HW4-Programming

● Graded

Student

Brian Bertness

Total Points

30 / 30 pts

Autograder Score

16.0 / 16.0

Passed Tests

test_combined_metric (test_DecisionTree.TestDecisionTree) (4/4)
test_entropy (test_DecisionTree.TestDecisionTree) (4/4)
test_generate_tree (test_DecisionTree.TestDecisionTree) (4/4)
test_gini_index (test_DecisionTree.TestDecisionTree) (4/4)

Question 2

Sanity Check — **14** / 14 pts

✔ **– 0 pts** Correct

– 14 pts Incorrect

Autograder Results

test_combined_metric (test_DecisionTree.TestDecisionTree) (4/4)

The ground truth combined_entropy is: 2.959717
Your combined_entropy is: 2.959717

test_entropy (test_DecisionTree.TestDecisionTree) (4/4)

The ground truth entropy is: 3.311140
Your entropy is: 3.311140

test_generate_tree (test_DecisionTree.TestDecisionTree) (4/4)

For testing, we assume min entropy is 0.1
The testing for generate tree is conducted on the test set rather than training set here
The groundtruth tree node list is:
[42, [30, [44, [33, [None], [19, [2, [None], [4, [None], [5, [None], [None]]]], [None]]], [28, [None], [None]]], [
Your tree node list is:
[42, [30, [44, [33, [None], [19, [2, [None], [4, [None], [5, [None], [None]]]], [None]]], [28, [None], [None]]], [
The tree node list follows the following pattern: [root, [left], [right]]
Please run with Python3 rather than Python2 code.

test_gini_index (test_DecisionTree.TestDecisionTree) (4/4)

The ground truth gini_index is: 0.898545
Your gini_index is: 0.898545

Submitted Files

MyDecisionTree.py Download

```python
import numpy as np


# You are going to implement functions for this file.

class Tree_node:
    """
    Data structure for nodes in the decision-tree
    """
    def __init__(self,):
        self.feature = None  # index of the selected feature (for non-leaf node)
        self.label = None  # class label (for leaf node), if not leaf node, label will be None
        self.left_child = None  # left child node
        self.right_child = None  # right child node


class Decision_tree:
    """
    Decision tree with binary features
    """
    def __init__(self,min_entropy, metric='entropy'):
        self.metricname = metric
        self.min_entropy = min_entropy
        self.root = None

    def fit(self,train_x,train_y):
        # construct the decision-tree with recursion
        self.root = self.generate_tree(train_x,train_y)

    def predict(self,test_x):
        # iterate through all samples
        prediction = []
        for i in range(len(test_x)):
            cur_data = test_x[i]
            # traverse the decision-tree based on the features of the current sample
            cur_node = self.root
            while True:
                if cur_node.label != None:
                    break
                elif cur_node.feature == None:
                    print("You haven't selected the feature yet")
                    exit()
                else:
                    if cur_data[cur_node.feature] == 0:
                        cur_node = cur_node.left_child
                    else:
                        cur_node = cur_node.right_child
            prediction.append(cur_node.label)

```

```
        prediction = np.array(prediction)

        return prediction

    # -------------------------------------------------- You are going to implement this function ------------------------------------
    # use recursion to build up the tree. Starting from the root node, you can call itself to determine what is the left_child and what is the right_child
    # return: cur_node: the current tree node you create (Type: Tree_Node)
    def generate_tree(self,data,label):
        # initialize the current tree node
        cur_node = Tree_node()

        # compute the node entropy or gini index and determine if the current node is a leaf node
        # Specifically, if entropy/gini_index (ie, self.metric(label)) < min_entropy,
        # determine what will be the label (by choosing the label with the largest count) for this leaf node
        # and directly return the leaf node
        node_entropy = self.metric(label)
        if node_entropy < self.min_entropy:
            # ------------------ Add your lines here -------------------
            cur_node.label = np.argmax( np.bincount(label) )

            return cur_node

        # select the feature that will best split the current non-leaf node
        # assign the feature index to cur_node.feature
        # ------------- Add your line here -----------------------------
        selected_feature = self.select_feature( data, label )
        cur_node.feature = selected_feature

        # split the data based on the selected feature
        # if the selected feature of the data equals to 0, assign the data and corresponding point to left_x, left,y
        # otherwise assinged to right_x, right_y
        select_x = data[:, selected_feature]
        left_x = data[select_x==0,:]
        left_y = label[select_x==0,]
        right_x = data[select_x==1,:]
        right_y = label[select_x==1,]

        # determine cur_node.left_child and cur_node.right_child by call itself, with left_x, left_y and right_x, right_y
        # ------------- Add your line here -----------------------------
        cur_node.left_child = self.generate_tree( left_x, left_y)
        cur_node.right_child = self.generate_tree( right_x, right_y)

        return cur_node

    # select the feature that maximize the information gains
    # return: best_feat, which is the index of the feature
    def select_feature(self,data,label):
```

```
        best_feat = 0
        min_entropy = float('inf')

        # iterate through all features and compute their corresponding entropy
        for i in range(len(data[0])):
            # split data based on i-th feature
            split_x = data[:,i]
            left_y = label[split_x==0,]
            right_y = label[split_x==1,]

            # compute the combined entropy which weightedly combine the entropy/gini of left_y and right_y
            cur_entropy = self.combined_metric(left_y,right_y)

            # select the feature with minimum entropy (set best_feat)
            if cur_entropy < min_entropy:
                min_entropy = cur_entropy
                best_feat = i

        return best_feat

    # ---------------------------------------------------------- You are going to implement this function ------------------------------------
    # weightedly combine the entropy/gini of left_y and right_y
    # the weights are [len(left_y)/(len(left_y)+len(right_y)), len(right_y)/(len(left_y)+len(right_y))]
    # return: result
    def combined_metric(self,left_y,right_y):
        # compute the entropy of a potential split
        result = 0
        left = len(left_y)/(len(left_y)+len(right_y))
        right = len(right_y)/(len(left_y)+len(right_y))
        result = left * self.metric(left_y) + right * self.metric( right_y )

        return result

    # ---------------------------------------------------------- You are going to implement this function ------------------------------------
    # compute entropy/gini_index based on the labels
    # entropy = -sum_i p_i*log2(p_i+1e-15) (add 1e-15 inside the log when computing the entropy to prevent numerical issue)
    # gini_index = 1 - sum_i p_i^2
    def metric(self, label):
        result = 0

        if self.metricname == 'entropy':
            class_label, count = np.unique(label,return_counts=True)
            count = count/len(label)
            result = -np.sum( count * np.log2( count + 1e-15 ))


        elif self.metricname == 'gini_index':
            class_label, count = np.unique(label,return_counts=True)
```

```python
            count = count/len(label)
            result = 1 - np.sum( count**2 )

        return result

```