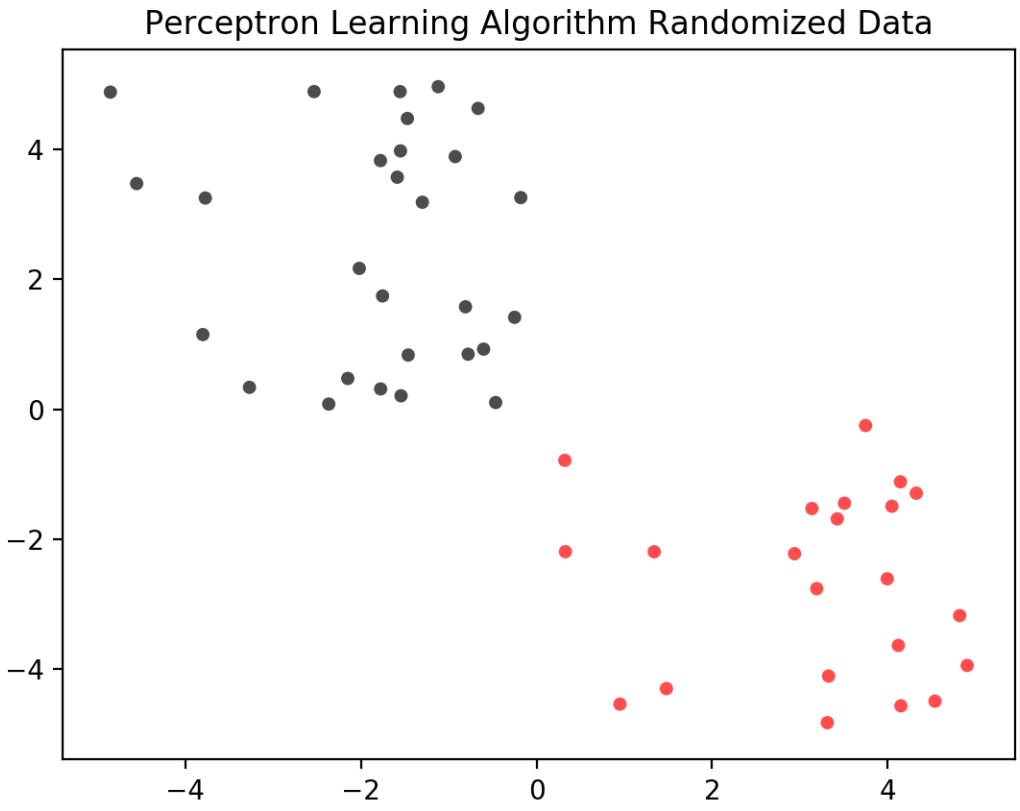
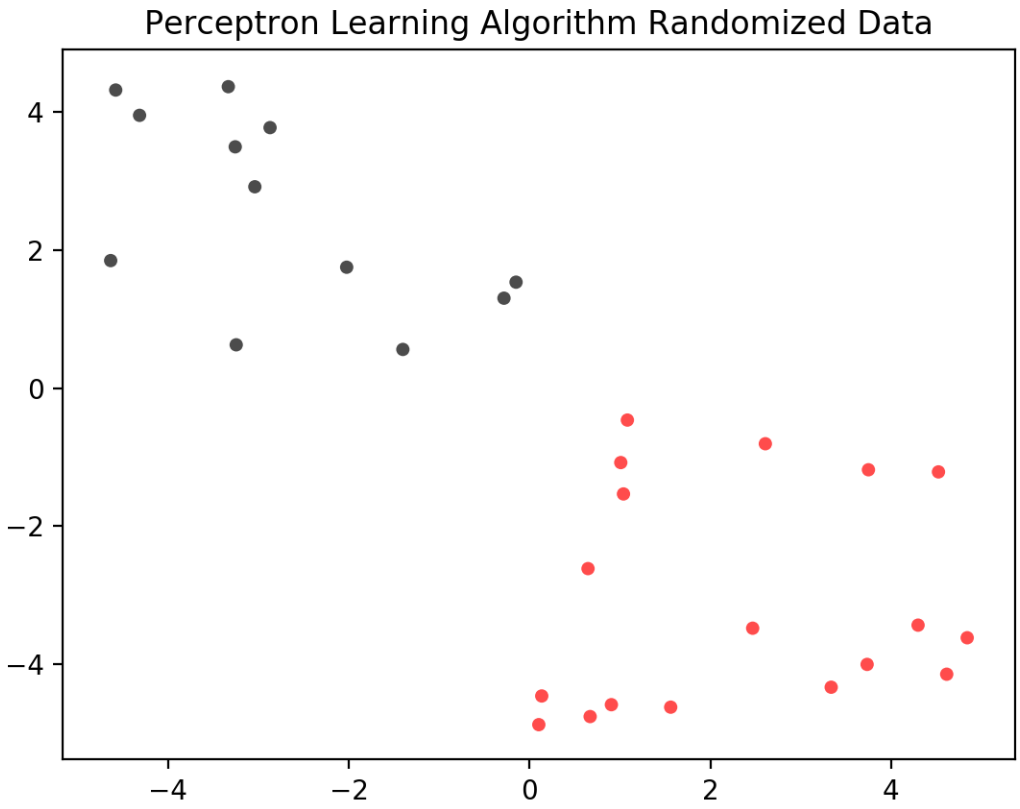
Chris Thompson
CS 4340
Project1
14 September 2017

Project 1 Notes

1. The plot of the training points below show the points to be linearly separable.

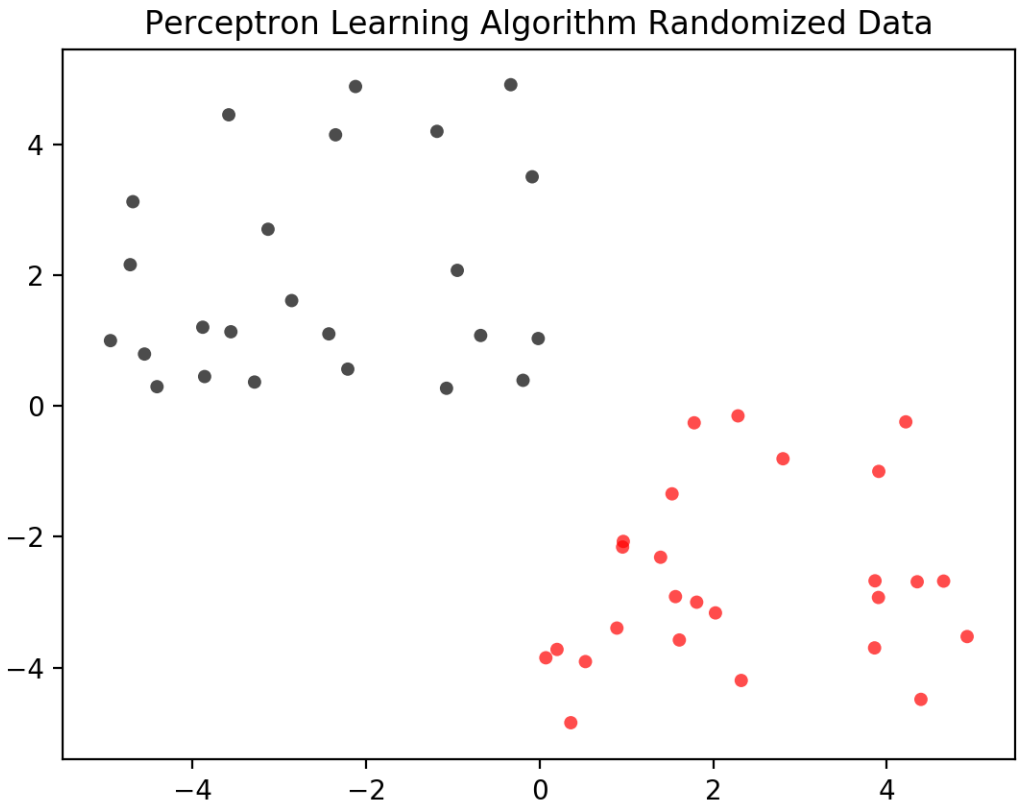


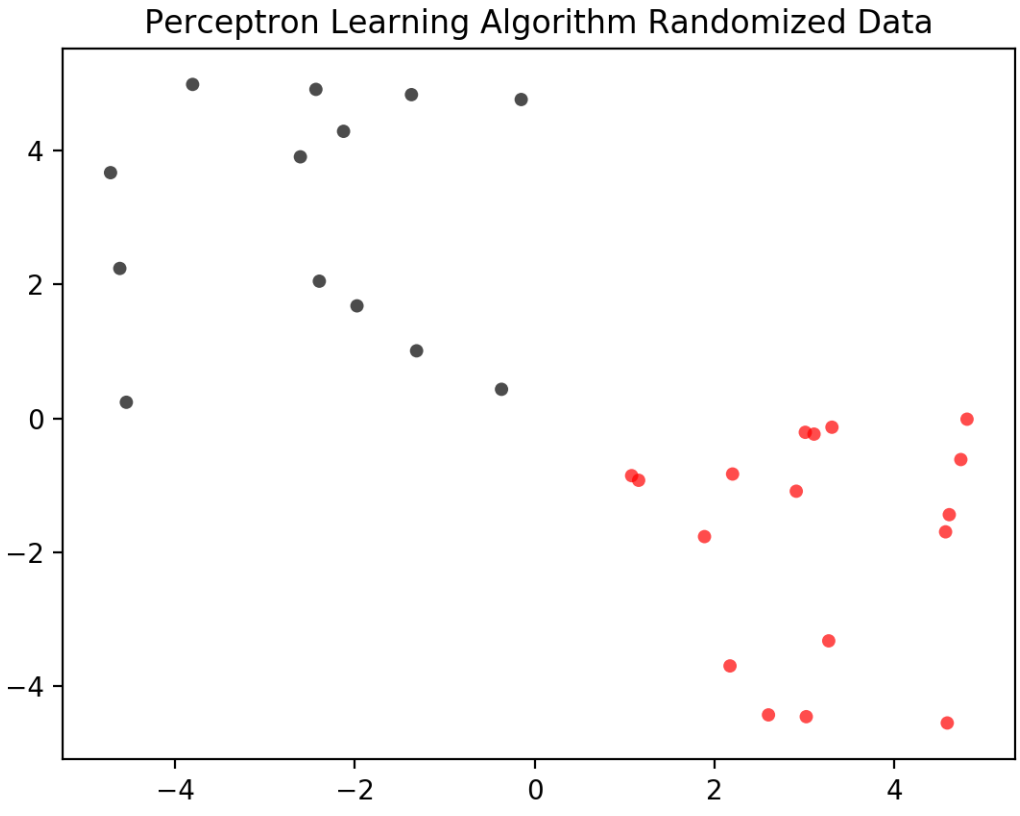
2. The plot of the test points below show the points to be linearly separable



3. My initial choice was to initialize the weights to *0.0000*. My initial choice of the constants were *c = 0.01* and *k = 1*. The reason I chose such low weights and constants was because I wanted to minimize the amount of drastic steps.
4. y = 0.0214800273475x / 0.00473091498525 + -0.01
5. 2 weight updates were made
6. There were a total of two iterations made on the training dataset
7. Final misclassification error:
    a. Training data: 0%
    b. Test data: 0%

8. For this iteration, I chose to increase the weight sizes to *w0 = 5, w1 = 10, w2 = 15* on the below respective training and test data.





a. The effect was:

    a. Number of iterations: 77

    b. Number of weight updates: 330

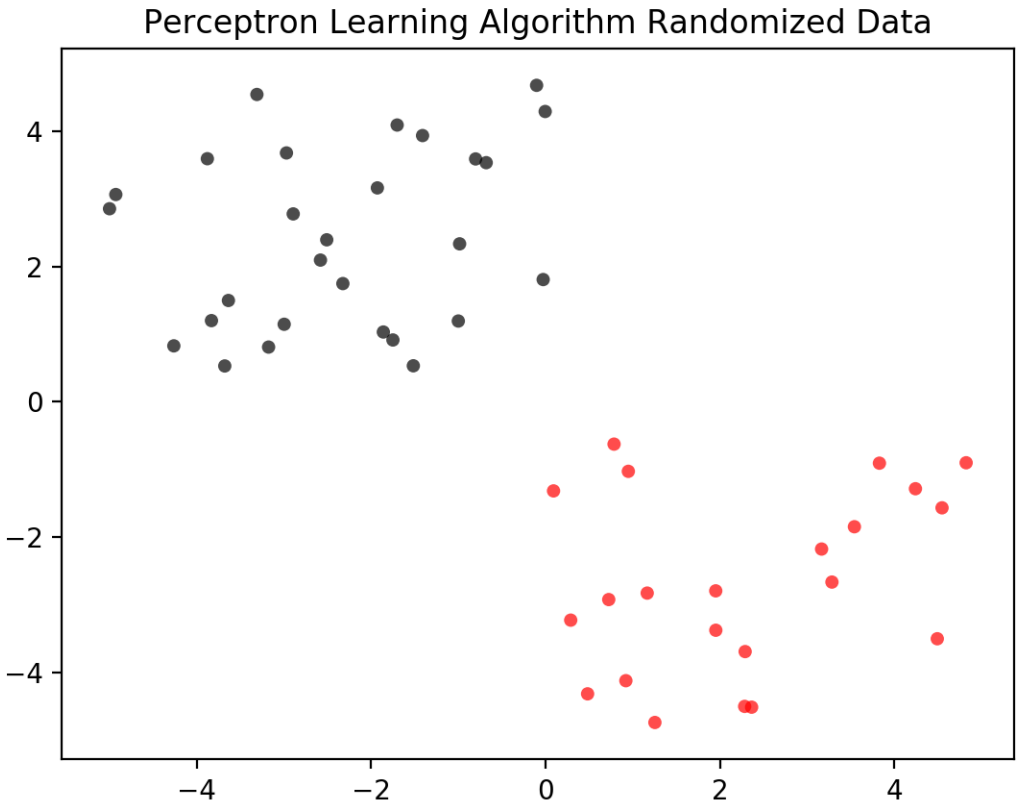
    c. Line equation: y = 0.612912403716x / 16.6276943133 + -3.9

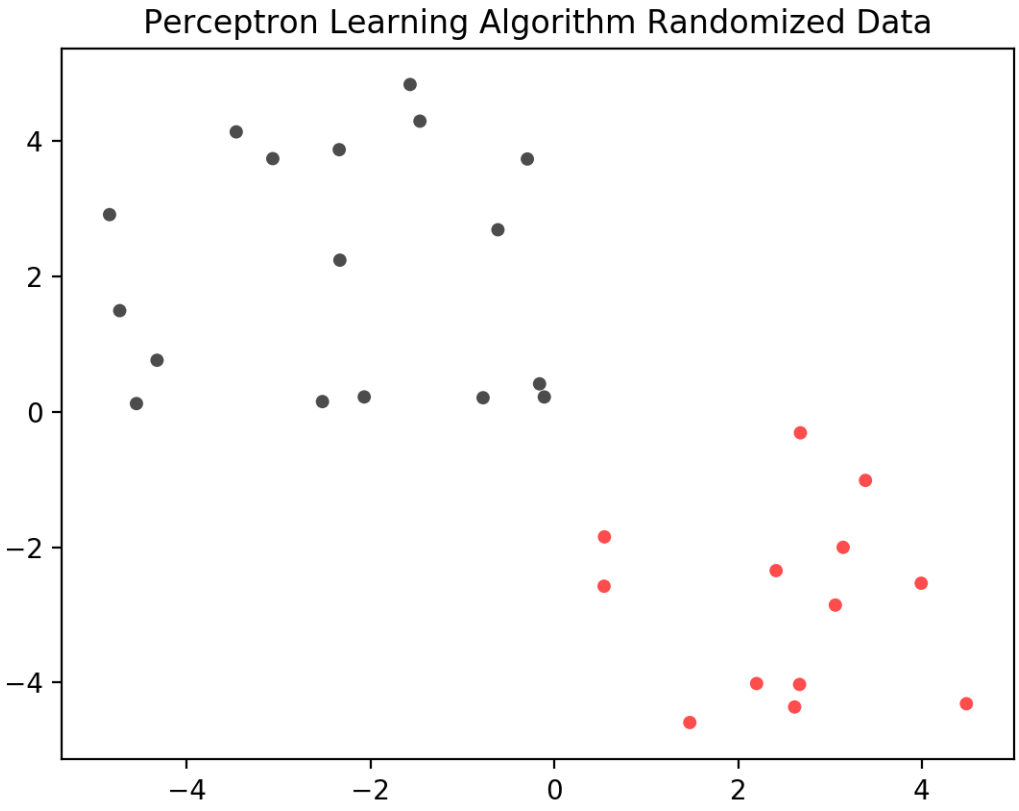
    d. Number of errors on test data: 1

    e. Error Rate: 3.3333333333333335%

    f. Success Rate: 96.66666666666667%

9. For this iteration (training and test data respectively shown below), I re-initialized the weights back to zero (to isolate the effects the constant *c*), but increase the step size to *c = 5*.


Perceptron Learning Algorithm Randomized Data
4
2
0
−2
−4
−4
−2
0
2
4



a. The effect was:

    a. Line equation: y = 8.50685306853x / 20.4508140557 + -5

    b. Number of weight updates: 1

    c. Number of iterations: 2

    d. Number of errors on test data: 0

    e. Error Rate: 0.0

    f. Success Rate: 100.0

10. So as you can see the effect of the weight variations caused the number of iterations to increase. Whereas the step-size constant *c* really had no effect on the number of iterations.

```
'''
Author: Chris Thompson
Project Description: Implementation of the Perceptron Learning Algorithm in Python.
The model is trained on the data
points and then tested on a completely different data set for evaluation.
'''


import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
import os

'''Globals'''
modelLog = open('modelLog.txt', 'w')


def createTrainingData():
    '''create some empty list variables to put randomized data into'''
    x = []
    y = []
    d = []

    '''create some random values for x and y. if the y value > 0, tag it with a 1,
otherwise tag it with a -1'''
    for i in range(50):
        x1 = random.uniform(-5, 5)
        x.append(x1)

        if x[i] < 0:
            y1 = random.uniform(0, 5)
            d1 = 1
            y.append(y1)
        else:
            y1 = random.uniform(-5, 0)
            d1 = -1
            y.append(y1)

        d.append(d1)

    df = pd.DataFrame()
    df['X'] = x
    df['Y'] = y
    df['Class'] = d

    return df

'''create the test data for the model'''
def createTestData():
```

```
    x = []
    y = []
    d = []

    for i in range(30):
        x1 = random.uniform(-5, 5)
        x.append(x1)

        if x[i] < 0:
            y1 = random.uniform(0, 5)
            d1 = 1
            y.append(y1)
        else:
            y1 = random.uniform(-5, 0)
            d1 = -1
            y.append(y1)

        d.append(d1)


    df = pd.DataFrame()
    df['X'] = x
    df['Y'] = y
    df['Class'] = d

    return df

'''this function will test whether the value is positive'''
def isPos(val):
    if val > 0:
        return True
    else:
        return False

'''This function will plot the linearly separable data'''
def plotVals(df,):
    x = df['X'].values
    y = df['Y'].values
    d = df['Class'].values

    '''create a plot and two subplots for -1 and 1'''
    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax2 = fig.add_subplot(111)

    '''iterate over the x,y values and tag and plot them appropriately'''
    for index in range(len(x)):
        if d[index] == -1:
            ax1.scatter(x[index], y[index], alpha=0.7, c='red', edgecolors='none',
s=25, label='red')
        else:
            ax2.scatter(x[index], y[index], alpha=0.7, c='black', edgecolors='none',
s=25, label='black')
    plt.title("Perceptron Learning Algorithm Randomized Data")
    plt.show()


'''This function will update the weights. 3 weights; 2 for the points themselves and 1
for bias'''
def weightsUpdate(weights, constantC, constantK, classificationd, x, y):
    weights[0] = weights[0] + constantC * classificationd * constantK # w0 = w0 + cdk
```

```
    weights[1] = weights[1] + constantC * classificationd * x #w1 = w1 + cdx
    weights[2] = weights[2] + constantC * classificationd * y #w2 = w2 + cdy

    return weights

'''this function will train the weights to make a correct discriminant output'''
def trainModel(df, weights, constantC, constantK, maxIter):
    #grab the values in a list
    x = df['X'].values
    y = df['Y'].values
    d = df['Class'].values

    #define some variables to keep track
    numTurns = 0
    numWeightUpdates = 0
    runningSuccessRate = []

    while numTurns < maxIter:
        localErrorRate = 0
        successRate = 0
        falsePosNeg = 0
        truePosNeg = 0

        for i in range(len(x)):

            #calculate the discriminant
            discriminant = weights[0] + (weights[1] * x[i]) + (weights[2] * y[i])

            if(isPos(discriminant) and d[i] == 1): #if sign(D) == d
                truePosNeg += 1 #add one to the successes

            elif(isPos(discriminant) == False and d[i] == -1):
                truePosNeg += 1

            else: #if sign(D) != d
                falsePosNeg += 1 #add one to the errors
                weights = weightsUpdate(weights, constantC, constantK, d[i], x[i],
y[i]) #update the weights
                numWeightUpdates += 1

        '''take some stats about the iteration and print some updates'''
        numTurns += 1 #increase number of turns by 1 iteration
        print("\n\nNumber of False Positive/Negative: " + str(falsePosNeg))
        modelLog.write("\nNumber of False Positive/Negative: " + str(falsePosNeg))
        print("Number of True Positive/Negative: " + str(truePosNeg))
        modelLog.write("\nNumber of True Positive/Negative: " + str(truePosNeg))
        localErrorRate = falsePosNeg / len(x) * 100
        successRate = truePosNeg / len(x) * 100
        print("Error rate: " + str(localErrorRate) + "%")
        modelLog.write("\nError rate: " + str(localErrorRate) + "%")
        print("Success rate: " + str(successRate) + "%")
        modelLog.write("\nSuccess rate: " + str(successRate) + "%" + "\n")
        print("Number of iterations: " + str(numTurns))
        runningSuccessRate.append(successRate)

        '''if the success rate reaches 100%, print the stats about the weights and
number
        of turns then break out of the loop, else if the model hasn't improved over 3
iterations break the loop
         Otherwise, continue until 100% success rate is reached.'''
        if successRate == 100:
```

```
            print("\nLine equation: " + lineEquation(weights))
            modelLog.write("\nLine equation: " + lineEquation(weights))
            print("Trained Weight Values: " + str(weights))
            modelLog.write("\nTrained Weight Values: " + str(weights))
            print("Number of weight updates: " + str(numWeightUpdates))
            modelLog.write("\nNumber of weight updates: " + str(numWeightUpdates))
            print("Number of iterations: " + str(numTurns))
            modelLog.write("\nNumber of iterations: " + str(numTurns) + "\n")
            break
        else:
            continue

    print("Error rate: " + str(localErrorRate) + "%")
    modelLog.write("\nError rate: " + str(localErrorRate) + "%")
    print("Success rate: " + str(successRate) + "%")
    modelLog.write("\nSuccess rate: " + str(successRate) + "%" + "")
    print("\nLine equation: " + lineEquation(weights))
    modelLog.write("\nLine equation: " + lineEquation(weights))
    print("Trained Weight Values: " + str(weights))
    modelLog.write("\nTrained Weight Values: " + str(weights))
    print("Number of weight updates: " + str(numWeightUpdates))
    modelLog.write("\nNumber of weight updates: " + str(numWeightUpdates))
    print("Number of iterations: " + str(numTurns))
    modelLog.write("\nNumber of iterations: " + str(numTurns) + "\n")

    return weights #return the trained weights

'''this function will test the accuracy of the model with one pass through the
dataset'''
def testModel(weights, testDF):

    #grab the values from the test data
    x = testDF['X'].values
    y = testDF['Y'].values
    d = testDF['Class'].values

    #create an empty list of predicted values
    D = []
    numErrors = 0
    numCorrect = 0

    for i in range(len(x)):
        discriminant = weights[0] + (weights[1] * x[i]) + (weights[2] * y[i]) #make
the prediction
        if(discriminant >= 0):
            D.append(1)
        else:
            D.append(-1)

    resultsDF = pd.DataFrame()
    resultsDF['Predicted Output'] = D
    resultsDF['Expected Output'] = d

    D1 = resultsDF['Predicted Output'].values
    d1 = resultsDF['Expected Output'].values

    for i in range(len(d1)):
        if d1[i] != D1[i]:
            numErrors += 1
        else:
            numCorrect += 1
```

```
    errorRate = numErrors / len(d) * 100
    successRate = numCorrect / len(d) * 100
    print("\nNumber of errors on test data: " + str(numErrors))
    modelLog.write("Number of errors on test data: " + str(numErrors))
    print("Error Rate: " + str(errorRate) + "")
    modelLog.write("Error Rate: " + str(errorRate) + "%")
    print("Success Rate: " + str(successRate) + "")
    modelLog.write("Success Rate: " + str(successRate) + "%")
    print("Line equation: " + lineEquation(weights) + "\n")
    modelLog.write("\nLine equation: " + lineEquation(weights))
    print(resultsDF)
    modelLog.write("\n" + str(resultsDF))


'''create function to give equation of a line --->  y = mx + b '''
def lineEquation(weights):
    '''Ax + By + c = 0 ---> m = -A/B ---> y = -A/B - c'''
    c = weights[0]
    A = weights[1]
    B = weights[2]

    #create the string for equation of the line
    string = "y = " + str(A * -1) + "x / " + str(B) + " + " + str(c * -1)

    return string

def main():
    weights = [0.0000, 0.0000, 0.0000]
    df = createTrainingData()
    plotVals(df) #just to show that the data is linearly separable
    weights = trainModel(df, weights, 0.01, 1, 30000)
    testDF = createTestData()
    plotVals(testDF)
    testModel(weights, testDF)
    modelLog.close()

main()
```