Final Project - Unit Test Cases

SCC461 - Programming for Data Scientists

Kyriakos Chiotis - 35278597

The implementation of decision tree is based on 2 classes, Node and MyDecisionTree. Node class represents the nodes on a tree and MydecisionTree represents the tree. Both classess with some extra helping functions are going to be explained and be tested, using a custom synthetic dataset. The dataset is provided below.

In [55]:

```
# Color and Length
x_train = [
    ['white', 30],
    ['black', 30],
    ['grey', 10],
    ['grey', 10],
    ['black', 30],
]
# Target classes
y_train = ['cat', 'cat', 'mouse', 'mouse', 'rat']
dataset = [x_train, y_train]
```

The first helping function is "split_records(split_point, records, feature_index)". This function splits the rows of the dataset based on a question. For example, "Is color white?" or "Is length >= 11?". According to the parameters, the question is formulated "Is records[0][feature_index] >= split_point ?" in case of numerical feature and "Is records[0][feature_index] == split_point ?" in case of categorical feature. The output is two lists like

In [56]:

```
def split_records(split_point, records, feature_index):
    true_records = [[], []]
    false_records = [[], []]
    # Check the type of feature. Numerical or categorical.
    # numerical case
    if isinstance(split_point, int) or isinstance(split_point, float):
        index = 0
        for rec in records[0]:
            if rec[feature_index] >= split_point:
                true_records[0].append(rec)
                true_records[1].append(records[1][index])
            else:
                false_records[0].append(rec)
                false_records[1].append(records[1][index])
            index += 1
    # categorical case
    else:
        index = 0
        for rec in records[0]:
            if rec[feature_index] == split_point:
                true_records[0].append(rec)
                # del true_records[0][-1][feature_index]
                true_records[1].append(records[1][index])
            else:
                false_records[0].append(rec)
                # del false_records[0][-1][feature_index]
                false_records[1].append(records[1][index])
            index += 1
    return true_records, false_records
```

Test cases

In [57]:

```
# Split dataset for color = white
left_branch, right_branch = split_records('white', dataset, 0)
left_branch
```

Out[57]:

```
[[['white', 30]], ['cat']]
```

In [58]:

```
right_branch
```

Out[58]:

```
[[['black', 30], ['grey', 10], ['grey', 10], ['black', 30]],
 ['cat', 'mouse', 'mouse', 'rat']]
```

In [59]:

```
# Split right branch(color =! white) for length >= 30
left_branch, right_branch = split_records(30, right_branch, 1)
left_branch
```

Out[59]:

```
[[['black', 30], ['black', 30]], ['cat', 'rat']]
```

In [60]:

```
right_branch
```

Out[60]:

```
[[['grey', 10], ['grey', 10]], ['mouse', 'mouse']]
```

A function to measure the rows.(just to make it a little bit simpler visually)

In [61]:

```
def n_rows(records):
    return len(records[1])
```

Test case

In [62]:

```
n_rows(dataset)
```

Out[62]:

```
5
```

The next very important function is unique_counts(lst). This function takes as an input a list and returns a dictionary with keys the unique items of the list and values the number of the items.

In [63]:

```
def unique_counts(lst):
    dict = {}
    for i in range(len(lst)):
        if lst[i] not in dict:
            dict[lst[i]] = 0
        dict[lst[i]] += 1
    return dict
```

Test case

In [64]:

```
unique_counts(y_train)
```

Out[64]:

```
{'cat': 2, 'mouse': 2, 'rat': 1}
```

This function is usefull for counting the class labels at each node and then make a prediction which leads to the next function, called get_prediction(leaf). This function returns the most frequent item of a list which means the prediction of the target class.

In [65]:

```
def get_prediction(leaf):
    return max(unique_counts(leaf[1]), key=unique_counts(leaf[1]).get)
```

Test cases According to the test case above it is obvious that cat has bigger frequency than mouse and rat. This is what get_prediction returns, too.

In [66]:

```
get_prediction(dataset)
```

Out[66]:

```
'cat'
```

It is essential to stress out that there is no case a function receives empty data. This case is checked later.

The next function calculates the gini impurity of the data. The formula is:

$$G = 1 - \sum_{k=1}^{n} p_k^2$$

where n is the number of training samples and p_k is the fraction of samples belonging to class k.

In [67]:

```
def gini(records):
    returned_gini = 1
    for val in unique_counts(records[1]).values():
        returned_gini -= (val / (n_rows(records))) ** 2
    return returned_gini
```

Test case

In [68]:

```
gini(dataset)
```

Out[68]:

```
0.6399999999999999
```

Ather setting these functions, Node class could be created.

In [69]:

```
class Node:

    def __init__(self, split_point, records, feature_index, threshold):
        self.split_point = split_point  # numeric or str value to split
        self.records = records  # list[features[[],[],[]], labels[0,1,1]]
        self.feature_index = feature_index
        self.true_records, self.false_records = split_records(split_point, records, fea
ture_index)  # sublist of records
        self.impurity = gini(records)
        self.left_node = None
        self.right_node = None
        self.threshold = threshold

    def info_gain(self):
        info = self.impurity - \
               (gini(self.true_records) * n_rows(self.true_records) / (n_rows(self.reco
rds))) - \
               (gini(self.false_records) * n_rows(self.false_records) / (n_rows(self.re
cords)))
        if info <= self.threshold:
            return 0
        return info
```

A node is constructed with input parameters the same with "split_records" function plus a threshold parameter which is the imprutiry decrease threshold or information gain threshold. Furthermore, in constructor each node set the true and false records with "split_records" function, the current impurity of the node with "gini" function and the left and right node to None.

The only method of this class is the info_gain(self) which is the Gini for the parent node minus the weighted average of Gini impurities for the children nodes. It is called information gain for the parent node and it is calculated under the formula below:

$$Info\_Gain = G - \left(\frac{i}{m}G^{left} + \frac{m-i}{m}G^{right}\right)$$

where m the elements on the parent node, i the elements on the left node and $m-i$ on the right.

Test cases

In [70]:

```
# How much information do we gain by splitting on color 'white'?
test_node = Node("white",dataset,0,0)
test_node.info_gain()
```

Out[70]:

```
0.13999999999999999
```

In [71]:

```
# On grey?
test_node = Node("grey",dataset,0,0)
test_node.info_gain()
```

Out[71]:

```
0.3733333333333332
```

In [72]:

```
# On black?
test_node = Node("black",dataset,0,0)
test_node.info_gain()
```

Out[72]:

```
0.17333333333333323
```

In [73]:

```
# How much information do we gain by splitting on length >=10? It will be 0 while it co
ntains all the original data
# and no information is gained with this split.
test_node = Node(10,dataset,1,0)
test_node.info_gain()
```

Out[73]:

```
0
```

In [74]:

```
# On length >= 30
test_node = Node(30,dataset,1,0)
test_node.info_gain()
```

Out[74]:

```
0.3733333333333332
```

It is shown that the best question that can be formulated in the first split is if length >=30 or color is 'grey' as it gets the higher information gain.

Therefore, it is essential a function that calculates the information gain of all possible nodes and find the best split which is actually the best node. This functions is demonstrated below:

In [75]:

```
def best_split(x_train, y_train, threshold):
    node = None
    best_gain = 0
    if not x_train:
        return node
    for feature_index in range(len(x_train[0])):
        unique_points = []
        for row in x_train:
            if row[feature_index] not in unique_points:
                unique_points.append(row[feature_index])
                temp_node = Node(row[feature_index], [x_train, y_train], feature_index, 
threshold)
                # This is row can change to take the first best node
                # Now it takes the last one
                if temp_node.info_gain() >= best_gain and temp_node.info_gain() != 0:
                    best_gain = temp_node.info_gain()
                    node = Node(row[feature_index], [x_train, y_train], feature_index, 
threshold)

    return node
```

Test case

In [76]:

```
best_split(x_train, y_train,0).feature_index
```

Out[76]:

```
1
```

In [77]:

```
best_split(x_train, y_train,0).split_point
```

Out[77]:

```
30
```

In [78]:

```
best_split(x_train, y_train,0).info_gain()
```

Out[78]:

```
0.3733333333333332
```

It is shown that indeed the function chose the node with the higher information gain. But the information gain is equal for two nodes. This can be changed to get the first or the last best node. The original decision tree uses randomness on this selection. For the purposes of this assignment we just change the if else statement to check different fitting.

Finally, MyDecisionTree class can be declared. This class is used to build the tree and make predictions.

In [79]:

```
class MyDecisionTree:

    def __init__(self, max_depth=100000, impurity_threshold=0):
        self.max_depth = max_depth
        self.impurity_threshold = impurity_threshold

    def create_tree(self, x_train, y_train, depth=0):
        tree = best_split(x_train, y_train, self.impurity_threshold)
        if tree is None or depth >= self.max_depth:
            return

        tree.left_node = self.create_tree(tree.true_records[0], tree.true_records[1], d
epth + 1)
        tree.right_node = self.create_tree(tree.false_records[0], tree.false_records[1
], depth + 1)
        return tree

    def predict(self, x_test, tree):
        y_pred = []
        for row in x_test:
            # Based on feature_index and splitting point of tree node choose branch
            # Iterate until the chosen branch is None and return the prediction
            tree_climber = tree
            while True:
                index = tree_climber.feature_index
                value = tree_climber.split_point
                # Numerical test case
                if isinstance(value, int) or isinstance(value, float):
                    if row[index] >= value:
                        if tree_climber.left_node is not None:
                            tree_climber = tree_climber.left_node
                            continue
                        else:
                            y_pred.append(get_prediction(tree_climber.true_records))
                            break
                    else:
                        if tree_climber.right_node is not None:
                            tree_climber = tree_climber.right_node
                            continue
                        else:
                            y_pred.append(get_prediction(tree_climber.false_records))
                            break
                # Categorical test case
                else:
                    if value == row[index]:
                        if tree_climber.left_node is not None:
                            tree_climber = tree_climber.left_node
                            continue
                        else:
                            y_pred.append(get_prediction(tree_climber.true_records))
                            break
                    else:
                        if tree_climber.right_node is not None:
                            tree_climber = tree_climber.right_node
                            continue
                        else:
                            y_pred.append(get_prediction(tree_climber.false_records))
                            break
```

```
        return y_pred

    # Debugging purposes
    def get_depth(self, node):
        if node is None:
            return 0
        else:
            left_depth = self.get_depth(node.left_node)
            right_depth = self.get_depth(node.right_node)

            if left_depth > right_depth:
                return left_depth + 1
            else:
                return right_depth + 1

    def print_tree(self, tree, space='', orientation='Root--> ', leaf=None):
        if tree is None:
            print(space, orientation, 'Leaf')
            print((len(space) + len(orientation)) * ' ', '-Samples:', len(leaf[1]))
            print((len(space) + len(orientation)) * ' ', '-value:', unique_counts(leaf[
1]))
            print((len(space) + len(orientation)) * ' ', '-class:', max(unique_counts(l
eaf[1]), key=unique_counts(leaf[1]).get))
            return
        print(space, orientation, 'Is feature ', tree.feature_index, ' <= ', tree.split
_point)
        print((len(space) + len(orientation)) * ' ', '-Impurity:', round(tree.impurity,
3))
        print((len(space) + len(orientation)) * ' ', '-Samples:', len(tree.records[1]))
        print((len(space) + len(orientation)) * ' ', '-value:', unique_counts(tree.reco
rds[1]))
        print((len(space) + len(orientation)) * ' ', '-class:', get_prediction(tree.rec
ords))

        self.print_tree(tree.right_node, space + '    ', 'Left--> ', tree.false_records
)
        self.print_tree(tree.left_node, space + '    ', 'Right--> ', tree.true_records)
```

This class is initialised base on the two parameters max depth and min impurity decrease. The two most important methods are "create_tree" and "predict". Method "create_tree" is using recursiveness in order to find find the best split nodes and build the tree. When no other split can be done, it returns the tree. Method "predict" is trying to predict the class in a test set based on the tree that was created. For each record in the test set, it is iterating the tree until the chosen branch is None and then it returns the prediction.

Finally, "print_tree" is used for debugging purposes and comparison with Sklearn's decision tree classifier.

Test cases

In [80]:

```
mydc = MyDecisionTree()
tree = mydc.create_tree(x_train, y_train)
mydc.print_tree(tree)
```

```
 Root-->  Is feature  1  <=  30
        -Impurity: 0.64
        -Samples: 5
        -value: {'cat': 2, 'mouse': 2, 'rat': 1}
        -class: cat
    Left-->  Leaf
            -Samples: 2
            -value: {'mouse': 2}
            -class: mouse
    Right-->  Is feature  0  <=  black
             -Impurity: 0.444
             -Samples: 3
             -value: {'cat': 2, 'rat': 1}
             -class: cat
        Left-->  Leaf
                -Samples: 1
                -value: {'cat': 1}
                -class: cat
        Right-->  Leaf
                 -Samples: 2
                 -value: {'cat': 1, 'rat': 1}
                 -class: cat
```

In [81]:

```
# Reducing the depth to 1.
mydc = MyDecisionTree(max_depth=1)
tree = mydc.create_tree(x_train, y_train)
mydc.print_tree(tree)
```

```
 Root-->  Is feature  1  <=  30
        -Impurity: 0.64
        -Samples: 5
        -value: {'cat': 2, 'mouse': 2, 'rat': 1}
        -class: cat
    Left-->  Leaf
            -Samples: 2
            -value: {'mouse': 2}
            -class: mouse
    Right-->  Leaf
             -Samples: 3
             -value: {'cat': 2, 'rat': 1}
             -class: cat
```

In this part, it is analysed the unit testing in order to verify that the classifier is operating properly. The comparison with Sklearn's classifier will be done with iris dataset in IrisTestCases.pdf. This dataset was chosen because we need numerical features in order to compare them.