Lab 2
Decision and Regression Trees

Description

The programs for this lab assignment are designed to implemented two types of predictor models for a decision tree: one as a classification model and the other as a regression model. In this paper, I present my proposed modifications to the program for both predictor models. Parts 1-3 of this assignment pertain to the *decision_tree* Python script. So, Parts 4-5 of this assignment are implemented in the *regression_tree* Python script.

Decision Tree Classifier

Part 1

The first problem was to modify the existing code to be a recursive call to the left and right children of each tree node. In this way, we will build a decision tree classifier instead of a single-node tree, that would provide one of two classifications automatically, as was provided in the original code. Therefore, only one small change was made in the return statement (see below).

Original return statement in _id3(3):

```
left = int(round(np.mean(y[less])))
right = int(round(np.mean(y[more])))
return DecisionTreeNode(best_att, thr[best_att],left,right)
```

Modified return statement in _id3(3):

```
lx = x[less]; ly = y[less]
rx = x[more]; ry = y[more]
return DecisionTreeNode(best_att, thr[best_att],
        self._id3(lx,ly,depth+1), self._id3(rx,ry,depth+1))
```

Part 2

The second task was to generate random values for each attribute to find the best threshold for them. Previously, we used the means of attributes (see $\_threshold(3)$). In order to generate these random values, I wrote the following code into a $\_threshold3(3)$ function to generate random values between the range of the lowest and highest values of each of the attributes in the training data.

```
thr = []
VALS = 10
for i in range(x.shape[1]):
    thr.append(np.random.uniform(min(x[:, i]),max(x[:, i]), size=(VALS)))
```

In order to be used and traversed as a NumPy array, I convert this list before accessing and using it for Part 3.

Part 3

The third (and final task for the *decision_tree* classification model) is to find the attribute and threshold combination that yields the highest information gain. Previously (before Part 2), we used the means of attributes. My new Python function $\_threshold3(3)$ is provided below, including the code for Part 2.

In the nested for loop section of this method is where the program will find the threshold for the attribute which will produce the highest information gain.

```
def _threshold3(self, x, y, orig_entropy):
    # code from Part 2 goes here
    thr = np.asarray(thr)
    entropy_attribute = np.zeros(len(thr))

    thresh = []
    for i in range(x.shape[1]):
        m = sys.maxsize; ind = 0
        for j in range(len(thr[i])):
            less = x[:, i] <= thr[i][j]
            more = ~ less
            new = self._entropy(y[less], y[more])
```

```
            if m>new:
                m = new
                ind = j
        thresh.append(thr[i][ind])
        entropy_attribute[i] = m
    gain = orig_entropy - entropy_attribute
    return np.asarray(thresh), np.argmax(gain)
```

Regression Tree

Part 4

The fourth task was to modify the existing code to be a recursive call to the left and right children of each tree node in the regression model. In this way, we will build a regression tree instead of a single-node tree, as was provided in the original code. Therefore, only one small change was made in the return statement (see below).

Original return statement in $\_id3(3)$:

```
left = int(round(np.mean(y[less])))
right = int(round(np.mean(y[more])))
return RegressionTreeNode(best_att, thr[best_att],left ,right)
```

Modified return statement in $\_id3(3)$:

```
lx = x[less]; ly = y[less]
rx = x[more]; ry = y[more]
return RegressionTreeNode(best_att, thr[best_att],
        self._id3(lx,ly,depth+1), self._id3(rx,ry,depth+1))
```

Much of this code is mirrored in Part 1 of this lab. The only real difference is to call the $RegressionTreeNode$ class rather than the $DecisionTreeNode$ class.

Part 5

For the last and final part of this assignment, we were tasked with generating several, random threshold values, and finding the one that would yield the lowest MSE. Therefore, I implemented a similar algorithm to the one in Part 3 in the $\_threshold5(3)$ Python function, provided below.

```
def _threshold5(self,x,y,orig_mse):
    thr = []

    VALS = 20
    for i in range(x.shape[1]):
        thr.append(np.random.uniform(min(x[:,i]),max(x[:,i]),size=(VALS)))
    thr = np.asarray(thr)
    mse_attribute = np.zeros(len(thr))

    thresh = []
    for i in range(x.shape[1]):
        m = sys.maxsize; ind = 0
        for j in range(len(thr)):
            less = x[:, i] <= thr[i][j]
            more = ~ less
            new = self._mse(y[less], y[more])
            if new<m:
                m = new; ind = j
        mse_attribute[i] = m
        thresh.append(thr[i][ind])
    gain = orig_mse - mse_attribute
    return np.asarray(thresh), np.argmax(gain)
```

Results

For the classification decision tree, in Parts 1-3, the accuracies are measured by correctly predicted samples. For Parts 4-5, the accuracies are presented as mean squared error (MSE). All times are reported in seconds. For Parts 1-3, I ran the tests over the entire gamma ray detection dataset. For Parts 4-5, I ran the tests over the entire solar particle dataset. For each run in both models, I randomly partitioned a section of the entire dataset to be used for training or testing the model. My results are presented as the average of 100 experiments.

For Part 1, the accuracy of the tree has improved in training and testing, see Table 1; time, however, had stayed about the same.

	Training Time	Training Accuracy	Testing Time	Testing Accuracy
Original	0.038	0.735	0.005	0.74
Modified	0.654	0.881	0.0243	0.842

Table 1: Part 1 Results

For Part 2, there were no experiments run; only runs were to debug the implemented code. Results will be tested for 10 and 20 randomly generated threshold values. This same code is used for Part 5, also, and results are presented for 10, 20, and 50 randomly generated threshold values, also. See Table 2 for results in Part 3 and Table 4 for results in Part 5.

For Part 3, which includes the code from Part 2 (shown here for 10, 20, and 50 values), I did not find a decline in accuracy (as expected). However, I did not find a significant increase in accuracy either (about 0.7% increase from Part 1). Time, however, did increase greatly, which makes sense, since my implemented algorithm runs in $O(n^2)$.

	Training		Testing	
	Time	Accuracy	Time	Accuracy
10 Threshold Values	3.819	0.888	0.09	0.849
20 Threshold Values	7.165	0.892	0.89	0.849
50 Threshold Values	16.098	0.895	0.91	0.849

Table 2: Part 3 Results

For Part 4, as expected, the time increased because we went from a one-node regression tree to a multi-node tree. The accuracy, of course, also increased (lower MSE means higher accuracy). See Table 3 below for results.

	Training Time	Training Accuracy	Testing Time	Testing Accuracy
Original	0.029	0.0869	0.0065	0.0364
Modified	4.387	0.0021	0.0504	0.0033

Table 3: Part 4 Results

For Part 5, some precular results are shown in the average time. It is expected for the time to increase when calculating MSE for more threshold values for each attribute. This might be explained by my computer's running extra processes at the time of these experiments. The accuracy, though already low, did increase (as shown in Table 4).

	Training		Testing	
	Time	Accuracy	Time	Accuracy
10 Threshold Values	3.234	0.00272	0.149	0.00259
20 Threshold Values	2.73	0.00253	0.648	0.0029
50 Threshold Values	2.86	0.00251	0.067	0.00285

Table 4: Part 5 Results

Conclusions and Discussion

For Part 1, as expected, the time to train and test increased (since we were building a tree which contained more than the one root node). Further, the accuracy improved by about 10%. Since I ran the tests over the entire gamma ray detection dataset. Each time, I randomly partitioned a section of the entire dataset to be used for training or testing the model to simulate more randomness in my testing and training data.

For Part 3, I expected my accuracy to improve, though perhaps not significantly. As shown in my results, the accuracy of the classification model improved by under 1%.

For Parts 4-5, I was happy with the results of the significant accuracy increase for the solar particle dataset, since it was expected. However, I'd have expected the time to increase with the number of randomly generated threshold values we used for the attributes. My results show that something happened to make the tests with 10 take longer than those with 20 threshold values to check for each attribute. I attribute this abnormality to unstable computer resource management.

Compared to my knn program, the accuracies in my decision tree models are not as good for classification but are better in terms of regression. Regarding time, the use of the decision trees are definitely less time-consuming. This is expected because trees allow us to use branches to come to predictions rather than calculating multiple distances and sorting each of these calculations for each new testing sample. Of course, my implementation of knn was not very optimized (it could have been improved). However, still comparing the run times for both algorithms (knn vs. decision trees), still decision trees win by a lot. This time would also add up when we use larger datasets.

In terms of accuracy, it seems that using either 10, 20 or 50 threshold values did not change accuracy much. There was no change for testing accuracy in the classification model. The training accuracy, however, was increased, but not by much. It might, however, be preferable to use more threshold values, but it would depend on the resources available to the program's environment. For instance, if using the classifier, we can reduce the number of threshold values to get approximately the same amount of testing accuracy, but we would be gaining time (cutting it significantly). For the regression model, the time stayed relatively the same despite the number of threshold values generated. Similarly, the accuracy (as MSE), did improve, but only in small increments, which decreased as I increased the number of threshold values.

I would like to see how *pruning* would affect the results, especially timewise. If we could prune the tree so that branches can make guesses quicker, then we can not only reduce time but also reduce the workload of the decision tree. In this way, the model won't have learned extra detail and will therefore be more generalizable but still as accurate.

Appendix

The appendix includes the source code of the described program implemented in Python 3.7. Methods/Files authored by me are denoted with an author tag ($@author\ mmafr$). The code is also available at https://github.com/mahdafr/19w_cs5361-labs on GitHub.

```python
# Program to build a one-node decision tree
# Programmed by Olac Fuentes
# Last modified September 16, 2019
import sys

import numpy as np
import time


class DecisionTreeNode(object):
    # Constructor
    def __init__(self, att, thr, left, right):
        self.attribute = att
        self.threshold = thr
        # left and right are either binary classifications or references to
        # decision tree nodes
        self.left = left
        self.right = right

    def print_tree(self, indent=''):
        # If prints the right subtree, corresponding to the condition x[attribute] > threshold
        # above the condition stored in the node
        if self.right in [0, 1]:
            print(indent + '       ', 'class=', self.right)
        else:
            self.right.print_tree(indent + '    ')

        print(indent, 'if x[' + str(self.attribute) + '] <=', self.threshold)

        if self.left in [0, 1]:
            print(indent + '       ', 'class=', self.left)
        else:
            self.left.print_tree(indent + '    ')


class DecisionTreeClassifier(object):
    # Constructor
    def __init__(self, max_depth=10, min_samples_split=10, min_accuracy=1):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_accuracy = min_accuracy

    def fit(self, x, y):
        self.root = self._id3(x, y, depth=0)

    def predict(self, x_test):
        pred = np.zeros(len(x_test), dtype=int)
        for i in range(len(x_test)):
            pred[i] = self._classify(self.root, x_test[i])
        return pred

    def _id3(self, x, y, depth):
        orig_entropy = self._entropy(y, [])
        mean_val = np.mean(y)

        # if accuracy not attained and cannot go further in tree
        if depth >= self.max_depth or len(y) <= self.min_samples_split or max(
                [mean_val, 1 - mean_val]) >= self.min_accuracy:
            return int(round(mean_val))

        # @author mahdafr for part2-part3
        thr, best_att = self._threshold3(x, y, orig_entropy)

        less = x[:, best_att] <= thr[best_att]
        more = ~ less
```

```python
        # @author mahdafr for part1
        lx = x[less]; ly = y[less]  # int(round(np.mean(y[less])))
        rx = x[more]; ry = y[more]  # int(round(np.mean(y[more])))
        return DecisionTreeNode(best_att, thr[best_att], self._id3(lx,ly,depth+1), self._id3(rx,ry,depth+1))

    # original code
    def _threshold(self, x, y, orig_entropy):
        thr = np.mean(x, axis=0)
        entropy_attribute = np.zeros(len(thr))

        # foreach training example, find entropy for each attribute
        for i in range(x.shape[1]):
            less = x[:, i] <= thr[i]
            more = ~ less
            entropy_attribute[i] = self._entropy(y[less], y[more])
        gain = orig_entropy - entropy_attribute
        # print('Gain:',gain)
        return thr, np.argmax(gain)

    # @author mahdafr for part3
    def _threshold3(self, x, y, orig_entropy):
        thr = []    # np.mean(x, axis=1)

        # @author mahdafr for part2
        # generate random values for each attribute
        VALS = 20
        for i in range(x.shape[1]):
            thr.append(np.random.uniform(min(x[:,i]),max(x[:,i]),size=(VALS)))
        thr = np.asarray(thr)
        entropy_attribute = np.zeros(len(thr))

        thresh = []
        # find entropy
        for i in range(x.shape[1]):
            m = sys.maxsize; ind = 0
            for j in range(len(thr[i])):
                less = x[:, i] <= thr[i][j]
                more = ~ less
                new = self._entropy(y[less], y[more])
                if new<m:
                    m = new
                    ind = j
            thresh.append(thr[i][ind])
            entropy_attribute[i] = m
        gain = orig_entropy - entropy_attribute
        # print('Gain:',gain)
        return np.asarray(thresh), np.argmax(gain)

    def _entropy(self, l, m):
        ent = 0
        for p in [l, m]:
            if len(p) > 0:
                pp = sum(p) / len(p)
                pn = 1 - pp
                if pp < 1 and pp > 0:
                    ent -= len(p) * (pp * np.log2(pp) + pn * np.log2(pn))
        ent = ent / (len(l) + len(m))
        return ent

    def _classify(self, dt_node, x):
        if dt_node in [0, 1]:
            return dt_node
        if x[dt_node.attribute] <= dt_node.threshold:
            return self._classify(dt_node.left, x)
        else:
```

```
                return self._classify(dt_node.right, x)

    def display(self):
        print('Model:')
        self.root.print_tree()


x = []; y = []
infile = open("magic04.txt", "r")
for line in infile:
    y.append(int(line[-2:-1] == 'g'))
    x.append(np.fromstring(line[:-2], dtype=float, sep=','))
infile.close()

xa = np.zeros((len(x), len(x[0])))
for i in range(len(xa)):
    xa[i] = x[i]
x = xa

x = np.array(x).astype(np.float32)
y = np.array(y)

TESTS = 100
test_acc = 0; train_acc = 0
test_time = 0; train_time = 0
print('Tests  ' + str(TESTS))
for i in range(TESTS):
    # Split data into training and testing
    ind = np.random.permutation(len(y))
    split_ind = int(len(y) * 0.8)
    x_train = x[ind[:split_ind]]
    x_test = x[ind[split_ind:]]
    y_train = y[ind[:split_ind]]
    y_test = y[ind[split_ind:]]

    model = DecisionTreeClassifier()
    start = time.time()
    model.fit(x_train, y_train)
    train_time += time.time() - start

    train_pred = model.predict(x_train)
    start = time.time()
    test_pred = model.predict(x_test)
    test_time += time.time() - start

    train_acc += np.sum(train_pred == y_train) / len(train_pred)
    test_acc += np.sum(test_pred == y_test) / len(test_pred)
    # model.display()

print('Elapsed_time training  {0:.6f} '.format(train_time/TESTS))
print('Elapsed_time testing  {0:.6f} '.format(test_time/TESTS))
print('train accuracy:', train_acc/TESTS)
print('test accuracy:', test_acc/TESTS)
```

```python
# Program to build a one-node regression tree
# Programmed by Olac Fuentes
# Last modified September 16, 2019
import sys

import numpy as np
import time


class RegressionTreeNode(object):
    # Constructor
    def __init__(self, att, thr, left, right):
        self.attribute = att
        self.threshold = thr
        # left and right are either binary classifications or references to
        # decision tree nodes
        self.left = left
        self.right = right

    def print_tree(self, indent=''):
        # If prints the right subtree, corresponding to the condition x[attribute] > threshold
        # above the condition stored in the node
        if isinstance(self.right, np.float32):
            print(indent + '        ', 'pred=', self.right)
        else:
            self.right.print_tree(indent + '    ')

        print(indent, 'if x[' + str(self.attribute) + '] <=', self.threshold)

        if isinstance(self.left, np.float32):
            print(indent + '        ', 'pred=', self.left)
        else:
            self.left.print_tree(indent + '    ')


class DecisionTreeRegressor(object):
    # Constructor
    def __init__(self, max_depth=10, min_samples_split=5, max_mse=0.001):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.max_mse = max_mse

    def fit(self, x, y):
        self.root = self._id3(x, y, depth=0)

    def predict(self, x_test):
        pred = np.zeros(len(x_test), dtype=np.float32)
        for i in range(len(x_test)):
            pred[i] = self._predict(self.root, x_test[i])
        return pred

    def _id3(self, x, y, depth):
        orig_mse = np.var(y)
        # print('original mse:',orig_mse)
        mean_val = np.mean(y)
        if depth >= self.max_depth or len(y) <= self.min_samples_split or orig_mse <= self.max_mse:
            return mean_val

        # @author mahdafr part5
        thr, best_att = self._threshold(x,y,orig_mse)

        # print('mse best attribute:',mse_attribute[best_att])
        less = x[:, best_att] <= thr[best_att]
        more = ~ less
```

```
        # print('subtree mse:',np.var(y[less]),np.var(y[more]))
        # @author mahdafr for part4
        lx = x[less]; ly = y[less]
        rx = x[more]; ry = y[more]
        return RegressionTreeNode(best_att, thr[best_att], self._id3(lx, ly, depth + 1), self._id3(rx, ry, depth + 1))
        # return RegressionTreeNode(best_att, thr[best_att], np.mean(y[less]), np.mean(y[more]))

    # original code
    def _threshold(self, x, y, orig_mse):
        thr = np.mean(x, axis=0)
        mse_attribute = np.zeros(len(thr))
        for i in range(x.shape[1]):
            less = x[:, i] <= thr[i]
            more = ~ less
            mse_attribute[i] = self._mse(y[less], y[more])
        gain = orig_mse - mse_attribute
        # print('Gain:',gain)
        return thr, np.argmax(gain)

    # @author mahdafr for part5
    def _threshold5(self,x,y,orig_mse):
        thr = []    # np.mean(x, axis=1)

        # @author mahdafr for part2
        # generate random values for each attribute
        VALS = 20
        for i in range(x.shape[1]):
            thr.append(np.random.uniform(min(x[:,i]),max(x[:,i]),size=(VALS)))
        thr = np.asarray(thr)
        mse_attribute = np.zeros(len(thr))

        thresh = []
        for i in range(x.shape[1]):
            m = sys.maxsize; ind = 0
            for j in range(len(thr)):
                less = x[:, i] <= thr[i][j]
                more = ~ less
                new = self._mse(y[less], y[more])
                if new<m:
                    m = new; ind = j
            mse_attribute[i] = m
            thresh.append(thr[i][ind])
        gain = orig_mse - mse_attribute
        # print('Gain:',gain)
        return np.asarray(thresh), np.argmax(gain)

    def _mse(self, l, m):
        err = np.append(l - np.mean(l), m - np.mean(m))  # It will issue a warning if either l or m is empty
        return np.mean(err * err)

    def _predict(self, dt_node, x):
        if isinstance(dt_node, np.float32):
            return dt_node
        if x[dt_node.attribute] <= dt_node.threshold:
            return self._predict(dt_node.left, x)
        else:
            return self._predict(dt_node.right, x)

    def display(self):
        print('Model:')
        self.root.print_tree()
```

```python
TESTS = 100
test_acc = 0; train_acc = 0
test_time = 0; train_time = 0
print('Tests  ' + str(TESTS))
dir = 'D:\Google Drive\skool\CS 5361\datasets\lab1\\'
for i in range(TESTS):
    skip = np.random.randint(40,50)
    x_train = np.load(dir + 'x_ray_data_train.npy')[::skip]
    y_train = np.load(dir + 'x_ray_target_train.npy')[::skip]
    x_test = np.load(dir + 'x_ray_data_test.npy')[::skip]
    y_test = np.load(dir + 'x_ray_target_test.npy')[::skip]

    model = DecisionTreeRegressor()
    start = time.time()
    model.fit(x_train, y_train)
    train_time += time.time() - start
    pred = model.predict(x_train)
    train_acc += np.mean(np.square(pred - y_train))

    start = time.time()
    pred = model.predict(x_test)
    test_time += time.time() - start
    test_acc += np.mean(np.square(pred - y_test))
    # model.display()

print('Elapsed_time training  {0:.6f} '.format(train_time/TESTS))
print('Elapsed_time testing  {0:.6f} '.format(test_time/TESTS))
print('Mean square error training set:', train_acc/TESTS)
print('Mean square error test set:', test_acc/TESTS)
```