AdaBoost Classifier

Team Number:

Jufeng Yang	ID: 20125011
Xingda Zhou	ID: 19107471
Zhongen Qin	ID: 19107579

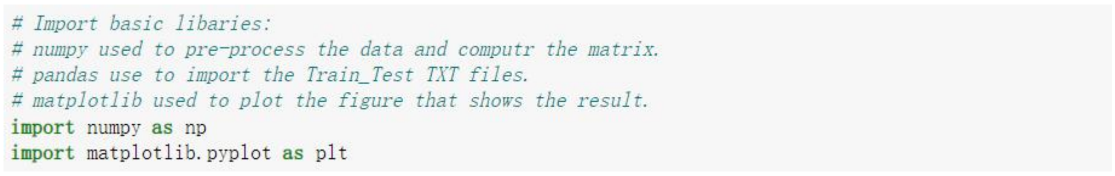
Content Table

1. Introduction

AdaBoost has a very good reputation in machine learning and has a remarkable classification power in binary classification. AdaBoost is based on the principle of forming a strong classifier by combining weak classifiers in a linear fashion, the effectiveness of the strong classifier depends on the applicability of the weak classifier to a given classification and the number of weak classifiers. In the past AdaBoost algorithm, decision tree stump was a very good weak classifier. Even in later developments, SVMs have been used as weak classifiers for non-linear classification and very good results have been obtained. However, in this project we will use a custom weak linear classifier as a weak classifier. In this weak linear classifier, the mean of the positive and negative samples is used as the initial classification base: the mean of the positive and negative samples are connected to form a line segment, the mean of the negative samples is considered as the origin, all sample points are mapped in this direction and the distance (with +/- sign) from the mapped point to the origin is calculated, the distance value will be the new classification feature value. As with the decision tree stump, the boundary values are moved over this distance to find the classification threshold with the least error. Such classification thresholds are combined linearly to form a strong classifier. This strong classifier using the mean for classification has a faster convergence rate, partly because the mean provides an overall feature of the sample that makes the misclassified sample more salient, and partly because combining two features and generating them into a single feature value is the threshold boundary for classification is no longer just a level. or linear dividing line. This weak classifier is therefore highly efficient and accurate when performing binary classification with few eigenvalues.

2. Data Procession

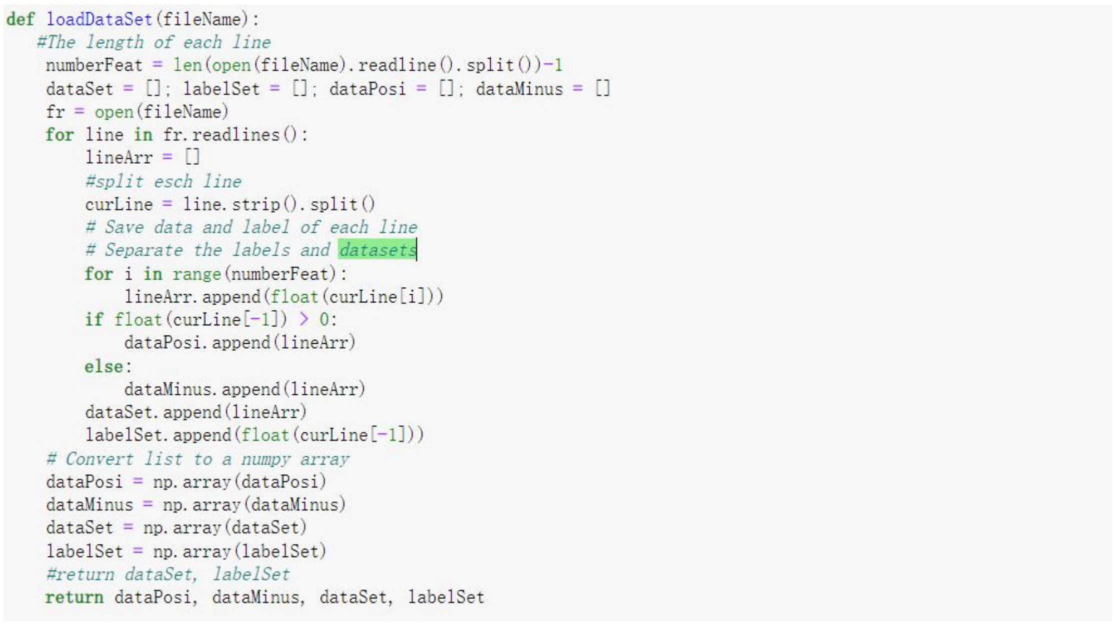
2.1 Import datasets



```python
# Import basic libaries:
# numpy used to pre-process the data and computr the matrix.
# pandas use to import the Train_Test TXT files.
# matplotlib used to plot the figure that shows the result.
import numpy as np
import matplotlib.pyplot as plt
```

Figure.1: Import libraries.

Import numpy for calculating between variables and matplotlib.pyplot for outputting images.

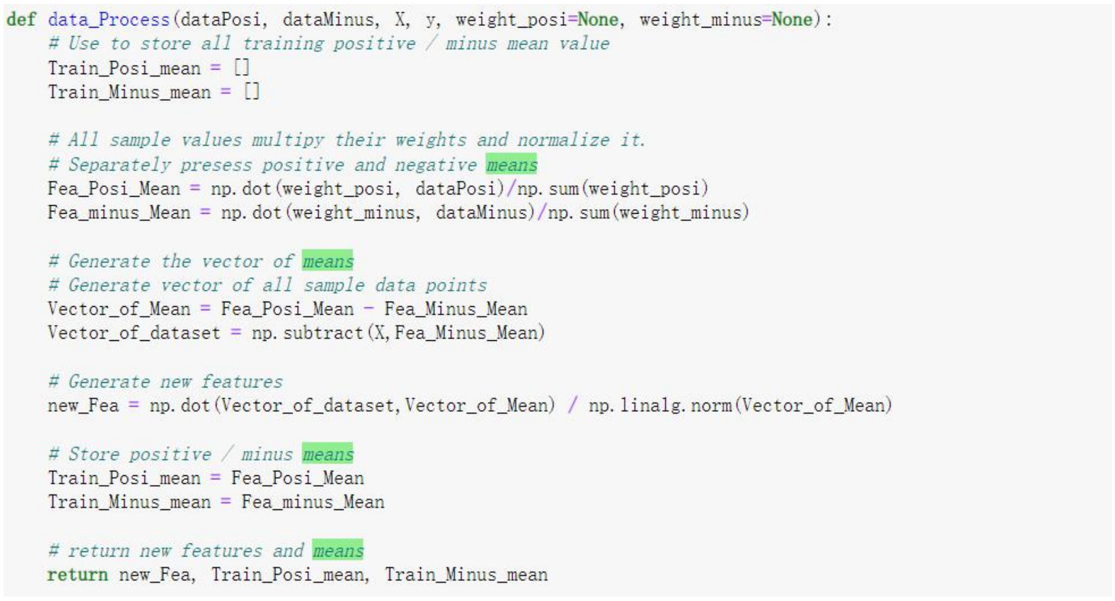


```python
def loadDataSet(fileName):
   #The length of each line
    numberFeat = len(open(fileName).readline().split())-1
    dataSet = []; labelSet = []; dataPosi = []; dataMinus = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        #split esch line
        curLine = line.strip().split()
        # Save data and label of each line
        # Separate the labels and datasets
        for i in range(numberFeat):
            lineArr.append(float(curLine[i]))
        if float(curLine[-1]) > 0:
            dataPosi.append(lineArr)
        else:
            dataMinus.append(lineArr)
        dataSet.append(lineArr)
        labelSet.append(float(curLine[-1]))
    # Convert list to a numpy array
    dataPosi = np.array(dataPosi)
    dataMinus = np.array(dataMinus)
    dataSet = np.array(dataSet)
    labelSet = np.array(labelSet)
    #return dataSet, labelSet
    return dataPosi, dataMinus, dataSet, labelSet
```

Figure.2: Load Datasets.

This function is used to separate the positive and negative samples in a dataset, separating the eigenvalues from the labelled values. The row slicing function is used to separate rows of data, using the labels as a basis for classification and separating positive and negative sample features stored in a list. In the second for loop the variable 'lineArray' is used as an intermediate variable to store the feature values in a row, and the last row in 'curline' is used to make a judgement and store the feature values in this row in the +/- sample list. Here, the variable 'lineArray' is required, and after each expansion, the list is not allowed to be emptied before the next expansion. Convert all lists to numpy arrays.

2.2 Feature Convertion



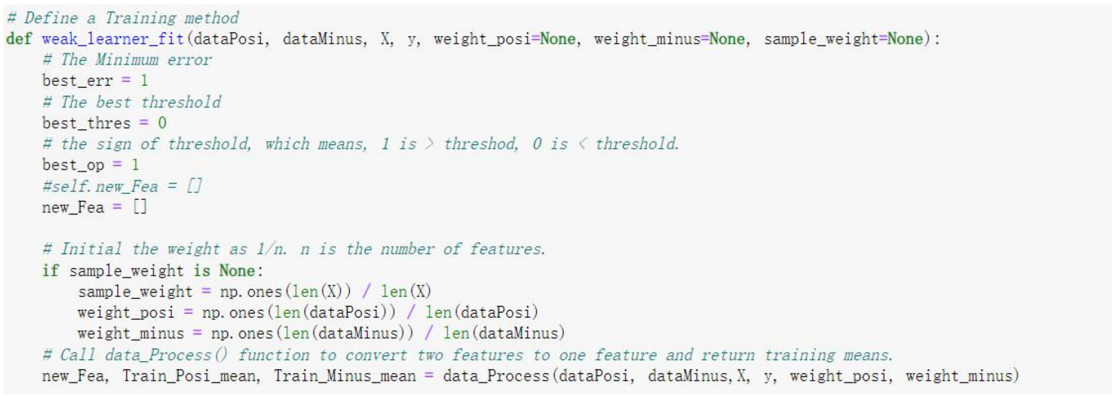
```python
def data_Process(dataPosi, dataMinus, X, y, weight_posi=None, weight_minus=None):
    # Use to store all training positive / minus mean value
    Train_Posi_mean = []
    Train_Minus_mean = []

    # All sample values multipy their weights and normalize it.
    # Separately presess positive and negative means
    Fea_Posi_Mean = np.dot(weight_posi, dataPosi)/np.sum(weight_posi)
    Fea_minus_Mean = np.dot(weight_minus, dataMinus)/np.sum(weight_minus)

    # Generate the vector of means
    # Generate vector of all sample data points
    Vector_of_Mean = Fea_Posi_Mean - Fea_Minus_Mean
    Vector_of_dataset = np.subtract(X,Fea_Minus_Mean)

    # Generate new features
    new_Fea = np.dot(Vector_of_dataset,Vector_of_Mean) / np.linalg.norm(Vector_of_Mean)

    # Store positive / minus means
    Train_Posi_mean = Fea_Posi_Mean
    Train_Minus_mean = Fea_minus_Mean

    # return new features and means
    return new_Fea, Train_Posi_mean, Train_Minus_mean
```

Figure.3: Two features mapped into one feature.

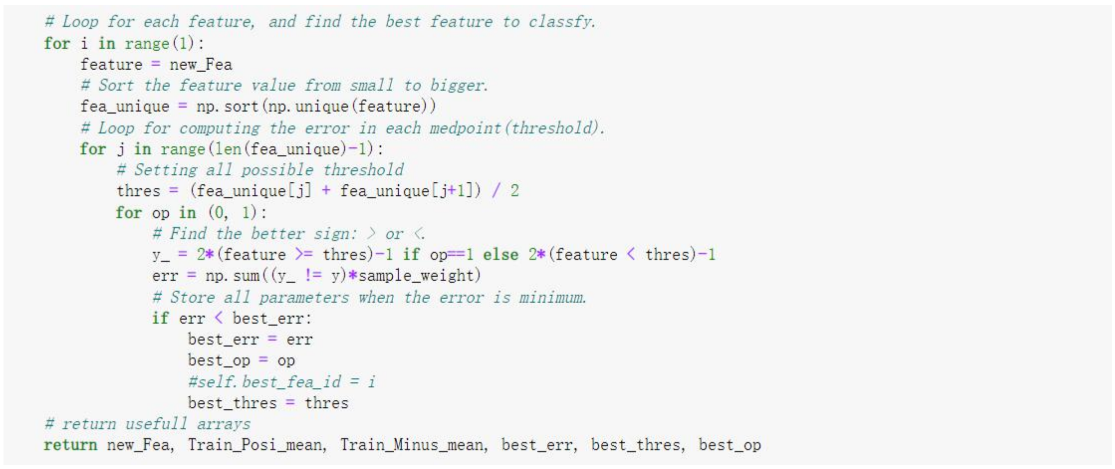
All sample values are multiplied by the corresponding weights and normalised. Transform the mean and sample points into a vector starting with a negative sample mean using the operation of coordinates. The projection of the sample vector onto the mean vector is derived using the formula for vectors as the new eigenvalues.

3. Weak Linear Classifier

3.1 Weak Learner Fit function



```
# Define a Training method
def weak_learner_fit(dataPosi, dataMinus, X, y, weight_posi=None, weight_minus=None, sample_weight=None):
    # The Minimum error
    best_err = 1
    # The best threshold
    best_thres = 0
    # the sign of threshold, which means, 1 is > threshod, 0 is < threshold.
    best_op = 1
    #self.new_Fea = []
    new_Fea = []

    # Initial the weight as 1/n. n is the number of features.
    if sample_weight is None:
        sample_weight = np.ones(len(X)) / len(X)
        weight_posi = np.ones(len(dataPosi)) / len(dataPosi)
        weight_minus = np.ones(len(dataMinus)) / len(dataMinus)
    # Call data_Process() function to convert two features to one feature and return training means.
    new_Fea, Train_Posi_mean, Train_Minus_mean = data_Process(dataPosi, dataMinus,X, y, weight_posi, weight_minus)
```



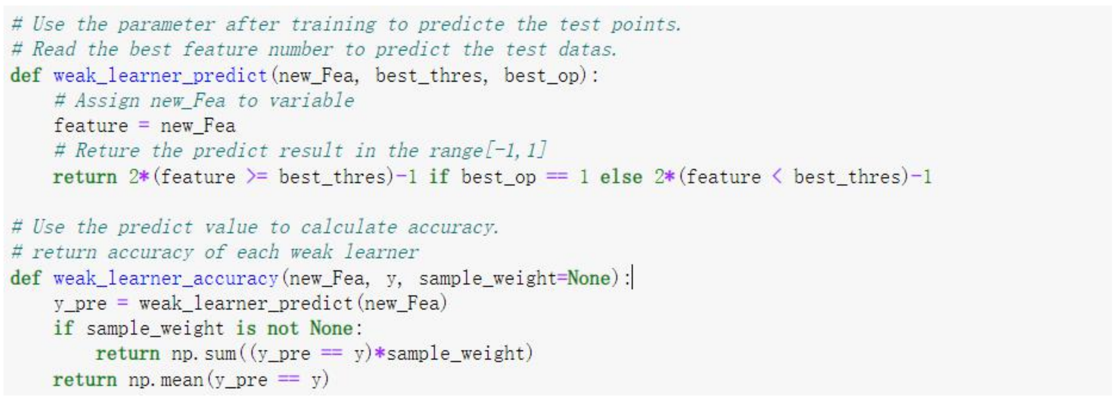
```
    # Loop for each feature, and find the best feature to classfy.
    for i in range(1):
        feature = new_Fea
        # Sort the feature value from small to bigger.
        fea_unique = np.sort(np.unique(feature))
        # Loop for computing the error in each medpoint(threshold).
        for j in range(len(fea_unique)-1):
            # Setting all possible threshold
            thres = (fea_unique[j] + fea_unique[j+1]) / 2
            for op in (0, 1):
                # Find the better sign: > or <.
                y_ = 2*(feature >= thres)-1 if op==1 else 2*(feature < thres)-1
                err = np.sum((y_ != y)*sample_weight)
                # Store all parameters when the error is minimum.
                if err < best_err:
                    best_err = err
                    best_op = op
                    #self.best_fea_id = i
                    best_thres = thres
    # return usefull arrays
    return new_Fea, Train_Posi_mean, Train_Minus_mean, best_err, best_thres, best_op
```

Figure.4: Weak Learner Fit function

Define four variables to store the minimum error value, the optimal threshold value, the optimal inverse value, and the new feature value, respectively. Initialise all the weights and assign them equally. Call the data processing function to return and store the positive and negative sample averages and the new features.

In a for loop, first remove all feature values that are duplicates and sort them from smallest to largest. Use the for loop to iterate through all the thresholds, determining the inverse of greater than and less than. Each sample is compared to the label and multiplied with the corresponding weight for each sample to get the error rate. Returns the optimal error rate, >/<, threshold, new features and positive and negative sample means.

3.2 Weak Learner Prediction & Accuracy Function



```
# Use the parameter after training to predicte the test points.
# Read the best feature number to predict the test datas.
def weak_learner_predict(new_Fea, best_thres, best_op):
    # Assign new_Fea to variable
    feature = new_Fea
    # Reture the predict result in the range[-1, 1]
    return 2*(feature >= best_thres)-1 if best_op == 1 else 2*(feature < best_thres)-1

# Use the predict value to calculate accuracy.
# return accuracy of each weak learner
def weak_learner_accuracy(new_Fea, y, sample_weight=None):
    y_pre = weak_learner_predict(new_Fea)
    if sample_weight is not None:
        return np.sum((y_pre == y)*sample_weight)
    return np.mean(y_pre == y)
```

Figure.5: Prediction and accuracy function.

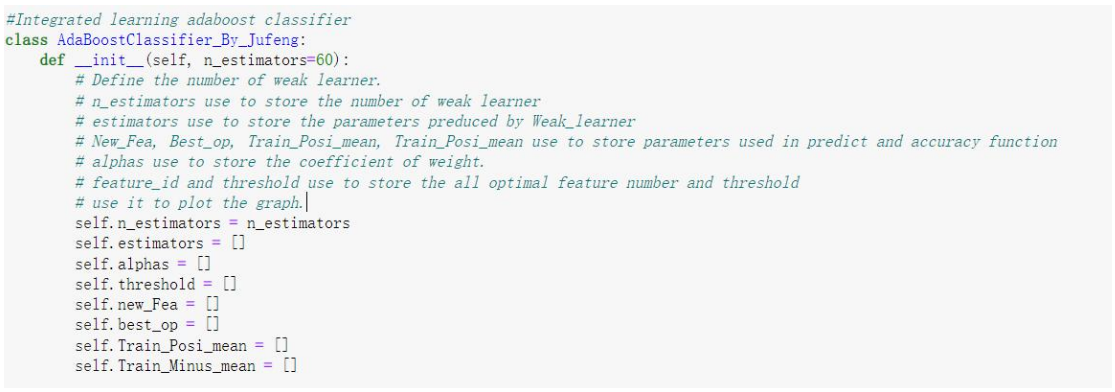
The prediction function uses a weak learner to make predictions, passing the feature value, the optimal operation symbol, and the optimal threshold into the function, using the optimal symbol as the judgment condition, and returning 1 if the feature value matches the threshold and the >/< symbol direction, and 0 otherwise.

The accuracy function uses the prediction function to return 0/1 data, compare it with the sample label, the same returns 1, different returns 0, finally calculate the proportion of 1 in all samples, the proportion value is the correct rate.

These are all the functions of the weak learner. It is easy to see that this weak linear learner is very similar to the decision tree stump in that both use thresholds, >/< symbols, to classify. Only instead of selecting the most suitable feature for classification, this weak learner uses only one feature for classification.

4. AdaBoost Classifier

4.1 AdaBoost Initial Function

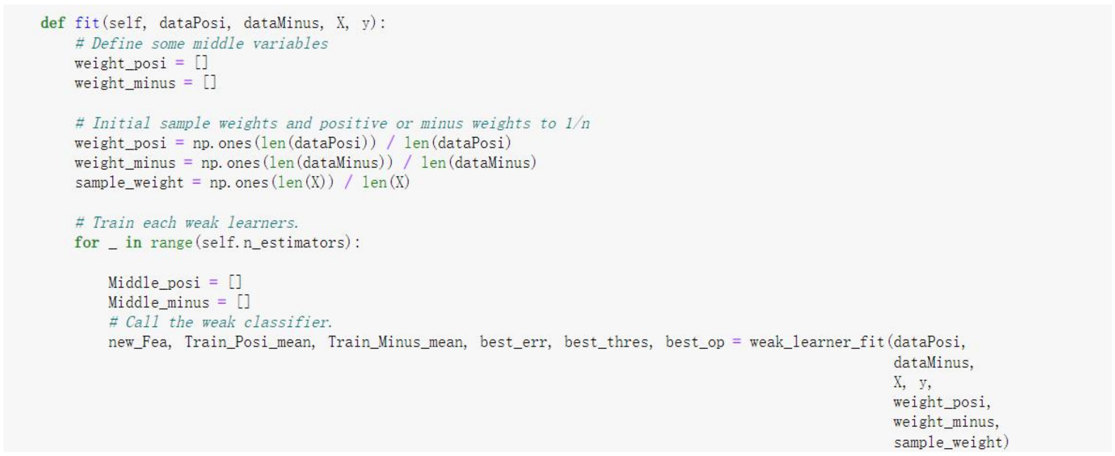


```python
#Integrated learning adaboost classifier
class AdaBoostClassifier_By_Jufeng:
    def __init__(self, n_estimators=60):
        # Define the number of weak learner.
        # n_estimators use to store the number of weak learner
        # estimators use to store the parameters preduced by Weak_learner
        # New_Fea, Best_op, Train_Posi_mean, Train_Posi_mean use to store parameters used in predict and accuracy function
        # alphas use to store the coefficient of weight.
        # feature_id and threshold use to store the all optimal feature number and threshold
        # use it to plot the graph.
        self.n_estimators = n_estimators
        self.estimators = []
        self.alphas = []
        self.threshold = []
        self.new_Fea = []
        self.best_op = []
        self.Train_Posi_mean = []
        self.Train_Minus_mean = []
```

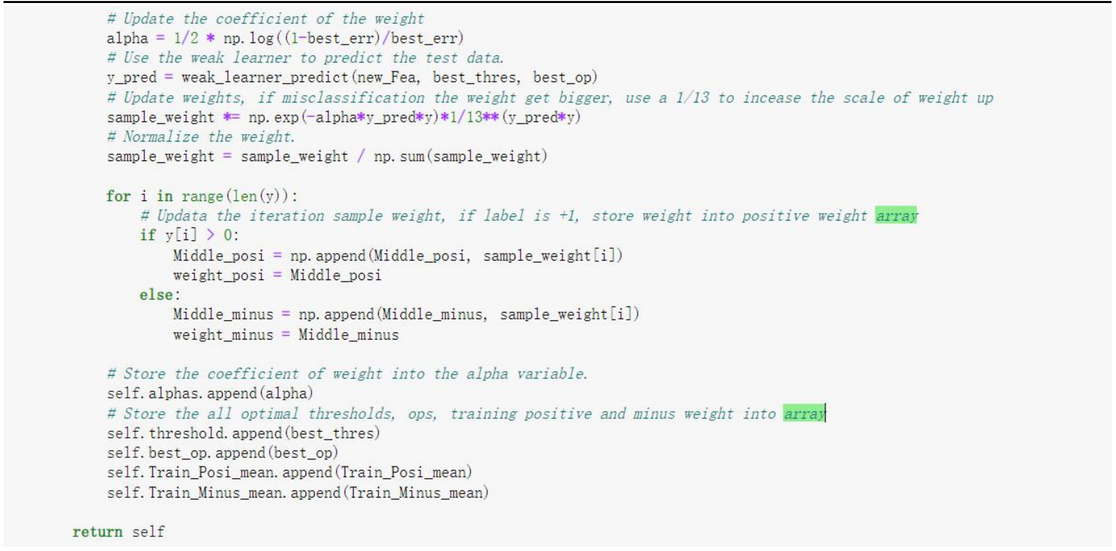
Figure.6: AdaBoost Initial Function.

This whole AdaBoost algorithm is partially set up as a class, and the first function defined inside the class is the class initialisation function. This function defines some global variables for the class, so that all functions in the class will be able to call the values in the variables already defined. Once set, each function can change the value of the variable and be called again after the change, completing the necessary linear operations. Initialising functions, for this AdaBoost, takes a lot of the hassle out of it.

4.2 AdaBoost Fit Function



```python
    def fit(self, dataPosi, dataMinus, X, y):
        # Define some middle variables
        weight_posi = []
        weight_minus = []

        # Initial sample weights and positive or minus weights to 1/n
        weight_posi = np.ones(len(dataPosi)) / len(dataPosi)
        weight_minus = np.ones(len(dataMinus)) / len(dataMinus)
        sample_weight = np.ones(len(X)) / len(X)

        # Train each weak learners.
        for _ in range(self.n_estimators):

            Middle_posi = []
            Middle_minus = []
            # Call the weak classifier.
            new_Fea, Train_Posi_mean, Train_Minus_mean, best_err, best_thres, best_op = weak_learner_fit(dataPosi,
                                                                                                         dataMinus,
                                                                                                         X, y,
                                                                                                         weight_posi,
                                                                                                         weight_minus,
                                                                                                         sample_weight)
```



```
        # Update the coefficient of the weight
        alpha = 1/2 * np.log((1-best_err)/best_err)
        # Use the weak learner to predict the test data.
        y_pred = weak_learner_predict(new_Fea, best_thres, best_op)
        # Update weights, if misclassification the weight get bigger, use a 1/13 to incease the scale of weight up
        sample_weight *= np.exp(-alpha*y_pred*y)*1/13**(y_pred*y)
        # Normalize the weight.
        sample_weight = sample_weight / np.sum(sample_weight)

        for i in range(len(y)):
            # Updata the iteration sample weight, if label is +1, store weight into positive weight array
            if y[i] > 0:
                Middle_posi = np.append(Middle_posi, sample_weight[i])
                weight_posi = Middle_posi
            else:
                Middle_minus = np.append(Middle_minus, sample_weight[i])
                weight_minus = Middle_minus

        # Store the coefficient of weight into the alpha variable.
        self.alphas.append(alpha)
        # Store the all optimal thresholds, ops, training positive and minus weight into array
        self.threshold.append(best_thres)
        self.best_op.append(best_op)
        self.Train_Posi_mean.append(Train_Posi_mean)
        self.Train_Minus_mean.append(Train_Minus_mean)

    return self
```

Figure.7: AdaBoost Fit Function.

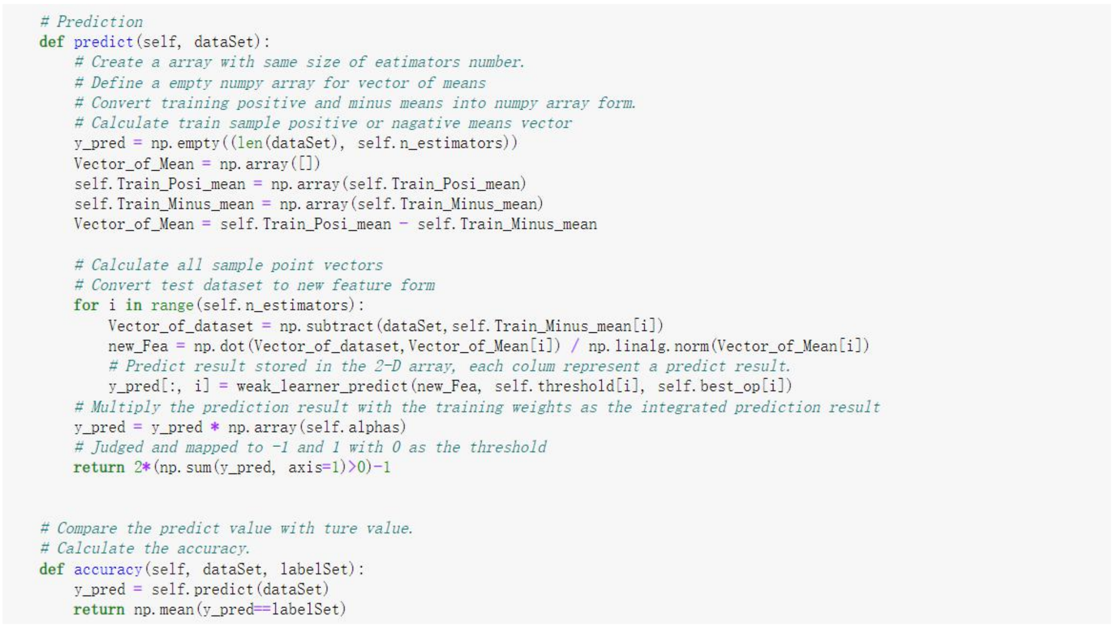
This is the most central part of the entire AdaBoost algorithm. Included in this function are, the weak learner call, the alpha update, the update of each sample weight, and the update of all positive and negative sample means.

The intermediate variables are defined and all sample weights and positive and negative sample weights are initialised (in fact, the positive and negative sample weights are derived from all sample weights). Call the weak learner n times through a for loop, return feature values through the weak learner, train positive and negative sample weights, optimal threshold, optimal >/<. Compute equation 1/2*log(1-best_err/best_err) to define the value of alpha. Pass the threshold and the </> symbol returned by the weak learner into the prediction function of the weak learner to get the prediction value of the weak learner. Multiply the weights by exp(-alpha*pre_y*y)*1/13**(pre_y*y) to update the weights. The 1/13 in the instrument is used to speed up the updating of the weights, which increases the instability of the classifier, but increases the convergence speed of the classifier. The resulting weights are then normalised.

Using a for loop, the sample labels are used as the basis for the judgement, and the weights belonging to the positive and negative samples are added to the corresponding arrays. The same intermediate variables are used here as in the load data function to temporarily store the weights. After these data operations have been completed, the alpha, training sample means, thresholds, etc. are stored in the list.

All the data operations described above are in for loops, where the number of loops represents the number of weak learners, and the relevant arguments are added to a list whose index represents the number of weak learners.

4.3 AdaBoost Predict & Accuracy Function



```
# Prediction
def predict(self, dataSet):
    # Create a array with same size of eatimators number.
    # Define a empty numpy array for vector of means
    # Convert training positive and minus means into numpy array form.
    # Calculate train sample positive or nagative means vector
    y_pred = np.empty((len(dataSet), self.n_estimators))
    Vector_of_Mean = np.array([])
    self.Train_Posi_mean = np.array(self.Train_Posi_mean)
    self.Train_Minus_mean = np.array(self.Train_Minus_mean)
    Vector_of_Mean = self.Train_Posi_mean - self.Train_Minus_mean

    # Calculate all sample point vectors
    # Convert test dataset to new feature form
    for i in range(self.n_estimators):
        Vector_of_dataset = np.subtract(dataSet, self.Train_Minus_mean[i])
        new_Fea = np.dot(Vector_of_dataset, Vector_of_Mean[i]) / np.linalg.norm(Vector_of_Mean[i])
        # Predict result stored in the 2-D array, each colum represent a predict result.
        y_pred[:, i] = weak_learner_predict(new_Fea, self.threshold[i], self.best_op[i])
    # Multiply the prediction result with the training weights as the integrated prediction result
    y_pred = y_pred * np.array(self.alphas)
    # Judged and mapped to -1 and 1 with 0 as the threshold
    return 2*(np.sum(y_pred, axis=1)>0)-1


# Compare the predict value with ture value.
# Calculate the accuracy.
def accuracy(self, dataSet, labelSet):
    y_pred = self.predict(dataSet)
    return np.mean(y_pred==labelSet)
```

Figrue.8: AdaBoost Predict & Accuracy Function.

Since the training samples are using the new eigenvalues, the test samples also need to be transformed into the form of the new eigenvalues, otherwise the test samples will not be tested because the matrix will not match. Of course, during the transformation of the test set, data such as the sample means left in the training set are used to change the form of the test data.

Define an array of predicted values and an array of sample means, transform the training sample means into a numpy array, and calculate the sample mean vector. A for loop is used to iterate through all the weak learners. In the loop, the sample data vector is computed and the sample vector is transformed to project onto the mean vector to generate new eigenvalues. The parameters associated with the weak learner saved in the weak learner are passed into the weak learner projection function to obtain the array of projection values.
The resulting array of predicted values is multiplied by the alpha array to obtain a weighted linear combination of all the weak learners. The predicted values are returned.
The accuracy function calls the above procedure to compare the returned predictions with the labels to obtain the accuracy.

This concludes the AdaBoost subject algorithm.

5. Validation and Test

5.1 Weak learner boundary

```
def show_Boundary(dataPosi,dataMinus,Posi_mean, Minus_mean, thres):
    # define a x space
    x = np.arange(-2, 3)
    # Plot
    for i in range(len(Posi_mean)):
        # Define figure size
        plt.figure(figsize=(5.5,5.5))
        # define the range of x, y axis
        plt.xlim(-2.5,3.5)
        plt.ylim(-2.5,3.5)

        # Plot the line of +/-sample means
        plt.plot([Posi_mean[i,0],Minus_mean[i,0]],[Posi_mean[i,1], Minus_mean[i,1]], color='red', linewidth=2.0)
        # Plot the boundary
        k = (Posi_mean[i,1] - Minus_mean[i,1]) / (Posi_mean[i,0] - Minus_mean[i,0])
        b = ((Posi_mean[i,1] - Minus_mean[i,1])/ np.linalg.norm(Posi_mean - Minus_mean))*thres[i]
        y = (-1/k)*x - (-1/k)*Minus_mean[i,0] + Minus_mean[i,1] +  b
        plt.plot(x,y,color='green', linewidth=2.0)

        #Plot the sample points
        plt.scatter(dataPosi[:,0],dataPosi[:,1])
        plt.scatter(dataMinus[:,0], dataMinus[:,1])
        plt.title('The origin data points and claasify boundary')
        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
```

Figure.9: Show Boundary Function

The show_Boundary function is used to display the boundaries generated by each weak learner. A for loop is used to iterate through all the weak learners, and the number of weak learners is found by measuring the length of the positive samples. Determine the size of the output graph and the range of the axes. Draw the line segments corresponding to all pairs of positive and negative sample means in the order of the i index, Use the mean to find the slope of the line corresponding to the threshold, and find the line over the negative sample mean by the line formula $(Y - Y_0) = -1/K (X - X_0)$, where K is the slope of the line segment $((Y_+ - Y_1)/ X_+ - X_-)$. The value of the threshold is converted into an intercept value by the mathematical relationship $(Y_+ - Y_- / \text{Norm(line segment)})$. At this point, it is sufficient to translate the line over the negative sample mean by an amount equal to the intercept value. Draw the straight line. Draw all sample points.

```
# Load training file
Train_dataPosi, Train_dataMinus,Train_feature, Train_label = loadDataSet('adaboost-train-21.txt')
# Number of esmistor
N = 4
# Call Adaboost and return the positive and nagative means and threshod values
Posi_mean = np.array(AdaBoostClassifier_By_Jufeng(N).fit(Train_dataPosi, Train_dataMinus, Train_feature, Train_label).Train_Posi_mean)
Minus_mean = np.array(AdaBoostClassifier_By_Jufeng(N).fit(Train_dataPosi, Train_dataMinus, Train_feature, Train_label).Train_Minus_mea
thres = np.array(AdaBoostClassifier_By_Jufeng(N).fit(Train_dataPosi, Train_dataMinus, Train_feature, Train_label).threshold)
# Call show_Boundary function
show_Boundary(Train_dataPosi, Train_dataMinus, Posi_mean, Minus_mean, thres)
```

Figure.10: Draw Diagrams.

Load the training dataset, pass the training dataset into the AdaBoost algorithm, return the positive and negative sample means, and pass the sample means into show_Boundary. In total there are four weak learners in this output and the boundary map is shown in the Figure.11.

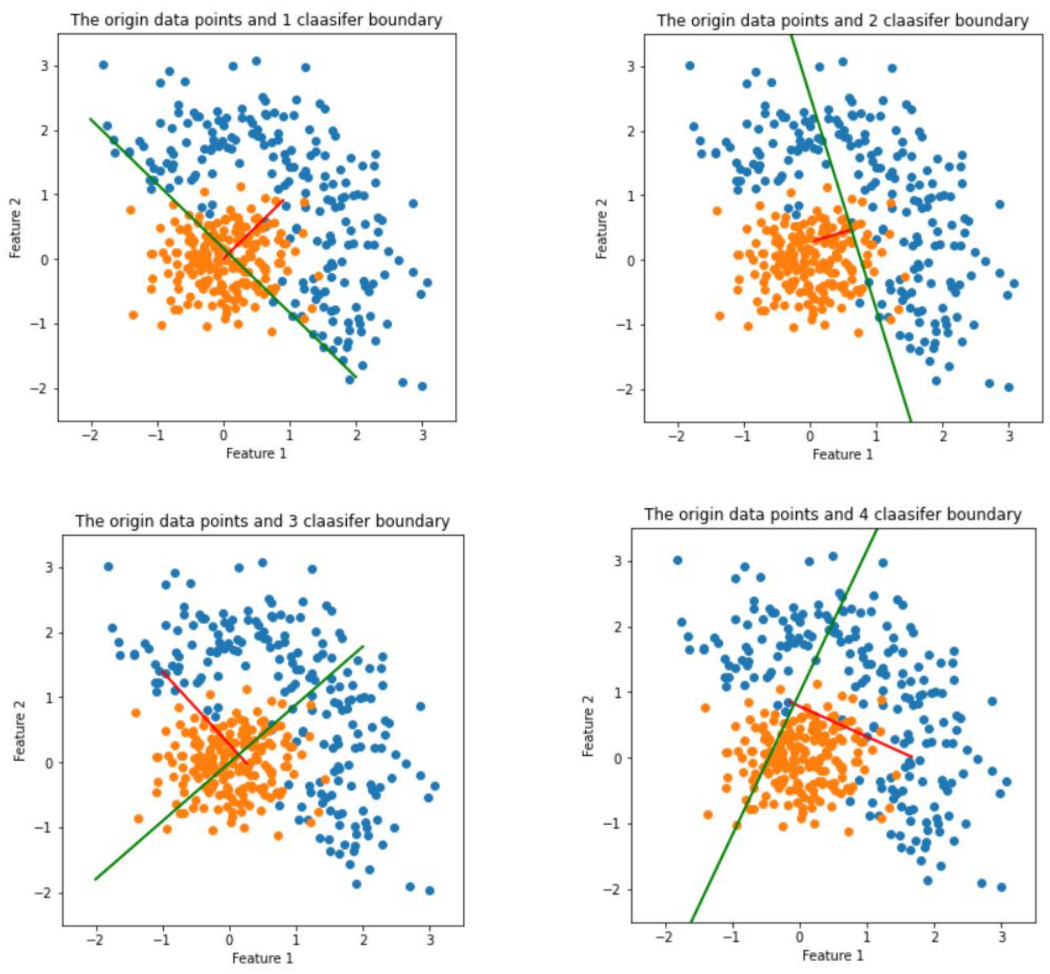


Figure.11: The Boundaries

Where the red line segment is the positive and negative sample mean segment and the green line is the dividing line. As you can see, the green line is perpendicular to the red line segment.

Here we need to explain the size and scale of the diagram in show_Boundary. In this diagram it is necessary to show that the red and green lines are perpendicular to each other, so the scales of the X and Y axes must be of equal length.

In fact, we can look at a plot of AdaBoost output using decision tree stumps as a weak learner.

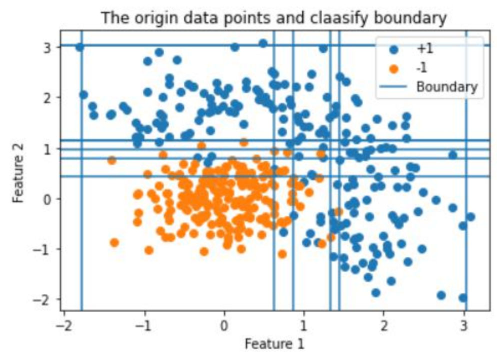
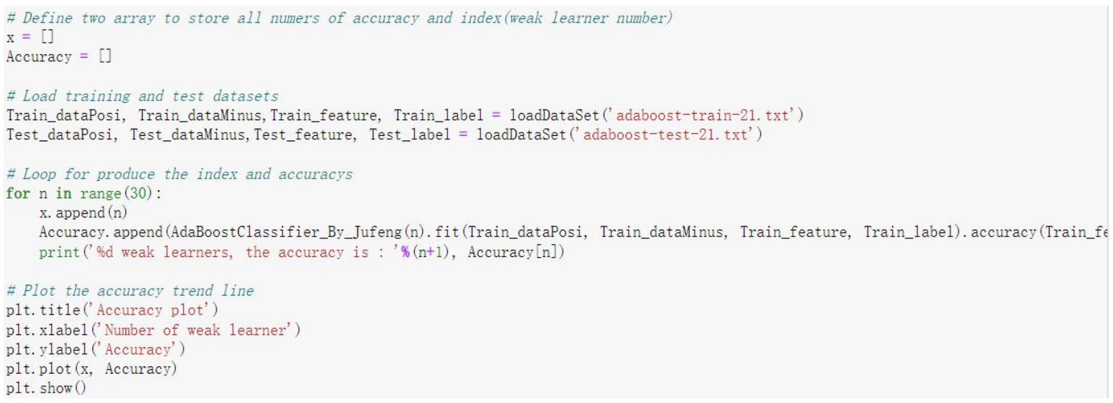


Figure.12: Boundaries by stump

Comparing Fig. 10 and Fig. 11 shows that the decision stump is divided by either a vertical or horizontal line. Whereas the dividing line for the weak linear classifier is not. From this we can also see that the dividing line of the weak linear learner is more flexible than that of the decision stump.

5.2 Accuracy



```
# Define two array to store all numers of accuracy and index(weak learner number)
x = []
Accuracy = []

# Load training and test datasets
Train_dataPosi, Train_dataMinus,Train_feature, Train_label = loadDataSet('adaboost-train-21.txt')
Test_dataPosi, Test_dataMinus,Test_feature, Test_label = loadDataSet('adaboost-test-21.txt')

# Loop for produce the index and accuracys
for n in range(30):
    x.append(n)
    Accuracy.append(AdaBoostClassifier_By_Jufeng(n).fit(Train_dataPosi, Train_dataMinus, Train_feature, Train_label).accuracy(Train_fe
    print('%d weak learners, the accuracy is : '%(n+1), Accuracy[n])

# Plot the accuracy trend line
plt.title('Accuracy plot')
plt.xlabel('Number of weak learner')
plt.ylabel('Accuracy')
plt.plot(x, Accuracy)
plt.show()
```

Figure.13: Test the Accuracy.

The data test set and dataset were first loaded to test 30 weak learners. A for loop is used to gradually increase the number of weak learners and output accuracy.

```
1 weak learners, the accuracy is :  0.5
2 weak learners, the accuracy is :  0.825
3 weak learners, the accuracy is :  0.825
4 weak learners, the accuracy is :  0.6125
5 weak learners, the accuracy is :  0.8375
6 weak learners, the accuracy is :  0.6775
7 weak learners, the accuracy is :  0.875
8 weak learners, the accuracy is :  0.74
9 weak learners, the accuracy is :  0.6425
10 weak learners, the accuracy is :  0.585
11 weak learners, the accuracy is :  0.635
12 weak learners, the accuracy is :  0.6275
13 weak learners, the accuracy is :  0.5475
14 weak learners, the accuracy is :  0.56
15 weak learners, the accuracy is :  0.6275
16 weak learners, the accuracy is :  0.575
17 weak learners, the accuracy is :  0.7325
18 weak learners, the accuracy is :  0.7725
19 weak learners, the accuracy is :  0.7275
20 weak learners, the accuracy is :  0.7675
21 weak learners, the accuracy is :  0.8425
22 weak learners, the accuracy is :  0.8275
23 weak learners, the accuracy is :  0.89
24 weak learners, the accuracy is :  0.8825
25 weak learners, the accuracy is :  0.8775
26 weak learners, the accuracy is :  0.845
27 weak learners, the accuracy is :  0.825
28 weak learners, the accuracy is :  0.8175
29 weak learners, the accuracy is :  0.8175
30 weak learners, the accuracy is :  0.7975
```

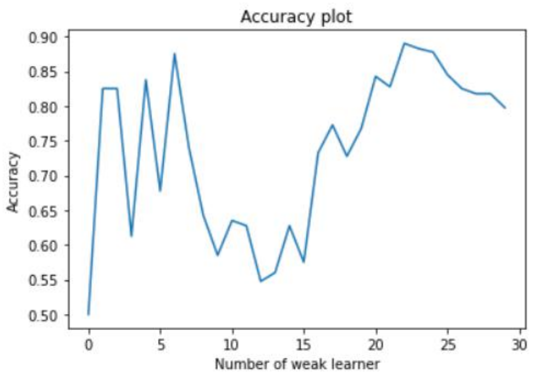
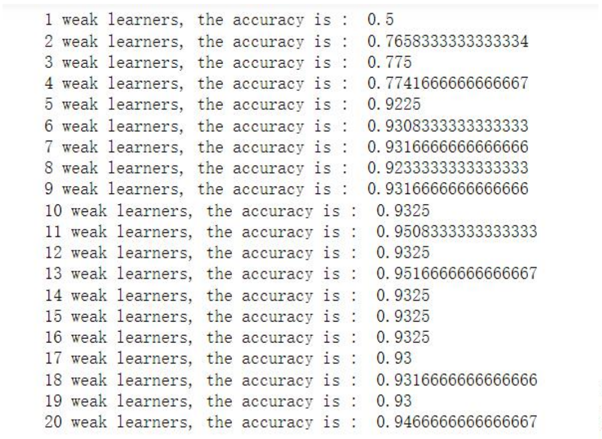


Figure.14: Accuracy figures (Left), Accuracy plot (Right).

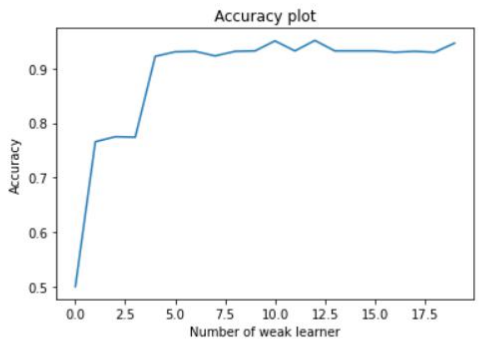
This result is clearly wrong, but for time reasons we did not have enough time to debug it. But this is the most comparable result since this time of debugging. In terms of expectations, if the training set is tested as a test set, we would probably be able to achieve 100% accuracy using

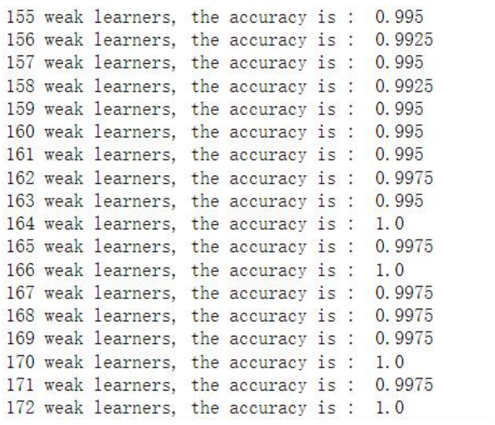
127 learners. If the test set is used as the test set, the highest accuracy rate is nearly 94-95%.

Fortunately, we were able to complete the AdaBoost algorithm using decision tree stakes as a weak learner and the results were satisfactory. The results are shown in Figure 15.



```
1 weak learners, the accuracy is :  0.5
2 weak learners, the accuracy is :  0.7658333333333334
3 weak learners, the accuracy is :  0.775
4 weak learners, the accuracy is :  0.7741666666666667
5 weak learners, the accuracy is :  0.9225
6 weak learners, the accuracy is :  0.9308333333333333
7 weak learners, the accuracy is :  0.9316666666666666
8 weak learners, the accuracy is :  0.9233333333333333
9 weak learners, the accuracy is :  0.9316666666666666
10 weak learners, the accuracy is :  0.9325
11 weak learners, the accuracy is :  0.9508333333333333
12 weak learners, the accuracy is :  0.9325
13 weak learners, the accuracy is :  0.9516666666666667
14 weak learners, the accuracy is :  0.9325
15 weak learners, the accuracy is :  0.9325
16 weak learners, the accuracy is :  0.9325
17 weak learners, the accuracy is :  0.93
18 weak learners, the accuracy is :  0.9316666666666666
19 weak learners, the accuracy is :  0.93
20 weak learners, the accuracy is :  0.9466666666666667
```





```
155 weak learners, the accuracy is :  0.995
156 weak learners, the accuracy is :  0.9925
157 weak learners, the accuracy is :  0.995
158 weak learners, the accuracy is :  0.9925
159 weak learners, the accuracy is :  0.995
160 weak learners, the accuracy is :  0.995
161 weak learners, the accuracy is :  0.995
162 weak learners, the accuracy is :  0.9975
163 weak learners, the accuracy is :  0.995
164 weak learners, the accuracy is :  1.0
165 weak learners, the accuracy is :  0.9975
166 weak learners, the accuracy is :  1.0
167 weak learners, the accuracy is :  0.9975
168 weak learners, the accuracy is :  0.9975
169 weak learners, the accuracy is :  0.9975
170 weak learners, the accuracy is :  1.0
171 weak learners, the accuracy is :  0.9975
172 weak learners, the accuracy is :  1.0
```

Figure.15: Accuracy of AdaBoost with stump. Test dataset(Left), Training datasets(right)

As can be seen from the graphs, the training set test results in roughly 100% correctness when 164 weak learners were used, so it is expected that the weak linear learner converges faster than the decision tree stump. However, the decision stump and the weak linear learner performed similarly in terms of the correctness of the test set.

6. Conclusion

The AdaBoost algorithm is an algorithm that can be matched to a neural network in terms of classification. Its small number of parameters to compute increases his computational speed. For a CPU with 400 weights and 180 learners, the computation time is only 1-2s. Its excellent convergence speed and computation speed will make people prefer AdaBoost in choosing some of the classification algorithms. In the context of deep learning, where a large number of machine learning algorithms are gradually becoming the basis for deep learning algorithms, AdaBoost improves its performance by continuously improving the capabilities of weak learners. AdaBoost is more like a way of thinking about how people approach a problem, by continuously learning and eventually combining the learning of all the weak learners. His performance depends on the capabilities of the weak learners, so we believe that AdaBoost is very widely applicable.

The AdaBoost code is clearly structured in terms of data import, data processing, training of the weak learner, prediction, training of the strong learner, prediction and accuracy functions. The most difficult part of this project was the method of constructing the weak learners and the linear combination of the strong learners. Transforming the sample point features of the dataset into new features, where the effect of each weight on the sample mean should also be considered. However, it is important to note here that if the increase in the weights for the wrong samples is too small, the mean will neutralise the weights and then the convergence of the learner will be slower. Choosing the right way to update the mean is therefore a matter of discretion. Although the final results of this project did not turn out as expected, it is what we would expect to see in a decision stump. After submitting this project, I will continue to finish debugging the code so that the output meets our expectations.

7. Reference

[1] C. Flanagan, Class Lecture, Topic: "WeightedWeakLinear." CE4041, Department of Electronic and Computer Engineering, University of Limerick, Limerick. Available: https://sulis.ul.ie/access/content/group/53588515-e7c0-47e0-90b5-a227730ded3a/Lecture%20Slides/neural/WeightedWeakLinear.4up.pdf