Faculty of Computer & Information Sciences
Ain Shams University
Subject: CSC 343 Artificial Intelligence
Year: (3rd year)undergraduate
Academic year: 2nd term 2019-2020

Research Topic (1)

Title: Classification, Neural Computing and Search

1. Introduction

1-Breadth-first search algorithm:

It is an algorithm for searching or traverse a tree or a graph, It starts from the first node the root and discover its children in the next level of the tree and after discovering the whole children and finish this level it discovers the next level and so on until discovering the whole tree or graph.

It's applications:

- ❖ Traverse a tree or graph.
- ❖ Search for an element in graph or tree.
- ❖ Find the shortest path or the Minimum spanning tree.
- ❖ GPS navigation system.

2- Neural network algorithm:

It is algorithm that is designed to make the machine learn and visualize like the human’s brain, It starts by making the machine train on some data with inputs and outputs and then make it think in another data inputs and predict the output by thinking in what it learns.

It’s applications:

- ❖ Data visualization and analysis.
- ❖ Data Compression.
- ❖ Face or voice Recognition and verification.

3- ID3 algorithm:

It is algorithm that is used in classification and prediction by making a decision tree by training on a dataset by portioning inputs into set of regions and use a simple predictor for each region.

It’s applications:

- ❖ Prediction of unknown result.
- ❖ Classification of large data.
- ❖ Could use in a lot of fields by training in large data to get a specific decision.

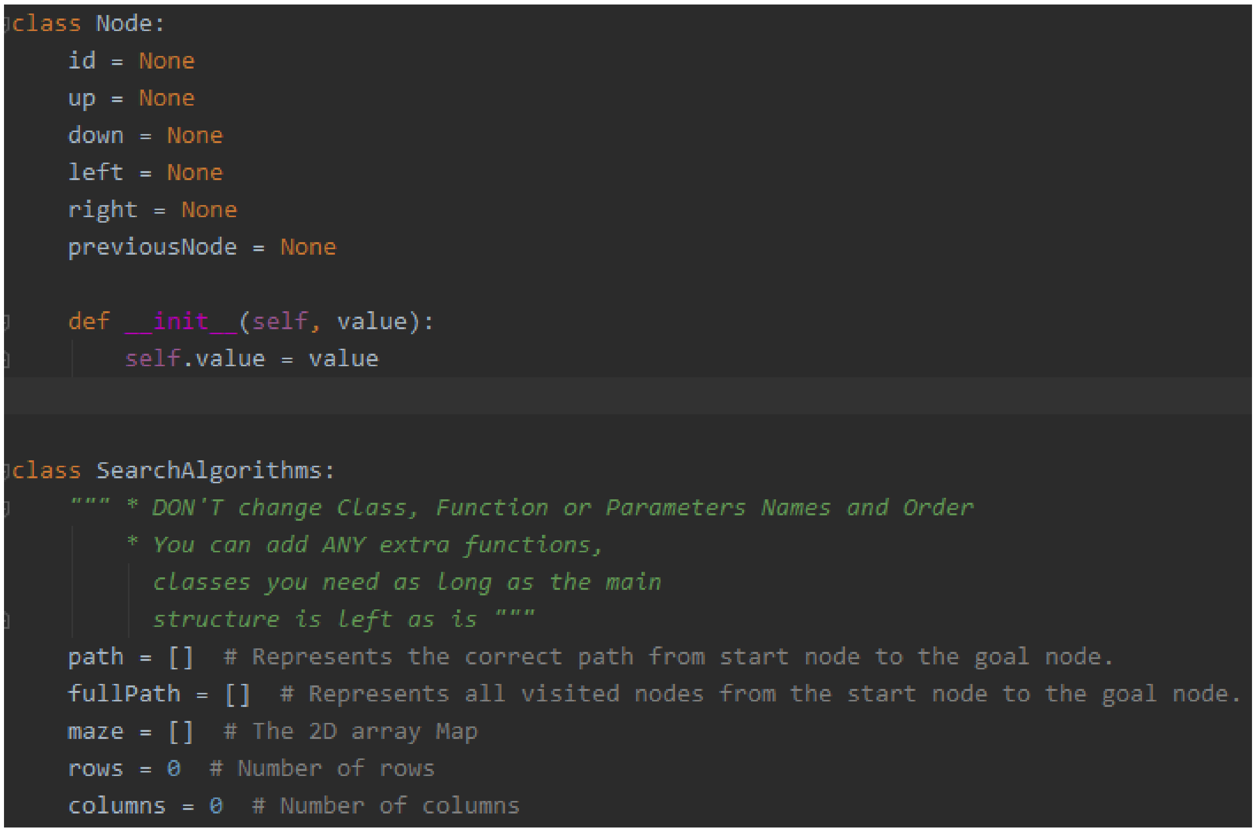
2. The algorithms

2.1. Breadth-first search

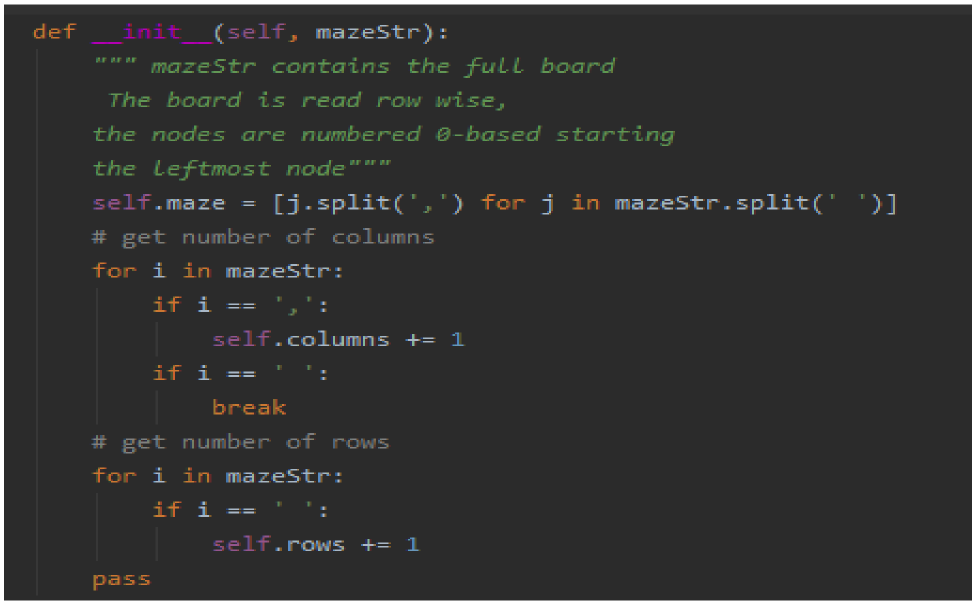
2.1.1. The main steps of the algorithm:

- Step 1: initialize a queue and enqueue the starting node.
- Step 2: initialize an empty list.
- Step 3: check if the queue is empty exit.
- Step 4: dequeue Node form the queue and append it on the list.
- Step 5: check if the Node is your final state exit.
- Step 6: enqueue all the Node’s children in the queue.
- Step 7: repeat from step 3.

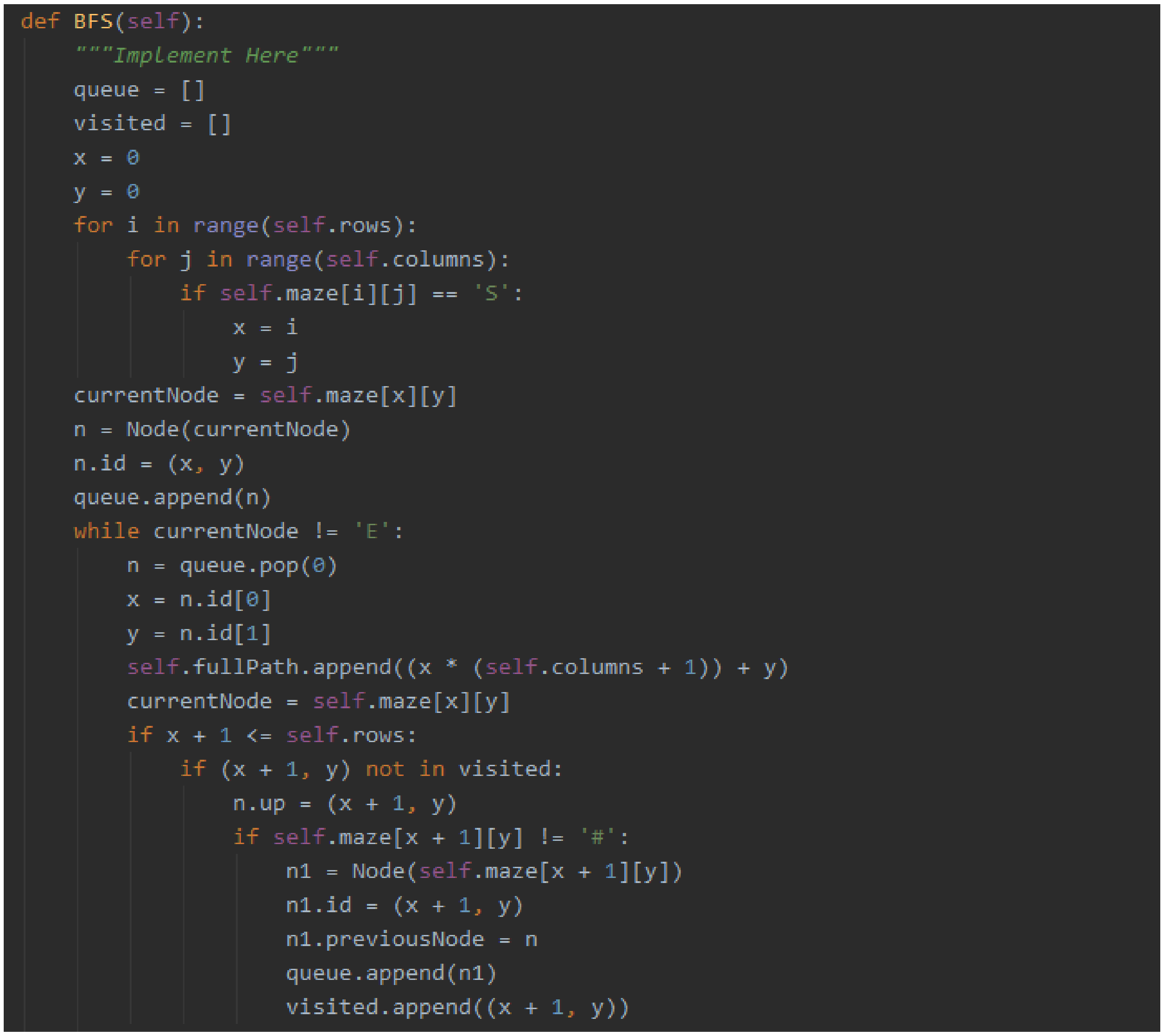
2.1.2. The implementation of the algorithm:



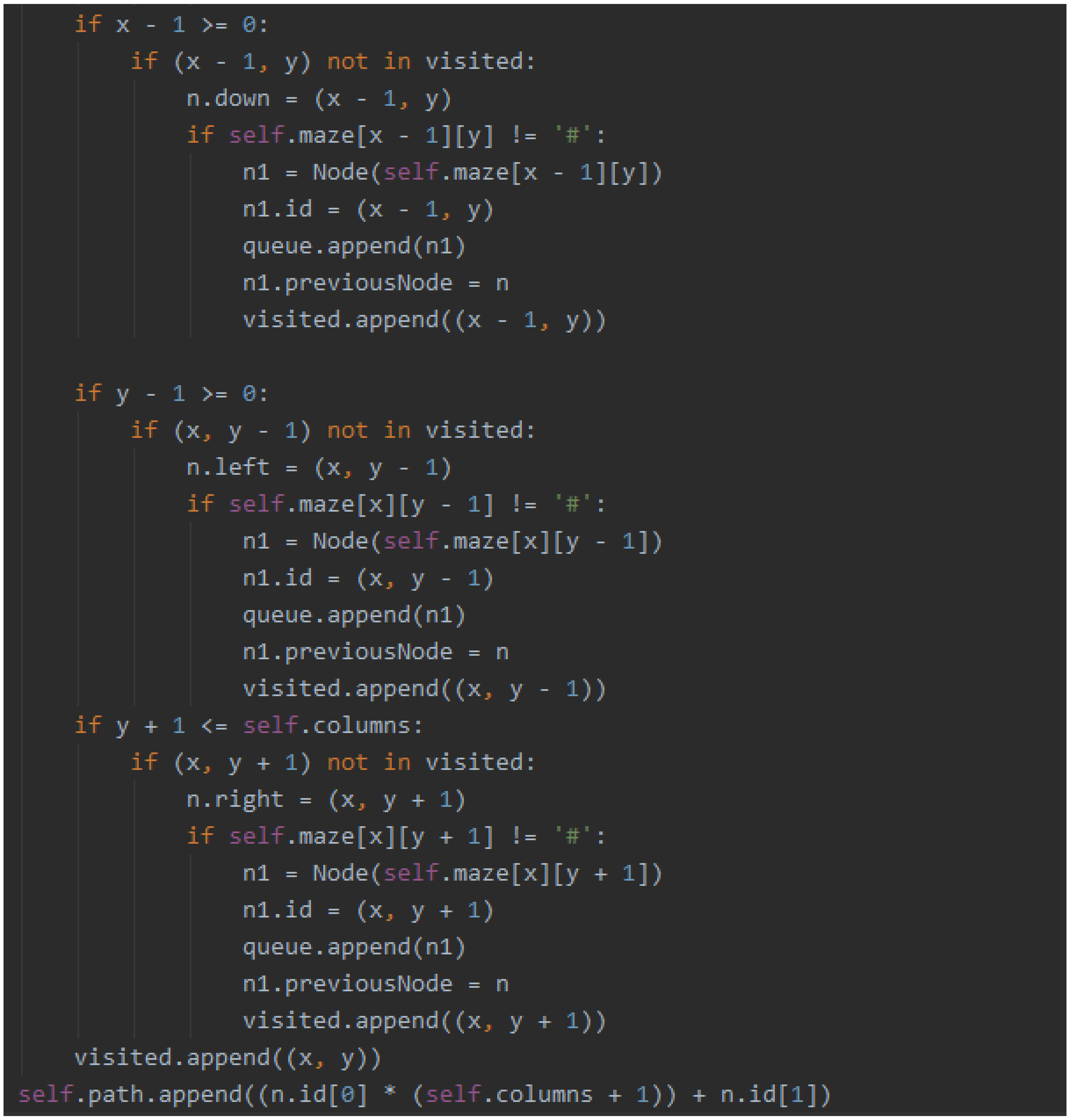
```python
class Node:
    id = None
    up = None
    down = None
    left = None
    right = None
    previousNode = None

    def __init__(self, value):
        self.value = value


class SearchAlgorithms:
    """ * DON'T change Class, Function or Parameters Names and Order
        * You can add ANY extra functions,
          classes you need as long as the main
          structure is left as is """
    path = []  # Represents the correct path from start node to the goal node.
    fullPath = []  # Represents all visited nodes from the start node to the goal node.
    maze = []  # The 2D array Map
    rows = 0  # Number of rows
    columns = 0  # Number of columns
```



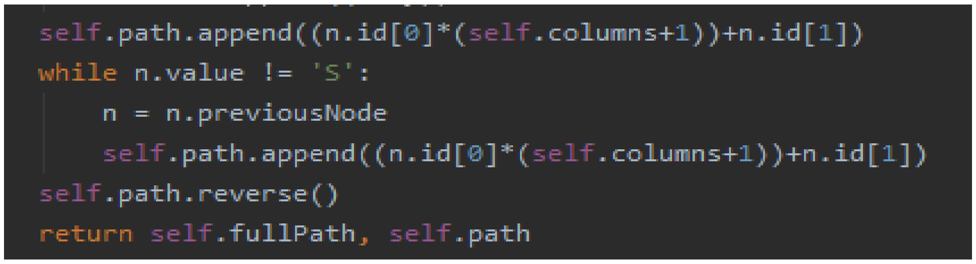
```
def __init__(self, mazeStr):
    """ mazeStr contains the full board
     The board is read row wise,
    the nodes are numbered 0-based starting
    the leftmost node"""
    self.maze = [j.split(',') for j in mazeStr.split(' ')]
    # get number of columns
    for i in mazeStr:
        if i == ',':
            self.columns += 1
        if i == ' ':
            break
    # get number of rows
    for i in mazeStr:
        if i == ' ':
            self.rows += 1
    pass
```



```
def BFS(self):
    """Implement Here"""
    queue = []
    visited = []
    x = 0
    y = 0
    for i in range(self.rows):
        for j in range(self.columns):
            if self.maze[i][j] == 'S':
                x = i
                y = j
    currentNode = self.maze[x][y]
    n = Node(currentNode)
    n.id = (x, y)
    queue.append(n)
    while currentNode != 'E':
        n = queue.pop(0)
        x = n.id[0]
        y = n.id[1]
        self.fullPath.append((x * (self.columns + 1)) + y)
        currentNode = self.maze[x][y]
        if x + 1 <= self.rows:
            if (x + 1, y) not in visited:
                n.up = (x + 1, y)
                if self.maze[x + 1][y] != '#':
                    n1 = Node(self.maze[x + 1][y])
                    n1.id = (x + 1, y)
                    n1.previousNode = n
                    queue.append(n1)
                    visited.append((x + 1, y))
```

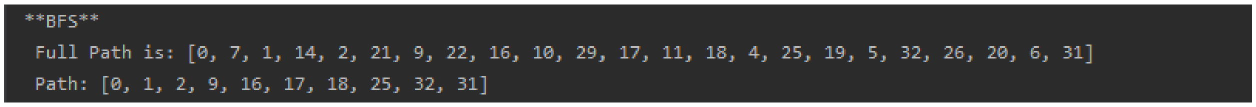


```
    if x - 1 >= 0:
        if (x - 1, y) not in visited:
            n.down = (x - 1, y)
            if self.maze[x - 1][y] != '#':
                n1 = Node(self.maze[x - 1][y])
                n1.id = (x - 1, y)
                queue.append(n1)
                n1.previousNode = n
                visited.append((x - 1, y))

    if y - 1 >= 0:
        if (x, y - 1) not in visited:
            n.left = (x, y - 1)
            if self.maze[x][y - 1] != '#':
                n1 = Node(self.maze[x][y - 1])
                n1.id = (x, y - 1)
                queue.append(n1)
                n1.previousNode = n
                visited.append((x, y - 1))
    if y + 1 <= self.columns:
        if (x, y + 1) not in visited:
            n.right = (x, y + 1)
            if self.maze[x][y + 1] != '#':
                n1 = Node(self.maze[x][y + 1])
                n1.id = (x, y + 1)
                queue.append(n1)
                n1.previousNode = n
                visited.append((x, y + 1))
    visited.append((x, y))
self.path.append((n.id[0] * (self.columns + 1)) + n.id[1])
```



```
self.path.append((n.id[0]*(self.columns+1))+n.id[1])
while n.value != 'S':
    n = n.previousNode
    self.path.append((n.id[0]*(self.columns+1))+n.id[1])
self.path.reverse()
return self.fullPath, self.path
```

2.1.3. Sample run (the output):



```
**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]
```

2.2. Neural network

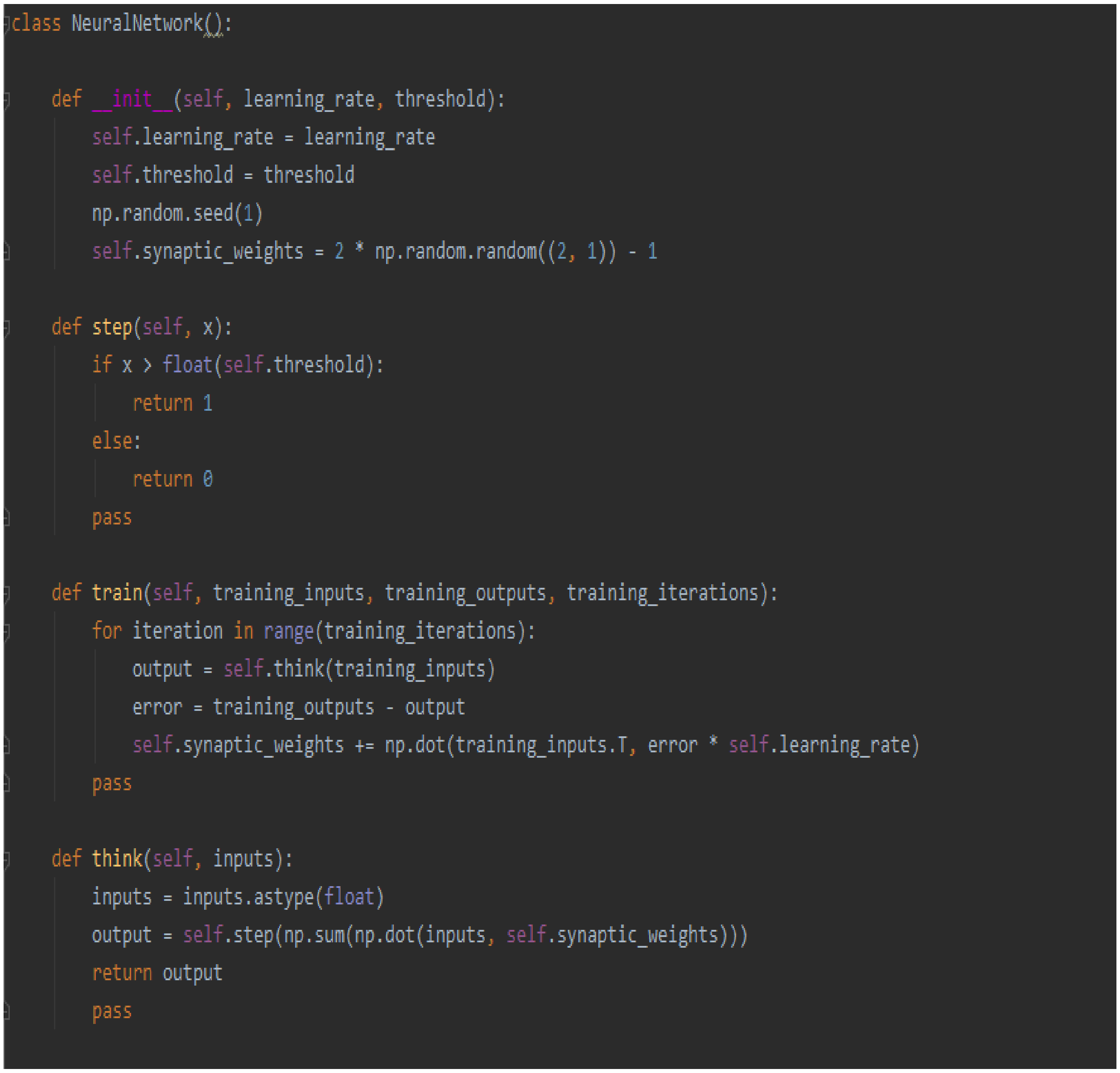
2.2.1. The main steps of the algorithm

- Step 1: Initialize random weights.
- Step 2: Determine pattern and target output.
- Step 3: Calculate expected output by applying the step of the activation function: expected output = Step $(\sum_{i=1}^{n} wi\ xi\ )$
- Step 4: Calculate error and update weights:

  Error = Expected output – Real output

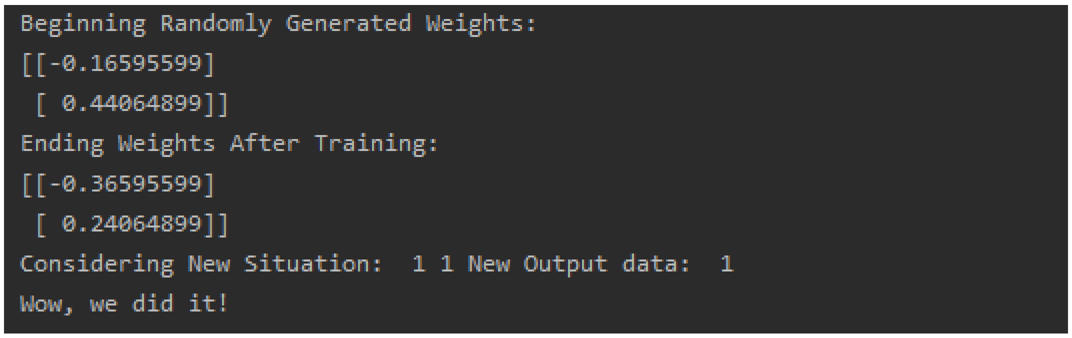
  New weight i = Old weight i + (Error * Learning rate * Xi)
- Step 5: Repeat from Step 2 until accepted error.

2.2.2. The implementation of the algorithm:



```python
class NeuralNetwork():

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

    def step(self, x):
        if x > float(self.threshold):
            return 1
        else:
            return 0
        pass

    def train(self, training_inputs, training_outputs, training_iterations):
        for iteration in range(training_iterations):
            output = self.think(training_inputs)
            error = training_outputs - output
            self.synaptic_weights += np.dot(training_inputs.T, error * self.learning_rate)
        pass

    def think(self, inputs):
        inputs = inputs.astype(float)
        output = self.step(np.sum(np.dot(inputs, self.synaptic_weights)))
        return output
        pass
```

2.2.3. Sample run (the output)



```
Beginning Randomly Generated Weights:
[[-0.16595599]
 [ 0.44064899]]
Ending Weights After Training:
[[-0.36595599]
 [ 0.24064899]]
Considering New Situation:  1 1 New Output data:  1
Wow, we did it!
```

2.3. ID3:

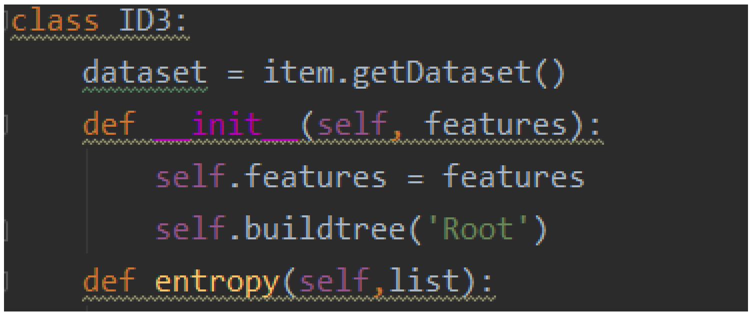
2.3.1. The main steps of the algorithm:

- Step 1: Calculate the gain of each feature on my data.

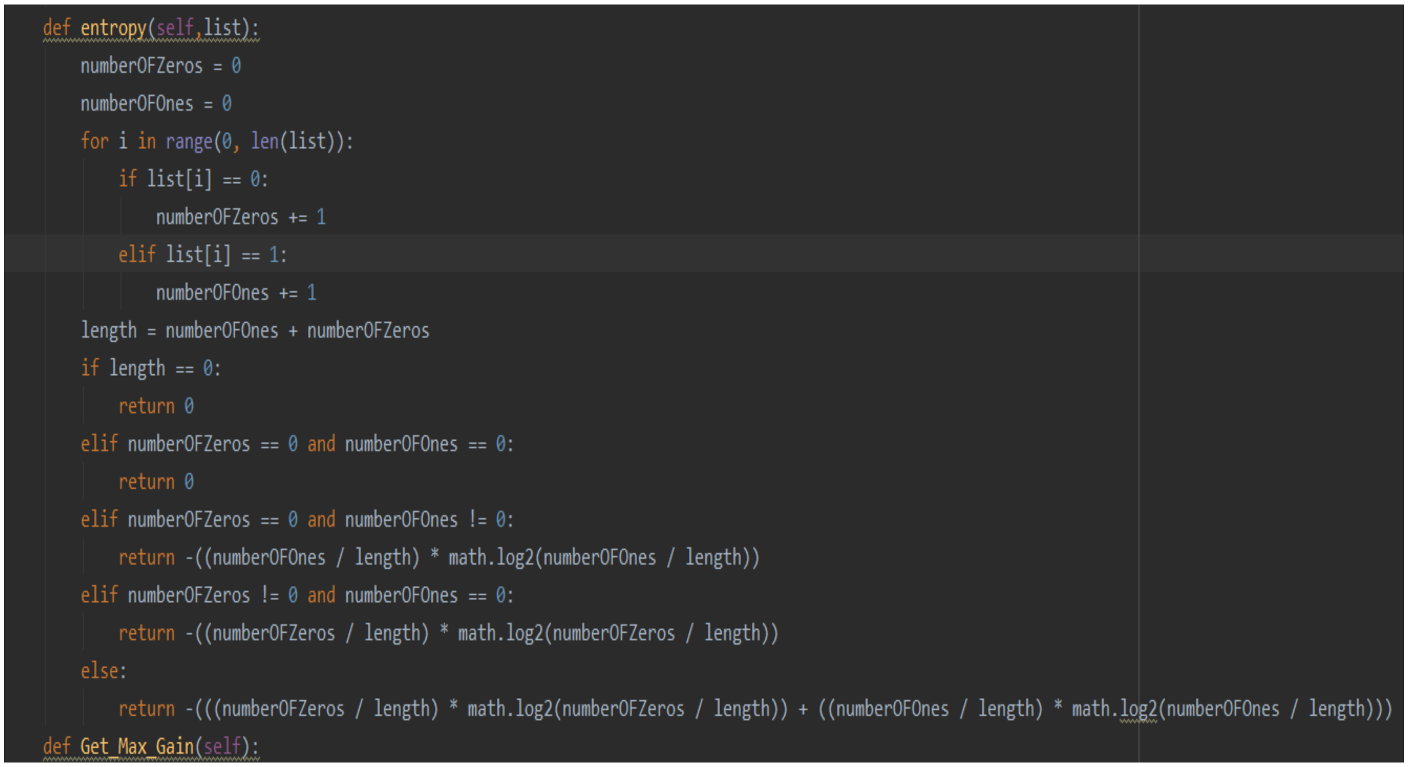
  Gain = Entropy - $\sum_{i=0}^{All\ children\ of\ attribute} \frac{Pi+Ni}{Total}(child)$

  Entropy = - $\frac{P}{Total}\log_2\frac{P}{Total}$ - - $\frac{N}{Total}\log_2\frac{N}{Total}$
- Step 2: The feature with the maximum gain will be the root node for the best splitting.
- Step 3: For each value of the attributes of this node create a new child node.
- Step 4: Split training to child nodes.
- Step 5: If pure classified Stop else iterate over the new child nodes and split again.

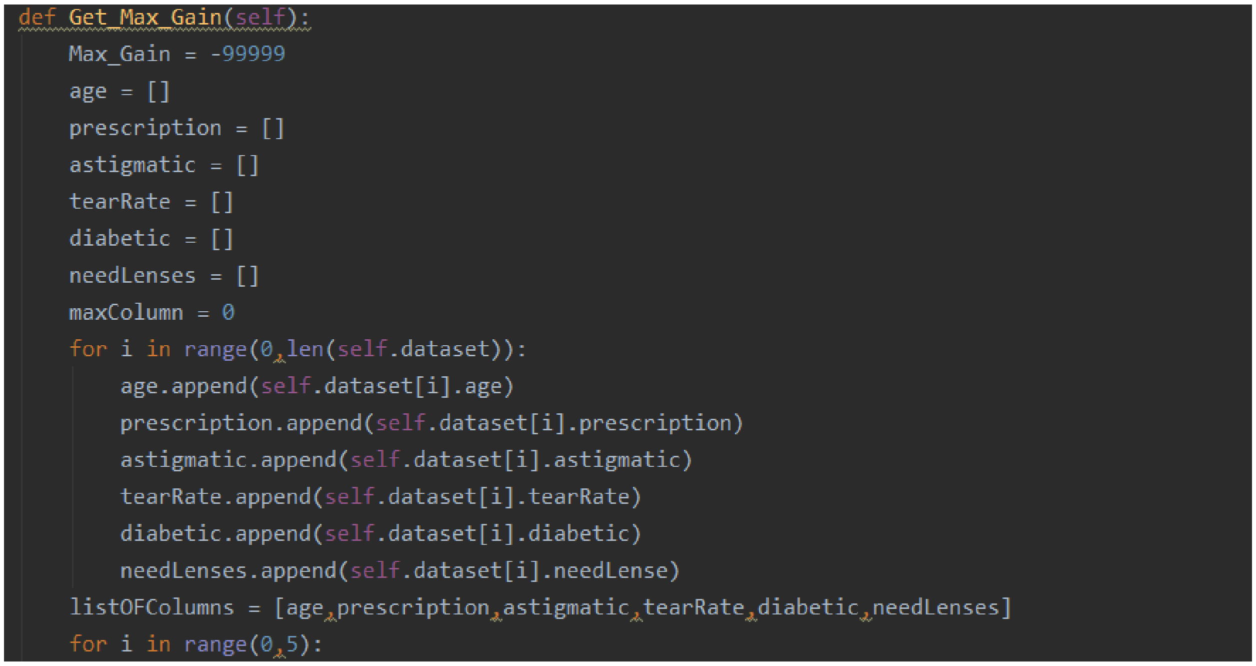
2.3.2. The implementation of the algorithm:



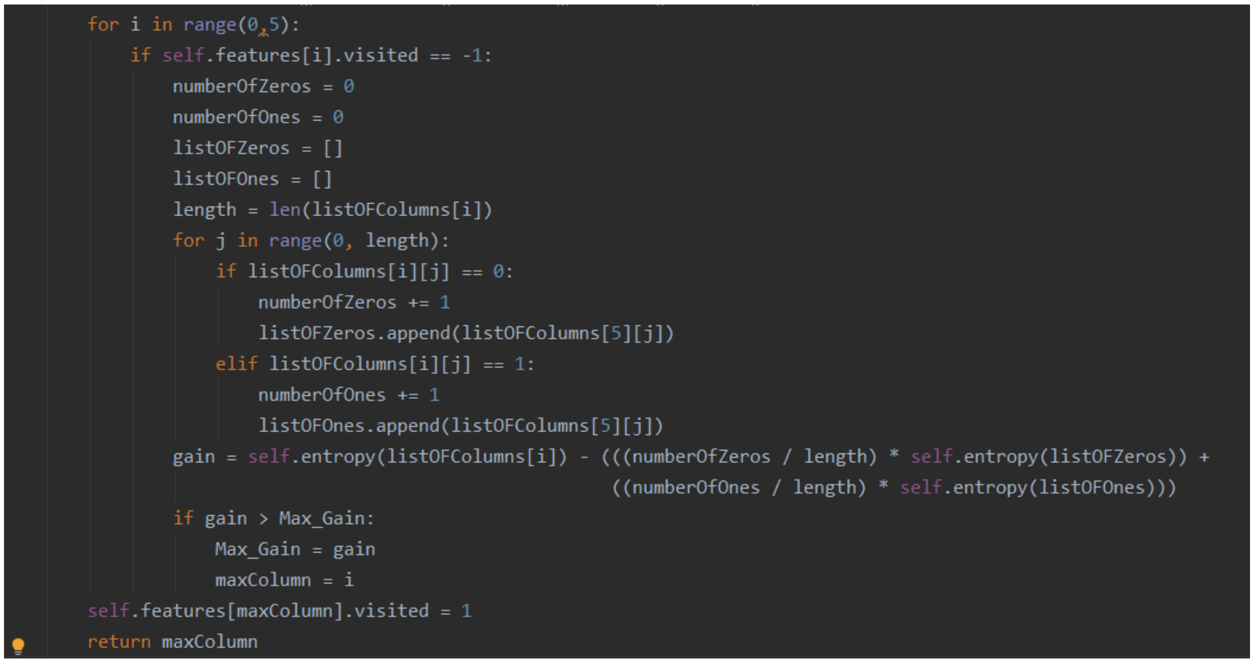
```
class ID3:
    dataset = item.getDataset()
    def __init__(self, features):
        self.features = features
        self.buildtree('Root')
    def entropy(self,list):
```



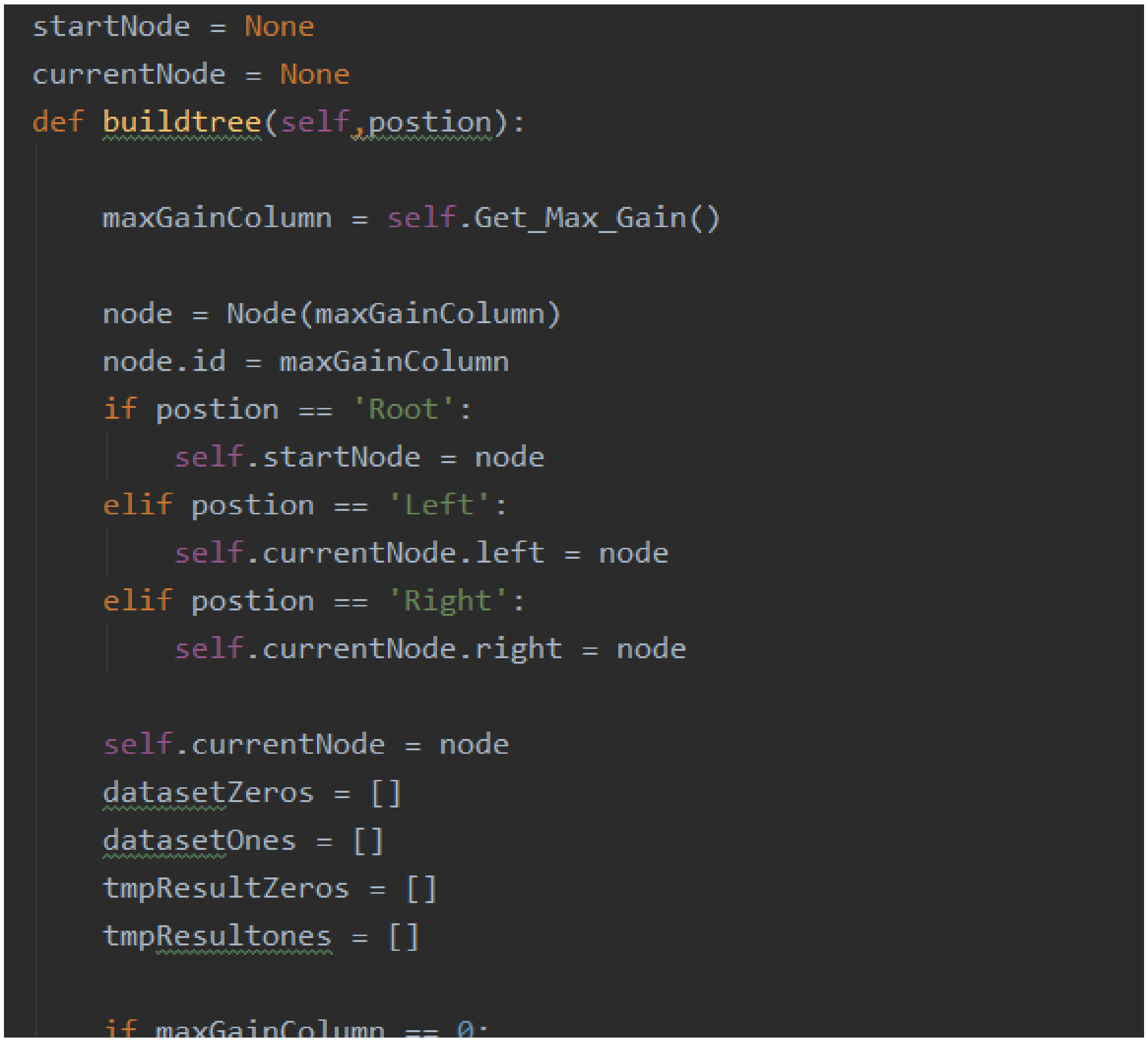
```
def entropy(self,list):
    numberOFZeros = 0
    numberOFOnes = 0
    for i in range(0, len(list)):
        if list[i] == 0:
            numberOFZeros += 1
        elif list[i] == 1:
            numberOFOnes += 1
    length = numberOFOnes + numberOFZeros
    if length == 0:
        return 0
    elif numberOFZeros == 0 and numberOFOnes == 0:
        return 0
    elif numberOFZeros == 0 and numberOFOnes != 0:
        return -((numberOFOnes / length) * math.log2(numberOFOnes / length))
    elif numberOFZeros != 0 and numberOFOnes == 0:
        return -((numberOFZeros / length) * math.log2(numberOFZeros / length))
    else:
        return -(((numberOFZeros / length) * math.log2(numberOFZeros / length)) + ((numberOFOnes / length) * math.log2(numberOFOnes / length)))
def Get_Max_Gain(self):
```



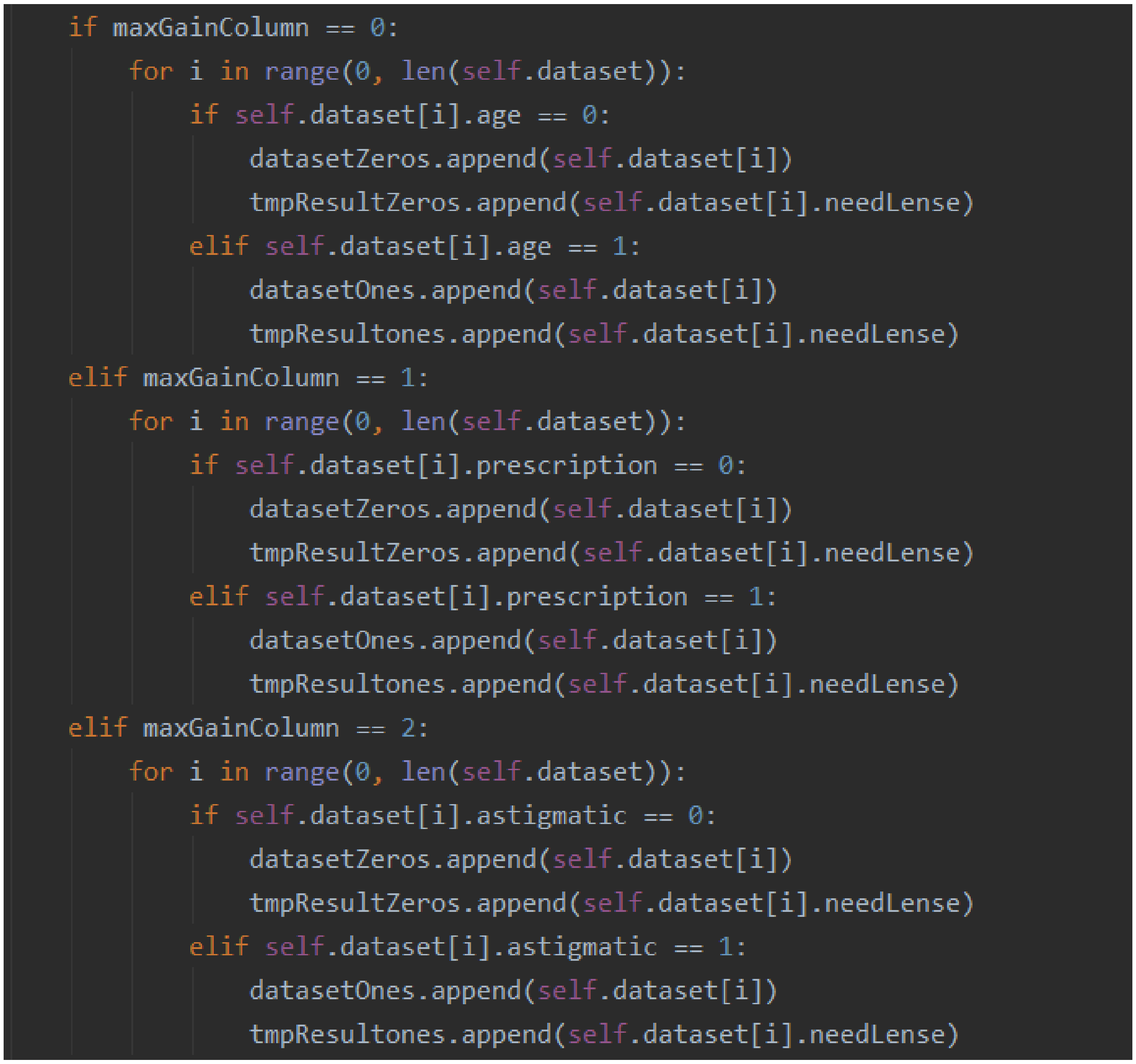
```
def Get_Max_Gain(self):
    Max_Gain = -99999
    age = []
    prescription = []
    astigmatic = []
    tearRate = []
    diabetic = []
    needLenses = []
    maxColumn = 0
    for i in range(0,len(self.dataset)):
        age.append(self.dataset[i].age)
        prescription.append(self.dataset[i].prescription)
        astigmatic.append(self.dataset[i].astigmatic)
        tearRate.append(self.dataset[i].tearRate)
        diabetic.append(self.dataset[i].diabetic)
        needLenses.append(self.dataset[i].needLense)
    listOFColumns = [age,prescription,astigmatic,tearRate,diabetic,needLenses]
    for i in range(0,5):
```



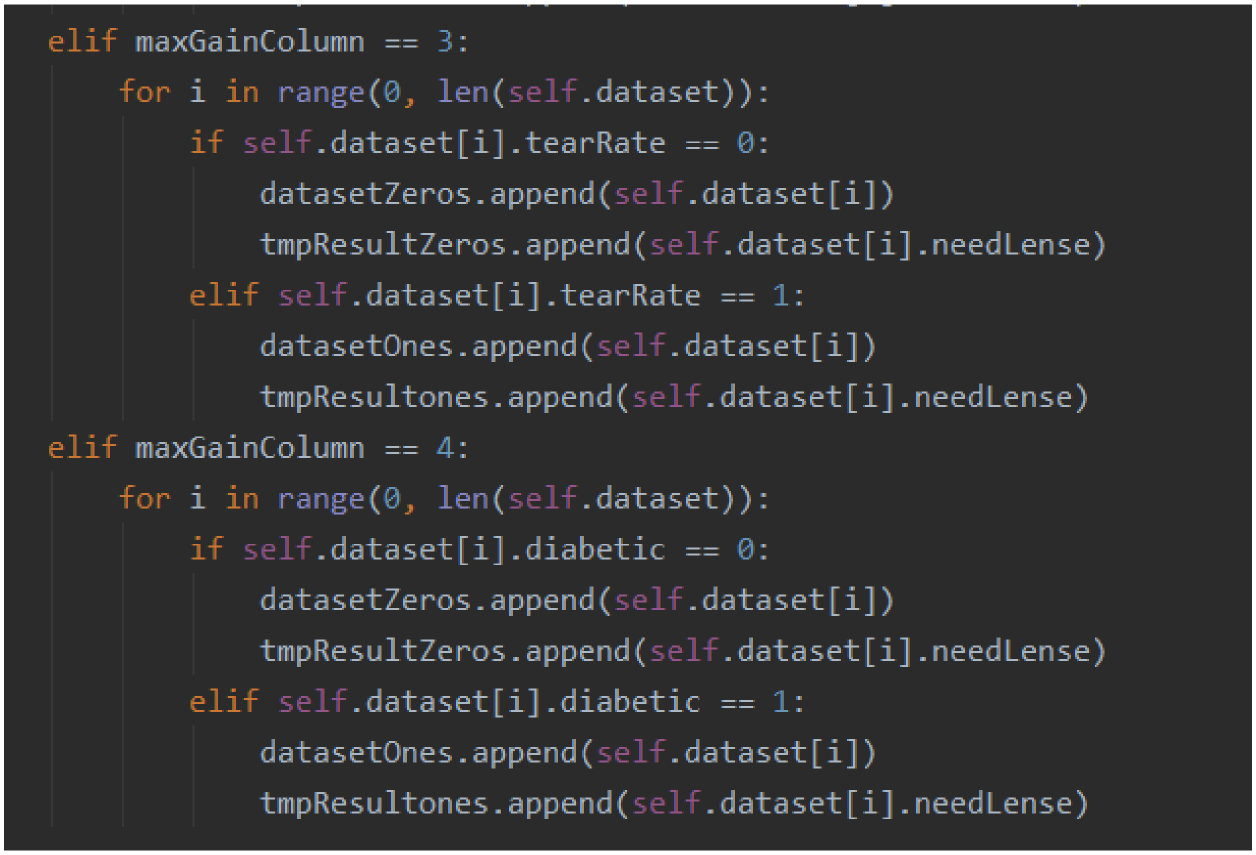
```
    for i in range(0,5):
        if self.features[i].visited == -1:
            numberOfZeros = 0
            numberOfOnes = 0
            listOFZeros = []
            listOFOnes = []
            length = len(listOFColumns[i])
            for j in range(0, length):
                if listOFColumns[i][j] == 0:
                    numberOfZeros += 1
                    listOFZeros.append(listOFColumns[5][j])
                elif listOFColumns[i][j] == 1:
                    numberOfOnes += 1
                    listOFOnes.append(listOFColumns[5][j])
            gain = self.entropy(listOFColumns[i]) - (((numberOfZeros / length) * self.entropy(listOFZeros)) +
                                                     ((numberOfOnes / length) * self.entropy(listOFOnes)))
            if gain > Max_Gain:
                Max_Gain = gain
                maxColumn = i
    self.features[maxColumn].visited = 1
    return maxColumn
```



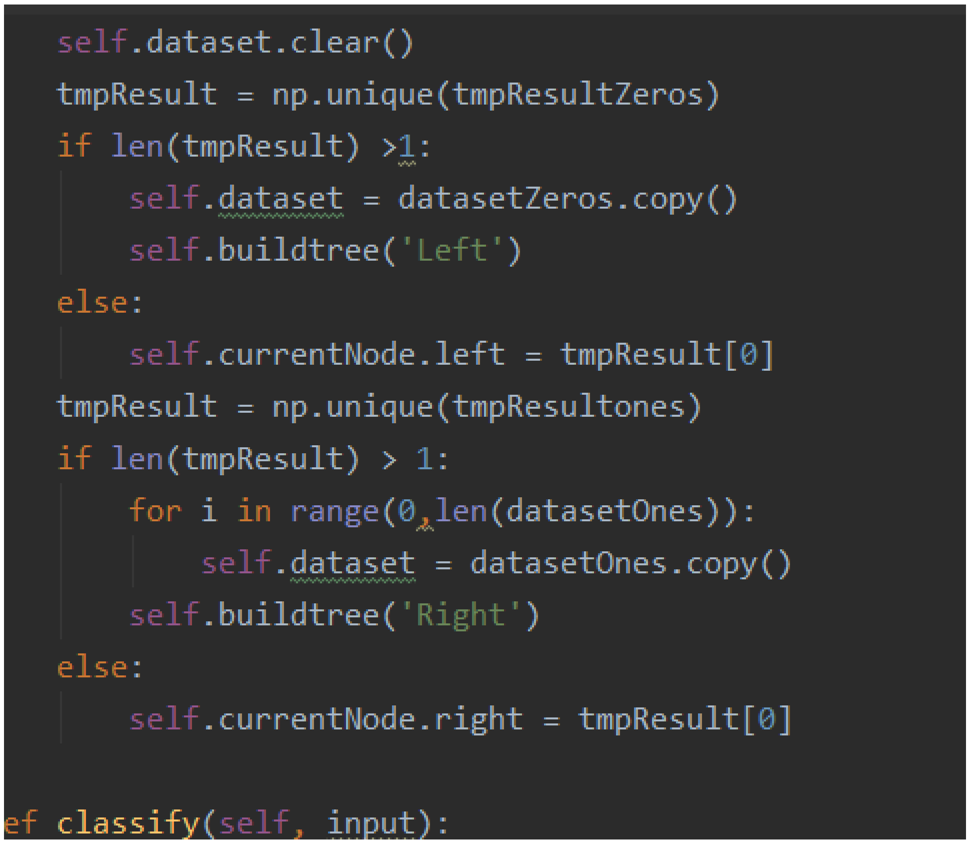
```
startNode = None
currentNode = None
def buildtree(self,postion):

    maxGainColumn = self.Get_Max_Gain()

    node = Node(maxGainColumn)
    node.id = maxGainColumn
    if postion == 'Root':
        self.startNode = node
    elif postion == 'Left':
        self.currentNode.left = node
    elif postion == 'Right':
        self.currentNode.right = node

    self.currentNode = node
    datasetZeros = []
    datasetOnes = []
    tmpResultZeros = []
    tmpResultones = []

    if maxGainColumn == 0:
```



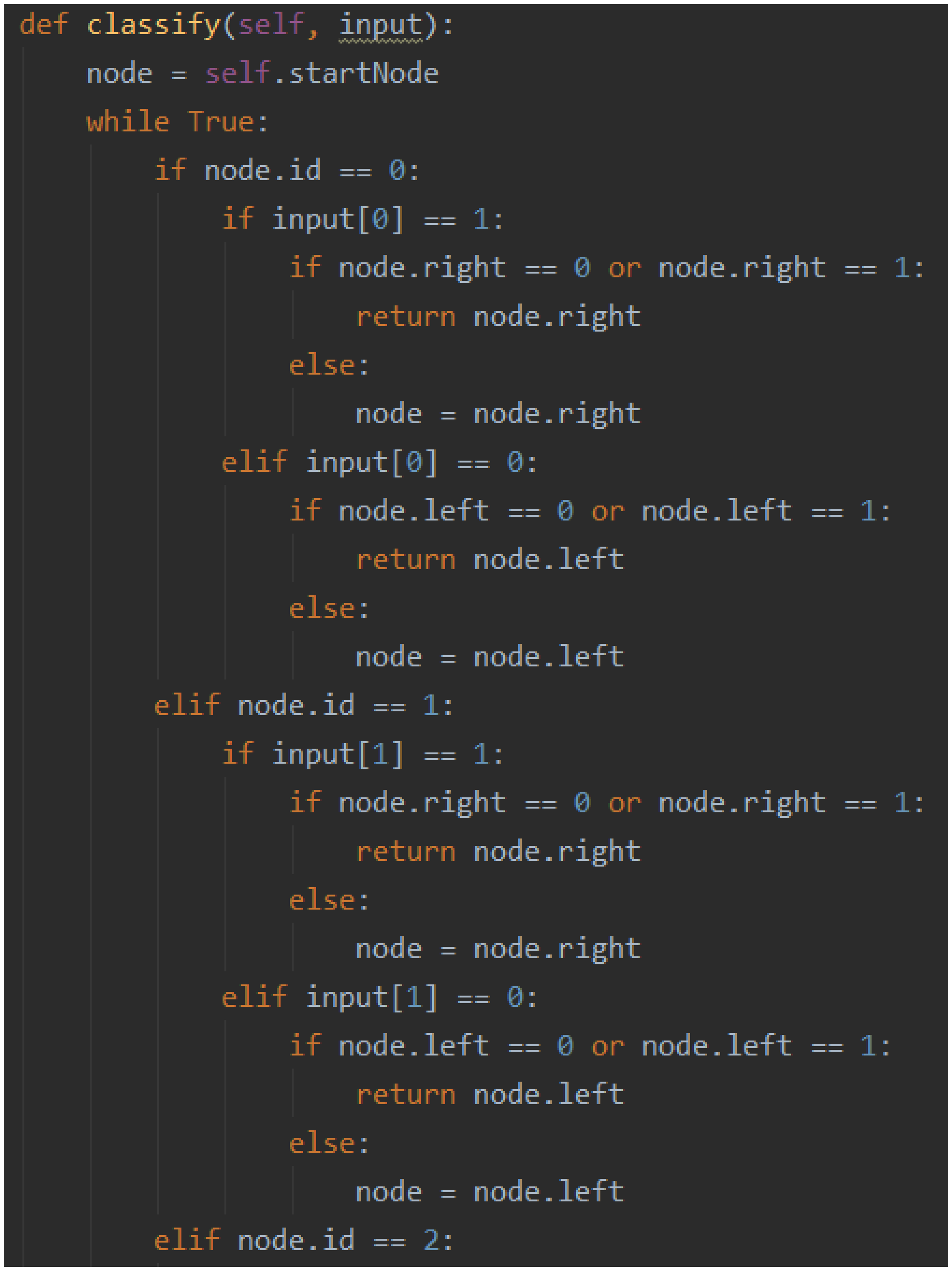
```
    if maxGainColumn == 0:
        for i in range(0, len(self.dataset)):
            if self.dataset[i].age == 0:
                datasetZeros.append(self.dataset[i])
                tmpResultZeros.append(self.dataset[i].needLense)
            elif self.dataset[i].age == 1:
                datasetOnes.append(self.dataset[i])
                tmpResultones.append(self.dataset[i].needLense)
    elif maxGainColumn == 1:
        for i in range(0, len(self.dataset)):
            if self.dataset[i].prescription == 0:
                datasetZeros.append(self.dataset[i])
                tmpResultZeros.append(self.dataset[i].needLense)
            elif self.dataset[i].prescription == 1:
                datasetOnes.append(self.dataset[i])
                tmpResultones.append(self.dataset[i].needLense)
    elif maxGainColumn == 2:
        for i in range(0, len(self.dataset)):
            if self.dataset[i].astigmatic == 0:
                datasetZeros.append(self.dataset[i])
                tmpResultZeros.append(self.dataset[i].needLense)
            elif self.dataset[i].astigmatic == 1:
                datasetOnes.append(self.dataset[i])
                tmpResultones.append(self.dataset[i].needLense)
```

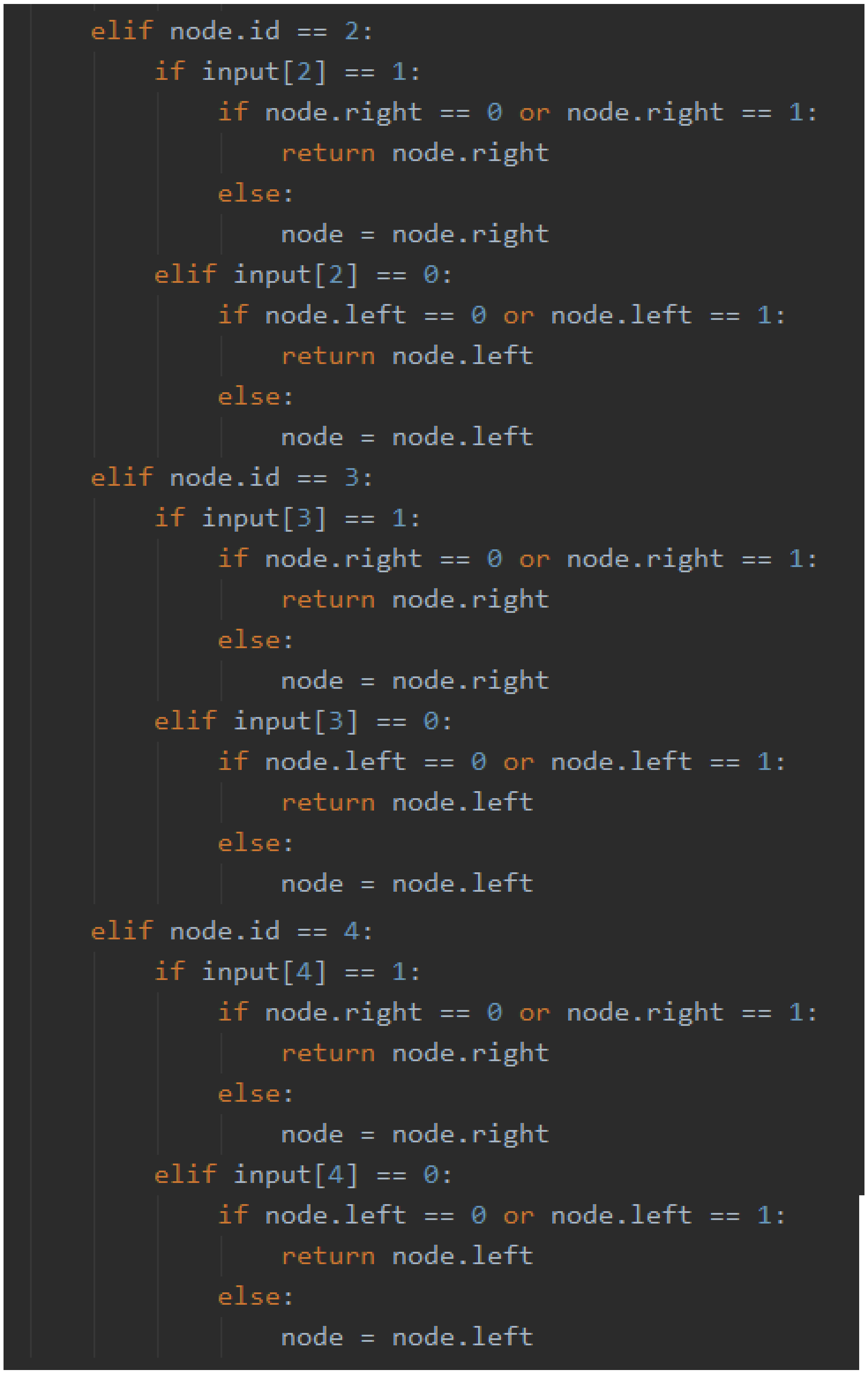


```
elif maxGainColumn == 3:
    for i in range(0, len(self.dataset)):
        if self.dataset[i].tearRate == 0:
            datasetZeros.append(self.dataset[i])
            tmpResultZeros.append(self.dataset[i].needLense)
        elif self.dataset[i].tearRate == 1:
            datasetOnes.append(self.dataset[i])
            tmpResultones.append(self.dataset[i].needLense)
elif maxGainColumn == 4:
    for i in range(0, len(self.dataset)):
        if self.dataset[i].diabetic == 0:
            datasetZeros.append(self.dataset[i])
            tmpResultZeros.append(self.dataset[i].needLense)
        elif self.dataset[i].diabetic == 1:
            datasetOnes.append(self.dataset[i])
            tmpResultones.append(self.dataset[i].needLense)
```



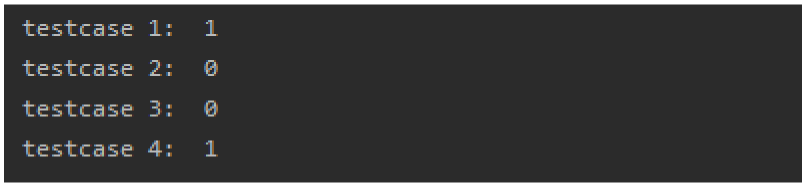
```
    self.dataset.clear()
    tmpResult = np.unique(tmpResultZeros)
    if len(tmpResult) >1:
        self.dataset = datasetZeros.copy()
        self.buildtree('Left')
    else:
        self.currentNode.left = tmpResult[0]
    tmpResult = np.unique(tmpResultones)
    if len(tmpResult) > 1:
        for i in range(0,len(datasetOnes)):
            self.dataset = datasetOnes.copy()
        self.buildtree('Right')
    else:
        self.currentNode.right = tmpResult[0]

ef classify(self, input):
```


```python
def classify(self, input):
    node = self.startNode
    while True:
        if node.id == 0:
            if input[0] == 1:
                if node.right == 0 or node.right == 1:
                    return node.right
                else:
                    node = node.right
            elif input[0] == 0:
                if node.left == 0 or node.left == 1:
                    return node.left
                else:
                    node = node.left
        elif node.id == 1:
            if input[1] == 1:
                if node.right == 0 or node.right == 1:
                    return node.right
                else:
                    node = node.right
            elif input[1] == 0:
                if node.left == 0 or node.left == 1:
                    return node.left
                else:
                    node = node.left
        elif node.id == 2:
```



```
    elif node.id == 2:
        if input[2] == 1:
            if node.right == 0 or node.right == 1:
                return node.right
            else:
                node = node.right
        elif input[2] == 0:
            if node.left == 0 or node.left == 1:
                return node.left
            else:
                node = node.left
    elif node.id == 3:
        if input[3] == 1:
            if node.right == 0 or node.right == 1:
                return node.right
            else:
                node = node.right
        elif input[3] == 0:
            if node.left == 0 or node.left == 1:
                return node.left
            else:
                node = node.left
    elif node.id == 4:
        if input[4] == 1:
            if node.right == 0 or node.right == 1:
                return node.right
            else:
                node = node.right
        elif input[4] == 0:
            if node.left == 0 or node.left == 1:
                return node.left
            else:
                node = node.left
```

2.3.3. Sample run (the output)



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

3. Discussion

1- Breadth-first search algorithm:

- In Class SearchAlgorithms initialize:
    1. Path list that will have the shortest path from start to end.
    2. FullPath list that will have all visited nodes from stat to end.
    3. Maze 2D array that will carry the whole maze in a 2D array form.
    4. Rows and columns have the number of rows and columns in the 2D array.
- In Class SearchAlgorithms write a function _init_ that will make 2 things:
    1. Take the string maze and covert it to 2D array and put it in the maze the 2D array initialized before.
    2. Count the number of Rows and columns and put it in the variables initialized before.

- In Class SearchAlgorithms write a function BFS that will make the BFS algorithms Steps:
    1. First initialize a queue that will be the queue used to implement the BFS and visited list that will carry the visited nodes to prevent processing on them again and x and y that’s carry the row and column of the Start node by searching for it on the 2D array maze and currentNode that will carry the value of the node will be proceeds on and n that will carry the node itself by creating and object from class Node and set its value and it’s id which is a pair of the node’s position (row and column) then append this node in the queue.
    2. Start a loop that will stop after reaching the End and in the loop dequeue from the queue node then put it in n. Then set x and y with n’s position stored in the id. Then append in the fullPath the ID of this node in the string by calculating it from x and y. Then set the currentNode with this node value.
    3. Enqueue the nodes from this node to all available moves in the 4 dimensions (Left, Right, Up, Down) and append them in the visited list and append the node itself.

4. Finally, outside the loop append the last point in the fullPath. Then loop from the end node and append it and its previous node until reaching the the start and append all this node in the path then reverse it.

2- Neural network algorithm:

- In NeuralNetwork Class there are 4 main functions:
    1. The initialize function which take two parameters the learning rate ant the threshold and initialize them in the class's variables and initialize number of random weights equal to the inputs number.
    2. Step Function which compare the output of the summation with the threshold if greater than return 1 else return 0.
    3. The think function which takes the inputs as a parameter and returns the expected output by applying the step of the activation function.
    4. The train function which takes three parameters the training inputs that the machine will train on them and their outputs which are the training outputs and the number of iterations used to get an accepted error, then loop by the number of iteration and in every iteration call the think function to get the expected output and then calculates the error and update the weights.

3- ID3 algorithm:

- In class ID3 there are five main functions:
  1. The initialization function which take the features as parameter and then call the buildtree function to build the decision tree from the dataset.
  2. The Entropy function that take a column feature and calculate the entropy by counting the number of zeros and ones then calculate and return the result of this equation.

$$\text{Entropy} = -\frac{P}{Total}\log_2\frac{P}{Total} - -\frac{N}{Total}\log_2\frac{N}{Total}$$

  3. The Get_Max_Gain Function that calculate the Gain for each column feature on the dataset by counting the number of zeros and ones and calculating the entropy by using the previous function entropy then calculate the result of this equation.

$$\text{Gain} = \text{Entropy} - \sum_{i=0}^{All\ children\ of\ attribute}\frac{Pi+Ni}{Total}(child)$$

     then return the column number of the maximum gain.

  4. The buildtree function that train on the dataset and make the decision tree by taking the position of the next node in the tree and by applying number of steps:
     - Get the column number of the maximum gain by calling the function Get_Max_Gain.

        - Initialize a node and add it in the tree according to the position send in the parameter either it is a root or a node that will be placed at the left of the previous node or to its right.
        - Take a copy of the data of the feature with the maximum gain then split the data according to the result of this copy.
        - Finally, check for the zeros sets and the ones if it is pure set all are zeros or ones or a further split will be done.
5. The classify function that takes an input array of features and determine the output one or zero according to the classification by following the decision tree starting from the root node to the required output.

4. References:

[1] Lab Video for ANN : https://cisasuedu-my.sharepoint.com/personal/halam_cis_asu_edu_eg/_layouts/15/

[2] ANN Lab Slides.

[3] ID3 Lab Video: https://web.microsoftstream.com/video/870ffce3-ba85-4a00-b5e4-c4933791e8f9?list=studio

[4] ID3 Lab Slides.

[5] BFS Lab 3 on coursesites AI course.