ISYE 6740 Homework 5

Submitted by: Arjun Mishra (amishra85)

Q.1. (A)

Data given to us contains 207 samples and 61 variables including our response. The decision tree that I have built to classify the response for this dataset uses a recursive algorithm. A purely random decision tree algorithm would pick a random feature from the data, then a random value of that feature and finally split the data based on this value. It would repeat this procedure until the dataset is completely split into decision boundaries around these "nodes". A node is simple represented by the feature and the value of the feature at a split. A mode which is not split any further is called a leaf. Finally, when the tree is completely split, we will input new data to this tree. Each sample from the new data would "trickle" down the tree and fit into a leaf, where we would predict its value based on the majority response in that leaf.

As we can imagine, a random tree would suffer from overfitting and poor predictions based on its completely random nature.

My recursive tree has improvements over a completely random tree. First, I have decided the splits of the nodes based on the Gini index. Gini evaluates various splits on several features and further several values for that feature. The split that gives us a homogenous split of the response is the one that is picked. This also ensures that we pick more number of features and values compared to a random tree. The pseudo code for my Gini implementation is:

```
gini_index(data, response, no_of_features, no_of_feat_val):

# Pick out random features from the dataset
# The number of random features we pick out for testing can be modified
using the parameter no_of_features

    indices of random features

# Now we will pick out random values of each feature – the number of
random values to check acan also be modified using the no_of_feat_val
argument
```

```
   Random values of each feature
# Finally the Gini is implemented in a double for loop

For each random feature:
       For each random value:
              Calculate Gini
              Store Gini

# From all the stored values of Gini, we will find the minimum one and
return the corresponding index and value for split

    Find (minimum of Gini)
    return (Feature index and value corresponding to minimum Gini)
```

I have also included a condition on the minimum number of samples required in a node for a further split. If the number of samples drops to a minimum threshold of, for example say 10, that node will automatically become a leaf and no further splitting will take place. This will ensure that the tree does not go down to the level of fitting each and every observation in a separate leaf. Thus, we avoid overfitting to a certain extent.

The tree that we get is a dictionary of dictionaries. The final prediction is again made based on the conditions satisfied by the new sample at each split. For example, suppose our first node is feature 24 with a split value of of 0.456. If the value of feature 24 in our new sample row is less than 0.456 we would go to right split. This right split will further have a feature number like 55 and a split value. We will keep on checking these conditions until we reach a leaf and finally make the prediction for that sample.

My tree looks like:

```
In[118]: tree_output
Out[118]:
defaultdict(collections.defaultdict,
            {'left_prediction': defaultdict(None, {}),
             'node_parent_feat_ind': 16,
             'node_parent_feat_val': 0.062399999999999997,
             'records in right leaf': 2,
             'right_prediction': 0,
             'sub_tree_left': defaultdict(collections.defaultdict,
                         {'left_prediction': defaultdict(None, {}),
                          'node_parent_feat_ind': 24,
                          'node_parent_feat_val': 0.074700000000000003,
                          'records in right leaf': 3,
                          'right_prediction': 0,
                          'sub_tree_left': defaultdict(collections.defaultdict,
                                      {'left_prediction': defaultdict(None,
                                                  {}),
                                       'node_parent_feat_ind': 51,
                                       'node_parent_feat_val': 0.0012999999999999999,
                                       'records in right leaf': 1,
                                       'right_prediction': 0,
                                       'sub_tree_left': defaultdict(collections.defaultdict,
```

The pseudo code for my tree implementation is:

```
create_tree(data, no_of_features, no_of_feat_val, leaf_min_rec, gini_index):

# Create an empty dictionary of dictionaries where tree will be stored
    tree_store

# Get the node index and value based on the Gini index
    index, value = gini_index(data, response, no_of_features, no_of_feat_val)

# Split the data based on the index and value we got from gini
    left_data, right_data

# Now, we check for conditions for deciding whether we have reached a leaf or not
# If we are at a leaf, we give out predictions.
# If not, we create another split on the data by recursively running the function
again
    if left_data.size < leaf_min_rec:
        # Make a leaf and give prediction
        # The prediction is the mode of the response of the data in our leaf

        Return Prediction = mean(response(left_data))

    if right_data.size < leaf_min_rec:
        # Make a leaf and give prediction
        # The prediction is the mode of the response of the data in our leaf

        Return Prediction = mean(response(right_data))

    if left_data.size > leaf_min_rec:
        # Make another split just on left_data now by running the function again
        # The value returned will come from running the tree again
```

```
        Return create_tree(left_data)

    if right_data.size > leaf_min_rec:
        # Make another split just on right_data now by running the function again
        # The value returned will come from running the tree again

        Return create_tree(left_data)


# Finally, we store all the split indexes, split values, predictions in the
dictionary of
# dictionaries and return this dictionary

    return tree_store
```

(B)

I evaluated the performance of my model using 10 fold cross-validation. Each 10 folds were run for a total of 100 times. The mean accuracy and the standard deviation for these runs are:

Mean accuracy = 61.6% with a standard deviation = 0.185%

```
Done with iteration: 97. Average accuracy score for iteration: 0.6171161321671526
Done with iteration: 98. Average accuracy score for iteration: 0.616976911976912
Done with iteration: 99. Average accuracy score for iteration: 0.6167095238095238
The mean of the accuracies is: 0.616229314388
The standard deviation of the accuracies is: 0.00184611636457
```

(C)

The accuracy that I got with my first tree was not spectacular. I also checked by fitting the whole data on the model after training the mode with the same data. I got an accuracy of 100%. This means that the model is overfitting. After looking at some of the leaves in the model, I saw that the leaves usually had 3-5 data points in them. This shows that the model is overfitting. Also, picking out random values for each feature is not the best or most optimized alternative.

To improve my decision tree, I have implemented two changes in my Gini function:

1. Tree pruning: To avoid overfitting, I will put a condition on the minimum number of data points that can be present in a leaf. This way, a leaf will not have

any lesser than "N" number of points in it. With this condition, I ensure that the decision boundaries are more generic and not fitting each and every point separately.

2. Feature value selection: Instead of selecting values randomly for each feature, we will pick values evenly spaced between the minima and the maxima of each feature we select. This improvement should result in better generalization of the decision boundaries and thus better prediction. Also, this will reduce the need to select and fit many random values as it will cover the whole range of values of the feature selected. This will result in an improvement in performance (time to run the code for the tree) as we will need to select less number of values.

In addition to the improvements, I also tried running the tree for different values of the minimum leaf requirement. Out of 10, 20 and 50, I got 20 as the best size for minimum number of samples in the leaf.

Running the code with these improvements, a 100 iteration 10-fold cross validation gave us the following results:

Mean accuracy = 62.4% with a standard deviation = 0.732%

```
Done with iteration: 97. Average accuracy score for iteration: 0.6307604470359572
Done with iteration: 98. Average accuracy score for iteration: 0.6301899951899952
Done with iteration: 99. Average accuracy score for iteration: 0.6299333333333332
The mean of the accuracies is: 0.624201479
The standard deviation of the accuracies is: 0.00732539608645
```

This is a slight improvement over the previous tree. The time to run this also reduced by ~20%.

Q.2. (A)

The AdaBoost algorithm that I have implemented is based on Freund & Schapire's description of how to use an ensemble of weak learners to create a strong one. In place of the weak learners in my implementation, I have used the same decision trees implemented in the previous section but with the slight change of making the random selection of only one feature in my Gini Index.
My algorithm takes three inputs: the weak learner model, the data and the number of weak learners we want to train. The Adaboost algorithm works based on calculating a set of weights and updating those weights after building one

weak learner so that the next tree is much better at prediction. Thus, it makes incremental and subsequent improvements in the weak learners.
First we calculate an error which is basically the mis-classification of the weak learner. Based on this error, we calculate alpha which is the ln(1-error/error). This is the weight that we give to each and every weak learner's predictions at the end. Adaboost makes predictions with each and every weak learner. These weak learners have a weight value associated with them. This is the alpha value. The final predictions based on the Adaboost model are a weighted average of the predictions from each weak learner.
In the algorithm, once we have the alpha after making a weak learner, we define sample weights by the formula: previous_weights * exponential(mis-classification * alpha). These weights are used to obtain the samples that we get in the next weak learner. This means that the samples that were mis-classified in the previous weak learner, will have a lower chance of being mis-classified in the next. I have done this by sampling my dataset with replacement, by giving a higher probability of being sampled to the mis-classified samples.

The pseudo code is as follows:

```
adaboost(weak_learner, data, no_of_weak_learners):

# Initializing weights
    weights = (1/no. of samples in data)

# These weights are used to arrive at sampling probabilities
# Initially the sampling probabilities are the same as the weights

sampling_probabilities = weights

# To calculate probabilistic weights from actual the actual weights, we need a
function that converts numbers to probs
# We use the softmax for this
    softmax(x) = exp(x)/sum of all (exp(x))

# Next we run a loop for the number of weak learner:
    for weak learner in range(number of weak learners):

        # Use probabilities to randomly sample data with replacement
        mask = indices to be sampled based on probabilities

        # Sample data based on the mask
```

```
        adjusted_data = data[mask]

        # Define the weak learner based on the new data

        weak_learner(adjusted_data)

        # Make predictions based on the weak learner

        Predictions

        # Calculate mis-classification

        Actual response - Prediction

        #Calculate Error metric

        error = sum(mis-classification * weights)/sum(weights)

        # Calculate alpha

        alpha = log((1 - error)/error)

        # Recompute the weights and probabilistic weights
        weights = weights * exponential(alpha * mis-classification)

        sampling probabilities = softmax(weights)

        # Output the weak learner and alpha for the weak learner

    return list of weak learners and their alphas
```

After we get these, we use each model to make a prediction on the new data. Finally, we weight all those predictions for each weak learner by their alphas and get a final weighted prediction.

(B)

As described above, the weak learner I have used in my adaboost implementation is a random decision tree. The only difference is that while I was sampling more number of features and more number of values to check my best Gini index in my previous tree model, here I have used a maximum of 10 features to randomly sample and a maximum of 5 values to randomly sample on which the Gini will be evaluated. Based on this calculated Gini, my data will be split.

This makes my weak learner run in a much lesser amount of time. However, compared to the tree that we made in the previous question, these trees will be very poor predictors individually. They will have only considered very few features and thus would not know much information about the whole dataset. The whole

idea of adaboost is this. Each of these weak learners will be good in a particular area and subset of the data. Together they will give a much better prediction after being mapped to the whole dataset.
The number of values also has the same effect. The lower the number of values selected, faster the algorithm runs but lesser it knows about the data.
Finally, I am using 20 trees in my algorithm. This number can be changed. However, 20 was computationally doable and also not bad at prediction.
The Gini index used is the improved on from Q1B – with linear space random values and tree pruning implemented to avoid overfit.

(C)

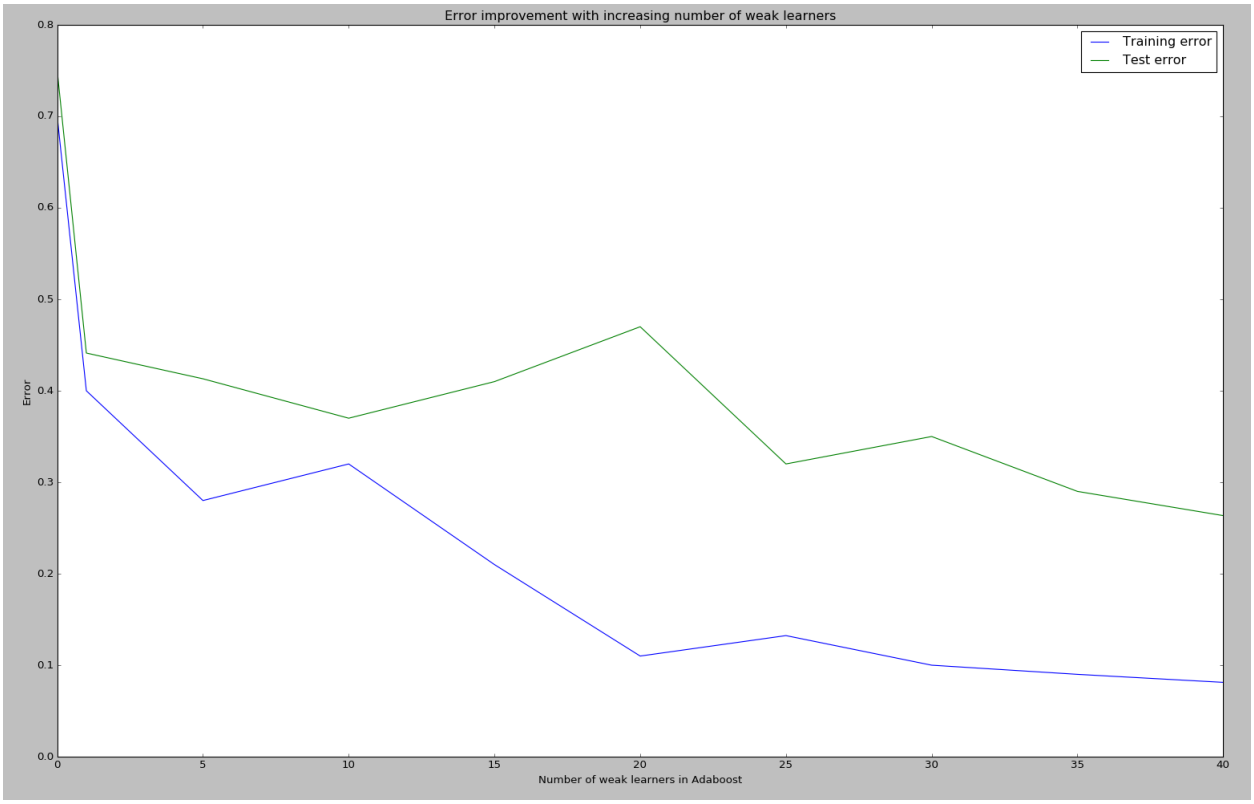
I evaluated the performance of my model using 10 fold cross-validation. Each 10 folds were run for a total of 100 times. The mean accuracy and the standard deviation for these runs are:

Mean accuracy = 64.8% with a standard deviation = 0.912%

```
Done with iteration: 99. Average accuracy score for iteration: 0.647365800865801
The mean of the accuracies is: 0.648538359789
The standard deviation of the accuracies is: 0.00912674027411
```

(D)

To look at the performance of the adaboost with a varying number of weak_learners, we plot the error graph for its performance on test and train data. I have varied the number of weak learners from 1 to 40 with an interval of 5 to keep the exercise computationally manageable and also get a good look at the general trend of how the performance varies. The test and train data was created using the standard 20/80 split.



As is apparent from the graph, there is a clear decreasing trend in the error with increasing number of weak learners. As expected, the training error decreases more than the test error. The training error approaches ~ (0, 0.1) and the test error approaches ~ (0.3, 0.35).
