



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
TIRUCHIRAPPALLI
SCHOOL OF COMPUTING

18CSC305J – Artificial Intelligence

Record Work

Register No. : _____

Name of the Student : _____

Semester : _____

Programme : B.Tech. _____

REGISTER NO: _____

BONAFIDE CERTIFICATE

Certified as the bonafide record of work done by _____,
Register No. _____ of _____ (Semester/Year),
B. Tech _____
programme in the practical course **18CSC305J –Artificial Intelligence** at SRM Institute of
Science and Technology, Tiruchirappalli during the academic year 2023 – 24.

Faculty In-charge

Head of the Department

Submitted for the End Semester Examination held on _____

Examiner-1

Examiner-2

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Tiruchirappalli

SCHOOL OF COMPUTING

VISION AND MISSION OF THE DEPARTMENT

Vision

To become a world class department in imparting high-quality knowledge and in providing students a unique learning and research experience in Computer Science and Engineering.

Mission

1. To impart knowledge in cutting edge Computer Science and Engineering technologies in par with industrial standards.
2. To collaborate with renowned academic institutions to uplift innovative research and development in Computer Science and Engineering and its allied fields to serve the needs of society
3. To demonstrate strong communication skills and possess the ability to design computing systems individually as well as part of a multidisciplinary teams.
4. To instill societal, safety, cultural, environmental, and ethical responsibilities in all professional activities
5. To produce successful Computer Science and Engineering graduates with personal and professional responsibilities and commitment to lifelong learning

Program Educational Objectives (PEO)

PEO - 1 Graduates will be able to perform in technical/managerial roles ranging from design, development, problem solving to production support in software industries and R&D sectors.

PEO - 2 Graduates will be able to successfully pursue higher education in reputed institutions.

PEO - 3 Graduates will have the ability to adapt, contribute and innovate new technologies and systems in the key domains of Computer Science and Engineering.

PEO - 4 Graduates will be ethically and socially responsible solution providers and entrepreneurs in Computer Science and other engineering disciplines.

PEO - 5 Graduates will possess the additional skills in core computer science discipline with knowledge of Hardware, Software, Programming, Logic & Reasoning.

Mission of the Department to Program Educational Objectives (PEO) Mapping

	Mission Stmt -1	Mission Stmt -2	Mission Stmt -3	Mission Stmt -4	Mission Stmt -5
PEO - 1	H	H	H	H	H
PEO - 2	L	H	H	H	H
PEO - 3	H	H	M	L	H
PEO - 4	M	H	M	H	H
PEO - 5	H	H	M	M	H

H – High correlation, M – Medium Correlation, L – Low Correlation

Program Outcomes as defined by NBA (PO)

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change. **PSO –**

Program Specific Outcomes (PSO)

PSO - 1 Ability to understand client requirements and suggest solutions

PSO - 2 Ability to create Software for automation and function

PSO - 3 Ability to utilize Logic & Reasoning Skills

Mapping Program Educational Objectives (PEO) to Program Learning Outcomes.

Program Learning Outcomes (PLO)														
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Engineering Knowledge	Problem Analysis	Design & Development	Analysis, Design, Research	Modern Tool Usage	Society & Culture	Environment & Sustainability	Ethics	Individual & Team Work	Communication	Project Mgt. & Finance	Life Long Learning	PSO-1	PSO-2	PSO-3
M	M	M	M	H	-	-	-	M	L	-	H	L	L	L
M	H	H	H	H	-	-	-	M	L	-	H	M	L	M
M	H	H	M	H	-	-	-	M	L	-	H	M	L	M
M	H	M	H	H	-	-	-	M	L	-	H	M	M	M
M	H	H	H	H	-	-	-	M	L	-	H	H	M	H
L	H	M	M	H	-	-	-	H	L	-	H	H	M	H

H – High correlation, M – Medium Correlation, L – Low Correlation

COURSE DESIGN

Course Learning Outcomes (CLO):	At the end of this course, learners will be able to:
CLO – 1	Formulate a problem and build intelligent agents
CLO – 2	Apply appropriate searching techniques to solve a real-world problem
CLO – 3	Analyse the problem and infer new knowledge using suitable knowledge representation schemes
CLO – 4	Develop planning and apply learning algorithms on real world problems
CLO – 5	Design an expert system and implement natural language processing techniques
CLO – 6	Implement advance techniques in Artificial Intelligence

CLO-PO Mapping

CLOs	Program Learning Outcomes (PLO)														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Engineering Knowledge	Problem Analysis	Design & Development	Analysis, Design, Research	Modern Tool Usage	Society & Culture	Environment & Sustainability	Ethics	Individual & Team Work	Communication	Project Mgt. & Finance	Life Long Learning	PSO - 1	PSO - 2	PSO – 3
CLO1	H	H	L	-	-	-	L	-	H	H	M	M	-	-	-
CLO2	H	H	H	H	H	-	M	-	H	H	H-	M	-	-	-
CLO3	H	H	M	H	H	M	M	L	H	H	M	-	-	-	-
CLO4	H	H	H	-	H	-	-	M	H	M	H	-	-	-	-
CLO5	H	M	M	M	M	M	M	-	H	H	-	M	-	-	-

List of Experiments

<i>S.No</i>	<i>Date</i>	<i>Name of the Experiment</i>	<i>Faculty Sign</i>
1.		Implementation of toy problems	
2.		Developing agent programs for real world problems	
3.		Implementation of constraint satisfaction problems	
4.		Implementation and Analysis of DFS and BFS for an application	
5.		Developing Best first search and A* Algorithm for real world problems	
6.		Implementation of minimax algorithm for an application	
7.		Implementation of unification and resolution for real world problems.	
8.		Implementation of knowledge representation schemes - use cases	
9.		Implementation of uncertain methods for an application	
10.		Implementation of block world problem	
11.		Implementation of learning algorithms for an application	
12 & 13		Development of ensemble model for any application	
14.		Implementation of NLP programs	
15.		Applying deep learning method for Automatic Handwriting recognition	

Experiment 1: Implementation of Toy problems

- Program to Play Tic TAC Toe with the player

Code:

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

''' We will make the board using dictionary
    in which keys will be the location(i.e : top-left,mid-right,etc.)
    and initialliy it's values will be empty space and then after every move
    we will change the value according to player's choice of move. '''

theBoard = {'7': ' ', '8': ' ', '9': ' ',
            '4': ' ', '5': ' ', '6': ' ',
            '1': ' ', '2': ' ', '3': ' '}

board_keys = []

for key in theBoard:
    board_keys.append(key)

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

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

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

    turn = 'X'
    count = 0

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

        move = input()

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

        # Now we will check if player X or O has won,for every move after 5 moves.
        if count >= 5:
            if theBoard['7'] == theBoard['8'] == theBoard['9'] != ' ': # across the top
                printBoard(theBoard)
```



```

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

# If neither X nor O wins and the board is full, we'll declare the result as 'tie'.
if count == 9:
    print("\nGame Over.\n")
    print("It's a Tie!!")

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

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

    game()

if __name__ == "__main__":
    game()

```

OUTPUT:

```
  | |
--+-+--
  | |
--+-+--
  | |
It's your turn,X.Move to which place?
1
  | |
--+-+--
  | |
--+-+--
X| |
It's your turn,O.Move to which place?
2
  | |
--+-+--
  | |
--+-+--
X|O|
It's your turn,X.Move to which place?
5
  | |
--+-+--
  |X|
--+-+--
X|O|
It's your turn,O.Move to which place?
4
  | |
--+-+--
O|X|
--+-+--
...--
```

```
4
  | |
-+-+
O|X|
-+-+
X|O|
It's your turn,X.Move to which place?
9
  | |X
-+-+
O|X|
-+-+
X|O|

Game Over.

**** X won. ****
Do want to play Again?(y/n)n
```

Result:

Thus, the implementation of toy problem tic tac toe has been successfully executed.

Experiment 2: DEVELOPING AGENT PROGRAMS FOR REAL-WORLD PROBLEMS

- Program to color code all territories in Australia

Code:

CSP.py:

```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod

V = TypeVar('V') # variable type
D = TypeVar('D') # domain type

# Base class for all constraints
class Constraint(Generic[V, D], ABC):

    # The variables that the constraint is between

    def __init__(self, variables: List[V]) -> None:

        self.variables = variables

    # Must be overridden by subclasses

    @abstractmethod

    def satisfied(self, assignment: Dict[V, D]) -> bool:

        ...

# A constraint satisfaction problem consists of variables of type V
# that have ranges of values known as domains of type D and constraints
# that determine whether a particular variable's domain selection is valid
class CSP(Generic[V, D]):

    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:

        self.variables: List[V] = variables # variables to be constrained

        Self. Domains: Dict [V, List [D]] = domains # domain of each variable

        Self. Constraints: Dict [V, List [Constraint [V, D]]] = {}

        For variable in self. Variables:

            Self. Constraints [variable] = []
```

```

        If variable not in self.domains:

            Raise LookupError("Every variable should have a domain assigned to it.")

def add_constraint(self, constraint: Constraint[V, D]) -> None:

    for variable in constraint.variables:

        if variable not in self.variables:

            raise LookupError("Variable in constraint not in CSP")

        else:

            self.constraints[variable].append(constraint)

# Check if the value assignment is consistent by checking all constraints
# for the given variable against it

def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:

    for constraint in self.constraints[variable]:

        if not constraint.satisfied(assignment):

            return False

    return True

def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:

    # assignment is complete if every variable is assigned (our base case)

    if len(assignment) == len(self.variables):

        return assignment

    # get all variables in the CSP but not in the assignment

    unassigned: List[V] = [v for v in self.variables if v not in assignment]

    # get the every possible domain value of the first unassigned variable

    first: V = unassigned[0]

    for value in self.domains[first]:

        local_assignment = assignment.copy()

        local_assignment[first] = value

```

```

# if we're still consistent, we recurse (continue)

if self.consistent(first, local_assignment):

    result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)

    # if we didn't find the result, we will end up backtracking

    if result is not None:

        return result

```

COLOR_MAPPING.py:

```

class MapColoringConstraint(Constraint[str, str]):

    def __init__(self, place1: str, place2: str) -> None:

        super().__init__([place1, place2])

        self.place1: str = place1

        self.place2: str = place2

    def satisfied(self, assignment: Dict[str, str]) -> bool:

        # If either place is not in the assignment then it is not

        # yet possible for their colors to be conflicting

        if self.place1 not in assignment or self.place2 not in assignment:

            return True

        # check the color assigned to place1 is not the same as the

        # color assigned to place2

        return assignment[self.place1] != assignment[self.place2]

if __name__ == "__main__":

    variables: List[str] = ["Western Australia", "Northern Territory", "South Australia",

```

```
"Queensland", "New South Wales", "Victoria", "Tasmania"]
```

```
domains: Dict[str, List[str]] = {}
```

```
for variable in variables:
```

```
    domains[variable] = ["red", "green", "blue"]
```

```
csp: CSP[str, str] = CSP(variables, domains)
```

```
csp.add_constraint(MapColoringConstraint("Western Australia", "Northern Territory"))
```

```
csp.add_constraint(MapColoringConstraint("Western Australia", "South Australia"))
```

```
csp.add_constraint(MapColoringConstraint("South Australia", "Northern Territory"))
```

```
csp.add_constraint(MapColoringConstraint("Queensland", "Northern Territory"))
```

```
csp.add_constraint(MapColoringConstraint("Queensland", "South Australia"))
```

```
csp.add_constraint(MapColoringConstraint("Queensland", "New South Wales"))
```

```
csp.add_constraint(MapColoringConstraint("New South Wales", "South Australia"))
```

```
csp.add_constraint(MapColoringConstraint("Victoria", "South Australia"))
```

```
csp.add_constraint(MapColoringConstraint("Victoria", "New South Wales"))
```

```
csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))
```

```
solution: Optional[Dict[str, str]] = csp.backtracking_search()
```

```
if solution is None:
```

```
    print("No solution found!")
```

```
else:
```

```
    print(solution)
```

OUTPUT:

```
Python 3.8.7 (tags/v3.8.7:6503f05, Dec 21 2020, 17:59:51) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: E:\DOWNLOADS\map_coloring.py =====
{'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia': 'blue', 'Queensland': 'red', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'green'}
```

Result:

Thus, the development of real-world agents has been successfully done.

Experiment 3: Implementation of Constraint satisfaction problems

- Program to identify territories in Australia

Code:

```
"cells": [
{
"cell_type": "code",
"execution_count": 2,
"metadata": {},
"outputs": [],
"source": [
"\n",
"from typing import Generic, TypeVar, Dict, List, Optional\n",
"from abc import ABC, abstractmethod\n",
"\n",
"V = TypeVar('V') # variable type\n",
"D = TypeVar('D') # domain type\n",
"\n",
"\n",
"# Base class for all constraints\n",
"class Constraint(Generic[V, D], ABC):\n",
"    # The variables that the constraint is between\n",
"    def __init__(self, variables: List[V]) -> None:\n",
"        self.variables = variables\n",
"\n",
"    # Must be overridden by subclasses\n",
"    @abstractmethod\n",
"    def satisfied(self, assignment: Dict[V, D]) -> bool:\n",
"        ...\n",
"\n",
"\n",
"\n",
"\n",
"class CSP(Generic[V, D]):\n",
"    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:\n",
"        self.variables: List[V] = variables # variables to be constrained\n",
"        self.domains: Dict[V, List[D]] = domains # domain of each variable\n",
"        self.constraints: Dict[V, List[Constraint[V, D]]] = {}\n",
"        for variable in self.variables:\n",
"            self.constraints[variable] = []\n",
"            if variable not in self.domains:\n",
"                raise LookupError("Every variable should have a domain assigned to it.")\n",
"\n",
"    def add_constraint(self, constraint: Constraint[V, D]) -> None:\n",
"        for variable in constraint.variables:\n",
"            if variable not in self.variables:\n",
"                raise LookupError("Variable in constraint not in CSP")\n",
"            else:\n",
"                self.constraints[variable].append(constraint)\n",
"\n",
"    # Check if the value assignment is consistent by checking all constraints\n",
"    # for the given variable against it\n",
"    def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
```

```

"    for constraint in self.constraints[variable]:\n",
"        if not constraint.satisfied(assignment):\n",
"            return False\n",
"    return True\n",
"\n",
"    def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:\n",
"        # assignment is complete if every variable is assigned (our base case)\n",
"        if len(assignment) == len(self.variables):\n",
"            return assignment\n",
"\n",
"        # get all variables in the CSP but not in the assignment\n",
"        unassigned: List[V] = [v for v in self.variables if v not in assignment]\n",
"\n",
"        # get the every possible domain value of the first unassigned variable\n",
"        first: V = unassigned[0]\n",
"        for value in self.domains[first]:\n",
"            local_assignment = assignment.copy()\n",
"            local_assignment[first] = value\n",
"            # if we're still consistent, we recurse (continue)\n",
"            if self.consistent(first, local_assignment):\n",
"                result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)\n",
"                # if we didn't find the result, we will end up backtracking\n",
"                if result is not None:\n",
"                    return result\n",
"        return None
    ]
},
{
    "cell_type": "code",
    "execution_count": 3,
    "metadata": {},
    "outputs": [
        {
            "name": "stdout",
            "output_type": "stream",
            "text": [
                "{ 'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia': 'blue', 'Queensland': 'red',\n",
                "New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'green'}\n"
            ]
        }
    ],
    "source": [
        "\n",
        "\n",
        "\n",
        "\n",
        "class MapColoringConstraint(Constraint[str, str]):\n",
"    def __init__(self, place1: str, place2: str) -> None:\n",
"        super().__init__([place1, place2])\n",
"        self.place1: str = place1\n",
"        self.place2: str = place2\n",
"\n",
"    def satisfied(self, assignment: Dict[str, str]) -> bool:\n",
"        # If either place is not in the assignment then it is not\n",
"        # yet possible for their colors to be conflicting\n",
"        if self.place1 not in assignment or self.place2 not in assignment:\n",
"            return True\n",
"        # check the color assigned to place1 is not the same as the\n",
"        # color assigned to place2\n",
"        return assignment[self.place1] != assignment[self.place2]

```

```

"\n",
"\n",
"if __name__ == \"__main__\":\n",
"    variables: List[str] = [\"Western Australia\", \"Northern Territory\", \"South Australia\",\n",
"        \"Queensland\", \"New South Wales\", \"Victoria\", \"Tasmania\"]\n",
"    domains: Dict[str, List[str]] = {}\n",
"    for variable in variables:\n",
"        domains[variable] = [\"red\", \"green\", \"blue\"]\n",
"    csp: CSP[str, str] = CSP(variables, domains)\n",
"    csp.add_constraint(MapColoringConstraint(\"Western Australia\", \"Northern Territory\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"Western Australia\", \"South Australia\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"South Australia\", \"Northern Territory\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"Queensland\", \"Northern Territory\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"Queensland\", \"South Australia\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"Queensland\", \"New South Wales\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"New South Wales\", \"South Australia\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"Victoria\", \"South Australia\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"Victoria\", \"New South Wales\"))\n",
"    csp.add_constraint(MapColoringConstraint(\"Victoria\", \"Tasmania\"))\n",
"    solution: Optional[Dict[str, str]] = csp.backtracking_search()\n",
"    if solution is None:\n",
"        print(\"No solution found!\")\n",
"    else:\n",
"        print(solution)\n",
"]\n",
},\n{\n    \"cell_type\": \"code\",\n    \"execution_count\": null,\n    \"metadata\": {},\n    \"outputs\": [],\n    \"source\": [\n    ],\n    \"metadata\": {\n        \"kernel_spec\": {\n            \"display_name\": \"Python 3\",\n            \"language\": \"python\",\n            \"name\": \"python3\"\n        },\n        \"language_info\": {\n            \"codemirror_mode\": {\n                \"name\": \"ipython\",\n                \"version\": 3\n            },\n            \"file_extension\": \".py\",\n            \"mimetype\": \"text/x-python\",\n            \"name\": \"python\",\n            \"nbconvert_exporter\": \"python\",\n            \"pygments_lexer\": \"ipython3\",\n            \"version\": \"3.8.5\"\n        }\n    },\n    \"nbformat\": 4,\n    \"nbformat_minor\": 4\n}\n
```

OUTPUT:

```
{'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia': 'blue', 'Queensland': 'red', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'green'}
```

Result:

Thus, the implementation of constraint satisfaction problem has been successfully executed.

Experiment 4: Implementation and Analysis of Breath First Search and Depth First Search for an application

- Program to implement BFS and DFS

Code:

Breath First search:

```
graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

visited = [] # List to keep track of visited nodes.
queue = []   #Initialize a queue

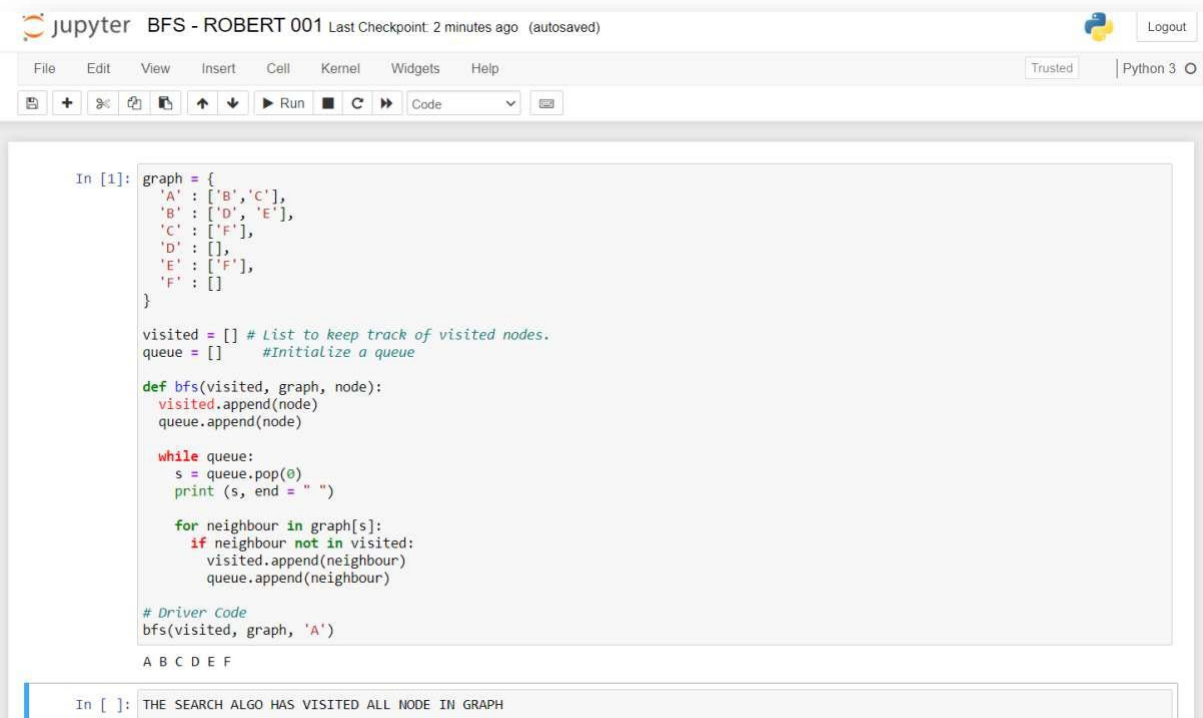
def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)

    while queue:
        s = queue.pop(0)
        print (s, end = " ")

        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'A')
```

OUTPUT:

A screenshot of a Jupyter Notebook interface. The title bar shows 'jupyter BFS - ROBERT 001' and 'Last Checkpoint: 2 minutes ago (autosaved)'. The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The toolbar has icons for saving, adding cells, zooming, and running code. The code cell contains a Python script for Breadth-First Search (BFS) on a graph. The graph is defined as a dictionary with nodes 'A' through 'F' and their neighbors. The script initializes a 'visited' list and a 'queue', then defines a 'bfs' function that processes nodes in the queue, printing them and adding their neighbors to the queue if they haven't been visited. The driver code calls 'bfs' starting from node 'A'. Below the code cell, the output shows the nodes 'A B C D E F' in a single line, followed by a message 'THE SEARCH ALGO HAS VISITED ALL NODE IN GRAPH' in the next cell.

```
In [1]: graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

visited = [] # List to keep track of visited nodes.
queue = []   #Initialize a queue

def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)

    while queue:
        s = queue.pop(0)
        print (s, end = " ")

        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'A')

A B C D E F

In [ ]: THE SEARCH ALGO HAS VISITED ALL NODE IN GRAPH
```

Depth first Search:

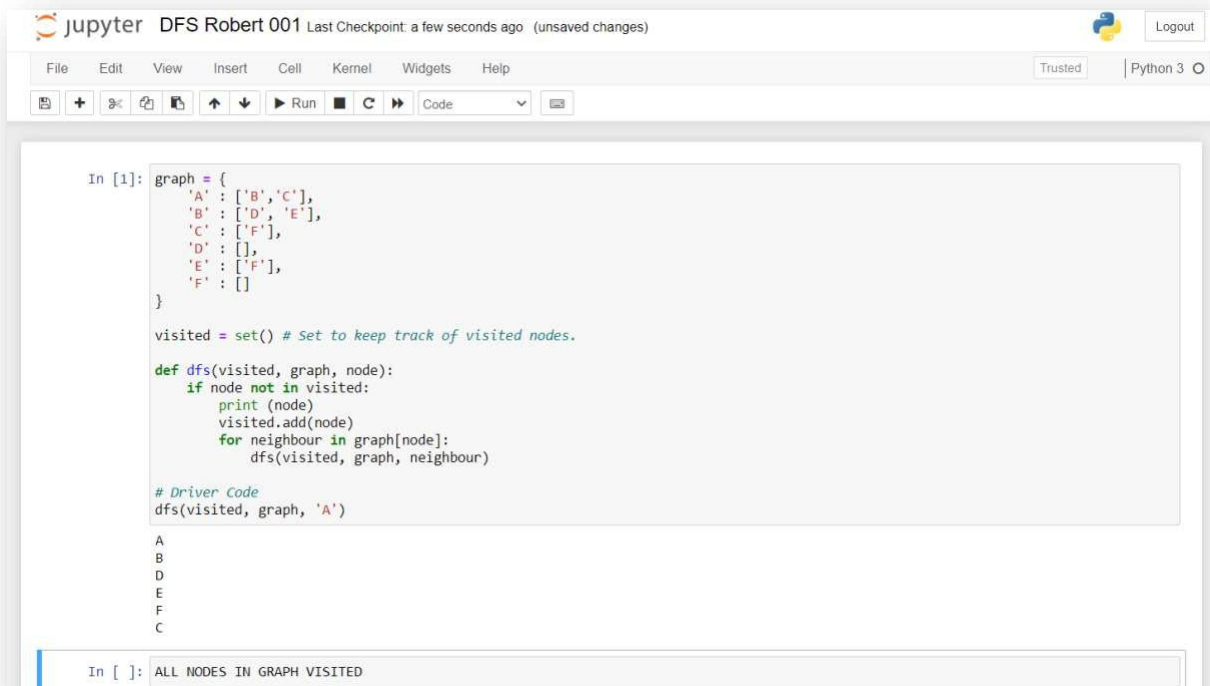
```
# Using a Python dictionary to act as an adjacency list
graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
```

OUTPUT:



A screenshot of a Jupyter Notebook interface. The title bar shows 'jupyter DFS Robert 001' and 'Last Checkpoint: a few seconds ago (unsaved changes)'. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. The toolbar contains icons for file operations, running, and code execution. The code cell contains the following Python code:

```
In [1]: graph = {
    'A' : ['B','C'],
    'B' : ['D','E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
```

The output of the code is displayed below the code cell:

```
A
B
D
E
F
C
```

At the bottom of the notebook, the status bar shows 'In []: ALL NODES IN GRAPH VISITED'.

Result:

Thus, the implementation of BFS and DFS has been successfully executed.

Experiment 5: Developing Best first search and A* Algorithm for real world problems

- Implementation of A* Algorithm via Python

Code:

```
        return path[::-1]
    children = []
    for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1),
                        (1, 1)]:
        node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])
        if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[0]) - 1) or node_position[1] < 0:
            continue
        if maze[node_position[0]][node_position[1]] != 0:
            continue
        new_node = Node(current_node, node_position)
        children.append(new_node)
    for child in children:
        for closed_child in closed_list:
            if child == closed_child:
                continue
        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
        for open_node in open_list:
            if child == open_node and child.g > open_node.g:
                continue
        open_list.append(child)

def main():
    maze = [[0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0]]
    graph = [[0, 1, 0, 0, 0, 0],
            [1, 0, 1, 0, 1, 0],
            [0, 1, 0, 0, 0, 1],
            [0, 0, 0, 0, 1, 0],
            [0, 1, 0, 1, 0, 0],
            [0, 0, 1, 0, 0, 0]]
    start = (0, 0)
    end = (5, 5)
    endl = (5, 5)
    path = astar(maze, start, end)
    print(path)
    path1 = astar(graph, start, endl)
    print(path1)

if __name__ == '__main__':
```


OUTPUT:

```
Python 3.8.7 (tags/v3.8.7:6503f05, Dec 21 2020, 17:59:51) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:/Users/Robert/AppData/Local/Programs/Python/Python38/astar.py ===
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (5, 5)]
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]
>>> |
```

- Implementation of Best First Search

Code:

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

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]

        # Displaying the path having lowest cost
        print(u, end=" ")

        if u == target:
            break

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

        print()

def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))

addege(0, 1, 3)
addege(0, 2, 6)
```

OUTPUT:

```
0 1 3 2 8 9
```

Result:

Thus, the implementation of best first search and A* algorithm has been successfully executed.

Experiment 6: Implementation of minimax algorithm for an application

- Program to implement minimax algorithm in Tic Tac Toe with Tkinter

Code:

```
# coding=UTF8
from Tkinter import Tk, Button
from tkFont import Font
from copy import deepcopy

class Board:

    def __init__(self, other=None):
        self.player = 'X'
        self.opponent = 'O'
        self.empty = '.'
        self.size = 3
        self.fields = {}
        for y in range(self.size):
            for x in range(self.size):
                self.fields[x,y] = self.empty
        # copy constructor
        if other:
            self.__dict__ = deepcopy(other.__dict__)

    def move(self, x, y):
        board = Board(self)
        board.fields[x,y] = board.player
        (board.player, board.opponent) = (board.opponent, board.player)
        return board

    def __minimax(self, player):
        if self.won():
            if player:
                return (-1, None)
            else:
                return (+1, None)
        elif self.tied():
            return (0, None)
        elif player:
            best = (-2, None)
            for x, y in self.fields:
                if self.fields[x,y]==self.empty:
                    value = self.move(x,y).__minimax(not player)[0]
                    if value > best[0]:
                        best = (value, (x,y))
            return best
        else:
            best = (+2, None)
            for x, y in self.fields:
                if self.fields[x,y]==self.empty:
                    value = self.move(x,y).__minimax(not player)[0]
                    if value < best[0]:
                        best = (value, (x,y))
            return best
```

```

def best(self):
    return self.__minimax(True)[1]

def tied(self):
    for (x,y) in self.fields:
        if self.fields[x,y]==self.empty:
            return False
    return True

def won(self):
    # horizontal
    for y in range(self.size):
        winning = []
        for x in range(self.size):
            if self.fields[x,y] == self.opponent:
                winning.append((x,y))
        if len(winning) == self.size:
            return winning
    # vertical
    for x in range(self.size):
        winning = []
        for y in range(self.size):
            if self.fields[x,y] == self.opponent:
                winning.append((x,y))
        if len(winning) == self.size:
            return winning
    # diagonal
    winning = []
    for y in range(self.size):
        x = y
        if self.fields[x,y] == self.opponent:
            winning.append((x,y))
    if len(winning) == self.size:
        return winning
    # other diagonal
    winning = []
    for y in range(self.size):
        x = self.size-1-y
        if self.fields[x,y] == self.opponent:
            winning.append((x,y))
    if len(winning) == self.size:
        return winning
    # default
    return None

def __str__(self):
    string = ""
    for y in range(self.size):
        for x in range(self.size):
            string+=self.fields[x,y]
        string+="\n"
    return string

class GUI:

    def __init__(self):
        self.app = Tk()
        self.app.title('TicTacToe')
        self.app.resizable(width=False, height=False)

```

```

self.board = Board()
self.font = Font(family="Helvetica", size=32)
self.buttons = {}
for x,y in self.board.fields:
    handler = lambda x=x,y=y: self.move(x,y)
    button = Button(self.app, command=handler, font=self.font, width=2, height=1)
    button.grid(row=y, column=x)
    self.buttons[x,y] = button
handler = lambda: self.reset()
button = Button(self.app, text='reset', command=handler)
button.grid(row=self.board.size+1, column=0, columnspan=self.board.size, sticky="WE")
self.update()

def reset(self):
    self.board = Board()
    self.update()

def move(self,x,y):
    self.app.config(cursor="watch")
    self.app.update()
    self.board = self.board.move(x,y)
    self.update()
    move = self.board.best()
    if move:
        self.board = self.board.move(*move)
    self.update()
    self.app.config(cursor="")

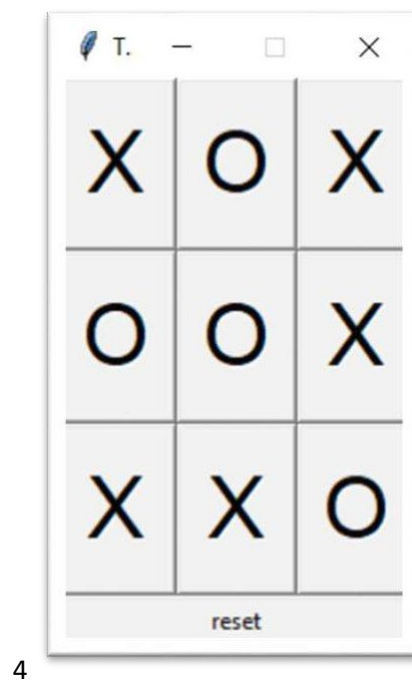
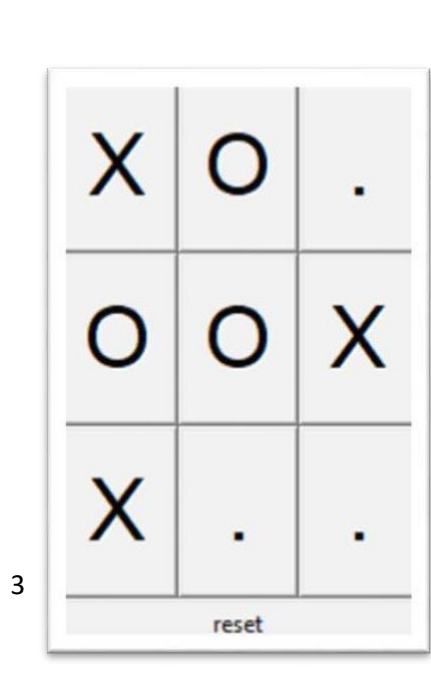
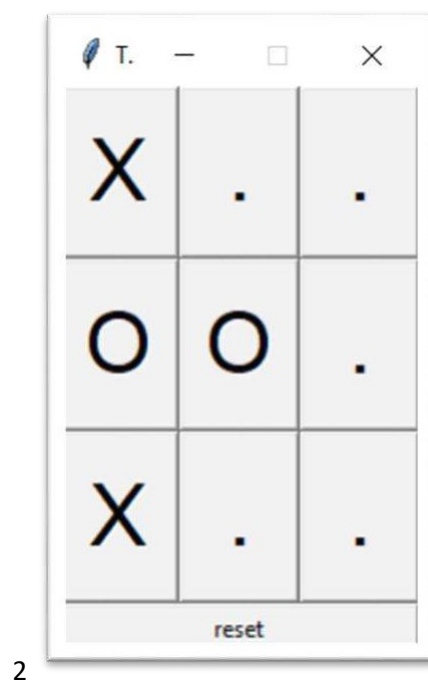
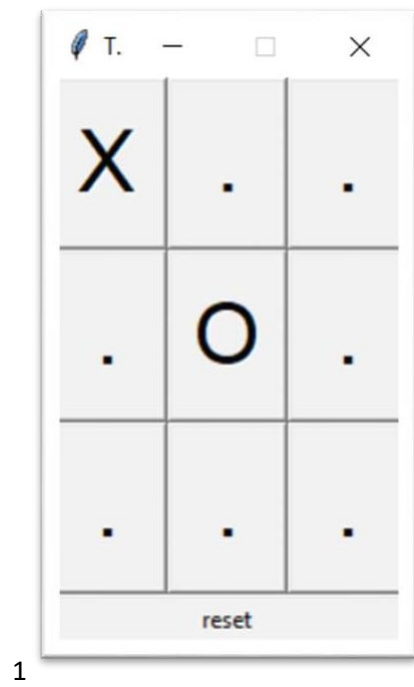
def update(self):
    for (x,y) in self.board.fields:
        text = self.board.fields[x,y]
        self.buttons[x,y]['text'] = text
        self.buttons[x,y]['disabledforeground'] = 'black'
        if text==self.board.empty:
            self.buttons[x,y]['state'] = 'normal'
        else:
            self.buttons[x,y]['state'] = 'disabled'
    winning = self.board.won()
    if winning:
        for x,y in winning:
            self.buttons[x,y]['disabledforeground'] = 'red'
        for x,y in self.buttons:
            self.buttons[x,y]['state'] = 'disabled'
    for (x,y) in self.board.fields:
        self.buttons[x,y].update()

def mainloop(self):
    self.app.mainloop()

if __name__ == '__main__':
    GUI().mainloop()

```

OUTPUT:



HERE THE DRAW IS DEPICTED SINCE It Was DELIBRATE.

Result:

Thus, the implementation of minmax algorithm has been successfully executed.

Experiment 7: Implementation of unification and resolution for real world problems

- Program to implement unification With Python

Code:

```
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  $\Psi_1$  or  $\Psi_2$  is a variable or constant, then:
        a, If  $\Psi_1$  or  $\Psi_2$  are identical, then return NULL.
        b, Else if  $\Psi_1$  is a variable:
            - then if  $\Psi_1$  occurs in  $\Psi_2$ , then return False
            - Else return ( $\Psi_2 / \Psi_1$ )
        c, Else if  $\Psi_2$  is a variable:
            - then if  $\Psi_2$  occurs in  $\Psi_1$ , then return False
            - Else return ( $\Psi_1 / \Psi_2$ )
        d, Else return False
    Step 2: If the initial Predicate symbol in  $\Psi_1$  and  $\Psi_2$  are not same, then return False.
    Step 3: IF  $\Psi_1$  and  $\Psi_2$  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  $\Psi_1$ .
        a, Call Unify function with the ith element of  $\Psi_1$  and ith element of  $\Psi_2$ , and put the
        result into S.
        b, If S = False then returns False
        c, If S  $\neq$  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!')

```

OUTPUT:

```
print('Unification failed!')

else:

    print('Unification successfully!')

    print(result)
```

Unification successfully!
f'h(A)/?'. 'f(V)/X'. 'a/?)/V'1

Result:

Thus, the implementation of unification and resolution for real world problems has been successfully executed.

Experiment 8: Implementation of knowledge representation schemes – Use cases

- Program to solve Sudoku using Knowledge Representation

Code:

```
size = 9
#empty cells have value zero
matrix = [
    [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]]

#print sudoku
def print_sudoku():
    for i in matrix:
        print (i)

#assign cells and check
def number_unassigned(row, col):
    num_unassign = 0
    for i in range(0,size):
        for j in range (0,size):
            #cell is unassigned
            if matrix[i][j] == 0:
                row = i
                col = j
                num_unassign = 1
                a = [row, col, num_unassign]
                return a
    a = [-1, -1, num_unassign]
    return a

#check validity of number
def is_safe(n, r, c):
    #checking in row
    for i in range(0,size):
        #there is a cell with same value
        if matrix[r][i] == n:
            return False
    #checking in column
```

```

    for i in range(0,size):
        #there is a cell with same value
        if matrix[i][c] == n:
            return False
    row_start = (r//3)*3
    col_start = (c//3)*3;
    #checking submatrix
    for i in range(row_start,row_start+3):
        for j in range(col_start,col_start+3):
            if matrix[i][j]==n:
                return False
    return True

#check validity of number
def solve_sudoku():
    row = 0
    col = 0
    #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
    a = number_unassigned(row, col)
    if a[2] == 0:
        return True
    row = a[0]
    col = a[1]
    #number between 1 to 9
    for i in range(1,10):
        #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 True
            #f we can't proceed with this solution
            #reassign the cell
            matrix[row][col]=0
    return False

if solve_sudoku():
    print_sudoku()
else:
    print("No solution")

```

OUTPUT:

```
[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]
```

Result:

Thus, the implementation of knowledge representation schemes & use cases has been successfully implemented.

Experiment 9: Implementation of uncertain methods for an application

- Program to solve the Monty Python Problem using Python

Code:

```
import matplotlib.pyplot as plt
import seaborn; seaborn.set_style('whitegrid')
import numpy

from pomegranate import *

numpy.random.seed(0)
numpy.set_printoptions(suppress=True)

# The guests initial door selection is completely random
guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

# The door the prize is behind is also completely random
prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

# Monty is dependent on both the guest and the prize.
monty = ConditionalProbabilityTable(
[[ 'A', 'A', 'A', 0.0 ],
 [ 'A', 'A', 'B', 0.5 ],
 [ 'A', 'A', 'C', 0.5 ],
 [ 'A', 'B', 'A', 0.0 ],
 [ 'A', 'B', 'B', 0.0 ],
 [ 'A', 'B', 'C', 1.0 ],
 [ 'A', 'C', 'A', 0.0 ],
 [ 'A', 'C', 'B', 1.0 ],
 [ 'A', 'C', 'C', 0.0 ],
 [ 'B', 'A', 'A', 0.0 ],
 [ 'B', 'A', 'B', 0.0 ],
 [ 'B', 'A', 'C', 1.0 ],
 [ 'B', 'B', 'A', 0.5 ],
 [ 'B', 'B', 'B', 0.0 ],
 [ 'B', 'B', 'C', 0.5 ],
 [ 'B', 'C', 'A', 1.0 ],
 [ 'B', 'C', 'B', 0.0 ],
 [ 'B', 'C', 'C', 0.0 ],
 [ 'C', 'A', 'A', 0.0 ],
 [ 'C', 'A', 'B', 1.0 ],
 [ 'C', 'A', 'C', 0.0 ],
 [ 'C', 'B', 'A', 1.0 ],
 [ 'C', 'B', 'B', 0.0 ],
 [ 'C', 'B', 'C', 0.0 ],
 [ 'C', 'C', 'A', 0.5 ],
 [ 'C', 'C', 'B', 0.5 ],
 [ 'C', 'C', 'C', 0.0 ]], [guest, prize])

# State objects hold both the distribution, and a high level name.
s1 = State(guest, name="guest")
s2 = State(prize, name="prize")
s3 = State(monty, name="monty")
# Create the Bayesian network object with a useful name
```

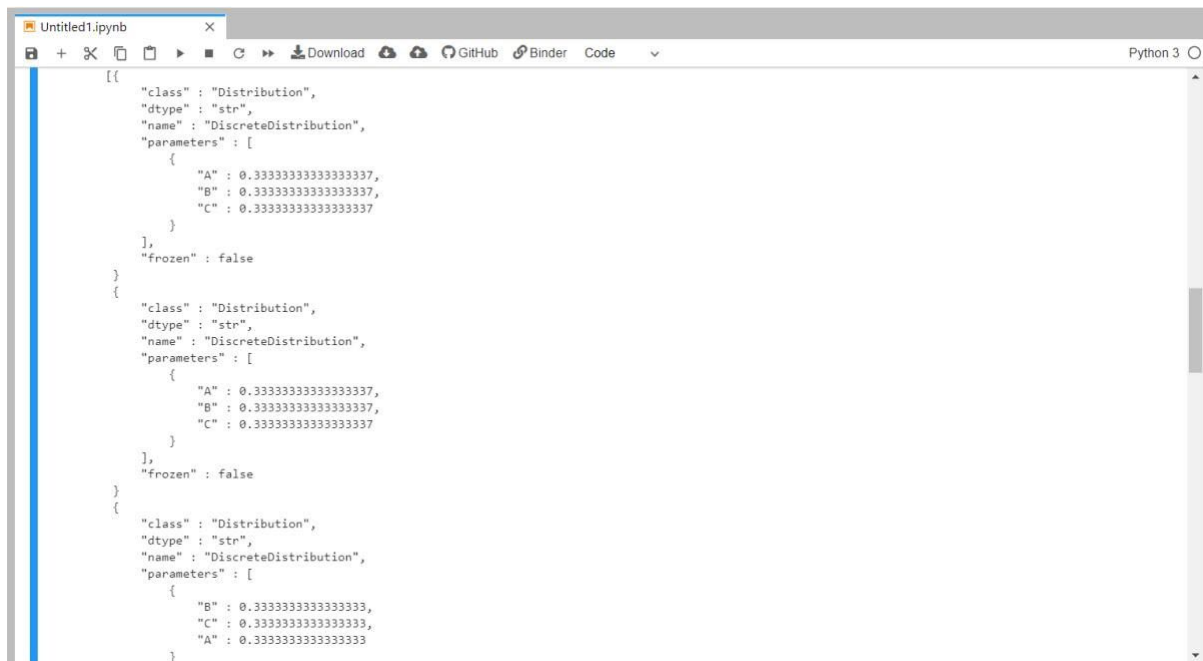
```

model = BayesianNetwork("Monty Hall Problem")

# Add the three states to the network
model.add_states(s1, s2, s3)
# Add edges which represent conditional dependencies, where the second node is
# conditionally dependent on the first node (Monty is dependent on both guest and prize)
model.add_edge(s1, s3)
model.add_edge(s2, s3)
model.bake()
model.probability(['A', 'B', 'C'])
model.probability(['A', 'B', 'C'])
print(model.predict_proba({}))
print(model.predict_proba([None, None, None]))
print(model.predict_proba(['A', None, None]))
print(model.predict_proba({'guest': 'A', 'monty': 'B'}))

```

OUTPUT:



```

[[{"class": "Distribution",
  "dtype": "str",
  "name": "DiscreteDistribution",
  "parameters": [
    {
      "A": 0.33333333333333337,
      "B": 0.33333333333333337,
      "C": 0.33333333333333337
    }
  ],
  "frozen": false
}, {"class": "Distribution",
  "dtype": "str",
  "name": "DiscreteDistribution",
  "parameters": [
    {
      "A": 0.33333333333333337,
      "B": 0.33333333333333337,
      "C": 0.33333333333333337
    }
  ],
  "frozen": false
}, {"class": "Distribution",
  "dtype": "str",
  "name": "DiscreteDistribution",
  "parameters": [
    {
      "B": 0.33333333333333333,
      "C": 0.33333333333333333,
      "A": 0.33333333333333333
    }
  ]
}]

```



```
Untitled1.ipynb
name : "DiscreteDistribution",
"parameters" : [
{
"A" : 0.3333333333333333,
"B" : 0.3333333333333333,
"C" : 0.3333333333333333
}
],
"frozen" : false
},
{
"class" : "Distribution",
"dtype" : "str",
"name" : "DiscreteDistribution",
"parameters" : [
{
"B" : 0.49999999999999994,
"C" : 0.49999999999999994,
"A" : 0.0
}
],
"frozen" : false
},
], dtype=object]]

[array(['A', {
"class" : "Distribution",
"dtype" : "str",
"name" : "DiscreteDistribution",
"parameters" : [
{
"A" : 0.33333333333333334,
"B" : 0.0,
"C" : 0.66666666666666664
}
],
"frozen" : false
}, 'B'], dtype=object)]
```

Result:

Thus, the implementation of uncertain methods for an application has been successfully executed.

Experiment 10: Implementation of block world problem

- Program to Implement Blockworld

Code:

```
import time
import sys
import os

from state import State
from searching import breadth_first_search
from utilities import read_input_file, write_output_file

def main():
    st = time.perf_counter()    #Start a time counter.

    if len(sys.argv) == 4:    #If the length of the keyword arguments is four...

        method = sys.argv[1]    #The second argument is the method/algorithm used to find a solution.

        input_file = sys.argv[2]    #The third argument is a .txt file containing the initial and final state of
        the problem.

        output_file = sys.argv[3]    #The fourth argument is a .txt file containing the solution of the
        problem.

        initial_state, goal_state = read_input_file(filename = input_file)    #Read the input file and return
        two state objects.

        if method == 'breadth':    #Check which method is selected and solve the problem accordingly.

            solution = breadth_first_search(current_state = initial_state, goal_state = goal_state, timeout =
            300)

        else:    #If the method argument is none of the above, print a usage message.

            solution = None

            print('Usage: python bw.py <method> <input filename> <output filename>')

        if solution == goal_state:    #If the solution is equal to the goal state...

            number_of_moves = write_output_file(solution = solution, filename = output_file)    #Write the
            solution file and return the number of moves.

            print('Solution found!')

            print('Number of blocks:', len(initial_state.layout.keys()))

            print('Method:', method)

            print('Number of moves:', number_of_moves)
```

```

        print('Execution time:', str(round(time.perf_counter() - st, 4)))

    else:    #Else, if the length of the keyword arguments is not equal to four, print a usage message.

        print('Usage: python bw.py <method> <input filename> <output filename>')

if __name__ == '__main__':

    main()

```

Searching.py

```

import time

from collections import deque

def breadth_first_search(current_state, goal_state, timeout = 60):
    """
    An implementation of the BFS algorithm, taking as arguments a current_state
    i.e. initial state, a goal state and an optional argument timeout (default 60)
    indicating the time before the algorithm stops if no solution is found.

    A queue is used as a structure for storing the nodes/states and a set for keeping the
    ids of the discovered states in order to check quicker whether a node has been discovered.
    """

    Q = deque([])    #A queue for storing the nodes/states.

    discovered = set()    #A set for keeping the ids of the discovered states.

    Q.append(current_state)    #Add the current/initial state to the queue.
    discovered.add(current_state.id)    #Add the id of the state to the set.

    st = time.perf_counter()    #Start a time counter.

    while Q:    #While Q is not empty...

        if time.perf_counter() - st > timeout:    #If the execution time exceeds the timeout...

            print("Timeout!")

            return None    #Break.

        state = Q.popleft()    #Dequeue an element from the left of Q.

```

```

if state == goal_state:    #If the state is the goal state, return it and break.
    return state

children = state.calcChildren()    #Else, calculate the children of this state.

for child in children:    #For each child...

    if child.id not in discovered:    #If this child has not been discovered...

        discovered.add(child.id)    #Mark it as discovered.

        child.parent = state    #Set the parent attribute of the child to be the state that has been
dequeued.

        Q.append(child)    #Append the child to Q.

```

State.py

```

from copy import deepcopy

```

```

class State:

```

```

    def __init__(self, layout, parent = None, move = [], distance = 0):

```

```

        self.layout = layout

```

```

        self.parent = parent

```

```

        self.move = move

```

```

        self.distance = distance

```

```

        values = list(self.layout.values())    #A list of the names of the blocks.

```

```

        self.id = ".join([str(i) for s in values for i in s])    #Create the id attribute.

```

```

    def __eq__(self, other_state):

```

```

        if other_state != None:

```

```

            return self.id == other_state.id

```

```

        else:

```

```

            return False

```

```

    def calcChildren(self):

```

```

        layout = self.layout

```

```

        children = []

```

```

        free_blocks = [key for key in layout if layout[key][1] == 'c']    #The blocks that can be moved.

```

```

        for moving_block in free_blocks:    #For each free block that will be moved...

```

```

            for target_block in free_blocks:

```

```

                if moving_block != target_block:

```

```

                    temp = deepcopy(layout)    #Copy the current layout in order to alter it.

```

```

                    move = []

```

```

        distance = 0

        released_block = temp[moving_block][0]    #The 'released_block' is the first item of the
list in layout with key == moving_block.

        temp[moving_block][0] = target_block    #The 'moving_block' now is on top of the
'target_block'.

        temp[target_block][1] = 'u'    #And the 'target_block' is now unclear.

        move.append(moving_block) #Add the 'moving_block' to 'move' list.

        if released_block != '-':    #If the 'released_block' is not '-' i.e. is not on the table...

            temp[released_block][1] = 'c'    #Set the block clear.

            move.append(released_block)    #Add the 'released_block' to 'move' list.

        else:

            move.append('table')

            move.append(target_block)    #Add the 'target_block' to 'move' list.

            distance = self.distance + 1 #The distance of the child is the distance of the parent plus
1.

            children.append(State(layout = temp, parent = self, move = move, distance = distance))
#Add to 'children' list a new State object.

            if layout[moving_block][0] != '-':    #If the 'moving_block' is not currently on the table, create a
state that it is.

                temp = deepcopy(layout)

                move = []

                distance = 0

                released_block = temp[moving_block][0]    #The 'released_block' is the first item of the list
in layout with key == moving_block.

                temp[moving_block][0] = '-'

                temp[released_block][1] = 'c'    #Set the block clear.

                move.append(moving_block)    #Add the 'moving_block' to 'move' list.

                move.append(released_block)    #Add the 'released_block' to 'move' list.

                move.append('table')

                distance = self.distance + 1    #The distance of the child is the distance of the parent plus
1.

                children.append(State(layout = temp, parent = self, move = move, distance = distance))
#Add to 'children' list a new State object.

            return children    #Return the children list

```

Utilities.py

```
from state import State
```

```
def read_input_file(filename):
```

```
    with open('problems/' + filename) as f:
```

```
        lines = [line.split() for line in f]    #Read the file by line and split it.
```

```
        blocks_names = lines[0][1:]    #Get the blocks names.
```

```
        blocks_names[-1] = blocks_names[-1][:-1]    #Remove the parenthesis from the last block name.
```

```
        initial = lines[1][1:-1]    #Get the initial state.
```

```
        initial = [i.replace('(', ')') for i in initial]    #Remove the parentheses.
```

```
        initial = [i.replace(')', '(') for i in initial]
```

```
        goal = lines[2][2:]    #Get the goal state.
```

```
        goal = [i.replace('(', ')') for i in goal]    #Remove the parentheses.
```

```
        goal = [i.replace(')', '(') for i in goal]
```

```
        initial_layout = {key: ['- ', 'c'] for key in blocks_names}    #Construct the initial layout.
```

```
        goal_layout = {key: ['- ', 'c'] for key in blocks_names}    #Construct the goal layout.
```

```
        for i in range(len(initial)):
```

```
            if initial[i] == 'ON':
```

```
                initial_layout[initial[i + 1]][0] = initial[i + 2]
```

```
                initial_layout[initial[i + 2]][1] = 'u'
```

```
        for i in range(len(goal)):
```

```
            if goal[i] == 'ON':
```

```
                goal_layout[goal[i + 1]][0] = goal[i + 2]
```

```
                goal_layout[goal[i + 2]][1] = 'u'
```

```
    return State(layout = initial_layout), State(layout = goal_layout)    #Return two state objects.
```

```
def write_output_file(solution, filename):
```

```
    current_state = solution    #The state we start, which is the last i.e. the solution.
```

```
    path = []    #The path from the solution towards the initial state.
```

```
    i = 1
```

```
    while True:
```

```
        path.append(current_state)    #Add the current state i.e. solution, in the list.
```

```
        current_state = current_state.parent    #The current state now becomes the parent of it.
```

```
        if current_state.parent == None:    #If the current state has no parent...
```

```

        path.append(current_state)    #Add the current state in the list.

        break

    path.reverse()    #Reverse the list.

    with open(filename, 'w') as f:    #Open the output file.

        for state in path[1:]:    #For every state in the path, except the first one which is the initial state
            that has no previous move...

                move = state.move    #Get the move.

                f.write(str(i) + '. Move(' + move[0] + ', ' + move[1] + ', ' + move[2] + ')\n')    #Write the move.

                i += 1    #Increment the counter.

    return len(path[1:])    #Return the number of steps.

```

OUTPUT:

```

Anaconda Prompt (ANACONDA)
OBJECTS: ['B', 'A', 'C']

##### INITIAL STATE #####

['CLEAR-C', 'CLEAR-A', 'ONTABLE-A', 'ONTABLE-B', 'ON-C-B']

##### GOAL STATE #####

['ON-C-B', 'ON-B-A']

##### SOLUTION #####

B: {'CLEAR': False, 'ON': 'A', 'UNDER': 'C', 'ONTABLE': False}
A: {'CLEAR': False, 'ON': -1, 'UNDER': 'B', 'ONTABLE': True}
C: {'CLEAR': True, 'ON': 'B', 'UNDER': -1, 'ONTABLE': False}
Number of moves: 5
Took: 0.0944068431854248

(base) C:\Users\afmik\Dropbox\My PC (LAPTOP-TGP7F9HA)\Desktop\block-world-problem-ai-master>

```

Result:

Thus, the implementation of block world problem has been successfully executed.

Experiment 11: Implementation of Learning Algorithms for an Application

- Implementation of Linear Regression algorithm

Code:

```
import numpy as np

import matplotlib.pyplot as plt

def estimate_coef(x, y):
    n = np.size(x)
    m_x = np.mean(x)
    m_y = np.mean(y)
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x
    return (b_0, b_1)

def plot_regression_line(x, y, b):
    plt.scatter(x, y, color = "m",
                marker = "o", s = 30)

    y_pred = b[0] + b[1]*x

    plt.plot(x, y_pred, color = "g")

    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()

def main():
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])
    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {} \
        \nb_1 = {}".format(b[0], b[1]))
```



```
plot_regression_line(x, y, b)
```

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

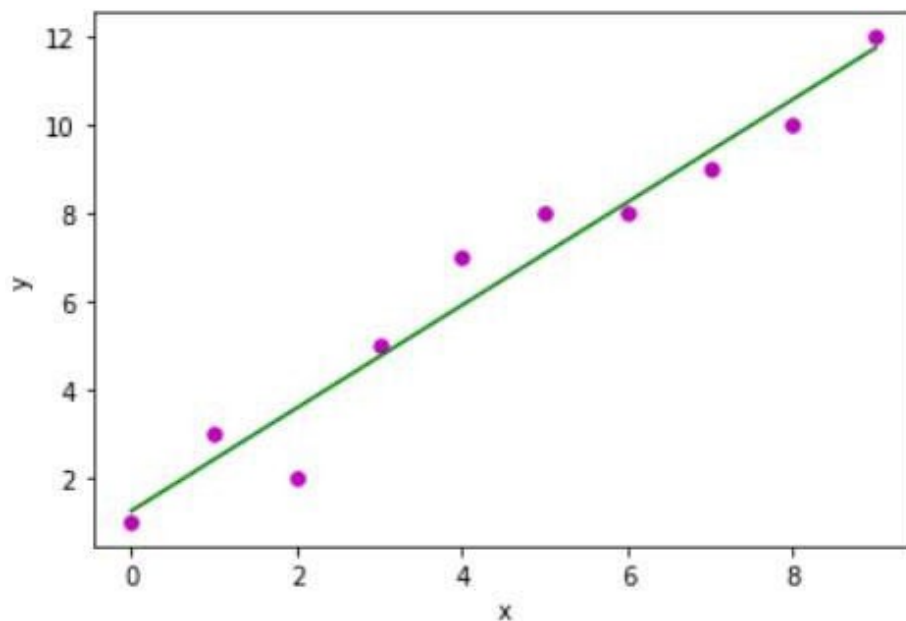
```
    main()
```

OUTPUT:

```
Estimated coefficients:
```

```
b_0 = 1.2363636363636363
```

```
b_1 = 1.1696969696969697
```



Result:

Thus, the implementation of learning algorithms for an application has been successfully executed.

Experiment 12 & 13: Development of ensemble model for any application

- To predict whether a bank currency note is authentic or not based on four attributes using Random Forest classification

Code:

```
import pandas as pd
import numpy as np
dataset = pd.read_csv("D:/Datasets/bill_authentication.csv")
dataset.head()
X = dataset.iloc[:, 0:4].values
y = dataset.iloc[:, 4].values
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
# Feature Scaling
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(accuracy_score(y_test, y_pred))
= sc.transform(X_test)
from sklearn.ensemble import RandomForestRegressor

regressor = RandomForestRegressor(n_estimators=20, random_state=0)
regressor.fit(X_train, y_train)
y_pred = regressor.predict(X_test)
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(accuracy_score(y_test, y_pred))
```

OUTPUT:

```
[[155    2]
  1  117]]
      precision    recall  f1-score   support

     0       0.99      0.99      0.99        157
     1       0.98      0.99      0.99        118

 avg / total       0.99      0.99      0.99        275

0.989090909091
```

- Program to use IRIS dataset for multi-class classification using AdaBoost algorithm

Code:

```
# 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 classifier object
abc = AdaBoostClassifier(n_estimators=50,
                        learning_rate=1)
# Train Adaboost Classifier
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))
```

OUTPUT:

```
Accuracy: 0.9555555555555556
```

- To build the classification model using the gradient boosting algorithm using Boston Housing Dataset

Code:

```
from sklearn.ensemble import GradientBoostingRegressor
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.datasets import load_boston
from sklearn.metrics import mean_absolute_error

boston = load_boston()
X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = pd.Series(boston.target)

X_train, X_test, y_train, y_test = train_test_split(X, y)


regressor = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=3,
    learning_rate=1.0
)
regressor.fit(X_train, y_train)

errors = [mean_squared_error(y_test, y_pred) for y_pred in regressor.staged_predict(X_test)]
best_n_estimators = np.argmin(errors)

best_regressor = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=best_n_estimators,
    learning_rate=1.0
)
best_regressor.fit(X_train, y_train)

y_pred = best_regressor.predict(X_test)
mean_absolute_error(y_test, y_pred)
```

OUTPUT:



3.6452601648381675

Result:

Thus, the implementation of ensemble model for an application has been successfully executed.

Experiment 14: Implementation of NLP Programs

- To discriminate between ham/spam messages automatically using UCI datasets.

Code:

```
# Import library
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.describe()

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)

# Tokenization

def transform_message(message):
    message_not_punc = [] # Message without punctuation
    i = 0
    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
vectorization = CountVectorizer(analyzer = transform_message )
X = vectorization.fit(data['messages'])
```

```

X_transform = X.transform([data['messages']])

# TF-IDF
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))

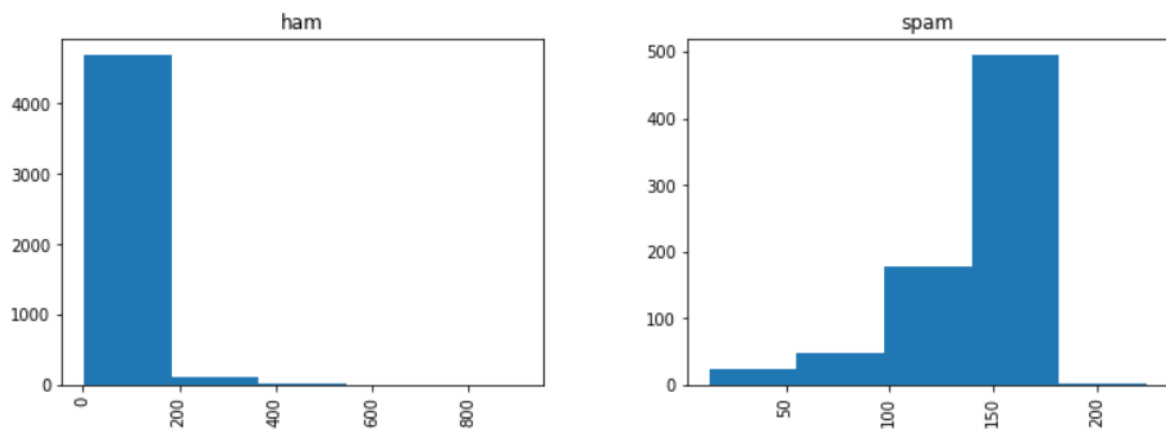
```

OUTPUT:

	label	messages
count	5574	5574
unique	2	5171
top	ham	Sorry, I'll call later
freq	4827	30

	label	messages	length
1085	ham	For me the love should start with attraction.i...	910
1863	ham	The last thing i ever wanted to do was hurt yo...	790
2434	ham	Indians r poor but India is not a poor country...	629
1579	ham	How to Make a girl Happy? It's not at all diff...	611
2849	ham	Sad story of a Man - Last week was my b'day. M...	588
2158	ham	Sad story of a Man - Last week was my b'day. M...	588
2380	ham	Good evening Sir, hope you are having a nice d...	482
3017	ham	& is fast approaching. So, Wish u a v...	461
1513	ham	Hey sweet, I was wondering when you had a mome...	458
2370	ham	A Boy loved a gal. He propsd bt she didnt mind...	446

```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x000001FF386FFD08>,  
      <matplotlib.axes._subplots.AxesSubplot object at 0x000001FF38705E88>],  
      dtype=object)
```



Result:

Thus, the implementation of NLP in any program has been successfully executed.

Experiment 15: Applying Deep Learning method for Automatic Handwriting Recognition

- Program to implement deep learning method for Automatic Handwriting Recognition

Code:

```
from keras.datasets import mnist
import matplotlib.pyplot as plt
import cv2
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Dropout
from keras.optimizers import SGD, Adam
from keras.callbacks import ReduceLROnPlateau, EarlyStopping
from keras.utils import to_categorical
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from keras.utils import np_utils
import matplotlib.pyplot as plt
from tqdm import tqdm_notebook
from sklearn.utils import shuffle

# Read the data...
data = pd.read_csv(r"D:\a-z alphabets\A_Z Handwritten Data.csv").astype('float32')

# Split data the X - Our data , and y - the prdict label
X = data.drop('0',axis = 1)
y = data['0']

# Reshaping the data in csv file so that it can be displayed as an image...

train_x, test_x, train_y, test_y = train_test_split(X, y, test_size = 0.2)
train_x = np.reshape(train_x.values, (train_x.shape[0], 28,28))
test_x = np.reshape(test_x.values, (test_x.shape[0], 28,28))

print("Train data shape: ", train_x.shape)
print("Test data shape: ", test_x.shape)

# Dictionary for getting characters from index values...
word_dict =
{0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'K',11:'L',12:'M',13:'N',14:'O',15:'P',16:'Q',17:'R',18:'S',19:'T',20:'U',21:'V',22:'W',23:'X', 24:'Y',25:'Z'}

# Plotting the number of alphabets in the dataset...

train_yint = np.int0(y)
count = np.zeros(26, dtype='int')
for i in train_yint:
    count[i] +=1
```

```

alphabets = []
for i in word_dict.values():
    alphabets.append(i)

fig, ax = plt.subplots(1,1, figsize=(10,10))
ax.barh(alphabets, count)

plt.xlabel("Number of elements ")
plt.ylabel("Alphabets")
plt.grid()
plt.show()

#Shuffling the data ...
shuff = shuffle(train_x[:100])

fig, ax = plt.subplots(3,3, figsize = (10,10))
axes = ax.flatten()

for i in range(9):
    axes[i].imshow(np.reshape(shuff[i], (28,28)), cmap="Greys")
plt.show()

#Reshaping the training & test dataset so that it can be put in the model...

train_X = train_x.reshape(train_x.shape[0],train_x.shape[1],train_x.shape[2],1)
print("New shape of train data: ", train_X.shape)

test_X = test_x.reshape(test_x.shape[0], test_x.shape[1], test_x.shape[2],1)
print("New shape of train data: ", test_X.shape)

# Converting the labels to categorical values...

train_yOHE = to_categorical(train_y, num_classes = 26, dtype='int')
print("New shape of train labels: ", train_yOHE.shape)

test_yOHE = to_categorical(test_y, num_classes = 26, dtype='int')
print("New shape of test labels: ", test_yOHE.shape)

# CNN model...

model = Sequential()

model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding = 'same'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding = 'valid'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Flatten())

model.add(Dense(64,activation = "relu"))
model.add(Dense(128,activation = "relu"))

model.add(Dense(26,activation = "softmax"))

```

```
model.compile(optimizer = Adam(learning_rate=0.001), loss='categorical_crossentropy',
metrics=['accuracy'])
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=1, min_lr=0.0001)
early_stop = EarlyStopping(monitor='val_loss', min_delta=0, patience=2, verbose=0, mode='auto')
```

```
history = model.fit(train_X, train_yOHE, epochs=1, callbacks=[reduce_lr, early_stop], validation_data
= (test_X, test_yOHE))
```

```
model.summary()
model.save(r'model_hand.h5')
```

```
# Displaying the accuracies & losses for train & validation set...
```

```
print("The validation accuracy is :", history.history['val_accuracy'])
print("The training accuracy is :", history.history['accuracy'])
print("The validation loss is :", history.history['val_loss'])
print("The training loss is :", history.history['loss'])
```

```
#Making model predictions...
```

```
pred = model.predict(test_X[:9])
print(test_X.shape)
```

```
# Displaying some of the test images & their predicted labels...
```

```
fig, axes = plt.subplots(3,3, figsize=(8,9))
axes = axes.flatten()

for i,ax in enumerate(axes):
    img = np.reshape(test_X[i], (28,28))
    ax.imshow(img, cmap="Greys")
    pred = word_dict[np.argmax(test_yOHE[i])]
    ax.set_title("Prediction: "+pred)
    ax.grid()
```

```
# Prediction on external image...
```

```
img = cv2.imread(r'C:\Users\abhij\Downloads\img_b.jpg')
img_copy = img.copy()

img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (400,440))

img_copy = cv2.GaussianBlur(img_copy, (7,7), 0)
img_gray = cv2.cvtColor(img_copy, cv2.COLOR_BGR2GRAY)
_, img_thresh = cv2.threshold(img_gray, 100, 255, cv2.THRESH_BINARY_INV)

img_final = cv2.resize(img_thresh, (28,28))
img_final = np.reshape(img_final, (1,28,28,1))
```

```

img_pred = word_dict[np.argmax(model.predict(img_final))]

cv2.putText(img, "Dataflair __", (20,25), cv2.FONT_HERSHEY_TRIPLEX, 0.7, color = (0,0,230))
cv2.putText(img, "Prediction: " + img_pred, (20,410), cv2.FONT_HERSHEY_DUPLEX, 1.3, color =
(255,0,30))
cv2.imshow('Dataflair handwritten character recognition __', img)

while (1):
    k = cv2.waitKey(1) & 0xFF
    if k == 27:
        break
cv2.destroyAllWindows()

```

OUTPUT:

```

localhost:8888/notebooks/Handwritten%20character%20recognition/Handwritten_text_recognition.ipynb#
jupyter Handwritten_text_recognition Last Checkpoint: 20 hours ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
Out[234]:
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 ... 0.639 0.640 0.641 0.642 0.643 0.644 0.645 0.646 0.647 0.648
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 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 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
3 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
6 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
7 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
8 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
9 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
10 rows x 785 columns

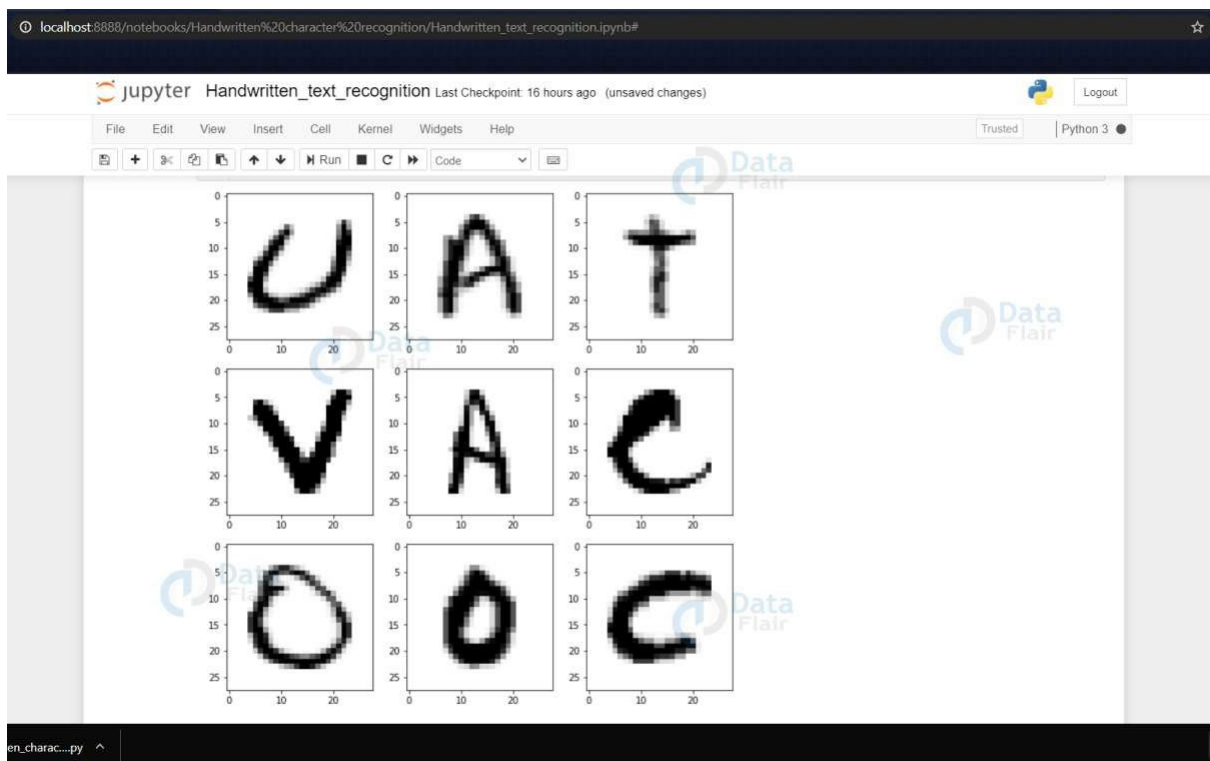
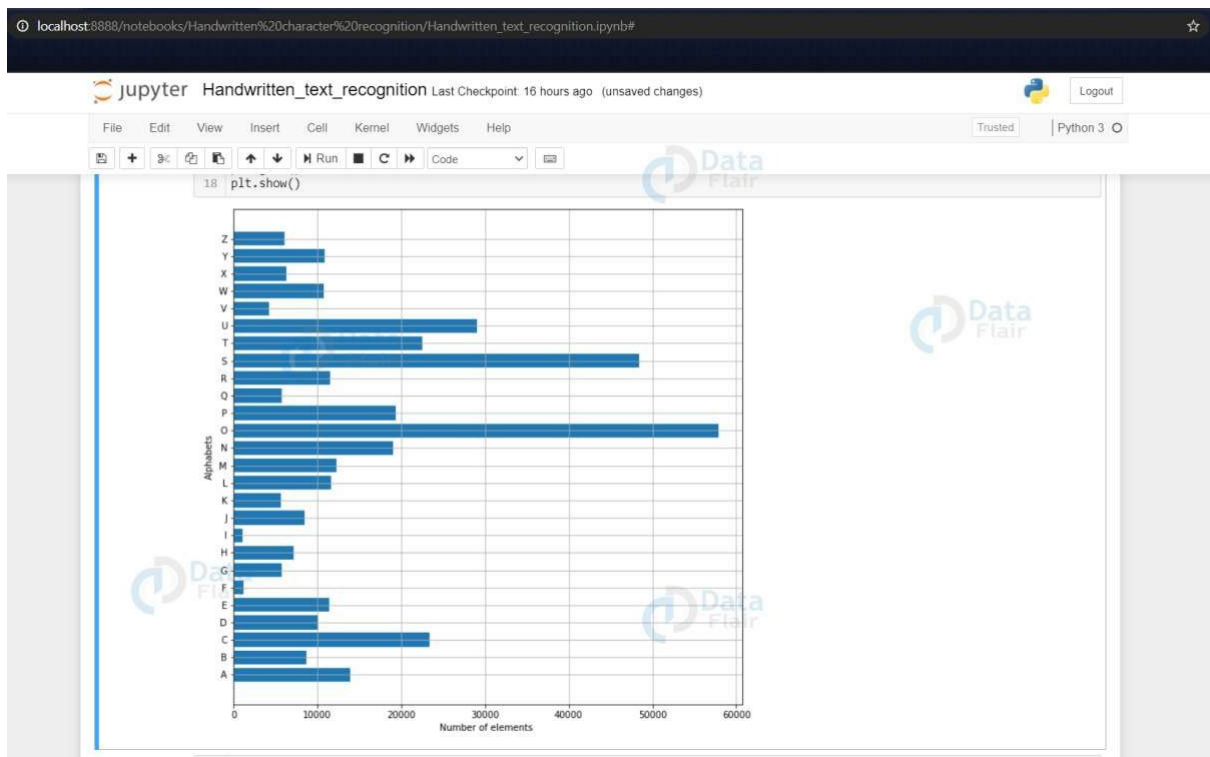
In [ ]: 1
In [189]: 1 # Reshaping the data in csv file so that it can be displayed as an image...
2
3 train_x, test_x, train_y, test_y = train_test_split(X, y, test_size = 0.2)
4 train_x = np.reshape(train_x.values, (train_x.shape[0], 28,28))
5 test_x = np.reshape(test_x.values, (test_x.shape[0], 28,28))

In [190]: 1 print("Train data shape: ", train_x.shape)
2 print("Test data shape: ", test_x.shape)

Train data shape: (297960, 28, 28)
Test data shape: (74490, 28, 28)

In [99]: 1 word_dict = {0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'K',11:'L',12:'M',13:'N',14:'O',15:'P',16:'Q',17

```



```
localhost:8888/notebooks/Handwritten%20character%20recognition/Handwritten_text_recognition.ipynb#
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [51]: 1 model.summary()
2 model.save(r'model_hand.h5')

Model: "sequential_1"

Layer (type) Output Shape Param #
-----
conv2d_1 (Conv2D) (None, 26, 26, 32) 320
max_pooling2d_1 (MaxPooling2D) (None, 13, 13, 32) 0
conv2d_2 (Conv2D) (None, 13, 13, 64) 18496
max_pooling2d_2 (MaxPooling2D) (None, 6, 6, 64) 0
conv2d_3 (Conv2D) (None, 4, 4, 128) 73856
max_pooling2d_3 (MaxPooling2D) (None, 2, 2, 128) 0
flatten_1 (Flatten) (None, 512) 0
dense_1 (Dense) (None, 64) 32832
dense_2 (Dense) (None, 128) 8320
dense_3 (Dense) (None, 26) 3354
-----
Total params: 137,178
Trainable params: 137,178
Non-trainable params: 0

In [51]: 1 history.history
```

```
localhost:8888/notebooks/Handwritten%20character%20recognition/Handwritten_text_recognition.ipynb#
jupyter Handwritten_text_recognition Last Checkpoint: 16 hours ago (unsaved changes) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [52]: 1 print("The validation accuracy is :", history.history['val_accuracy'])
2 print("The training accuracy is :", history.history['accuracy'])
3 print("The validation loss is :", history.history['val_loss'])
4 print("The training loss is :", history.history['loss'])

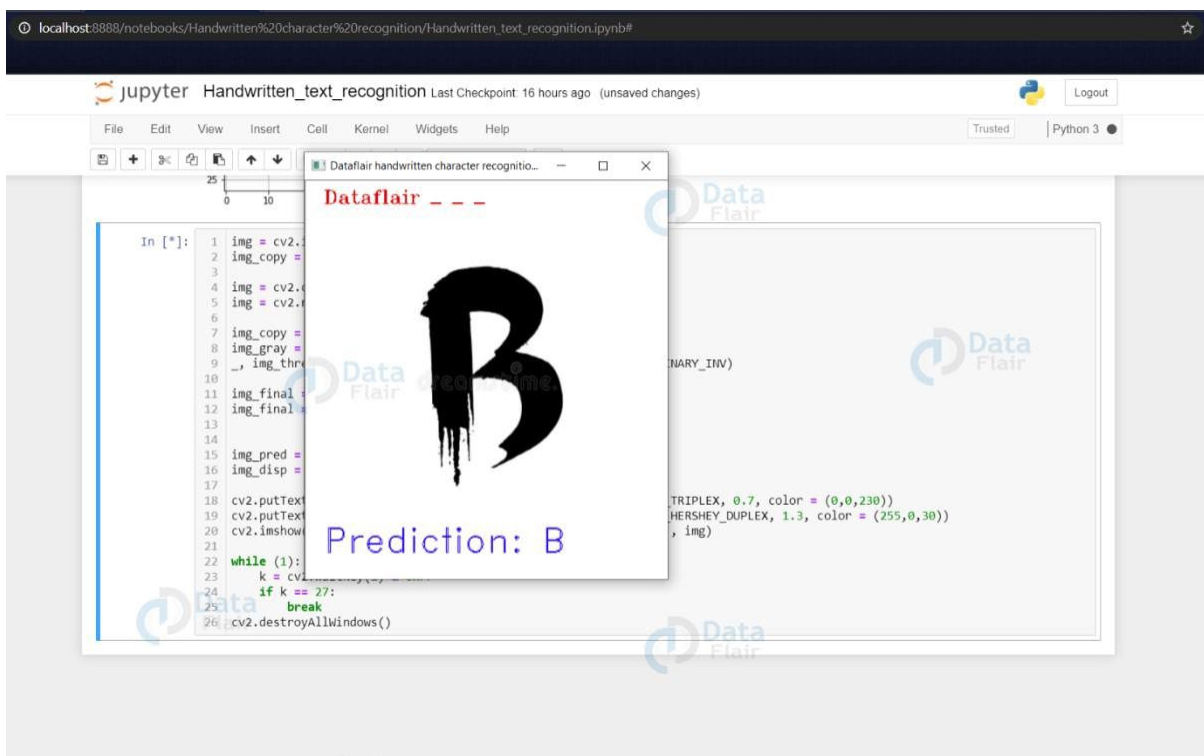
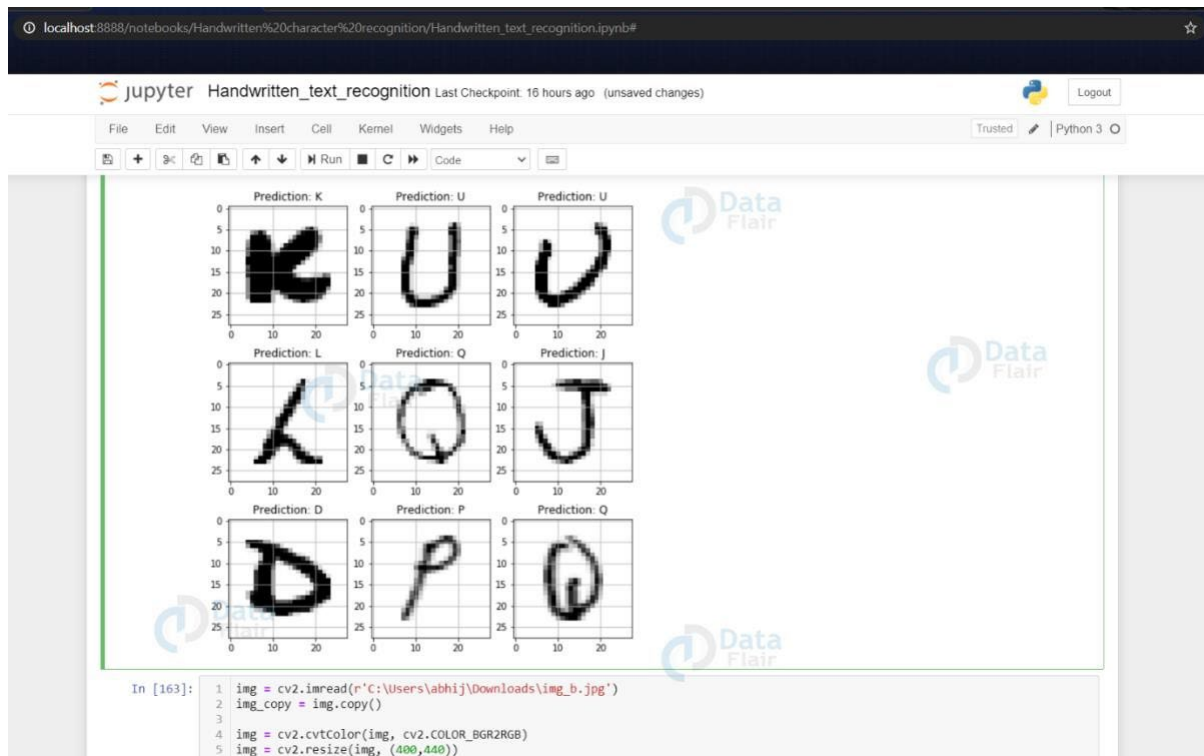
The validation accuracy is : [0.975057065486908]
The training accuracy is : [0.94837564]
The validation loss is : [0.0921439248674415]
The training loss is : [0.1830781325123433]

In [99]: 1 word_dict = {'A':0, 'B':1, 'C':2, 'D':3, 'E':4, 'F':5, 'G':6, 'H':7, 'I':8, 'J':9, 'K':10, 'L':11, 'M':12, 'N':13, 'O':14, 'P':15, 'Q':16, 'R':17}

In [79]: 1 #Making model predictions...
2
3 pred = model.predict(test_X[:9])
4 print(test_X.shape)

(74490, 28, 28, 1)

In [162]: 1 fig, axes = plt.subplots(3,3, figsize=(8,9))
2 axes = axes.flatten()
3
4 for i, ax in enumerate(axes):
5     img = np.reshape(test_X[i], (28,28))
6     ax.imshow(img, cmap="Greys")
7     pred = word_dict[np.argmax(test_yOHE[i])]
8     ax.set_title("Prediction: "+pred)
9     ax.grid()
10
11 y = []
12 for i in pred:
13     y.append(np.argmax(i))
14 print("Predicted values: ", y)
15
16 y_labels = []
```



Result:

Thus, the implementation of deep learning for successful implementation in hand writing detection has been done.