Perceptron Project (Java)

We have modified our code to accommodate the testing of a separate set of test cases by using the weights we synthesized from training to identify a smaller set of test cases that we did not train with.

The **learning_rate** parameter specifies the specific amount of influence that is applied to **update_weights** through every file parse during training. The learning rate parameter is set to 0.1, meaning every time a file is parsed through, 0.1 * error * the pixel value at img array is added to the weight variable, which would not amount to any significant changes in the value of **update_weights** (per instance).

We defined the **training_size** parameter to be the number of files given in the train8 folder and the trainOthers folder. It was important to include all the files given to us in this parameter because it is more efficient to learn from a larger subset of files rather than repeating training on a smaller subset of files. There is more to learn from a larger sample size.

The **max_iterations** parameter specifies the number of times we wanted our function to parse through all 207 text files. The larger that this integer is, the more accurate our weights would be. The problem here is that the larger that this integer is, the longer the runtime and cost of the process. We tried to find reasonable middle ground where we calculated relatively accurate values without utilizing too much overhead.

The **bias** parameter is used to augment the change to the value of **update_weights**.

We used these parameters to track the change in the values of **update_weights** and **err** through various integer values for **max_iterations**. We decided the value of **max_iterations** to be the value that we used when we determined that the change in values of **learning_rate * file[g][j] * error** were becoming marginally smaller. Thus, the more iterations that we ran through, the finer the accuracy. We tried to find reasonable middle ground where we calculated relatively accurate values without utilizing too much overhead. We found that with a **max_iterations** value of 150, we achieved a perfect 100% accuracy. But, we were able to tune that number down while only marginally reducing the accuracy of our perceptron. With a **max_iterations** value of 20, we achieved about a 90% accuracy with both sets of files.

$$\boxed{k_{max} = \frac{\|\mathbf{w}_o\|^2 \beta}{\alpha^2}.}$$

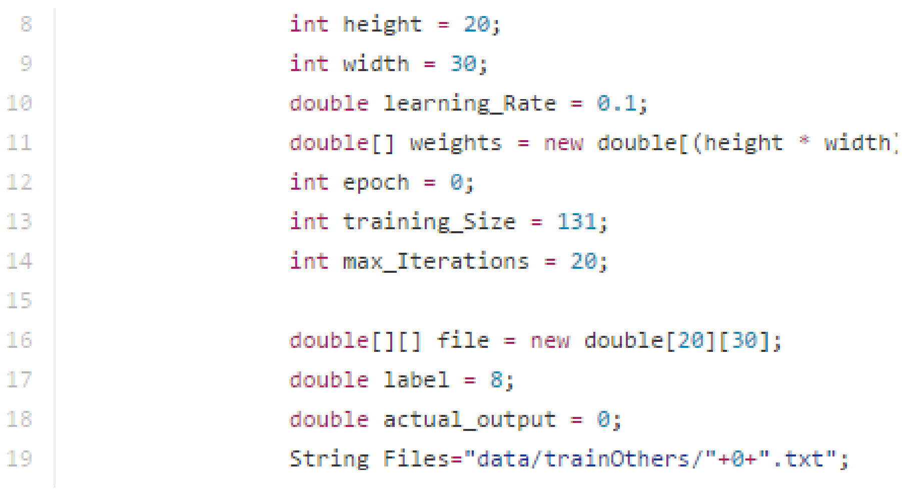
determines maximum number of iteration for the two linearly separable classes to converge.

update bias (weight[2] = weight[2] + LEARNING_RATE*localError;)

Our implementation of this project differed slightly from the pseudocode. Instead of reading the value of the label for each file, we assigned a static value for each label dependant on which file we were passing into the system.

String Files = “File Path” by default as we are reading from trainOthers, we assign the label value to **NOT** 8.
Once we reach end of folder, Files= Train8 and we set the label value to 8.
Once we are done with the first iteration, Files = “File Path” back to the default and label to a value of **NOT** 8 and get read for the coming iteration.

Algorithm And Coding:



```
int height = 20;
int width = 30;
double learning_Rate = 0.1;
double[] weights = new double[(height * width
int epoch = 0;
int training_Size = 131;
int max_Iterations = 20;

double[][] file = new double[20][30];
double label = 8;
double actual_output = 0;
String Files="data/trainOthers/"+0+".txt";
```

<- Variables we Use for the implementation.

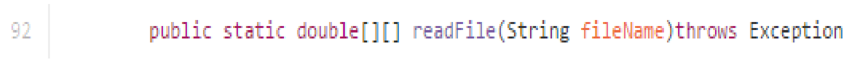
Max_Itteration is a parameter, and we saw that 100 is enough and efficient(time,result) wise

Files-> holds the path name, and you will see later on how do we dynamically change its value

We have created 3 main functions that run the algorithm

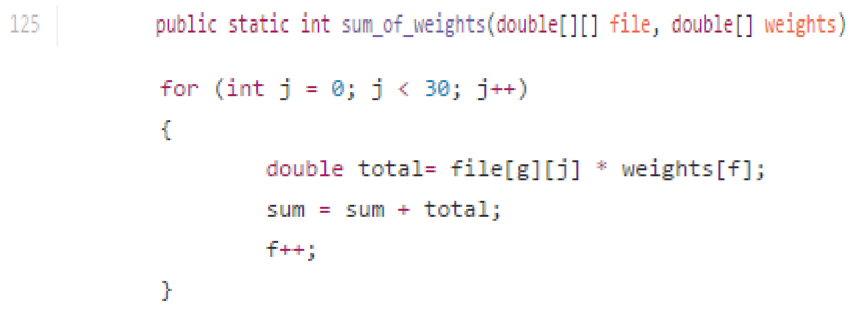
First

readFiles-> it takes the file name, adjust the file in 2d array and converts “ “ to 0 and “+”,”#” to1 And returns an array back.



```
public static double[][] readFile(String fileName)throws Exception
```

Second



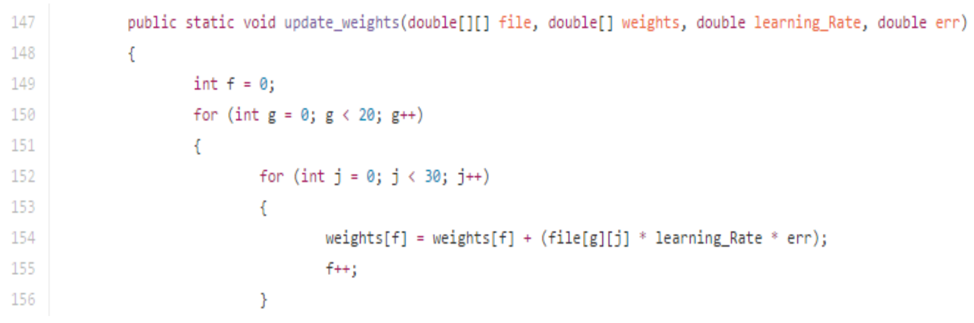
```
public static int sum_of_weights(double[][] file, double[] weights)

for (int j = 0; j < 30; j++)
{
        double total= file[g][j] * weights[f];
        sum = sum + total;
        f++;
}
```

Sum_of_weights-> it simply sum the value of the weights

Third

Update weights-> it adjust the weight



```
public static void update_weights(double[][] file, double[] weights, double learning_Rate, double err)
{
        int f = 0;
        for (int g = 0; g < 20; g++)
        {
                for (int j = 0; j < 30; j++)
                {
                        weights[f] = weights[f] + (file[g][j] * learning_Rate * err);
                        f++;
                }
```

based on the current file

update bias (weight[2] = weight[2] + LEARNING_RATE*localError;)