Practical-1

Aim: Implement Breadth first search or Depth first search.

Solution:

1. Breadth first search:

Code:

```
# Python3 Program to print BFS traversal from a given source vertex. BFS (int s) traverses vertices
reachable from s.
from collections import defaultdict
# This class represents a directed graph using adjacency list representation
class Graph:
    # Constructor
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)
    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
    # Function to print a BFS of graph
    def BFS(self, s):
        # Mark all the vertices as not visited
        visited = [False] * (len(self.graph))
        # Create a queue for BFS
        queue = []
        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True
        while queue:
            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
```

```
            print (s, end = " ")
            # Get all adjacent vertices of the dequeued vertex s. If a adjacent has not been
visited, then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
# print(g)
print ("Following is Breadth First Traversal"
                " (starting from vertex 2)")
g.BFS(1)
```

Output:

```
Following is Breadth First Traversal (starting from vertex 2)
1 2 0 3
```

2. Depth first search:

Code:

```
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set() # Set to keep track of visited nodes of graph.
def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output:

```
Following is the Depth-First Search
5
3
2
4
8
7
```

Practical-2

Aim: Implement solution of Water Jug problem or 8-puzzle problem using Best First Search or A*.

Solution:

1. Water Jug problem:

Code:

```
# This function is used to initialize the
# dictionary elements with a default value.
from collections import defaultdict
# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
jug1, jug2, aim = 4, 3, 2
# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)
# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):
	# Checks for our goal and
	# returns 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 the
```

```
    # combination or not. If not, then it proceeds further.
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        # Changes the boolean value of
        # the combination as it is visited.
        visited[(amt1, amt2)] = True
        # Check for all the 6 possibilities and
        # see 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 otherwise
    # recursion 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
True
```

2. 8-puzzle problem using Best First Search:

Code:

```
# Python code to display the way from the root
# node to the final destination node for N*N-1 puzzle
# algorithm by the help of Branch and Bound technique
# The answer assumes that the instance of the
# puzzle can be solved
# Importing the 'copy' for deepcopy method
import copy
# Importing the heap methods from the python
# library for the Priority Queue
from heapq import heappush, heappop
# This particular var can be changed to transform
# the program from 8 puzzle(n=3) into 15
# puzzle(n=4) and so on ...
n = 3
# bottom, left, top, right
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]
# creating a class for the Priority Queue
```

```
class priorityQueue:
    # Constructor for initializing a
    # Priority Queue
    def __init__(self):
        self.heap = []
    # Inserting a new key 'key'
    def push(self, key):
        heappush(self.heap, key)
    # funct to remove the element that is minimum,
    # from the Priority Queue
    def pop(self):
        return heappop(self.heap)
    # funct to check if the Queue is empty or not
    def empty(self):
        if not self.heap:
            return True
        else:
            return False
# structure of the node
class nodes:
    def __init__(self, parent, mats, empty_tile_posi,
                costs, levels):
        # This will store the parent node to the
        # current node And helps in tracing the
        # path when the solution is visible
        self.parent = parent
        # Useful for Storing the matrix
        self.mats = mats
        # useful for Storing the position where the
        # empty space tile is already existing in the matrix
        self.empty_tile_posi = empty_tile_posi
```

```
         # Store no. of misplaced tiles
        self.costs = costs
        # Store no. of moves so far
        self.levels = levels
    # This func is used in order to form the
    # priority queue based on
    # the costs var of objects
    def __lt__(self, nxt):
        return self.costs < nxt.costs
# method to calc. the no. of
# misplaced tiles, that is the no. of non-blank
# tiles not in their final posi
def calculateCosts(mats, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and
                (mats[i][j] != final[i][j])):
                count += 1
    return count
def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
            levels, parent, final) -> nodes:
    # Copying data from the parent matrixes to the present matrixes
    new_mats = copy.deepcopy(mats)
    # Moving the tile by 1 position
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]
```

```
    # Setting the no. of misplaced tiles
    costs = calculateCosts(new_mats, final)
    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                costs, levels)
    return new_nodes
# func to print the N by N matrix
def printMatsrix(mats):
    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")
        print()
# func to know if (x, y) is a valid or invalid
# matrix coordinates
def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < n
# Printing the path from the root node to the final node
def printPath(root):
    if root == None:
        return
    printPath(root.parent)
    printMatsrix(root.mats)
    print()
# method for solving N*N - 1 puzzle algo
# by utilizing the Branch and Bound technique. empty_tile_posi is
# the blank tile position initially.
def solve(initial, empty_tile_posi, final):
    # Creating a priority queue for storing the live
    # nodes of the search tree
    pq = priorityQueue()
    # Creating the root node
    costs = calculateCosts(initial, final)
```

```
root = nodes(None, initial,
         empty_tile_posi, costs, 0)
# Adding root to the list of live nodes
pq.push(root)
# Discovering a live node with min. costs,
# and adding its children to the list of live
# nodes and finally deleting it from
# the list.
while not pq.empty():
  # Finding a live node with min. estimatsed
  # costs and deleting it form the list of the
  # live nodes
  minimum = pq.pop()
  # If the min. is ans node
  if minimum.costs == 0:
    # Printing the path from the root to
    # destination;
    printPath(minimum)
    return
  # Generating all feasible children
  for i in range(n):
    new_tile_posi = [
      minimum.empty_tile_posi[0] + rows[i],
      minimum.empty_tile_posi[1] + cols[i], ]
    if isSafe(new_tile_posi[0], new_tile_posi[1]):
      # Creating a child node
      child = newNodes(minimum.mats,
              minimum.empty_tile_posi,
              new_tile_posi,
              minimum.levels + 1,
              minimum, final,)
```

```
                # Adding the child to the list of live nodes
                pq.push(child)
# Main Code
# Initial configuration
# Value 0 is taken here as an empty space
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]
# Final configuration that can be solved
# Value 0 is taken as an empty space
final = [ [ 1, 2, 3 ],
        [ 5, 8, 6 ],
        [ 0, 7, 4 ] ]

# Blank tile coordinates in the
# initial configuration
empty_tile_posi = [ 1, 2 ]
# Method call for solving the puzzle
solve(initial, empty_tile_posi, 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
```

3. 8-puzzle problem using A*:

Code:

```
from copy import deepcopy
import numpy as np
import time

# takes the input of current states and evaluvates the best path to goal state
def bestsolution(state):
    bestsol = np.array([], int).reshape(-1, 9)
    count = len(state) - 1
    while count != -1:
        bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)
        count = (state[count]['parent'])
    return bestsol.reshape(-1, 3, 3)
```

```
# this function checks for the uniqueness of the iteration(it) state, weather it has been previously
    traversed or not.
def all(checkarray):
    set=[]
    for it in set:
        for checkarray in it:
            return 1
        else:
            return 0


# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state
def manhattan(puzzle, goal):
    a = abs(puzzle // 3 - goal // 3)
    b = abs(puzzle % 3 - goal % 3)
    mhcost = a + b
    return sum(mhcost[1:])


# will calcuates the number of misplaced tiles in the current state as compared to the goal state
def misplaced_tiles(puzzle,goal):
    mscost = np.sum(puzzle != goal) - 1
    return mscost if mscost > 0 else 0


#3[on_true] if [expression] else [on_false]
```

```
# will indentify the coordinates of each of goal or initial state values
def coordinates(puzzle):
    pos = np.array(range(9))
    for p, q in enumerate(puzzle):
        pos[q] = p
    return pos



# start of 8 puzzle evaluvation, using Manhattan heuristics
def evaluvate(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],
     1)],
                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

     # initializing the parent, gn and hn, where hn is manhattan distance function call
    costg = coordinates(goal)
    parent = -1
    gn = 0
    hn = manhattan(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]
    priority = np.array( [(0, hn)], dtpriority)
```

```
while 1:
    priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
    position, fn = priority[0]
    priority = np.delete(priority, 0, 0)
    # sort priority queue using merge sort,the first element is picked for exploring remove from
queue what we are exploring
    puzzle, parent, gn, hn = state[position]
    puzzle = np.array(puzzle)
    # Identify the blank square in input
    blank = int(np.where(puzzle == 0)[0])
    gn = gn + 1
    c = 1
    start_time = time.time()
    for s in steps:
        c = c + 1
        if blank not in s['position']:
            # generate new state as copy of current
            openstates = deepcopy(puzzle)
            openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
openstates[blank]
            # The all function is called, if the node has been previously explored or not
            if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                end_time = time.time()
                if (( end_time - start_time ) > 2):
                    print(" The 8 puzzle is unsolvable ! \n")
                    exit
                # calls the manhattan function to calcuate the cost
                hn = manhattan(coordinates(openstates), costg)
                 # generate and add new state in the list
                q = np.array([(openstates, position, gn, hn)], dtstate)
```

```
            state = np.append(state, q, 0)
            # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal
state
            fn = gn + hn

            q = np.array([(len(state) - 1, fn)], dtpriority)
            priority = np.append(priority, q, 0)
             # Checking if the node in openstates are matching the goal state.
            if np.array_equal(openstates, goal):
                print(' The 8 puzzle is solvable ! \n')
                return state, len(priority)


    return state, len(priority)


# start of 8 puzzle evaluvation, using Misplaced tiles heuristics
def evaluvate_misplaced(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],
     1)],
                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

    costg = coordinates(goal)
    # initializing the parent, gn and hn, where hn is misplaced_tiles  function call
    parent = -1
    gn = 0
    hn = misplaced_tiles(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)
```

```
# We make use of priority queues with position as keys and fn as value.
dtpriority = [('position', int),('fn', int)]

priority = np.array([(0, hn)], dtpriority)

while 1:
    priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
    position, fn = priority[0]
    # sort priority queue using merge sort,the first element is picked for exploring.
    priority = np.delete(priority, 0, 0)
    puzzle, parent, gn, hn = state[position]
    puzzle = np.array(puzzle)
     # Identify the blank square in input
    blank = int(np.where(puzzle == 0)[0])
    # Increase cost g(n) by 1
    gn = gn + 1
    c = 1
    start_time = time.time()
    for s in steps:
        c = c + 1
        if blank not in s['position']:
             # generate new state as copy of current
            openstates = deepcopy(puzzle)
            openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
 openstates[blank]
            # The check function is called, if the node has been previously explored or not.
            if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                end_time = time.time()
                if (( end_time - start_time ) > 2):
                    print(" The 8 puzzle is unsolvable \n")
                    break
```

```
            # calls the Misplaced_tiles function to calcuate the cost
            hn = misplaced_tiles(coordinates(openstates), costg)
            # generate and add new state in the list
            q = np.array([(openstates, position, gn, hn)], dtstate)
            state = np.append(state, q, 0)
            # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal
    state
            fn = gn + hn

            q = np.array([(len(state) - 1, fn)], dtpriority)
            priority = np.append(priority, q, 0)
            # Checking if the node in openstates are matching the goal state.
            if np.array_equal(openstates, goal):
                print(' The 8 puzzle is solvable \n')
                return state, len(priority)

    return state, len(priority)



# ----------  Program start -----------------


 # User input for initial state
puzzle = []
print(" Input vals from 0-8 for start state ")
for i in range(0,9):
    x = int(input("enter vals :"))
    puzzle.append(x)

 # User input of goal state
```

```
goal = []
print(" Input vals from 0-8 for goal state ")
for i in range(0,9):
    x = int(input("Enter vals :"))
    goal.append(x)



n = int(input("1. Manhattan distance \n2. Misplaced tiles"))

if(n ==1 ):
    state, visited = evaluvate(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))

if(n == 2):
    state, visited = evaluvate_misplaced(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
        print('Total generated:', len(state))
```

Output:

```
 Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :3
enter vals :0
enter vals :4
enter vals :6
enter vals :7
enter vals :5
enter vals :8
 Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :4
Enter vals :5
Enter vals :6
Enter vals :7
Enter vals :8
Enter vals :0
1. Manhattan distance
2. Misplaced tiles2
 The 8 puzzle is solvable

   1 2 3
   0 4 6
   7 5 8

   1 2 3
   4 0 6
   7 5 8

   1 2 3
   4 5 6
   7 0 8

   1 2 3
   4 5 6
   7 8 0
Steps to reach goal: 3
Total nodes visited:  3

Total generated: 9
```

1. Manhattan Distance

```
 Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :3
enter vals :0
enter vals :4
enter vals :6
enter vals :7
enter vals :5
enter vals :8
 Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :4
Enter vals :5
Enter vals :6
Enter vals :7
Enter vals :8
Enter vals :0
1. Manhattan distance
2. Misplaced tiles1
 The 8 puzzle is solvable !

   1 2 3
   0 4 6
   7 5 8

   1 2 3
   4 0 6
   7 5 8

   1 2 3
   4 5 6
   7 0 8

   1 2 3
   4 5 6
   7 8 0
Steps to reach goal: 3
Total nodes visited:  3

Total generated: 9
```

2. Misplaced Tiles

Practical-3

Aim: Write a program to solve a given cryptarithmetic problem.

Solution:

Code:

```
# importing the necessary libraries
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod

# declaring a type variable V as variable type and D as domain type
V = TypeVar('V') # variable type
D = TypeVar('D') # domain type

# this is a Base class for all constraints
class Constraint(Generic[V, D], ABC):
    # the variables that the constraint is between
    def __init__(self, variables: List[V]) -> None:
        self.variables = variables

    # this is an abstract method which must be overridden by subclasses
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        ...

# A constraint satisfaction problem consists of variables of type V
# that have ranges of values known as domains of type D and constraints
# that determine whether a particular variable's domain selection is valid
class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
        # variables to be constrained
        # assigning variables parameter to self.variables
```

```
        self.variables: List[V] = variables
        # domain of each variable
        # assigning domains parameter to self.domains
        self.domains: Dict[V, List[D]] = domains
        # assigning an empty dictionary to self.constraints
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        # iterating over self.variables
        for variable in self.variables:
            self.constraints[variable] = []
            # if the variable is not in domains, then raise a LookupError("Every variable should have a
domain assigned to it.")
            if variable not in self.domains:
                raise LookupError("Every variable should have a domain assigned to it.")
    # this method adds constraint to variables as per their domains
    def add_constraint(self, constraint: Constraint[V, D]) -> None:
        for variable in constraint.variables:
            if variable not in self.variables:
                raise LookupError("Variable in constraint not in CSP")
            else:
                self.constraints[variable].append(constraint)

    # checking if the value assignment is consistent by checking all constraints
    # for the given variable against it
    def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
        # iterating over self.constraints[variable]
        for constraint in self.constraints[variable]:
            # if constraint not satisfied then returning False
            if not constraint.satisfied(assignment):
                return False
        # otherwise returning True
        return True
```

```
    # this method is performing the backtracking search to find the result
    def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:
        # assignment is complete if every variable is assigned (our base case)
        if len(assignment) == len(self.variables):
            return assignment

        # get all variables in the CSP but not in the assignment
        unassigned: List[V] = [v for v in self.variables if v not in assignment]

        # get the every possible domain value of the first unassigned variable
        first: V = unassigned[0]
        # iterating over self.domains[first]
        for value in self.domains[first]:
            local_assignment = assignment.copy()
            # assign the value
            local_assignment[first] = value
            # if we're still consistent, we recurse (continue)
            if self.consistent(first, local_assignment):
                # recursively calling the self.backtracking_search method based on the local_assignment
                result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)
                # if we didn't find the result, we will end up backtracking
                if result is not None:
                    return result
        return None

# SendMoreMoneyConstraint is a subclass of Constraint class
class SendMoreMoneyConstraint(Constraint[str, int]):

    def __init__(self, letters: List[str]) -> None:
        super().__init__(letters)
```

```
        self.letters: List[str] = letters

    def satisfied(self, assignment: Dict[str, int]) -> bool:
        # if there are duplicate values then it's not a solution
        if len(set(assignment.values())) < len(assignment):
            return False

        # if all variables have been assigned, check if it adds correctly
        if len(assignment) == len(self.letters):
            s: int = assignment["S"]
            e: int = assignment["E"]
            n: int = assignment["N"]
            d: int = assignment["D"]
            m: int = assignment["M"]
            o: int = assignment["O"]
            r: int = assignment["R"]
            y: int = assignment["Y"]
            send: int = s * 1000 + e * 100 + n * 10 + d
            more: int = m * 1000 + o * 100 + r * 10 + e
            money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
            return send + more == money
        return True # no conflict

if __name__ == "__main__":
    letters: List[str] = ["S", "E", "N", "D", "M", "O", "R", "Y"]
    possible_digits: Dict[str, List[int]] = {}
    print("*****************************************************************")
    print("\nHere are the results:\n")
    for letter in letters:
        possible_digits[letter] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    possible_digits["M"] = [1]  # so we don't get answers starting with a 0
```

```
csp: CSP[str, int] = CSP(letters, possible_digits)
csp.add_constraint(SendMoreMoneyConstraint(letters))
solution: Optional[Dict[str, int]] = csp.backtracking_search()
if solution is None:
    print("No solution found!")
else:
    print(solution)
print("\n********************************************************************")
```

Output:

```
******************************************************************

Here are the results:

{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}

******************************************************************
```

Practical-4

Aim: Write a program to perform following operation

- Load the data from file
- Find out null and missing value
- Handle missing Value using different approach
- Plot the data using scatter plot, histogram, box plot

Solution:

Code:

1. Load the data from file

```
import pandas as pd
import matplotlib.pyplot as plt
# read csv file
df = pd.read_csv("food.csv")
# display DataFrame
print(df)
```

Output:

```
        Country  Real coffee  Instant coffee  Tea  Sweetener  Biscuits  \
0       Germany           90              49   88       19.0      57.0
1         Italy           82              10   60        2.0      55.0
2        France           88              42   63        4.0      76.0
3       Holland           96              62   98       32.0      62.0
4       Belgium           94              38   48       11.0      74.0
5    Luxembourg           97              61   86       28.0      79.0
6       England           27              86   99       22.0      91.0
7      Portugal           72              26   77        2.0      22.0
8       Austria           55              31   61       15.0      29.0
9   Switzerland           73              72   85       25.0      31.0
10       Sweden           97              13   93       31.0       NaN
11      Denmark           96              17   92       35.0      66.0
12       Norway           92              17   83       13.0      62.0
13      Finland           98              12   84       20.0      64.0
14        Spain           70              40   40        NaN      62.0
15      Ireland           30              52   99       11.0      80.0
```

```
    Powder soup  Tin soup  Potatoes  Frozen fish  ...  Apples  Oranges  \
0            51        19        21           27  ...      81       75
1            41         3         2            4  ...      67       71
2            53        11        23           11  ...      87       84
3            67        43         7           14  ...      83       89
4            37        23         9           13  ...      76       76
5            73        12         7           26  ...      85       94
6            55        76        17           20  ...      76       68
7            34         1         5           20  ...      22       51
8            33         1         5           15  ...      49       42
9            69        10        17           19  ...      79       70
10           43        43        39           54  ...      56       78
11           32        17        11           51  ...      81       72
12           51         4        17           30  ...      61       72
13           27        10         8           18  ...      50       57
14           43         2        14           23  ...      59       77
15           75        18         2            5  ...      57       52

    Tinned fruit  Jam  Garlic  Butter  Margarine  Olive oil  Yoghurt  \
0             44   71      22      91         85         74     30.0
1              9   46      80      66         24         94      5.0
2             40   45      88      94         47         36     57.0
3             61   81      15      31         97         13     53.0
4             42   57      29      84         80         83     20.0
5             83   20      91      94         94         84     31.0
6             89   91      11      95         94         57     11.0
7              8   16      89      65         78         92      6.0
8             14   41      51      51         72         28     13.0
9             46   61      64      82         48         61     48.0
10            53   75       9      68         32         48      2.0
11            50   64      11      92         91         30     11.0
12            34   51      11      63         94         28      2.0
13            22   37      15      96         94         17      NaN
14            30   38      86      44         51         91     16.0
15            46   89       5      97         25         31      3.0

    Crisp bread
0            26
1            18
2             3
```

2. Find out null and missing value

total number of missing values in the dataframe

df.isnull().sum().sum()

Output:

```
3
```

3. Handle missing Value using different approach

```
#filling missing values
df.fillna(method='bfill')
```

Output:

	Country	Real coffee	Instant coffee	Tea	Sweetener	Biscuits	Powder soup	Tin soup	Potatoes	Frozen fish	...	Apples	Oranges	Tinned fruit	Jam	Garlic	Butter	Margarine	Olive oil	Yoghurt	Crisp bread
0	Germany	90	49	88	19.0	57.0	51	19	21	27	...	81	75	44	71	22	91	85	74	30.0	26
1	Italy	82	10	60	2.0	55.0	41	3	2	4	...	67	71	9	46	80	66	24	94	5.0	18
2	France	88	42	63	4.0	76.0	53	11	23	11	...	87	84	40	45	88	94	47	36	57.0	3
3	Holland	96	62	98	32.0	62.0	67	43	7	14	...	83	89	61	81	15	31	97	13	53.0	15
4	Belgium	94	38	48	11.0	74.0	37	23	9	13	...	76	76	42	57	29	84	80	83	20.0	5
5	Luxembourg	97	61	86	28.0	79.0	73	12	7	26	...	85	94	83	20	91	94	94	84	31.0	24
6	England	27	86	99	22.0	91.0	55	76	17	20	...	76	68	89	91	11	95	94	57	11.0	28
7	Portugal	72	26	77	2.0	22.0	34	1	5	20	...	22	51	8	16	89	65	78	92	6.0	9
8	Austria	55	31	61	15.0	29.0	33	1	5	15	...	49	42	14	41	51	51	72	28	13.0	11
9	Switzerland	73	72	85	25.0	31.0	69	10	17	19	...	79	70	46	61	64	82	48	61	48.0	30
10	Sweden	97	13	93	31.0	66.0	43	43	39	54	...	56	78	53	75	9	68	32	48	2.0	93
11	Denmark	96	17	92	35.0	66.0	32	17	11	51	...	81	72	50	64	11	92	91	30	11.0	34
12	Norway	92	17	83	13.0	62.0	51	4	17	30	...	61	72	34	51	11	63	94	28	2.0	62
13	Finland	98	12	84	20.0	64.0	27	10	8	18	...	50	57	22	37	15	96	94	17	16.0	64
14	Spain	70	40	40	11.0	62.0	43	2	14	23	...	59	77	30	38	86	44	51	91	16.0	13
15	Ireland	30	52	99	11.0	80.0	75	18	2	5	...	57	52	46	89	5	97	25	31	3.0	9

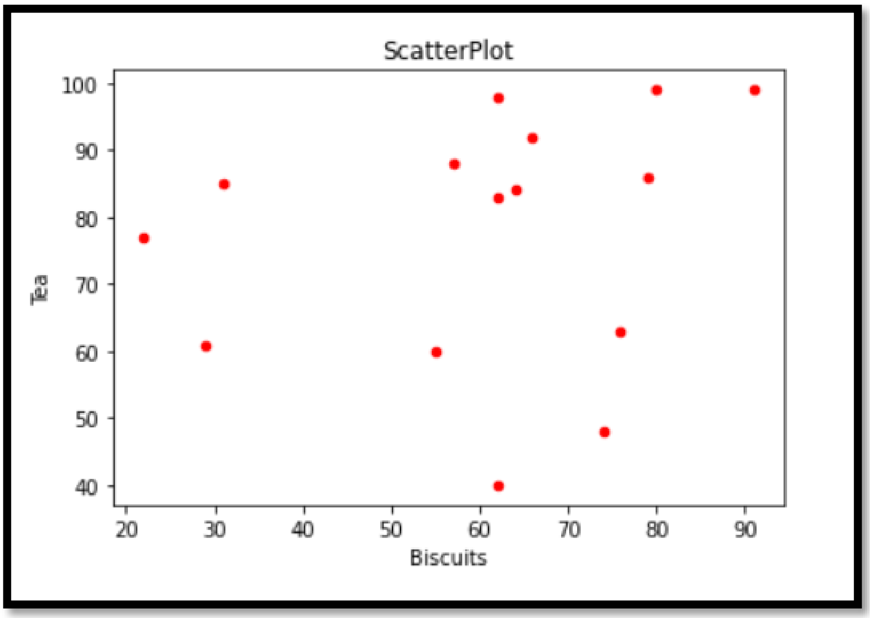
16 rows × 21 columns

4. Plot the data using scatter plot, histogram, box plot

- ### Scatter Plot

```
# scatter plot
df.plot(kind = 'scatter',
        x = 'Biscuits',
        y = 'Tea',
        color = 'red')
# set the title
plt.title('ScatterPlot')
# show the plot
plt.show()
```

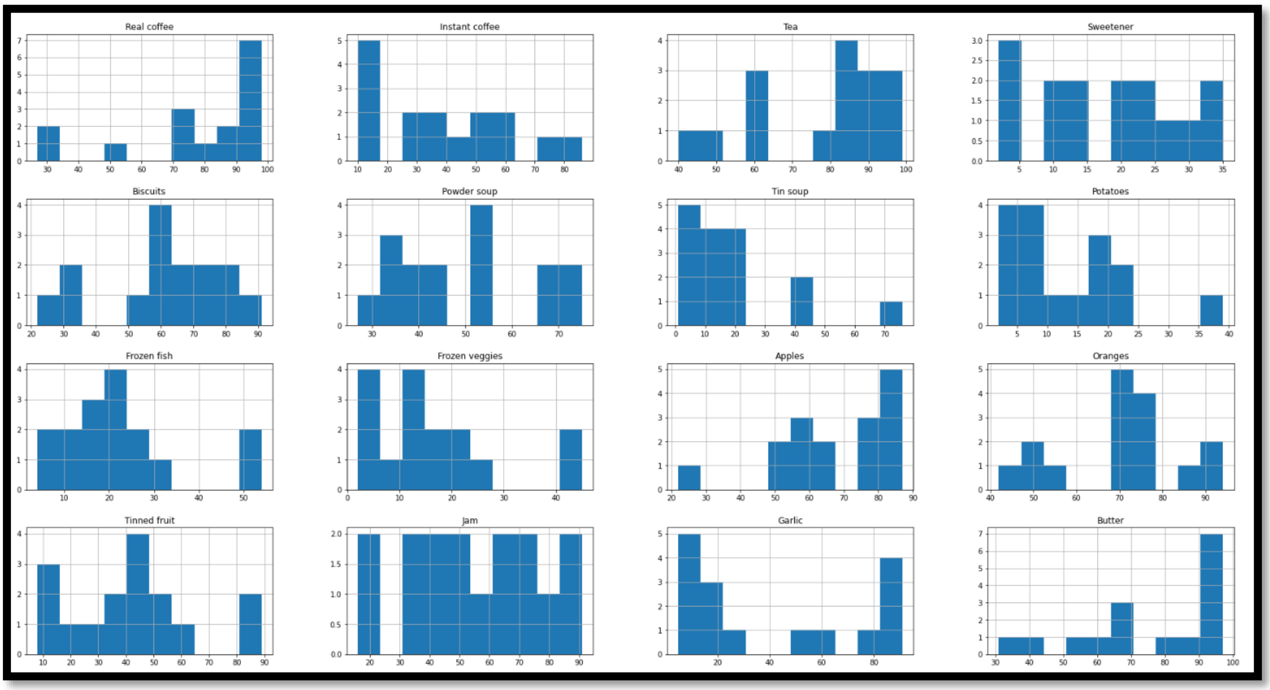
Output:


ScatterPlot
Tea
Biscuits
100
90
80
70
60
50
40
20
30
40
50
60
70
80
90

- ## Histogram

```
# creating a basic histogram
df.hist(figsize=[30, 20])
plt.show()
```

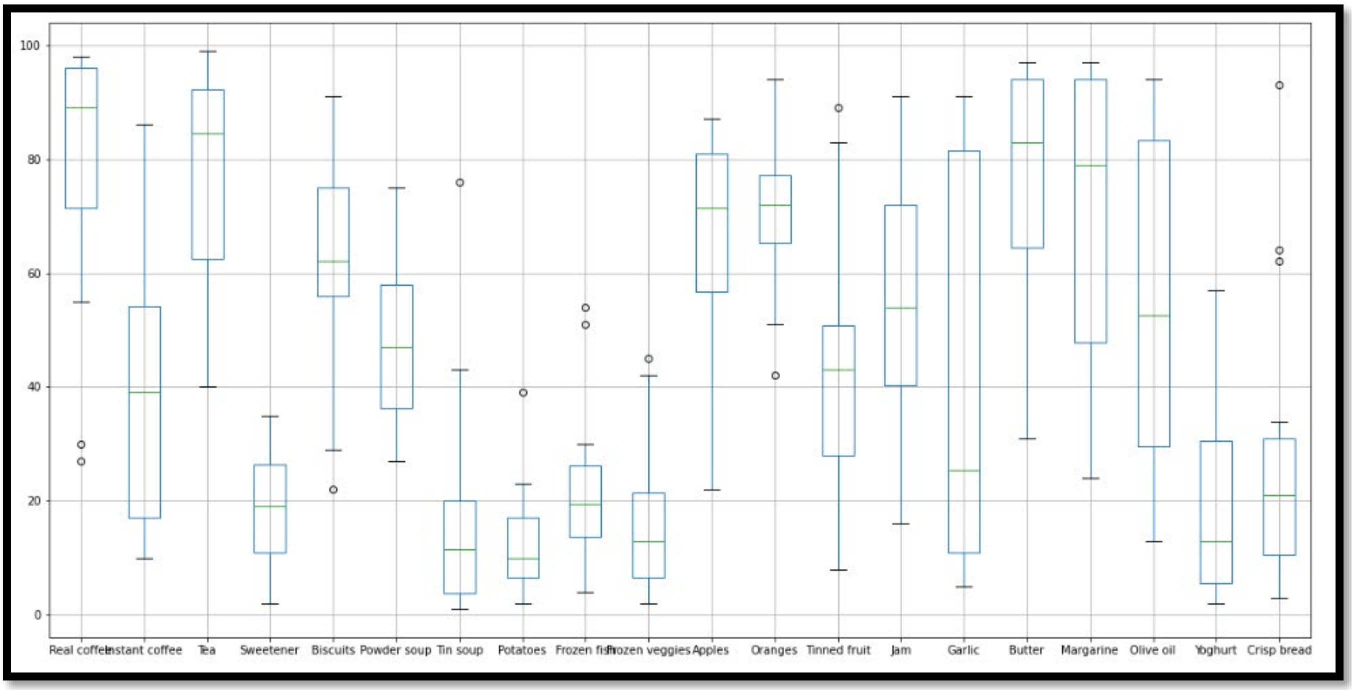
Output:



- **Box plot**

# creating a basic baxplot
df.boxplot(figsize=[20, 10])
plt.show()

Output:



Practical-5

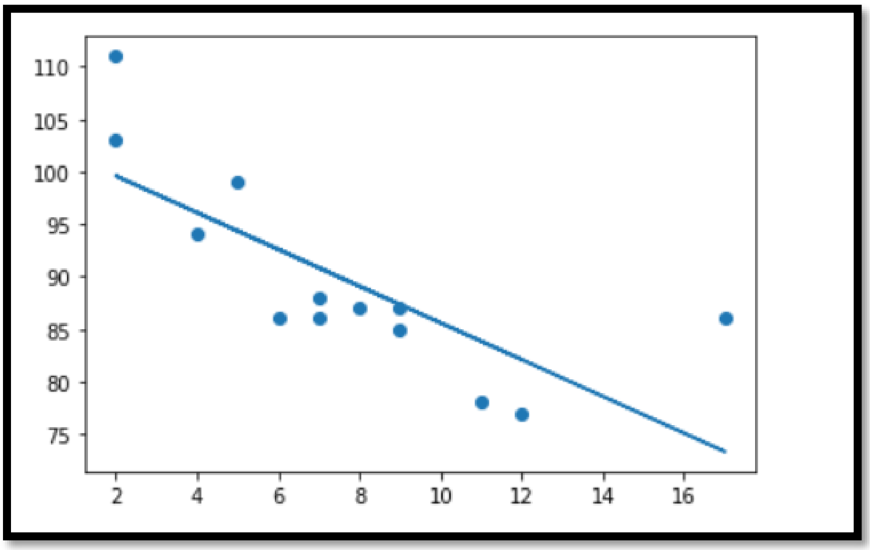
Aim: Write a program to implement Linear Regression.

Solution:

Code:

```
import matplotlib.pyplot as plt
from scipy import stats
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
slope, intercept, r, p, std_err = stats.linregress(x, y)
def myfunc(x):
  return slope * x + intercept
mymodel = list(map(myfunc, x))
plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
```

Output:



Practical-6

Aim: Write a program to implement k-Nearest Neighbor algorithm to classify the iris data set. Print both correct and wrong predictions.

Solution:

Code:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
class color:
   BOLD = '\033[1m'
   END = '\033[0m'
# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=15, random_state=50)
# Define the k-Nearest Neighbor algorithm
k = 3
knn = KNeighborsClassifier(n_neighbors=k)
# Train the k-Nearest Neighbor model on the training data
knn.fit(X_train, y_train)
# Use the k-Nearest Neighbor model to predict the classes of the testing data
y_pred = knn.predict(X_test)
# Print both correct and wrong predictions
correct_predictions = 0
wrong_predictions = 0
for i in range(len(y_test)):
```

```
    if y_test[i] == y_pred[i]:
        print(f"Correct prediction: Actual = {y_test[i]}, Predicted = {y_pred[i]}")
        correct_predictions += 1
    else:
        print(color.BOLD+"Wrong prediction: Actual = {y_test[i]}, Predicted = 
{y_pred[i]}"+color.END)
        wrong_predictions += 1
# Print the accuracy of the k-Nearest Neighbor model on the testing data
accuracy = correct_predictions / (correct_predictions + wrong_predictions)
print(f"Accuracy: {accuracy}")
```

Output:

```
Correct prediction: Actual = 1, Predicted = 1
Wrong prediction: Actual = {y_test[i]}, Predicted = {y_pred[i]}
Correct prediction: Actual = 0, Predicted = 0
Correct prediction: Actual = 0, Predicted = 0
Correct prediction: Actual = 2, Predicted = 2
Correct prediction: Actual = 2, Predicted = 2
Correct prediction: Actual = 2, Predicted = 2
Correct prediction: Actual = 0, Predicted = 0
Correct prediction: Actual = 0, Predicted = 0
Correct prediction: Actual = 1, Predicted = 1
Correct prediction: Actual = 0, Predicted = 0
Correct prediction: Actual = 2, Predicted = 2
Correct prediction: Actual = 0, Predicted = 0
Correct prediction: Actual = 2, Predicted = 2
Correct prediction: Actual = 1, Predicted = 1
Accuracy: 0.9333333333333333
```

Practical-7

Aim: Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

Solution:

Code:

```
import pandas as pd
import math
import numpy as np
data = pd.read_csv("3-dataset.csv")
features = [feat for feat in data]
features.remove("answer")
class Node:
    def __init__(self):
        self.children = []
        self.value = ""
        self.isLeaf = False
        self.pred = ""
def entropy(examples):
    pos = 0.0
    neg = 0.0
    for _, row in examples.iterrows():
        if row["answer"] == "yes":
            pos += 1
        else:
            neg += 1
    if pos == 0.0 or neg == 0.0:
        return 0.0
    else:
```

```
        p = pos / (pos + neg)
        n = neg / (pos + neg)
        return -(p * math.log(p, 2) + n * math.log(n, 2))
def info_gain(examples, attr):
    uniq = np.unique(examples[attr])
    #print ("\n",uniq)
    gain = entropy(examples)
    #print ("\n",gain)
    for u in uniq:
        subdata = examples[examples[attr] == u]
        #print ("\n",subdata)
        sub_e = entropy(subdata)
        gain -= (float(len(subdata)) / float(len(examples))) * sub_e
        #print ("\n",gain)
    return gain
def ID3(examples, attrs):
    root = Node()
    max_gain = 0
    max_feat = ""
    for feature in attrs:
        #print ("\n",examples)
        gain = info_gain(examples, feature)
        if gain > max_gain:
            max_gain = gain
            max_feat = feature
    root.value = max_feat
    #print ("\nMax feature attr",max_feat)
    uniq = np.unique(examples[max_feat])
    #print ("\n",uniq)
    for u in uniq:
        #print ("\n",u)
```

```
        subdata = examples[examples[max_feat] == u]
        #print ("\n",subdata)
        if entropy(subdata) == 0.0:
            newNode = Node()
            newNode.isLeaf = True
            newNode.value = u
            newNode.pred = np.unique(subdata["answer"])
            root.children.append(newNode)
        else:
            dummyNode = Node()
            dummyNode.value = u
            new_attrs = attrs.copy()
            new_attrs.remove(max_feat)
            child = ID3(subdata, new_attrs)
            dummyNode.children.append(child)
            root.children.append(dummyNode)
    return root
def printTree(root: Node, depth=0):
    for i in range(depth):
        print("\t", end="")
    print(root.value, end="")
    if root.isLeaf:
        print(" -> ", root.pred)
    print()
    for child in root.children:
        printTree(child, depth + 1)
def classify(root: Node, new):
    for child in root.children:
        if child.value == new[root.value]:
            if child.isLeaf:
                print ("Predicted Label for new example", new," is:", child.pred)
```

```
            exit
        else:
            classify (child.children[0], new)
root = ID3(data, features)
print("Decision Tree is:")
printTree(root)
print ("------------------")
new = {"outlook":"sunny", "temperature":"hot", "humidity":"normal", "wind":"strong"}
classify (root, new)
```

Output:

```
Decision Tree is:
outlook
        overcast ->  ['yes']

        rain
                wind
                        strong ->  ['no']

                        weak ->  ['yes']

        sunny
                humidity
                        high ->  ['no']

                        normal ->  ['yes']

------------------
Predicted Label for new example {'outlook': 'sunny', 'temperature': 'hot', 'humidity': 'normal', 'wind': 'strong'}  is: ['yes']
```

Practical-8

Aim: Write a program to classify IRIS data using Random forest classifier.

Solution:

Code:

```
#Import scikit-learn dataset library
from sklearn import datasets
#Load dataset
iris = datasets.load_iris()
# print the label species(setosa, versicolor,virginica)
print(iris.target_names)
print("\n")
# print the names of the four features
print(iris.feature_names)
print("\n")
# print the iris data (top 5 records)
print(iris.data[0:5])
print("\n")
# print the iris labels (0:setosa, 1:versicolor, 2:virginica)
print(iris.target)
print("\n")
# Creating a DataFrame of given iris dataset.
import pandas as pd
data=pd.DataFrame({
    'sepal length':iris.data[:,0],
    'sepal width':iris.data[:,1],
    'petal length':iris.data[:,2],
    'petal width':iris.data[:,3],
    'species':iris.target
})
data.head()
```

Import train_test_split function

from sklearn.model_selection import train_test_split

X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features

y=data['species'] # Labels

Split dataset into training set and test set

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

#Import Random Forest Model

from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier

clf=RandomForestClassifier(n_estimators=100)

#Train the model using the training sets y_pred=clf.predict(X_test)

clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

Model Accuracy, how often is the classifier correct?

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

Output:

```
['setosa' 'versicolor' 'virginica']


['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']


[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]


[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]


Accuracy: 0.8888888888888888
```

Practical-9

Aim: Write a program to classify iris dataset using SVM. Experiment with different kernel functions.

Solution:

Code:

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# Load the IRIS dataset
iris = load_iris()
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.1,
random_state=52)
# Create SVM objects with different kernel functions
svm_linear = SVC(kernel='linear')
svm_poly = SVC(kernel='poly', degree=3)
svm_rbf = SVC(kernel='rbf')
# Train the SVM classifiers on the training set
svm_linear.fit(X_train, y_train)
svm_poly.fit(X_train, y_train)
svm_rbf.fit(X_train, y_train)
# Make predictions on the testing set
y_pred_linear = svm_linear.predict(X_test)
y_pred_poly = svm_poly.predict(X_test)
y_pred_rbf = svm_rbf.predict(X_test)
# Calculate the accuracy of the classifiers
accuracy_linear = accuracy_score(y_test, y_pred_linear)
```

```
accuracy_poly = accuracy_score(y_test, y_pred_poly)
accuracy_rbf = accuracy_score(y_test, y_pred_rbf)
# Print the accuracy of the classifiers
print("Accuracy (Linear Kernel):", accuracy_linear)
print("Accuracy (Polynomial Kernel):", accuracy_poly)
print("Accuracy (RBF Kernel):", accuracy_rbf)
```

Output:

```
Accuracy (Linear Kernel): 0.9333333333333333
Accuracy (Polynomial Kernel): 0.9333333333333333
Accuracy (RBF Kernel): 0.9333333333333333
```

Practical-10

Aim: Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

Solution:

Code:

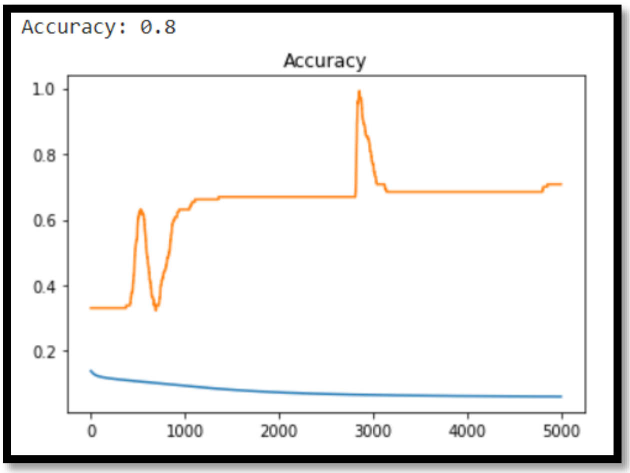
```
# Import Libraries
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
# Load dataset
data = load_iris()
# Get features and target
X=data.data
y=data.target
# Get dummy variable
y = pd.get_dummies(y).values
y[:3]
#Split data into train and test data
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=20, random_state=4)
# Initialize variables
learning_rate = 0.1
iterations = 5000
N = y_train.size
# number of input features
input_size = 4
# number of hidden layers neurons
hidden_size = 2
```

```
# number of neurons at the output layer
output_size = 3
results = pd.DataFrame(columns=["mse", "accuracy"])
# Initialize weights
np.random.seed(10)
# initializing weight for the hidden layer
W1 = np.random.normal(scale=0.5, size=(input_size, hidden_size))
# initializing weight for the output layer
W2 = np.random.normal(scale=0.5, size=(hidden_size , output_size))
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def mean_squared_error(y_pred, y_true):
    return ((y_pred - y_true)**2).sum() / (2*y_pred.size)
def accuracy(y_pred, y_true):
    acc = y_pred.argmax(axis=1) == y_true.argmax(axis=1)
    return acc.mean()
for itr in range(iterations):
    # feedforward propagation
    # on hidden layer
    Z1 = np.dot(x_train, W1)
    A1 = sigmoid(Z1)
    # on output layer
    Z2 = np.dot(A1, W2)
    A2 = sigmoid(Z2)
    # Calculating error
    mse = mean_squared_error(A2, y_train)
    acc = accuracy(A2, y_train)
    results=results.append({"mse":mse, "accuracy":acc},ignore_index=True )
    # backpropagation
    E1 = A2 - y_train
```

```
    dW1 = E1 * A2 * (1 - A2)
    E2 = np.dot(dW1, W2.T)
    dW2 = E2 * A1 * (1 - A1)
    # weight updates
    W2_update = np.dot(A1.T, dW1) / N
    W1_update = np.dot(x_train.T, dW2) / N
    W2 = W2 - learning_rate * W2_update
    W1 = W1 - learning_rate * W1_update
results.mse.plot(title="Mean Squared Error")
results.accuracy.plot(title="Accuracy")
# feedforward
Z1 = np.dot(x_test, W1)
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2)
A2 = sigmoid(Z2)
acc = accuracy(A2, y_test)
print("Accuracy: {}".format(acc))
```

Output:



Practical-11

Aim: Write a Program to implement K-Means clustering Algorithm.

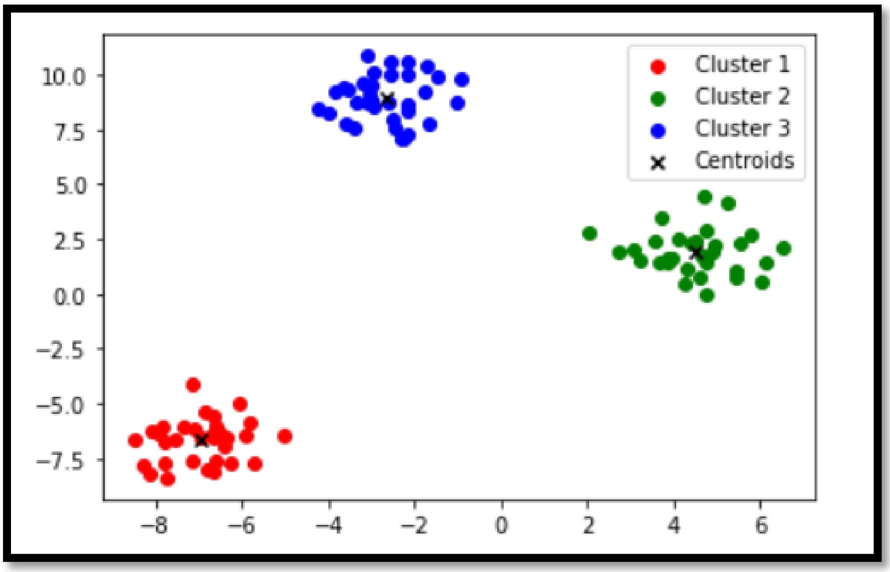
Solution:

Code:

```
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
class KMeans:
    def __init__(self, k, max_iter=100):
        self.k = k
        self.max_iter = max_iter
    def initialize_centroids(self, X):
        centroids = X[np.random.choice(len(X), self.k, replace=False)]
        return centroids
    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2)**2))
    def assign_clusters(self, X, centroids):
        clusters = [[] for _ in range(self.k)]
        for i, x in enumerate(X):
            distances = [self.euclidean_distance(x, c) for c in centroids]
            closest_cluster = np.argmin(distances)
            clusters[closest_cluster].append(i)
        return clusters
    def update_centroids(self, X, clusters):
        centroids = np.zeros((self.k, X.shape[1]))
        for i, cluster in enumerate(clusters):
            cluster_mean = np.mean(X[cluster], axis=0)
            centroids[i] = cluster_mean
        return centroids
```

```
    def fit(self, X):
        self.centroids = self.initialize_centroids(X)
        for i in range(self.max_iter):
            clusters = self.assign_clusters(X, self.centroids)
            old_centroids = self.centroids
            self.centroids = self.update_centroids(X, clusters)
            if np.allclose(old_centroids, self.centroids):
                break
        return clusters
    def predict(self, X):
        clusters = self.assign_clusters(X, self.centroids)
        y_pred = np.zeros(len(X))
        for i, cluster in enumerate(clusters):
            for j in cluster:
                y_pred[j] = i
        return y_pred
# generate sample data
X, y = make_blobs(n_samples=100, centers=3, n_features=2, random_state=42)
# create a KMeans object and fit the data
kmeans = KMeans(k=3, max_iter=100)
clusters = kmeans.fit(X)
# plot the data with color-coded clusters
colors = ['r', 'g', 'b']
for i, cluster in enumerate(clusters):
    plt.scatter(X[cluster, 0], X[cluster, 1], c=colors[i], label=f'Cluster {i+1}')
plt.scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1], marker='x', color='k', label='Centroids')
plt.legend()
plt.show()
```

Output:


Cluster 1
Cluster 2
Cluster 3
Centroids
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−8
−6
−4
−2
0
2
4
6

Practical-12

Aim: Case study/Project: Implementation of any real time application using suitable machine learning technique.

Solution:

Detecting Vehicle License Plate System

1) Background/ Problem Statement

Traffic control and vehicle owner identification have become a major problem in every country. Sometimes, it becomes difficult to identify vehicle owner who violates traffic rules and drives too fast.

Therefore, it is not possible to catch and punish those kinds of people because the traffic personnel might not be able to retrieve the vehicle number from the moving vehicle because of the speed of the vehicle. Therefore, there is a need to develop the system as one of the solutions to this problem.

Our Detect Vehicle License Plate will detect the license plate from an image. It can also detect in real-time using OpenCV and Python-tesseract. The user will just need to upload an image.

2) Working of the Project

This system will detect vehicle license plates from a user's uploaded file. The system will detect the license plate and if it's been detected, it will be displayed to the user. The back-end involves Python.

We have implemented OpenCV and Python-tesseract libraries. Python-tesseract is a wrapper for Google's Tesseract-OCR Engine. It is also useful as a stand-alone invocation script to tesseract, as it can read all image types supported.

3) Advantages

- It is easy to maintain.
- It is user-friendly.
- The system will automatically detect the license plate.
- It can detect vehicle license plates from an image.

4) System Description

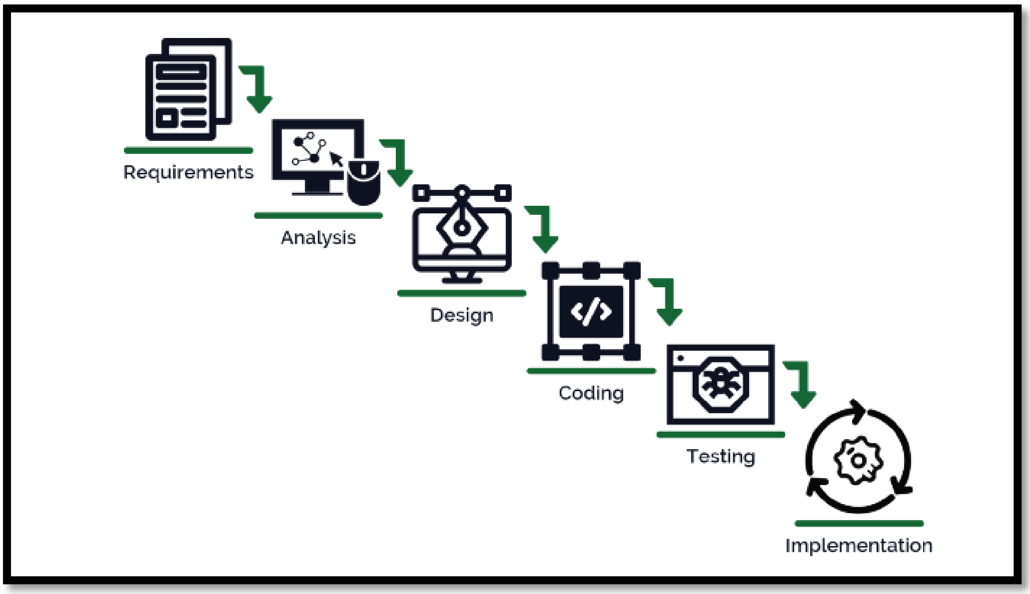
The system comprises 1 major module with their sub-modules as follows:

❖ **USER:**

- **Detection**
  - **Image**
    - User must upload an image for detection.

5) Project Life Cycle

The waterfall model is a classical model used in the system development life cycle to create a system with a linear and sequential approach. It is termed a waterfall because the model develops systematically from one phase to another in a downward fashion. The waterfall approach does not define the process to go back to the previous phase to handle changes in requirements. The waterfall approach is the earliest approach that was used for software development.



6) System Requirements

I. Hardware Requirement

i. Laptop or PC

- Windows 7 or higher
- I3 processor system or higher
- 4 GB RAM or higher
- 100 GB ROM or higher

II. Software Requirement

ii. Laptop or PC

- Python
- Sublime Text Editor

7) Limitation/Disadvantages

- The system might not be able to detect vehicle license plates if there's a dim light environment while capturing.

8) Application

- This application detects the vehicle number plate automatically.

Code:

```
import cv2
np_cascade=cv2.CascadeClassifier('np.xml')
img =cv2.imread('1.jpg')
grey=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
np=np_cascade.detectMultiScale (grey, 1.1, 4)
for (x, y, w, h) in np:
    cv2.rectangle(img, (x, y), (x+w , y+h), (255, 0 ,0), 2)
cv2. imshow('Detected Plate', img)
cv2.waitKey(0)
```

Output:

9) Reference

- https://www.researchgate.net/publication/334424122_Research_and_Application_of_License_Plate_Recognition_Technology_Based_on_Deep_Learning
- https://www.researchgate.net/publication/236888959_Automatic_Number_Plate_Recognition_System_ANPR_A_Survey
- https://mospace.umsystem.edu/xmlui/bitstream/handle/10355/43016/research.pdf