MyDT

September 27, 2019

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_val_score,train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import timeit
import copy
```

In [2]:
```
def Gini_index(input_vec):
    _, counts = np.unique(input_vec, return_counts=True)
    gini_index_val = 1
    for i in range(counts.shape[0]):
        gini_index_val -= (counts[i]/input_vec.shape[1])**2
    return gini_index_val
```

In [3]:
```
from scipy.stats import entropy
def Entropy(input_vec):
    value, counts = np.unique(input_vec, return_counts=True)
    entropy_val = entropy(counts, base=2)
    return entropy_val
```

In [4]:
```
def purity(true_set, false_set, impurity_measure='entropy'):
    """Checks the purity of sets passed to it based on their labels
    information gain of the criterion used to divide the dataset
    input: two sets divided by a specific criterion along with
    the specification of impurity measure
    output: the gain or purity of such division"""

    true_set_classes = np.array([true_set[:,len(true_set[0,:])-1]])
    false_set_classes = np.array([false_set[:,len(false_set[0,:])-1]])
    All_classes = np.array([np.concatenate((true_set_classes, false_set_classes), axis=

    if impurity_measure == 'gini':
        purity = Gini_index(All_classes)-\
        (len(true_set_classes[0,:])/len(All_classes[0,:])) * Gini_index(true_set_classe
        (len(false_set_classes[0,:])/len(All_classes[0,:])) * Gini_index(false_set_clas
    else:
        # if purity measure is information gain
```

```
            purity = Entropy(All_classes)-\
            (len(true_set_classes[0,:])/len(All_classes[0,:])) * Entropy(true_set_classes)-
            (len(false_set_classes[0,:])/len(All_classes[0,:])) * Entropy(false_set_classes

        return purity
```

In [5]:
```
class Criterion:
    """The criterion objects instantiated from Criterion class are used to devide a da

    def __init__(self, attribute_index, middle_point):
        self.attribute_index = attribute_index
        self.middle_point = middle_point

    def check(self, sampel):
        """Check if the value in the specified attribute of the sampel is greater or l
        return sampel[self.attribute_index] >= self.middle_point
```

In [6]:
```
def divide(data, criterion):
    """Divides the input dataset to two sets based on the passed criterion
    input: a dataset (to be divided), and a criterion
    output: two sets of data divided according to the passed criterion"""

    true_set, false_set = np.array([]), np.array([])

    for sampel in data:
        if criterion.check(sampel):
#             true_set.append(sampel)
            if true_set.size == 0:
                true_set = np.array([sampel])
            else:
                true_set = np.concatenate((true_set, np.array([sampel])), axis=0)
        else:
#             false_set.append(sampel)
            if false_set.size == 0:
                false_set = np.array([sampel])
            else:
                false_set = np.concatenate((false_set, np.array([sampel])), axis=0)

    return true_set, false_set
```

In [7]:
```
def find_best_attribute(data, impurity_measure='entropy'):
    """Find the best attribute for using in the node for classification.
    input: a subset of data, including all the attributes and the labels.
    output: best_value (best purity value),...
    best_criterion (best attribute to be used for classification along with the criter

    best_value = 0  # best value for purity measure (with information gain or gini ind
    best_criterion = None  # best attribute along with the criterion for classificatio
```

```
            for attribute_i in range(len(data[0,:]) - 1):  # for each attribute
                # middle point in the range of values for one attribute
                mid_range = np.amin(data[:,attribute_i]) +\
                (np.amax(data[:,attribute_i])-np.amin(data[:,attribute_i]))/2
                criterion = Criterion(attribute_i,mid_range)

                # devide the data based on the selected attribute
                # (and the mean point of the range of values of
                # that attribute)
                true_set, false_set = divide(data, criterion)


                if not (true_set.shape[0] == 0 or false_set.shape[0] == 0):
                    # calculate the purity measure after devistion
                    # adopting the above criterion
                    purity_val = purity(true_set, false_set, impurity_measure)
                    if purity_val>best_value:
                        best_value, best_criterion = purity_val, criterion

            return best_value, best_criterion
```

```
In [31]: class End_Node:
             def __init__(self, data):
                 unique, counts = np.unique(data[:,len(data[0,:])-1], return_counts=True)
                 self.label = unique[np.argmax(counts)]

             def __del__(self):
                 pass
```

```
In [34]: class Node:
             def __init__(self,criterion, true_branch, false_branch):
                 self.criterion = criterion
                 self.true_branch = true_branch
                 self.false_branch = false_branch

             def __del__(self):
                 pass
```

```
In [10]: def grow_tree(data, impurity_measure='entropy'):
             """A recursive function for growing a decision tree by learning from data
             input: a set of data
             output: a learnt tree based on the input data"""

             purity_val, criterion = find_best_attribute(data, impurity_measure)

             if purity_val==0:
                 return End_Node(data)
```

```
            else:
                true_set, false_set = divide(data, criterion)

                # the result of grow_tree() can be an end node (leaf)
                # or a node (tree). Thus, true_branch and false_branch
                # can be either leaf or tree.
                true_branch = grow_tree(true_set, impurity_measure)
                false_branch = grow_tree(false_set, impurity_measure)

                node = Node(criterion, true_branch, false_branch)
                return node
```

```
In [11]: def Tree_Predict(tree, sampel):
            if isinstance(tree, End_Node):
                prediction = tree.label
            else:
                if tree.criterion.check(sampel):
                    prediction = Tree_Predict(tree.true_branch, sampel)
                else:
                    prediction = Tree_Predict(tree.false_branch, sampel)
            return prediction
```

```
In [12]: def tree_visualization(tree, depth=0):
            space = '                '
            if isinstance(tree, End_Node):
                print(depth*space, "Leaf from depth {}:".format(depth), tree.label)
        #         print(tree.label)
            else:
                tree_criterion_att = tree.criterion.attribute_index
                tree_criterion_mid = tree.criterion.middle_point
                print(depth*space, "Attribute {}. ".format(tree_criterion_att),\
                      'Is X[{}]>={}?'.format(tree_criterion_att, tree_criterion_mid))
                tree_true_branch = tree.true_branch
                tree_false_branch = tree.false_branch
                print(depth*space, "true branch from depth {}:".format(depth))
                tree_visualization(tree_true_branch, depth+1)
                print(depth*space, "false branch from depth {}:".format(depth))
                tree_visualization(tree_false_branch, depth+1)
            return
```

```
In [35]: def accuracy_for_a_set(tree, data):
            """calculates the accuracy of a tree for a set of data
            input: a tree and a set of data
            output: accuracy value for classification for the given data based on the given t

            num_correctly_classified = 0
            for i in range(data.shape[0]-1):
                if Tree_Predict(tree, data[i,:])==data[i,data.shape[1]-1]:
```

```
                num_correctly_classified += 1
        accuracy_val = num_correctly_classified/data.shape[0]
        return accuracy_val
```

In [16]:
```
def prune_tree(tree, pruning_set, father_node=None, true_side=False, false_side=False)
    """prunes the passed tree based on the passed pruning set
    input: the tree and the pruning set
    output: the pruned tree"""

    def change(PT_node, input_set):
        change_flag = False
        unique, counts = np.unique(input_set[:,len(input_set[0,:])-1], return_counts=T
        current_accuracy = accuracy_for_a_set(PT_node, input_set)
        changing_accuracy = (np.max(counts)/pruning_set.shape[0])
        if current_accuracy<changing_accuracy:
            change_flag = True
        return change_flag


    pruned_tree = copy.deepcopy(tree)
    # if passed tree is Leaf
    if isinstance(tree, End_Node):
        pruned_tree = End_Node(pruning_set)
    else: #passed tree is a Node
        true_set, false_set = divide(pruning_set, tree.criterion) # divide the pruning
        if true_set.shape[0] == 0: # when true set is empty
            """true set is empty"""
            pruned_tree.true_branch = copy.deepcopy(tree.true_branch) # true side is
            if isinstance(tree.true_branch, End_Node) and isinstance(tree.false_branch
                if change(tree.false_branch, pruning_set): # should we change the fal
                    pass
#                     pruned_tree.false_branch = End_Node(pruning_set) # the false si
            elif isinstance(tree.true_branch, End_Node): # the true side is leaf (fal
                pruned_tree.false_branch = copy.deepcopy(prune_tree(tree.false_branch
            elif isinstance(tree.false_branch, End_Node): # the false side is leaf
                if change(tree.false_branch, pruning_set): # should we change the fal
                    pass
#                     pruned_tree.false_branch = End_Node(pruning_set) # the false si
            else: # both childeren are decision node (not leaf)
                pruned_tree.false_branch = copy.deepcopy(prune_tree(tree.false_branch
            """true set is empty"""
        elif false_set.shape[0] == 0: # when false set is empty
            """false set is empty"""
            pruned_tree.false_branch = copy.deepcopy(tree.false_branch)
            if isinstance(tree.true_branch, End_Node) and isinstance(tree.false_branch
                if change(tree.true_branch, pruning_set): # should we change the true
                    pass
#                     pruned_tree.true_branch = End_Node(pruning_set)  # the true sid
```

```
            elif isinstance(tree.true_branch, End_Node): # the true side is leaf
                if change(tree.true_branch, pruning_set): # should we change the true
                    pass
#                     pruned_tree.true_branch = End_Node(pruning_set)  # the true sid
            elif isinstance(tree.false_branch, End_Node): # the false side is leaf (t
                pruned_tree.true_branch = copy.deepcopy(prune_tree(tree.true_branch, p
            else: # both childeren are decision node (not leaf)
                pruned_tree.true_branch = copy.deepcopy(prune_tree(tree.true_branch, p
            """false set is empty"""
        else: # when both sets are not empty
            """both sets are not empty"""
            if isinstance(tree.true_branch, End_Node) and isinstance(tree.false_branch
                if change(tree.true_branch, true_set): # should we change the true si
                    pass
#                     pruned_tree.true_branch = End_Node(true_set)
                if change(tree.false_branch, false_set): # should we change the false
                    pass
#                     pruned_tree.false_branch = End_Node(false_set)
            elif isinstance(tree.true_branch, End_Node): # the true side is leaf
                if change(tree.true_branch, true_set): # should we change the true si
                    pass
#                     pruned_tree.true_branch = End_Node(true_set)
                pruned_tree.false_branch = copy.deepcopy(prune_tree(tree.false_branch
            elif isinstance(tree.false_branch, End_Node): # the false side is leaf
                pruned_tree.true_branch = copy.deepcopy(prune_tree(tree.true_branch, t
                if change(tree.false_branch, false_set): # should we change the false
                    pass
#                     pruned_tree.false_branch = End_Node(false_set)
            else: # both childeren are decision node (not leaf)
                pruned_tree.true_branch = copy.deepcopy(prune_tree(tree.true_branch, t
                pruned_tree.false_branch = copy.deepcopy(prune_tree(tree.false_branch
            """both sets are not empty"""
        if change(pruned_tree, pruning_set):  # after pruning bothsides, should we ch
            pruned_tree = End_Node(pruning_set)


    return pruned_tree
```