HW#3 Decision Tree

2016112083 김연웅

1. Codes Explanation

- **Class Node**

```
class Node:
    def __init__(self, column, value, dataset):
        self.column= column
        self.value = value
        self.dataset = dataset

        self.depth = None
        self.left = None
        self.right = None

        self.isLeaf = False

    def leaf(self):
        """leaf(self) : leafify(mark this node as a leaf) this node"""
        self.isLeaf = True
        self.left = None
        self.right = None

    def print_node(self):
        """
        print_node(self): print the node information depends on two cases
                  if this node is not a leaf node-> (XCOLUMN, VALUE). ex) (X1, 4.12346)
                  if this node is a leaf node -> Labele that this tree has decided. ex) (0)
        """


        space = ""

        # case not a leaf node
        if not self.isLeaf:
            space += (self.depth - 1) * "  "  # add spaces according to depth of this node
            print("%s(X%d, %f)" % (space, self.column + 1, self.value))
        # case leaf node
        else:
            space += (self.depth) * "  " # add spaces according to depth of this node
            print("%s(%d)"% (space, self.count_labels()))

    def count_labels(self):
```

```
        """
        count_labels : method for leaf node.  count numbers of labels in the given node`s dataset.
                returns a label value which appear the most in the given node`s dataset
        """
        temp =[]

        # put all labels of data
        for data in self.dataset:
            temp.append(data[-1])

        min = -1
        labels = list(set(temp))

        # for every labele appeared in this data
        for i in range(len(labels)):

            # count the appearance
            cnt = temp.count(labels[i])

            # save the most appeared label as a return value
            if cnt > min:
                min = cnt
                final_label = labels[i]

        return final_label

```

class Node represents a node in a tree. As the assignment specification says, its left branch represents a subset of DATA that meet “COLUMN < VALUE” and the right branch represents “COLUMN >= VALUE”.

It has 7 member variables and 3 methods which are explained below.

<members>

column: features of data to be splitted on

value: value of data of feature to be splited on

dataset: dataset which to be spliited and has been splited before

depth: depth of node in a tree

left : pointer to left node

right: pointer to right node

isLeaf: boolean value which represent that this node is wether a leaf or not.

True -> it is a leaf node, False -> not a leaf node

<methods>

leaf(self) : leafify(mark this node as a leaf) this node

print_node(self): print the node information depends on two cases

if this node is not a leaf node-> (XCOLUMN, VALUE). ex) (X1, 4.12346)

if this node is a leaf node -> final result Label that this tree has decided so far. ex) (0)

count_labels : method for leaf node. count numbers of labels in the given node`s dataset.

returns a label value which appear the most in the given node`s dataset

■ **Function gini_impurity**

```
def gini_impurity(group_A, group_B):

    """
    Calculate the Gini impurity (a.k.a Gini index) of two groups of dataset
.(left and right)

    """
    # variables initialization
    num_A = len(group_A)
    num_B = len(group_B)
    num_total = num_A + num_B

    table_A = []
    table_B = []
    labels = []

    # save labels info into each group`s list
  for data in group_A:
        labels.append(data[-1])
        table_A.append(data[-1])
    for data in group_B:
        labels.append(data[-1])
        table_B.append(data[-1])

    # variable which contains all the labels appeared among two datasets
    labels = list(set(labels))



    if num_A == 0:      # avoid devide by zero
        gini_A = 0
    else:
        gini_A = 1      # calculate gini index for group A
        for label in labels:
            term = (table_A.count(label) / num_A) ** 2
            gini_A -= term

    if num_B == 0:      # avoid devide by zero
        gini_B = 0
    else:
        gini_B = 1       # calculate gini index for group B
        for label in labels:
            term = (table_B.count(label) / num_B) ** 2
            gini_B -= term

    # calculate final gini index between two groups and return
    gini = gini_A * (num_A / num_total) + gini_B * (num_B / num_total)

    return gini
```

This function calculates gini impurity of two data groups which were given as parameters. Group A consist of data which has smaller COLUMN VALUE and group B consists of data which has larger or equal to COLUMN VALUE when split performed in the given node. First, it figure out what kinds of class labels are in those two data groups, then it calculates gini values.

- **Function split**

```
def split(dataset):

    # initialize variables
    node = Node(0,0,[])
    min_gini = 100000;

    # for every featrues(columns) in the given dataset... (except label column)
    for feature in range(len(dataset[0]) - 1):
        # for every data in the given dataset...
        for data in dataset:
            group_A = []
            group_B = []

            # define a thresh for this iteraation which is a combination of COLUMN and VALUE
            thresh = data[feature]

            # split into two groups
            for i in range(len(dataset)):
                if dataset[i][feature] < thresh:
                    group_A.append(dataset[i])
                elif dataset[i][feature] >= thresh:
                    group_B.append(dataset[i])
            # calculate gini_impurity of this divide
            cur_gini = gini_impurity(group_A, group_B)

            # if the gini_impurity is lowest so far, save it and continue
            if cur_gini < min_gini:
                min_gini = cur_gini
                node.column = feature
                node.value = thresh
                node.dataset = [group_A,group_B]

            # if the gini impurity tie
            elif cur_gini == min_gini:

                # choose the COLUMN and the VALUE that lead to the most balanced split,
                # i.e., the absolute different between size(DATA) of the left child and size(DATA) of the right child should be minimized.
                # If tie again for the balance, just pass this iteration.
                cur_diff = abs(len(group_A) - len(group_B))
```

```
                    min_diff = abs(len(node.dataset[0]) - len(node.dataset[1]))
                    if cur_diff < min_diff:
                        min_gini = cur_gini
                        node.column = feature
                        node.value = thresh
                        node.dataset = [group_A,group_B]

    return node

```

This is a helper function of recursive_split. It splits dataset into two groups by a combination of best feature and value. Returns a node which has spitted datasets into two groups (group A and group B)
It is done by computing the combination of which yields purest two dataset split in terms of labels by using gini_impurity function above. It uses brute force methods to find out best combination of COLUMN, VALUE. It loops every combination of columns and data value and this combination is used for split threshold. For each loop, it loops again for every data in dataset to figure out whether the data`s column value is smaller than current threshold or larger threshold. If current data`s column value is smaller, it goes to group A, and otherwise, group B. After splits has done, it calculates gini impurity by using gini_impurity function above. If these two group`s gini_impurity is the smallest value of all, the two groups and the combination of COLUMN, VALUE are saved for returning node`s dataset. In case for tie gini impurity, it chooses the COLUMN and the VALUE that lead to the most balanced split, (the absolute different between size(DATA) of the group A and size(DATA) of the group B) If tie again for the balance, it goes on with original minimum combinations.

- **Function recursive_split**

```
def recursive_split(node):
    group_A = node.dataset[0]
    group_B = node.dataset[1]

     #1. if either of each group consist of zero data, then make the next left and right nodes to be leaf with data it got so far.
     # if group A has no data
    if len(group_A) == 0:
        left_leaf =  Node(node.column, node.value, group_B)
        left_leaf.depth = node.depth  # for a leaf, no increase depth value
        left_leaf.leaf()              # leafify left branch

        right_leaf = Node(node.column, node.value, group_B)
        right_leaf.depth= node.depth     # for a leaf, no increase depth value
        right_leaf.leaf()                # leafify right branch

        node.left = left_leaf
        node.right = right_leaf
        return;

```

```
     #1. if either of each group consist of zero data, then make the next left and right nodes to be leaf with data it got so far.
     # if group B has no data
    elif len(group_B) == 0:
        left_leaf =  Node(node.column, node.value, group_A)
        left_leaf.depth = node.depth    # for a leaf, no increase depth value
        left_leaf.leaf()                 # leafify left branch

        right_leaf = Node(node.column, node.value, group_A)
        right_leaf.depth= node.depth    # for a leaf, no increase depth value
        right_leaf.leaf()               # leafify right branch

        node.left = left_leaf
        node.right = right_leaf
        return;

    # 2. if we reach  max depth , make next nodes to be leafs(groupA for left, group B for right)
    if node.depth >= max_depth:

        # leafify left and right child
        left_leaf =  Node(node.column, node.value, group_A)
        left_leaf.depth = node.depth
        left_leaf.leaf()

        right_leaf = Node(node.column, node.value, group_B)
        right_leaf.depth=node.depth
        right_leaf.leaf()


        node.left = left_leaf
        node.right= right_leaf

        return;
    # 3. check if left group(group A) is big enough than min_samples_split.
    # 3-1) if it has smaller number of data, then make left child as a leaf
    if len(group_A) <= min_samples_split:
        left_leaf = Node(node.column, node.value, group_A)
        left_leaf.depth = node.depth
        left_leaf.leaf()

        node.left= left_leaf

    # 3-2) otherwise add left node by recursively.
    else:
        left_node = split(group_A)
```

```
        left_node.depth = node.depth + 1 # add depth value because next left node is not a leaf
        node.left = left_node

        recursive_split(left_node)      # recursively perform these sequences



    # 4. do the same thing with 3-1~3-2 for right group
    if len(group_B) <= min_samples_split:
        right_leaf = Node(node.column, node.value, group_B)
        right_leaf.depth = node.depth
        right_leaf.leaf()

        node.right = right_leaf
    else:
        right_node = split(group_B)
        right_node.depth = node.depth + 1   # add depth value because next left node is not a leaf
        node.right = right_node

        recursive_split(right_node)         # recursively perform these sequences
```

Recursively split nodes(grow trees) until it met some conditions. It has a Node as a parameter and perform several check to whether proceed splitting or not. If all the check steps are done, create a child node by using split function above and then recursively call the child node itself to further grow the tree.

The sequences of function are as follows:

1. if either of each group consist of zero data, then make the next left and right nodes to be leaf with data it got so far.
2. if current node reaches max depth , make next nodes to be leafs (group A for left, group B for right)
3. check if left group(group A) is big enough than min_samples_split.
   3-1) if it has smaller number of data, then make left child as a leaf.
   3-2) otherwise create(split) and add left node by recursively.
4. do the same thing with 3-1~3-2 for the right child.

Few more things for implementation. If next child node is a leaf node, the newly created leaf node`s depth value won`t increase. Only when creating new decision node, depth value will be increased.

- **Function my_tree(dataset)**

```
def my_tree(dataset):
  # split the initial root node
  root = split(dataset)

  # set depth value for root node
  root.depth = 1
```

```
    # recursively build tree
    recursive_split(root)

    return root
```

Build tree using methods explained above. It gets raw dataset as a parameter. It make first root node of a tree by using split method. Then it calls recursive_split for growing tree. It returns root node, which is connected with every nodes of a created tree.

- **Function print_tree**

```
def print_tree(node):
    """
    print the trained tree in the depth-
first manner. refer to the lecture slides.
    pre-order traverse
    """
    if node is not None:
        node.print_node()
        print_tree(node.left)
        print_tree(node.right)
```

simple tree-printing method in the depth -first manner (pre-order tree traverse)

2. Testing Result.

I tested my decision tree codes according to the assignment`s requirement. In this section, I will attach screen captures of result tree printing, for each requiring configurations.

The test dataset is as followed.

```
[[2.2343124,1.123123,0],
[1.43523,1.54245,0],
[3.53467889,2.234987,0],
[3.1249876,2.09237512893,0],
[2.1238756,9.3253154,1],
[7.0981274,3.89074,1],
[1.129875,3.0987234,0],
[7.0897345,0.089745,1],
[6.0987214,3.0978214,1],
[6.1325,3.98763,1],
[1.35765,2.43663,0],
[2.345,3.3456,0],
[0.2345,1.4356,0],
[2.4356,5.67534,0],
[5.234,5.23465,1],
[4.12346,2.975,1],
[2.5467,4.72345,0],
[8.4612,1.6269,1],
[5.215690,2.5362,1],
```

```
[4.762,1.76567,1]]
```

Max_depth and min_samples_split configurations are as followed.

- (max_depth = 1, min_samples_split = 2)
- (max_depth = 2, min_samples_split = 2)
- (max_depth = 2, min_samples_split = 10)
- (max_depth = 3, min_samples_split = 2)

Testing Codes

```
# datasets
dataset = [ [2.2343124,1.123123,0],
            [1.43523,1.54245,0],
            [3.53467889,2.234987,0],
            [3.1249876,2.09237512893,0],
            [2.1238756,9.3253154,1],
            [7.0981274,3.89074,1],
            [1.129875,3.0987234,0],
            [7.0897345,0.089745,1],
            [6.0987214,3.0978214,1],
            [6.1325,3.98763,1],
            [1.35765,2.43663,0],
            [2.345,3.3456,0],
            [0.2345,1.4356,0],
            [2.4356,5.67534,0],
            [5.234,5.23465,1],
            [4.12346,2.975,1],
            [2.5467,4.72345,0],
            [8.4612,1.6269,1],
            [5.215690,2.5362,1],
            [4.762,1.76567,1]
          ]

# configure parameters
max_depth = 1
min_samples_split = 2

# codes for building tree and print!
tree = my_tree(dataset)

print("Decision Tree")
print("max_depth : %d" % (max_depth))
print("min_samples_split: %d" % (min_samples_split))
print()
print_tree(tree)
```

For following screen capture`s result, I used above same test codes, only changing values of max_depth and min_samples_splits variables.

■ (max_depth = 1, min_samples_split = 2)

```
Decision Tree
max_depth : 1
min_samples_split: 2

(X1, 4.123460)
  (0)
  (1)
```

■ (max_depth = 2, min_samples_split = 2)

```
Decision Tree
max_depth : 2
min_samples_split: 2

(X1, 4.123460)
  (X2, 9.325315)
    (0)
    (1)
  (X1, 6.098721)
    (1)
    (1)
```

■ (max_depth = 2, min_samples_split = 10)

```
Decision Tree
max_depth : 2
min_samples_split: 10

(X1, 4.123460)
  (X2, 9.325315)
    (0)
    (1)
  (1)
```

■ (max_depth = 3, min_samples_split = 2)

```
Decision Tree
max_depth : 3
min_samples_split: 2

(X1, 4.123460)
  (X2, 9.325315)
    (X1, 2.345000)
      (0)
      (0)
    (1)
  (X1, 6.098721)
    (X1, 5.215690)
      (1)
      (1)
    (X1, 7.098127)
      (1)
      (1)
```