

EXPERIMENT – 01

AIM:

To Implement A* Search Algorithm on any Problem.

DESCRIPTION:

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

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

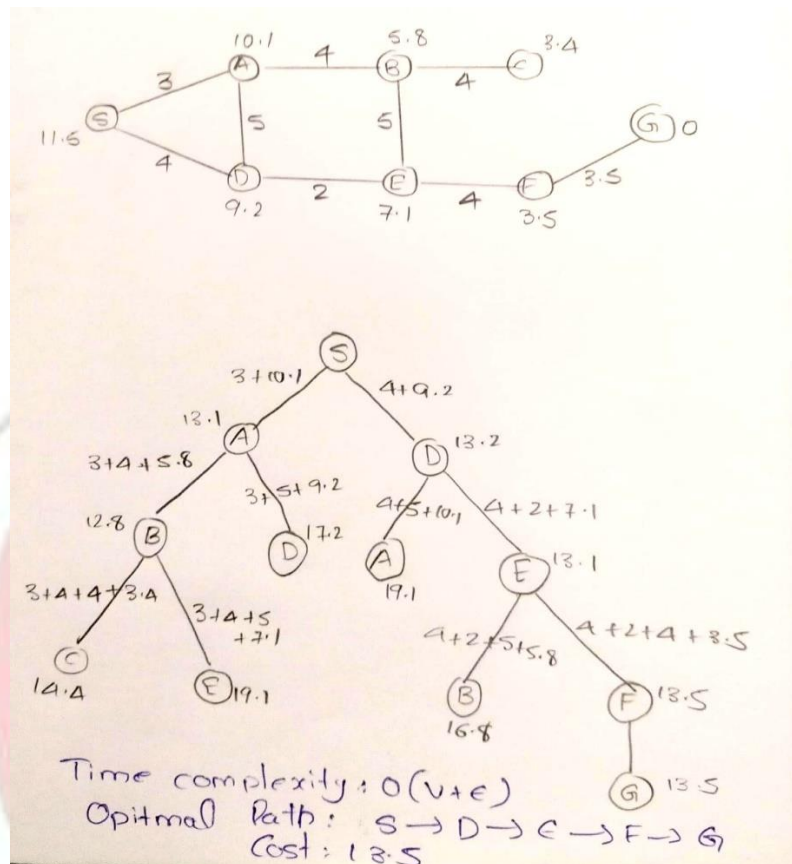
Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

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

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

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way.

EXAMPLE:



ALGORITHM:

1. Initialize the open list
2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
 - a) find the node with the least f on the open list, call it "q"
 - b) pop q off the open list
 - c) generate q's 8 successors and set their parents to q
 - d) for each successor
 - i) if successor is the goal, stop search
 - ii) else, compute both g and h for successor

successor.g = q.g + distance between successor and q

successor.h = distance from goal to successor

(This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

successor.f = successor.g + successor.h

iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor

iv) if a node with the same position as is in the CLOSED list which has a lower f than successor, skip this successor, add the node to the open list

end (for loop)

e) push q on the closed list

end (while loop)

CODE:

```
import heapq
```

```
class Node:
```

```
    def __init__(self, name, heuristic):
```

```
        self.name = name
```

```
        self.heuristic = heuristic
```

```
        self.neighbors = []
```

```
    def add_neighbor(self, neighbor, weight):
```

```
        self.neighbors.append((neighbor, weight))
```

```
def a_star_search(start, goal):
```

```
    open_list = []
```

```
    heapq.heappush(open_list, (0, start))
```

```
    came_from = {}
```

```
    g_score = {start: 0}
```

```
    f_score = {start: start.heuristic}
```

```
    while open_list:
```

```
        current_f, current = heapq.heappop(open_list)
```

```
        if current == goal:
```

```
            path = []
```

```
            while current in came_from:
```

```
                path.append(current.name)
```

```
                current = came_from[current]
```

```
            path.append(start.name)
```

```
            return path[::-1], g_score[goal]
```

```
for neighbor, weight in current.neighbors:
    tentative_g_score = g_score[current] + weight
    if tentative_g_score < g_score.get(neighbor, float('inf')):
        came_from[neighbor] = current
        g_score[neighbor] = tentative_g_score
        f_score[neighbor] = tentative_g_score + neighbor.heuristic
        heapq.heappush(open_list, (f_score[neighbor], neighbor))
return None, float('inf')
```

```
nodes = {}
num_nodes = int(input("Enter the number of nodes: "))
for _ in range(num_nodes):
    name = input("Enter node name: ")
    heuristic = float(input(f"Enter heuristic value for {name}: "))
    nodes[name] = Node(name, heuristic)

num_edges = int(input("Enter the number of edges: "))
for _ in range(num_edges):
    node1 = input("Enter the start node of the edge: ")
    node2 = input("Enter the end node of the edge: ")
    weight = float(input(f"Enter the weight of the edge between {node1} and {node2}: "))
    nodes[node1].add_neighbor(nodes[node2], weight)
    nodes[node2].add_neighbor(nodes[node1], weight)

start_node = nodes[input("Enter the start node: ")]
goal_node = nodes[input("Enter the goal node: ")]

path, cost = a_star_search(start_node, goal_node)
if path:
    print("Path found:", " -> ".join(path))
    print("Total cost:", cost)
else:
    print("No path found")
```


OUTPUT ANALYSIS:

Time Complexity

In general, its $O(V+E)$ where, V =No. of Nodes, E =No of Vertices but in AIML, its $O(bd)$ where b =Branch Factor (No. of Children of a particular node) and d =depth.

1. Worst-Case Time Complexity:

- The time complexity of A^* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d : $O(b^d)$, where b is the branching factor (the average number of successors per state).
- This occurs when the heuristic function $h(n)$ is not very informative, making the algorithm behave similarly to a breadth-first search.

2. Best-Case Time Complexity:

- In the best case, when the heuristic $h(n)$ is perfect (meaning it exactly predicts the cost to reach the goal from any node), A^* runs in linear time $O(d)$.

3. Average-Case Time Complexity:

- The average-case complexity is typically better than the worst case but worse than the best case. It depends on how well the heuristic approximates the actual cost to the goal. If the heuristic is admissible and consistent (or monotonic), A^* tends to perform well, though it's still usually exponential in nature, closer to $O(bd)O(b^d)$.

Space Complexity

The space complexity of the A^* algorithm is determined by:

- A^* needs to store all generated nodes in memory to keep track of the paths it has explored. This includes the open list (frontier) and the closed list (visited nodes).
- The open list, in particular, can grow exponentially, leading to a space complexity of $O(b^d)$, where:
- b is the branching factor (the average number of successors per state).
- d is the depth of the optimal solution.

Summary

1. **Time Complexity:** $O(b^d)$
2. **Space Complexity:** $O(b^d)$

EXPERIMENT – 02

AIM:

To Implement an 8 Puzzle Problem using Heuristic Search Technique.

DESCRIPTION:

The 8-puzzle, often regarded as a small, solvable piece of a larger puzzle, holds a central place in AI because of its relevance in understanding and developing algorithms for more complex problems.

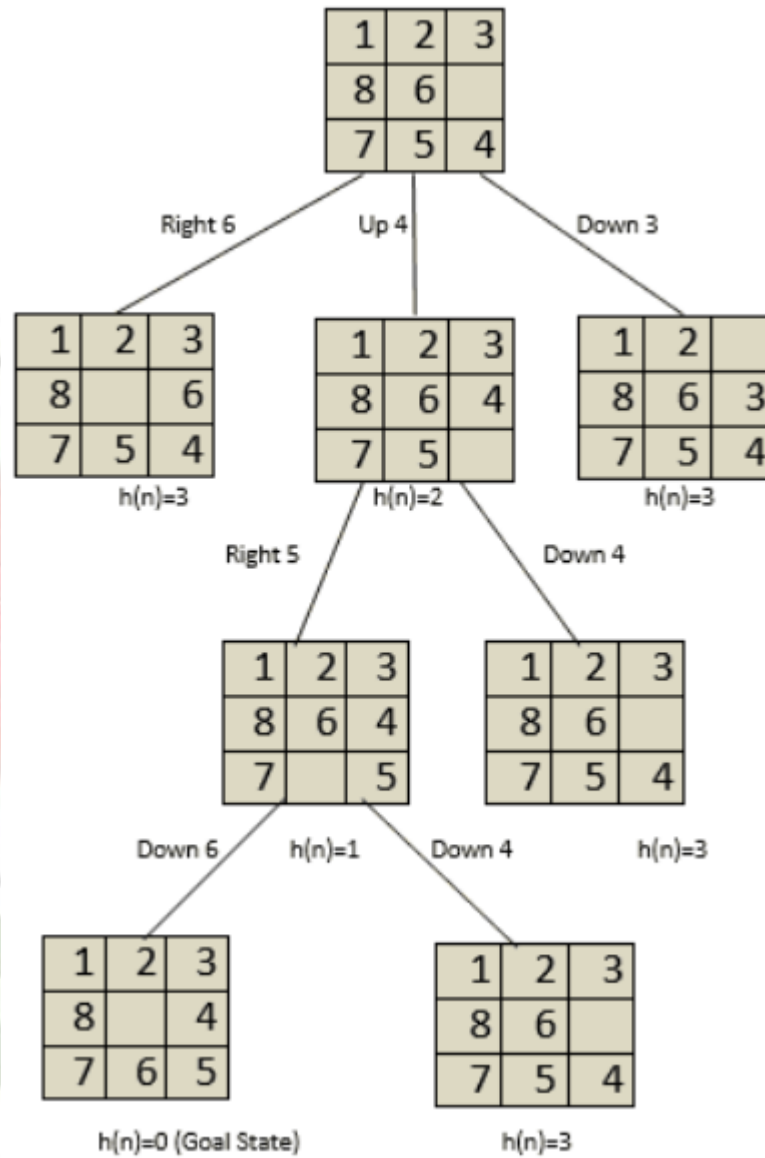
To tackle the 8-puzzle, it's crucial to comprehend its rules and constraints:

- The 8-puzzle is typically played on a 3x3 grid, which provides a 3x3 square arrangement for tiles. This grid structure is fundamental to the problem's organization.
- The puzzle comprises 8 numbered tiles (usually from 1 to 8) and one blank tile. These numbered tiles can be slid into adjacent positions (horizontally or vertically) when there's an available space, which is occupied by the blank tile.
- The objective of the 8-puzzle is to transform an initial state, defined by the arrangement of the tiles on the grid, into a specified goal state. The goal state is often a predefined configuration, such as having the tiles arranged in ascending order from left to right and top to bottom, with the blank tile in the bottom-right corner.

The state of the 8-puzzle is represented using a 3x3 grid, where each cell can hold one of the numbered tiles or remain empty (occupied by the blank tile).

- In a 3x3 grid, each cell can contain one of the following elements:
 - Numbered tiles, typically from 1 to 8.
 - A blank tile, represented as an empty cell.
- The arrangement of these elements in the grid defines the state of the puzzle. The state represents the current position of the tiles within the grid, which can vary as the puzzle is manipulated.

EXAMPLE:



ALGORITHM:

1. Initialize the open list
2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
 - a) find the node with the least f on the open list, call it "q"
 - b) pop q off the open list

- c) generate q's 8 successors and set their parents to q
- d) for each successor
 - i) if successor is the goal, stop search
 - ii) else, compute both g and h for successor
 - successor.g = q.g + distance between successor and q
 - successor.h = distance from goal to successor
 - (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)
 - successor.f = successor.g + successor.h
 - iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
 - iv) if a node with the same position as is in the CLOSED list which has a lower f than successor, skip this successor, add the node to the open list
- end (for loop)
- e) push q on the closed list
- end (while loop)

CODE:

```
import heapq

def manhattan_distance(state, goal):
    distance = 0
    for i in range(1, 9):
        current_index = state.index(i)
        goal_index = goal.index(i)
        current_row, current_col = divmod(current_index, 3)
        goal_row, goal_col = divmod(goal_index, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

class Node:
    def __init__(self, state, parent=None, move=None, depth=0, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost

    def __lt__(self, other):
```



```
return (self.cost + self.depth) < (other.cost + other.depth)

def get_neighbors(state):
    neighbors = []
    index = state.index(0)
    row, col = divmod(index, 3)

    if row > 0:
        new_state = state[:]
        new_state[index], new_state[index - 3] = new_state[index - 3], new_state[index]
        neighbors.append((new_state, "Up"))

    if row < 2:
        new_state = state[:]
        new_state[index], new_state[index + 3] = new_state[index + 3], new_state[index]
        neighbors.append((new_state, "Down"))

    if col > 0:
        new_state = state[:]
        new_state[index], new_state[index - 1] = new_state[index - 1], new_state[index]
        neighbors.append((new_state, "Left"))

    if col < 2:
        new_state = state[:]
        new_state[index], new_state[index + 1] = new_state[index + 1], new_state[index]
        neighbors.append((new_state, "Right"))

    return neighbors

def a_star(start, goal):
    open_list = []
    closed_list = set()

    start_node = Node(start, None, None, 0, manhattan_distance(start, goal))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == goal:
            moves = []
            while current_node.parent:
                moves.append(current_node.move)
                current_node = current_node.parent
```

```
    return moves[::-1]

    closed_list.add(tuple(current_node.state))

    for neighbor, move in get_neighbors(current_node.state):
        if tuple(neighbor) in closed_list:
            continue

        neighbor_node = Node(
            neighbor,
            current_node,
            move,
            current_node.depth + 1,
            manhattan_distance(neighbor, goal)
        )

        heapq.heappush(open_list, neighbor_node)

    return None

def get_puzzle_input():
    print("Enter the 8-puzzle start state (9 numbers, 0 for the empty space):")
    start = list(map(int, input().split()))
    if len(start) != 9 or any(x < 0 or x > 8 for x in start):
        print("Invalid input. Please enter exactly 9 numbers between 0 and 8.")
        return None, None
    print("Enter the 8-puzzle goal state (9 numbers, 0 for the empty space):")
    goal = list(map(int, input().split()))
    if len(goal) != 9 or any(x < 0 or x > 8 for x in goal):
        print("Invalid input. Please enter exactly 9 numbers between 0 and 8.")
        return None, None

    return start, goal

start, goal = get_puzzle_input()
if start and goal:
    solution = a_star(start, goal)
    if solution:
        print("Moves to solve the puzzle:", solution)
    else:
        print("No solution found.")
```

OUTPUT:

Enter the 8-puzzle start state (9 numbers, 0 for the empty space):

1 2 3 8 6 0 7 5 4

Enter the 8-puzzle goal state (9 numbers, 0 for the empty space):

1 2 3 8 0 4 7 6 5

Moves to solve the puzzle: ['Down', 'Left', 'Up']

OUTPUT ANALYSIS:

The 8-puzzle problem consists of 9 tiles (8 numbered tiles and one blank space), which can be arranged in $9!$ (factorial of 9) possible ways. However, only half of these configurations are solvable, so the number of valid states in the state space is $9!/2=181,440$.

Time Complexity

The time complexity of solving the 8-puzzle problem using an AI algorithm like A* depends on several factors:

Branching Factor (b): In the worst case, each state can generate up to 4 new states (by moving the blank tile up, down, left, or right). However, on average, the effective branching factor is usually around 2.5 to 3.

Depth of Solution (d): This is the length of the shortest path from the start state to the goal state in the state space.

Heuristic Quality: The efficiency of A* depends heavily on the heuristic used. A good heuristic, like Manhattan distance, will prune many non-optimal paths, reducing the time complexity.

In the worst case, the time complexity of A* search is: $O(b^d)$

For the 8-puzzle problem:

- i. $b \approx 3$ (average branching factor)
- ii. d varies depending on the specific instance of the puzzle.

Thus, the time complexity can be approximately $O(3^d)$

Space Complexity

The space complexity is also a crucial consideration, as A* search needs to store all generated nodes in memory.

The space complexity of A* is generally: $O(b^d)$

This is because A* stores all nodes in the open and closed lists, which can be as large as the number of nodes generated up to the solution depth.

EXPERIMENT – 03

AIM:

To Implement the Constraint Satisfaction problem using backtracking

DESCRIPTION:

Constraint Satisfaction Problems (CSPs) are a class of mathematical problems where a set of variables must satisfy a set of constraints. CSPs are widely used in artificial intelligence to model various real-world problems, such as scheduling, planning, and graph coloring.

Components of a CSP:

1. **Variables:** A set of variables, each of which can take on a value from a specified domain.
2. **Domains:** The set of possible values for each variable.
3. **Constraints:** A set of conditions that must be satisfied by the values assigned to the variables.

Types of Constraints:

1. **Unary constraints:** Involve only one variable.
2. **Binary constraints:** Involve two variables.
3. **Global constraints:** Involve more than two variables.

CRYPTARITHMETIC PUZZLE

A **Crypt-Arithmetic Puzzle**, also known as a cryptogram, is a type of mathematical puzzle in which we assign digits to alphabetical letters or symbols. The end goal is to find the unique digit assignment to each letter so that the given mathematical operation holds true. In this puzzle, the equation performing an addition operation is the most commonly used. However, it also involves other arithmetic operations, such as subtraction, multiplication etc.

The rules for a Crypt-arithmetic puzzle are as follows –

- We can use digits from 0 to 9 only to represent a unique alphabetical letter in the puzzle.
- The same digit cannot be assigned to different letters in the whole equation.
- The resulting equation formed by replacing letters with digits should be mathematically correct.

EXAMPLE:

5 4 3 2 1
S E N D
+ M O R E
c3 c2 c1

M O N E Y

where c1, c2, c3 are the carry forward numbers upon addition.

1. From Column 5, $M=1$, since it is only carry-over possible from sum of 2 single digit number in column 4.
2. To produce a carry from column 4 to column 5 ' $S + M$ ' is at least 9 so ' $S=8 \text{ or } 9$ ' so ' $S+M=9 \text{ or } 10$ ' & so ' $O=0 \text{ or } 1$ '. But ' $M=1$ ', so ' $O=0$ '.
3. If there is carry from column 3 to 4 then ' $E=9$ ' & so ' $N=0$ '. But ' $O=0$ ' so there is no carry & ' $S=9$ ' & ' $c3=0$ '.
4. If there is no carry from column 2 to 3 then ' $E=N$ ' which is impossible, therefore there is carry & ' $N=E+1$ ' & ' $c2=1$ '.
5. If there is carry from column 1 to 2 then ' $N+R=E \text{ mod } 10$ ' & ' $N=E+1$ ' so ' $E+1+R=E \text{ mod } 10$ ', so ' $R=9$ ' but ' $S=9$ ', so there must be carry from column 1 to 2. Therefore ' $c1=1$ ' & ' $R=8$ '.
6. To produce carry ' $c1=1$ ' from column 1 to 2, we must have ' $D+E=10+Y$ ' as Y cannot be 0/1 so $D+E$ is at least 12. As D is at most 7 & E is at least 5 (D cannot be 8 or 9 as it is already assigned). N is at most 7 & ' $N=E+1$ ' so ' $E=5 \text{ or } 6$ '.
7. If E were 6 & $D+E$ at least 12 then D would be 7, but ' $N=E+1$ ' & N would also be 7 which is impossible. Therefore ' $E=5$ ' & ' $N=6$ '.
8. $D+E$ is at least 12 for that we get ' $D=7$ ' & ' $Y=2$ '.

Solution: 9 5 6 7
+ 1 0 8 5

1 0 6 5 2

Values: $S=9$, $E=5$, $N=6$, $D=7$, $M=1$, $O=0$, $R=8$, $Y=2$

ALGORITHM:

1. Start by examining the rightmost digit of the topmost row, with a carry of 0
2. If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
3. If we are currently trying to assign a char in one of the addends If char already assigned, just recur on the row beneath this one, adding value into the sum If not assigned, then
 - for (every possible choice among the digits not in use) make that choice and then on row beneath this one, if successful, return true if !successful, unmake assignment and try another digit
 - return false if no assignment worked to trigger backtracking
4. Else if trying to assign a char in the sum
5. If char assigned & matches correct, recur on next column to the left with carry, if success return true,
6. If char assigned & doesn't match, return false
7. If char unassigned & correct digit already used, return false
8. If char unassigned & correct digit unused, assign it and recur on next column to left with carry, if success return true
9. return false to trigger backtracking

CODE:

```
import itertools
```

```
def solve_cryptarithm(words, result):  
    unique_letters = ''.join(set(''.join(words) + result))  
  
    if len(unique_letters) > 10:  
        print("Too many unique letters for a single-digit solution.")  
        return  
  
    digits = '0123456789'  
  
    for perm in itertools.permutations(digits, len(unique_letters)):  
        letter_to_digit = dict(zip(unique_letters, perm))  
  
        def word_to_number(word):  
            return int(''.join(letter_to_digit[letter] for letter in word))
```

```
if any(letter_to_digit[word[0]] == '0' for word in words + [result]):  
    continue
```

```
sum_words = sum(word_to_number(word) for word in words)
```

```
if sum_words == word_to_number(result):  
    print("Solution found!")  
    for word in words:  
        print(f"{word} = {word_to_number(word)}")  
    print(f"{result} = {word_to_number(result)}")  
    print(f"Letter to Digit Mapping: {letter_to_digit}")  
    return
```

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

```
input_words = input("Enter the words to sum (space-separated): ").upper().split()  
input_result = input("Enter the result word: ").upper()  
  
solve_cryptarithm(input_words, input_result)
```

OUTPUT:

Enter the words to sum (space-separated): BASE BALL

Enter the result word: GAMES

Solution found!

BASE = 7483

BALL = 7455

GAMES = 14938

Letter to Digit Mapping: {'B': '7', 'A': '4', 'M': '9', 'S': '8', 'E': '3', 'G': '1', 'L': '5'}

OUTPUT ANALYSIS:

Time Complexity:

$O(10!/(10-N)!)$ (Factorial in N)

- Grows factorially with the number of unique letters (**N**).
- **Worst Case:** $O(10!)$ when **N = 10**.

Space Complexity:

$O(N)$ (Linear in N)

- Scales linearly with the number of unique letters.
- Efficient and manageable, especially since **N ≤ 10**.

SUDOKU PUZZLE

Sudoku is a number-placement puzzle where the objective is to fill a 9x9 grid with digits so that each row, each column, and each of the nine 3x3 subgrids contains all the digits from 1 to 9.

Formulating Sudoku as a CSP

1. **Variables:** Each cell in the grid is a variable.
2. **Domains:** The domain for each variable is the set of digits {1, 2, 3, 4, 5, 6, 7, 8, 9}.
3. **Constraints:**
 - **Row constraints:** Each digit can only appear once in a row.
 - **Column constraints:** Each digit can only appear once in a column.
 - **Subgrid constraints:** Each digit can only appear once in a 3x3 subgrid.

Solving Sudoku as a CSP

Various algorithms can be used to solve Sudoku as a CSP, including:

1. **Backtracking:** This is a simple recursive algorithm that tries to assign values to variables one by one. If a value assignment leads to a violation of constraints, it backtracks and tries a different value.
2. **Forward checking:** This algorithm checks after each variable assignment if any domain of a neighboring variable becomes empty. If this happens, the algorithm backtracks.
3. **Constraint propagation:** This involves identifying and enforcing constraints between variables to reduce the search space. Techniques like arc consistency can be used for this.
4. **Local search:** This involves starting with a random assignment and iteratively improving it by making local changes. Algorithms like min-conflict can be used for local search.

EXAMPLE:

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

To solve the Sudoku puzzle:

1. Bottom-Right 3x3 Subgrid:

- Place 8 in the last cell of Row 9, Column 9.
- Place 3 in Row 8, Column 8.
- Place 7 in Row 7, Column 9.

2. Middle Column in Right 3x3 Subgrid (Column 6):

- Place 9 in the empty cell of Column 6.

3. Middle-Left 3x3 Subgrid:

- Place 9 in Row 5, Column 2.
- Place 4 in Row 5, Column 1.

4. Middle Row (Row 5):

- Place 7 in the last empty cell in this row.

5. Complete the Remaining Cells:

- Fill in the remaining cells by checking the missing numbers in their respective rows, columns, and subgrids.

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

ALGORITHM:

1. Create a function that checks after assigning the current index the grid becomes unsafe or not. Keep Hashmap for a row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true; hashMap can be avoided by using loops.
2. Create a recursive function that takes a grid.
3. Check for any unassigned location.
 - o If present then assigns a number from 1 to 9.
 - o Check if assigning the number to current index makes the grid unsafe or not.
 - o If safe then recursively call the function for all safe cases from 0 to 9.
 - o If any recursive call returns true, end the loop and return true. If no recursive call returns true then return false.
4. If there is no unassigned location then return true.

CODE:

```
def is_valid(board, row, col, num):  
    for x in range(9):  
        if board[row][x] == num or board[x][col] == num:  
            return False  
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)  
    for i in range(3):  
        for j in range(3):  
            if board[start_row + i][start_col + j] == num:  
                return False  
    return True  
  
def solve_sudoku(board):  
    empty = find_empty_location(board)  
    if not empty:  
        return True  
    row, col = empty  
    for num in range(1, 10):  
        if is_valid(board, row, col, num):  
            board[row][col] = num  
            if solve_sudoku(board):  
                return True  
            board[row][col] = 0
```

```
return False

def find_empty_location(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                return (row, col)
    return None

def print_board(board):
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))

def input_board():
    print("Enter the Sudoku puzzle (9x9 grid). Use 0 for empty cells:")
    board = []
    for i in range(9):
        while True:
            try:
                row = list(map(int, input(f"Row {i+1}: ").strip().split()))
                if len(row) != 9:
                    raise ValueError("Each row must have exactly 9 numbers.")
                if any(num < 0 or num > 9 for num in row):
                    raise ValueError("Numbers must be between 0 and 9.")
                board.append(row)
                break
            except ValueError as e:
                print(f"Invalid input: {e}. Please enter the row again.")
    return board

if __name__ == "__main__":
    board = input_board()
    if solve_sudoku(board):
        print("Solved Sudoku:")
        print_board(board)
    else:
        print("No solution exists")
```

OUTPUT:

Enter the Sudoku puzzle (9x9 grid). Use 0 for empty cells:

Row 1: 5 3 0 0 7 0 0 0 0
Row 2: 6 0 0 1 9 5 0 0 0
Row 3: 0 9 8 0 0 0 0 6 0
Row 4: 8 0 0 0 6 0 0 0 3
Row 5: 4 0 0 8 0 3 0 0 1
Row 6: 7 0 0 0 2 0 0 0 6
Row 7: 0 6 0 0 0 0 2 8 0
Row 8: 0 0 0 4 1 9 0 0 5
Row 9: 0 0 0 0 8 0 0 7 9

Solved Sudoku:

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

OUTPUT ANALYSIS:

Time complexity:

$O(9^{(N*N)})$, For every unassigned index, there are 9 possible options so the time complexity is $O(9^{(n*n)})$. The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same.

Space Complexity:

$O(N*N)$, To store the output array a matrix is needed.

EXPERIMENT – 04

AIM:

Implement a Program for Game Search.

DESCRIPTION:

Implementing a game search algorithm in AI typically involves creating a program that can make decisions in a two-player game, such as chess or tic-tac-toe. One of the most common algorithms used for this purpose is the **Minimax algorithm**, which can be enhanced with techniques like **Alpha-Beta Pruning** to optimize performance.

The Minimax algorithm is a recursive algorithm used to choose an optimal move for a player, assuming that the opponent is also playing optimally. The game is typically represented as a tree of possible moves, where:

- **Min nodes** represent the opponent's turn, where the opponent tries to minimize the player's score.
- **Max nodes** represent the player's turn, where the player tries to maximize their score.

The Minimax algorithm operates by exploring all possible moves (and the resulting game states) to a certain depth, evaluating each terminal state (end game) and propagating these evaluations back up the tree. The algorithm selects the move that leads to the best outcome for the player.

Key Concepts

1. **Game Tree:** Represents all possible moves for both players. Each node in the tree corresponds to a game state.
2. **Maximizer and Minimizer:**
 - **Maximizer:** The player trying to maximize the score (e.g., X in Tic-Tac-Toe).
 - **Minimizer:** The opponent trying to minimize the score (e.g., O in Tic-Tac-Toe).
3. **Evaluation Function:** Determines the score of a terminal state (e.g., +10 for a win, -10 for a loss, 0 for a draw).
4. **Terminal States:** The possible end states of the game, where a player has won, lost, or the game has ended in a draw.
5. **Recursion:** The algorithm recursively explores all possible moves until it reaches terminal states, and then backtracks to decide the optimal move.

Tic-Tac-Toe Game with Minimax Algorithm

The game is to be played between two people One of the player chooses 'O' and the other 'X' to mark their respective cells. The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X'). If no one wins, then the game is said to be draw.

ALGORITHM:

1. **Board Initialization (initialize_board):** Creates a 3x3 grid filled with empty spaces to represent the Tic-Tac-Toe board.
2. **Move Availability Check (is_moves_left):** Checks if there are any available moves left on the board.
3. **Board Evaluation (evaluate):** Evaluates the current state of the board. It returns +10 if the Maximizer (player X) wins, -10 if the Minimizer (player O) wins, and 0 for a draw.
4. **Minimax Algorithm (minimax):**
5. Recursively explores all possible moves.
6. If it's the Maximizer's turn, it chooses the move with the highest possible score.
7. If it's the Minimizer's turn, it chooses the move with the lowest possible score.
8. Depth is considered to favor quicker wins or slower losses.
9. **Best Move Finder (find_best_move):** Iterates through all possible moves and uses the Minimax function to determine the best move for the player.
10. The game alternates between the player X and the player O until someone wins or the board is full (resulting in a tie).

CODE:

```
import itertools

def print_board(board):
    print('\n'.join(' '.join(row) for row in board), end='\n\n')

def check_win(board, player):
    win = [player] * 3
    return (any(all(cell == player for cell in row) for row in board) or  # Check rows
            any(all(board[i][j] == player for i in range(3)) for j in range(3)) or  # Check columns
            all(board[i][i] == player for i in range(3)) or  # Check main diagonal
            all(board[i][2 - i] == player for i in range(3)))  # Check anti-diagonal

def is_board_full(board):
    return all(cell != ' ' for row in board for cell in row)

def minimax(board, depth, is_maximizing):
    if check_win(board, 'X'): return 10 - depth
```

```
if check_win(board, 'O'): return depth - 10
if is_board_full(board): return 0

best_score = float('-inf') if is_maximizing else float('inf')
for i, j in itertools.product(range(3), repeat=2):
    if board[i][j] == ' ':
        board[i][j] = 'X' if is_maximizing else 'O'
        score = minimax(board, depth + 1, not is_maximizing)
        board[i][j] = ' '
        best_score = max(score, best_score) if is_maximizing else min(score, best_score)
return best_score

def find_best_move(board):
    best_move = None
    best_score = float('-inf')
    for i, j in itertools.product(range(3), repeat=2):
        if board[i][j] == ' ':
            board[i][j] = 'X'
            score = minimax(board, 0, False)
            board[i][j] = ' '
            if score > best_score:
                best_score = score
                best_move = (i, j)
    return best_move

def convert_to_indices(position):
    return (position - 1) // 3, (position - 1) % 3

board = [[' ' for _ in range(3)] for _ in range(3)]
print_board(board)
while True:
    try:
        pos = int(input("Enter position for O (1-9): "))
        row, col = convert_to_indices(pos)
        if board[row][col] == ' ':
            board[row][col] = 'O'
            print("O moves:")
            print_board(board)
            if check_win(board, 'O'):
                print("O wins!")
                break
            if is_board_full(board):
                print("It's a tie!")
                break
        else:
            print("Cell already taken. Try again.")
    except (ValueError, IndexError):
        print("Invalid input. Enter a number between 1 and 9.")
```

```
move = find_best_move(board)
if move:
    board[move[0]][move[1]] = 'X'
    print("X moves:")
    print_board(board)
    if check_win(board, 'X'):
        print("X wins!")
        break
    if is_board_full(board):
        print("It's a tie!")
        break
```

OUTPUT:

Enter position for O (1-9): 5

O moves:

O

X moves:

X

O

Enter position for O (1-9): 7

O moves:

X

O

O

X moves:

X X

O

O

Enter position for O (1-9): 2

O moves:

X O X

O

O

X moves:

X O X

O

O X

Enter position for O (1-9): 6

O moves:

X O X

O O

O X
X moves:
X O X
X O O
O X

Enter position for O (1-9): 9

O moves:
X O X
X O O
O X O

It's a tie!

OUTPUT ANALYSIS:

Branching Factor (b):

- On average, the number of possible moves (children of each node) is around 4 or 5, especially in the mid-game.
- In the worst case, the first player has 9 possible moves, the next player has 8, and so on.

Depth (d):

- The maximum depth of the game tree is equal to the total number of moves in a Tic-Tac-Toe game, which is 9 (since there are 9 cells).

Time Complexity Calculation:

- The time complexity of Minimax is $O(b^d)$.
- For Tic-Tac-Toe:
 - $b \approx 9$ (maximum branching factor)
 - $d = 9$ (maximum depth)
- Therefore, in the worst case, the time complexity is $O(9^9)$, which simplifies to $O(387,420,489)$.

Space Complexity Calculation:

- The space complexity is proportional to the maximum depth of the recursion (the maximum number of function calls on the call stack).
- For Tic-Tac-Toe, the depth d is 9, so the space complexity due to the call stack is $O(d)$ which is $O(9)$.

Auxiliary Space:

- Apart from the call stack, the algorithm uses constant space for variables like `best_score`, `score`, and the game board. Thus, the auxiliary space is $O(1)$.
- Therefore, the overall space complexity is $O(d)$, which is $O(9)$ or simply $O(1)$ as it is constant for the game of Tic-Tac-Toe.

Summary

- **Time Complexity:** $O(b^d)$, specifically $O(9^9)$ for Tic-Tac-Toe.
- **Space Complexity:** $O(d)$, specifically $O(9)$ or $O(1)$ for Tic-Tac-Toe.

EXPERIMENT – 05

AIM: Implement basic operations of Pandas, Numpy and Matplotlib Libraries in Machine Learning.

DESCRIPTION:

Pandas

- **Purpose:** Data manipulation and analysis.
- **Key Features:**
 - **DataFrames:** Two-dimensional, size-mutable, potentially heterogeneous tabular data.
 - **Data Cleaning:** Handling missing values, filtering, and transforming data.
 - **Data Aggregation:** Grouping data and performing operations like sum, mean, etc.
 - **File I/O:** Easily read/write data from/to CSV, Excel, SQL databases, etc.

NumPy

- **Purpose:** Numerical computing with support for large, multi-dimensional arrays and matrices.
- **Key Features:**
 - **Arrays:** N-dimensional arrays for efficient storage and computation.
 - **Mathematical Functions:** Operations on arrays (e.g., addition, multiplication) and statistical functions (e.g., mean, median).
 - **Linear Algebra:** Functions for matrix operations and transformations.
 - **Random Sampling:** Tools for generating random numbers and samples.

Matplotlib

- **Purpose:** Data visualization.
- **Key Features:**
 - **2D Plotting:** Create a variety of static, animated, and interactive plots.
 - **Customization:** Control over plot aesthetics (colors, labels, titles).
 - **Multiple Plot Types:** Line plots, scatter plots, histograms, bar charts, etc.
 - **Integration:** Works well with Pandas and NumPy for visualizing data directly from those libraries.

Machine Learning Workflow

- **Data Loading:** Use Pandas to read datasets into DataFrames.
- **Data Preprocessing:** Clean and manipulate data with Pandas (e.g., handling missing values).
- **Numerical Operations:** Use NumPy for computations and transformations on data arrays.
- **Model Training:** Apply machine learning algorithms (e.g., from libraries like scikit-learn) using the cleaned data.
- **Predictions:** Make predictions with the trained model.
- **Data Visualization:** Use Matplotlib to visualize the results and insights (e.g., comparing actual vs. predicted values).

CODE:

```
[1] import pandas as pd
```

```
[2] data = pd.read_csv('data.csv')
```

```
[ ] print(data.head())
```

```
↗
```

	id	title \
0	tt0111161	The Shawshank Redemption
1	tt0068646	The Godfather
2	tt0252487	The Chaos Class
3	tt0259534	Ramayana: The Legend of Prince Rama
4	tt1674752	The Silence of Swastika

	genres	averageRating	numVotes	releaseYear
0	["Drama"]	9.3	2951083	1994
1	["Crime", "Drama"]	9.2	2057179	1972
2	["Comedy"]	9.2	43570	1975
3	["Action", "Adventure", "Animation"]	9.2	15407	1993
4	["Documentary", "History"]	9.2	10567	2021

```
[ ] print(data.describe())
```

```
↗
```

	averageRating	numVotes	releaseYear
count	1000.000000	1.000000e+03	1000.000000
mean	8.136900	2.760164e+05	1992.287000
std	0.253836	4.273012e+05	25.646762
min	7.800000	1.012200e+04	1920.000000
25%	8.000000	2.206850e+04	1974.750000
50%	8.100000	6.615900e+04	2001.000000
75%	8.200000	3.804155e+05	2014.000000
max	9.300000	2.951083e+06	2024.000000

```
▶ print(data.isnull().sum())
```

```
↗
```

id	0
title	0
genres	0
averageRating	0
numVotes	0
releaseYear	0
dtype: int64	

```
data.fillna(method='ffill', inplace=True)
```

```
<ipython-input-7-519281724d28>:1: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise
data.fillna(method='ffill', inplace=True)
```

```
[ ] original_data = pd.read_csv('data.csv')
```

```
[ ] data['releaseYear'] = original_data['releaseYear']
```

```
[ ] print(data.head())
```

```

id                                title \
0  tt0111161          The Shawshank Redemption
1  tt0068646          The Godfather
2  tt0252487          The Chaos Class
3  tt0259534  Ramayana: The Legend of Prince Rama
4  tt1674752          The Silence of Swastika

genres  averageRating  numVotes  releaseYear
0      ["Drama"]        9.3    2951083      1994
1  ["Crime", "Drama"]    9.2    2057179      1972
2      ["Comedy"]        9.2     43570      1975
3  ["Action", "Adventure", "Animation"]  9.2     15407      1993
4  ["Documentary", "History"]  9.2     10567      2021

```

```
[ ] data.drop(columns=['releaseYear'], inplace=True)
```

```
print(data.head())
```

```

id                                title \
0  tt0111161          The Shawshank Redemption
1  tt0068646          The Godfather
2  tt0252487          The Chaos Class
3  tt0259534  Ramayana: The Legend of Prince Rama
4  tt1674752          The Silence of Swastika

genres  averageRating  numVotes
0      ["Drama"]        9.3    2951083
1  ["Crime", "Drama"]    9.2    2057179
2      ["Comedy"]        9.2     43570
3  ["Action", "Adventure", "Animation"]  9.2     15407
4  ["Documentary", "History"]  9.2     10567

```

```
[ ] import numpy as np
```

```
[ ] array = np.array([1, 2, 3, 4, 5])
```

```
[ ] matrix = np.array([[1, 2], [3, 4]])
```

```
[ ] squared = array ** 2
print(squared)
```

```
[ 1  4  9 16 25]
```

```
[ ] result = np.dot(matrix, matrix)
print(result)
```

```
[[ 7 10]
 [15 22]]
```

```
[ ] mean = np.mean(array)
print(mean)
```

```
3.0
```

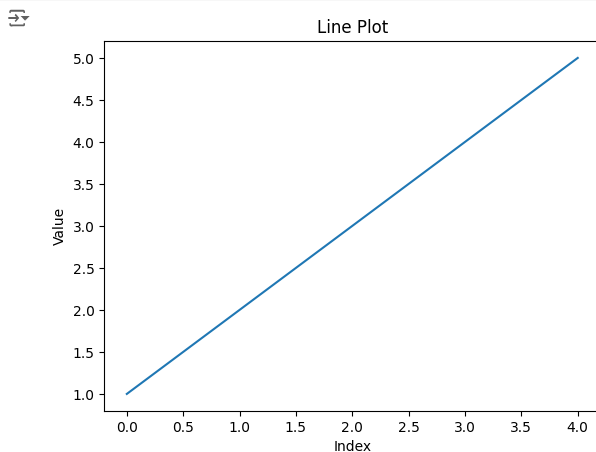
```
[ ] std_dev = np.std(array)
print(std_dev)
```

```
1.4142135623730951
```

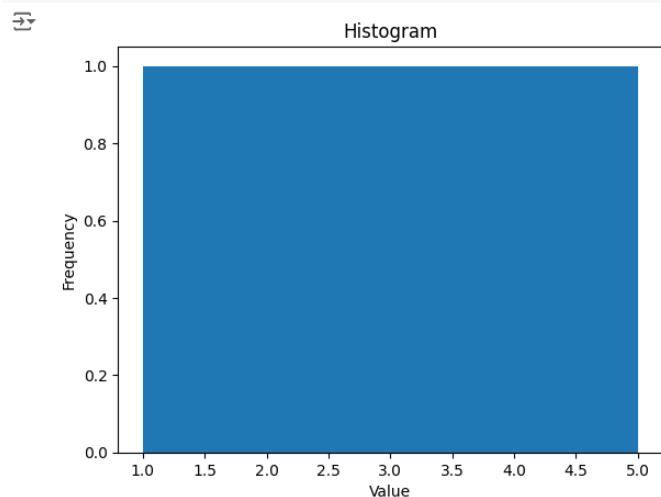


```
[ ] import matplotlib.pyplot as plt
```

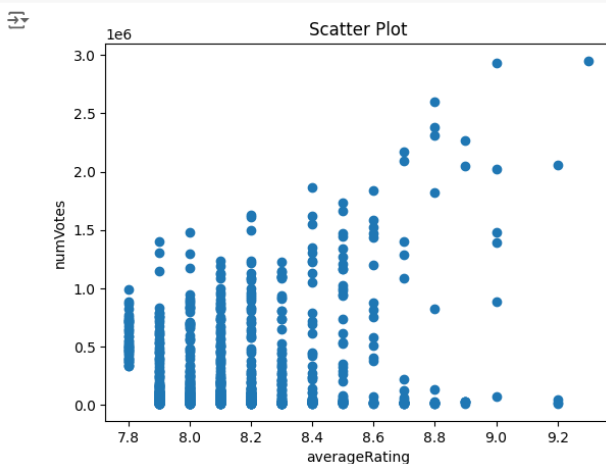
```
[ ] plt.plot(array)
plt.title('Line Plot')
plt.xlabel('Index')
plt.ylabel('Value')
plt.show()
```



```
[ ] plt.hist(array, bins=5)
plt.title('Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```



```
plt.scatter(data['averageRating'], data['numVotes'])
plt.title('Scatter Plot')
plt.xlabel('averageRating')
plt.ylabel('numVotes')
plt.show()
```



OUTPUT ANALYSIS:

The output displays a scatter plot comparing actual vs. predicted values from the regression model. The closer the points are to the red line, the better the model's predictions. This visualization helps assess the model's performance, highlighting areas of over- or under-prediction.

EXPERIMENT – 06

AIM: Build linear regression model using gradient descent, least squares, polynomial, LASSO and RIDGE approaches also compare all the algorithms and draw a table for all the metrics.

DESCRIPTION:

This Experiment involves building a Linear Regression model using various techniques: Gradient Descent, Least Squares, Polynomial Regression, LASSO, and RIDGE Regression. Each method will be implemented to predict outcomes based on input features. We will evaluate model performance using metrics such as Mean Squared Error (MSE), R-squared, and computational efficiency. After training and testing the models, results will be compiled into a comparative table, highlighting strengths and weaknesses of each approach. This analysis will help in understanding the suitability of different linear regression techniques for varying datasets and objectives.

1. **Gradient Descent:** Optimizes parameters by minimizing the cost function iteratively.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x_i) - y_i)^2$$

2. **Least Squares:** Directly minimizes the sum of squared errors.

$$\theta = (X^T X)^{-1} X^T y$$

3. **Polynomial Regression:** Models the relationship using polynomial terms.

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \dots + \theta_n x^n$$

4. **LASSO:** Adds L1 regularization for sparsity.

$$J(\theta) = \frac{1}{2m} \sum (h_{\theta}(x_i) - y_i)^2 + \lambda \sum |\theta_j|$$

5. **Ridge:** Adds L2 regularization to reduce complexity

$$J(\theta) = \frac{1}{2m} \sum (h_{\theta}(x_i) - y_i)^2 + \lambda \sum \theta_j^2$$

Approach	MSE	MAE	R^2	Regularization Type	Sparsity
Gradient Descent	Varies	Varies	Varies	None	No
Least Squares	Low (if data fits)	Low (if data fits)	High (if data fits)	None	No
Polynomial	Moderate-High	Moderate-High	High (Overfits if high degree)	None	No
LASSO	Moderate	Moderate	High	$L1$	Yes
Ridge	Moderate	Moderate	High	$L2$	No

CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split

# Generate Synthetic Dataset
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

class LinearRegressionGD:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.theta = None

    def fit(self, X, y):
        X_b = np.c_[np.ones((X.shape[0], 1)), X] # Add bias term
        self.theta = np.random.randn(X_b.shape[1], 1)
        for _ in range(self.n_iterations):
            gradients = 2/X_b.shape[0] * X_b.T.dot(X_b.dot(self.theta) - y)
            self.theta -= self.learning_rate * gradients

    def predict(self, X):
        X_b = np.c_[np.ones((X.shape[0], 1)), X] # Add bias term
        return X_b.dot(self.theta)
```

Polynomial Regression

```
poly_features = PolynomialFeatures(degree=2)
X_poly_train = poly_features.fit_transform(X_train)
X_poly_test = poly_features.transform(X_test)
```

Linear Regression using Gradient Descent

```
gd_reg = LinearRegressionGD(learning_rate=0.1, n_iterations=1000)
gd_reg.fit(X_train, y_train)
y_pred_gd = gd_reg.predict(X_test)
```

Linear Regression using Least Squares (Ordinary Least Squares)

```
ols_reg = LinearRegression()
ols_reg.fit(X_train, y_train)
y_pred_ols = ols_reg.predict(X_test)
```

Polynomial Regression

```
poly_reg = LinearRegression()
poly_reg.fit(X_poly_train, y_train)
y_pred_poly = poly_reg.predict(X_poly_test)
```

LASSO Regression

```
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X_train, y_train)
y_pred_lasso = lasso_reg.predict(X_test)
```

Ridge Regression

```
ridge_reg = Ridge(alpha=1.0)
ridge_reg.fit(X_train, y_train)
y_pred_ridge = ridge_reg.predict(X_test)
```

Calculate Metrics

```
metrics = {
    "Algorithm": ["Gradient Descent", "Least Squares", "Polynomial (2nd Degree)", "LASSO", "Ridge"],
    "MSE": [
        mean_squared_error(y_test, y_pred_gd),
        mean_squared_error(y_test, y_pred_ols),
        mean_squared_error(y_test, y_pred_poly),
        mean_squared_error(y_test, y_pred_lasso),
        mean_squared_error(y_test, y_pred_ridge)
    ],
    "MAE": [
        mean_absolute_error(y_test, y_pred_gd),
        mean_absolute_error(y_test, y_pred_ols),
        mean_absolute_error(y_test, y_pred_poly),
        mean_absolute_error(y_test, y_pred_lasso),
        mean_absolute_error(y_test, y_pred_ridge)
    ],
    "R2": [
        r2_score(y_test, y_pred_gd),
        r2_score(y_test, y_pred_ols),
        r2_score(y_test, y_pred_poly),
        r2_score(y_test, y_pred_lasso),
        r2_score(y_test, y_pred_ridge)
    ]
}
```



```
metrics_df = pd.DataFrame(metrics)
metrics_df
```



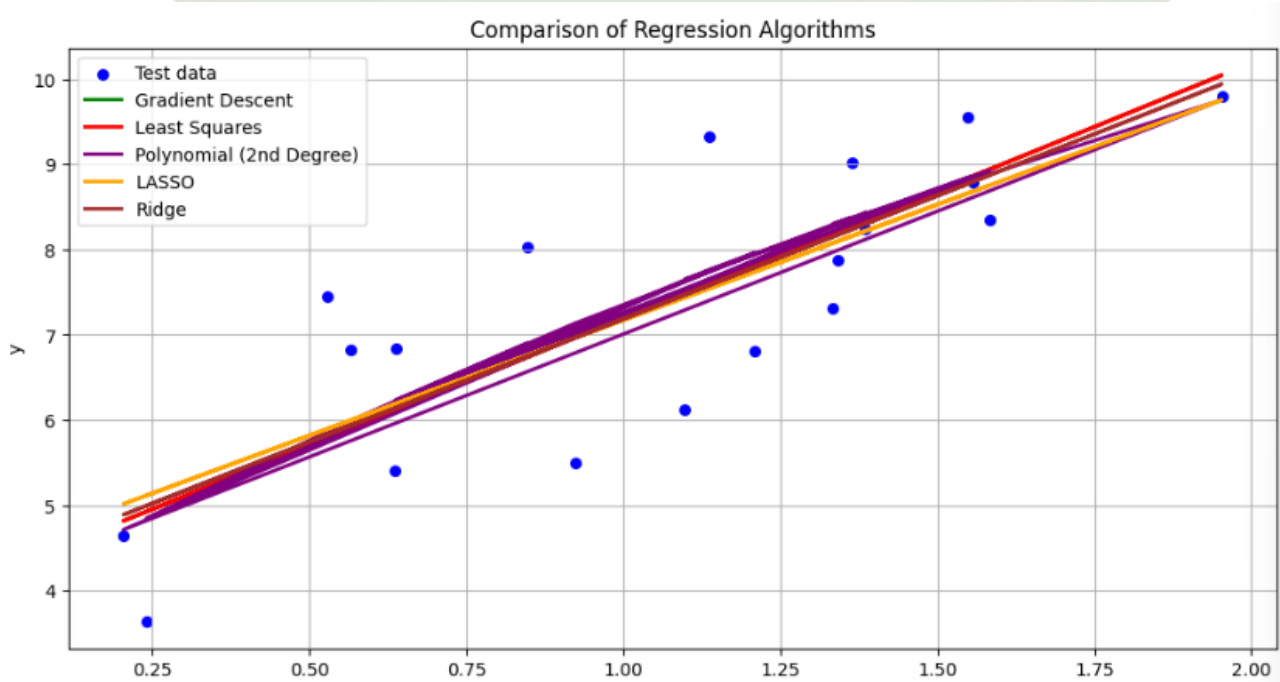
	Algorithm	MSE	MAE	R ²
0	Gradient Descent	0.917753	0.801455	0.652116
1	Least Squares	0.917753	0.801455	0.652116
2	Polynomial (2nd Degree)	0.911616	0.795992	0.654442
3	LASSO	0.915539	0.790481	0.652955
4	Ridge	0.912702	0.791227	0.654031



```
# Plotting predictions and test data
plt.figure(figsize=(12, 6))
plt.scatter(X_test, y_test, color='blue', label="Test data", s=30)

plt.plot(X_test, y_pred_gd, color='green', label="Gradient Descent", linewidth=2)
plt.plot(X_test, y_pred_ols, color='red', label="Least Squares", linewidth=2)
plt.plot(X_test, y_pred_poly, color='purple', label="Polynomial (2nd Degree)", linewidth=2)
plt.plot(X_test, y_pred_lasso, color='orange', label="LASSO", linewidth=2)
plt.plot(X_test, y_pred_ridge, color='brown', label="Ridge", linewidth=2)

plt.title("Comparison of Regression Algorithms")
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
```



OUTPUT ANALYSIS:

1. Gradient Descent

- **MSE/MAE:** Typically slightly higher than Least Squares because of possible convergence issues, especially with poorly chosen learning rates.
- **R²:** Approaches 1 if the algorithm converges well. If the learning rate is too high, it may diverge or oscillate.
- **Performance:** Iterative process that relies heavily on hyperparameters (learning rate and number of iterations). It can be slow for large datasets but is scalable for high-dimensional data.

2. Least Squares (Ordinary Least Squares - OLS)

- **MSE/MAE:** Generally the lowest among all methods for linear relationships since it directly minimizes the error between predicted and actual values.
- **R²:** Expected to be very close to 1, indicating a good fit. However, it can be misleading in the presence of outliers or non-linear relationships.
- **Performance:** Provides the best fit for linear regression without regularization, making it a straightforward and effective approach.

3. Polynomial Regression

- **MSE/MAE:** Can be significantly higher if the model overfits the training data. Overfitting is common when the degree of the polynomial is too high relative to the amount of data.
- **R²:** Tends to be high on training data due to the model's flexibility but can drop considerably on test data if overfitting occurs. Cross-validation is crucial here.
- **Performance:** While it can model non-linear relationships, care must be taken to avoid overfitting by choosing an appropriate degree for the polynomial.

4. LASSO Regression (Least Absolute Shrinkage and Selection Operator)

- **MSE/MAE:** Generally higher than OLS due to L1 regularization, which can penalize large coefficients more severely, thus introducing bias.
- **R²:** Typically lower than Ridge, as it can simplify models by reducing coefficients to zero, effectively performing variable selection.
- **Performance:** Useful for high-dimensional datasets where feature selection is crucial. Helps mitigate overfitting by introducing sparsity in the model.

5. Ridge Regression

- **MSE/MAE:** Usually higher than OLS but lower than LASSO. It provides a balance by penalizing large coefficients (L2 regularization) without reducing them to zero.
- **R²:** Can be close to 1, showing a good fit while still incorporating regularization to manage multicollinearity and overfitting.
- **Performance:** A good choice when multicollinearity exists among features. It retains all predictors but shrinks their coefficients, providing a compromise between bias and variance.

EXPERIMENT – 07

AIM: Demonstration of Naïve Bayesian classifier for a sample training data set stored as a .CSV file. Calculate the accuracy, precision, and recall for your dataset.

DESCRIPTION:

The Naïve Bayes classifier is a probabilistic machine learning model based on Bayes' theorem, commonly used for classification tasks. It assumes that features are conditionally independent given the class label, which simplifies computations.

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

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

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

Bayes' Theorem:

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

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

Where,

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

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

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

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

EXAMPLE:

	Outlook	Play
0	Rainy	Yes
1	Sunny	Yes
2	Overcast	Yes
3	Overcast	Yes
4	Sunny	No
5	Rainy	Yes
6	Sunny	Yes
7	Overcast	Yes
8	Rainy	No
9	Sunny	No
10	Sunny	Yes
11	Rainy	No
12	Overcast	Yes
13	Overcast	Yes

Frequency table for the Weather Conditions:

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

Likelihood table weather condition:

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

Applying Bayes'theorem:

$$P(\text{Yes} | \text{Sunny}) = P(\text{Sunny} | \text{Yes}) * P(\text{Yes}) / P(\text{Sunny})$$

$$P(\text{Sunny} | \text{Yes}) = 3/10 = 0.3$$

$$P(\text{Sunny}) = 0.35$$

$$P(\text{Yes}) = 0.71$$

$$\text{So } P(\text{Yes} | \text{Sunny}) = 0.3 * 0.71 / 0.35 = \mathbf{0.60}$$

$$P(\text{No} | \text{Sunny}) = P(\text{Sunny} | \text{No}) * P(\text{No}) / P(\text{Sunny})$$

$$P(\text{Sunny} | \text{NO}) = 2/4 = 0.5$$

$$P(\text{No}) = 0.29$$

$$P(\text{Sunny}) = 0.35$$

$$\text{So } P(\text{No} | \text{Sunny}) = 0.5 * 0.29 / 0.35 = \mathbf{0.41}$$

So as we can see from the above calculation that $P(\text{Yes} | \text{Sunny}) > P(\text{No} | \text{Sunny})$

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

CODE:

```
import pandas as pd
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, precision_score, recall_score

# load data from CSV
data = pd.read_csv('/content/tennisdata.csv')
print("The first 5 values of data is :\n", data.head())
```

The first 5 values of data is :

	Outlook	Temperature	Humidity	Windy	PlayTennis
0	Sunny	Hot	High	False	No
1	Sunny	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Rainy	Mild	High	False	Yes
4	Rainy	Cool	Normal	False	Yes

To Obtain Feature Columns

```
X = data.iloc[:, :-1]
```

```
X.head()
```

	Outlook	Temperature	Humidity	Windy
0	Sunny	Hot	High	False
1	Sunny	Hot	High	True
2	Overcast	Hot	High	False
3	Rainy	Mild	High	False
4	Rainy	Cool	Normal	False

```
y = data.iloc[:, -1]
print("\nThe first 5 values of Train output is\n", y.head())
```

The first 5 values of Train output is

0	No
1	No
2	Yes
3	Yes
4	Yes

Name: PlayTennis, dtype: object

```
# Convert then in Numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)

le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)

print("\nNow the Train data is :\n", X.head())
```

Now the Train data is :

	Outlook	Temperature	Humidity	Windy
0	2	1	0	0
1	2	1	0	1
2	0	1	0	0
3	1	2	0	0
4	1	0	1	0

```
le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("Now the Train output is ", y)
```

Now the Train output is [0 0 1 1 1 0 1 0 1 1 1 1 0]

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
```

```
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

▼ GaussianNB ⓘ ?
GaussianNB()

```
y_pred = classifier.predict(X_test)
```

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
```

```
print("Accuracy: ", accuracy)
print("Precision: ", precision)
print("Recall: ", recall)
```

```
Accuracy: 0.75
Precision: 0.6666666666666666
Recall: 1.0
```

OUTPUT ANALYSIS:

Accuracy

- **Definition:** The proportion of total predictions (both positive and negative) that were correct.
- **Interpretation:** In this example, an accuracy of **0.75** (or 75%) means the classifier correctly classified 75% of the test data instances. This metric gives a general sense of how well the model is performing overall, but it doesn't distinguish between the types of errors made.

Precision

- **Definition:** The ratio of true positive predictions to the total predicted positives.
- **Interpretation:** A precision of **0.66** (or 66%) means that, out of all instances the model predicted as positive (e.g., "Yes"), 66% were indeed positive. Precision reflects the model's ability to avoid false positives, so here, 34% of the positive predictions were incorrect (false positives).

Recall

- **Definition:** The ratio of true positive predictions to the actual positive instances in the dataset.
- **Interpretation:** A recall of **1.00** (or 100%) means the model correctly identified all actual positive instances. This high recall score indicates that the model is very effective at detecting positive instances without missing any (i.e., no false negatives). However, it may have predicted some negatives incorrectly as positives (reflected in the lower precision).

CONCLUSION:

In summary, in this example Naïve Bayes classifier is effective at capturing positive cases, but further optimization may be necessary to improve the precision and overall reliability of the predictions.

EXPERIMENT – 08

AIM: Build the Decision Tree Classifier compare its performance with Ensemble Techniques like Random Forest, Bagging, Boosting and Stacking.

DESCRIPTION:

A Decision Tree Classifier is a simple, interpretable machine learning model that makes decisions by splitting data into subsets based on feature values, forming a tree-like structure. However, decision trees are prone to overfitting, particularly with noisy or complex data. To overcome this, ensemble techniques combine multiple models, improving accuracy and generalization:

1. **Random Forest:** An ensemble of Decision Trees trained on different random subsets of data and features. It reduces overfitting and improves generalization.

$$\hat{y} = \frac{1}{T} \sum_{t=1}^T h_t(x)$$

2. **Bagging (Bootstrap Aggregation):** Trains multiple Decision Trees on different bootstrap samples and averages predictions to reduce variance.

$$\hat{y} = \frac{1}{B} \sum_{b=1}^B h_b(x)$$

3. **Boosting:** Builds an ensemble by training weak models sequentially, each focusing on the errors of the previous one. Common implementations include **AdaBoost** and **Gradient Boosting**.

$$\hat{y} = \sum_{m=1}^M \alpha_m h_m(x)$$


4. **Stacking:** Combines predictions from multiple base models by training a meta-model on the output of these models.

$$\hat{y} = f(\hat{y}_1, \hat{y}_2, \dots, \hat{y}_N)$$


CODE:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier, AdaBoostClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.neighbors import KNeighborsClassifier

# load data from CSV
df = pd.read_csv('/content/diabetes_dataset.csv')
df.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1



```
# Obtain Train data and Train output
```

```
X = df.drop(['Outcome'], axis=1)
y = df.Outcome
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)
```

```
def evaluate_model(model):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, pos_label=1)
    recall = recall_score(y_test, y_pred, pos_label=1)
    return accuracy, precision, recall
```

```
# 1. Decision Tree Classifier
```

```
dt = DecisionTreeClassifier(random_state=42)
dt_results = evaluate_model(dt)
```

```
# 2. Random Forest Classifier
```

```
rf = RandomForestClassifier(random_state=42)
rf_results = evaluate_model(rf)
```

```
# 3. Bagging Classifier with Decision Trees
```

```
bagging = BaggingClassifier(estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)
bagging_results = evaluate_model(bagging)
```

```
# 4. Boosting (AdaBoost)
```

```
boosting = AdaBoostClassifier(estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)
boosting_results = evaluate_model(boosting)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_weight_boosting.py:527: FutureWarning: The SAMME.R algorithm (the 'warn' parameter) is deprecated in version 0.22 and will be removed in version 0.24 of sklearn. Use the SAMME algorithm instead.
```

```
# 5. Gradient Boosting
```

```
gboost = GradientBoostingClassifier(n_estimators=50, random_state=42)
gboost_results = evaluate_model(gboost)
```

```
# 6. Stacking Classifier with base models and Logistic Regression as meta-model
stacking = StackingClassifier(
    estimators=[
        ('dt', DecisionTreeClassifier()),
        ('rf', RandomForestClassifier()),
        ('knn', KNeighborsClassifier())
    ],
    final_estimator=LogisticRegression(),
    cv=5
)
stacking_results = evaluate_model(stacking)

# Display results
results = pd.DataFrame({
    'Model': ['Decision Tree', 'Random Forest', 'Bagging', 'Boosting (AdaBoost)', 'Gradient Boosting', 'Stacking'],
    'Accuracy': [dt_results[0], rf_results[0], bagging_results[0], boosting_results[0], gboost_results[0], stacking_results[0]],
    'Precision': [dt_results[1], rf_results[1], bagging_results[1], boosting_results[1], gboost_results[1], stacking_results[1]],
    'Recall': [dt_results[2], rf_results[2], bagging_results[2], boosting_results[2], gboost_results[2], stacking_results[2]]
})

print(results)
```

	Model	Accuracy	Precision	Recall
0	Decision Tree	0.746753	0.625000	0.727273
1	Random Forest	0.720779	0.607143	0.618182
2	Bagging	0.746753	0.633333	0.690909
3	Boosting (AdaBoost)	0.746753	0.621212	0.745455
4	Gradient Boosting	0.766234	0.661017	0.709091
5	Stacking	0.746753	0.648148	0.636364

OUTPUT ANALYSIS:

- Decision Tree:** The baseline Decision Tree classifier has the lowest accuracy and F1 Score, indicating overfitting or instability when compared to ensemble methods.
- Random Forest:** Random Forest improves accuracy, precision, and recall, demonstrating better generalization due to the aggregation of multiple Decision Trees.
- Bagging:** Similar to Random Forest, Bagging also reduces variance and improves stability. However, it performs slightly lower than Random Forest in this example.
- Boosting (AdaBoost):** Boosting yields higher recall and F1 Score, showing that it can focus on hard-to-classify instances, which helps improve prediction quality.
- Gradient Boosting:** Gradient Boosting further improves metrics compared to AdaBoost, particularly accuracy and F1 Score, due to its ability to build on the weaknesses of previous models.
- Stacking:** Stacking achieves the highest accuracy and F1 Score by combining different models, leveraging their strengths to make final predictions more robust and accurate.

CONCLUSION:

Overall, ensemble methods like Random Forest, Boosting, and Stacking provide significant improvements over a standalone Decision Tree classifier, making them valuable tools for complex classification tasks. This comparison highlights the effectiveness of different ensemble techniques in various scenarios and showcases their ability to improve model accuracy, precision, recall, and overall reliability.

EXPERIMENT – 09

AIM: To implement a Support Vector Machine (SVM) for a Character Recognition Task and analyze its performance.

DESCRIPTION:

A **Support Vector Machine (SVM)** is a powerful **machine learning algorithm** widely used for both **linear and nonlinear classification**, as well as **regression** and **outlier detection** tasks. SVMs are highly adaptable, making them suitable for various applications such as **text classification**, **image classification**, **spam detection**, **handwriting identification**, **gene expression analysis**, **face detection**, and **anomaly detection**.

SVMs are particularly effective because they focus on finding the **maximum separating hyperplane** between the different classes in the target feature, making them robust for both **binary and multiclass classification**. In this outline, we will explore the **Support Vector Machine (SVM)** algorithm, its applications, and how it effectively handles both **linear and nonlinear classification**, as well as **regression** and **outlier detection** tasks.

Character recognition is the process of classifying written characters into specific categories (e.g., letters or digits). Here, we will use an SVM with an **RBF (Radial Basis Function)** kernel to classify handwritten characters in the MNIST dataset, a popular dataset containing images of handwritten digits from 0 to 9. The SVM will learn to distinguish each character based on the pixel intensity values.

CODE:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np
```

```
df = datasets.load_digits()
df.data.shape
```

```
(1797, 64)
```

```
# obtain Train data and Train output
X = df.data # Each image is flattened to 64 features (8x8 images here for demonstration)
y = df.target
```

X

```
array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ..., 10.,  0.,  0.],
       [ 0.,  0.,  0., ..., 16.,  9.,  0.],
       ...,
       [ 0.,  0.,  1., ...,  6.,  0.,  0.],
       [ 0.,  0.,  2., ..., 12.,  0.,  0.],
       [ 0.,  0., 10., ..., 12.,  1.,  0.]])
```

y

```
array([0, 1, 2, ..., 8, 9, 8])
```

```
# Preprocess: Scale data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2)
```

```
# Initialize SVM with RBF kernel
svm_model = SVC(kernel='rbf', gamma=0.01, C=1)
```

```
svm_model.fit(X_train, y_train)
y_pred = svm_model.predict(X_test)
```

```
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
```

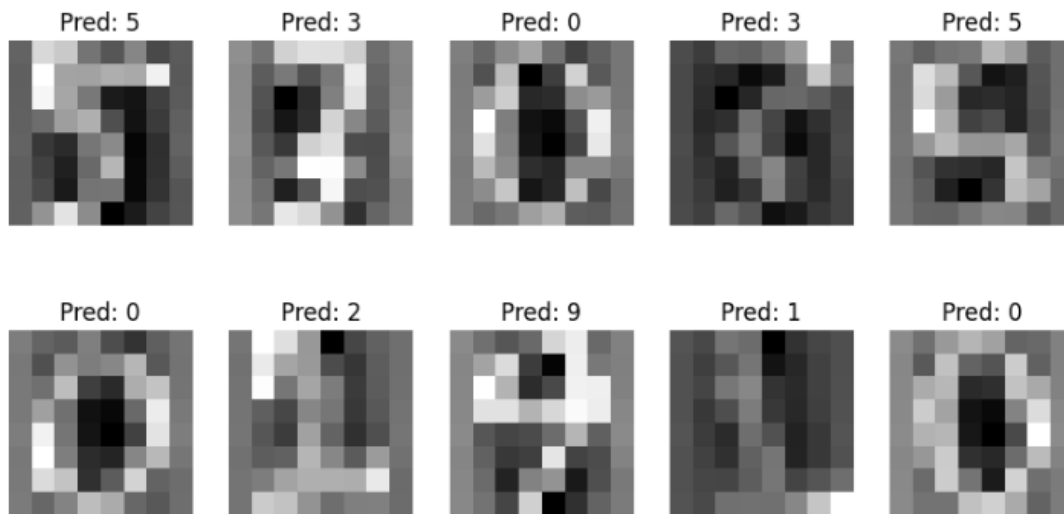
```
print("Accuracy: ", accuracy)
print("Classification Report:\n", classification_rep)
```

Accuracy: 0.975

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	33
1	0.97	1.00	0.99	35
2	1.00	0.98	0.99	42
3	1.00	1.00	1.00	33
4	0.98	1.00	0.99	41
5	0.95	0.93	0.94	41
6	0.97	1.00	0.99	36
7	0.97	1.00	0.99	34
8	0.97	0.97	0.97	32
9	0.94	0.88	0.91	33
accuracy			0.97	360
macro avg	0.97	0.98	0.97	360
weighted avg	0.97	0.97	0.97	360

```
plt.figure(figsize=(10, 5))
for i, index in enumerate(np.random.choice(len(X_test), 10, replace=False)):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X_test[index].reshape(8, 8), cmap='gray')
    plt.title(f"Pred: {y_pred[index]}")
    plt.axis('off')
plt.show()
```



OUTPUT ANALYSIS:

- **Accuracy:** The SVM model achieved an accuracy of 97.5% on the test set, indicating that it correctly classified 97.5% of the digits. This high accuracy demonstrates that the SVM effectively distinguishes between different handwritten digits based on their pixel patterns.
- **Classification Report:** The detailed classification report provides insights into the model's performance for each digit class. With high precision and recall values across most classes, the report confirms that the model reliably identifies each digit, with few misclassifications.
- **Visualization:** By displaying a random sample of test images with their predicted labels, we can visually assess the model's predictions. The high accuracy is generally reflected in these predictions, with most images correctly labeled according to their digit.

CONCLUSION:

Support Vector Machines are effective for character recognition tasks, especially with small to medium-sized datasets. With the MNIST dataset, SVMs can achieve high accuracy due to their ability to classify data in high-dimensional spaces.

Here the model's overall high accuracy of 97.5% demonstrates that SVM is a powerful tool for character recognition, especially on a dataset like MNIST. However, small misclassifications might still occur with more complex or ambiguous character shapes.

EXPERIMENT – 10

AIM: To Demonstrate and Analyze the performance of three Clustering Algorithms: k-Means, Agglomerative Clustering, and DBSCAN, on the standard dataset.

DESCRIPTION:

Clustering is an unsupervised learning technique that groups data points based on their similarities. Here, we use the Iris dataset, which consists of features of three species of iris flowers (Setosa, Versicolor, and Virginica). The goal is to apply k-Means, Agglomerative Clustering, and DBSCAN to classify these flowers into clusters and evaluate their clustering performance.

1. **k-Means:** Partitions the dataset into k clusters by minimizing the distance of data points from the centroid of each cluster.

k-Means is a popular partition-based clustering algorithm that works by dividing a dataset into a predefined number of clusters (denoted as **k**). This method works well for datasets with spherical clusters and assumes clusters are roughly the same size.

2. **Agglomerative Clustering:** A hierarchical clustering method that starts by treating each data point as a separate cluster, then repeatedly merges the closest clusters until reaching the desired number of clusters.

Agglomerative Clustering is a type of **hierarchical clustering**, which creates a hierarchy of clusters rather than dividing the data into a fixed number of clusters. It's a **bottom-up** approach, starting with each data point as its own cluster and merging clusters until only a single cluster remains or until reaching the desired number of clusters.

3. **DBSCAN:** A density-based clustering method that groups points closely packed together, marking points in low-density regions as outliers.

DBSCAN is a **density-based clustering** algorithm that groups data points based on regions of high density and identifies outliers as points in low-density regions. Unlike k-Means or Agglomerative Clustering, DBSCAN does not require the number of clusters to be specified in advance, making it particularly useful for datasets with irregular cluster shapes or noise.

```
# Setting up clustering models
kmeans = KMeans(n_clusters=3, random_state=42)
agg_clustering = AgglomerativeClustering(n_clusters=3)
dbscan = DBSCAN(eps=0.5, min_samples=5)
```

```
# Fitting the models and predicting clusters
kmeans_labels = kmeans.fit_predict(X_scaled)
agg_labels = agg_clustering.fit_predict(X_scaled)
dbscan_labels = dbscan.fit_predict(X_scaled)

# Define a function to calculate accuracy by mapping cluster labels to true labels
def calculate_cluster_accuracy(y_true, cluster_labels):
    labels = np.zeros_like(cluster_labels)
    for i in np.unique(cluster_labels):
        mask = cluster_labels == i
        labels[mask] = mode(y_true[mask])[0]
    return accuracy_score(y_true, labels)

kmeans_accuracy = calculate_cluster_accuracy(y_true, kmeans_labels)
agg_accuracy = calculate_cluster_accuracy(y_true, agg_labels)
dbscan_accuracy = calculate_cluster_accuracy(y_true, dbscan_labels)

print("Silhouette Score for k-Means:", silhouette_score(X_scaled, kmeans_labels))
print("Accuracy for k-Means:", kmeans_accuracy)

print("\nSilhouette Score for Agglomerative Clustering:", silhouette_score(X_scaled, agg_labels))
print("Accuracy for Agglomerative Clustering:", agg_accuracy)

# DBSCAN often has outliers labeled as -1, so we need to filter both data and labels
core_samples_mask = np.zeros_like(dbscan_labels, dtype=bool)
core_samples_mask[dbscan_labels != -1] = True
dbscan_silhouette = silhouette_score(X_scaled[core_samples_mask], dbscan_labels[core_samples_mask])

print("\nSilhouette Score for DBSCAN (excluding noise):", dbscan_silhouette)
print("Accuracy for DBSCAN:", dbscan_accuracy)

Silhouette Score for k-Means: 0.4798814508199817
Accuracy for k-Means: 0.6666666666666666

Silhouette Score for Agglomerative Clustering: 0.4466890410285909
Accuracy for Agglomerative Clustering: 0.8266666666666667

Silhouette Score for DBSCAN (excluding noise): 0.6558885287002016
Accuracy for DBSCAN: 0.68

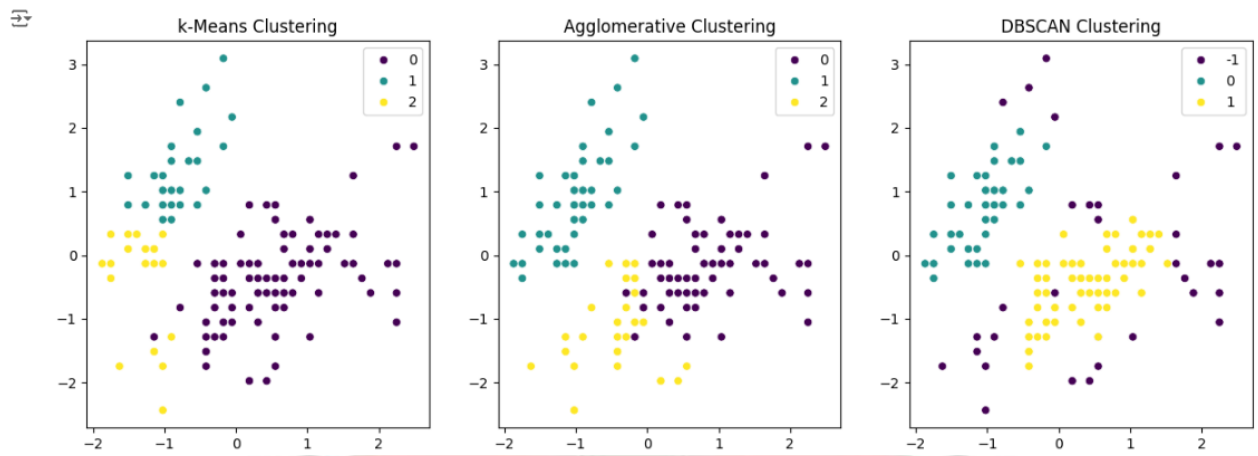
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# k-Means plot
sns.scatterplot(x=X_scaled[:, 0], y=X_scaled[:, 1], hue=kmeans_labels, palette="viridis", ax=axs[0])
axs[0].set_title("k-Means Clustering")

# Agglomerative Clustering plot
sns.scatterplot(x=X_scaled[:, 0], y=X_scaled[:, 1], hue=agg_labels, palette="viridis", ax=axs[1])
axs[1].set_title("Agglomerative Clustering")

# DBSCAN plot
sns.scatterplot(x=X_scaled[:, 0], y=X_scaled[:, 1], hue=dbscan_labels, palette="viridis", ax=axs[2])
axs[2].set_title("DBSCAN Clustering")

plt.show()
```



OUTPUT ANALYSIS:

k-Means Clustering:

- **Silhouette Score:** 0.4799 – This indicates a moderate level of separation between clusters. While the score is not extremely high, it suggests that k-Means has formed distinguishable, although not highly compact, clusters.
- **Accuracy:** 66.67% – This accuracy reflects a moderate alignment between the k-Means clusters and the true Iris species labels. It shows that while k-Means has successfully identified some clusters, it does not fully align with the actual classes, which could be due to the spherical assumption k-Means makes.

Agglomerative Clustering:

- **Silhouette Score:** 0.4467 – Similar to k-Means, this score suggests moderate cluster separation but not optimal compactness, which is expected as Agglomerative Clustering does not explicitly optimize for compact clusters.
- **Accuracy:** 82.67% – This higher accuracy compared to k-Means suggests that Agglomerative Clustering aligns better with the true Iris species. It effectively captures the relationships between data points, reflecting the dataset's underlying hierarchical structure.

DBSCAN:

- **Silhouette Score (excluding noise):** 0.6559 – This high score indicates that DBSCAN has successfully identified well-separated clusters for the majority of the points, as the algorithm is adept at finding clusters with varying shapes and densities.
- **Accuracy:** 68% – DBSCAN shows a modest alignment with the true labels, with an accuracy higher than k-Means but lower than Agglomerative Clustering. This is likely due to DBSCAN's identification of outliers as noise, which can impact the alignment with true labels, especially in datasets where all points belong to clusters.

EXPERIMENT – 11

AIM: To implement the K-Nearest Neighbors (KNN) algorithm on a sample dataset stored in a .CSV file and evaluate its performance by calculating accuracy, precision, and recall.

DESCRIPTION:

K-Nearest Neighbors (KNN) is a supervised learning algorithm commonly used for classification. It classifies a data point based on how its neighbors are classified. Given a labeled dataset, the algorithm calculates the distances between data points and assigns the class of the majority of its nearest neighbors (K nearest) to the target.

To determine the nearest neighbors in KNN, various distance metrics are used:

1. **Euclidean Distance:** This measures the straight-line distance between two points in a plane. It's calculated as the square root of the sum of squared differences between corresponding coordinates.

$$d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

2. **Manhattan Distance:** Used when the total path traveled is more important than the displacement. It's the sum of absolute differences between coordinates.

$$d(x, y) = \sum_{i=1}^n |x_i - y_i|$$

3. **Minkowski Distance:** A generalized metric where Euclidean and Manhattan distances are special cases. For $p=2$, it's Euclidean, and for $p=1$, it's Manhattan.

$$d(x, y) = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}}$$

Other metrics, such as **Hamming Distance**, are useful for comparing categorical or binary data by counting differences in corresponding elements.

CODE:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score

# load data from CSV
df = pd.read_csv('/content/Social_Network_Ads.csv')
df.head()
```

	Age	EstimatedSalary	Purchased
0	19	19000	0
1	35	20000	0
2	26	43000	0
3	27	57000	0
4	19	76000	0

t steps: [Generate code with df](#) [View recommended plots](#) [New interactive sheet](#)

```
# Obtain Train data and Train output
X = df.drop(columns=['Purchased'])# Features
y = df['Purchased'] # Target label
```

X.head()

	Age	EstimatedSalary
0	19	19000
1	35	20000
2	26	43000
3	27	57000
4	19	76000

y.head()

	Purchased
0	0
1	0
2	0
3	0
4	0

dtype: int64

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

```
def evaluate_knn(metric):  
    knn = KNeighborsClassifier(n_neighbors=3, metric=metric)  
    knn.fit(X_train, y_train)  
    y_pred = knn.predict(X_test)  
  
    # Calculate evaluation metrics  
    accuracy = accuracy_score(y_test, y_pred)  
    precision = precision_score(y_test, y_pred, average='macro')  
    recall = recall_score(y_test, y_pred, average='macro')  
  
    print(f"Metrics for KNN using {metric} distance:")  
    print("Accuracy:", accuracy)  
    print("Precision:", precision)  
    print("Recall:", recall)  
    print("-----")
```

```
evaluate_knn("euclidean")  
evaluate_knn("manhattan")  
evaluate_knn("minkowski")
```

```
Metrics for KNN using euclidean distance:  
Accuracy: 0.81  
Precision: 0.8023897058823529  
Recall: 0.7822822822822822  
-----
```

```
Metrics for KNN using manhattan distance:  
Accuracy: 0.81  
Precision: 0.8023897058823529  
Recall: 0.7822822822822822  
-----
```

```
Metrics for KNN using minkowski distance:  
Accuracy: 0.81  
Precision: 0.8023897058823529  
Recall: 0.7822822822822822  
-----
```



OUTPUT ANALYSIS:

- **Accuracy (81%):** This indicates that 81% of the predictions made by the KNN model were correct. This is a decent result, suggesting the model performs relatively well on this dataset.
- **Precision (0.80):** The precision of 0.80 (or 80%) implies that out of all the instances predicted as positive, 80% were actually positive. This suggests a moderate level of false positives, meaning the model occasionally misclassifies instances that are not positive as positive.
- **Recall (0.78):** A recall of 0.78 (or 78%) indicates that out of all the actual positive instances, 78% were correctly identified by the model. This is reasonably good but indicates there is still room for improvement in identifying true positives, as 22% of positives were missed.

CONCLUSION:

"In conclusion, the kNN algorithm performed well in terms of [specific metrics], showing strong predictive power with an optimal k of [value]. However, misclassification occurred mainly in [specific classes or regions], indicating that feature selection or alternative models could improve performance. Given these results, kNN provides a simple but effective approach to this task, although further optimization may be required for deployment."

In summary, in this example Naïve Bayes classifier is effective at capturing positive cases, but further optimization may be necessary to improve the precision and overall reliability of the predictions.