TOPIC 1 SOLUTION:
CLASSIFICATION, NEURAL COMPUTING, AND SEARCH

1. Introduction

1.1. When to use BFS Algorithm :

Is used to solve many problems including finding the shortest path in a graph and solving puzzle games as Rubik's Cubes and also Graph Problems as analyzing networks, mapping routes, and scheduling.

1.2. When To use NN Algorithm :

is that once trained, they learn on their own. In this way, they emulate human brains, which are made up of neurons, the fundamental building block of both human and neural network information transmission as Learn Machine How to can expect the right result for a specific problem.

1.3. When to use ID3 Algorithm:

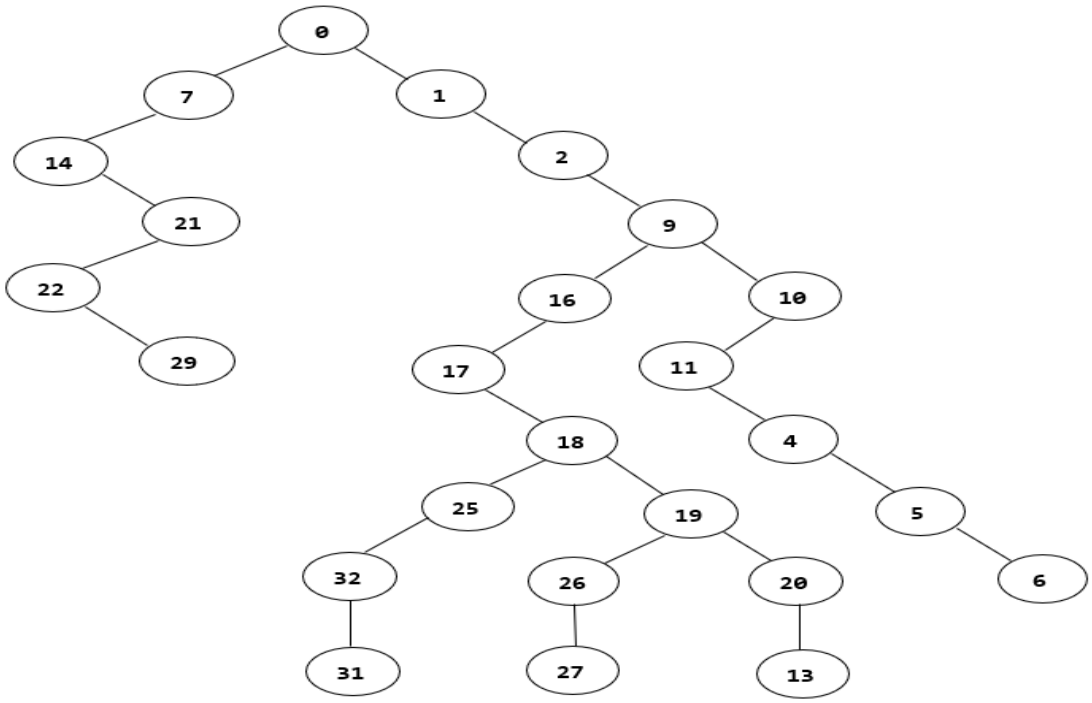
Is used to build the fastest and shortest tree by testing enough attributes until all data is classified.

2. The Algorithms

1. Breadth-first Search

1.1. The main steps of the algorithm

- Get the node in *master_list* [the one that included all nodes]
- Then Get the child of this node *up, down, left, right*
- Then Check if one of them exist in *master_list* do not add it If not add it to also get its children and *assign it to parent*
- Do the same For *all children* for a *node*
- Check if *end node* reached or not If reached stop repeating and return the *Full path* and *path* and If not *repeat*
- **TIP :** instead of making visited and unvisited list I only make one list [*master_list*] and add, check, and do all operation I want on it and it gives the same solution [just check the graph Below]



[BFS Tree Graph]

1.2. The implementation of the algorithm

```
class Node:
    # Defined Attributes
    id = None
    up = None
    down = None
    left = None
    right = None
    previousNode = None

    # Additional Attributes
    node_counter = 0

    # constructor
    def __init__(self, value):
        self.id = value

class SearchAlgorithms:
    # Defined Attributes
    path = []
    fullPath = []

    # Additional Attributes
    rows = 1
    columns = 1
    maze_board = []             # represent the board that will be created
    master_list = [0]           # tmp list to make operations in BFS algorithm included start ID 0 For S
    nodes_object = []           # represent list of object
    blocks = []                 # represent the IDs for [#] in string

    def __init__(self, mazeStr):
        # Counting number of columns and Rows
        self.GetColumns(mazeStr)
        self.GetRows(mazeStr)

        # Drawing Maze Board
        self.PrintMazeBoard(mazeStr)

        # Counting Num of Nodes in The Maze and Save their IDs
        self.PassMazeIDs()

        # Get Nodes Directions
        self.GetNodesDirections()
```

```
def BFS(self):
    """
    --> How it Works ?
    Apply BFS Algorithm But there is some thing to keep up in BFS i should work with 2 lists
    visited and unvisited lists but here its different here i work with one list adding child
    and checking unvisited child in same time in other meaning check for node and then adding
    their child(s) to list then checking the added child(s) also and so on ..

    :return: Full path [BFS Traverse] and path [Best Way to Go From S to E]
    """
    start_node = 0          # 0 Represent S
    end_node = 31           # 31 Represent E
    index = start_node

    while index < len(self.master_list):
        node = self.master_list[index]

        # Get Direction [Child] for Node
        up = self.nodes_object[node].up
        down = self.nodes_object[node].down
        left = self.nodes_object[node].left
        right = self.nodes_object[node].right


        # Get the direction(s) and define it as a child for that Node
        # also Get for this child its child and so on until reaching to Goal
        if up is not None and up not in self.master_list:
            self.master_list.append(up)
            self.nodes_object[up].previousNode = node

        if down is not None and down not in self.master_list:
            self.master_list.append(down)
            self.nodes_object[down].previousNode = node

        if left is not None and left not in self.master_list:
            self.master_list.append(left)
            self.nodes_object[left].previousNode = node

        if right is not None and right not in self.master_list:
            self.master_list.append(right)
            self.nodes_object[right].previousNode = node

        # Stop if we Reach to Goal
        if node == end_node or up == end_node or down == end_node or left == end_node or right == end_node:
            break
        index += 1

    # then the master list consider the Full path after Applying BFS
    self.fullPath = self.master_list

    # Then Get the direct path from start to goal
    self.path.append(end_node)
    helper_attr = end_node
    while helper_attr != start_node:
        helper_attr = self.nodes_object[helper_attr].previousNode
        self.path.append(helper_attr)
    self.path.reverse()

    # return Paths
    return self.fullPath, self.path
```

```
# Additional Functions
def GetColumns(self, mazeStr):
    """
    --> How it Works?
    get the given maze string and calculating number of Columns

    :param mazeStr: represent maze in string
    :return: void Just set the columns number
    """
    for char in mazeStr:
        if char == ',':
            self.columns += 1
        elif char == ' ':
            break

def GetRows(self, mazeStr):
    """
    --> How it Works?
    get the given maze string and calculating number of rows

    :param mazeStr: represent maze in string
    :return: void Just set the rows number
    """
    for char in mazeStr:
        if char == ' ':
            self.rows += 1

def PrintMazeBoard(self, mazeStr):
    """
    --> How it Works?
    get the given maze string and printing the maze board
    by putting each character in ID array then reshaping it
    according to number of columns

    :param mazeStr: represent maze in string
    :return: void Just set the columns number
    """
    tmp_maze = []
    for char in mazeStr:
        if char != ',' and char != ' ':
            tmp_maze.append(char)
    self.maze_board = np.reshape(tmp_maze, (-1, self.columns))

    print("The maze Board : ")
    for row in range(self.rows):
        for node in self.maze_board[row]:
            print(node, end=" ")
        print()
    print()

def PassMazeIDs(self):
    """
    --> How it Works?
    give each node an ID included blocks but also add
    it to blocked list to avoid using in applying BFS

    :return: void Just Set the IDs and Block List
    """
    for row in range(self.rows):
        for node in self.maze_board[row]:
            if node == '#':
                self.blocks.append(Node.node_counter)
            self.nodes_object.append(Node(Node.node_counter))
            Node.node_counter += 1
```

```
def GetNodesDirections(self):
    """
    --> How it Works?
    get the moves that allowed for any node [char] by some conditions
    will be explained in its part

    :return: void just setting the node directions
    """
    for count in range(Node.node_counter):
        # check if this Node is Block [#] or Not
        if self.nodes_object[count].id not in self.blocks:

            # get up, down, left and right directions for node [for any case]
            self.nodes_object[count].up = self.nodes_object[count].id - self.columns
            self.nodes_object[count].down = self.nodes_object[count].id + self.columns

            # Except left and right the next lines of code works as
            # if that node in first line so the left for it will be the
            # last node in previous line according to [self.nodes_object[count].id - 1]
            # but this not valid so we put this check and also for right.
            # -1000 --> just unique value to check later
            if self.nodes_object[count].id % self.columns != 0:
                self.nodes_object[count].left = self.nodes_object[count].id - 1
            else:
                self.nodes_object[count].left = -1000
            if (self.nodes_object[count].id + 1) % self.columns != 0:
                self.nodes_object[count].right = self.nodes_object[count].id + 1
            else:
                self.nodes_object[count].right = -1000

            # checking if any node thought the range of maze or Not
            if self.nodes_object[count].up > Node.node_counter-1 or self.nodes_object[count].up < 0:
                self.nodes_object[count].up = None
            if self.nodes_object[count].down > Node.node_counter-1 or self.nodes_object[count].down < 0:
                self.nodes_object[count].down = None
            if self.nodes_object[count].left > Node.node_counter-1 or self.nodes_object[count].left < 0 \
                    or self.nodes_object[count].left == -1000:
                self.nodes_object[count].left = None
            if self.nodes_object[count].right > Node.node_counter-1 or self.nodes_object[count].right < 0 \
                    or self.nodes_object[count].right == -1000:
                self.nodes_object[count].right = None

            # checking if any side is Block
            if self.nodes_object[count].up in self.blocks:
                self.nodes_object[count].up = None
            if self.nodes_object[count].down in self.blocks:
                self.nodes_object[count].down = None
            if self.nodes_object[count].left in self.blocks:
                self.nodes_object[count].left = None
            if self.nodes_object[count].right in self.blocks:
                self.nodes_object[count].right = None
```

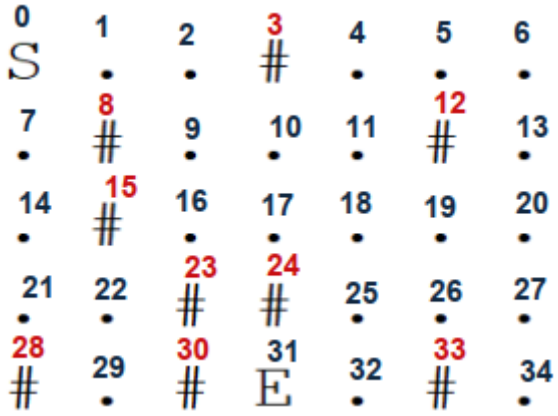
1.3. Sample run

```
The maze Board :
S . . # . . .
. # . . . # .
. # . . . . .
. . # # . . .
# . # E . # .

**BFS**
 Full Path is: [0, 7, 1, 14, 2, 21, 9, 22, 16, 10, 29, 17, 11, 18, 4, 25, 19, 5, 32, 26, 20, 6, 31]
 Path: [0, 1, 2, 9, 16, 17, 18, 25, 32, 31]
```

- **Additional Attributes and Functions**
  Full of Description of each exist in Code [Topic 1] File
  - in Class Node :
    ```
    node_counter = 0
    ```
  - in Class Search Algorithm :
    ```
    rows = 1
    columns = 1
    maze_board = []
    master_list = [0]
    nodes_objects = []
    blocks = []

    def GetColumns(self, MazeStr)
    def GetRows(self, MazeStr)
    def PrintMazeBoard(self, MazeStr)
    def PassMazeIDs(self)
    def GeNodesDirections(self)
    ```
  - Maze Board with Each Node ID



2. Neural Network

2.1. The main steps of the algorithm.

- According to random weight and Given input calculate *f(x)* using equation $f(x) = \Sigma\, Xi\, Wi$ for i = 1 to n [2 in our case].
- Then check is *f(x)* less or greater than threshold if greater than threshold *return 1* and if less than threshold *return 0*.
- Then calculate the error *E* using equation $E = y - f(x)$
- Finally update weight using equation $Wj = Wj + \alpha\, Xj\, E$

2.2. The implementation of Algorithm

```
# region NeuralNetwork
class NeuralNetwork():

    # the Constructor
    def __init__(self, learning_rate, threshold):
        self.learning_rate = learning_rate
        self.threshold = threshold
        self.synaptic_weights = 2 * np.random.random((2, 1)) - 1

    def step(self, x):
        """
        :param x: represent the net input [real output f(x)]
        :return: 1 if x >= threshold or 0 if not
        """
        if x >= self.threshold:
            return 1
        return 0

    def train(self, training_inputs, training_outputs, training_iterations):
        """
        :param training_inputs: represent the gate AND inputs
        :param training_outputs: represent the gate AND expected outputs [y]
        :param training_iterations: represent num of iterations allowed to learn the machine
        :return: void just if error exist assign new weight
        """
        for iteration in range(training_iterations):
            # call function think to get the real output f(x) [represent net input]
            output = self.think(training_inputs)

            # then Calculate the error using equation E = y - f(x)
            error = training_outputs - output

            # then update weight using Equation Wj = Wj + α Xj E
            alpha_by_e = error * self.learning_rate
            self.synaptic_weights += np.dot(training_inputs.transpose(), alpha_by_e)
```

```
def think(self, inputs):
    """
    :param inputs: represent the gate AND inputs
    :return: 0 or 1 according to threshold Rulex
    """
    # applying equation f(x) = Σ Xi Wi for i = 1 to n [2 in our case]
    fx = np.sum(np.dot(inputs, self.synaptic_weights))

    # then check it according to threshold Rule
    output = self.step(fx)

    # then return the returned value
    return output
```

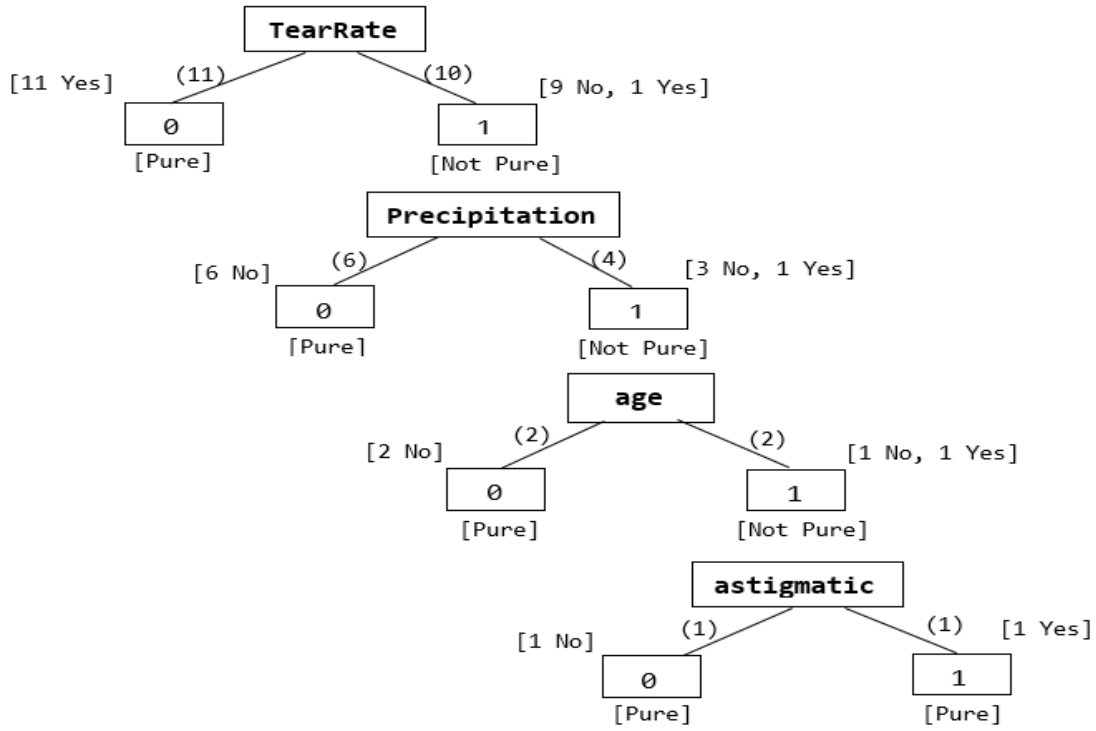
2.3. Simple Run

```
Beginning Randomly Generated Weights:
[[0.18352724]
 [0.2329725 ]]
Ending Weights After Training:
[[-0.01647276]
 [ 0.0329725 ]]
Considering New Situation:  1 1 New Output data:  1
Wow, we did it!
```

3. ID3

3.1. The main steps of the algorithm.

- Count number of Yes and No in parent as *age, prescription* … etc, and child as *0,1* according to *needLenses* feature.
- Calculate *entropy* for each parent and its child then calculate the *gain*.
- Choosing the best attribute to split according to *max* gain
- For this attribute determine *its child*
- Check *if* it classified well *[all pure]* then Stop which means we construct the decision tree and *if not* classified well *[Not pure exist]* then *split again* over new child.



[Decision Tree Graph]

3.2.The Implementation of algorithm.

```
# region ID3
class item:
    def __init__(self, age=None, prescription=None, astigmatic=None, tearRate=None, diabetic=None, needLense=None):
        self.age = age
        self.prescription = prescription
        self.astigmatic = astigmatic
        self.tearRate = tearRate
        self.diabetic = diabetic
        self.needLense = needLense

    # master data set [Given]
    def getDataset(self):
        data = []
        labels = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0]
        data.append(item(0, 0, 0, 0, 1, labels[0]))  # 0
        data.append(item(0, 0, 0, 1, 1, labels[1]))  # 1
        data.append(item(0, 0, 1, 0, 1, labels[2]))  # 0
        data.append(item(0, 0, 1, 1, 1, labels[3]))  # 1
        data.append(item(0, 1, 0, 0, 1, labels[4]))  # 0
        data.append(item(0, 1, 0, 1, 1, labels[5]))  # 1
        data.append(item(0, 1, 1, 0, 1, labels[6]))  # 0
        data.append(item(0, 1, 1, 1, 1, labels[7]))  # 1
        data.append(item(1, 0, 0, 0, 1, labels[8]))  # 0
        data.append(item(1, 0, 0, 1, 1, labels[9]))  # 1
        data.append(item(1, 0, 1, 0, 1, labels[10]))  # 0
        data.append(item(1, 0, 1, 1, 0, labels[11]))  # 1
        data.append(item(1, 1, 0, 0, 0, labels[12]))  # 0
        data.append(item(1, 1, 0, 1, 0, labels[13]))  # 1
```

```
        data.append(item(1, 1, 1, 0, 0, labels[14]))  # 0
        data.append(item(1, 1, 1, 1, 0, labels[15]))  # 0
        data.append(item(1, 0, 0, 0, 0, labels[16]))  # 0
        data.append(item(1, 0, 0, 1, 0, labels[17]))  # 1
        data.append(item(1, 0, 1, 0, 0, labels[18]))  # 0
        data.append(item(1, 0, 1, 1, 0, labels[19]))  # 1
        data.append(item(1, 1, 0, 0, 0, labels[20]))  # 0
        return data

    # split data set function
    def SplitDataSet(self, previous_data_set, parent, child):
        """
         --> How it works ?
         according to parent and its child split the data set in other meaning
         add this object to list which form the new data set to work with and
         it used to overcome if data Set not pure

        :param previous_data_set: represent the data set want to work with [list of objects]
        :param parent: represent the feature name as age, prescription ... ect
        :param child: represent the value of this child which are 0 or 1
        :return: the new data set after splitting
        """
        new_data_set = []
        if parent == 'age':
            for rec in previous_data_set:
                if rec.age == child:
                    new_data_set.append(rec)
        elif parent == 'prescription':
            for rec in previous_data_set:
                if rec.prescription == child:
                    new_data_set.append(rec)

        elif parent == 'astigmatic':
            for rec in previous_data_set:
                if rec.astigmatic == child:
                    new_data_set.append(rec)

        elif parent == 'tearRate':
            for rec in previous_data_set:
                if rec.tearRate == child:
                    new_data_set.append(rec)

        elif parent == 'diabetic':
            for rec in previous_data_set:
                if rec.diabetic == child:
                    new_data_set.append(rec)

        return new_data_set
    # counting yes and no for parent
    def GetLensesStatusForParent(self, data_set):
        """
        --> How it works ?
        I choose the need lenses attribute to check thought program so
        counting yes and no for 5 Features according to the data set

        :param data_set: represent the data set to work on
        :return: number of yes [0] and no [1]
        """
        yes_lenses = 0
        no_lenses = 0
        for person in data_set:
            if person.needLense == 0:
                yes_lenses += 1
            else:
                no_lenses += 1
        return yes_lenses, no_lenses

    # counting yes and no for data in any feature
    def GetLensesStatusForChild(self, parent, child, data_set):
        """
        --> How it works ?
        I choose the need lenses attribute to check thought program so
        counting yes and no for the child for a specific feature
```

```
        :param parent: represent a feature as age, prescription ... ect
        :param child: represent the value in that parent whether 0 or 1
        :param data_set: represent the data set to work on
        :return: number of yes and no
        """
        yes_lenses = 0
        no_lenses = 0
        persons_data = data_set
        if parent == 'age':
            for person in persons_data:
                if person.age == child:
                    if person.needLense == 0:
                        yes_lenses += 1
                    else:
                        no_lenses += 1

        elif parent == 'prescription':
            for person in persons_data:
                if person.prescription == child:
                    if person.needLense == 0:
                        yes_lenses += 1
                    else:
                        no_lenses += 1

        elif parent == 'astigmatic':
            for person in persons_data:
                if person.astigmatic == child:
                    if person.needLense == 0:
                        yes_lenses += 1
                    else:
                        no_lenses += 1

        elif parent == 'tearRate':
            for person in persons_data:
                if person.tearRate == child:
                    if person.needLense == 0:
                        yes_lenses += 1
                    else:
                        no_lenses += 1

        elif parent == 'diabetic':
            for person in persons_data:
                if person.diabetic == child:
                    if person.needLense == 0:
                        yes_lenses += 1
                    else:
                        no_lenses += 1

        return yes_lenses, no_lenses

    # Calculate entropy for parent and child items
    def CalculteEntropy(self, yes_values, no_values):
        """
        --> How it works ?
        applying the entropy rule H(s) = - (p/total) log2(p/total) - (n/total) log2(n/total)
        this function throw exception in case log2(0) which is un defined so i put it between try and catch

        :param yes_values: represent p in previous Rule
        :param no_values: represent n in previous Rule
        :return: the entropy Value
        """
        entropy = 0
        try:
            summation = yes_values + no_values
            first_part = np.dot(-1 * (yes_values / summation), math.log2(yes_values / summation))
            second_part = np.dot(no_values / summation, math.log2(no_values / summation))
            entropy = first_part - second_part
        except:
            pass
        finally:
            return entropy
```

```
class Feature:
    # Additional Attributes
    decision_tree = {}
    features_gain_dic = {}

    # constructor
    def __init__(self, name=None):
        self.name = name
        self.visited = -1       # i don't understand the benefit from it so i don't use it [iam sorry]
        self.infoGain = -1

    # Calculate Gain For 5 Features [age, prescription, ... ect]
    def CalulateGain(self, feature):
        """
        --> How it Works?
        apply the rule Gain(s) = H(s) - I(child)

        :param feature: represent the feature name as age, prescription ..ect
        :return: Void just add this feature and its gain to dictionary
        """
        item_object = item()

        # calculate entropy for parent
        yes_counter, no_counter = item.GetLensesStatusForParent(item_object, item_object.getDataset())
        sum_parent = yes_counter + no_counter
        parent_entropy = item.CalculteEntropy(item_object, yes_counter, no_counter)

        # calculate entropy for parent child zero
        yes_counter, no_counter = item.GetLensesStatusForChild(item_object, feature, 0, item_object.getDataset())
        sum_child = yes_counter + no_counter
        child_zero_entropy = item.CalculteEntropy(item_object, yes_counter, no_counter)
        child_zero_entropy *= sum_child / sum_parent

        # calculate entropy for parent child one
        yes_counter, no_counter = item.GetLensesStatusForChild(item_object, feature, 1, item_object.getDataset())
        sum_child = yes_counter + no_counter
        child_one_entropy = item.CalculteEntropy(item_object, yes_counter, no_counter)
        child_one_entropy *= sum_child / sum_parent

        # Calculate gain
        self.infoGain = parent_entropy - (child_zero_entropy + child_one_entropy)

        # Add it To Dictionary
        self.name = feature
        self.features_gain_dic[self.name] = self.infoGain

    def ID3(self, features_gain, dataset):
        """
        --> How it Works?
        apply ID3 Algorithm and print Decision Tree Result

        :param features_gain: represent the value for feature []
        :param dataset: represent the data set to work on
        :return: void just Display the Decision Tree Result
        """
        item_object = item()

        # Stopping Condition
        if len(features_gain) == 0:
            return 0

        # Step 1 : Get Best Attribute For Splitting
        the_root = max(features_gain, key=features_gain.get)
        features_gain.pop(the_root)

        # Step 2 : Creating the Child for this Best Attribute Child always [0, 1] as Defined
        # Step 3 : Check if this Child is pure or not
        status = self.IsChildPure(the_root, 0, dataset)
        if status == 'not pure':
            print("\t   the root {} in child 0 status {}".format(the_root, status))
            self.decision_tree[the_root] = 0
```

```
            # Step 4 : split the data set to check for pureness
            new_data_set = item.SplitDataSet(item_object, dataset, the_root, 0)

            # Step 5 : Recurrence if not pure and Stop if it pure
            self.ID3(features_gain, new_data_set)
        else:
            print("\t - the root {} in child 0 status {}".format(the_root, status))

        # repeat step 4 and 5 for another child
        status = self.IsChildPure(the_root, 1, dataset)
        if status == 'not pure':
            print("\t   the root {} in child 1 status {}".format(the_root, status))
            self.decision_tree[the_root] = 1
            new_data_set = item.SplitDataSet(item_object, dataset, the_root, 1)
            self.ID3(features_gain, new_data_set)
        else:
            print("\t   the root {} in child 1 status {}".format(the_root, status))


    # Is Child pure Implementation
    def IsChildPure(self, parent, child, data_set):
        """
        --> How it Works?
        Get number of yes and no for child according to its parent and specific
        data set to check for pureness

        :param parent: represent a feature as age, prescription ... ect
        :param child: represent the 0 or 1 to check for it
        :param data_set: represent the data set to check pure and work on
        :return: status for this data set
        """
        obj = item()
        status = 'not pure'
        yes_counter, no_counter = item.GetLensesStatusForChild(obj, parent, child, data_set)

        if yes_counter == 0:
            status = 'its pure all no'
            return status
        elif no_counter == 0:
            status = 'its pure all yes'
            return status
        return status


class ID3:
    def __init__(self, features):
        self.features = features

        # pass features [age, prescription ..] to class
        feature_object = Feature()
        for feature in self.features:
            feature_object.CalulateGain(feature.name)

        # The Explanation of decision tree
        item_object = item()
        print("the Decision Tree pure and not pure test:")
        feature_object.ID3(feature_object.features_gain_dic, item_object.getDataset())
        print("\nThis Dictionary included the root [represented in key] and its not pure child [represented in value].")
        print(feature_object.decision_tree)
        print("HINT:")
        print("- If child equal 1 that means the other child that equal 0 is pure")
        print("- If child equal 0 that means the other child that equal 1 is pure")
        print("- If child equal [1,0] that means The both are not pure.")
        print("- If Root Doesn't Exist that means the Both childs are Pure as astigmatic in our case")
        print()
```

```
    def classify(self, input):
        """ after applying ID3 Algorithm we reached to this Tree
            But i fail to draw it so i make this pattern
            to let you sir understand the Design of tree
            ------------------------------------------------------------------------------------
            | Decision Tree :                                                                  |
            | ----------------                                                                 |
            | tearRate if 0 so yes                                                             |
            |          if 1 call prescription if 0 so No                                       |
            |                                  if 1 Call age if 0 so No                        |
            |                                                 if 1 Call astigmatic if 0 so No  |
            |                                                                   if 1 So Yes    |
            ------------------------------------------------------------------------------------
            So after knowing the efficient how way to classify we could
            now easily check for input according to this tree
        """

        # get list features and here we don't need diabetic Test according to Decision tree
        age = input[0]
        prescription = input[1]
        astigmatic = input[2]
        tearRate = input[3]

        # here is the Classification of Decision Tree
        if tearRate == 0:
            return 0
        else:
            if prescription == 0:
                return 1
            else:
                if age == 0:
                    return 1
                else:
                    if astigmatic == 0:
                        return 1
                    else:
                        return 0
# endregion
```

3.3. Sample Run.

```
the Decision Tree pure and not pure test:
     - the root tearRate in child 0 status its pure all yes
       the root tearRate in child 1 status not pure
     - the root prescription in child 0 status its pure all no
       the root prescription in child 1 status not pure
     - the root age in child 0 status its pure all no
       the root age in child 1 status not pure
     - the root astigmatic in child 0 status its pure all no
       the root astigmatic in child 1 status its pure all yes

This Dictionary included the root [represented in key] and its not pure child [represented in value].
{'tearRate': 1, 'prescription': 1, 'age': 1}
HINT:
- If child equal 1 that means the other child that equal 0 is pure
- If child equal 0 that means the other child that equal 1 is pure
- If child equal [1,0] that means The both are not pure.
- If Root Doesn't Exist that means the Both childs are Pure as astigmatic in our case

testcase 1:  1
testcase 2:  0
testcase 3:  0
testcase 4:  1
```

- **Additional Attributes and Functions**
  - In Class Item :

```
def SplitDataSet(self, previous_data_set, parent, child)
def CalculteEntropy(self, yes_values, no_values)
def GetLensesStatusForParent(self, data_set)
def GetLensesStatusForChild(self, parent, child, data_set)
```

  - In Class Feature :

```
decision_tree = {}
features_gain_dic = {}

def CalulateGain(self, feature)
def ID3(self, features_gain, dataset)
def IsChildPure(self, parent,child, data_set)
```

3. Discussion

My code is not so efficient and that because sometimes I implement a function from scratch while I could include it from its library or exist other functions do same with less complexity and I didn't use it, I am not good at libraries but I'll try to improve that skill in Shaa Allah.

4. References

- Dr Marco Alphons, "Lecture 4 – Machine Learning Supervised Learning ", Computer Science Ain Shams University.
- Dr Marco Alphons, "Lecture 11 – Introduction To ANN ", Computer Science Ain Shams University, in Sponsored organization Location.
- Dr Zena, "Lab 3 – uninformed and informed search ", Computer Science Ain Shams University, in Sponsored organization Location.
- TA Staff Member, "Lab 11 – ANN ", Computer Science Ain Shams University, in Sponsored organization Location.
- Dr Zena, "Lab 6 – ID3 ", Computer Science Ain Shams University, in Sponsored organization Location.

Thank You