestCharErrorRate = charErrorRate
noImprovementSince = 0
model.save()
else:
print(f'Character error rate not improved, best so far: {charErrorRate * 100.0}%')
noImprovementSi
nce += 1

# stop training if no more improvement in the last x epochs
if noImprovementSince >= earlyStopping:
print(f'No more improvement since {earlyStopping} epochs. Training stopped.')
break

def validate(model, loader):
```

```
"validate NN"
print('Validate NN')
loader.validationSet()
numCharErr = 0
numCharTotal = 0
numWordOK = 0
numWordTotal = 0

while loader.hasNext():
        iterInfo = loader.getIteratorInfo()
        print(f'Batch: {iterInfo[0]} / {iterInfo[1]}')
      batch = loader.getNext()
      (recognized, _) =
     model.inferBatch(batch)

        print('Ground truth -> Recognized')
      for i in range(len(recognized)):
        numWordOK += 1 if batch.gtTexts[i] == recognized[i] else 0
      numWordTotal += 1
      dist = editdistance.eval(recognized[i], batch.gtTexts[i])
      numCharErr += dist
      numCharTotal += len(batch.gtTexts[i])
      print('[OK]' if dist == 0 else '[ERR:%d]' % dist, '"' + batch.gtTexts[i] + '"','->', '"' +
     recognized[i] + '"')
```

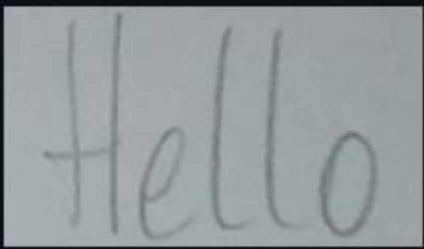
 # print validation result

```
     charErrorRate = numCharErr / numCharTotal
     wordAccuracy = numWordOK / numWordTotal
     print(f'Character error rate: {charErrorRate * 100.0}%. Word accuracy:
     {wordAccuracy *100.0}%.')
     return
     charErrorRate,
     wordAccuracy 1
def infer(model, fnImg):
"recognize text in image provided by file path"

img = preprocess(cv2.imread(fnImg, cv2.IMREAD_GRAYSCALE), Model.imgSize)
batch = Batch(None, [img])
```

```
(recognized, probability) = model.inferBatch(batch, True)
print(f'Recognized: "{recognized[0]}"')
print(f'Probability: {probability[0]}')

def main():
"main function"
parser = argparse.ArgumentParser()
parser.add_argument('--train', help='train the NN', action='store_true')
parser.add_argument('--validate', help='validate the NN', action='store_true')
parser.add_argument('--decoder', choices=['bestpath', 'beamsearch',
'wordbeamsearch'], default='bestpath',
help='CTC decoder')
parser.add_argument('--batch_size', help='batch size', type=int, default=100)
1parser.add_argument('--data_dir', help='directory containing IAM dataset',
type=Path, required=False)
parser.add_argument('--fast', help='use lmdb to load images', action='store_true')
parser.add_argument('--dump', help='dump output of NN to CSV file(s)',
action='store_true')
args = parser.parse_args()

# set chosen CTC decoder
if args.decoder == 'bestpath':
decoderType = DecoderType.BestPath
elif args.decoder == 'beamsearch':
decoderType = DecoderType.BeamSearch
elif args.decoder == 'wordbeamsearch':
decoderType = DecoderType.WordBeamSearch

# train or validate on IAM dataset
if args.train or args.validate:

# load training data, create TF model
loader = DataLoaderIAM(args.data_dir, args.batch_size, Model.imgSize,
Model.maxTextLen, args.fast)

# save characters of model for inference mode
open(FilePaths.fnCharList, 'w').write(str().join(loader.charList))
```

```
# save words contained in dataset into file
open(FilePaths.fnCorpus, 'w').write(str(' ').join(loader.trainWords +
loader.validationWords))

# execute training or validation
if args.train:
model = Model(loader.charList, decoderType)
train(model, loader)

elif args.validate:
model = Model(loader.charList, decoderType, mustRestore=True)
validate(model, loader)

# infer text on test image
else:
model = Model(open(FilePaths.fnCharList).read(), decoderType, mustRestore=True,
dump=args.dump)
infer(model, FilePaths.fnInfer) 155

if name == ' main ':
main()
```

OUTPUT:




```
> python main.py
Init with stored values from ../model/snapshot-39
Recognized: "Hello"
Probability: 0.42098119854927063
```