Accredited by NAAC & NBA*

SVCE BENGALURU

SRI VENKATESHWARA COLLEGE OF ENGINEERING

— Affiliated to VTU, Approved by AICTE, Recognised by UGC u/s 2(f) & 12(B)—

ARTIFICIAL INTELLIGENCE & MACHINE LEARNING LABORATORY

18CSL76

Prepared By

Ms. AMRUTHA C

Department of ISE, SVCE, Bengaluru

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

INSTITUTE VISION

To be a premier institute for addressing the challenges in global perspective.

INSTITUTE MISSION

M1: Nurture students with professional and ethical outlook to identify needs, analyze, design and innovate sustainable solutions through lifelong learning in service of society as individual or a team.

M2. Establish State of the Art Laboratories and Information Resource Centre for education and research.

M3. Collaborate with Industry, Government Organization and Society to align the curriculum and outreach activities.

DEPARTMENT VISION

Excellence in Information Science and Engineering Education, Research and Development.

DEPARTMENT MISSION

M1: Accomplish academic achievement in Information Science and Engineering through student-centered creative teaching learning, qualified faculty and staff members, assessment and effective ICT.

M2: Establish a Center of Excellence in a various verticals of Information science and engineering to encourage collaborative research and Industry-institute collaboration.

M3: Transform the engineering students to socially responsible, ethical, technically competent and Value added professional or entrepreneur through holistic education.

PROGRAM EDUCATIONALOBJECTIVES

PEO-1: Information Science and Engineering Graduates will have professional technical career in inter disciplinary domains providing innovative and sustainable solutions using modern tools.

PEO-2: Information Science and Engineering Graduates will have effective communication, leadership, team building, problem solving, decision making and creative skills.

PEO-3: Information Science and Engineering Graduates will practice ethical responsibilities towards their peers, employers and society.

PROGRAM SPECIFIC OUTCOMES

PSO1: Ability to learn and effectively implement the Information Science and Engineering notions in less span of time using modern tools.

PSO2: Ability to visualize the operations of existing and future software applications.

PROGRAMME OUTCOME

PO1-Engineering knowledge:

Applythe knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complexengineering problems.

PO2-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.

PO3-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.

PO4-Conduct investigations of complex problems:

Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of theinformation to provide valid conclusions.

PO5-Modern tool usage:

Create, select, and apply appropriate techniques, resources, andmodern engineering and IT tools including prediction and modeling to complex engineering activities withan understanding of the limitations.

PO6-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 theprofessional engineering practice.

PO7-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.

PO8-Ethics:

Apply ethical principles and commit to professional ethics and responsibilities and normsof the engineering practice.

PO9-Individual and teamwork:

Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10-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.

PO11-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.

PO12-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.

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

DESCRIPTION

1. COURSE OUTCOMES:

CO1: Apply the Algorithms for A* search, AO* search(KL3)

CO2: Apply the Candidate Elimination, Decision Tree, Artificial Neural Network Algorithms(KL3)

CO3: Apply the Algorithms for Naïve Bayesian Classifier and EM Classifier(KL3)

CO4: Apply the K-Nearest Neighbor and Regression Algorithms(KL3)

2. RESOURCES REQUIRED:

1. Hardware resources:

- Desktop PC
- Windows/Linux operating system

2. Software resources:

- Anaconda IDE, Python

3. Dataset from standard repositories

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

PROGRAMS LIST

1.	Implement A* Search algorithm.
2.	Implement AO* Search algorithm.
3.	For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples
4.	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
5.	Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.
6.	Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.
7.	Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.
8.	Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.
9.	Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

INTRODUCTION

1. Artificial intelligence:

Artificial intelligence is the simulation of human intelligence processes by machines, especially computer systems.

2. Machine learning:

Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.

- **How does machine learning works?**

  Machine learning is a form of artificial intelligence (AI) that teaches computers to think in a similar way to how humans do: Learning and improving upon past experiences. It works by exploring data and identifying patterns, and involves minimal human intervention.

3. Types of Machine Learning:

1. Supervised.
2. Unsupervised.
3. Reinforcement**.**

➢ **Supervised Learning:**

Supervised learning is one of the most basic types of machine learning. In this type, the machine learning algorithm is trained on labeled data. Even though the data needs to be labeled accurately for this method to work, supervised learning is extremely powerful when used in the right circumstances.

➢ **Unsupervised Learning:**

Unsupervised machine learning holds the advantage of being able to work with unlabeled data. This means that human labor is not required to make the dataset machine-readable, allowing much larger datasets to be worked on by the program.

➢ **Reinforcement Learning:**

Reinforcement learning directly takes inspiration from how human beings learn from data in their lives. It features an algorithm that improves upon itself and learns from new situations using a trial-and-error method. Favorable outputs are encouraged or 'reinforced', and non-favorable outputs are discouraged or 'punished'.

Note:

- Classification and Regression are types of supervised learning algorithms
- Clustering is a type of unsupervised algorithm.

4. Classification:

Classification is a type of supervised machine learning algorithm. For any given input, the classification algorithms help in the prediction of the class of the output variable. There can be multiple types of classifications like binary classification, multi-class classification, etc. It depends upon the number of classes in the output variable.

Types of Classification algorithms:

- **Logistic Regression:**

It is one of the linear models which can be used for classification. It uses the sigmoid function to calculate the probability of a certain event occurring. It is an ideal method for the classification of binary variables.

- **K-Nearest Neighbours (KNN):**

It uses distance metrics like Euclidean distance, Manhattan distance, etc. to calculate the distance of one data point from every other data point. To classify the output, it takes a majority vote from k nearest neighbors of each data point.

- **Decision Trees:**

It is a non-linear model that overcomes a few of the drawbacks of linear algorithms like Logistic regression. It builds the classification model in the form of a tree structure that includes nodes and leaves. This algorithm involves multiple if-else statements which help in breaking down the structure into smaller structures and eventually providing the final outcome. It can be used for regression as well as classification problems.

- **Random Forest:**

It is an ensemble learning method that involves multiple decision trees to predict the outcome of the target variable. Each decision tree provides its own outcome. In the case of the classification problem, it takes the majority vote of these multiple decision trees to classify the final outcome. In the case of the regression problem, it takes the average of the values predicted by the decision trees.

- **Naïve Bayes:**

It is an algorithm that is based upon Bayes' theorem. It assumes that any particular feature is independent of the inclusion of other features. i.e. They are not correlated to one another. It generally does not work well with complex data due to this assumption as in most of the data sets there exists some kind of relationship between the features.

- **Support Vector Machine:**

It represents the data points in multi-dimensional space. These data points are then segregated into classes with the help of hyperplanes. It plots an n-dimensional space for the n number of features in the dataset and then tries to create the hyperplanes such that it divides the data points with maximum margin.

5. Clustering:

Clustering is a type of unsupervised machine learning algorithm. It is used to group data points having similar characteristics as clusters. Ideally, the data points in the same cluster should exhibit similar properties and the points in different clusters should be as dissimilar as possible.

Clustering is divided into two groups:

- Hard clustering.
- Soft clustering.

In hard clustering, the data point is assigned to one of the clusters only whereas in soft clustering, it provides a probability likelihood of a data point to be in each of the clusters.

Types of Clustering algorithms:

- **K-Means Clustering**:

It initializes a pre-defined number of k clusters and uses distance metrics to calculate the distance of each data point from the centroid of each cluster. It assigns the data points into one of the k clusters based on its distance.

- **Agglomerative Hierarchical Clustering (Bottom-Up Approach)**:

It considers each data point as a cluster and merges these data points on the basis of distance metric and the criterion which is used for linking these clusters.

- **Divisive Hierarchical Clustering (Top-Down Approach)**:

It initializes with all the data points as one cluster and splits these data points on the basis of distance metric and the criterion. Agglomerative and Divisive clustering can be represented as a dendrogram and the number of clusters to be selected by referring to the same.

- **DBSCAN (Density-based Spatial Clustering of Applications with Noise):**

It is a density-based clustering method. Algorithms like K-Means work well on the clusters that are fairly separated and create clusters that are spherical in shape. DBSCAN is used when the data is in arbitrary shape and it is also less sensitive to the outliers. It groups the data points that have many neighbouring data points within a certain radius.

- **OPTICS (Ordering Points to Identify Clustering Structure)**:

It is another type of density-based clustering method and it is similar in process to DBSCAN except that it considers a few more parameters. But it is more computationally complex than DBSCAN. Also, it does not separate the data points into clusters, but it creates a reachability plot which can help in the interpretation of creating clusters.

- **BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies)**:

It creates clusters by generating a summary of the data. It works well with huge datasets as it first summarises the data and then uses the same to create clusters.

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

PROGRAMS

1. EXPERIMENT NO: 1

2. TITLE: Implement A* Search algorithm.

3. THEORY: This Algorithm is the advanced form of the BFS algorithm (Breadth-first search), which searches for the shorter path first than, the longer paths. It is a **complete** as well as an **optimal** solution for solving path and grid problems.

Optimal – find the least cost from the starting point to the ending point.

Complete – It means that it will find all the available paths from start to end.

4. ALGORITHM:

1. Firstly, Place the starting node into OPEN and find its f (n) value.

2. Then remove the node from OPEN, having the smallest f (n) value. If it is a goal node, then stop and return to success.

3: Else remove the node from OPEN, and find all its successors.

4: Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

5: Goto Step-2.

6: Exit.

5. SOLUTION:

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)
    closed_set = set()
    g = {} #store distance from starting node
    parents = {}# parents contains an adjacency map of all nodes
```

```
#ditance of starting node from itself is zero
g[start_node] = 0
#start_node is root node i.e it has no parent nodes
#so start_node is set to its own parent node
parents[start_node] = start_node


while len(open_set) > 0:
    n = None

    #node with lowest f() is found
    for v in open_set:
        if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
            n = v


    if n == stop_node or Graph_nodes[n] == None:
        pass
    else:
        for (m, weight) in get_neighbors(n):
            #nodes 'm' not in first and last set are added to first
            #n is set its parent
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            #for each node m,compare its distance from start i.e g(m) to the
            #from start through n node
            else:
                if g[m] > g[n] + weight:
                    #update g(m)
                    g[m] = g[n] + weight
                    #change parent of m to n
                    parents[m] = n
```

```
                    #if m in closed set,remove and add to open
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                path = []

                while parents[n] != n:
                    path.append(n)
                    n = parents[n]

                path.append(start_node)

                path.reverse()

                print('Path found: {}'.format(path))
                return path
            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_set.remove(n)
            closed_set.add(n)

        print('Path does not exist!')
        return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
```

```
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```

6. OUTPUT:

Path found: ['A', 'E', 'D', 'G']

OR

7. VARIANT-2:

```
from __future__ import print_function
import matplotlib.pyplot as plt


class AStarGraph(object):
    #Define a class board like grid with two barriers


    def __init__(self):
            self.barriers = []

    self.barriers.append([(2,4),(2,5),(2,6),(3,6),(4,6),(5,6),(5,5),(5,4),(5,3),(5,2),(
4,2),(3,2)])


    def heuristic(self, start, goal):
            #Use Chebyshev distance heuristic if we can move one square either
            #adjacent or diagonal
            D = 1
            D2 = 1
            dx = abs(start[0] - goal[0])
            dy = abs(start[1] - goal[1])
            return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

```
    def get_vertex_neighbours(self, pos):
        n = []
        #Moves allow link a chess king
        for dx, dy in [(1,0),(-1,0),(0,1),(0,-1),(1,1),(-1,1),(1,-1),(-1,-1)]:
            x2 = pos[0] + dx
            y2 = pos[1] + dy
            if x2 < 0 or x2 > 7 or y2 < 0 or y2 > 7:
                continue
            n.append((x2, y2))
        return n

    def move_cost(self, a, b):
        for barrier in self.barriers:
            if b in barrier:
                return 100 #Extremely high cost to enter barrier squares
        return 1 #Normal movement cost

def AStarSearch(start, end, graph):

    G = {} #Actual movement cost to each position from the start position
    F = {} #Estimated movement cost of start to end going via this position

    #Initialize starting values
```

```
G[start] = 0
F[start] = graph.heuristic(start, end)

closedVertices = set()
openVertices = set([start])
cameFrom = {}

while len(openVertices) > 0:
    #Get the vertex in the open list with the lowest F score
    current = None
    currentFscore = None
    for pos in openVertices:
        if current is None or F[pos] < currentFscore:
            currentFscore = F[pos]
            current = pos

    #Check if we have reached the goal
    if current == end:
        #Retrace our route backward
        path = [current]
        while current in cameFrom:
            current = cameFrom[current]
            path.append(current)
```

```
                path.reverse()
                return path, F[end] #Done!


        #Mark the current vertex as closed
        openVertices.remove(current)
        closedVertices.add(current)

        #Update scores for vertices near the current position
        for neighbour in graph.get_vertex_neighbours(current):
                if neighbour in closedVertices:
                        continue #We have already processed this node
exhaustively
                candidateG = G[current] + graph.move_cost(current,
neighbour)

                if neighbour not in openVertices:
                        openVertices.add(neighbour) #Discovered a new vertex
                elif candidateG >= G[neighbour]:
                        continue #This G score is worse than previously found

                #Adopt this G score
                cameFrom[neighbour] = current
                G[neighbour] = candidateG
                H = graph.heuristic(neighbour, end)
```

```
                    F[neighbour] = G[neighbour] + H


        raise RuntimeError("A* failed to find a solution")


    if __name__=="__main__":
        graph = AStarGraph()
        result, cost = AStarSearch((0,0), (7,7), graph)
        print ("route", result)
        print ("cost", cost)
        plt.plot([v[0] for v in result], [v[1] for v in result])
        for barrier in graph.barriers:
                plt.plot([v[0] for v in barrier], [v[1] for v in barrier])
        plt.xlim(-1,8)
        plt.ylim(-1,8)
        plt.show()
```

OR

8. VARIANT-3:

```
from collections import deque


class Graph:
    def __init__(self, adjac_lis):
```

```
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    # This is heuristic function which is having equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start, stop):
        # In this open_lst is a lisy of nodes which have been visited, but who's
        # neighbours haven't all been always inspected, It starts off with the start
#node
        # And closed_lst is a list of nodes which have been visited
        # and who's neighbors have been always inspected
```

```
open_lst = set([start])
closed_lst = set([])

# poo has present distances from start to all other nodes
# the default value is +infinity
poo = {}
poo[start] = 0

# par contains an adjac mapping of all nodes
par = {}
par[start] = start

while len(open_lst) > 0:
    n = None

    # it will find a node with the lowest value of f() -
    for v in open_lst:
        if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
            n = v;

    if n == None:
        print('Path does not exist!')
```

```
        return None
    # if the current node is the stop
    # then we start again from start
    if n == stop:
        reconst_path = []

        while par[n] != n:
            reconst_path.append(n)
            n = par[n]
        reconst_path.append(start)
        reconst_path.reverse()
        print('Path found: {}'.format(reconst_path))
        return reconst_path
    # for all the neighbors of the current node do
    for (m, weight) in self.get_neighbors(n):
      # if the current node is not presentin both open_lst and closed_lst
        # add it to open_lst and note n as it's par
        if m not in open_lst and m not in closed_lst:
            open_lst.add(m)
            par[m] = n
            poo[m] = poo[n] + weight
        # otherwise, check if it's quicker to first visit n, then m
```

```
            # and if it is, update par data and poo data
            # and if the node was in the closed_lst, move it to open_lst
            else:
                if poo[m] > poo[n] + weight:
                    poo[m] = poo[n] + weight
                    par[m] = n
                    if m in closed_lst:
                        closed_lst.remove(m)
                        open_lst.add(m)
        # remove n from the open_lst, and add it to closed_lst
        # because all of his neighbors were inspected
        open_lst.remove(n)
        closed_lst.add(n)
    print('Path does not exist!')
    return None
adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

1. EXPERIMENT NO: 2

2. TITLE: Implement AO* Search algorithm.

3. THEORY:

AO* Algorithm basically based on problem decompositon (Breakdown problem into small pieces)
When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution.

4. ALGORITHM:

Step-1: Create an initial graph with a single node (start node).
Step-2: Transverse the graph following the current path, accumulating node that has not yet been expanded or solved.
Step-3: Select any of these nodes and explore it. If it has no successors then call this value- FUTILITY else calculate f'(n) for each of the successors.
Step-4: If **f'(n)=0**, then mark the node as **SOLVED**.
Step-5: Change the value of f'(n) for the newly created node to reflect its successors by backpropagation.
Step-6: Whenever possible use the most promising routes, If a node is marked as SOLVED then mark the parent node as SOLVED.
Step-7: If the starting node is SOLVED or value is greater than **FUTILITY** then stop else repeat from Step-2.

5. SOLUTION:

```
class Graph:

    def __init__(self, graph, heuristicNodeList, startNode):

        self.graph = graph

        self.H=heuristicNodeList
```

```
    self.start=startNode
    self.parent={}
    self.status={}
    self.solutionGraph={}
def applyAOStar(self): # starts a recursive AO* algorithm
    self.aoStar(self.start, False)
def getNeighbors(self, v): # gets the Neighbors of a given node
    return self.graph.get(v,'')
def getStatus(self,v): # return the status of a given node
    return self.status.get(v,0)
def setStatus(self,v, val): # set the status of a given node
    self.status[v]=val
def getHeuristicNodeValue(self, n):
    return self.H.get(n,0) # always return the heuristic value of a given node
def setHeuristicNodeValue(self, n, value):
    self.H[n]=value # set the revised heuristic value of a given node
def printSolution(self):
```

```
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE
START NODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")
    def computeMinimumCostChildNodes(self, v): # Computes the Minimum Cost
of child nodes of a given node v
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v): # iterate over all the set of
child node/s
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                print("c, weight",nodeInfoTupleList)
```

```
            cost=cost+self.getHeuristicNodeValue(c)+weight
            nodeList.append(c)
        if flag==True: # initialize Minimum Cost with the cost of first set of child node/s
            minimumCost=cost
            costToChildNodeListDict[minimumCost]=nodeList #set the Minimum Cost child node/s
            print("Cost to Child Node",costToChildNodeListDict)
            flag=False
        else: # checking the Minimum Cost nodes with the current Minimum Cost
            if minimumCost>cost:
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList #set the Minimum Cost child node/s
      return minimumCost, costToChildNodeListDict[minimumCost] #return Minimum Cost and Minimum Cost child node/s
   def aoStar(self, v, backTracking): # AO* algorithm for a start node and
```

```
backTracking status flag

        print("HEURISTIC VALUES :", self.H)

        print("SOLUTION GRAPH :", self.solutionGraph)

        print("PROCESSING NODE :", v)

       # print("------------------------------------------------------------------------------------
-----")

        if self.getStatus(v) >= 0: # if status node v >= 0,compute Minimum Cost
nodes of v

            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)

            self.setHeuristicNodeValue(v, minimumCost)

            self.setStatus(v,len(childNodeList))

            solved=True # check the Minimum Cost nodes of v are solved

        for childNode in childNodeList:

            self.parent[childNode]=v

            if self.getStatus(childNode)!=-1:

                solved=solved & False

        if solved==True: # if the Minimum Cost nodes of v are solved, set the current
```

```
node status as solved(-1)
        self.setStatus(v,-1)
        self.solutionGraph[v]=childNodeList # update the solution graph with the solved nodes which may be a part of solution
    if v!=self.start: # check the current node is the start node for backtracking the current node value
        self.aoStar(self.parent[v], True) # backtracking the current node value with backtracking status set to true
    if backTracking==False: # check the current call is not for backtracking
        for childNode in childNodeList: # for each Minimum Cost child node
            self.setStatus(childNode,0) # set the status of child node to 0(needs exploration)
            self.aoStar(childNode, False) # Minimum Cost child node is further explored with backtracking status as false
h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7,'I': 7, 'J': 1}
graph1 = {
  'A': [[('B', 1), ('C', 1)], [('D', 1)]],
```

```
    'B': [[('G', 1)], [('H', 1)]],

    'C': [[('J', 1)]],

    'D': [[('E', 1), ('F', 1)]],

    'G': [[('I', 1)]]

}

G1= Graph(graph1, h1, 'A')

G1.applyAOStar()

G1.printSolution()
```

6. OUTPUT:

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}

OR

7. VARIANT-2:

```
def recAOStar(n):
    global finalPath
    print("Expanding Node:",n)
    and_nodes = []
```

```
    or_nodes =[]
    if(n in allNodes):
        if 'AND' in allNodes[n]:
            and_nodes = allNodes[n]['AND']
        if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']
    if len(and_nodes)==0 and len(or_nodes)==0:
        return
    solvable = False
    marked ={}
    while not solvable:
        if len(marked)==len(and_nodes)+len(or_nodes):
            min_cost_least,min_cost_group_least=
least_cost_group(and_nodes,or_nodes,{})
            solvable = True
            change_heuristic(n,min_cost_least)
            optimal_child_group[n] = min_cost_group_least
            continue
        min_cost,min_cost_group = least_cost_group(and_nodes,or_nodes,marked)
        is_expanded = False
        if len(min_cost_group)>1:
            if(min_cost_group[0] in allNodes):
```

```
            is_expanded = True
            recAOStar(min_cost_group[0])
        if(min_cost_group[1] in allNodes):
            is_expanded = True
            recAOStar(min_cost_group[1])
    else:
        if(min_cost_group in allNodes):
            is_expanded = True
            recAOStar(min_cost_group)
    if is_expanded:
        min_cost_verify, min_cost_group_verify = least_cost_group(and_nodes, or_nodes, {})
        if min_cost_group == min_cost_group_verify:
            solvable = True
            change_heuristic(n, min_cost_verify)
            optimal_child_group[n] = min_cost_group
    else:
        solvable = True
        change_heuristic(n, min_cost)
        optimal_child_group[n] = min_cost_group
    marked[min_cost_group]=1
  return heuristic(n)
```

```
def least_cost_group(and_nodes, or_nodes, marked):
    node_wise_cost = {}
    for node_pair in and_nodes:
        if not node_pair[0] + node_pair[1] in marked:
            cost = 0
            cost = cost + heuristic(node_pair[0]) + heuristic(node_pair[1]) + 2
            node_wise_cost[node_pair[0] + node_pair[1]] = cost
    for node in or_nodes:
        if not node in marked:
            cost = 0
            cost = cost + heuristic(node) + 1
            node_wise_cost[node] = cost
    min_cost = 999999
    min_cost_group = None
    for costKey in node_wise_cost:
        if node_wise_cost[costKey] < min_cost:
            min_cost = node_wise_cost[costKey]
            min_cost_group = costKey
    return [min_cost, min_cost_group]
def heuristic(n):
    return H_dist[n]
def change_heuristic(n, cost):
```

```
    H_dist[n] = cost
    return
def print_path(node):
    print(optimal_child_group[node], end="")
    node = optimal_child_group[node]
    if len(node) > 1:
        if node[0] in optimal_child_group:
            print("->", end="")
            print_path(node[0])
        if node[1] in optimal_child_group:
            print("->", end="")
            print_path(node[1])
    else:
        if node in optimal_child_group:
            print("->", end="")
            print_path(node)
H_dist = {
 'A': -1,
 'B': 4,
 'C': 2,
 'D': 3,
 'E': 6,
```

```
 'F': 8,
 'G': 2,
 'H': 0,
 'I': 0,
 'J': 0
}
allNodes = {
 'A': {'AND': [('C', 'D')], 'OR': ['B']},
 'B': {'OR': ['E', 'F']},
 'C': {'OR': ['G'], 'AND': [('H', 'I')]},
 'D': {'OR': ['J']}
}
optimal_child_group = {}
optimal_cost = recAOStar('A')
print('Nodes which gives optimal cost are')
print_path('A')
print('\nOptimal Cost is :: ', optimal_cost)
```

OR

8. VARIANT-3:

```
from queue import PriorityQueue

def heuristic(node):
    # Define your heuristic function here
```

```
    return 0

def ao_star(start, goal, successors):
    closed = set()
    fringe = PriorityQueue()
    fringe.put((heuristic(start), 0, start, None))

    while not fringe.empty():
        _, g, current, parent = fringe.get()

        if current == goal:
            path = [current]
            while parent is not None:
                path.append(parent)
                parent = parent[3]
            path.reverse()
            return path

        closed.add(current)

        for neighbor in successors(current):
            if neighbor in closed:
                continue
            ng = g + neighbor[2]
            h = heuristic(neighbor[0])
            fringe.put((ng + h, ng, neighbor[0], (current, neighbor[1], neighbor[2],
parent)))

    return None  # No path found

# Define your start and goal nodes
start = (0, 0)
goal = (5, 5)

# Define your successors function
```

```
def successors(node):
    x, y = node
    neighbors = [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]
    successors = []
    for neighbor in neighbors:
        if 0 <= neighbor[0] < 6 and 0 <= neighbor[1] < 6:
            successors.append((neighbor, "move", 1))
    return successors

# Call the AO* algorithm
path = ao_star(start, goal, successors)

# Print the result
if path is not None:
    print("Found path:", path)
else:
    print("No path found.")
```

1. EXPERIMENT NO: 3

2. TITLE: For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples

3. THEORY:

- The key idea in the Candidate-Elimination algorithm is to output a description of the set of all hypotheses consistent with the training examples.
- It computes the description of this set without explicitly enumerating all of its members.
- This is accomplished by using the more-general-than partial ordering and maintaining a compact representation of the set of consistent hypotheses.
- The algorithm represents the set of all hypotheses consistent with the observed training examples. This subset of all hypotheses is called the

version space with respect to the hypothesis space H and the training examples D, because it contains all plausible versions of the target concept.

- A version space can be represented with its general and specific boundary sets.
- The Candidate-Elimination algorithm represents the version space by storing only its most general members G and its most specific members S.
- Given only these two sets S and G, it is possible to enumerate all members of a version space by generating hypotheses that lie between these two sets in general-to-specific partial ordering over hypotheses. Every member of the version space lies between these boundaries.

4. ALGORITHM:

Step1: Load Data set
Step2: Initialize General Hypothesis and Specific Hypothesis.
Step3: For each training example
Step4: If example is positive example
    if attribute_value == hypothesis_value:
      Do nothing
    else:
      replace attribute value with '?' (Basically generalizing it)
Step5: If example is Negative example
    Make generalize hypothesis more specific.

5. DATASET:

sky	airtemp	humidity	wind	water	forcast	enjoysport
sunny	warm	normal	strong	warm	same	yes
sunny	warm	high	strong	warm	same	yes
rainy	cold	high	strong	warm	change	no
sunny	warm	high	strong	cool	change	yes

6. SOLUTION:

```
import numpy as np
import pandas as pd
data=pd.DataFrame(data=pd.read_csv('D:/enjoysport.csv'))
print(data)
concepts=np.array(data.iloc[:,0:-1])
```

```
print(concepts)
target=np.array(data.iloc[:,-1])
print(target)
def learn(concepts,target):
    specific_h=concepts[0].copy()
    print("initialization of specific_h and general_h")
    print(specific_h)
    general_h=[["?" for i in range(len(specific_h))] for i in
range(len(specific_h))]
    print(general_h)

    for i,h in enumerate(concepts):
        if target[i]=="yes":
            for x in range(len(specific_h)):
                if h[x]!=specific_h[x]:
                    specific_h[x]='?'
                    general_h[x][x]='?'

        if target[i]=="no":
            for x in range(len(specific_h)):
                if h[x]!=specific_h[x]:
                    general_h[x][x]=specific_h[x]
                else:
                    general_h[x][x]='?'

        print("steps of candidate Elimination Algorithm",i+1)
        print(specific_h)
        print(general_h)

    indices=[i for i,val in enumerate(general_h)if val==['?','?','?','?','?','?']]
    for i in indices:
        general_h.remove(['?','?','?','?','?','?'])
    return specific_h,general_h
s_final, g_final = learn(concepts, target)
print("\nFinal Specific_h:", s_final, sep="\n")
```

```
print("\nFinal General_h:", g_final, sep="\n")
```

7. OUTPUT:

```
Final Specific_h:
['sunny' 'warm' '?' 'strong' '?' '?']

Final General_h:
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]
```

OR

8. VARIANT-2:

```
import csv

with open('D:/enjoysport.csv')as f:
    csv_file=csv.reader(f)
    data=list(csv_file)

    s=data[1][:-1]
    g=[['?' for i in range(len(s))] for j in range(len(s))]

    for i in data:
        if i[-1]=="Yes":
            for j in range(len(s)):
                if i[j]!=s[j]:
                    s[j]='?'
                    g[j][j]='?'

        elif i[-1]=="No":
            for j in range(len(s)):
                if i[j]!=s[j]:
                    g[j][j]=s[j]
                else:
                    g[j][j]="?"
```

```
            print("\nSteps of Candidate Elimination Algorithm",data.index(i)+1)
            print(s)
            print(g)
        gh=[]
        for i in g:
            for j in i:
                if j!='?':
                    gh.append(i)
                    break
        print("\nFinal specific hypothesis:\n",s)

print("\nFinal general hypothesis:\n",gh)
```

OR

9. VARIANT-3:

```
import csv
# open the CSVFile and keep all rows as list of tuples
with open('D:/enjoysport.csv') as csvFile:
    examples = [tuple(line) for line in csv.reader(csvFile)]
print(examples)
# To obtain the domain of attribute values defined in the instances X
def get_domains(examples):
    # set function returns the unordered collection of items with no duplicates
    d = [set() for i in examples[0]]
    for x in examples:
        #Enumerate() function adds a counter to an iterable and returns it in a form
of enumerate object i.e(index,value)
        for i, xi in enumerate(x):
            d[i].add(xi)
    return [list(sorted(x)) for x in d]
# Test the get_domains function
get_domains(examples)
# Repeat the '?' and '0' length of domain no of times
def g_0(n):
```

```
    return ('?',)*n

def s_0(n):
    return ('0',)*n
# Function to check generality between two hypothesis
def more_general(h1, h2):
    more_general_parts = []
    for x, y in zip(h1, h2):
        mg = x == '?' or (x != '0' and (x == y or y == '0'))
        more_general_parts.append(mg)
    return all(more_general_parts) # Returns true if all elements of list or tuple
are true

# Function to check whether train examples are consistent with hypothesis
def consistent(hypothesis,example):
    return more_general(hypothesis, example)
# Function to add min_generalizations
def min_generalizations(h, x):
    h_new = list(h)
    for i in range(len(h)):
        if not consistent(h[i:i+1],x[i:i+1]):
            if h[i] != '0':
                h_new[i] = '?'
            else:
                h_new[i] = x[i]
    return [tuple(h_new)]

# Function to generalize Specific hypto
def generalize_S(x, G, S):
    S_prev = list(S)
    for s in S_prev:
        if s not in S:
            continue
        if not consistent(s,x):
            S.remove(s)
```

```
        Splus = min_generalizations(s, x)
        # Keep only generalizations that have a counterpart in G
        S.update([h for h in Splus if any([more_general(g,h)
                                           for g in G])])
        # Remove from S any hypothesis more general than any other
hypothesis in S
        S.difference_update([h for h in S if
                             any([more_general(h, h1)
                                  for h1 in S if h != h1])])
    return S
# Function to add min_specializations
def min_specializations(h, domains, x):
    results = []
    for i in range(len(h)):
        if h[i] == '?':
            for val in domains[i]:
                if x[i] != val:
                    h_new = h[:i] + (val,) + h[i+1:]
                    results.append(h_new)
        elif h[i] != '0':
            h_new = h[:i] + ('0',) + h[i+1:]
            results.append(h_new)
    return results

# Function to specialize General hypotheses boundary
def specialize_G(x, domains, G, S):
    G_prev = list(G)
    for g in G_prev:
        if g not in G:
            continue
        if consistent(g,x):
            G.remove(g)
            Gminus = min_specializations(g, domains, x)
            # Keep only specializations that have a counterpart in S
            G.update([h for h in Gminus if any([more_general(h, s)
```

```
                                    for s in S])])
            # Remove hypothesis less general than any other hypothesis in G
            G.difference_update([h for h in G if
                                 any([more_general(g1, h)
                                      for g1 in G if h != g1])])
    return G
# Function to perform CandidateElimination
def candidate_elimination(examples):
    domains = get_domains(examples)[:-1]

    G = set([g_0(len(domains))])
    S = set([s_0(len(domains))])
    i=0
    print('All the hypotheses in General and Specific boundary are:\n')
    print('\n G[{0}]:'.format(i),G)
    print('\n S[{0}]:'.format(i),S)
    for xcx in examples:
        i=i+1
        x, cx = xcx[:-1], xcx[-1]  # Splitting data into attributes and decisions
        if cx=='Yes': # x is positive example
            G = {g for g in G if consistent(g,x)}
            S = generalize_S(x, G, S)
        else: # x is negative example
            S = {s for s in S if not consistent(s,x)}
            G = specialize_G(x, domains, G, S)

        print('\n G[{0}]:'.format(i),G)
        print('\n S[{0}]:'.format(i),S)
    return

#import pixiedust # Visual debugger
#%%pixie_debugger
candidate_elimination(examples)
```

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

1. EXPERIMENT NO: 4

2. TITLE: 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

3. THEORY:

- ID3 algorithm is a basic algorithm that learns decision trees by constructing them topdown, beginning with the question "which attribute should be tested at the root of the tree?".
- To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples. The best attribute is selected and used as the test at the root node of the tree.
- A descendant of the root node is then created for each possible value of this attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute).
- The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree.
- A simplified version of the algorithm, specialized to learning boolean-valued functions (i.e.,concept learning), is described below.

4. ALGORITHM:

ID3 (Examples, Target_Attribute, Attributes)
Create a root node for the tree
If all examples are positive, Return the single-node tree Root, with label = +.
If all examples are negative, Return the single-node tree Root, with label = -.
If number of predicting attributes is empty, then Return the single node tree Root,
with label = most common value of the target attribute in the examples.
Otherwise Begin
A ← The Attribute that best classifies examples.
Decision Tree attribute for Root = A.

For each possible value, v_i, of A,
Add a new tree branch below Root, corresponding to the test A = v_i.
Let Examples (v_i) be the subset of examples that have the value v_i for A.
If Example v_i is empty
Then below this new branch add a leaf node with label = most common target value in the examples
Else below this new branch add the subtree ID3 (Examples(v_i), Target_Attribute, Attributes – {A})
End
Return Root

5. DATASET:

Outlook	Temperature	Humidity	Wind	Target
sunny	hot	high	weak	no
sunny	hot	high	strong	no
overcast	hot	high	weak	yes
rain	mild	high	weak	yes
rain	cool	normal	weak	yes
rain	cool	normal	strong	no
overcast	cool	normal	strong	yes
sunny	mild	high	weak	no
sunny	cool	normal	weak	yes
rain	mild	normal	weak	yes
sunny	mild	normal	strong	yes
overcast	mild	high	strong	yes
overcast	hot	normal	weak	yes
rain	mild	high	strong	no

6. SOLUTION:

```
import pandas as pd
import math
import numpy as np

data = pd.read_csv("D:/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)

        root = ID3(data, features)
printTree(root)
```

7. OUTPUT:

```
outlook
        overcast ->  ['yes']

        rain
                wind
                        strong ->  ['no']

                        weak ->  ['yes']

        sunny
                humidity
                        high ->  ['no']

                        normal ->  ['yes']
```

OR

8. VARIANT-2:

```
import numpy as np
import math
import csv

def read_data(filename):
    with open(filename, 'r') as csvfile:
        datareader = csv.reader(csvfile, delimiter=',')
        headers = next(datareader)
        metadata = []
        traindata = []
        for name in headers:
            metadata.append(name)
        for row in datareader:
            traindata.append(row)

    return (metadata, traindata)

class Node:
    def __init__(self, attribute):
        self.attribute = attribute
        self.children = []
        self.answer = ""

    def __str__(self):
        return self.attribute

def subtables(data, col, delete):
    dict = {}
    items = np.unique(data[:, col])
    count = np.zeros((items.shape[0], 1), dtype=np.int32)

    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
```

```
                count[x] += 1

        for x in range(items.shape[0]):
            dict[items[x]] = np.empty((int(count[x]), data.shape[1]),
    dtype="|S32")
            pos = 0
            for y in range(data.shape[0]):
                if data[y, col] == items[x]:
                    dict[items[x]][pos] = data[y]
                    pos += 1
            if delete:
                dict[items[x]] = np.delete(dict[items[x]], col, 1)

        return items, dict

    def entropy(S):
        items = np.unique(S)

        if items.size == 1:
            return 0

        counts = np.zeros((items.shape[0], 1))
        sums = 0

        for x in range(items.shape[0]):
            counts[x] = sum(S == items[x]) / (S.size * 1.0)

        for count in counts:
            sums += -1 * count * math.log(count, 2)
        return sums

    def gain_ratio(data, col):
        items, dict = subtables(data, col, delete=False)

        total_size = data.shape[0]
```

```
        entropies = np.zeros((items.shape[0], 1))
        intrinsic = np.zeros((items.shape[0], 1))

        for x in range(items.shape[0]):
            ratio = dict[items[x]].shape[0]/(total_size * 1.0)
            entropies[x] = ratio * entropy(dict[items[x]][:, -1])
            intrinsic[x] = ratio * math.log(ratio, 2)

        total_entropy = entropy(data[:, -1])
        iv = -1 * sum(intrinsic)

        for x in range(entropies.shape[0]):
            total_entropy -= entropies[x]

        return total_entropy / iv

    def create_node(data, metadata):
        if (np.unique(data[:, -1])).shape[0] == 1:
            node = Node("")
            node.answer = np.unique(data[:, -1])[0]
            return node

        gains = np.zeros((data.shape[1] - 1, 1))

        for col in range(data.shape[1] - 1):
            gains[col] = gain_ratio(data, col)

        split = np.argmax(gains)

        node = Node(metadata[split])
        metadata = np.delete(metadata, split, 0)

        items, dict = subtables(data, split, delete=True)

        for x in range(items.shape[0]):
```

```
            child = create_node(dict[items[x]], metadata)
            node.children.append((items[x], child))

        return node

    def empty(size):
        s = ""
        for x in range(size):
            s += "   "
        return s

    def print_tree(node, level):
        if node.answer != "":
            print(empty(level), node.answer)
            return
        print(empty(level), node.attribute)
        for value, n in node.children:
            print(empty(level + 1), value)
            print_tree(n, level + 2)

    metadata, traindata = read_data("D:/ID3-dataset.csv")
    data = np.array(traindata)
    node = create_node(data, metadata)
print_tree(node, 0)
```

OR

9. VARIANT-3:

```
import pandas as pd
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.externals.six import StringIO
```

```
data = pd.read_csv("D:/ID3-dataset.csv")
print("The first 5 values of data is \n",data.head())

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


le_outlook = LabelEncoder()
X.Outlook =  le_outlook.fit_transform(X.Outlook)
le_Temperature = LabelEncoder()
X.Temperature =  le_Temperature.fit_transform(X.Temperature)
le_Humidity = LabelEncoder()
X.Humidity =  le_Humidity.fit_transform(X.Humidity)
le_Windy = LabelEncoder()
X.Windy =  le_Windy.fit_transform(X.Windy)

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

le_PlayTennis = LabelEncoder()
y =  le_PlayTennis.fit_transform(y)
print("\nNow the Train data is\n",y)

classifier = DecisionTreeClassifier()
classifier.fit(X,y)

def labelEncoderForInput(list1):
    list1[0] =  le_outlook.transform([list1[0]])[0]
    list1[1] =  le_Temperature.transform([list1[1]])[0]
    list1[2] =  le_Humidity.transform([list1[2]])[0]
    list1[3] =  le_Windy.transform([list1[3]])[0]
    return [list1]
```

```
inp = ["Rainy","Mild","High","False"]
inp1=["Rainy","Cool","High","False"]
pred1 = labelEncoderForInput(inp1)
y_pred = classifier.predict(pred1)
y_pred
print("\nfor input {0}, we obtain {1}".format(inp1,
le_PlayTennis.inverse_transform(y_pred[0])))
```

1. EXPERIMENT NO: 5

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

3. THEORY:

- Artificial neural networks (ANNs) provide a general, practical method for learning real- valued, discrete-valued, and vector-valued functions from examples.
- Algorithms such as BACKPROPAGATION gradient descent to tune network parameters tobest fit a training set of input-output pairs.
- ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot controlstrategies.

4. ALGORITHM:

1. Create a feed-forward network with ni inputs, nhidden hidden units, and nout output units.
2. Initialize each wi to some small random value (e.g., between -.05 and .05).
3. Until the termination condition is met, do
   For each training example <(x1,…xn),t>, do
   // Propagate the input forward through the network:
   a. Input the instance (x1, ..,xn) to the n/w & compute the n/w outputs ok for every unit
   // Propagate the errors backward through the network:
   b. For each output unit k, calculate its error term δk ; $o_k = o_k(1-o_k)(t_k-o_k)$
   c. For each hidden unit h, calculate its error term oh; $o_h=o_h(1-o_h) \Sigma_k w_{h,k} \delta k$

d. For each network weight wi,j do; wi,j=wi,j+Δwi,j where Δwi,j= η δj xi,j

5. SOLUTION:

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0)      # maximum of X array longitudinally
y=y/100
#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))
#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)
#Variable initialization
epoch = 5000                    #Setting training iterations
lr = 0.1                    #Setting learning rate
inputlayer_neurons = 2         #number of features in data set
hiddenlayer_neurons = 3       #number of hidden layers neurons
output_neurons = 1              #number of neurons at output layer
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
```

```
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
#Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)
#Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)
#how much hidden layer wts contributed to error
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad
# dotproduct of nextlayererror and currentlayerop
    wout += hlayer_act.T.dot(d_output) *lr
```

```
    wh += X.T.dot(d_hiddenlayer) *lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)
```

6. OUTPUT:

```
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.89504121]
 [0.88252504]
 [0.8926262 ]]
```

OR

7. VARIANT-2:

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)

X = X/np.amax(X, axis=0)
y = y/100

class Neural_Network(object):
    def __init__(self):
```

```
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

        self.W1 = np.random.randn(self.inputSize, self.hiddenSize)
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)

    def forward(self, X):

        self.z = np.dot(X, self.W1)
        self.z2 = self.sigmoid(self.z)
        self.z3 = np.dot(self.z2, self.W2)
        o = self.sigmoid(self.z3)
        return o

    def sigmoid(self, s):
        return 1/(1+np.exp(-s))

    def sigmoidPrime(self, s):
        return s * (1 - s)

    def backward(self, X, y, o):

        self.o_error = y - o        # error in output
        self.o_delta = self.o_error*self.sigmoidPrime(o)
        self.z2_error = self.o_delta.dot(self.W2.T)
        self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2)
        self.W1 += X.T.dot(self.z2_delta)
        self.W2 += self.z2.T.dot(self.o_delta)

    def train (self, X, y):
        o = self.forward(X)
        self.backward(X, y, o)

NN = Neural_Network()
```

```
print ("\nInput: \n" + str(X))
print ("\nActual Output: \n" + str(y))
print ("\nPredicted Output: \n" + str(NN.forward(X)))
print ("\nLoss: \n" + str(np.mean(np.square(y - NN.forward(X)))))
NN.train(X, y)
```

OR

8. VARIANT-3:

```
import numpy as np

class NeuralNetwork:

    def __init__(self, input_layer_size, hidden_layer_size, output_layer_size,
learning_rate):
        self.input_layer_size = input_layer_size
        self.hidden_layer_size = hidden_layer_size
        self.output_layer_size = output_layer_size
        self.learning_rate = learning_rate

        self.weights1 = np.random.randn(self.input_layer_size,
self.hidden_layer_size)
        self.weights2 = np.random.randn(self.hidden_layer_size,
self.output_layer_size)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward(self, X):
        self.layer1 = self.sigmoid(np.dot(X, self.weights1))
        self.layer2 = self.sigmoid(np.dot(self.layer1, self.weights2))
```

```
        return self.layer2

    def backward(self, X, y, output):
        self.output_error = y - output
        self.output_delta = self.output_error * self.sigmoid_derivative(output)

        self.layer1_error = self.output_delta.dot(self.weights2.T)
        self.layer1_delta = self.layer1_error * self.sigmoid_derivative(self.layer1)

        self.weights1 += X.T.dot(self.layer1_delta) * self.learning_rate
        self.weights2 += self.layer1.T.dot(self.output_delta) * self.learning_rate

    def train(self, X, y, num_epochs):
        for epoch in range(num_epochs):
            output = self.forward(X)
            self.backward(X, y, output)

    def predict(self, X):
        return self.forward(X)
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the 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=0.2,
random_state=42)

# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)

# Initialize the neural network
nn = NeuralNetwork(input_layer_size=X.shape[1], hidden_layer_size=4,
output_layer_size=len(np.unique(y)), learning_rate=0.1)

# Train the neural network
nn.train(X_train, np.eye(len(np.unique(y)))[y_train], num_epochs=1000)

# Make predictions on the testing set
y_pred = np.argmax(nn.predict(X_test), axis=1)

# Evaluate the performance of the model
accuracy = np.mean(y_pred == y_test)
print("Accuracy: {:.2f}%".format(accuracy * 100))
```

1. EXPERIMENT NO: 6

2. TITLE: Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

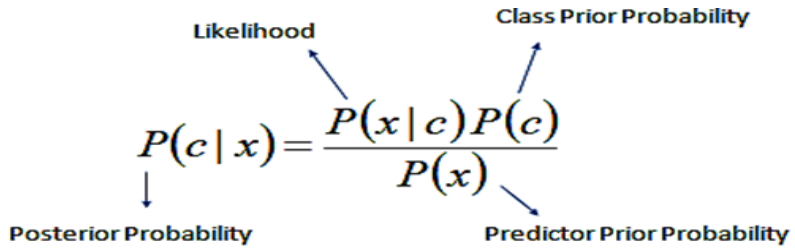
3. THEORY & ALGOTITHM:

Naive Bayes algorithm is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly

sophisticated classification methods.

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c).Look at the equation below:



$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

where

P(c|x) is the posterior probability of class (c, target) given predictor (x, attributes).
P(c) is the prior probability of class.
P(x|c) is the likelihood which is the probability of predictor given class.
P(x) is the prior probability of predictor.

The naive Bayes classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target function f (x) can take on any value from some finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values (a1, a2, ... ,an). The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach to classifying the new instance is to assign the most probable target value,

v_{MAP}, given the attribute values (a_1, a_2, ..., a_n) that describe the instance.

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

We can use Bayes theorem to rewrite this expression as

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

$$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j)$$

Now we could attempt to estimate the two terms in Equation based on the training data. It is easy to estimate each of the P(vj) simply by counting the frequency with which each target value vj occurs in the training data.

The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the conjunction $a_1, a_2, \ldots, a_n$, is just the product of the probabilities for the individual attributes: $P(a_1, a_2, \ldots, a_n \mid v_j) = \Pi_i P(a_i \mid v_j)$. Substituting this, we have the approach used by the naive Bayes classifier.

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j)$$

where v_{NB} denotes the target value output by the naive Bayes classifier.

When dealing with continuous data, a typical assumption is that the continuous values associated with each class are distributed according to a Gaussian distribution. For example, suppose the training data contains a continuous attribute, x. We first segment the data by the class, and then compute the mean and variance of x in each class.

Let μ be the mean of the values in x associated with class Ck, and let σ^2k be the variance of the values in x associated with class Ck. Suppose we have collected some observation value v. Then, the probability distribution of v given a class Ck, p(x=v|Ck) can be computed by plugging v into the equation for a Normal distribution parameterized by μ and σ^2k . *That is*

$$p(x = v \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(v-\mu_k)^2}{2\sigma_k^2}}$$

4. DATASET:

Outlook	Temperature	Humidity	Wind	Target
sunny	hot	high	weak	no
sunny	hot	high	strong	no
overcast	hot	high	weak	yes
rain	mild	high	weak	yes
rain	cool	normal	weak	yes
rain	cool	normal	strong	no
overcast	cool	normal	strong	yes
sunny	mild	high	weak	no
sunny	cool	normal	weak	yes
rain	mild	normal	weak	yes
sunny	mild	normal	strong	yes
overcast	mild	high	strong	yes
overcast	hot	normal	weak	yes
rain	mild	high	strong	no

5. SOLUTION:

```
import numpy as np

import math

import csv

def read_data(filename):
```

```
    with open(filename,'r') as csvfile:
        datareader = csv.reader(csvfile)
        metadata = next(datareader)
        traindata=[]
        for row in datareader:
            traindata.append(row)
    return (metadata, traindata)
def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    testset = list(dataset)
    i=0
    while len(trainSet) < trainSize:
        trainSet.append(testset.pop(i))
    return [trainSet, testset]
def classify(data,test):
    total_size = data.shape[0]
    print("training data size=",total_size)
    print("test data size=",test.shape[0])
    countYes = 0
    countNo = 0
    probYes = 0
```

```
probNo = 0
print("target    count    probability")
for x in range(data.shape[0]):
    if data[x,data.shape[1]-1] == 'yes':
        countYes +=1
    if data[x,data.shape[1]-1] == 'no':
        countNo +=1
probYes=countYes/total_size
probNo= countNo / total_size
print('Yes',"\t",countYes,"\t",probYes)
print('No',"\t",countNo,"\t",probNo)
prob0 =np.zeros((test.shape[1]-1))
prob1 =np.zeros((test.shape[1]-1))
accuracy=0
print("instance prediction  target")
for t in range(test.shape[0]):
    for k in range (test.shape[1]-1):
        count1=count0=0
        for j in range (data.shape[0]):
            #how many times appeared with no
            if test[t,k] == data[j,k] and data[j,data.shape[1]-1]=='no':
                count0+=1
```

```
            #how many times appeared with yes
            if test[t,k]==data[j,k] and data[j,data.shape[1]-1]=='yes':
                count1+=1
        prob0[k]=count0/countNo
        prob1[k]=count1/countYes
    probno=probNo
    probyes=probYes
    for i in range(test.shape[1]-1):
        probno=probno*prob0[i]
        probyes=probyes*prob1[i]
    if probno>probyes:
        predict='no'
    else:
        predict='yes'
    print(t+1,"\t",predict,"\t    ",test[t,test.shape[1]-1])
    if predict == test[t,test.shape[1]-1]:
        accuracy+=1
final_accuracy=(accuracy/test.shape[0])*100
print("accuracy",final_accuracy,"%")
return
metadata,traindata= read_data("d:/dataset.csv")
splitRatio=0.6
```

```
trainingset, testset=splitDataset(traindata, splitRatio)

training=np.array(trainingset)

testing=np.array(testset)

classify(training,testing)
```

6. OUTPUT:

```
training data size= 8
test data size= 6
target    count    probability
Yes       4        0.5
No        4        0.5
instance prediction  target
1        no          yes
2        yes         yes
3        no          yes
4        yes         yes
5        yes         yes
6        no          no
accuracy 66.66666666666666 %
```

OR

7. VARIANT-2:

```
# import necessary libarities

import pandas as pd

from sklearn import tree

from sklearn.preprocessing import LabelEncoder

from sklearn.naive_bayes import GaussianNB

# load data from CSV
```

```
data = pd.read_csv("d:/dataset.csv")
print("THe first 5 values of data is :\n",data.head())
# obtain Train data and Train output
X = data.iloc[:,:-1]
print("\nThe First 5 values of train data is\n",X.head())
y = data.iloc[:,-1]
print("\nThe first 5 values of Train output is\n",y.head())
# Convert then in numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)
le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)
le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)
le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)
print("\nNow the Train data is :\n",X.head())
le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n",y)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.20)
```

```
classifier = GaussianNB()

classifier.fit(X_train,y_train)

from sklearn.metrics import accuracy_score

print("Accuracy is:",accuracy_score(classifier.predict(X_test),y_test))
```

OR

8. VARIANT-3:

```
import csv import random import math

#1.Load Data
def loadCsv(filename):
lines = csv.reader(open(filename, "rt")) dataset = list(lines)
for i in range(len(dataset)):
dataset[i] = [float(x) for x in dataset[i]] return dataset

#Split the data into Training and Testing randomly
def splitDataset(dataset, splitRatio):
trainSize = int(len(dataset) * splitRatio) trainSet = []
copy = list(dataset)
while len(trainSet) < trainSize:
index = random.randrange(len(copy)) trainSet.append(copy.pop(index))
return [trainSet, copy]

#Seperatedata by Class
def separateByClass(dataset): separated = {}
for i in range(len(dataset)): vector = dataset[i]
if (vector[-1] not in separated): separated[vector[-1]] = []
separated[vector[-1]].append(vector) return separated
#Calculate Mean
def mean(numbers):
return sum(numbers)/float(len(numbers))
```

```
#Calculate Standard Deviation
def stdev(numbers):
avg = mean(numbers)
variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1) return
math.sqrt(variance)

#Summarize the data
def summarize(dataset):
summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
del summaries[-1]

return summaries

#Summarize Attributes by Class
def summarizeByClass(dataset): separated = separateByClass(dataset)
print(len(separated))
summaries = {}
for classValue, instances in separated.items(): summaries[classValue] =
summarize(instances)
print(summaries) return summaries

#Calculate Gaussian Probability Density Function
def calculateProbability(x, mean, stdev):
exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2)))) return (1 /
(math.sqrt(2*math.pi) * stdev)) * exponent

#Calculate Class Probabilities
def calculateClassProbabilities(summaries, inputVector): probabilities = {}
for classValue, classSummaries in summaries.items(): probabilities[classValue]
= 1
for i in range(len(classSummaries)): mean, stdev = classSummaries[i] x =
inputVector[i]
probabilities[classValue] *= calculateProbability(x, mean, stdev) return
probabilities
```

```
#Make a Prediction
def predict(summaries, inputVector):
probabilities = calculateClassProbabilities(summaries, inputVector) bestLabel,
bestProb = None, -1
for classValue, probability in probabilities.items():
if bestLabel is None or probability > bestProb: bestProb = probability
bestLabel = classValue return bestLabel

#return a list of predictions for each test instance.
def getPredictions(summaries, testSet): predictions = []
for i in range(len(testSet)):
result = predict(summaries, testSet[i]) predictions.append(result)
return predictions

#calculate accuracy ratio.
def getAccuracy(testSet, predictions): correct = 0
for i in range(len(testSet)):
if testSet[i][-1] == predictions[i]: correct += 1
return (correct/float(len(testSet))) * 100.0

filename = 'd:/dataset.csv' splitRatio = 0.70

dataset = loadCsv(filename)
trainingSet, testSet = splitDataset(dataset, splitRatio)
print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),
len(trainingSet), len(testSet)))
# prepare model
summaries = summarizeByClass(trainingSet)

# test model
predictions = getPredictions(summaries, testSet)
accuracy = getAccuracy(testSet, predictions)
print('Accuracy: {0}%'.format(accuracy))
```

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

1. EXPERIMENT NO: 7

2. TITLE: Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

3. THEORY & ALGORITHM:

Expectation Maximization algorithm

- The basic approach and logic of this clustering method is as follows.
- Suppose we measure a single continuous variable in a large sample of observations. Further, suppose that the sample consists of two clusters of observations with different means (and perhaps different standard deviations); within each sample, the distribution of values for the continuous variable follows the normal distribution.
- The goal of EM clustering is to estimate the means and standard deviations for each cluster so as to maximize the likelihood of the observed data (distribution).
- Put another way, the EM algorithm attempts to approximate the observed distributions of values based on mixtures of different distributions in different clusters. The results of EM clustering are different from those computed by k-means clustering.
- The latter will assign observations to clusters to maximize the distances between clusters. The EM algorithm does not compute actual assignments of observations to clusters, but classification probabilities.
- In other words, each observation belongs to each cluster with a certain probability. Of course, as a final result we can usually review an actual assignment of observations to clusters, based on the (largest) classification probability.

K means Clustering

- The algorithm will categorize the items into k groups of similarity. To calculate that similarity, we will use the euclidean distance as measurement.
- The algorithm works as follows:
  1. First we initialize k points, called means, randomly.

2. We categorize each item to its closest mean and we update the mean's coordinates, which are the averages of the items categorized in that mean so far.
3. We repeat the process for a given number of iterations and at the end, we have our clusters.

- The "points" mentioned above are called means, because they hold the mean values of the items categorized in it. To initialize these means, we have a lot of options. An intuitive method is to initialize the means at random items in the data set. Another method is to initialize the means at random values between the boundaries of the data set (if for a feature x the items have values in [0,3], we will initialize the means with values for x at [0,3]).

Pseudocode:

1. Initialize k means with random values
2. For a given number iterations:
    Iterate through items:
        Find the mean closest to the item
        Assign item to mean
        Update mean

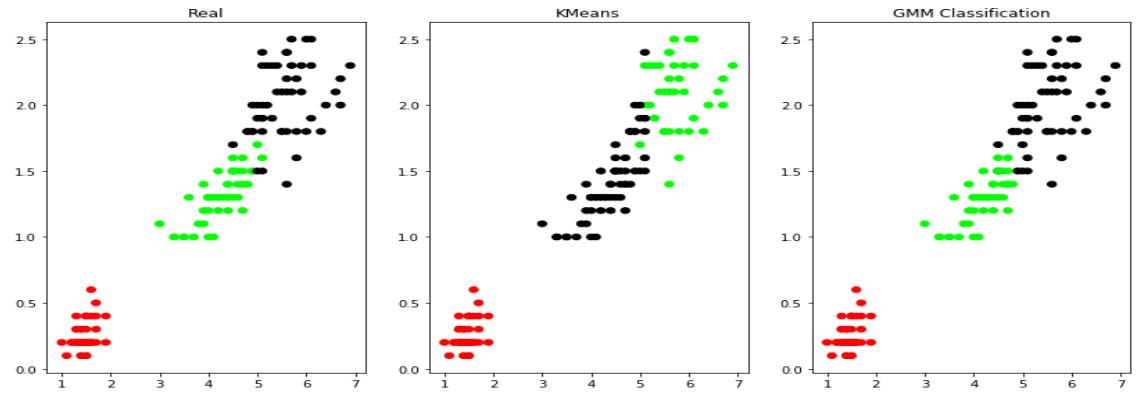
4. SOLUTION:

```
from sklearn.cluster import KMeans
from sklearn import preprocessing
from sklearn.mixture import GaussianMixture
from sklearn.datasets import load_iris
import sklearn.metrics as sm
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

dataset=load_iris()
# print(dataset)
```

```
X=pd.DataFrame(dataset.data)
X.columns=['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']
y=pd.DataFrame(dataset.target)
y.columns=['Targets']
# print(X)

plt.figure(figsize=(14,7))
colormap=np.array(['red','lime','black'])

# REAL PLOT
plt.subplot(1,3,1)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.Targets],s=40)
plt.title('Real')

# K-PLOT
plt.subplot(1,3,2)
model=KMeans(n_clusters=3)
model.fit(X)
predY=np.choose(model.labels_,[0,1,2]).astype(np.int64)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[predY],s=40)
plt.title('KMeans')

# GMM PLOT
scaler=preprocessing.StandardScaler()
scaler.fit(X)
xsa=scaler.transform(X)
xs=pd.DataFrame(xsa,columns=X.columns)
gmm=GaussianMixture(n_components=3)
gmm.fit(xs)
y_cluster_gmm=gmm.predict(xs)
plt.subplot(1,3,3)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y_cluster_gmm],s=40)
plt.title('GMM Classification')
```

5. OUTPUT:

```
Text(0.5, 1.0, 'GMM Classification')
```



OR

6. VARIANT-2:

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
import pandas as pd
import numpy as np
%matplotlib inline
# import some data to play with
iris = datasets.load_iris()
#print("\n IRIS DATA :",iris.data);
#print("\n IRIS FEATURES :\n",iris.feature_names)
#print("\n IRIS TARGET :\n",iris.target)
```

```
#print("\n IRIS TARGET NAMES:\n",iris.target_names)
# Store the inputs as a Pandas Dataframe and set the column names
X = pd.DataFrame(iris.data)
#print(X)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']
#print(X.columns)
#print("X:",x)
#print("Y:",y)
y = pd.DataFrame(iris.target)
y.columns = ['Targets']
# Set the size of the plot
plt.figure(figsize=(14,7))
 # Create a colormap
colormap = np.array(['red', 'lime', 'black'])
# Plot Sepal
plt.subplot(1, 2, 1)
plt.scatter(X.Sepal_Length,X.Sepal_Width, c=colormap[y.Targets], s=40)
plt.title('Sepal')
plt.subplot(1, 2, 2)
plt.scatter(X.Petal_Length,X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Petal')
```

```
# K Means Cluster
model = KMeans(n_clusters=3)
model.fit(X)
# This is what KMeans thought
model.labels_
# View the results
# Set the size of the plot
plt.figure(figsize=(14,7))
# Create a colormap
colormap = np.array(['red', 'lime', 'black'])
# Plot the Original Classifications
plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Classification')
# Plot the Models Classifications
plt.subplot(1, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K Mean Classification')
# The fix, we convert all the 1s to 0s and 0s to 1s.
predY = np.choose(model.labels_, [0, 1, 2]).astype(np.int64)
print (predY)
# View the results
```

```
# Set the size of the plot
plt.figure(figsize=(14,7))
# Create a colormap
colormap = np.array(['red', 'lime', 'black'])
# Plot Orginal
plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Classification')
# Plot Predicted with corrected values
plt.subplot(1, 2, 2)
plt.scatter(X.Petal_Length,X.Petal_Width, c=colormap[predY], s=40)
plt.title('K Mean Classification')
sm.accuracy_score(y, model.labels_)
# Confusion Matrix
sm.confusion_matrix(y, model.labels_)
from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
xs.sample(5)
from sklearn.mixture import GaussianMixture
```

```
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
y_cluster_gmm = gmm.predict(xs)
y_cluster_gmm
plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')
sm.accuracy_score(y, y_cluster_gmm)
# Confusion Matrix
sm.confusion_matrix(y, y_cluster_gmm)
```

OR

7. VARIANT-3:

```
#Importing Important Libraries
from numpy import hstack
from numpy.random import normal
import matplotlib.pyplot as plt

#Defining samples from processes
sample1 = normal(loc=20, scale=5 , size=4000)
sample2 = normal(loc=40, scale=5 , size=8000)
sample = hstack((sample1,sample2))
plt.hist(sample, bins=50, density=True)
```

```
#Showing the data points
plt.show()
# Gaussian mixture model with expectation maximization
from numpy import hstack
from numpy.random import normal
#Importing Gaussian mixture model
from sklearn.mixture import GaussianMixture
# generate a sample
sample1 = normal(loc=20, scale=5, size=4000)
sample2 = normal(loc=40, scale=5, size=8000)
sample = hstack((sample1, sample2))
# reshape into a table with one column to fit the data
sample = sample.reshape((len(sample), 1))
# training the model
model = GaussianMixture(n_components=2, init_params='random')
model.fit(sample)
# predict latent values
yhat = model.predict(sample)
# check latent value for first few points
print(yhat[:80])
# check latent value for last few points
print(yhat[-80:])
```

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

1. EXPERIMENT NO: 8

2. TITLE: Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

3. THEORY:

- K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.
- It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data.

4. ALGORITHM:

Input: Let m be the number of training data samples. Let p be an unknown point. Method:

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. for i=0 to m
   Calculate Euclidean distance d(arr[i], p).
3. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
4. Return the majority label among S.

5. SOLUTION:

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import numpy as np

dataset=load_iris()
X_train,X_test,y_train,y_test=train_test_split(dataset["data"],dataset["target"
],random_state=0)
kn=KNeighborsClassifier(n_neighbors=1)
kn.fit(X_train,y_train)
```

```
for i in range(len(X_test)):
    x=X_test[i]
    x_new=np.array([x])
    prediction=kn.predict(x_new)
    print("TARGET=",y_test[i],dataset["target_names"]
[y_test[i]],"PREDICTED=",prediction,dataset["target_names"][prediction])

print(kn.score(X_test,y_test))
```

6. OUTPUT:

```
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [2] ['virginica']
0.9736842105263158
```

OR

8. VARIANT-2:

```
import sklearn

import pandas as pd

from sklearn.datasets import load_iris

iris=load_iris()

iris.keys()
```

```
df=pd.DataFrame(iris['data'])
print(df)
print(iris['target_names'])
iris['feature_names']
X=df
y=iris['target']
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=42)
from sklearn.neighbors import KNeighborsClassifier
knn=KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train,y_train)
import numpy as np
x_new=np.array([[5,2.9,1,0.2]])
prediction=knn.predict(x_new)
iris['target_names'][prediction]
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
y_pred=knn.predict(X_test)
cm=confusion_matrix(y_test,y_pred)
print(cm)
```

```
print(" correct predicition",accuracy_score(y_test,y_pred))
```

OR

8. VARIANT-3:

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.model_selection import train_test_split
iris_dataset=load_iris()

print("\n IRIS FEATURES \ TARGET NAMES: \n ", iris_dataset.target_names)
for i in range(len(iris_dataset.target_names)):
    print("\n[{0}]:[{1}]".format(i,iris_dataset.target_names[i]))

print("\n IRIS DATA :\n",iris_dataset["data"])

X_train, X_test, y_train, y_test = train_test_split(iris_dataset["data"],
iris_dataset["target"], random_state=0)

print("\n Target :\n",iris_dataset["target"])
print("\n X TRAIN \n", X_train)
print("\n X TEST \n", X_test)
print("\n Y TRAIN \n", y_train)
print("\n Y TEST \n", y_test)
kn = KNeighborsClassifier(n_neighbors=1)
kn.fit(X_train, y_train)
prediction = kn.predict(x_new)

print("\n Predicted target value: {}\n".format(prediction))
print("\n Predicted feature name: {}\n".format
    (iris_dataset["target_names"][prediction]))

i=1
```

```
x= X_test[i]
x_new = np.array([x])
print("\n XNEW \n",x_new)

for i in range(len(X_test)):
    x = X_test[i]
    x_new = np.array([x])
    prediction = kn.predict(x_new)
    print("\n Actual : {0} {1}, Predicted
:{2}{3}".format(y_test[i],iris_dataset["target_names"][y_test[i]],prediction,iris_
dataset["target_names"][prediction]))

print("\n TEST SCORE[ACCURACY]: {:.2f}\n".format(kn.score(X_test,
y_test)))
```

1. EXPERIMENT NO: 9

2. TITLE: Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

3. THEORY:

- Given a dataset X, y, we attempt to find a linear model h(x) that minimizes residual sum ofsquared errors. The solution is given by Normal equations.
- Linear model can only fit a straight line, however, it can be empowered by polynomial features to get more powerful models. Still, we have to decide and fix the number and typesof features ahead.
- Alternate approach is given by locally weighted regression.
- Given a dataset X, y, we attempt to find a model h(x) that minimizes residual sum ofweighted squared errors.
- The weights are given by a kernel function which can be chosen arbitrarily and in my case Ichose a Gaussian kernel.
- The solution is very similar to Normal equations, we only need to insert diagonal weightmatrix W.

4. ALGORITHM:

```
def local_regression(x0, X, Y,
tau):
            # add bias term
            x0 = np.r_[1, x0]
            X = np.c_[np.ones(len(X)), X]

            # fit model: normal equations with
         kernel
           xw = X.T * radial_kernel(x0, X, tau)
           beta = np.linalg.pinv(xw @ X) @ xw @ Y
         # predict value
         return x0 @ beta
def radial_kernel(x0, X, tau):
                return np.exp(np.sum((X - x0) ** 2, axis=1) / (-2 * tau * tau))
```

5. DATASET:

Tips.csv(256 rows)

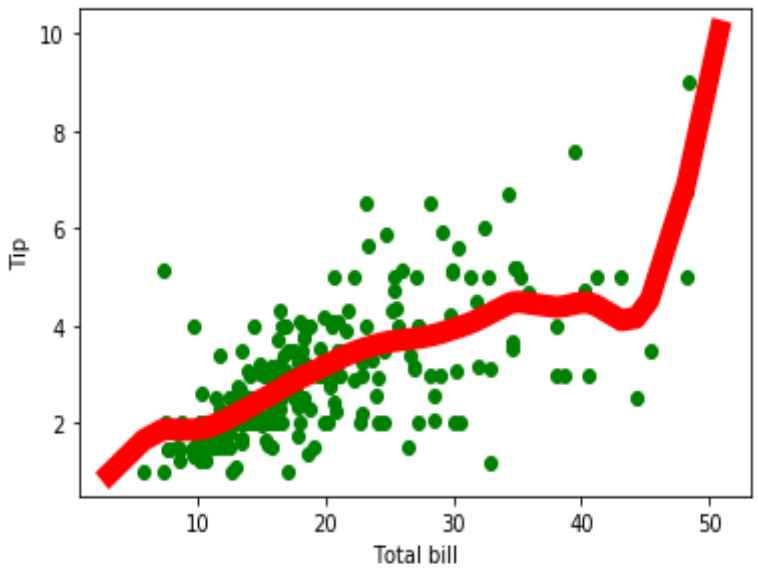
6. SOLUTION:

```
import matplotlib.pyplot as plt

import pandas as pd

import numpy as np


def kernel(point, xmat, k):

    m,n = np.shape(xmat)

    weights = np.mat(np.eye((m)))

    for j in range(m):

        diff = point - X[j]
```

```
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights
def localWeight(point, xmat, ymat, k):
    wei = kernel(point,xmat,k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W
def localWeightRegression(xmat, ymat, k):
    m,n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
    return ypred
# load data points
data = pd.read_csv('D:/Tips_dataset.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)
#preparing and add 1 in bill
mbill = np.mat(bill)
mtip = np.mat(tip)
m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T))
```

```
#set k here
ypred = localWeightRegression(X,mtip,2)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0]
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(bill,tip, color='green')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=10)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show();
```

7. OUTPUT:



OR

8. VARIANT-2:

```
from math import ceil
import numpy as np
from scipy import linalg

def lowess(x, y, f, iterations):
   n = len(x)
   r = int(ceil(f * n))
   h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]
   w = np.clip(np.abs((x[:, None] - x[None, :]) / h), 0.0, 1.0)
   w = (1 - w ** 3) ** 3
   yest = np.zeros(n)
   delta = np.ones(n)
   for iteration in range(iterations):
      for i in range(n):
         weights = delta * w[:, i]
         b = np.array([np.sum(weights * y), np.sum(weights * y * x)])
         A = np.array([[np.sum(weights), np.sum(weights * x)],[np.sum(weights
* x), np.sum(weights * x * x)]])
         beta = linalg.solve(A, b)
         yest[i] = beta[0] + beta[1] * x[i]

      residuals = y - yest
      s = np.median(np.abs(residuals))
      delta = np.clip(residuals / (6.0 * s), -1, 1)
      delta = (1 - delta ** 2) ** 2

   return yest

import math
n = 100
x = np.linspace(0, 2 * math.pi, n)
```

```
y = np.sin(x) + 0.3 * np.random.randn(n)
f =0.25
iterations=3
yest = lowess(x, y, f, iterations)

import matplotlib.pyplot as plt
plt.plot(x,y,"r.")
plt.plot(x,yest,"b-")
```

OR

9. VARIANT-3:

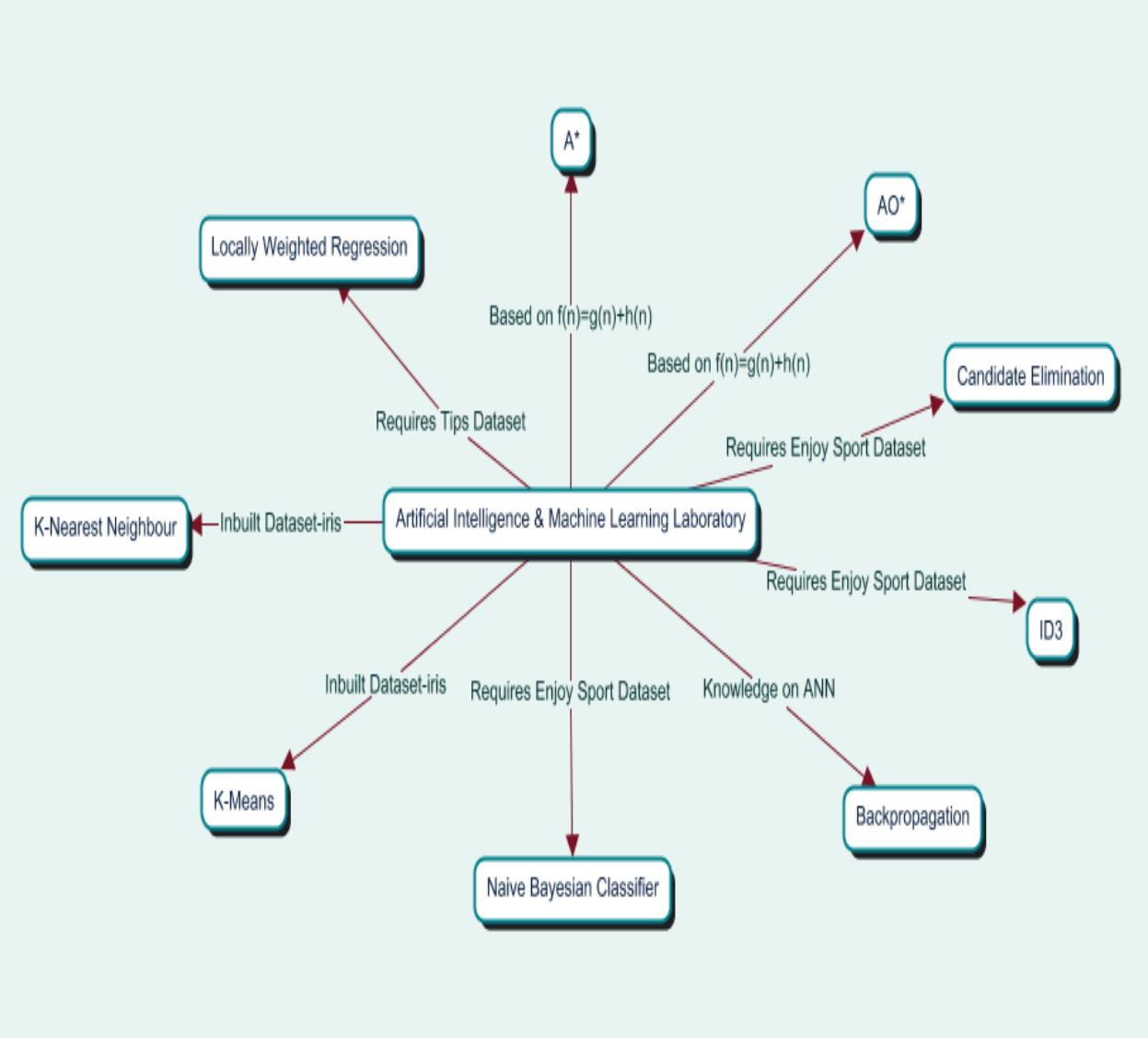
```
import numpy as np
import matplotlib.pyplot as plt
X=np.linspace(-3,3,1000)
print(X)
X+=np.random.normal(scale=0.05,size=1000)
Y=np.log(np.abs((X**2)-1)+0.5)
print(Y)
np.linspace(2.0, 3.0, num=5)
plt.scatter(X,Y,alpha=0.32)
def local_regression(x0,X,Y,tau):
    x0=np.r_[1,x0]
    # print(x0)
    #print("len(X)",len(X))
    X=np.c_[np.ones(len(X)),X]
    #print(X)
    xw=X.T *radial_kernel(x0,X,tau)
    print(xw)
    beta=np.linalg.pinv(xw@X)@xw@Y
    return x0@beta
```

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

VIVA QUESTIONS

- What is A* algorithm.
- What is the formula used in A* algorithm.
- What is AO* algorithm.
- What is the formula used in AO* algorithm.
- What is Artificial Intelligence.
- What is Machine Learning.
- Define Supervised learning.
- Define Unsupervised learning.
- Define Reinforcement learning.
- What is Classification.
- What is clustering.
- What do you mean by hypothesis.
- What is Gain.
- What is Entropy.
- How k-means clustering is different from KNN.
- State Bayes Theorem.
- Define Bias.
- What is learning rate? Why it is needed.
- Why KNN algorithm is known as lazy learning algorithm.
- What is K-means clustering algorithm.
- What are the advantages and disadvantages of K-means clustering algorithm.
- What is K in KNN algorithm.
- What are the advantages and disadvantages of KNN algorithm.
- What are the some applications of K-Means clustering.
- What is the difference between training data and test data.
- What is naïve in naïve bayes algorithm.

Concept Map


A*
AO*
Locally Weighted Regression
Based on f(n)=g(n)+h(n)
Based on f(n)=g(n)+h(n)
Candidate Elimination
Requires Tips Dataset
Requires Enjoy Sport Dataset
K-Nearest Neighbour
Inbuilt Dataset-iris
Artificial Intelligence & Machine Learning Laboratory
Requires Enjoy Sport Dataset
ID3
Inbuilt Dataset-iris
Requires Enjoy Sport Dataset
Knowledge on ANN
K-Means
Backpropagation
Naive Bayesian Classifier

Additional Programs for Practice

1. Tic-Tac-Toe Program using random number

```
# 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()))
```

2. 8-Puzzle Program

```
class EightPuzzleSolver:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.actions = []
    def solve(self):
        frontier = [(self.initial_state, self.actions)]
        explored = set()
        while frontier:
            current_state, current_actions = frontier.pop(0)
            if current_state == self.goal_state:
                self.actions = current_actions
```

```
            return True
        explored.add(current_state)
        for action, state in self.get_successors(current_state):
            if state not in explored:
                frontier.append((state, current_actions + [action]))
    return False
  def get_successors(self, state):
    successors = []
    zero_pos = state.index(0)
    row, col = divmod(zero_pos, 3)
    if row > 0:
        new_state = list(state)
        new_state[zero_pos], new_state[zero_pos - 3] = new_state[zero_pos - 3],
new_state[zero_pos]
        successors.append(('Up', tuple(new_state)))
    if row < 2:
        new_state = list(state)
        new_state[zero_pos], new_state[zero_pos + 3] = new_state[zero_pos + 3],
new_state[zero_pos]
        successors.append(('Down', tuple(new_state)))
    if col > 0:
        new_state = list(state)
        new_state[zero_pos], new_state[zero_pos - 1] = new_state[zero_pos - 1],
```

```
new_state[zero_pos]
            successors.append(('Left', tuple(new_state)))
        if col < 2:
            new_state = list(state)
            new_state[zero_pos], new_state[zero_pos + 1] = new_state[zero_pos + 1],
new_state[zero_pos]
            successors.append(('Right', tuple(new_state)))
        return successors
initial_state = (2, 8, 3, 1, 6, 4, 7, 0, 5)
goal_state = (1, 2, 3, 8, 0, 4, 7, 6, 5)
solver = EightPuzzleSolver(initial_state, goal_state)
solver.solve()
print(solver.actions)
```

3. Water-Jug Program

```
from collections import deque
def Solution(a, b, target):
            m = {}
            isSolvable = False
            path = []

            q = deque()

            #Initializing with jugs being empty
            q.append((0, 0))

            while (len(q) > 0):
```

```
# Current state
u = q.popleft()
if ((u[0], u[1]) in m):
        continue
if ((u[0] > a or u[1] > b or
        u[0] < 0 or u[1] < 0)):
        continue
path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target):
        isSolvable = True

        if (u[0] == target):
                if (u[1] != 0):
                        path.append([u[0], 0])
        else:
                if (u[0] != 0):

                        path.append([0, u[1]])

        sz = len(path)
        for i in range(sz):
                print("(", path[i][0], ",",
                        path[i][1], ")")
        break

q.append([u[0], b]) # Fill Jug2
q.append([a, u[1]]) # Fill Jug1

for ap in range(max(a, b) + 1):
        c = u[0] + ap
        d = u[1] - ap
```

```
                        if (c == a or (d == 0 and d >= 0)):
                                q.append([c, d])

                        c = u[0] - ap
                        d = u[1] + ap

                        if ((c == 0 and c >= 0) or d == b):
                                q.append([c, d])

                q.append([a, 0])

                q.append([0, b])

        if (not isSolvable):
                print("Solution not possible")

if __name__ == '__main__':

        Jug1, Jug2, target = 4, 3, 2
        print("Path from initial state "
                "to solution state ::")

        Solution(Jug1, Jug2, target)
```

4. Tower of Hanoi Program

```
def TowerOfHanoi(n , source, destination, auxiliary):
   if n==1:
      print ("Move disk 1 from source",source,"to destination",destination)
      return
   TowerOfHanoi(n-1, source, auxiliary, destination)
   print ("Move disk",n,"from source",source,"to destination",destination)
```

```
    TowerOfHanoi(n-1, auxiliary, destination, source)

    # Driver code
n = 4
TowerOfHanoi(n,'A','B','C')
```