COURSE MATERIAL

III Year B. Tech II- Semester

MECHANICAL ENGINEERING

AY: 2023-24

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LAB

R20A0580

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF MECHANICAL ENGINEERING

(Autonomous Institution-UGC, Govt. of India)
Secunderabad-500100, Telangana State, India.
www.mrcet.ac.in

MRCET CAMPUS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS INSTITUTION - UGC, GOVT. OF INDIA)

Affiliated to JNTUH; Approved by AICTE, NBA-Tier 1 & NAAC with A-GRADE | ISO 9001:2015

Maisammaguda, Dhulapally, Komaplly, Secunderabad - 500100, Telangana State, India

LABORATORY MANUAL & RECORD

Name:.....

Roll No:.....Branch:.....

Year:.....Sem:.....

MRCET CAMPUS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS INSTITUTION - UGC, GOVT. OF INDIA)

Affiliated to JNTUH; Approved by AICTE, NBA-Tier 1 & NAAC with A-GRADE | ISO 9001:2015

Maisammaguda, Dhulapally, Komaplly, Secunderabad - 500100, Telangana State, India

Certificate

Certified that this is the Bonafide Record of the Work Done by

Mr./Ms.....Roll.No.....of

B.Tech year ..... Semester for Academic year 2023 - 2024

in.....Laboratory.

Date: Faculty Incharge HOD

Internal Examiner External Examiner

INDEX

S.No	Date	Name of the Activity/Experiment	Grade/ Marks	Faculty Signature

INDEX

S.No	Date	Name of the Activity/Experiment	Grade/ Marks	Faculty Signature

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

DEPARTMENT OF MECHANICAL ENGINEERING

CONTENTS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

VISION

- To establish a pedestal for the integral innovation, team spirit, originality and competence in the students, expose them to face the global challenges and become technology leaders of Indian vision of modern society.

MISSION

- To become a model institution in the fields of Engineering, Technology and Management.
- To impart holistic education to the students to render them as industry ready engineers.
- To ensure synchronization of MRCET ideologies with challenging demands of International Pioneering Organizations.

QUALITY POLICY

- To implement best practices in Teaching and Learning process for both UG and PG courses meticulously.
- To provide state of art infrastructure and expertise to impart quality education.
- To groom the students to become intellectually creative and professionally competitive.
- To channelize the activities and tune them in heights of commitment and sincerity, the requisites to claim the never - ending ladder of **SUCCESS** year after year.

For more information: www.mrcet.ac.in

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

www.mrcet.ac.in

Department of Mechanical Engineering

VISION

To become an innovative knowledge center in mechanical engineering through state-of-the-art teaching-learning and research practices, promoting creative thinking professionals.

MISSION

The Department of Mechanical Engineering is dedicated for transforming the students into highly competent Mechanical engineers to meet the needs of the industry, in a changing and challenging technical environment, by strongly focusing in the fundamentals of engineering sciences for achieving excellent results in their professional pursuits.

Quality Policy

- ✓ To pursuit global Standards of excellence in all our endeavors namely teaching, research and continuing education and to remain accountable in our core and support functions, through processes of self-evaluation and continuous improvement.
- ✓ To create a midst of excellence for imparting state of art education, industry-oriented training research in the field of technical education.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

www.mrcet.ac.in

Department of Mechanical Engineering

PROGRAM OUTCOMES

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and teamwork**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

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

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1 Ability to analyze, design and develop Machine learning systems to solve the Engineering problems by integrating design and manufacturing Domains.

PSO2 Ability to succeed in competitive examinations or to pursue higher studies or research.

PSO3 Ability to apply the learned Mechanical Engineering knowledge for the Development of society and self.

Program Educational Objectives (PEOs)

The Program Educational Objectives of the program offered by the department are broadly listed below:

PEO1: PREPARATION

To provide sound foundation in mathematical, scientific and engineering fundamentals necessary to analyze, formulate and solve engineering problems.

PEO2: CORE COMPETANCE

To provide thorough knowledge in Mechanical Engineering subjects including theoretical knowledge and practical training for preparing Artificial models pertaining to Automobile Engineering, Element Analysis,Production Technology,Mechatronics etc.,

PEO3: INVENTION, INNOVATION AND CREATIVITY

To make the students to design, experiment, analyze, interpret in the core field with the help of other inter disciplinary concepts wherever applicable.

PEO4: CAREER DEVELOPMENT

To inculcate the habit of lifelong learning for career development through successful completion of advanced degrees, professional development courses, industrial training etc.

PEO5: PROFESSIONALISM

To impart technical knowledge, ethical values for professional development of the student to solve complex problems and to work in multi-disciplinary ambience, whose solutions lead to significant societal benefits.

CODE OF CONDUCT

1. Students should bring lab Manual/Record for every laboratory session and should enter the readings/observations in the manual while performing the experiment.
2. The group- wise division made in the beginning should be adhered to, and no mix up of students among different groups will be permitted later.
3. The components required pertaining to the experiment should be collected from stores in –charge after duly filling in the requisition form.
4. When the experiment is completed, students should disconnect the setup made by them, and should return all the components/instruments taken for the purpose.
5. Any damage to the apparatus that occurs during the experiment should be brought to the notice of lab in-charge, consequently, the cost of repair or new apparatus should be brought by the students.
6. After completion of the experiment, certification of the concerned staff in –charge in the observation book is necessary.
7. Students should be present in the labs for the total scheduled duration.
8. Students should not carry any food items inside the laboratory.
9. Use of cell phones and IPODs are forbidden.
10. Students should not write on or deface any lab desks, computers, or any equipment provided to them during the experiment.
11. Every student should keep his/her work area properly before leaving the laboratory.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(AutonomousInstitution– UGC,Govt.ofIndia)

www.mrcet.ac.in

Department of Mechanical Engineering

INDEX

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

III Year B.Tech. ME- II Sem **L/T/P/C**
-/-/3/1.5

(R20A0580) ARTIFICIAL INTELLIGENCE & MACHINE LEARNING LAB

LAB OBJECTIVES:

1. Familiarity with the Prolog programming environment.
2. To introduce students to the basic concepts and techniques of Machine Learning.
3. To implement classification and clustering methods.
4. To become familiar with Dimensionality reduction Techniques.
5. Learning basic concepts of Prolog through illustrative examples and small exercises &Understanding list data structure in Prolog.

STUDY OF PROLOG; WRITE THE FOLLOWING PROGRAMS USING PROLOG/PYTHON

week-1. Write a program to implement all set operations(Union, Intersection, Complement etc)
week-2.Implementation of DFS for water jug problem using PROLOG
week-3. Implementation of BFS for tic-tac-toe problem using PROLOG
week-4. Solve 8-puzzle problem using best first search
week-5. Write a program to solve 8 queens problem
week-6. Implementation of Hill-climbing to solve 8- Puzzle Problem

MACHINE LEARNING

WEEK-1
Data Extraction, Wrangling
1. Loading different types of dataset in Python
2. Arranging the data

WEEK-2
Data Visualization
1. Handling missing values
2. Plotting the graphs

WEEK-3
Supervised Learning
Implementation of Linear Regression

WEEK-4
Implementation of K-nearest Neighbor

WEEK-5
Unsupervised Learning
Implementing K-means Clustering

WEEK-6
Unsupervised Learning
Implementing Hierarchical Clustering

LAB OUTCOMES:
1. Apply various AI search algorithms (uninformed, informed, heuristic, constraint satisfaction,)

2. Understand the fundamentals of knowledge representation, inference using AI tools..
3. Solve the problems using various machine learning techniques
4. Design application using machine learning techniques

PROGRAM-1

Sets and Set Operations in Python

A set is defined by enclosing all of the items (i.e., elements) in curly brackets and separating them with a comma or using the built-in set() method. It can include an unlimited number of elements of various categories (integer, float, tuple, string, etc.).

However, a set may not contain mutable items such as lists, sets, or dictionaries. Empty sets can be slightly tricky to use in Python. In Python, empty curly braces result in an empty dictionary; however, we cannot use them to initialize an empty set. Instead, we use the set() function without any arguments to create a set with no elements.

```
# Program to perform different set operations

# as we do in mathematics

# sets are define

A = {0, 2, 4, 6, 8};

B = {1, 2, 3, 4, 5};

# union

print("Union :", A | B)

# intersection

print("Intersection :", A & B)

# difference

print("Difference :", A - B)

# symmetric difference

print("Symmetric difference :", A ^ B)
```

Output:

```
('Union :', set([0, 1, 2, 3, 4, 5, 6, 8]))

('Intersection :', set([2, 4]))

('Difference :', set([8, 0, 6]))

('Symmetric difference :', set([0, 1, 3, 5, 6, 8]))
```

PROGRAM-2

Implement Water-Jug problem using Python

This function is used to initialize thedictionary elements with a default value.

```
from collections import defaultdict

# jug1 and jug2 contain the valuefor max capacity in respective jugsand aim is the amount of water to be measured.

jug1, jug2, aim = 4, 3, 2

# Initialize dictionary withdefault value as false.

visited = defaultdict(lambda: False)

# Recursive function which prints theintermediate steps to reach the finalsolution and return Boolean value(True if solution is possible, otherwise False).amt1 and amt2 are the amount of water presentin both jugs at a certain point of time.

def waterJugSolver(amt1, amt2):

# Checks for our goal andreturns true if achieved.

        if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

                print (amt1, amt2)

                return True

# Checks if we have already visited thecombination or not. If not, then it proceeds further.

        if visited [(amt1, amt2)] == False:

                print (amt1, amt2)

# Changes the Boolean value ofthe combination as it is visited.

                Visited [(amt1, amt2)] = True

# Check for all the 6 possibilities andsee if a solution is found in any one of them.

                return (waterJugSolver(0, amt2) or

                                waterJugSolver(amt1, 0) or
```

```
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))
```

Return False if the combination is already visited to avoid repetition otherwiserecursion will enter an infinite loop.

```
        else:
                return False
print ("Steps: ")
```

Call the function and pass the initial amount of water present in both jugs.

```
waterJugSolver(0, 0)
```

OUTPUT:

Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2

PROGRAM-3

Tic-Tac-Toe Program using random number in Python

```
# importing all necessary libraries

import numpy as np

import random

from time import sleep

# Creates an empty board

def create_board():

        return(np.array([[0, 0, 0],

                                        [0, 0, 0],

                                        [0, 0, 0]]))

# Check for empty places on board

def possibilities(board):

        l = []

        for i in range(len(board)):

                for j in range(len(board)):

                        if board[i][j] == 0:

                                l.append((i, j))

        return(l)

# Select a random place for the player

def random_place(board, player):

        selection = possibilities(board)

        current_loc = random.choice(selection)

        board[current_loc] = player
```

```
        return(board)
# Checks whether the player has three of their marks in a horizontal row
def row_win(board, player):
        for x in range(len(board)):
                win = True
                for y in range(len(board)):
                        if board[x, y] != player:
                                win = False
                                continue
                if win == True:
                        return(win)
        return(win)


# Checks whether the player has three of their marks in a vertical row
def col_win(board, player):
        for x in range(len(board)):
                win = True
                for y in range(len(board)):
                        if board[y][x] != player:
                                win = False
                                continue
                if win == True:
                        return(win)
        return(win)
```

```
# Checks whether the player has three of their marks in a diagonal row
def diag_win(board, player):
    win = True
    y = 0
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
    if win:
        return win
    win = True
    if win:
        for x in range(len(board)):
            y = len(board) - 1 - x
            if board[x, y] != player:
                win = False
    return win


# Evaluates whether there is a winner or a tie
def evaluate(board):
    winner = 0
    for player in [1, 2]:
        if (row_win(board, player) or
                col_win(board, player) or
```

```
                                diag_win(board, player)):
                        winner = player
        if np.all(board != 0) and winner == 0:
                winner = -1
        return winner
# Main function to start the game
def play_game():
        board, winner, counter = create_board(), 0, 1
        print(board)
        sleep(2)

        while winner == 0:
                for player in [1, 2]:
                        board = random_place(board, player)
                        print("Board after " + str(counter) + " move")
                        print(board)
                        sleep(2)
                        counter += 1
                        winner = evaluate(board)
                        if winner != 0:
                                break
        return(winner)
# Driver Code
print("Winner is: " + str(play_game()))
```

OUTPUT:

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 0]
 [1 0 0]]
Board after 2 move
[[0 0 0]
 [0 2 0]
 [1 0 0]]
Board after 3 move
[[0 1 0]
 [0 2 0]
 [1 0 0]]
Board after 4 move
[[0 1 0]
 [2 2 0]
 [1 0 0]]
Board after 5 move
[[1 1 0]
 [2 2 0]
 [1 0 0]]
Board after 6 move
[[1 1 0]
 [2 2 0]
 [1 2 0]]
Board after 7 move
[[1 1 0]
```

[2 2 0]

[1 2 1]]

Board after 8 move

[[1 1 0]

[2 2 2]

[1 2 1]]

Winner is: 2

PROGRAM-4

Solving 8-Puzzle using BFS

The solution assumes that instance of puzzle is solvable

```
# Importing copy for deep-copy function

import copy


# Importing the heap functions from python library for Priority Queue

from heapq import heappush, heappop


# This variable can be changed to change the program from 8 puzzle(n=3) to 15 puzzle(n=4)
to 24 puzzle(n=5)...

n = 3

# bottom, left, top, right

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]

# A class for Priority Queue

class priorityQueue:

# Constructor to initialize a Priority Queue

        def __init__(self):

                self.heap = []



# Inserts a new key 'k'

        def push(self, k):

                heappush(self.heap, k)
```

```
# Method to remove minimum element from Priority Queue
        def pop(self):
                return heappop(self.heap)


# Method to know if the Queue is empty
        def empty(self):
                if not self.heap:
                        return True
                else:
                        return False
# Node structure
class node:
        def __init__(self, parent, mat, empty_tile_pos,
                                cost, level):
# Stores the parent node of the current node helps in tracing path when the answer is found
                self.parent = parent
# Stores the matrix
                self.mat = mat
# Stores the position at which the empty space tile exists in the matrix
                self.empty_tile_pos = empty_tile_pos
# Stores the number of misplaced tiles
                self.cost = cost
# Stores the number of moves so far
```

```
        self.level = level

# This method is defined so that the priority queue is formed based on the cost variable of the objects
    def __lt__(self, nxt):
        return self.cost<nxt.cost
# Function to calculate the number of misplaced tiles ie. number of non-blank tiles not in their goal position
def calculateCost(mat, final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1
    return count
def newNode(mat, empty_tile_pos, new_empty_tile_pos,
                level, parent, final) -> node:
# Copy data from parent matrix to current matrix
    new_mat = copy.deepcopy(mat)
# Move tile by 1 position
    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
```

```
        y2 = new_empty_tile_pos[1]
        new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]


# Set number of misplaced tiles
        cost = calculateCost(new_mat, final)
        new_node = node(parent, new_mat, new_empty_tile_pos,
                                        cost, level)
        return new_node


# Function to print the N x N matrix
def printMatrix(mat):
        for i in range(n):
                for j in range(n):
                        print("%d " % (mat[i][j]), end = " ")
                print()
# Function to check if (x, y) is a valid matrix coordinate
def isSafe(x, y):
        return x >= 0 and x < n and y >= 0 and y < n
# Print path from root node to destination node
def printPath(root):
        if root == None:
                return
        printPath(root.parent)
        printMatrix(root.mat)
```

```
        print()
# Function to solve N*N - 1 puzzle algorithm using Branch and Bound. empty_tile_pos is the blank tile position in the initial state.
def solve(initial, empty_tile_pos, final):
# Create a priority queue to store live nodes of search tree
        pq = priorityQueue()
# Create the root node
        cost = calculateCost(initial, final)
        root = node(None, initial,
                                empty_tile_pos, cost, 0)
# Add root to list of live nodes
        pq.push(root)
# Finds a live node with least cost, add its children to list of live nodes and finally deletes it from the list.
        while not pq.empty():

# Find a live node with least estimated cost and delete it form the list of live nodes
                minimum = pq.pop()

# If minimum is the answer node
                if minimum.cost == 0:

# Print the path from root to destination;
                        printPath(minimum)
                        return
```

```
# Generate all possible children
                for i in range(4):
                        new_tile_pos = [
                                minimum.empty_tile_pos[0] + row[i],
                                minimum.empty_tile_pos[1] + col[i], ]

                        if isSafe(new_tile_pos[0], new_tile_pos[1]):
# Create a child node
                                child = newNode(minimum.mat,
                                                minimum.empty_tile_pos,
                                                new_tile_pos,
                                                minimum.level + 1,
                                                minimum, final,)

# Add child to list of live nodes
                                pq.push(child)

# Driver Code

# Initial configuration Value 0 is used for empty space
initial = [ [ 1, 2, 3 ],
                        [ 5, 6, 0 ],
                        [ 7, 8, 4 ] ]
```

```
# Solvable Final configuration Value 0 is used for empty space
final = [ [ 1, 2, 3 ],
          [ 5, 8, 6 ],
          [ 0, 7, 4 ] ]

# Blank tile coordinates in initial configuration
empty_tile_pos = [ 1, 2 ]

# Function call to solve the puzzle
solve(initial, empty_tile_pos, final)
```

Output:

```
1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4
```

PROGRAM-5

Program to solve 8-queen problem

Python program to solve N QueenProblem using backtrackingglobal N

```
N = 4

def printSolution(board):
        for i in range(N):
                for j in range(N):
                        print (board[i][j],end=' ')
                print()
```

A utility function to check if a queen can be placed on board[row][col]. Note that this function is called when "col" queens are already placed in columns from 0 to col -1. So we need to check only left side for attacking queens

```
def isSafe(board, row, col):
# Check this row on left side
        for i in range(col):
                if board[row][i] == 1:
                        return False


# Check upper diagonal on left side
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
                if board[i][j] == 1:
                        return False


# Check lower diagonal on left side
        for i, j in zip(range(row, N, 1), range(col, -1, -1)):
```

```
                if board[i][j] == 1:
                        return False

        return True

def solveNQUtil(board, col):
        # base case: If all queens are placed then return true
        if col >= N:
                return True

# Consider this column and try placing this queen in all rows one by one
        for i in range(N):

                if isSafe(board, i, col):
# Place this queen in board[i][col]
                        board[i][col] = 1

# recur to place rest of the queens
                        if solveNQUtil(board, col + 1) == True:
                                return True

# If placing queen in board[i][col doesn't lead to a solution, then queen from board[i][col]
                        board[i][col] = 0
```

```
# if the queen can not be placed in any row in this column col then return false
        return False

# This function solves the N Queen problem using Backtracking. It mainly uses solveNQUtil() to solve the problem. It returns false if queens cannot be placed, otherwise return true and placement of queens in the form of 1s. note that there may be more than one solutions, this function prints one of the feasible solutions.
def solveNQ():
        board = [ [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0]
                        ]

        if solveNQUtil(board, 0) == False:
                print ("Solution does not exist")
                return False

        printSolution(board)
        return True

# driver program to test above function
solveNQ()
```

Output:

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

Python3 implementation of the above approach

```
from random import randint

N = 8

# A utility function that configures the 2D array "board" and array "state" randomly to provide a starting point for the algorithm.

def configureRandomly(board, state):

# Iterating through the column indices

        for i in range(N):

# Getting a random row index

                state[i] = randint(0, 100000) % N;


# Placing a queen on the obtained place in chessboard.

                board[state[i]][i] = 1;


# A utility function that prvars the 2D array "board".

def prvarBoard(board):


        for i in range(N):

                print(*board[i])


# A utility function that prvars the array "state".

def prvarState( state):

        print(*state)


# A utility function that compares two arrays, state1 and state2 and returns True if equal and False otherwise.
```

```
def compareStates(state1, state2):
        for i in range(N):
                if (state1[i] != state2[i]):
                        return False;
        return True;


# A utility function that fills the 2D array "board" with values "value"
def fill(board, value):
        for i in range(N):
                for j in range(N):
                        board[i][j] = value;


# This function calculates the objective value of the state(queens attacking each other) using the board by the following logic.
def calculateObjective( board, state):


# For each queen in a column, we check for other queens falling in the line of our current queen and if found, any, then we increment the variable attacking count.
# Number of queens attacking each other, initially zero.
        attacking = 0;


# Variables to index a particular row and column on board.
        for i in range(N):


# At each column 'i', the queen is placed at row 'state[i]', by the definition of our state.
# To the left of same row (row remains constant and col decreases)
                row = state[i]
                col = i - 1;
                while (col >= 0 and board[row][col] != 1) :
                        col -= 1
                if (col >= 0 and board[row][col] == 1) :
```

```
                attacking += 1;

# To the right of same row (row remains constant and col increases)
                row = state[i]
                col = i + 1;
                while (col < N and board[row][col] != 1):
                        col += 1;

                if (col < N and board[row][col] == 1) :
                        attacking += 1;

# Diagonally to the left up (row and col simultaneously decrease)
                row = state[i] - 1
                col = i - 1;
                while (col >= 0 and row >= 0 and board[row][col] != 1) :
                        col-= 1;
                        row-= 1;
                if (col >= 0 and row >= 0 and board[row][col] == 1) :
                        attacking+= 1;

# Diagonally to the right down (row and col simultaneously increase)
                row = state[i] + 1
                col = i + 1;
                while (col < N and row < N and board[row][col] != 1) :
                        col+= 1;
                        row+= 1;
                if (col < N and row < N and board[row][col] == 1) :
                        attacking += 1;

# Diagonally to the left down (col decreases and row increases)
```

```
                row = state[i] + 1
                col = i - 1;
                while (col >= 0 and row < N and board[row][col] != 1) :
                        col -= 1;
                        row += 1;

                if (col >= 0 and row < N and board[row][col] == 1) :
                        attacking += 1;

# Diagonally to the right up (col increases and row decreases)
                row = state[i] - 1
                col = i + 1;
                while (col < N and row >= 0 and board[row][col] != 1) :
                        col += 1;
                        row -= 1;
                if (col < N and row >= 0 and board[row][col] == 1) :
                        attacking += 1;

# Return pairs.
        return int(attacking / 2);

# A utility function that generates a board configuration given the state.
def generateBoard( board, state):
        fill(board, 0);
        for i in range(N):
                board[state[i]][i] = 1;
# A utility function that copies contents of state2 to state1.
def copyState( state1, state2):
        for i in range(N):
                state1[i] = state2[i];
```

```
# This function gets the neighbour of the current state having the least objective value amongst all neighbours as well as the current state.
def getNeighbour(board, state):

# Declaring and initializing the optimal board and state with the current board and the state as the starting point.
        opBoard = [[0 for _ in range(N)] for _ in range(N)]
        opState = [0 for _ in range(N)]
        copyState(opState, state);
        generateBoard(opBoard, opState);

# Initializing the optimal objective value
        opObjective = calculateObjective(opBoard, opState);

# Declaring and initializing the temporary board and state for the purpose of computation.
        NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]

        NeighbourState = [0 for _ in range(N)]
        copyState(NeighbourState, state);
        generateBoard(NeighbourBoard, NeighbourState);

# Iterating through all possible neighbours of the board.
        for i in range(N):
                for j in range(N):

# Condition for skipping the current state
                        if (j != state[i]) :

# Initializing temporary neighbour with the current neighbour.
                                NeighbourState[i] = j;
```

```
                NeighbourBoard[NeighbourState[i]][i] = 1;
                NeighbourBoard[state[i]][i] = 0;

# Calculating the objective value of the neighbour.
                temp = calculateObjective( NeighbourBoard, NeighbourState);
# Comparing temporary and optimal neighbour objectives and if temporary is less than optimal then updating accordingly.

                if (temp <= opObjective) :
                    opObjective = temp;
                    copyState(opState, NeighbourState);
                    generateBoard(opBoard, opState);
# Going back to the original configuration for the next iteration.
                NeighbourBoard[NeighbourState[i]][i] = 0;
                NeighbourState[i] = state[i];
                NeighbourBoard[state[i]][i] = 1;
# Copying the optimal board and state thus found to the current board and, state since c+= 1 doesn't allow returning multiple values.
        copyState(state, opState);
        fill(board, 0);
        generateBoard(board, state);
```

PROGRAM-6

Implementation of Hill-climbing to solve 8-puzzle problem

```
def hillClimbing(board, state):
# Declaring and initializing the neighbour board and state with the current board and the state as the starting point.
        neighbourBoard = [[0 for _ in range(N)] for _ in range(N)
        neighbourState = [0 for _ in range(N)]
        copyState(neighbourState, state);
        generateBoard(neighbourBoard, neighbourState);
        while True:
# Copying the neighbour board and state to the current board and state, since a neighbour becomes current after the jump.
                copyState(state, neighbourState);
                generateBoard(board, state);
# Getting the optimal neighbour
                getNeighbour(neighbourBoard, neighbourState);
                if (compareStates(state, neighbourState)) :
# If neighbour and current are equal then no optimal neighbour exists and therefore output the result and break the loop.
                        prvarBoard(board);
                        break;
                elif (calculateObjective(board, state) == calculateObjective( neighbourBoard,neighbourState)):
```

```
# If neighbour and current arenot equal but their objectivesare equal then we are eitherapproaching a shoulder or alocal optimum, in any case,jump to a random neighbourto escape it.

# Random neighbour

                    neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;

                    generateBoard(neighbourBoard, neighbourState);



# Driver code

state = [0] * N

board = [[0 for _ in range(N)] for _ in range(N)]


# Getting a starting point by randomly configuring the board

configureRandomly(board, state);

# Do hill climbing on the board obtained

hillClimbing(board, state);
```

Output:

```
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 1 0
 0 0 0 0 1 0 0 0
 0 1 0 0 0 0 0 0
```

PROGRAM-1

LABEL ENCODING

In machine learning, we usually deal with datasets that contain multiple labels in one or more than one columns. These labels can be in the form of words or numbers. To make the data understandable or in human-readable form, the training data is often labelled in words.

Label Encoding refers to converting the labels into a numeric form so as to convert them into the machine-readable form. Machine learning algorithms can then decide in a better way how those labels must be operated. It is an important pre-processing step for the structured dataset in supervised learning.

Example:

Suppose we have a column *Height* in some dataset.

Height
Tall
Medium
Short

After applying label encoding, the Height column is converted into:

Height
0
1
2

where 0 is the label for tall, 1 is the label for medium, and 2 is a label for short height.

We apply *Label Encoding* on iris dataset on the target column which is Species. It contains three species *Iris-setosa, Iris-versicolor, Iris-virginica*.

```
# Import libraries

import numpy as np

import pandas as pd
```

```
# Import dataset
df = pd.read_csv('../../data/Iris.csv')

df['species'].unique()
```

Output:
array(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], dtype=object)

After applying Label Encoding

```
# Import label encoder
from sklearn import preprocessing

# label_encoder object knows how to understand word labels.
label_encoder = preprocessing.LabelEncoder()

# Encode labels in column 'species'.
df['species']= label_encoder.fit_transform(df['species'])

df['species'].unique()
```

Output:
array([0, 1, 2], dtype=int64)

Limitation of label Encoding:

Label encoding converts the data in machine-readable form, but it assigns a unique number (starting from 0) to each class of data. This may lead to the generation of priority issues in the training of data sets. A label with a high value may be considered to have high priority than a label having a lower value.

PROGRAM-2

Implementing Simple Linear Regression

Simple linear regression is an approach for predicting a **response** using a **single feature**. It is assumed that the two variables are linearly related. Hence, we try to find a linear function that predicts the response value(y) as accurately as possible as a function of the feature

Let us consider a dataset where we have a value of response y for every feature x:

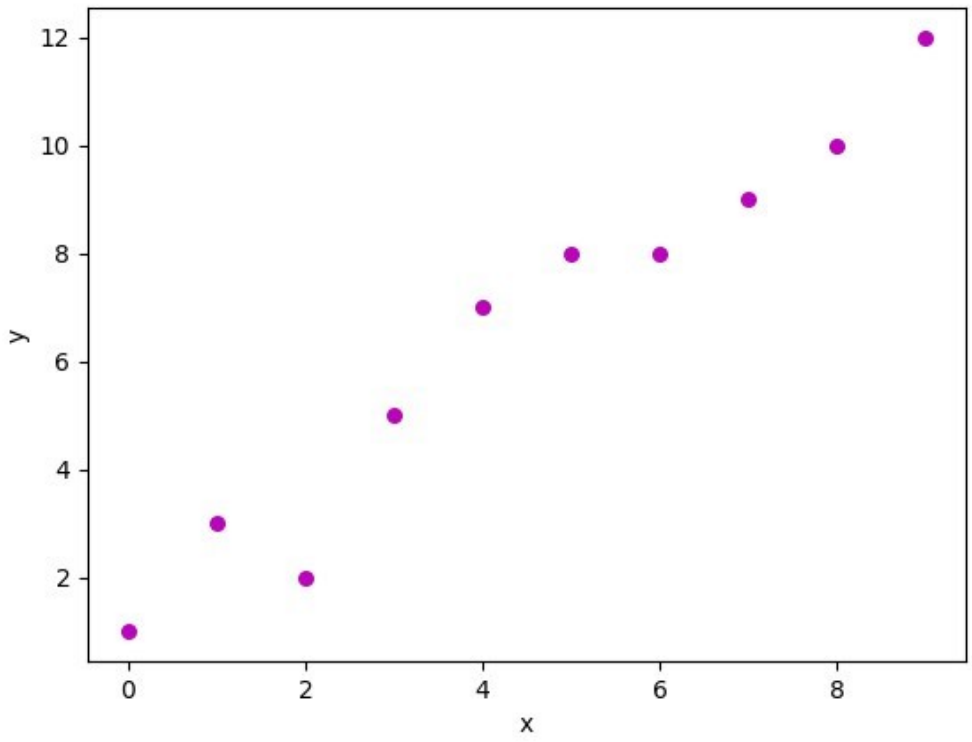
x	0	1	2	3	4	5	6	7	8	9
y	1	3	2	5	7	8	8	9	10	12

For generality, we define:

x as **feature vector**, i.e x = [x_1, x_2, …., x_n],

y as **response vector**, i.e y = [y_1, y_2, …., y_n]

for **n** observations (in above example, n=10). A scatter plot of the above dataset looks like:-



Now, the task is to find a **line that fits best** in the above scatter plot so that we can predict the response for any new feature values. (i.e a value of x not present in a dataset)

Thisline is called a **regression line**.

The equation of regression line is represented as:

$H(x_i) = B_0 + B_1 x_i$

Here,

- h(x_i) represents the **predicted response value** for i^{th} observation.
- b_0 and b_1 are regression coefficients and represent **y-intercept** and **slope** of regression line respectively.

To create our model, we must "learn" or estimate the values of regression coefficients b_0 and b_1. And once we've estimated these coefficients, we can use the model to predict responses!

In this article, we are going to use the principle of **Least Squares**.

Now consider:

$Y_i = B_0 + B_1 x_i + E_i$

$E_i = Y_i - H(x_i)$

PROGRAM-

Python implementation of above technique on our small dataset

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

def estimate_coef(x, y):
# number of observations/points
        n = np.size(x)

# mean of x and y vector
        m_x = np.mean(x)
        m_y = np.mean(y)

# calculating cross-deviation and deviation about x
        SS_xy = np.sum(y*x) - n*m_y*m_x
```

```
        SS_xx = np.sum(x*x) - n*m_x*m_x

# calculating regression coefficients
        b_1 = SS_xy / SS_xx
        b_0 = m_y - b_1*m_x

        return (b_0, b_1)

def plot_regression_line(x, y, b):
# plotting the actual points as scatter plot
        plt.scatter(x, y, color = "m",
                        marker = "o", s = 30)

# predicted response vector
        y_pred = b[0] + b[1]*x

# plotting the regression line
        plt.plot(x, y_pred, color = "g")

# putting labels
        plt.xlabel('x')
        plt.ylabel('y')

# function to show plot
        plt.show()

def main():
# observations / data
        x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
        y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

# estimating coefficients
        b = estimate_coef(x, y)
        print("Estimated coefficients:\nb_0 = {} \
```

```
                \nb_1 = {}".format(b[0], b[1]))

# plotting regression line
        plot_regression_line(x, y, b)

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

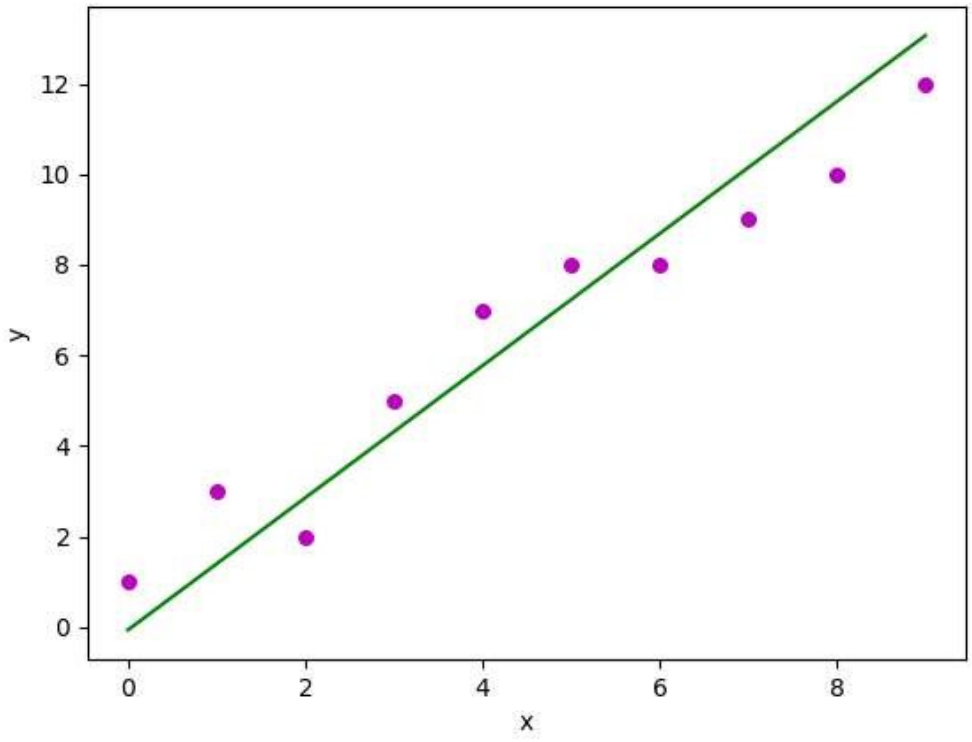
OUTPUT:

Estimated coefficients:

b_0 = -0.0586206896552

b_1 = 1.45747126437

And graph obtained looks like this:

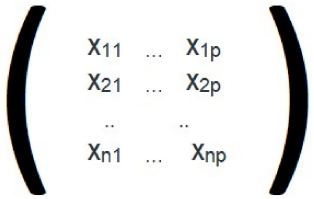


Multiple linear regression

Multiple linear regression attempts to model the relationship between **two or morefeatures** and a response by fitting a linear equation to the observed data.Clearly, it is nothing but an extension of simple linear regression.Consider a dataset with **p** features(or independent variables) and one response(or dependent variable). Also, the datasetcontains **n** rows/observations.

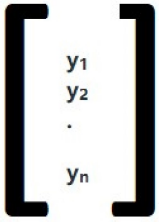
We define:X (**feature matrix**) = a matrix of size **n X p** where x_{ij} denotes the values of j_{th}-feature--for--i_{th}-observation.

So,



and

y (**response vector**) = a vector of size **n** where y_{i} denotes the value of response for i_{th} observation.



The **regression line** for **p** features is represented as:

$H(xi) = B_0+B_1x_{i1}+\ldots\ldots B_px_{ip}$

Program-3

Python implementation of multiple linear regression techniques on the Boston house pricing dataset using Scikit-learn.

```
import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets, linear_model, metrics

# load the boston dataset

boston = datasets.load_boston(return_X_y=False)

# defining feature matrix(X) and response vector(y)

X = boston.data

y = boston.target

# splitting X and y into training and testing sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,


        random_state=1)

# create linear regression object

reg = linear_model.LinearRegression()

# train the model using the training sets

reg.fit(X_train, y_train)

# regression coefficients
```

```
print('Coefficients: ', reg.coef_)

# variance score: 1 means perfect prediction

print('Variance score: {}'.format(reg.score(X_test, y_test)))

# plot for residual error

## setting plot style

plt.style.use('fivethirtyeight')

## plotting residual errors in training data

plt.scatter(reg.predict(X_train), reg.predict(X_train) - y_train,

                        color = "green", s = 10, label = 'Train data')

## plotting residual errors in test data

plt.scatter(reg.predict(X_test), reg.predict(X_test) - y_test,

                        color = "blue", s = 10, label = 'Test data')

## plotting line for zero residual error

plt.hlines(y = 0, xmin = 0, xmax = 50, linewidth = 2)

## plotting legend

plt.legend(loc = 'upper right')

## plot title

plt.title("Residual errors")

## method call for showing the plot

plt.show()
```

OUTPUT:

Coefficients:

[-8.80740828e-02 6.72507352e-02 5.10280463e-02 2.18879172e+00

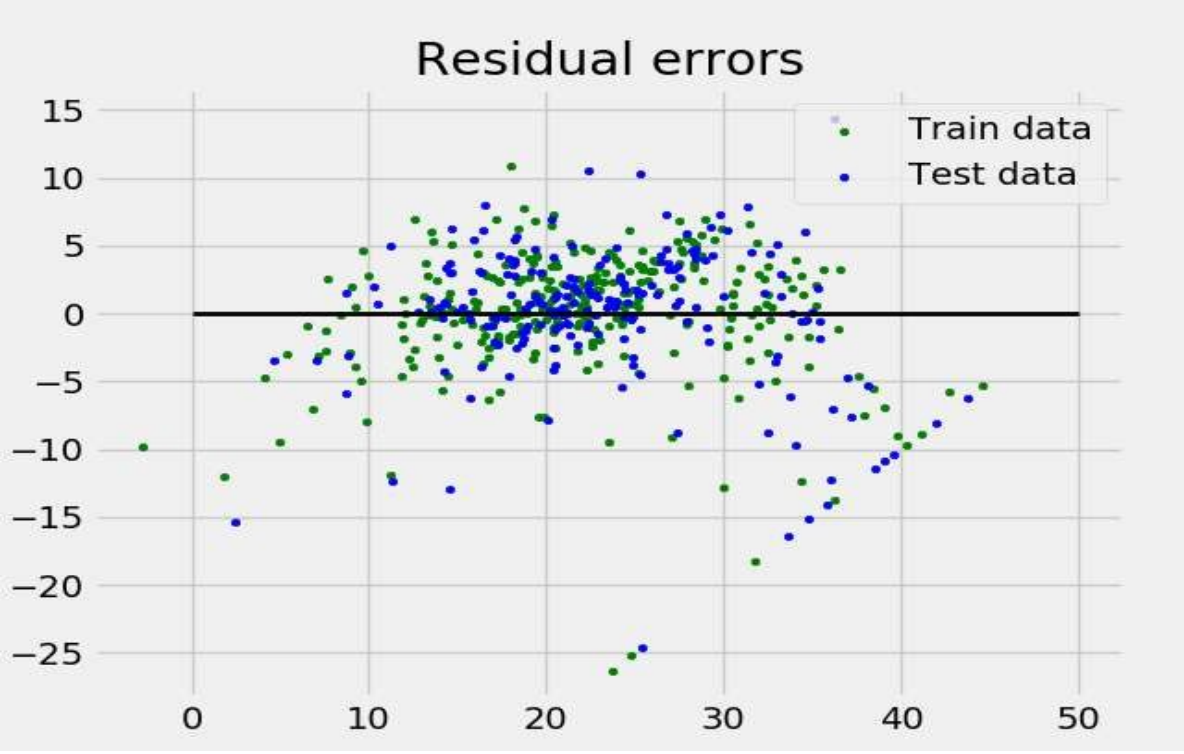
-1.72283734e+01 3.62985243e+00 2.13933641e-03 -1.36531300e+00

2.88788067e-01 -1.22618657e-02 -8.36014969e-01 9.53058061e-03

-5.05036163e-01]

Variance score: 0.720898784611

Residual Error plot :



PROGRAM-4

K-nearest neighbor algorithm

This algorithm is used to solve the classification model problems. K-nearest neighbor or K-NN algorithm basically creates an imaginary boundary to classify the data. When new data points come in, the algorithm will try to predict that to the nearest of the boundary line.

Therefore, larger k value means smother curves of separation resulting in less complexmodels. Whereas, smaller k value tends to overfit the data and resulting in complex models.

Note: It's very important to have the right k-value when analysing the dataset to avoid overfitting and underfitting of the dataset.Using the k-nearest neighbor algorithm we fit the historical data (or train the model) and predict the future.

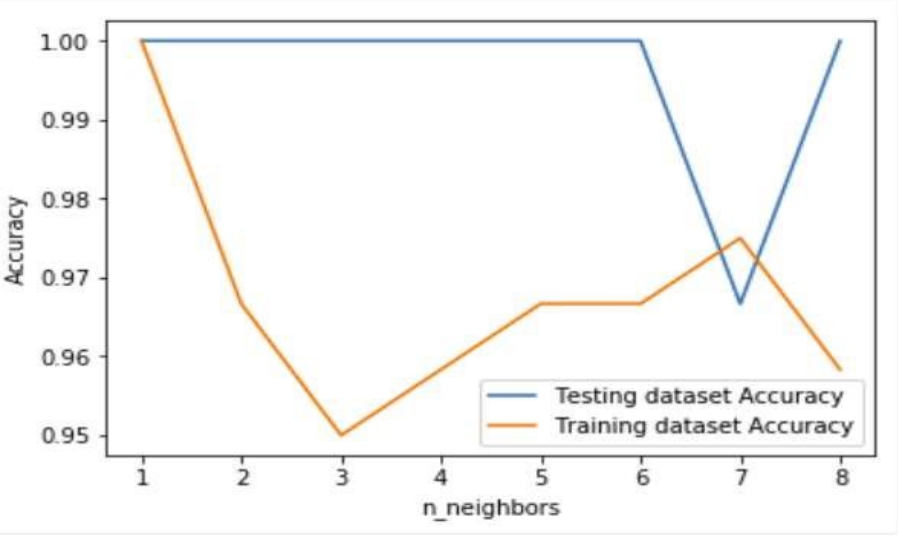
PROGRAM

```
# Import necessary modules

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import train_test_split

from sklearn.datasets import load_iris

import numpy as np

import matplotlib.pyplot as plt

irisData = load_iris()

# Create feature and target arrays

X = irisData.data

y = irisData.target

# Split into training and test set
```

```
X_train, X_test, y_train, y_test = train_test_split(
                                X, y, test_size = 0.2, random_state=42)
neighbors = np.arange(1, 9)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

# Loop over K values
for i, k in enumerate(neighbors):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)

# Compute training and test data accuracy
        train_accuracy[i] = knn.score(X_train, y_train)
        test_accuracy[i] = knn.score(X_test, y_test)

# Generate plot
plt.plot(neighbors, test_accuracy, label = 'Testing dataset Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training dataset Accuracy')
plt.legend()
plt.xlabel('n_neighbors')
```

```
plt.ylabel('Accuracy')

plt.show()
```

Output:



PROGRAM-5

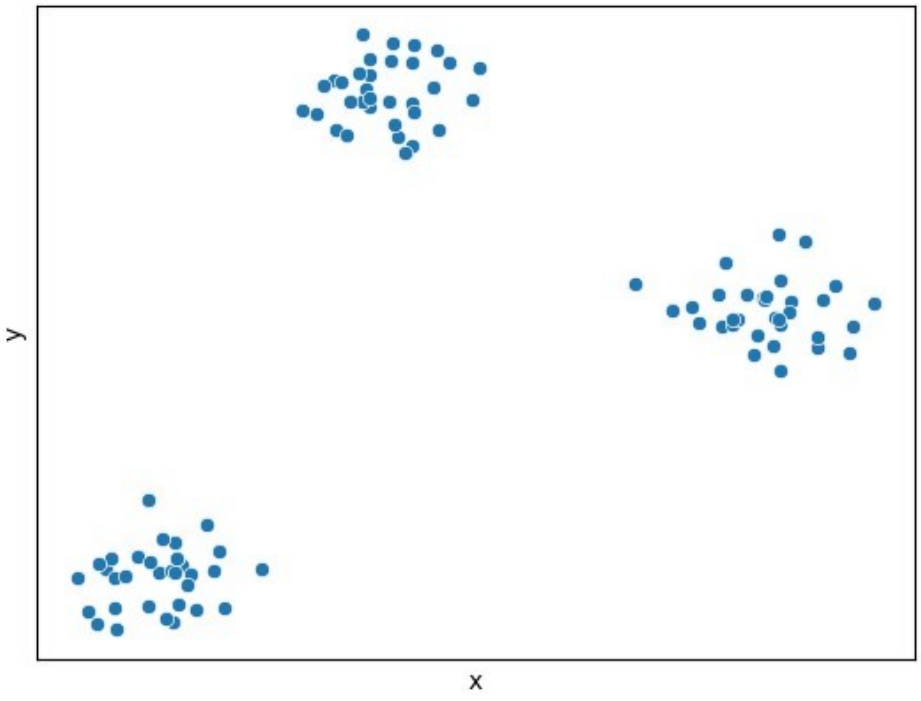
Implementing k-means clustering

k-means clustering is an **unsupervised** machine learning algorithm that seeks to segment a dataset into groups based on the similarity of datapoints. An unsupervised model has **independent variables** and no **dependent variables**.

Suppose you have a dataset of 2-dimensional scalar attributes:

$$[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$$

If the points in this dataset belong to distinct groups with attributes significantly varying between groups but not within, the points should form clusters when plotted.



Algorithm

For a given dataset, k is specified to be the number of distinct groups the points belong to. These k centroids are first randomly initialized, then iterations are performed to optimize the locations of these k centroids as follows:

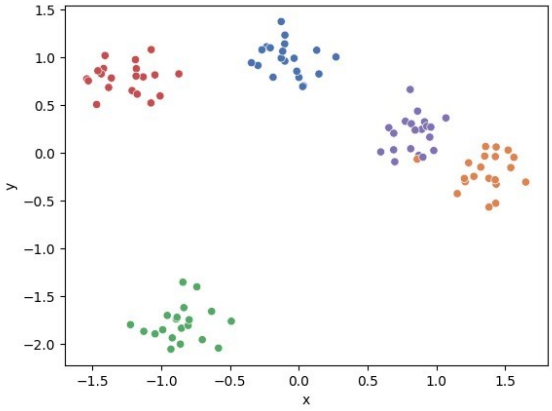
1. The distance from each point to each centroid is calculated.
2. Points are assigned to their nearest centroid.
3. Centroids are shifted to be the average value of the points belonging to it. If the centroids did not move, the algorithm is finished, else repeat.

Data

To evaluate our algorithm, we'll first generate a dataset of groups in 2-dimensional space. The sklearn.datasets function make_blobs creates groupings of 2-dimensional normal distributions, and assigns a label corresponding to the group said point belongs to.

```
import seaborn as sns
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScalercenters = 5
X_train, true_labels = make_blobs(n_samples=100, centers=centers, random_state=42)
X_train = StandardScaler().fit_transform(X_train)sns.scatterplot(x=[X[0] for X in X_train],
            y=[X[1] for X in X_train],
            hue=true_labels,
            palette="deep",
            legend=None
            )plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

output



Helper Functions

We'll need to calculate the distances between a point and a dataset of points multiple times in this algorithm. To do so, lets define a function that calculates Euclidean distances.

```
def euclidean(point, data):
    """
    Euclidean distance between point & data.
    Point has dimensions (m,), data has dimensions (n,m), and output will be of size (n,).
    """
    return np.sqrt(np.sum((point - data)**2, axis=1))
```

Implementation

First, the k-means clustering algorithm is initialized with a value for k and a maximum number of iterations for finding the optimal centroid locations. If a maximum number of iterations is not considered when optimizing centroid locations, there is a risk of running an infinite loop.

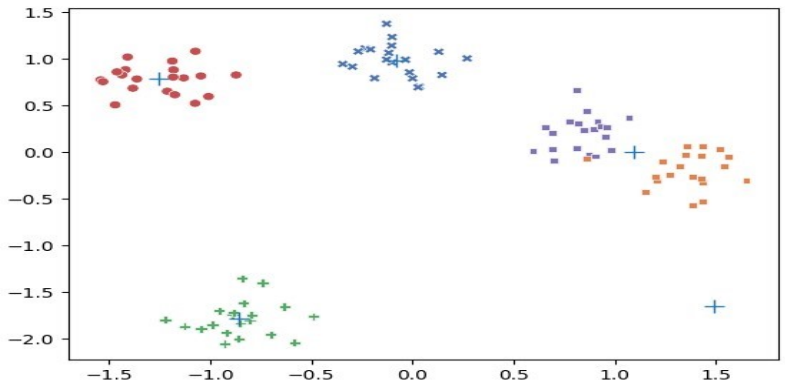
```
class KMeans:    def __init__(self, n_clusters=8, max_iter=300):
self.n_clusters = n_clusters
self.max_iter = max_iter
```

Now, the bulk of the algorithm is performed when fitting the model to a training dataset.First we'll initialize the centroids randomly in the domain of the test dataset, with a uniform distribution.

Now we can finally deploy our model. Lets train and test it on our original dataset and see the results. We'll keep our original method of plotting our data, by separating the true labels by color, but now we'll additionally separate the predicted labels by marker style, to see how the model performs.

```
kmeans = KMeans(n_clusters=centers)
kmeans.fit(X_train)# View results
class_centers, classification = kmeans.evaluate(X_train)
sns.scatterplot(x=[X[0] for X in X_train],
                y=[X[1] for X in X_train],
                hue=true_labels,
                style=classification,
                palette="deep",
                legend=None
                )
plt.plot([x for x, _ in kmeans.centroids],
        [y for _, y in kmeans.centroids],
        '+',
markersize=10,
        )plt.show()
```

OUTPUT:



PROGRAM-6

Implementing Hierarchical clustering

In data mining and statistics, hierarchical clustering analysis is a method of cluster analysis that seeks to build a hierarchy of clusters i.e. tree-type structure based on the hierarchy.

Basically, there are two types of hierarchical cluster analysis strategies –

1. Agglomerative Clustering: Also known as bottom-up approach or hierarchical agglomerative clustering (HAC). A structure that is more informative than the unstructured set of clusters returned by flat clustering. This clustering algorithm does not require us to prespecify the number of clusters. Bottom-up algorithms treat each data as a singleton cluster at the outset and then successively agglomerates pairs of clusters until all clusters have been merged into a single cluster that contains all data.

Algorithm:

given a dataset (d_1, d_2, d_3, ...d_N) of size N

compute the distance matrix

for i=1 to N:

  # as the distance matrix is symmetric about

  # the primary diagonal so we compute only lower

  # part of the primary diagonal

  for j=1 to i:

dis_mat[i][j] = distance[d_i, d_j]

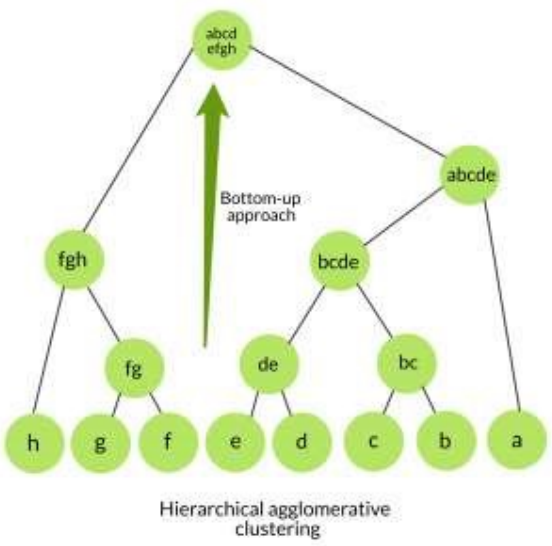
each data point is a singleton cluster

repeat

  merge the two cluster having minimum distance

  update the distance matrix

until only a single cluster remains



Hierarchical agglomerative clustering

Python implementation of the above algorithm using the scikit-learn library:

```
from sklearn.cluster import AgglomerativeClustering
import numpy as np


# randomly chosen dataset
X = np.array([[1, 2], [1, 4], [1, 0],
                        [4, 2], [4, 4], [4, 0]])


# here we need to mention the number of clusters
# otherwise the result will be a single cluster
# containing all the data
clustering = AgglomerativeClustering(n_clusters = 2).fit(X)


# print the class labels
print(clustering.labels_)
```

OUTPUT:

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

Computing Distance Matrix: While merging two clusters we check the distance between two every pair of clusters and merge the pair with least distance/most similarity. But the question is how is that distance determined. There are different ways of defining Inter Cluster distance/similarity. Some of them are:

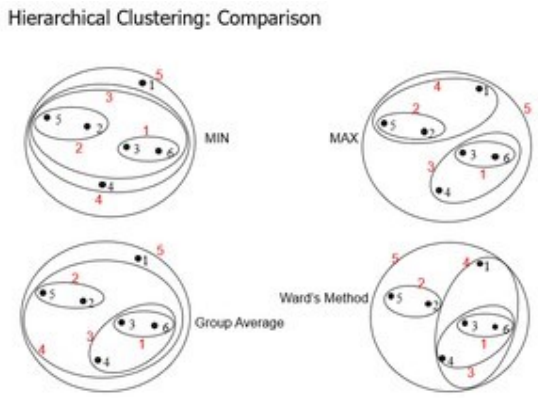
1. Min Distance: Find minimum distance between any two points of the cluster.

2. Max Distance: Find maximum distance between any two points of the cluster.

3. Group Average: Find average of distance between every two points of the clusters.

4. Ward's Method: Similarity of two clusters is based on the increase in squared error when two clusters are merged.

For example, if we group a given data using different method, we may get different results:



2. Divisive clustering: Also known as a top-down approach. This algorithm also does not require to prespecify the number of clusters. Top-down clustering requires a method for splitting a cluster that contains the whole data and proceeds by splitting clusters recursively until individual data have been split into singleton clusters.

Algorithm :

given a dataset (d_1, d_2, d_3, ...d_N) of size N

at the top we have all data in one cluster

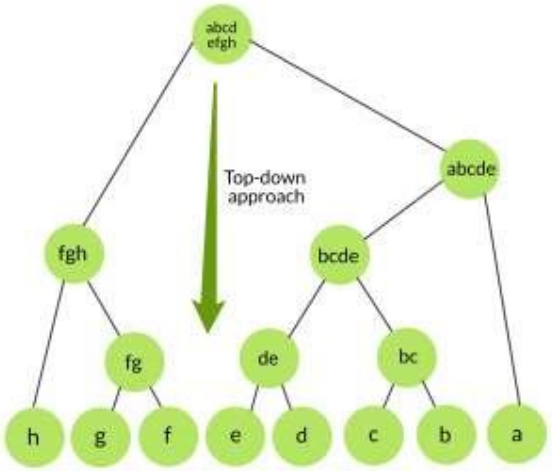
the cluster is split using a flat clustering method eg. K-Means etc

repeat

choose the best cluster among all the clusters to split

split that cluster by the flat clustering algorithm

until each data is in its own singleton cluster



Hierarchical divisive clustering