Date: 17-06-2025

Experiment 1: Tic Tac Toe using Exhaustive Search

Aim :

To implement tic tac toe using exhaustive search

Code:

```
board = [" " for x in range(9)]

def print_board():

    row1 = "| {} | {} | {} |".format(board[0], board[1],

    board[2])     row2 = "| {} | {} | {} |".format(board[3],

    board[4], board[5])        row3  =  "|  {}  |  {}  |  {}

    |".format(board[6], board[7], board[8])    print()

    print(row1)

    print(row2)

    print(row3)

    print()

def player_move(icon):

        if icon == "X":

      number = 1

elif icon == "O":

      number = 2

print("Your turn player {}".format(number))

choice = int(input("Enter your move (1-9): ").strip())

if board[choice - 1] == " ":

      board[choice - 1] = icon

else:
```

```
        print()

        print("That space is already taken!")

def is_victory(icon):

if (board[0] == icon and board[1] == icon and board[2]

==icon) or \

        (board[3] == icon and board[4] == icon and board[5] == icon) or \

        (board[6] == icon and board[7] == icon and board[8] == icon) or \

        (board[0] == icon and board[3] == icon and board[6] == icon) or \

        (board[1] == icon and board[4] == icon and board[7] == icon) or \

        (board[2] == icon and board[5] == icon and board[8] == icon) or \

        (board[0] == icon and board[4] == icon and board[8] == icon) or \

        (board[2] == icon and board[4] == icon and board[6] == icon):

        return True
        else:
        return False


def is_draw():

        if " " not in

board:

return True   else:

        return
False while True:

print_board()

player_move("X")

print_board()
        if
is_victory("X"):
```

```
      print("X wins!
Congratulations!")            break
elif is_draw():
      print("It's a draw!")
break
   player_move("O")
         if
is_victory("O"):
print_board()
print("O wins!
Congratulations!")
break  elif
is_draw():
      print("It's a draw!")
break
```

Output:

```
|   |   |   |
|   |   |   |
|   |   |   |

Your turn player 1
Enter your move (1-9):  5

|   |   |   |
|   | X |   |
|   |   |   |

Your turn player 2
Enter your move (1-9):  3

|   |   | O |
|   | X |   |
|   |   |   |

Your turn player 1
Enter your move (1-9):  2

|   | X | O |
|   | X |   |
|   |   |   |

Your turn player 2
Enter your move (1-9):  7

|   | X | O |
|   | X |   |
| O |   |   |

Your turn player 1
Enter your move (1-9):  6

|   | X | O |
|   | X | X |
| O |   |   |

Your turn player 2
Enter your move (1-9):  8

|   | X | O |
|   | X | X |
| O | O |   |

Your turn player 1
Enter your move (1-9):  4

|   | X | O |
| X | X | X |
| O | O |   |

X wins! Congratulations!
```

Date: 24-06-2025

Experiment 2: Water-Jug Problem using BFS and DFS

Aim :

To implement water-jug problem using DFS and BFS

DFS

Code:

```
def water_jug_dfs(c1, c2, t):
    visited = set()
    parent = {}

    def dfs(j1, j2):
        if (j1, j2) in visited:
            return False
        visited.add((j1, j2))

        if j1 == t or j2 == t:
            return True

        # All possible next states
        next_states = [
            (c1, j2),  # fill jug1
            (j1, c2),  # fill jug2
            (0, j2),   # empty jug1
            (j1, 0),   # empty jug2
            (max(0, j1 - (c2 - j2)), min(c2, j1 + j2)),  # pour jug1 -> jug2
            (min(c1, j1 + j2), max(0, j2 - (c1 - j1)))   # pour jug2 -> jug1
        ]

        for state in next_states:
            if state not in visited:
                parent[state] = (j1, j2)
                if dfs(*state):
                    return True
        return False

    start = (0, 0)
    parent[start] = None
    if dfs(*start):
        # reconstruct path
```

```
        path = []
        curr = next(s for s in visited if s[0] == t or s[1] == t)
        while curr is not None:
            path.append(curr)
            curr = parent[curr]
        return path[::-1]
    else:
        return []

# Example usage
c1 = 5
c2 = 3
t = 4

steps = water_jug_dfs(c1, c2, t)
if steps:
    print("The steps are:")
    for step in steps:
        print(step)
else:
    print("No solution found.")
```

OUTPUT :

```
The steps are:
(0, 0)
(4, 0)
(4, 3)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

BFS

Code:

```
from collections import deque

def water_jug_bfs(c1, c2, t):
    visited = set()
    parent = dict()
    queue = deque()
    queue.append((0, 0))
    visited.add((0, 0))
    parent[(0, 0)] = None

    while queue:
```

```
        j1, j2 = queue.popleft()
        if j1 == t or j2 == t:
            path = []
            current = (j1, j2)
            while current is not None:
                path.append(current)
                current = parent[current]
            path.reverse()
            return path
        next_states = [
            (c1, j2),  # Fill jug1
            (j1, c2),  # Fill jug2
            (0, j2),   # Empty jug1
            (j1, 0),   # Empty jug2
            (max(0, j1 - (c2 - j2)), min(c2, j1 + j2)),  # Pour jug1 → jug2
            (min(c1, j1 + j2), max(0, j2 - (c1 - j1)))   # Pour jug2 → jug1
        ]
        for state in next_states:
            if state not in visited:
                visited.add(state)
                parent[state] = (j1, j2)
                queue.append(state)

    return None
c1 = 5
c2 = 3
t = 4
steps = water_jug_bfs(c1, c2, t)
if steps:
    print("The steps are:")
    for step in steps:
        print(step)
else:
    print("There are no steps.")
```

OUTPUT:

```
The steps are:
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

Date: 08-07-2025

Experiment 4: Shortest Path Using GBFS and A* Algorithm

Aim :

To implement the shortest path using greedy best first search and A* algorithm.

Code:

```
import heapq

graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 3), ('E', 1)],
    'C': [('F', 5)],
    'D': [('G', 2)],
    'E': [('G', 1)],
    'F': [('G', 2)],
    'G': []
}

heuristics_to_goal_G = {
    'A': 6.32,
    'B': 5.0,
    'C': 4.47,
    'D': 3.61,
    'E': 2.0,
    'F': 2.23,
    'G': 0.0
}

def a_star_search(graph, heuristics, start, goal):
    open_set = []
    heapq.heappush(open_set, (0 + heuristics[start], 0, start, [start]))
    visited = set()
    while open_set:
        f_score, cost_so_far, current_node, path =
heapq.heappop(open_set)
        if current_node in visited:
            continue
        visited.add(current_node)
        if current_node == goal:
            return path
```

```
        for neighbor, weight in graph[current_node]:
            if neighbor not in visited:
                g = cost_so_far + weight
                h = heuristics[neighbor]
                f = g + h
                heapq.heappush(open_set, (f, g, neighbor, path +
[neighbor]))
    return None

def greedy_best_first_search(graph, heuristics, start, goal):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue, (heuristics[start], start, [start]))
    while priority_queue:
        _, current_node, path = heapq.heappop(priority_queue)
        if current_node in visited:
            continue
        visited.add(current_node)
        if current_node == goal:
            return path
        for neighbor, _ in graph[current_node]:
            if neighbor not in visited:
                heapq.heappush(priority_queue, (
                    heuristics[neighbor],
                    neighbor,
                    path + [neighbor]
                ))
    return None

start_node = 'A'
goal_node = 'G'

print("A* Path:", a_star_search(graph, heuristics_to_goal_G,
start_node, goal_node))
print("Greedy BFS Path:", greedy_best_first_search(graph, heuristics_to_goal_G,
start_node, goal_node))
```

OUTPUT :

```
A* Path: ['A', 'B', 'E', 'G']
Greedy BFS Path: ['A', 'C', 'F', 'G']
```

Date: 15-07-2025

Experiment 3: Hill Climb Racing Problem

Aim :

Develop a search strategy to determine peak element in an array and find the square root of the peak number.

Pseudocode:

```
Function HILL-CLIMBING(problem) returns a state that is a local
maximum.
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Code:

```
import math

def find(arr):
    n = len(arr)
    curr_in = 0

    while True:
        left = arr[curr_in - 1] if curr_in - 1 >= 0 else float('-inf')
        right = arr[curr_in + 1] if curr_in + 1 < n else float('-inf')
        current = arr[curr_in]

        if current >= left and current >= right:
            return arr[curr_in], math.sqrt(arr[curr_in])

        if right > left:
            curr_in += 1
        else:
            curr_in -= 1

if current_in <= 0 or current_in >= n-1:
            return arr[current_in], math.sqrt(arr[current_in])
```

```
arr = [1, 2, 3, 4, 5, 4, 3, 2, 1]
peak, sqrt_peak = find(arr)

print(f"Peak element: {peak}")
print(f"Square root of peak: {sqrt_peak}")
```

OUTPUT:

```
Peak element: 5
Square root of peak: 2.23606797749979
```

Date: 22-07-2025

Experiment 5: Minimax Algorithm and Alpha-Beta Pruning

Aim :

To implement Minimax Algorithm and Alpha-Beta Pruning for Tic-Tac-Toe Game.

A. Minimax algorithm for Tic-Tac-Toe Game:

Pseudocode :

```
function MINIMAX(board, depth, isMaximizing):
    if CHECK_WINNER(board, "O"):
        return +1
    if CHECK_WINNER(board, "X"):
        return -1
    if IS_FULL(board):
        return 0
    if isMaximizing:
        bestScore = -infinity
        for each cell in board:
            if cell is empty:
                place "O" in cell
                score = MINIMAX(board, depth + 1, false)
                undo move
                bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = +infinity
        for each cell in board:
            if cell is empty:
```

```
            place "X" in cell
            score = MINIMAX(board, depth + 1, true)
            undo move
            bestScore = min(score, bestScore)
        return bestScore
function FIND_BEST_MOVE(board):
    bestScore = -infinity
    bestMove = null
    for each cell in board:
        if cell is empty:
            place "O" in cell
            score = MINIMAX(board, 0, false)
            undo move
            if score > bestScore:
                bestScore = score
                bestMove = cell
    return bestMove

function CHECK_WINNER(board, player):
    return true if player has 3 in a row/column/diagonal

function IS_FULL(board):
    return true if no empty cells
```

Code :

```
import math
board = [" " for _ in range(9)]
def print_board():
    for row in [board[i*3:(i+1)*3] for i in range(3)]:
        print("| " + " | ".join(row) + " |")
```

```
def check_winner(b, player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # columns
        [0, 4, 8], [2, 4, 6]              # diagonals
    ]
    for condition in win_conditions:
        if b[condition[0]] == b[condition[1]] == b[condition[2]] == player:
            return True
    return False
def is_full(b):
    return " " not in b
def minimax(b, depth, is_maximizing):
    if check_winner(b, "O"):
        return 1
    elif check_winner(b, "X"):
        return -1
    elif is_full(b):
        return 0
    if is_maximizing:
        best_score = -math.inf
        for i in range(9):
            if b[i] == " ":
                b[i] = "O"
                score = minimax(b, depth + 1, False)
                b[i] = " "
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for i in range(9):
```

```
        if b[i] == " ":
            b[i] = "X"
            score = minimax(b, depth + 1, True)
            b[i] = " "
            best_score = min(score, best_score)
    return best_score
def ai_move():
    best_score = -math.inf
    move = 0
    for i in range(9):
        if board[i] == " ":
            board[i] = "O"
            score = minimax(board, 0, False)
            board[i] = " "
            if score > best_score:
                best_score = score
                move = i
    board[move] = "O"
def play_game():
    print("You are X, AI is O")
    print_board()
    while True:
        move = int(input("Enter your move (1-9): ")) - 1
        if board[move] != " ":
            print("Invalid move. Try again.")
            continue
        board[move] = "X"
        print_board()
        if check_winner(board, "X"):
            print("You win!")
            break
```

```
        elif is_full(board):
            print("It's a draw!")
            break
        ai_move()
        print("\nAI move:")
        print_board()
        if check_winner(board, "O"):
            print("AI wins!")
            break
        elif is_full(board):
            print("It's a draw!")
            break
play_game()
```

OUTPUT :

```
You are X, AI is O
|   |   |   |
|   |   |   |
|   |   |   |
Enter your move (1-9): 1
| X |   |   |
|   |   |   |
|   |   |   |

AI move:
| X |   |   |
|   | O |   |
|   |   |   |
Enter your move (1-9): 2
| X | X |   |
|   | O |   |
|   |   |   |

AI move:
| X | X | O |
|   | O |   |
|   |   |   |
```

```
Enter your move (1-9): 7
| X | X | O |
|   | O |   |
| X |   |   |

AI move:
| X | X | O |
| O | O |   |
| X |   |   |
Enter your move (1-9): 8
| X | X | O |
| O | O |   |
| X | X |   |

AI move:
| X | X | O |
| O | O | O |
| X | X |   |
AI wins!
```

B. Alpha Beta pruning for Tic-Tac-Toe Game:

Pseudocode :

```
function alpha_beta(board, depth, isMaximizing, alpha, beta):
    if game_over(board) or depth == 0:
        return evaluate(board)  // returns +1, -1 or 0

    if isMaximizing:  // Maximizing player: 'X'
        maxEval = -infinity
        for each move in get_available_moves(board):
            make_move(board, move, 'X')
            eval = alpha_beta(board, depth - 1, false, alpha, beta)
            undo_move(board, move)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  // Beta cut-off
        return maxEval

    else:  // Minimizing player: 'O'
        minEval = +infinity
        for each move in get_available_moves(board):
            make_move(board, move, 'O')
            eval = alpha_beta(board, depth - 1, true, alpha, beta)
            undo_move(board, move)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  // Alpha cut-off
        return minEval
function get_available_moves(board):
```

```
    return list of empty cells


function game_over(board):
    return true if a player has won or no moves left


function evaluate(board):
    return +1 if 'X' wins, -1 if 'O' wins, 0 for draw or ongoing
```

Code :

```
AI_PLAYER = 'X'
HUMAN_PLAYER = 'O'
EMPTY = ' '
def terminal_state(board):
    return check_winner(board) is not None or EMPTY not in board
def evaluate(board):
    winner = check_winner(board)
    if winner == AI_PLAYER:
        return +1
    elif winner == HUMAN_PLAYER:
        return -1
    else:
        return 0
def check_winner(board):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]
    for condition in win_conditions:
        a, b, c = condition
        if board[a] == board[b] == board[c] != EMPTY:
            return board[a]
    return None
```

```
def minimax(board, depth, is_maximizing, alpha, beta):
    if terminal_state(board):
        return evaluate(board)
    if is_maximizing:
        max_eval = float('-inf')
        for i in range(9):
            if board[i] == EMPTY:
                board[i] = AI_PLAYER
                eval = minimax(board, depth + 1, False, alpha, beta)
                board[i] = EMPTY
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if alpha >= beta:
                    break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(9):
            if board[i] == EMPTY:
                board[i] = HUMAN_PLAYER
                eval = minimax(board, depth + 1, True, alpha, beta)
                board[i] = EMPTY
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if alpha >= beta:
                    break
        return min_eval
def find_best_move(board):
    best_score = float('-inf')
    best_move = -1
    for i in range(9):
        if board[i] == EMPTY:
            board[i] = AI_PLAYER
            score = minimax(board, 0, False, float('-inf'), float('inf'))
```

```
        board[i] = EMPTY
        if score > best_score:
            best_score = score
            best_move = i
    return best_move
def print_board(board):
    print()
    for i in range(0, 9, 3):
        print(' ' + ' | '.join(board[i:i+3]))
        if i < 6:
            print("---+---+---")
    print()
def get_human_move(board):
    while True:
        try:
            move = int(input("Your move (0-8): "))
            if move < 0 or move > 8:
                print("Invalid input. Enter number between 0 and 8.")
            elif board[move] != EMPTY:
                print("That spot is already taken. Choose another.")
            else:
                return move
        except ValueError:
            print("Please enter a valid integer between 0 and 8.")
def play_game():
    board = [EMPTY] * 9
    current_player = AI_PLAYER
    print_board(board)
    while not terminal_state(board):
        if current_player == AI_PLAYER:
            move = find_best_move(board)
            print(f"AI chooses position: {move}")
            board[move] = AI_PLAYER
        else:
```

```
        move = get_human_move(board)
        board[move] = HUMAN_PLAYER
      print_board(board)
      current_player = HUMAN_PLAYER if current_player == AI_PLAYER else
AI_PLAYER
    winner = check_winner(board)
    if winner:
      print(f"Winner is: {winner}")
    else:
      print("It's a draw!")
play_game()
```

OUTPUT :

```
   |   |
---+---+---
   |   |
---+---+---
   |   |

AI chooses position: 0

 X |   |
---+---+---
   |   |
---+---+---
   |   |

Your move (0-8): 2

 X |   | O
---+---+---
   |   |
---+---+---
   |   |

AI chooses position: 3

 X |   | O
---+---+---
 X |   |
---+---+---
   |   |

Your move (0-8): 6

 X |   | O
---+---+---
 X |   |
---+---+---
 O |   |

AI chooses position: 4

 X |   | O
---+---+---
 X | X |
---+---+---
 O |   |

Your move (0-8): 8

 X |   | O
---+---+---
 X | X |
---+---+---
 O |   | O

AI chooses position: 5

 X |   | O
---+---+---
 X | X | X
---+---+---
 O |   | O

Winner is: X
```

Date: 12-08-2025

Experiment 6: CSP Backtracking Algorithm

Aim :

Develop an approach to solve crypto arithmetic problem using CSP.

Pseudocode :

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)
function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var - SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences - INFERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result- BACKTRACK(assignment, csp)
                if result ≠ failure then
                    return result
        remove {var = value} and inferences from assignment
    return failure
```

Code:

```
from itertools import permutations
def solve_cryptarithmetic(words, result):
  letters = set("".join(words) + result)
```

```
    letters = list(letters)
    if len(letters) > 10:
        raise ValueError("Too many unique letters (max 10 allowed).")
    def word_to_num(word, mapping):
        return int("".join(str(mapping[ch]) for ch in word))
    leading_letters = set(w[0] for w in words + [result])
    for perm in permutations(range(10), len(letters)):
        mapping = dict(zip(letters, perm))
        if any(mapping[l] == 0 for l in leading_letters):
            continue
        word_sum = sum(word_to_num(w, mapping) for w in words)
        result_val = word_to_num(result, mapping)
        if word_sum == result_val:
            return mapping  # Found solution

    return None
solution = solve_cryptarithmetic(["SEND", "MORE"], "MONEY")
print("Solution mapping:", solution)
if solution:
    print("SEND =", int("".join(str(solution[ch]) for ch in "SEND")))
    print("MORE =", int("".join(str(solution[ch]) for ch in "MORE")))
    print("MONEY =", int("".join(str(solution[ch]) for ch in "MONEY")))
```

Output:

```
Solution mapping: {'M': 1, 'S': 9, 'N': 6, 'O': 0, 'E': 5, 'R': 8, 'D': 7, 'Y': 2}
SEND = 9567
MORE = 1085
MONEY = 10652
```

Date: 19-08-2025

Experiment 7: First Order Logic

Aim :

To implement **First Order Logic (FOL)** in Python using constants, predicates, rules, and queries.

Code :

```
students = {"Alice", "Bob", "Charlie", "David"}
courses = {"Math", "Physics", "AI", "Biology"}
def Student(x):
    return x in studentsdef Course(y):
    return y in courses
enrollments = {
    ("Alice", "Math"),
    ("Alice", "AI"),
    ("Bob", "Physics"),
    ("Charlie", "AI"),
    ("David", "Biology"),
    ("David", "Math"),
    ("Charlie", "Physics")
}
def Enrolled(x, y):
    return (x, y) in enrollments
def HasCourse(student):
    return any(Enrolled(student, c) for c in courses)
def Classmates(student1, student2):
    if student1 == student2:
```

```
        return False
    return any(Enrolled(student1, c) and Enrolled(student2, c) for c in courses)
def CoursesOf(student):
    return [c for c in courses if Enrolled(student, c)]
def StudentsIn(course):
    return [s for s in students if Enrolled(s, course)]
print("All students:", students)
print("All courses:", courses)
print("Is Alice a student?", Student("Alice"))
print("Is Biology a student?", Student("Biology"))
print("Is AI a course?", Course("AI"))
print("Is Charlie a course?", Course("Charlie"))
print("Is Alice enrolled in Math?", Enrolled("Alice", "Math"))
print("Is Bob enrolled in AI?", Enrolled("Bob", "AI"))
print("Courses of Alice:", CoursesOf("Alice"))
print("Courses of David:", CoursesOf("David"))print("Students in AI:", StudentsIn("AI"))
print("Students in Math:", StudentsIn("Math"))
print("Does Bob have at least one course?", HasCourse("Bob"))
print("Are Alice and Charlie classmates?", Classmates("Alice", "Charlie"))
print("Are Alice and David classmates?", Classmates("Alice", "David"))
```

Output:

```
All students: {'David', 'Alice', 'Bob', 'Charlie'}
All courses: {'AI', 'Biology', 'Math', 'Physics'}
Is Alice a student? True
Is Biology a student? False
Is AI a course? True
Is Charlie a course? False
Is Alice enrolled in Math? True
Is Bob enrolled in AI? False
Courses of Alice: ['AI', 'Math']
Courses of David: ['Biology', 'Math']
Students in AI: ['Alice', 'Charlie']
Students in Math: ['David', 'Alice']
Does Bob have at least one course? True
Are Alice and Charlie classmates? True
Are Alice and David classmates? True
```

Experiment 8: Forward Chaining and Backward Chaining

Aim :

To implement forward chaining and backward chaining.

Code :

```
#mammal(A) ==> vertebrate(A).
#vertebrate(A) ==> animal(A).
#vertebrate(A),flying(A) ==> bird(A).
#vertebrate("duck").
#flying("duck").
#mammal("cat").
global facts
global is_changed

is_changed = True
facts = [["vertebrate","duck"],["flying","duck"],["mammal","cat"]]

def assert_fact(fact):
  global facts
  global is_changed
  if not fact in facts:
    facts += [fact]
    is_changed = True

while is_changed:
  is_changed = False
```

```
  for A1 in facts:
    if A1[0] == "mammal":
      assert_fact(["vertebrate",A1[1]])
    if A1[0] == "vertebrate":
      assert_fact(["animal",A1[1]])
    if A1[0] == "vertebrate" and ["flying",A1[1]] in facts:
      assert_fact(["bird",A1[1]])

print(facts)
```

Output:

```
E:\5BTCS\AIML>python exp8.py
[['vertebrate', 'duck'], ['flying', 'duck'], ['mammal', 'cat'], ['animal', 'duck'], ['bird', 'duck'], ['vertebrate', 'cat'], ['animal', 'cat']]
```

Date: 20-09-2025

Experiment 9: Linear Regression

Aim :

To implement Linear regression on real time dataset and evaluate its performance.

Steps to Implement Linear Regression in KNIME:

1. Import real time dataset using CSV Reader node.

2. Use Column Filter node to select required columns for performing linear regression.

3. Use Partitioning node to set split ratio (e.g., 70% training, 30% testing).

4. Use the Linear Regression Learner node. Select target (dependent variable) in the configuration.

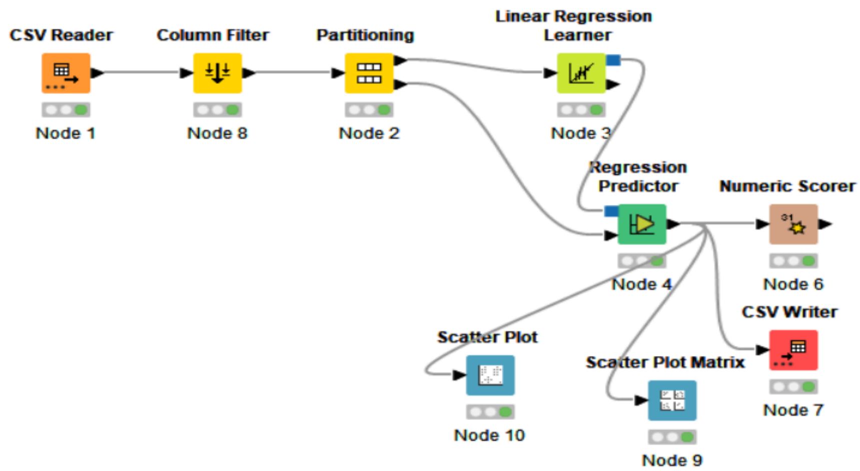
5. Use the Linear Regression Predictor node. Make connections to Linear Regression Learner node and Partitioning node.

6. Use Numeric Scorer node for regression problems. It gives metrics like R^2, MSE, RMSE.

7. Use Scatter Plot and Scatter Plot matrix nodes to compare predicted vs. actual values and to visualize linear regression.

8. Use CSV Writer node to save the predicted values in the existing csv file.

Connection Diagram:



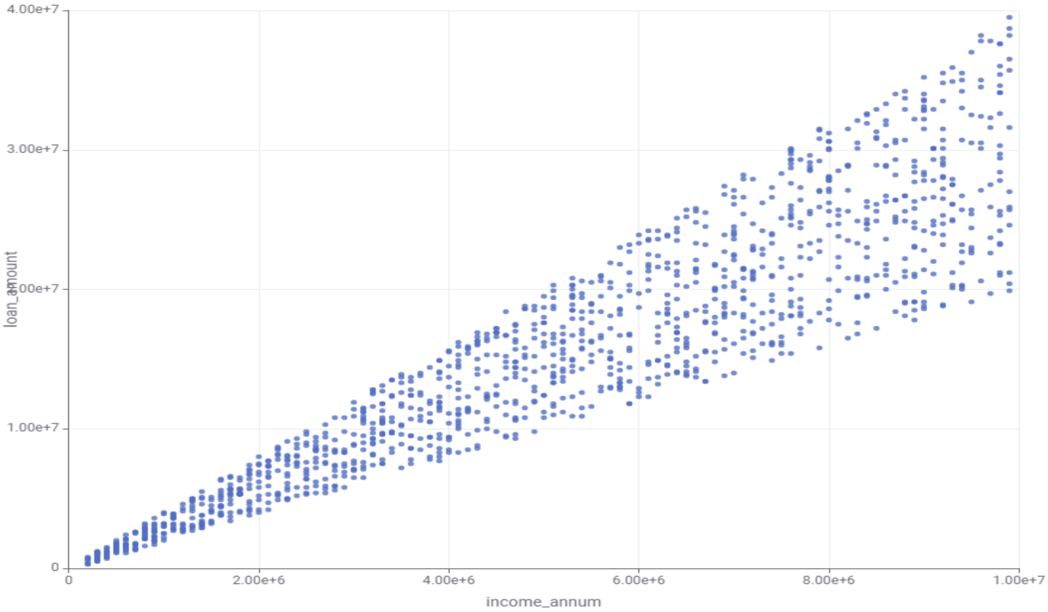
Results:

Statistics (Statistics)

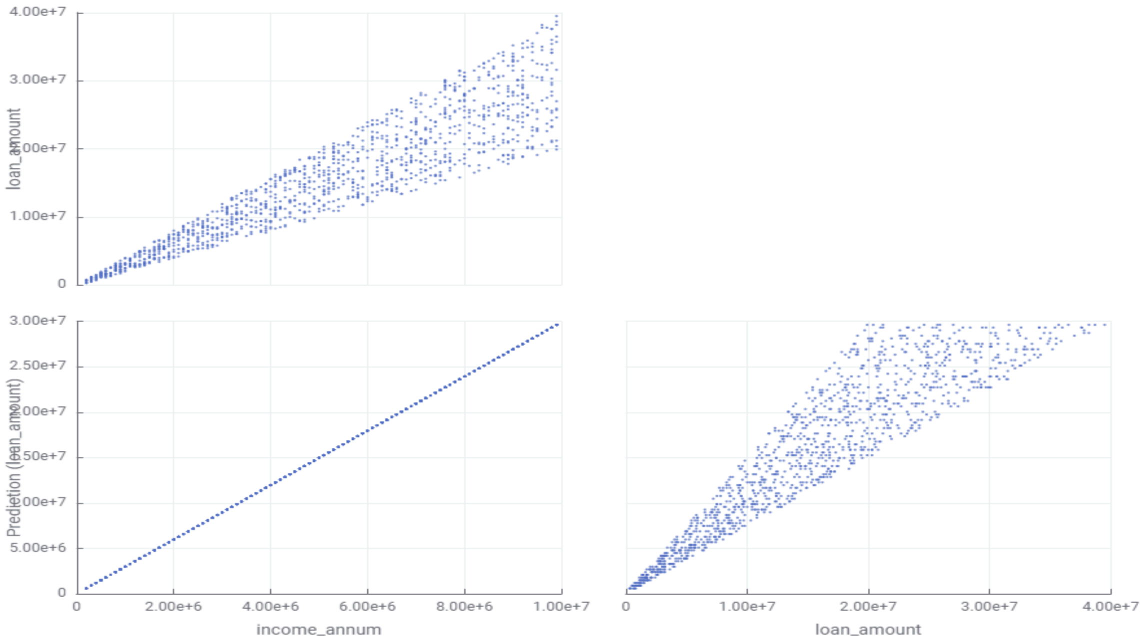
Rows: 1 | Columns: 14

Name	Type	# Missing val...	# Unique val...	Minimum	Maximum	25% Quantile	50% Quantile...	75% Quantile	Mean
Prediction (loar	Number (doubl	0	6	0.181	11,365,659,806	0.861	22,937.833	3,371,299.424	1,623,666,535,!

Scatter Plot



Scatter Plot Matrix



Date: 20-09-2025

Experiment 10: Decision Tree Classifier

Aim :

To build a model using Decision trees on real time dataset and evaluate its performance

Steps to Implement:

1. Import real time dataset using CSV Reader node.

2. Use Column Filter node to select required columns for performing linear regression.

3. Use Color Manger Node (optional) for visualizing classified data

4. Use Partitioning node to set split ratio (e.g., 70% training, 30% testing).

5. Use the Decision Tree Learner node. Select appropriate class column in the

configuration.

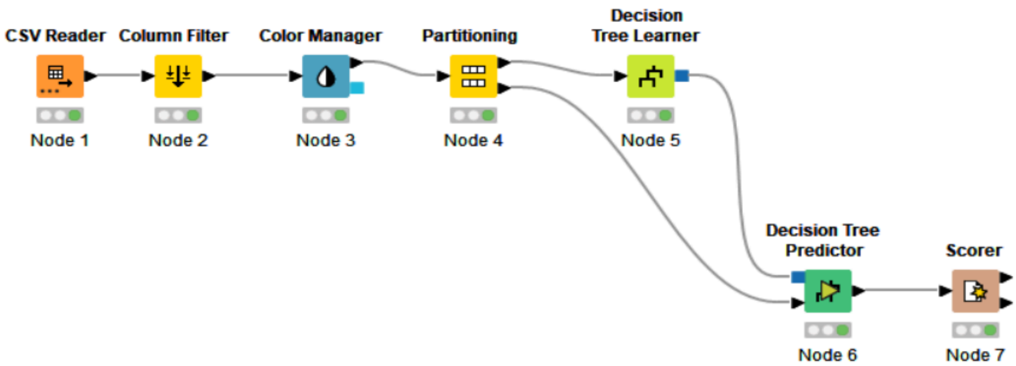
6. Use the Decision Tree Predictor node. Make connections to Decision Tree Learner

node and Partitioning node.

7. Use Scorer node for classification problems. It gives metrics like Accuracy, Precision,

Recall, Confusion Matrix.

Connection Diagram:



Results:

Rows: 2 | Columns: 14

Name	Type	# Missing val...	# Unique val...	Minimum	Maximum	25% Quantile	50% Quantile...	75% Quantile	Mean
Approved	Number (intege	0	2	22	787	22	404.5	787	404.5
Rejected	Number (intege	0	2	12	460	12	236	460	236

Rows: 11 | Columns: 14

Name	Type	# Missing val...	# Unique val...	Minimum	Maximum	25% Quantile	50% Quantile...	75% Quantile	Mean
TruePositives	Number (intege	1	2	460	787	460	623.5	787	623.5
FalsePositives	Number (intege	1	2	12	22	12	17	22	17
TrueNegatives	Number (intege	1	2	460	787	460	623.5	787	623.5
FalseNegatives	Number (intege	1	2	12	22	12	17	22	17
Recall	Number (doubl	1	2	0.954	0.985	0.954	0.97	0.985	0.97
Precision	Number (doubl	1	2	0.973	0.975	0.973	0.974	0.975	0.974
Sensitivity	Number (doubl	1	2	0.954	0.985	0.954	0.97	0.985	0.97
Specificity	Number (doubl	1	2	0.954	0.985	0.954	0.97	0.985	0.97
F-measure	Number (doubl	1	2	0.964	0.979	0.964	0.972	0.979	0.972
Accuracy	Number (doubl	2	1	0.973	0.973	0.973	0.973	0.973	0.973
Cohen's kappa	Number (doubl	2	1	0.943	0.943	0.943	0.943	0.943	0.943

Confusion Matrix - 3:8 - Scorer

File Hilite

loan_statu...	Approved	Rejected
Approved	787	12
Rejected	22	460

Correct classified: 1,247 Wrong classified: 34

Accuracy: 97.346% Error: 2.654%

Cohen's kappa (κ): 0.943%