TABLE OF CONTENTS

Sl.No	***Content***	***Page No.***
1.	Implement a A* search algorithm	1
2.	Implement a AO* search algorithm	5
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.	9
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.	24
5.	Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data set	36
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.	47
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.	56
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.	66
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.	72
10	Additional Lab Exercise	78

VISION OF THE DEPARTMENT

Impart magnificent learning atmosphere establishing innovative practices among the students aiming to strengthen their software application knowledge and technical skills.

MISSION OF THE DEPARTMENT

M1: To deliver quality technical training on software application domain.

M2: To nurture team work in order to transform individual as responsible leader and entrepreneur for future trends.

M3: To inculcate research practices in teaching thus ensuring research blend among students.

M4: To ensure more doctorates in the department, aiming at professional strength.

M5: To inculcate the core information science engineering practices with hardware blend by providing advanced laboratories.

M6: To establish innovative labs, start-ups and patent culture.

Program Educational Objectives (PEOs)

PEO1: Graduates shall have successful careers as information science engineers and will be able to lead and manage teams across the globe.

PEO2: Graduates shall be professional in engineering practice and shall demonstrate good problem solving, communication skills and contribute to address societal issues.

PEO3: Graduates shall be pursuing distinctive education, entrepreneurship and research in an excellent environment which helps in the process of life-long learning.

Program Specific Outcomes (PSOs):

PSO1: Apply appropriate programming knowledge in software development, operations and maintenance of real-time applications.

PSO2: Meet the industry requirements in adapting to cutting edge technologies.

PSO3: Develop business and entrepreneurial ideas to support society requirements.

Program Outcomes (POs)

PO1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering 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 the information to provide valid conclusions.

PO5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

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 the professional 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 norms of the engineering practice.

PO9. Individual and team work: 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.

DAYANANDA SAGAR ACADEMY OF TECHNOLOGY & MANAGEMENT
(Affiliated to Visvesvaraya Technological University,Belagavi & Approved by AICTE,New Delhi)
Opp. Art of Living, Udayapura, Kanakapura Road, Bangalore- 560082
DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING
Accredited by NBA, New Delhi

SUBJECT: Artificial Intelligence and Machine Learning Laboratory

SUBJECT CODE: 18CSL76

SEMESTER: VII

Course outcomes

CO1	Implement AI & ML concepts using python programs
CO2	Identify and Apply Artificial Intelligence concepts to solve real world problems.
CO3	Select and apply appropriate algorithms and ML techniques to solve complex problems
CO4	Construct Machine learning programs for Supervised, Unsupervised and Semi supervised learning models.

Program 1

Implement a A* search algorithm

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.

Basic concepts of A*

f(n)=g(n)+h(n)

Where

g (n) : The actual cost path from the start node to the current node.

h (n) : The actual cost path from the current node to goal node.

f (n) : The actual cost path from the start node to the goal node.

For the implementation of A* algorithm we have to use two arrays namely OPEN and CLOSE.

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.

Note:

Node (also called State) — All potential position or stops with a unique identification

Transition — The act of moving between states or nodes.

Starting Node — Whereto start searching

Goal Node — The target to stop searching.

Search Space — A collection of nodes, like all board positions of a board game

Cost — Numerical value (say distance, time, or financial expense) for the path from a node to another node.

g(n) — this represents the exact cost of the path from the starting node to any node n

h(n) — this represents the heuristic estimated cost from node n to the goal node.

f(n) — lowest cost in the neighboring node n

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path.

Program:

```
from pyamaze import maze
from queue import PriorityQueue  #create prioirty queue for visited node cost selection
            #create maze object with size
#'maze attriutes to implement the algorithm such as rows, coloumns'

# 1 print(m.maze_map)    #check each cell North,South,West,East whether open or close
                # 1-is path direction is open and 0 is close
```

```
# 2 print(m.grid)              #List of all cells inside the maze


#define huristic function with 2input&cal. manhattan distance between 2 cells
#both will be as a tuple and output as manhattan distance between 2 cells
def h(cell1,cell2):
    x1,y1=cell1
    x2,y2=cell2

    #return  manhattan value as  ouput,abs-absolute value
    return abs(x1-x2) + abs(y1-y2)

# Create a function Astar to A* algorithm that input is one maze-m
def aStar(m):

#defining the start cell which last row and col of the maze
    start=(m.rows,m.cols)

#create two distance for gscore and fscore using distance
#computation for cell in maze dot credit list and value is infinity
    g_score = {row: float("inf") for row in m.grid}

    #g_score value of start cell is 0
    g_score[start]=0

     #similarly for f_score
     #f_score of the start cell is huristic value of the start cell +g_score
     #of the start cell
         f_score = {row: float("inf") for row in m.grid}

 #'total f_score of the start cell distance b/w start cell and goal cell'
             f_score[start]=h(start, (1,1))

 #'need a priority queue and should import it on line 2'
#create a prioirty queue named as open
open=PriorityQueue()
#put first entry in the priority queue which is a information of start cell as a tuple
open.put((h(start, (1,1),h(start, (1,1),start))
         #huristic cost+g(n),huris  tic cost,cell value itself

         #while loop -priority queue is not empty
while not open.empty():
    #add goal condition inside the loop and pickn the value from the priority queue
    #that will be the minimum value on the basis of fcost and h cost
    '''but i need cell value which is the third element in the tuple specify the index 2
    to get the third element which is cell, if that current cell is the goal i will
    end the process'''
    currCell=open.get()[2]
    if currCell==(1,1):
        break
    '''otherwise explore all four neighbors east,south,north and west'''
    for d in 'ESNW':
        #check path in that direction is open to use maze_map
```

```
        #{(1,1):{'E':1, 'W':0, 'N':0, 'S'=0}} if 1 find neighbor or child cell in each direction
        # using 4- if condition for 4 direction
        if m.maze_map[currCell][d]==True
                if d=='E':
  #if direction is E child cell is on the right side similarly find child cell for other three diretion
childCell=(currCell[0],currCell[1]+1)

if d=='E':
childCell=(currCell[0],currCell[1]+1)

if d=='W':
childCell=(currCell[0]-1,currCell[1])

if d=='N':
childCell=(currCell[0]+1,currCell[1])    #only one if condition true in each iteration

'"once child cell is found calculate new g_score i.e g_score of the current cell+1"'
temp_g_score=g_score[currCell]+1

temp_f_score=temp_g_score+h(childCell,(1,1))
# check if new fscore is less than the previous fscore update gscore and fscore
if temp_f_score < f_score[childCell]:
    g_score[childCell]=temp_g_score

     f_score[childCell]=temp_f_score
'" add one entry in to the prority queue mention 3 values fscore child cell,
hscore child cell and child cell itself"'
open.put((temp_f_score,h(childCell,(1,1)),childCell))    #completes the algorithm

m=maze(5,5)
m.CreateMaze()    #apply the method object to create maze
aStar(m)
m.run()            #run function to run simulation
```

Output:

```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]
```

Program 2

Implement a AO* search algorithm

AO* Algorithm basically based on problem decomposition (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.

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.

Program:

```
# Program to Implement recursive AO* Algorithm
def recAOStar(n):
 global finalPath
print("Expanding Node : ", n)
and_nodes = []
or_nodes = []
          #Segregation of AND and 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 leaf node then return
          if len(and_nodes) == 0 and len(or_nodes) == 0:
   return
solvable = False
marked = {}
while not solvable:
   # If all the child nodes are visited and expanded, take the least cost of all the child nodes
   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
    # Least cost of the unmarked child nodes
    min_cost, min_cost_group = least_cost_group(and_nodes, or_nodes, marked)
    is_expanded = False
   # If the child nodes have sub trees then recursively visit them to recalculate the heuristic of the child node
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 the child node had any subtree and expanded, verify if the new heuristic value is still the least among all
nodes
        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
    # If the child node does not have any subtrees then no change in heuristic, so update the min cost of the
current node
    else:
        solvable = True
        change_heuristic(n, min_cost)
        optimal_child_group[n] = min_cost_group

    #Mark the child node which was expanded
    marked[min_cost_group] = 1
    return heuristic(n)
    # Function to calculate the min cost among all the child nodes
    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
    # Calculates the min heuristic
    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]
    # Returns heuristic of a node
    def heuristic(n):
        return H_dist[n]
    # Updates the heuristic of a node
    def change_heuristic(n, cost):
        H_dist[n] = cost
        return
    # Function to print the optimal cost nodes
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)
                        #Describe the heuristic here
                        H_dist = { 'A': -1,'B': 4, 'C': 2, 'D': 3, 'E': 6,'F': 8, 'G': 2,'H': 0, 'I': 0, 'J': 0}
                        #Describe your graph here
                        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)
```

Output:

```
OUTPUT:-
Expanding Node : A
Expanding Node : B
Expanding Node : C
Expanding Node : D
Nodes which gives optimal cost are
CD->HI->J
Optimal Cost is :: 5
```

Program 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.

Task: The CANDIDATE-ELIMINATION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples.

Dataset: Weather training examples

Example

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Initialization

G0 ← {<?, ?, ?, ?, ?, ?>}

S0 ← {<Ø, Ø, Ø, Ø, Ø, Ø >}

G0 ← {<?, ?, ?, ?, ?, ?>}

S0 ← {<Ø, Ø, Ø, Ø, Ø, Ø >}

Iteration 1

x1 = <Sunny, Warm, Normal, Strong, Warm, Same>

G1 ← {<?, ?, ?, ?, ?, ?>}

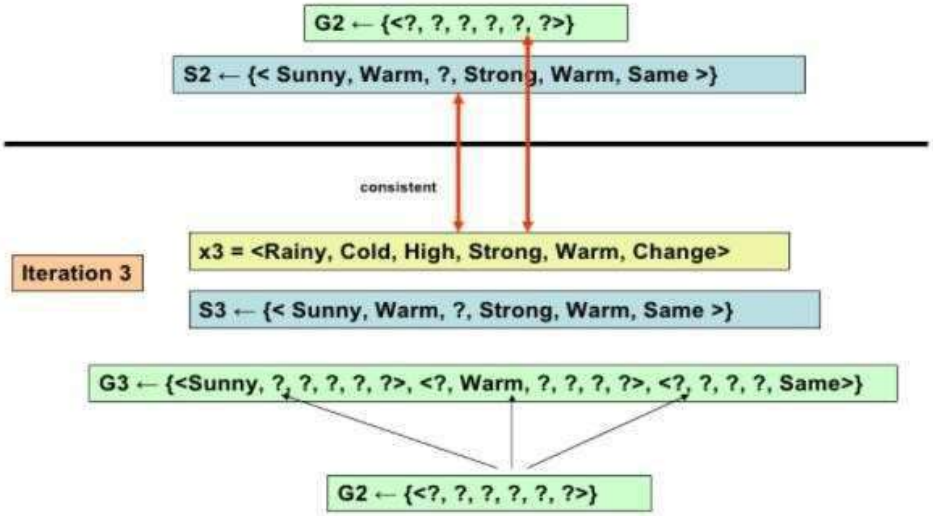
S1 ← {< Sunny, Warm, Normal, Strong, Warm, Same >}

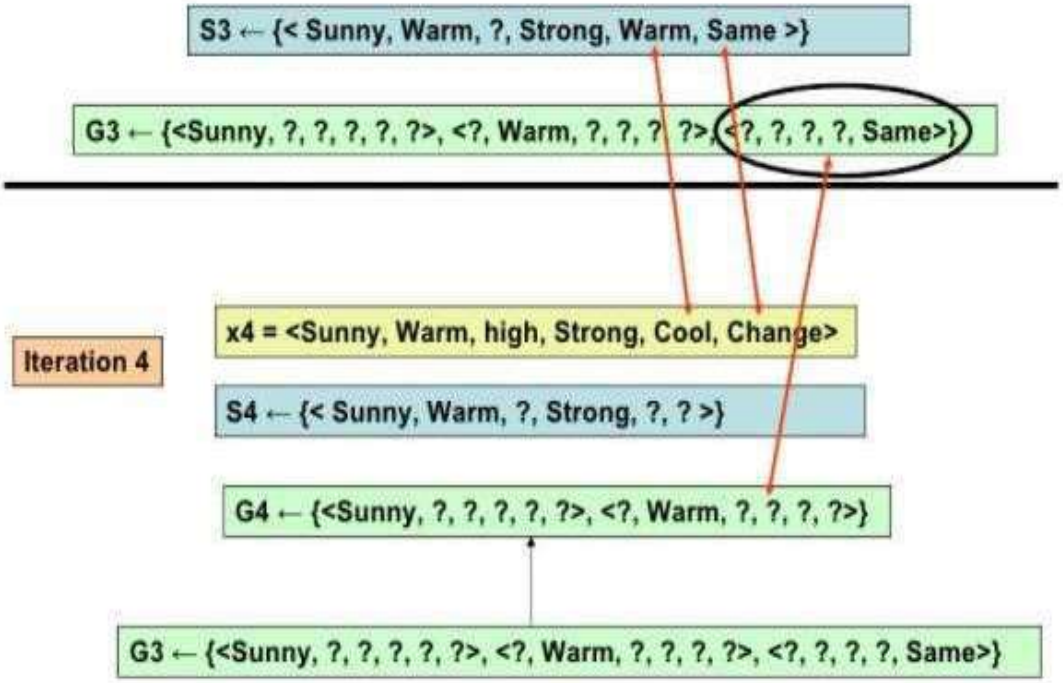
Iteration 2

x2 = <Sunny, Warm, High, Strong, Warm, Same>

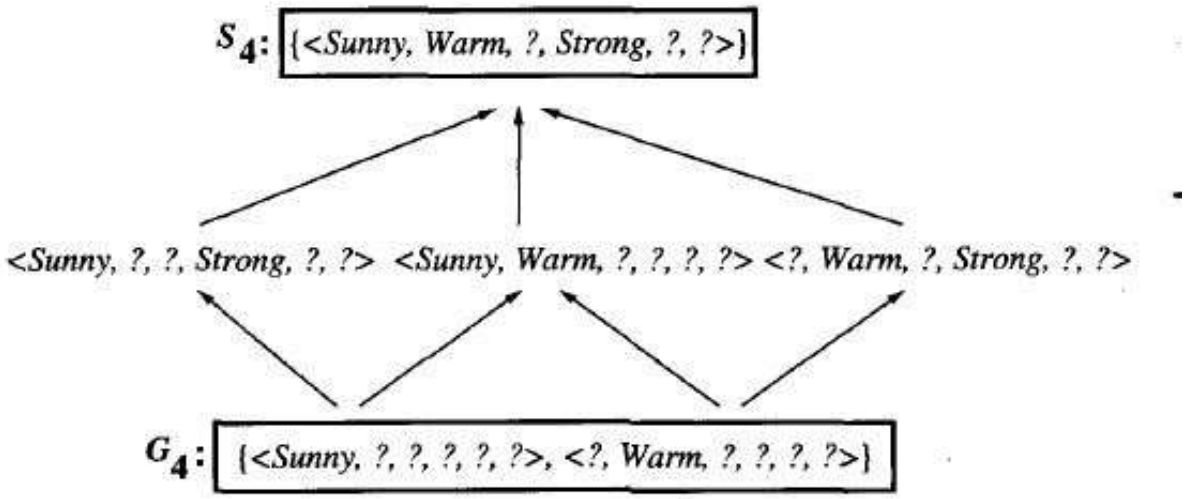
G2 ← {<?, ?, ?, ?, ?, ?>}

S2 ← {< Sunny, Warm, ?, Strong, Warm, Same >}


G2 ← {<?, ?, ?, ?, ?, ?>}
S2 ← {< Sunny, Warm, ?, Strong, Warm, Same >}
consistent
Iteration 3
x3 = <Rainy, Cold, High, Strong, Warm, Change>
S3 ← {< Sunny, Warm, ?, Strong, Warm, Same >}
G3 ← {<Sunny, ?, ?, ?, ?, ?>, <?, Warm, ?, ?, ?, ?>, <?, ?, ?, ?, Same>}
G2 ← {<?, ?, ?, ?, ?, ?>}


S3 ← {< Sunny, Warm, ?, Strong, Warm, Same >}
G3 ← {<Sunny, ?, ?, ?, ?, ?>, <?, Warm, ?, ?, ?, ?>, <?, ?, ?, ?, Same>}
Iteration 4
x4 = <Sunny, Warm, high, Strong, Cool, Change>
S4 ← {< Sunny, Warm, ?, Strong, ?, ? >}
G4 ← {<Sunny, ?, ?, ?, ?, ?>, <?, Warm, ?, ?, ?, ?>}
G3 ← {<Sunny, ?, ?, ?, ?, ?>, <?, Warm, ?, ?, ?, ?>, <?, ?, ?, ?, Same>}

Final version space



Candidate-Elimination Algorithm:

Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do

- If d is a positive example
  - Remove from G any hypothesis inconsistent with d
  - For each hypothesis s in S that is not consistent with d
    - Remove s from S
    - Add to S all minimal generalizations h of s such that
      - h is consistent with d, and some member of G is more general than h
    - Remove from S any hypothesis that is more general than another hypothesis in S
- If d is a negative example
  - Remove from S any hypothesis inconsistent with d
  - For each hypothesis g in G that is not consistent with d
    - Remove g from G
    - Add to G all minimal specializations h of g such that
      - h is consistent with d, and some member of S is more specific than h
    - Remove from G any hypothesis that is less general than another hypothesis in G

Candidate-Elimiation in Python:

```
class Holder:
factors={} #Initialize an empty dictionary
attributes = () #declaration of dictionaries parameters with an arbitrary length

   '''
   Constructor of class Holder holding two parameters,
self refers to the instance of the class
   '''
def init(self,attr): #
       self.attributes = attr
for i in attr:
           self.factors[i]=[]
def add_values(self,factor,values):
self.factors[factor]=values

class CandidateElimination:
   Positive={} #Initialize positive empty dictionary
   Negative={} #Initialize negative empty dictionary

def init(self,data,fact):
       self.num_factors = len(data[0][0])
       self.factors = fact.factors
       self.attr = fact.attributes
       self.dataset = data
       #print self.attr
```

```
def run_algorithm(self):
# print self.dataset
        '''
        Initialize the specific and general boundaries, and loop the dataset against the algorithm
        '''
        G = self.initializeG()
        S = self.initializeS()
print("initialize General",G)
print("initialize Specific",S)


        '''
        Programmatically populate list in the iterating variable trial_set
        '''
        #count=0
for trial_set in self.dataset:
if self.is_positive(trial_set): #if trial set/example consists of positive examples
                G = self.remove_inconsistent_G(G,trial_set[0]) #remove inconsitent data from the general
boundary
                S_new = S[:] #initialize the dictionary with no key-value pair
print(S_new)
for s in S:
if not self.consistent(s,trial_set[0]):
                        S_new.remove(s)
generalization = self.generalize_inconsistent_S(s,trial_set[0])
generalization = self.get_general(generalization,G)
if generalization:
                            S_new.append(generalization)
```

```
                S = S_new[:]
                S = self.remove_more_general(S)

    print(S)
    else:#if it is negative
            S = self.remove_inconsistent_S(S,trial_set[0]) #remove inconsitent data from the specific
    boundary
            G_new = G[:] #initialize the dictionary with no key-value pair (dataset can take any value)
    print(G_new)
    for g in G:
    if self.consistent(g,trial_set[0]):
                    G_new.remove(g)
    specializations = self.specialize_inconsistent_G(g,trial_set[0])
    specializationss = self.get_specific(specializations,S)
    if specializations != []:
                        G_new += specializations
                G = G_new[:]
    print(G)
                G = self.remove_more_specific(G)
    print(S)
    print(G)
    def initializeS(self):
        ''' Initialize the specific boundary '''
        S = tuple(['-' for factor in range(self.num_factors)]) #6 constraints in the vector
    return [S]
    def initializeG(self):
        ''' Initialize the general boundary '''
```

```
        G = tuple(['?' for factor in range(self.num_factors)]) # 6 constraints in the vector
    return [G]


    def is_positive(self,trial_set):
        ''' Check if a given training trial_set is positive '''
    if trial_set[1] == 'Y':
    return True
    elif trial_set[1] == 'N':
    return False
    else:
    raise TypeError("invalid target value")


    def is_negative(self,trial_set):
        ''' Check if a given training trial_set is negative '''
    if trial_set[1] == 'N':
    return False
    elif trial_set[1] == 'Y':
    return True
    else:
    raise TypeError("invalid target value")
    def match_factor(self,value1,value2):
        ''' Check for the factors values match,
    necessary while checking the consistency of
    training trial_set with the hypothesis '''
    if value1 == '?' or value2 == '?':
    return True
```

```
    elif value1 == value2 :
    return True
    return False
    def consistent(self,hypothesis,instance):
        ''' Check whether the instance is part of the hypothesis '''
    for i,factor in enumerate(hypothesis):
    if not self.match_factor(factor,instance[i]):
    return False
    return True

    def remove_inconsistent_G(self,hypotheses,instance):
        ''' For a positive trial_set, the hypotheses in G
    inconsistent with it should be removed '''
        G_new = hypotheses[:]
    for g in hypotheses:
    if not self.consistent(g,instance):
                G_new.remove(g)
    return G_new

    def remove_inconsistent_S(self,hypotheses,instance):
        ''' For a negative trial_set, the hypotheses in S
    inconsistent with it should be removed '''
        S_new = hypotheses[:]
    for s in hypotheses:
    if self.consistent(s,instance):
                S_new.remove(s)
```

```
    return S_new


    def remove_more_general(self,hypotheses):
    ''' After generalizing S for a positive trial_set, the hypothesis in S
    general than others in S should be removed '''
        S_new = hypotheses[:]
    for old in hypotheses:
    for new in S_new:
    if old!=new and self.more_general(new,old):
                S_new.remove[new]
    return S_new


    def remove_more_specific(self,hypotheses):
        ''' After specializing G for a negative trial_set, the hypothesis in G
    specific than others in G should be removed '''
        G_new = hypotheses[:]
    for old in hypotheses:
    for new in G_new:
    if old!=new and self.more_specific(new,old):
                G_new.remove[new]
    return G_new


    def generalize_inconsistent_S(self,hypothesis,instance):
        ''' When a inconsistent hypothesis for positive trial_set is seen in the specific boundary S,
    it should be generalized to be consistent with the trial_set ... we will get one hypothesis'''
    hypo = list(hypothesis) # convert tuple to list for mutability
```

```
for i,factor in enumerate(hypo):
if factor == '-':
hypo[i] = instance[i]
elif not self.match_factor(factor,instance[i]):
hypo[i] = '?'
generalization = tuple(hypo) # convert list back to tuple for immutability
return generalization

def specialize_inconsistent_G(self,hypothesis,instance):
    ''' When a inconsistent hypothesis for negative trial_set is seen in the general boundary G
should be specialized to be consistent with the trial_set.. we will get a set of hypotheses '''
specializations = []
hypo = list(hypothesis) # convert tuple to list for mutability
for i,factor in enumerate(hypo):
if factor == '?':
values = self.factors[self.attr[i]]
for j in values:
if instance[i] != j:
hyp=hypo[:]
hyp[i]=j
hyp=tuple(hyp) # convert list back to tuple for immutability
specializations.append(hyp)
return specializations

def get_general(self,generalization,G):
    ''' Checks if there is more general hypothesis in G
```

```
    for a generalization of inconsistent hypothesis in S
    in case of positive trial_set and returns valid generalization '''

    for g in G:
    if self.more_general(g,generalization):
    return generalization
    return None

    def get_specific(self,specializations,S):
        ''' Checks if there is more specific hypothesis in S
    for each of hypothesis in specializations of an
    inconsistent hypothesis in G in case of negative trial_set
    and return the valid specializations'''
        valid_specializations = []
    for hypo in specializations:
    for s in S:
    if self.more_specific(s,hypo) or s==self.initializeS()[0]:
                    valid_specializations.append(hypo)
    return valid_specializations

    def exists_general(self,hypothesis,G):
        '''Used to check if there exists a more general hypothesis in
    general boundary for version space'''

    for g in G:
    if self.more_general(g,hypothesis):
```

```
    return True
    return False

    def exists_specific(self,hypothesis,S):
        '''Used to check if there exists a more specific hypothesis in
    general boundary for version space'''

    for s in S:
    if self.more_specific(s,hypothesis):
    return True
    return False

    def get_version_space(self,specific,general):
        ''' Given the specific and the general boundary of the
    version space, evaluate the version space in between '''
    while get_order(VS):
    for hypothesis in VS[:]:
    hypo = list(hypothesis) # convert tuple to list for mutability
    for i,factor in enumerate(hypo):
    if factor != '?':
    hyp=hypo[:]
    hyp[i]='?'
    if self.exists_general(hyp,general)and self.exists_specific(hyp,specific):
    VS.append(tuple(hyp))

    return VS
```

```
def get_order(self,hypothesis):
pass

def more_general(self,hyp1,hyp2):
    ''' Check whether hyp1 is more general than hyp2 '''
hyp = list(zip(hyp1,hyp2))
for i,j in hyp:
if i == '?':
continue
elif j == '?':
if i != '?':
return False
elif i != j:
return False
else:
continue
return True

def more_specific(self,hyp1,hyp2):
    ''' hyp1 more specific than hyp2 is
equivalent to hyp2 being more general than hyp1 '''
return self.more_general(hyp2,hyp1)
```

```
dataset=[(('sunny','warm','normal','strong','warm','same'),'Y'),(('sunny','warm','high','strong','warm','same'),'Y'),(('rainy','cold','high','strong','warm','change'),'N'),(('sunny','warm','high','strong','cool','change'),'Y')]
```

```
#print((dataset[0][1]))

attributes =('Sky','Temp','Humidity','Wind','Water','Forecast')


f = Holder(attributes)

f.add_values('Sky',('sunny','rainy','cloudy')) #sky can be sunny rainy or cloudy

f.add_values('Temp',('cold','warm')) #Temp can be sunny cold or warm

f.add_values('Humidity',('normal','high')) #Humidity can be normal or high

f.add_values('Wind',('weak','strong')) #wind can be weak or strong

f.add_values('Water',('warm','cold')) #water can be warm or cold

f.add_values('Forecast',('same','change')) #Forecast can be same or change

a = CandidateElimination(dataset,f)

#pass the dataset to the algorithm class and call the run algoritm method

a.run_algorithm()
```

Output

```
 ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

0

['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

1

['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

2

['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
```

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

3

['Sunny' 'Warm' '?' 'Strong' '?' '?']

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final S:

['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final G:

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

Process finished with exit code 0

Program 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.

Task: ID3 determines the information gain for each candidate attribute (i.e., Outlook, Temperature, Humidity, and Wind), then selects the one with highest information gain as the root node of the tree. The information gain values for all four attributes are calculated using the following formula:

$$\text{Entropy}(S)=\sum - P(I).\log_2 P(I)$$

$$\text{Gain}(S,A)=\text{Entropy}(S)-\sum [\ P(S/A).\text{Entropy}(S/A)\ ]$$

Dataset:

Table: Training examples for the target concept PlayTennis.

outlook	temperature	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

Calculation:

Decision/play column consists of 14 instances and includes two labels: yes and no.
There are 9 decisions labeled yes and 5 decisions labeled no.

$$\text{Entropy[Decision]} = -P(\text{yes}).\log_2 P(\text{yes}) - P(\text{no}).\log_2 P(\text{no})$$
$$= -(9/14).\log_2(9/14) - (5/14).\log_2(5/14)$$

= 0.940

Now, we need to find out the most dominant factor for decision.

1) Wind factor on decision:

Gain(Decision,wind)= Entropy(Decision) - ∑ [P(Decision/Wind]. Entropy(Decision/Wind)]

Wind attribute has two labels : Weak and Strong

Gain(Decision,Wind)= Entropy(Decision) – [P(Decision/Wind=Weak). Entropy(Decision/Wind=Weak)] - [[P(Decision/Wind=strong). Entropy(Decision/Wind=strong)]

There are 8 instances for weak. In that decision of 2 items are no and 6 items are yes.

- Entropy[Decision/Wind= Weak] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
  = -[2/8].$\log_2$(2/8) – [6/8].$\log_2$(6/8)
  = 0.811

- Entropy[Decision/Wind= Strong] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
  = -[3/6].$\log_2$(3/6) – [3/6].$\log_2$(3/6)
  = 1

  Note: There are 6 instances for strong. In that decisionof 3 items are yes and 3 items are no.

- Gain(Decision,Wind) = 0.940 – [(8/14).0.811] – [(6/14).1]
  = 0.048

  Similarly calculate gain for other factors:

2) Outlook factor on decision:

1) Gain(Decision,outlook) = Entropy (decision) - ∑ [P (Decision /Outlook). Entropy(Decision/Outlook)]

outlook has three parameters : Sunny, Overcast, and rain

Gain(Decision,outlook) = Entropy(decision)- [P(Decision/Outlook=Sunny).Entropy(Decision/Outlook=Sunny)]-[P(Decision/Outlook=overcast).Entropy(Decision/Outlook=overcast)]- [P(P(Decision/Outlook=Rain).Entropy(Decision/Outlook=Rain)]

Sunny	Overcast:	Rain:
Instances: 5	Instances:4	Instances:5
yes:2	yes:4	yes:3
No:3	No: -	No: 2

1) Entropy[Decision/Outlook= Sunny] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
=0.97094

2) Entropy[Decision/Outlook= Overcast] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
= 0

3) Entropy[Decision/Wind= Rain] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
=0.9708

Gain(Decision,Outlook)= 0.940-(5/14)(0.9709)-(4/14)(0)-(5/14)(0.9708)
=0.2473

3) Temperature factor on decision:

Gain(Decision,Temperature) = Entropy(decision)-
∑[P(Decision/Temperature).Entropy(Decision/Temperature)]

Temperature has 3 parameters: hot, mild, cool

Gain(Decision,Temp) = Entropy(decision)-
[P(Decision/Temp=hot).Entropy(Decision/Temp=hot)]-[
P(Decision/Temp=mild).Entropy(Decision/Temp=mild)]-
[P(P(Decision/Temp=cool).Entropy(Decision/Temp=cool)]

Hot	Mild:	cool:
Instances:4	Instances:6	Instances:4
Yes:2	Yes:	Yes:3
No:2	No: 2	No:1

Entropy[Decision/Temp= hot] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
= 1
Entropy[Decision/Temp= mild] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
=0.9182
Entropy[Decision/Temp= cool] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)

=0.8112

Gain(Decision,Temp) = 0.940 - (4/14)(1) - (6/14)(0.9182) - (4/14)(0.8112)
= 0.0291

4) <u>Humidity factor on decision:</u>

Gain(Decision,Humidity) = Entropy(decision)-
∑[P(Decision/Humidity).Entropy(Decision/Humidity)]

Humidity has 2 factors: high and normal
Gain(Decision,humidity) = Entropy(decision)-
[P(Decision/humidity=high).Entropy(Decision/humidity=high]-[
P(Decision/humidity=normal).Entropy(Decision/humidity=normal)]

<u>High</u>	<u>Normal:</u>
Instances:7	Instances:7
Yes:3	Yes:6
No:4	No: 1

Entropy[Decision/ humidity= high] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
= 0.9851

Entropy[Decision/humidity= normal] = -P[no].$\log_2$P(no) – p(yes).$\log_2$P(yes)
= 0.5916

Gain(Decision, Humidity)= 0.940 - (7/14)(0.9851) - (7/14)(0.5916)
= 0.1517

Thus the outlook factor on decision produces the highest score. That's why outlook decision will appear in the root node of the tree. Since Outlook has three possible values, the root node has three branches (sunny, overcast, rain). The next question is "what attribute should be tested at the Sunny branch node?" Since we have used Outlook at the root, we only decide on the remaining three attributes: Humidity, Temperature, or Wind.

Now calculate sunny outlook on decision, overcast outlook on decision, and rain outlook on decision to generate the decision tree.

Sunny outlook on decision:

5 instances of sunny : In that 3 instances are NO and 2 instances are YES

Gain(Outlook = Sunny/Temp)= 0.570

Gain (Outlook = Sunny/Humidity) = 0.970

Gain (Outlook = Sunny/ Wind) = 0.019

Since humidity produces the highest score, if outlook were Sunny.

Overcast outlook on decision:

Decision will always be yes, if outlook were overcast.

Rain outlook on decision:

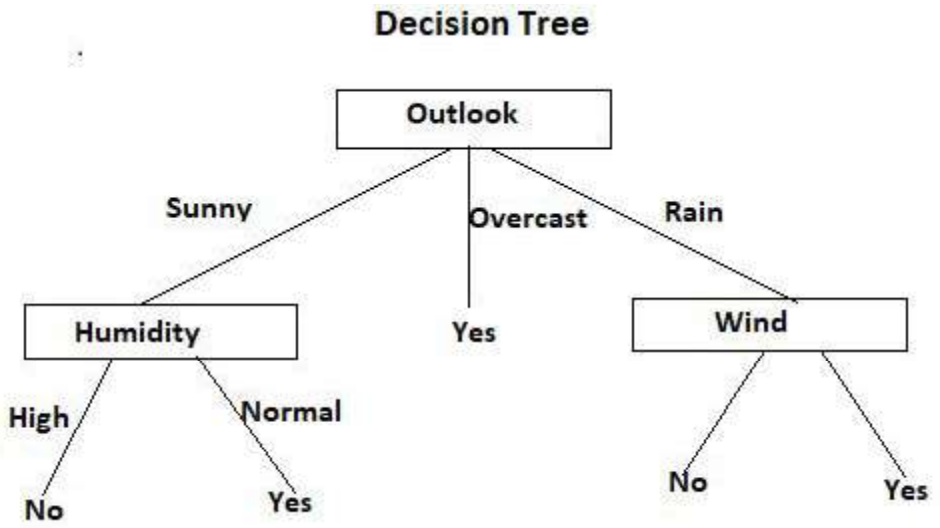
5 instances of rain : In that 3 instances are YES and 2 instances are NO.

Gain(Outlook= Rain/Temp)

Gain(Outlook= Rain/Humidity)

Gain(Outlook= Rain/Wind)

Here, wind produces the highest score . And wind has two attributes namely strong and weak.



ID3 Algorithm:

ID3(*Examples, Target_attribute, Attributes*)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- 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 *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
  - Otherwise Begin

❖ A ← the attribute from *Attributes* that best* classifies *Examples*

❖ The decision attribute for *Root* ←A

❖ For each possible value, υ_i, of A,

- Add a new tree branch below *Root*, corresponding to the test A = υ_i
- Let $Examples_{vi}$,be the subset of *Examples* that have value υ_i for A
- If $Examples_{vi}$, is empty
  - ❖ Then below this new branch add a leaf node with label=most common value of *Target_attribute* in *Examples*
  - ❖ Else below this new branch add the subtree

ID3($Examples_{vi}$,*Target_attribute, Attributes–{A}*))

End

Return Root

ID3 in Python:

```
import ast
import csv
import sys
import math
import os
def load_csv_to_header_data(filename):
    path = os.path.normpath(os.getcwd() + filename)
    fs = csv.reader(open(path))
```

```
    all_row = []
    for r in fs:
        all_row.append(r)

    headers = all_row[0]
    idx_to_name, name_to_idx = get_header_name_to_idx_maps(headers)

    data = {
        'header': headers,
        'rows': all_row[1:],
        'name_to_idx': name_to_idx,
        'idx_to_name': idx_to_name
    }
    return data


def get_header_name_to_idx_maps(headers):
    name_to_idx = {}
    idx_to_name = {}
    for i in range(0, len(headers)):
        name_to_idx[headers[i]] = i
        idx_to_name[i] = headers[i]
    return idx_to_name, name_to_idx


def project_columns(data, columns_to_project):
    data_h = list(data['header'])
    data_r = list(data['rows'])

    all_cols = list(range(0, len(data_h)))

    columns_to_project_ix = [data['name_to_idx'][name] for name in
columns_to_project]
    columns_to_remove = [cidx for cidx in all_cols if cidx not in
columns_to_project_ix]

    for delc in sorted(columns_to_remove, reverse=True):
        del data_h[delc]
        for r in data_r:
            del r[delc]
```

```
    idx_to_name, name_to_idx = get_header_name_to_idx_maps(data_h)

    return {'header': data_h, 'rows': data_r,
            'name_to_idx': name_to_idx,
            'idx_to_name': idx_to_name}


def get_uniq_values(data):
    idx_to_name = data['idx_to_name']
    idxs = idx_to_name.keys()

    val_map = {}
    for idx in iter(idxs):
        val_map[idx_to_name[idx]] = set()

    for data_row in data['rows']:
        for idx in idx_to_name.keys():
            att_name = idx_to_name[idx]
            val = data_row[idx]
            if val not in val_map.keys():
                val_map[att_name].add(val)
    return val_map


def get_class_labels(data, target_attribute):
    rows = data['rows']
    col_idx = data['name_to_idx'][target_attribute]
    labels = {}
    for r in rows:
        val = r[col_idx]
        if val in labels:
            labels[val] = labels[val] + 1
        else:
            labels[val] = 1
    return labels


def entropy(n, labels):
    ent = 0
    for label in labels.keys():
        p_x = labels[label] / n
```

```
            ent += - p_x * math.log(p_x, 2)
        return ent


    def partition_data(data, group_att):
        partitions = {}
        data_rows = data['rows']
        partition_att_idx = data['name_to_idx'][group_att]
        for row in data_rows:
            row_val = row[partition_att_idx]
            if row_val not in partitions.keys():
                partitions[row_val] = {
                    'name_to_idx': data['name_to_idx'],
                    'idx_to_name': data['idx_to_name'],
                    'rows': list()
                }
            partitions[row_val]['rows'].append(row)
        return partitions


    def avg_entropy_w_partitions(data, splitting_att, target_attribute):
        # find uniq values of splitting att
        data_rows = data['rows']
        n = len(data_rows)
        partitions = partition_data(data, splitting_att)

        avg_ent = 0

        for partition_key in partitions.keys():
            partitioned_data = partitions[partition_key]
            partition_n = len(partitioned_data['rows'])
            partition_labels = get_class_labels(partitioned_data, target_attribute)
            partition_entropy = entropy(partition_n, partition_labels)
            avg_ent += partition_n / n * partition_entropy

        return avg_ent, partitions


    def most_common_label(labels):
        mcl = max(labels, key=lambda k: labels[k])
        return mcl
```

```
def id3(data, uniqs, remaining_atts, target_attribute):
    labels = get_class_labels(data, target_attribute)

    node = {}

    if len(labels.keys()) == 1:
        node['label'] = next(iter(labels.keys()))
        return node

    if len(remaining_atts) == 0:
        node['label'] = most_common_label(labels)
        return node

    n = len(data['rows'])
    ent = entropy(n, labels)

    max_info_gain = None
    max_info_gain_att = None
    max_info_gain_partitions = None

    for remaining_att in remaining_atts:
        avg_ent, partitions = avg_entropy_w_partitions(data, remaining_att,
target_attribute)
        info_gain = ent - avg_ent
        if max_info_gain is None or info_gain > max_info_gain:
            max_info_gain = info_gain
            max_info_gain_att = remaining_att
            max_info_gain_partitions = partitions

    if max_info_gain is None:
        node['label'] = most_common_label(labels)
        return node

    node['attribute'] = max_info_gain_att
    node['nodes'] = {}

    remaining_atts_for_subtrees = set(remaining_atts)
    remaining_atts_for_subtrees.discard(max_info_gain_att)

    uniq_att_values = uniqs[max_info_gain_att]
```

```
    for att_value in uniq_att_values:
        if att_value not in max_info_gain_partitions.keys():
            node['nodes'][att_value] = {'label': most_common_label(labels)}
            continue
        partition = max_info_gain_partitions[att_value]
        node['nodes'][att_value] = id3(partition, uniqs, remaining_atts_for_subtrees,
target_attribute)

    return node


def load_config(config_file):
    with open(config_file, 'r') as myfile:
        data = myfile.read().replace('\n', '')
    return ast.literal_eval(data)


def pretty_print_tree(root):
    stack = []
    rules = set()

    def traverse(node, stack, rules):
        if 'label' in node:
            stack.append(' THEN ' + node['label'])
            rules.add(''.join(stack))
            stack.pop()
        elif 'attribute' in node:
            ifnd = 'IF ' if not stack else ' AND '
            stack.append(ifnd + node['attribute'] + ' EQUALS ')
            for subnode_key in node['nodes']:
                stack.append(subnode_key)
                traverse(node['nodes'][subnode_key], stack, rules)
                stack.pop()
            stack.pop()

    traverse(root, stack, rules)
    print(os.linesep.join(rules))


def main():
    argv ='tennis.cfg'
```

```
    print("Command line args are {}: ".format(argv))

    config = load_config(argv)

    data = load_csv_to_header_data(config['data_file'])
    data = project_columns(data, config['data_project_columns'])

    target_attribute = config['target_attribute']
    remaining_attributes = set(data['header'])
    remaining_attributes.remove(target_attribute)

    uniqs = get_uniq_values(data)

    root = id3(data, uniqs, remaining_attributes, target_attribute)

    pretty_print_tree(root)


if__name__== "_main_": main()
```

Output:

```
IF Outlook EQUALS Rainy AND Windy EQUALS True THEN No
IF Outlook EQUALS Sunny AND Humidity EQUALS Normal THEN Yes
IF Outlook EQUALS Overcast THEN Yes
IF Outlook EQUALS Rainy AND Windy EQUALS False THEN Yes
IF Outlook EQUALS Sunny AND Humidity EQUALS High THEN No
```

Program 5:

Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data set

Back propagation Algorithm

The Backpropagation algorithm is a supervised learning method for multilayer feed-forward networks from the field of Artificial Neural Networks, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer. Backpropagation can be used for both classification and regression problems.

Wheat Seeds Dataset

The seeds dataset involves the prediction of species given measurements seeds from different varieties of wheat. There are 201 records and 7 numerical input variables. It is a classification problem with 3 output classes. The scale for each numeric input value vary, so some data normalization may be required for use with algorithms that weight inputs like the backpropagation algorithm.

Below is a sample of the first 5 rows of the dataset

```
15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1
```

You can download the seeds dataset from the UCI Machine Learning Repository.
Download the seeds dataset and place it into your current working directory with the filename **seeds_dataset.csv**.
The dataset is in tab-separated format, so you must convert it to CSV using a text editor or a spreadsheet program.

Working of algorithm:

1. Initialize Network.
2. Forward Propagate.
3. Back Propagate Error.
4. Train the network

1. Initialize Network

Initialize the network weights to small random numbers in the range of 0 to 1. Below is a function named initialize_network() that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs.

The hidden layer we create n_hidden neurons and each neuron in the hidden layer has n_inputs + 1 weights, one for each input column in a dataset and an additional one for the bias. The output layer that connects to the hidden layer has n_outputs neurons, each with n_hidden + 1 weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

Let's test out this function. Below is a complete example that creates a small network.

```
from random import seed
from random import random

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
network = list()
hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in
range(n_hidden)]
network.append(hidden_layer)
output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
range(n_outputs)]
network.append(output_layer)
return network

seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
print(layer)
```

Running the example, you can see that the code prints out each layer one by one. You can see the hidden layer has one neuron with 2 input weights plus the bias. The output layer has 2 neurons, each with 1 weight plus the bias.

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights':
[0.4494910647887381, 0.651592972722763]}]
```

Forward Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. We call this forward-propagation.

We can break forward propagation down into three parts:

- Neuron Activation.
- Neuron Transfer.
- Forward Propagation.

Neuron Activation

The first step is to calculate the activation of one neuron given an input.

The input could be a row from our training dataset, as in the case of the hidden layer. It may also be the outputs from each neuron in the hidden layer, in the case of the output layer. Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression.

```
activation = sum(weight_i * input_i) + bias
```

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

Where weight is a network weight, input is an input, i is the index of a weight or an input and bias is a special weight that has no input to multiply with (or you can think of the input as always being 1.0). Below is an implementation of this in a function named activate(). You can see that the function assumes that the bias is the last weight in the list of weights. This helps here and later to make the code easier to read.

```
# Calculate neuron activation for an input
def activate(weights, inputs):
activation = weights[-1]
for i in range(len(weights)-1):
activation += weights[i] * inputs[i]
return activation
```

Neuron Transfer

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used. It is traditional to use the sigmoid activation function. It is also a function of which we can easily calculate the derivative (slope) that we will need later when backpropagating error. We can transfer an activation function using the sigmoid function as follows:

```
output = 1 / (1 + e^(-activation))
```

Where e is the base of the natural logarithms (Euler's number).

Below is a function named transfer() that implements the sigmoid equation.

```
# Transfer neuron activation
def transfer(activation):
return 1.0 / (1.0 + exp(-activation))
```

Forward Propagation

Forward propagating an input is straightforward. All of the outputs from one layer become inputs to the neurons on the next layer. Below is a function named forward_propagate() that implements the forward propagation for a row of data from our dataset with our neural network.

The function returns the outputs from the last layer also called the output layer.

```
# Forward propagate input to a network output
def forward_propagate(network, row):
inputs = row
for layer in network:
new_inputs = []
for neuron in layer:
activation = activate(neuron['weights'], inputs)
neuron['output'] = transfer(activation)
new_inputs.append(neuron['output'])
inputs = new_inputs
return inputs
```

Running the example propagates the input pattern [1, 0] and produces an output value that is printed. Because the output layer has two neurons, we get a list of two numbers as output.

```
[0.6629970129852887, 0.7253160725279748]
```

Back Propagate Error

The backpropagation algorithm is named for the way in which weights are trained. Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go. The math for backpropagating error is rooted in calculus, but we will remain high level in this section and focus on what is calculated and how rather than why the calculations take this particular form.

This part is broken down into two sections.

Transfer Derivative.

Error Backpropagation.

Transfer Derivative

Given an output value from a neuron, we need to calculate it's slope. We are using the sigmoid transfer function, the derivative of which can be calculated as follows:

```
derivative = output * (1.0 - output)
```

Below is a function named transfer_derivative() that implements this equation.

```
# Calculate the derivative of an neuron output
def transfer_derivative(output):
return output * (1.0 - output)
```

Now, let's see how this can be used.

Error Backpropagation

The first step is to calculate the error for each output neuron, this will give us our error signal (input) to propagate backwards through the network.

The error for a given neuron can be calculated as follows:

```
error = (expected - output) * transfer_derivative(output)
```

Where expected is the expected output value for the neuron, output is the output value for the neuron and transfer_derivative() calculates the slope of the neuron's output value, as shown above.

The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the neurons in the hidden layer.

The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:

```
error = (weight_k * error_j) * transfer_derivative(output)
```

Where error_j is the error signal from the jth neuron in the output layer, weight_k is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron.

Below is a function named backward_propagate_error() that implements this procedure. You can see that the error signal calculated for each neuron is stored with the name 'delta'. You can see that the layers of the network are iterated in reverse order, starting at the output and working backwards. This ensures that the neurons in the output layer have 'delta' values calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name 'delta' to reflect the change the error implies on the neuron (e.g. the weight delta). You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number j is also the index of the neuron's weight in the output layer neuron['weights'][j].

```
# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
for i in reversed(range(len(network))):
layer = network[i]
errors = list()
if i != len(network)-1:
for j in range(len(layer)):
error = 0.0
for neuron in network[i + 1]:
error += (neuron['weights'][j] * neuron['delta'])
errors.append(error)
else:
for j in range(len(layer)):
neuron = layer[j]
errors.append(expected[j] - neuron['output'])
for j in range(len(layer)):
```

```
17  neuron = layer[j]
18  neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

Running the example prints the network after the backpropagation of error is complete. You can see that error values are calculated and stored in the neurons for the output layer and the hidden layer.

```
1  [{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327,
2  0.763774618976614], 'delta': -0.0005348048046610517}]
   [{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095],
   'delta': -0.14619064683582808}, {'output': 0.6573693455986976, 'weights':
   [0.4494910647887381, 0.651592972722763], 'delta': 0.0771723774346327}]
```

Now let's use the backpropagation of error to train the network.

Train Network

The network is trained using stochastic gradient descent. This involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, backpropagating the error and updating the network weights.

This part is broken down into two sections:

- Update Weights.
- Train Network.

Update Weights

Once errors are calculated for each neuron in the network via the back propagation method above, they can be used to update weights. Network weights are updated as follows:

```
weight = weight + learning_rate * error * input
```

Where weight is a given weight, learning_rate is a parameter that you must specify, error is the error calculated by the backpropagation procedure for the neuron and input is the input value that caused the error.

Learning rate controls how much to change the weight to correct for the error. Below is a function named update_weights() that updates the weights for a network given an input row of data, a learning rate and assume that a forward and backward propagation have already been performed. Remember that the input for the output layer is a collection of outputs from the hidden layer.

```
1  # Update network weights with error
2  def update_weights(network, row, l_rate):
3  for i in range(len(network)):
4  inputs = row[:-1]
5  if i != 0:
6  inputs = [neuron['output'] for neuron in network[i - 1]]
7  for neuron in network[i]:
8  for j in range(len(inputs)):
9  neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
```

```
10  neuron['weights'][-1] += l_rate * neuron['delta']
```

Now we know how to update network weights, let's see how we can do it repeatedly.

Train Network

As mentioned, the network is updated using stochastic gradient descent. This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset. Below is a function that implements the training of an already initialized neural network with a given training dataset, learning rate, fixed number of epochs and an expected number of output values. The sum squared error between the expected output and the network output is accumulated each epoch and printed.

```
1  # Train a network for a fixed number of epochs
2  def train_network(network, train, l_rate, n_epoch, n_outputs):
3  for epoch in range(n_epoch):
4  sum_error = 0
5  for row in train:
6  outputs = forward_propagate(network, row)
7  expected = [0 for i in range(n_outputs)]
8  expected[row[-1]] = 1
9  sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
10 backward_propagate_error(network, expected)
11 update_weights(network, row, l_rate)
12 print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
```

ANN in python:

```
from math import exp
from random import seed
from random import random

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
        network = list()
        hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
        network.append(hidden_layer)
        output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
```

```
        network.append(output_layer)
        return network

# Calculate neuron activation for an input
def activate(weights, inputs):
        activation = weights[-1]
        for i in range(len(weights)-1):
                activation += weights[i] * inputs[i]
        return activation

# Transfer neuron activation
def transfer(activation):
        return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
        inputs = row
        for layer in network:
                new_inputs =  []
                for neuron in layer:
                        activation = activate(neuron['weights'], inputs)
                        neuron['output'] = transfer(activation)
                        new_inputs.append(neuron['output'])
                inputs = new_inputs
        return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
        return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
        for i in reversed(range(len(network))):
                layer = network[i]
                errors = list()
                if i != len(network)-1:
                        for j in range(len(layer)):
                                error = 0.0
                                for neuron in network[i + 1]:
                                        error += (neuron['weights'][j] * neuron['delta'])
                                errors.append(error)
                else:
                        for j in range(len(layer)):
                                neuron = layer[j]
```

```
                                        errors.append(expected[j] - neuron['output'])
                        for j in range(len(layer)):
                                neuron = layer[j]
                                neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
        for i in range(len(network)):
                inputs = row[:-1]
                if i != 0:
                        inputs = [neuron['output'] for neuron in network[i - 1]]
                for neuron in network[i]:
                        for j in range(len(inputs)):
                                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
                        neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
        for epoch in range(n_epoch):
                sum_error = 0
                for row in train:
                        outputs = forward_propagate(network, row)
                        expected = [0 for i in range(n_outputs)]
                        expected[row[-1]] = 1
                        sum_error += sum([(expected[i]-outputs[i])**2 for i in
range(len(expected))])
                        backward_propagate_error(network, expected)
                        update_weights(network, row, l_rate)
                print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))

# Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],
        [1.465489372,2.362125076,0],
        [3.396561688,4.400293529,0],
        [1.38807019,1.850220317,0],
        [3.06407232,3.005305973,0],
        [7.627531214,2.759262235,1],
        [5.332441248,2.088626775,1],
        [6.922596716,1.77106367,1],
        [8.675418651,-0.242068655,1],
        [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
```

```
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
        print(layer)
```

Output

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.22147
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output':
0.029980305604426185, 'delta': -0.0059546604162323625}, {'weights': [0.37711098142462157, -
```

0.0625909894552989, 0.2765123702642716], 'output': 0.9456229000211323, 'delta': 0.0026279652850863837}]

[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357587, 'delta': -0.04270059278364587}, {'weights': [-2.5584149848484263, 1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta': 0.03803132596437354}]

>>>

Program 6

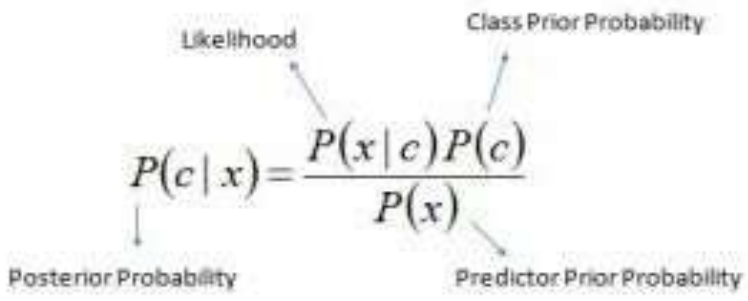
Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

Task: It 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'.

Data Set : PlayTennis example

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c).



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

Above,

- 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.

So here we have 4 attributes. What we need to do is to create "look-up tables" for each of these attributes, and write in the probability that a game of tennis will be played based on this attribute. In these tables we have to note that there are 5 cases of not being able to play a game, and 9 cases of being able to play a game.

We also must note the following probabilities for P(C):

P(Play=Yes) = 9/14

P(Play=No) = 5/14

OUTLOOK	Play = Yes	Play = No	Total
Sunny	2/9	3/5	5/14
Overcast	4/9	0/5	4/14
Rain	3/9	2/5	5/14

TEMPERATURE	Play = Yes	Play = No	Total
Hot	2/9	2/5	4/14
Mild	4/9	2/5	6/14
Cool	3/9	1/5	4/14

HUMIDITY	Play = Yes	Play = No	Total
High	3/9	4/5	7/14
Normal	6/9	1/5	7/14

WIND	**Play = Yes**	**Play = No**	**Total**

Strong	**3/9**	**3/5**	**6/14**
Weak	**6/9**	**2/5**	**8/14**

Testing

For this, say we were given a new instance, and we want to know if we can play a game or not, then we need to lookup the results from the tables above. So, this new instance is:

X = (Outlook=Sunny, Temperature=Cool, Humidity=High, Wind=Strong)

Firstly we look at the probability that we can play the game, so we use the lookup tables to get:

P(Outlook=Sunny | Play=Yes) = 2/9
P(Temperature=Cool | Play=Yes) = 3/9
P(Humidity=High | Play=Yes) = 3/9
P(Wind=Strong | Play=Yes) = 3/9
P(Play=Yes) = 9/14

Next we consider the fact that we cannot play a game:

P(Outlook=Sunny | Play=No) = 3/5
P(Temperature=Cool | Play=No) = 1/5
P(Humidity=High | Play=No) = 4/5
P(Wind=Strong | Play=No) = 3/5
P(Play=No) = 5/14

Then, using those results, you have to multiple the whole lot together. So you multiple all the probabilities for Play=Yes such as:

P(X|Play=Yes)P(Play=Yes) = (2/9) * (3/9) * (3/9) * (3/9) * (9/14) = 0.0053

And this gives us a value that represents ‘P(X|C)P(C)’, or in this case ‘P(X|Play=Yes)P(Play=Yes)’.

We also have to do the same thing for Play=No:

P(X|Play=No)P(Play=No) = (3/5) * (1/5) * (4/5) * (3/5) * (5/14) = 0.0206

Finally, we have to divide both results by the evidence, or ‘P(X)’. The evidence for both equations is the same, and we can find the values we need within the ‘Total’ columns of the look-up tables. Therefore:

P(X) = P(Outlook=Sunny) * P(Temperature=Cool) * P(Humidity=High) * P(Wind=Strong)
P(X) = (5/14) * (4/14) * (7/14) * (6/14)
P(X) = 0.02186
Then, dividing the results by this value:

P(Play=Yes | X) = 0.0053/0.02186 = 0.2424
P(Play=No | X) = 0.0206/0.02186 = 0.9421

So, given the probabilities, can we play a game or not? To do this, we look at both probabilities and see which once has the highest value, and that is our answer. Therefore:

P(Play=Yes | X) = 0.2424
P(Play=No | X) = 0.9421

Since 0.9421 is greater than 0.2424 then the answer is 'no', we cannot play a game of tennis today.

Let's understand it using an example. Below I have a training data set of weather and corresponding target variable 'Play' (suggesting possibilities of playing). Now, we need to classify whether players will play or not based on weather condition. Let's follow the below steps to perform it.

Step 1: Convert the data set into a frequency table

Step 2: Create Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing is 0.64.

Step 3: Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9

Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

Naïve Bayes for PlayTennis dataset in python:

```
print("\nNaive Bayes Classifier for concept learning problem")
import csv
import random
import math
import operator
def safe_div(x,y):
if y == 0:
```

```
    return 0
    return x / y

    def loadCsv(filename):
            lines = csv.reader(open(filename))
            dataset = list(lines)
            for i in range(len(dataset)):
                    dataset[i] = [float(x) for x in dataset[i]]
            return dataset

    def splitDataset(dataset, splitRatio):
            trainSize = int(len(dataset) * splitRatio)
            trainSet = []
            copy = list(dataset)
            i=0
            while len(trainSet) < trainSize:
                    #index = random.randrange(len(copy))

                    trainSet.append(copy.pop(i))
            return [trainSet, copy]

    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

    def mean(numbers):
            return safe_div(sum(numbers),float(len(numbers)))

    def stdev(numbers):
            avg = mean(numbers)
            variance = safe_div(sum([pow(x-avg,2) for x in numbers]),float(len(numbers)-1))
            return math.sqrt(variance)

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

    def summarizeByClass(dataset):
```

```
        separated = separateByClass(dataset)
        summaries = {}
        for classValue, instances in separated.items():
                summaries[classValue] = summarize(instances)
        return summaries

def calculateProbability(x, mean, stdev):
        exponent = math.exp(-safe_div(math.pow(x-mean,2),(2*math.pow(stdev,2))))
        final = safe_div(1 , (math.sqrt(2*math.pi) * stdev)) * exponent
        return final

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

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

def getPredictions(summaries, testSet):
        predictions = []
        for i in range(len(testSet)):
                result = predict(summaries, testSet[i])
                predictions.append(result)
        return predictions

def getAccuracy(testSet, predictions):
        correct = 0
        for i in range(len(testSet)):
                if testSet[i][-1] == predictions[i]:
                        correct += 1
        accuracy = safe_div(correct,float(len(testSet))) * 100.0
        return accuracy
```

```
def main():
        filename = 'ConceptLearning.csv'
        splitRatio = 0.9
        dataset = loadCsv(filename)
        trainingSet, testSet = splitDataset(dataset, splitRatio)
        print('Split {0} rows into'.format(len(dataset)))
        print('Number of Training data: ' + (repr(len(trainingSet))))
        print('Number of Test Data: ' + (repr(len(testSet))))
        print("\nThe values assumed for the concept learning attributes are\n")
        print("OUTLOOK=> Sunny=1 Overcast=2 Rain=3\nTEMPERATURE=> Hot=1 Mild=2
Cool=3\nHUMIDITY=> High=1 Normal=2\nWIND=> Weak=1 Strong=2")
        print("TARGET CONCEPT:PLAY TENNIS=> Yes=10 No=5")
        print("\nThe Training set are:")
        for x in trainingSet:
                print(x)
        print("\nThe Test data set are:")
        for x in testSet:
                print(x)
        print("\n")
        # prepare model
        summaries = summarizeByClass(trainingSet)
        # test model
        predictions = getPredictions(summaries, testSet)
        actual = []
        for i in range(len(testSet)):
                vector = testSet[i]
                actual.append(vector[-1])
# Since there are five attribute values, each attribute constitutes to 20% accuracy. So if all attributes
#match with predictions then 100% accuracy
        print('Actual values: {0}%'.format(actual))
        print('Predictions: {0}%'.format(predictions))
        accuracy = getAccuracy(testSet, predictions)
        print('Accuracy: {0}%'.format(accuracy))

main()
```

The input CSV file is as below

Output:

Naive Bayes Classifier for concept learning problem

Split 16 rows into

Number of Training data: 14

Number of Test Data: 2

The values assumed for the concept learning attributes are

OUTLOOK=> Sunny=1 Overcast=2 Rain=3

TEMPERATURE=> Hot=1 Mild=2 Cool=3

HUMIDITY=> High=1 Normal=2

WIND=> Weak=1 Strong=2

TARGET CONCEPT:PLAY TENNIS=> Yes=10 No=5

The Training set are:

[1.0, 1.0, 1.0, 1.0, 5.0]

[1.0, 1.0, 1.0, 2.0, 5.0]

[2.0, 1.0, 1.0, 2.0, 10.0]

[3.0, 2.0, 1.0, 1.0, 10.0]

[3.0, 3.0, 2.0, 1.0, 10.0]

[3.0, 3.0, 2.0, 2.0, 5.0]

[2.0, 3.0, 2.0, 2.0, 10.0]

[1.0, 2.0, 1.0, 1.0, 5.0]

[1.0, 3.0, 2.0, 1.0, 10.0]

[3.0, 2.0, 2.0, 2.0, 10.0]

[1.0, 2.0, 2.0, 2.0, 10.0]

[2.0, 2.0, 1.0, 2.0, 10.0]

[2.0, 1.0, 2.0, 1.0, 10.0]

[3.0, 2.0, 1.0, 2.0, 5.0]

The Test data set are:

[1.0, 2.0, 1.0, 2.0, 5.0]

[1.0, 2.0, 1.0, 2.0, 5.0]

Actual values: [5.0, 5.0]%

Predictions: [5.0, 5.0]%

Accuracy: 100.0%

Program 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.

Introduction to Expectation-Maximization (EM)

The EM algorithm tends to get stuck less than K-means algorithm. The idea is to assign data points partially to different clusters instead of assigning to only one cluster. To do this partial assignment, we model each cluster using a probabilistic distribution So a data point associates with a cluster with certain probability and it belongs to the cluster with the highest probability in the final assignment.

Expectation-Maximization (EM) algorithm

Step 1: An initial guess is made for the model's parameters and a probability distribution is created. This is sometimes called the "E-Step" for the "Expected" distribution.

Step 2: Newly observed data is fed into the model.

Step 3: The probability distribution from the E-step is drawn to include the new data. This is sometimes called the "M-step."

Step 4: Steps 2 through 4 are repeated until stability.

Data set:

1	1
3	3
5	5
8	8
2	2
11	11
14	14
18	18

Expectation-Maximization in Python:

```
import csv
import math
import copy
k=2
```

```
class cluster:
    def init (self,cluster_head_data):
        self.head=cluster_head_data
        self.l=[]
        self.mean=cluster_head_data[0]
        self.variance=1.0
    def display_head(self):
        print("m=",self.mean)
        print("v=",self.variance)
    def add_ele_cluster(self,data):
        self.l.append(data)
        print(self.l)
    def display_ele(self):
        print('list contains',self.l)


def find_insert_individual_cluster(cluster,element,n):
    ele=float(element[0])
    prob=[]

    for i in range(len(cluster)):
        pxb= 1/math.sqrt(2*3.142*cluster[i].variance)* math.exp(-1* ( ele - float(cluster[i].mean)
)**2 / (2*float(cluster[i].variance)))
        print('pxb exact==',pxb)
        prob.append(pxb)
    print('prob elem',prob)
    bi_den=0
    for i in range(len(prob)):
        bi_den=bi_den+prob[i]*1/n
        print('bi den', bi_den)
    #insert ele in to the cluster-- ele+pxb+bi
    for i in range(len(cluster)):
        clust_data=[]
        clust_data.append(ele)
        bi=(prob[i]*1/n)/bi_den
        clust_data.append(bi)
        #add the contents on to the cluster
        cluster[i].add_ele_cluster(clust_data)

def recalculate_cluster_mean_variance(cluster):
    l1=cluster.l
```

```
        print('list enteries',l1)
        #recalculating mean
        mean_num=0.0
        mean_den=0.0
        var_num=0.0
        var_den=0.0
        for i in range(len(l1)):
            mean_num=mean_num+l1[i][0]*l1[i][1]
            mean_den=mean_den+l1[i][1]
        mean=mean_num/mean_den
        cluster.mean=mean
        #recalculating varaiance
        for i in range(len(l1)):
            var_num=var_num+l1[i][1]*(l1[i][0]-mean)**2
            var_den=var_den+l1[i][1]
        variance=var_num/var_den
        cluster.variance=mean

    def find_nearest(cluster,ele):
        ele=float(ele[0])
        prob=[]
        nearest_prob=None
        index=1
        for i in range(len(cluster)):
            pxb= 1/math.sqrt(2*3.142*cluster[i].variance)* math.exp(-1* ( ele - float(cluster[i].mean)
    )**2 / (2*float(cluster[i].variance)))
            print('pxb for cluster i',i,'=',pxb)
            if nearest_prob is None:
                nearest_prob=pxb
                index=i
            else:
                if nearest_prob < pxb:
                    nearest_prob=pxb
                    index=i
        print('index',index,'nearest_prob=',nearest_prob)
        cluster[index].l.append(ele)

    #read the contents of CSV file
    with open('cluster.csv') as csvfile:
        spamreader = csv.reader(csvfile, delimiter=',')
        #inserting elelments in to the list
```

```
    db=[]
    for row in spamreader:
        db.append(row)
    #creating individual cluster heads
#displaying elelments of the list
print('Db entries')
print(db)

#intialize the cluster
c=[]

#intialize the cluster heads
for i in range(k):
    new_clust=cluster(db[i])
    c.append(new_clust)
print('initial cluster Mean Variance')
#print cluster mean and variance
for i in range(k):
    print('----cluster',i,' ----- ')
    c[i].display_head()

error_ratio=1
# Iteration and including elelments in the cluster
while error_ratio>0:
    error_ratio=0
    prevc=copy.deepcopy(c)
    #estimation
    for i in range(len(db)):
        find_insert_individual_cluster(c,db[i],len(db))
    #recalculate cluster mean and variance
    for i in range(len(c)):
        recalculate_cluster_mean_variance(c[i])
    #display recalculated mean and varaiance of cluster
    for i in range(k):
        print('----cluster',i,' ----- ')
        c[i].display_head()
    #clear all the values of the cluster list for the next iteration
        for i in range(k):
            c[i].l=[]
    #calculate the error
        error_ratio=0
```

```
        for i in range(len(c)):
            if abs(c[i].variance - prevc[i].variance)>0.1:
                error_ratio=error_ratio+1
    #display the cluster elements
    #clear all the values of the cluster list for the next iteration
    for i in range(k):
        c[i].l=[]
    #claculate the nearest prob for each elelment and include it in the resultant cluster
    for i in range(len(db)):
        find_nearest(c,db[i])

    #display cluster elelments
    for i in range(len(c)):
        print(c[i].l)
```

Output

```
----cluster 0 ------
m= 2.7213340517024402
v= 2.7213340517024402
----cluster 1 ------
m= 12.504957731742246
v= 12.504957731742246
pxb for cluster i 0 = 0.14030103544739295
pxb for cluster i 1 = 0.0005673529507970876
index 0 nearest_prob= 0.14030103544739295
pxb for cluster i 0 = 0.23839358908896524
pxb for cluster i 1 = 0.003044550243209151
index 0 nearest_prob= 0.23839358908896524
pxb for cluster i 0 = 0.09314788538736465
pxb for cluster i 1 = 0.01186516649339238
index 0 nearest_prob= 0.09314788538736465
pxb for cluster i 0 = 0.0014456877959066348
pxb for cluster i 1 = 0.05011024116755676
index 1 nearest_prob= 0.05011024116755676
pxb for cluster i 0 = 0.2197718847381247
pxb for cluster i 1 = 0.0013678961391086488
index 0 nearest_prob= 0.2197718847381247
pxb for cluster i 0 = 8.216307351891605e-07
pxb for cluster i 1 = 0.10304124381579008
index 1 nearest_prob= 0.10304124381579008
```

```
pxb for cluster i 0 = 1.709934543851912e-11
pxb for cluster i 1 = 0.10316387232255203
index 1 nearest_prob= 0.10316387232255203
pxb for cluster i 0 = 5.707857566318809e-20
pxb for cluster i 1 = 0.033728682873350235
index 1 nearest_prob= 0.033728682873350235
[1.0, 3.0, 5.0, 2.0]
[8.0, 11.0, 14.0, 18.0]
>>>
```

K-Means in Python:

```
import csv
import math
import copy
k=3

class cluster:
    def init (self,cluster_head_data):
        self.head=cluster_head_data
        self.l=[]
    def display_head(self):
        print(self.head)
    def add_ele_cluster(self,data):
        self.l.append(data)
    def display_ele(self):
        print('list contains',self.l)

def compare_the_values(first,secound):
    x=float(first[0])
    y=float(first[1])
    x1=float(secound[0])
    x2=float(secound[1])
    val=math.sqrt( math.pow(math.fabs(x-x1),2)+math.pow(math.fabs(x-x1),2) )
    return val

def compare_the_nearest_cluster(cluster,data):
    #intialize the nearest cluster to 0
    dist_measure=None
    nearest=0
    for i in range(len(cluster)):
            dist=compare_the_values(cluster[i].head,data)
            if dist_measure is None:
```

```
                dist_measure=dist
                nearest=i
            if dist<dist_measure:
                dist_measure=dist
                nearest=i
    return nearest




def recal_head(cluster):
    for i in range(len(cluster)):
        l1=cluster[i].l
        xval=0.0
        yval=0.0
        for j in l1:
            xval=xval+float(j[0])
            yval=yval+float(j[1])
        xavg=xval/len(l1)
        yavg=yval/len(l1)
        avgl=[]
        avgl.append(xavg)
        avgl.append(yavg)
        cluster[i].head=avgl


#read the contents of CSV file
with open('cluster.csv') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')
    #inserting elelments in to the list
    db=[]
    for row in spamreader:
        db.append(row)
    #creating individual cluster heads
#displaying elelments of the list
print('Db entries')
print(db)

#intialize the cluster
c=[]

#intialize the cluster heads
for i in range(k):
    new_clust=cluster(db[i])
```

```
    c.append(new_clust)
print('initial cluster head values')
#print display heads
for i in range(k):
    print('----cluster',i,'----- ')
    c[i].display_head()



error_ratio=1
# Iteration and including elelments in the cluster
while error_ratio>0:
    prevc=copy.deepcopy(c)
    for ele in db:
        r=compare_the_nearest_cluster(c,ele)
        c[r].add_ele_cluster(ele)

    # display all the elements
    for clust in c:
        clust.display_ele()
    #recalculate the avg value
        recal_head(c)

    for i in range(k):
        print('----cluster',i,'------')
        c[i].display_head()
    #remove the ele of cluter head for the next iter
    for i in range(k):
        c[i].l=[]
    #calculate the error
        error_ratio=0
    for i in range(k):
        if c[i].head != prevc[i].head:
            error_ratio=error_ratio+1
#final cluster ele
```

Output

```
Db entries
[['1', '1'], ['3', '3'], ['5', '5'], ['8', '8'], ['2', '2'], ['11', '11'], ['14', '14'], ['18', '18']]
initial cluster head values
----cluster 0 ------
['1', '1']
----cluster 1 ------
```

```
['3', '3']
----cluster 2 ------
['5', '5']
list contains [['1', '1'], ['2', '2']]
list contains [['3', '3']]
list contains [['5', '5'], ['8', '8'], ['11', '11'], ['14', '14'], ['18', '18']]
----cluster 0 ------
[1.5, 1.5]
----cluster 1 ------
[3.0, 3.0]
----cluster 2 ------
[11.2, 11.2]
list contains [['1', '1'], ['2', '2']]
list contains [['3', '3'], ['5', '5']]
list contains [['8', '8'], ['11', '11'], ['14', '14'], ['18', '18']]
----cluster 0 ------
[1.5, 1.5]
----cluster 1 ------
[4.0, 4.0]
----cluster 2 ------
[12.75, 12.75]
list contains [['1', '1'], ['2', '2']]
list contains [['3', '3'], ['5', '5'], ['8', '8']]
list contains [['11', '11'], ['14', '14'], ['18', '18']]
----cluster 0 ------
[1.5, 1.5]
----cluster 1 ------
[5.333333333333333, 5.333333333333333]
----cluster 2 ------
[14.333333333333334, 14.333333333333334]
list contains [['1', '1'], ['3', '3'], ['2', '2']]
list contains [['5', '5'], ['8', '8']]
list contains [['11', '11'], ['14', '14'], ['18', '18']]
----cluster 0 ------
[2.0, 2.0]
----cluster 1 ------
[6.5, 6.5]
----cluster 2 ------
[14.333333333333334, 14.333333333333334]
list contains [['1', '1'], ['3', '3'], ['2', '2']]
list contains [['5', '5'], ['8', '8']]
```

```
list contains [['11', '11'], ['14', '14'], ['18', '18']]
----cluster 0 ------
[2.0, 2.0]
----cluster 1 ------
[6.5, 6.5]
----cluster 2 ------
[14.333333333333334, 14.333333333333334]
>>>
```

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.

TASK: The task of this program is to classify the IRIS data set examples by using the k-Nearest Neighbour algorithm. The new instance has to be classified based on its k nearest neighbors.

KNN falls in the supervised learning family of algorithms. Informally, this means that we are given a labeled dataset consisting of training observations (x,y) and would like to capture the relationship between x and y. More formally, our goal is to learn a function h:X→Y so that given an unseen observation x, h(x) can confidently predict the corresponding output y.
The KNN classifier is also a non parametric and instance-based learning algorithm.

- Non-parametric means it makes no explicit assumptions about the functional form of h, avoiding the dangers of mismodeling the underlying distribution of the data. For example, suppose our data is highly non-Gaussian but the learning model we choose assumes a Gaussian form. In that case, our algorithm would make extremely poor predictions.
- Instance-based learning means that our algorithm doesn't explicitly learn a model. Instead, it chooses to memorize the training instances which are subsequently used as "knowledge" for the prediction phase. Concretely, this means that only when a query to our database is made (i.e. when we ask it to predict a label given an input), will the algorithm use the training instances to spit out an answer.

In the classification setting, the K-nearest neighbor algorithm essentially boils down to forming a majority vote between the K most similar instances to a given "unseen" observation. Similarity is defined according to a distance metric between two data points. A popular choice is the Euclidean distance given by

$$d(x, x') = \sqrt{(x_1 - x'_1)^2 + (x_2 - x'_2)^2 + \ldots + (x_n - x'_n)^2}$$

but other measures can be more suitable for a given setting and include the Manhattan, Chebyshev and Hamming distance.
More formally, given a positive integer K, an unseen observation x and a similarity metric d, KNN classifier performs the following two steps:

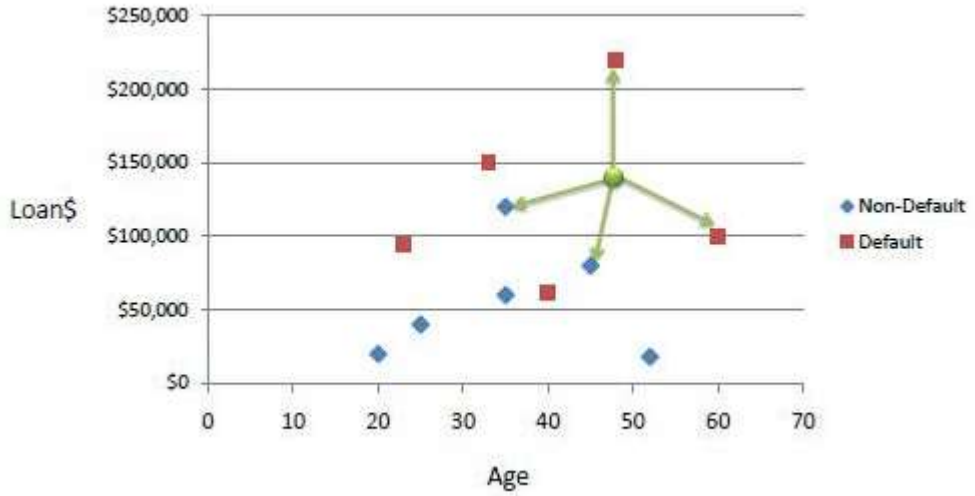
- It runs through the whole dataset computing d between x and each training observation. The K points in the training data that are closest to x are called the set A. Note that K is usually odd to prevent tie situations.
- It then estimates the conditional probability for each class, that is, the fraction of points in A with that given class label. (Note I(x) is the indicator function which evaluates to 1 when the argument x is true and 0 otherwise)

$$P(y = j|X = x) = \frac{1}{K} \sum_{i \in A} I(y^{(i)} = j)$$

Finally, the input x gets assigned to the class with the largest probability.

EXAMPLE:

Consider the following data concerning credit default. Age and Loan are two numerical variables (predictors) and Default is the target.



We can now use the training set to classify an unknown case (Age=48 and Loan=$142,000) using Euclidean distance. If K=1 then the nearest neighbor is the last case in the training set with Default=Y.

D = Sqrt[(48-33)^2 + (142000-150000)^2] = 8000.01 >> Default=Y

Age	Loan	Default	Distance	
25	$40,000	N	102000	
35	$60,000	N	82000	
45	$80,000	N	62000	
20	$20,000	N	122000	
35	$120,000	N	22000	2
52	$18,000	N	124000	
23	$95,000	Y	47000	
40	$62,000	Y	80000	
60	$100,000	Y	42000	3
48	$220,000	Y	78000	
33	$150,000	Y	8000	1
48	**$142,000**	**?**		

Euclidean Distance $D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

With K=3, there are two Default=Y and one Default=N out of three closest neighbors.
The prediction for the unknown case is again Default=Y.

ALGORITHM:

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

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.

KNN in Python:

```
import csv
import random
import math
import operator
def loadDataset(filename, split, trainingSet=[], testSet=[]):
        with open(filename) as csvfile:
        lines = csv.reader(csvfile)
        dataset = list(lines)
        for x in range(len(dataset)-1):
        for y in range(4):
        dataset[x][y] = float(dataset[x][y])
        if random.random() < split:
        trainingSet.append(dataset[x])
        else:
        testSet.append(dataset[x])
def euclideanDistance(instance1, instance2, length):
        distance = 0
        for x in range(length):
                distance += pow((instance1[x] - instance2[x]), 2)
        return math.sqrt(distance)

def getNeighbors(trainingSet, testInstance, k):
        distances = []
        length = len(testInstance)-1
        for x in range(len(trainingSet)):
                dist = euclideanDistance(testInstance, trainingSet[x], length)
                distances.append((trainingSet[x], dist))
        distances.sort(key=operator.itemgetter(1))
```

```
        neighbors = []
        for x in range(k):
                neighbors.append(distances[x][0])
        return neighbors

def getResponse(neighbors):
        classVotes = {}
        for x in range(len(neighbors)):
                response = neighbors[x][-1]
                if response in classVotes:
                        classVotes[response] += 1
                else:
                        classVotes[response] = 1
        sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reverse=True)
        return sortedVotes[0][0]

def getAccuracy(testSet, predictions):
        correct = 0
        for x in range(len(testSet)):
                if testSet[x][-1] == predictions[x]:
                        correct += 1
        return (correct/float(len(testSet))) * 100.0

def main():
        # prepare data
        trainingSet=[]
        testSet=[]
        split = 0.67
        loadDataset('iris_data.csv', split, trainingSet, testSet)
        print ('\n Number of Training data: ' + (repr(len(trainingSet))))
        print (' Number of Test Data: ' + (repr(len(testSet))))
        # generate predictions
        predictions=[]
        k = 3
        print('\n The predictions are: ')
        for x in range(len(testSet)):
                neighbors = getNeighbors(trainingSet, testSet[x], k)
                result = getResponse(neighbors)
                predictions.append(result)
                print(' predicted=' + repr(result) + ', actual=' + repr(testSet[x][-1]))
        accuracy = getAccuracy(testSet, predictions)
        print('\n The Accuracy is: ' + repr(accuracy) + '%')

main()
```

Output:

```
Number of Training data: 106
Number of Test Data: 43
The predictions are:
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
```

```
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
The Accuracy is: 100.0%
```

Program 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.

Nonparametric regression: is a category of regression analysis in which the predictor does not take a predetermined form but is constructed according to information derived from the data. Nonparametric regression requires larger sample sizes than regression based on parametric models because the data must supply the model structure as well as the model estimates.

Nonparametric regression is used for prediction and is reliable even if hypotheses of linear regression are not verified.

Locally weighted Learning also known as memory-based learning, instance-based learning, lazy-learning, and closely related to kernel density estimation, similarity searching and case-based reasoning.

LOWESS (Locally Weighted Scatterplot Smoothing), sometimes called LOESS (locally weighted smoothing), is a popular tool used in regression analysis that creates a smooth line through a timeplot or scatter plot to help you to see relationship between variables and foresee trends.

Locally weighted regression is a very powerful non-parametric model used in statistical learning.

Introduction :

Scatter-diagram smoothing (e.g. using the **lowess()** or **loess()** functions) involves drawing a smooth curve on a scatter diagram to summarize a relationship, in a fashion that makes few assumptions initially about the form or strength of the relationship. It is related to (and is a special case of) *nonparametric regression*, in which the objective is to represent the relationship between a response variable and one or more predictor variables, again in way that makes few assumptions about the form of the relationship. In other words, in contrast to "standard" linear regression analysis, no assumption is made that the relationship is represented by a straight line (although one could certainly think of a straight line as a special case of nonparametric regression).

If the basic decomposition-of-the-data model is:

data = predictable component + noise,

then for the standard bivariate or multiple (linear) regression, the model is

data = straight-line, polynomial or linearizable function + noise,

while for nonparametric regression, the model is

data = smooth function determined by data + noise.

Another way of looking at scatter diagram smoothing is as a way of depicting the "local" relationship between a response variable and a predictor variable over parts of their ranges, which may differ from a "global" relationship determined using the whole data set. Nonparametric regression can be thought of as generalizing the scatter plot smoothing idea to the multiple-regression context.

Locally Weighted Learning is a class of function approximation techniques, where a prediction is done by using an approximated local model around the current point of interest.

The **goal** of function approximation and regression is to find the underlying relationship between input and output. In a supervised learning problem training data, where each input is associated to one output, is used to create a model that predicts values which come close to the true function. All of these models use complete training data to derive global function.

Locally weighted regression

Local means using nearby points (i.e. a nearest neighbors approach)

Weighted means we value points based upon how far away they are.

Regression means approximating a function

This is **an instance-based learning method**

The idea: whenever you want to classify a sample:

- Build a local model of the function (using a linear function, quadratic, neural network, etc.)
- Use the model to predict the output value
- Throw the model away.

Locally Weighted Regression

Our final method combines advantages of parametric methods with non-parametric. The idea is to fit a regression model locally, weighting examples by the kernel K.

Locally Weighted Regression Algorithm

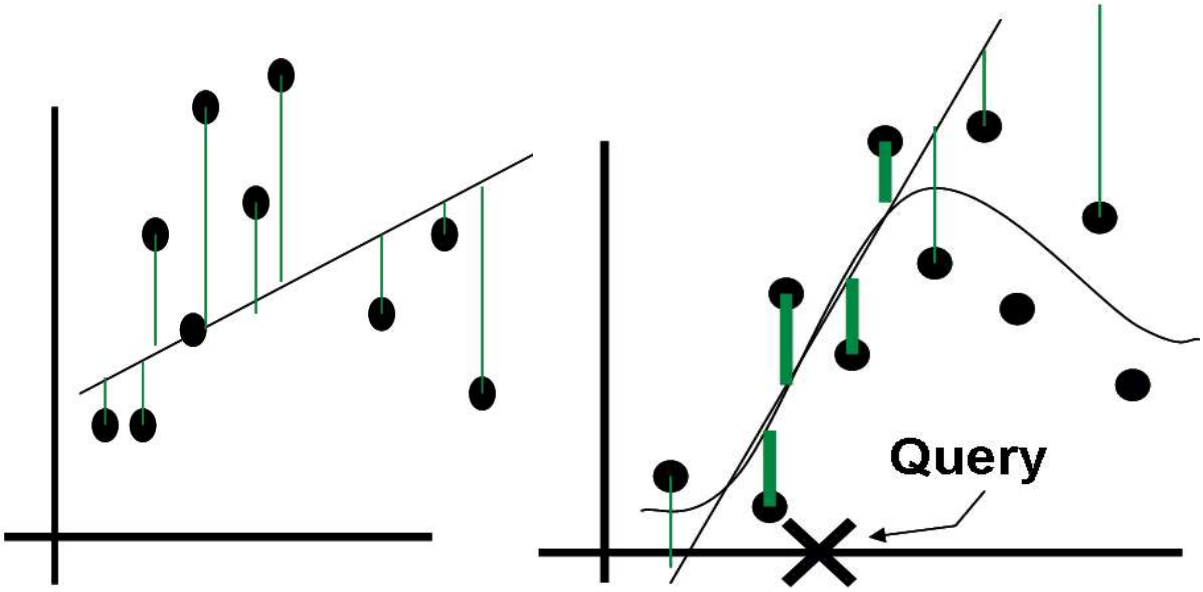
1. Given training data $D=\{\mathbf{x}i\ yi\}$, Kernel function $K(\ ,\ )$ and input $\mathbf{x}$
2. Fit weighted regression $\mathbf{w}\ (\mathbf{x})=\text{argmin}w\ \sum ni=1K(\mathbf{x}\ \mathbf{x}i)(\mathbf{w}\ \mathbf{x}i-yi)2$
3. Return regression prediction $\mathbf{w}\ (\mathbf{x})\ \mathbf{x}$

Note that we can do the same for classification, fitting a locally weighted logistic regression:

Locally Weighted Logistic Regression Algorithm

1. Given training data $D=\{\mathbf{x}i\ yi\}$, Kernel function $K(\ ,\ )$ and input $\mathbf{x}$
2. Fit weighted logistic regression $\mathbf{w}\ (\mathbf{x})=\text{argmin}w\ \sum ni=1K(\mathbf{x}\ \mathbf{x}i)\log(1+\exp\{-yi\mathbf{w}\ \mathbf{x}i\})$
3. Return logistic regression prediction $sign(\mathbf{w}\ (\mathbf{x})\ \mathbf{x})$.

The difference between regular linear regression and locally weighted linear regression can be visualized as follows:



x ---- is an instance,

D ---- is the set of possible instances

ai(x)-----is the value of the i th attribute

wi----- The weights form our hypothesis

f---- is the target function

f ˆ ----is our approximation to the target function

In this case, we use a linear model to do the local approximation f ˆ :

f ̂(x) = w0 + w1a1(x) + ⋯⋯+ wnan(x)

LOWESS is typically used for:

- Fitting a line to a scatter plot or time plot where noisy data values, sparse data points or weak interrelationships interfere with your ability to see a line of best fit.
- Linear regression where least squares fitting doesn't create a line of good fit or is too labor-intensive to use.
- Data exploration and analysis in the social sciences, particularly in elections and voting behavior.

Parametric and Non-Parametric Fitting

LOWESS, and least squares fitting in general, are **non-parametric** strategies for fitting a smooth curve to data points. "Parametric" means that the researcher or analyst assumes in advance that the data fits

some type of distribution (i.e. the normal distribution). Because some type of distribution is assumed in advance, parametric fitting can lead to fitting a smooth curve that misrepresents the data. In those cases, non-parametric smoothers may be a better choice. Non-parametric smoothers like LOESS try to find a curve of best fit without assuming the data must fit some distribution shape. In general, both types of smoothers are used for the same set of data to offset the advantages and disadvantages of each type of smoother.

Benefits of Non-Parametric Smoothing

- Provides a flexible approach to representing data.
- Ease of use.
- Computations are relatively easy.

Disadvantages of Non-Parametric Smoothing

- Can't be used to obtain a simple equation for a set of data.
- Less well understood than parametric smoothers.
- Requires the analyst to use a little guesswork to obtain a result.

Locally Weighted Regrassion in Python:

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


def lowess(x, y, f=2./3., iter=3):
    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(iter):
        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
```

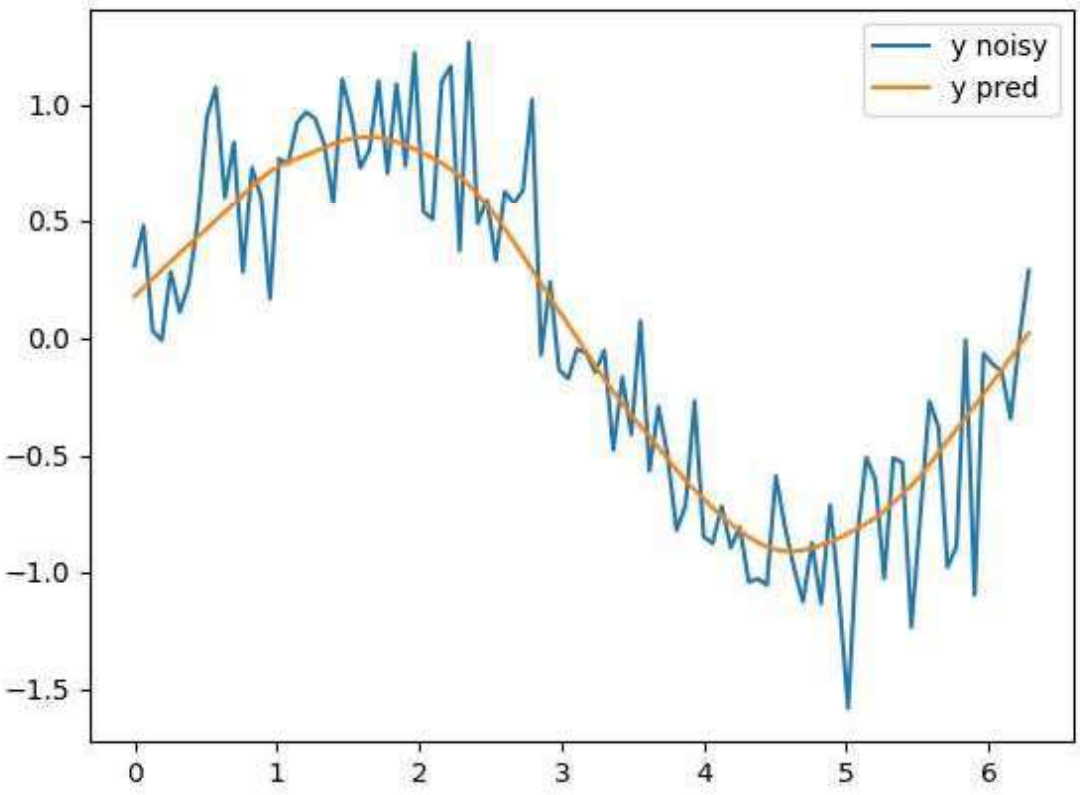
```
if__name__== '_main_':
    import math
    n = 100
    x = np.linspace(0, 2 * math.pi, n)
    print("==========================values of x====================")
    print(x)
    y = np.sin(x) + 0.3*np.random.randn(n)
    print("==================================Values of y==================")
    print(y)
    f = 0.25
    yest = lowess(x, y, f=f, iter=3)

    import pylab as pl
    pl.clf()
    pl.plot(x, y, label='y noisy')
    pl.plot(x, yest, label='y pred')
    pl.legend()
    pl.show()
```

Output

```
==========================values of x====================
[0.         0.06346652 0.12693304 0.19039955 0.25386607 0.31733259
 0.38079911 0.44426563 0.50773215 0.57119866 0.63466518 0.6981317
 0.76159822 0.82506474 0.88853126 0.95199777 1.01546429 1.07893081
 1.14239733 1.20586385 1.26933037 1.33279688 1.3962634 1.45972992
 1.52319644 1.58666296 1.65012947 1.71359599 1.77706251 1.84052903
 1.90399555 1.96746207 2.03092858 2.0943951 2.15786162 2.22132814
 2.28479466 2.34826118 2.41172769 2.47519421 2.53866073 2.60212725
 2.66559377 2.72906028 2.7925268 2.85599332 2.91945984 2.98292636
 3.04639288 3.10985939 3.17332591 3.23679243 3.30025895 3.36372547
 3.42719199 3.4906585  3.55412502 3.61759154 3.68105806 3.74452458
 3.8079911  3.87145761 3.93492413 3.99839065 4.06185717 4.12532369
 4.1887902 4.25225672 4.31572324 4.37918976 4.44265628 4.5061228
 4.56958931 4.63305583 4.69652235 4.75998887 4.82345539 4.88692191
 4.95038842 5.01385494 5.07732146 5.14078798 5.2042545 5.26772102
 5.33118753 5.39465405 5.45812057 5.52158709 5.58505361 5.64852012
 5.71198664 5.77545316 5.83891968 5.9023862 5.96585272 6.02931923
 6.09278575 6.15625227 6.21971879 6.28318531]
==================================Values of y==================
[-0.12131282 0.00704817 0.16084684 0.48415764  0.5790074       0.53758041
  0.41471799  0.31435684   0.42541349 0.56873662 0.64857746  0.56416606
  0.66906299  0.62626102   0.78385793 0.52007101 0.57661707  0.50246257
  0.59223656  1.19576243    0.9781296 0.76416587 0.85825567  0.68360755
```

```
 0.88965032 0.63976862 0.90159422 0.87464446 1.60256473 1.26886601
 0.4200674 1.43656644 0.48378272 1.20891979 1.02955476 0.85766616
 0.96519144  0.58303717  0.74996692  0.36357818  0.38481967  0.87027853
 0.11686969  0.15356422  0.30403899  0.28014845 -0.15378752  0.56020242
 0.20123909 -0.68626527 -0.19912738 -0.20357156 -0.1440503  -0.26493741
-0.32157543 -0.22139059 -0.42434559 -0.11447119 -0.67075934 -0.62973277
-1.18204429 -0.43386851 -1.12424791 -0.80806532 -0.85818104 -1.14149911
-0.23009694 -0.95399329 -0.62350893 -0.29176666 -0.55523556 -0.87957338
-1.78651723 -0.55784149 -1.1570492 -0.91445927 -0.99808369 -1.05836202
-0.55710891 -0.57815623 -0.60534368 -1.23588355 -1.34863514 -1.22421771
-1.01493553 -1.22922209 -0.48436674 -0.25055899 -0.61074483 -0.69197299
-0.52765904 -0.31451055 -0.55313683 -0.03399613 -0.53184918  0.33832978
 -0.28385608 0.03227412 -0.38242201 0.1601836 ]
>>>
```



Additional Lab Exercise

1. Implement a Breadth First Search Concepts using python programming
2. Implement a Depth First Search Concepts using python programming
3. Implement any two AI technique for tic-tac-toe problem using python programming
4. Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.
5. Implement a Support Vector Machine (SVM) concepts using Python.