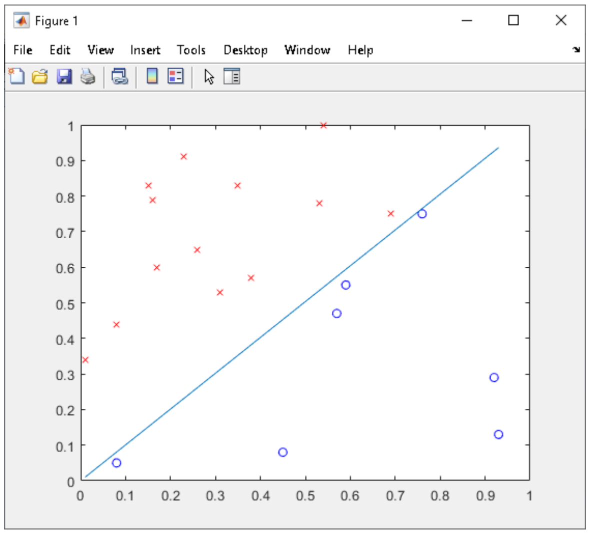
Introduction:

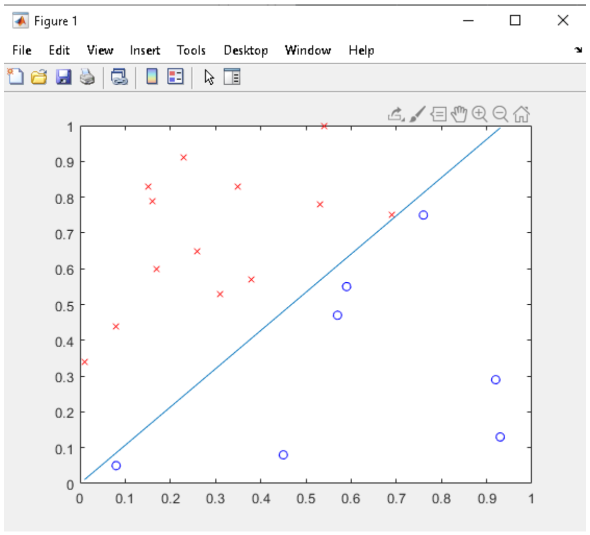
I think this assignment is about creating code that is able to take in matrices and create an algorithm that checks whether vectors of the matrix are on the correct side of the hypothesis perceptron. If a vector ends up being on the wrong side of the perceptron, the hypothesis will be corrected by adding the product of the vector and the ground truth. The ground truth is represented by a 1 or -1. Therefore, this assignment mainly pertains to finding a boundary line to separate the points represented by -1 from the points represented by 1. The algorithm is an attempt to make it as accurate as possible whether the data is linear separable or not. This algorithm also checks the percentage of errors that occur from the total number of iterations done.

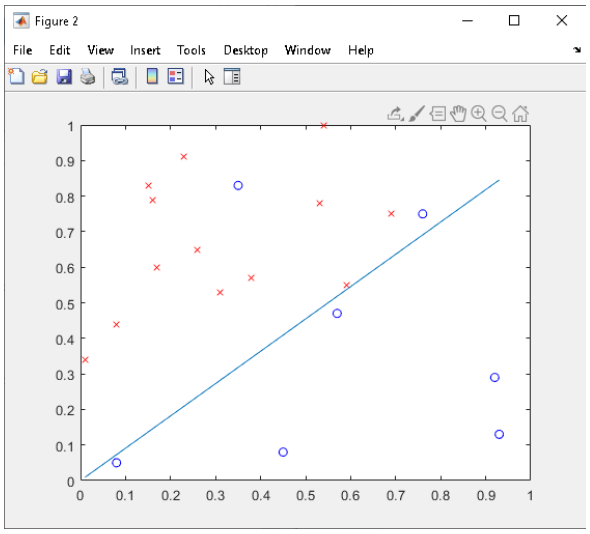
Method:

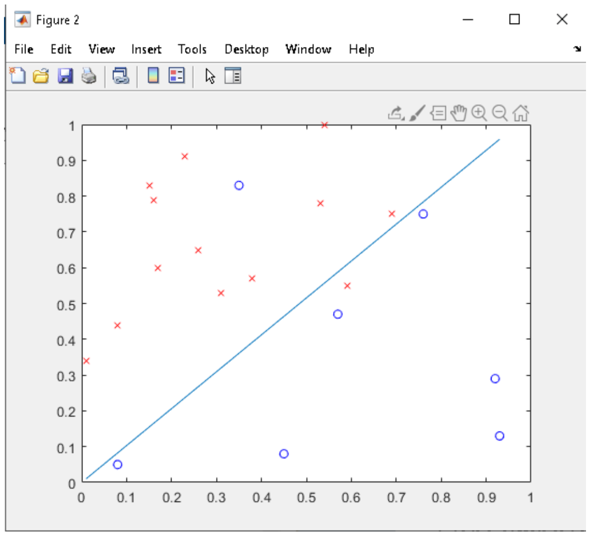
The method used for this assignment was the perceptron learning algorithm, otherwise known as PLA. The first part was to initialize the weights to zeros. I then ran a while loop that ran until there were no errors going through a full iteration of each point in the graph. Since part 1 is supposed to deal with linear separable data, there should be a perceptron that does not have wrong points on either side. Therefore, there will be a point where 0 errors will occur, thus allowing there to be an optimal perceptron. Inside of the while loop, like said before, I iterated through every vector by using a for loop. Within the for loop, I generated a random number between 1 and the number of columns in the matrix to get a random index. I multiplied the transpose of weights to the x vector at the random index to get a scalar number. If the sign of that number and the sign of the ground truth differed, then weights would be adjusted by adding the product of the x vector and the ground truth. This would also be considered an error so 1 was added to the error variable. The error variable was then divided by the total number of iterations executed to find the error rate. Next was to use the pocket perceptron algorithm for parts 2-4 to deal with non-linear sets of data. By non-linear sets of data, I am talking about how there will not be a line that divides the ground truth 1 points from the ground truth -1 points. The algorithm is very similar, but because there will be no clear-cut perceptron, there needs to be a limit as to how many executions the loop goes through before breaking out of the loop. Therefore, the same idea of PLA applies that when the sign of the transpose of weights multiplied by the x vector differs from the sign of the ground truth, the weights need to be compared to a set of weights that experienced less errors. This was defined before the algorithm started where the less error weights (w*) compared to the current weights (w_{t+1}). If w_{t+1} experienced less errors than w*, then w* becomes w_{t+1}. This was repeated until the maximum number of executions was reached. These components make up the entire algorithm, yet sometimes there are some differences with each time the code runs. Because the code runs by taking random indexes at a time and adding to the perceptron when there is an error, there might be some slight differences in the boundary line, although it should be very similar each time.

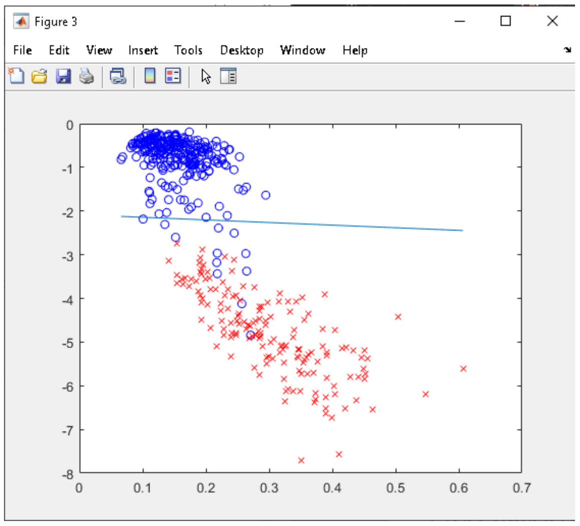
Experiments:

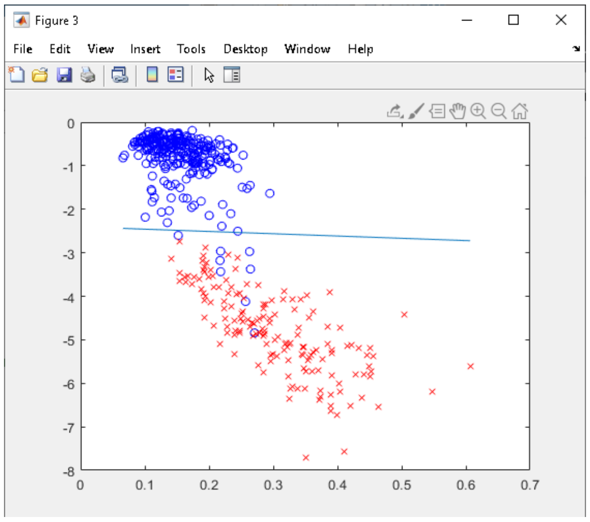












	Trial 1 Error	**Trial 2 Error**	**Trial 3 Error**
Linear Separable Data	0.15	0.157143	0.139130
Noisy (Non-Linear) Data	0.222826	0.220652	0.227273
Handcrafted Feature Data	0.037736	0.070755	0.023585
Raw Pixel Feature Data	0.009434	0.011792	0.028302

Discussion:

As we can see in the images shown above, there are slight differences due to the usage of randomizing, which points are being checked. But, even though that will cause differences, the differences are not drastic. If there were more points that were closer to the perceptron, then there would not be as big of a difference since any error will account for that. In addition, it also seems that in terms of linear separable data and non-linear separable data, the error rate is lower for linear separable data. This makes sense since it is expected that once the linear separable data has no errors, the perceptron is optimal. On the other hand, with noisy (non-linear) data, if the maximum number of iterations has not been reached, there will continue to be errors since there will never be a perceptron that can divide the points properly. In addition, the handcrafted feature data seemed to have a very low error rate and that had to do with the fact that there were so many points bunched together of ground truth 1 on top and ground truth -1 on the bottom. The errors mainly resulted from the points in the middle that intertwine more or less. As for the raw pixel feature data, it seemed that there was a very low error rate, thus causing the need for a lower number of iterations. Every time it ran, it typically had between a 0.9% and 2.8% error rate, which is very low. The reason for this range, as mentioned previously, is because of the randomized iterations instead of doing everything linearly each time. Though, with the randomization and possible differences each time the code runs, the perceptrons turn out to be very similar each time it runs. One aspect I noticed that needed to be fixed, however, was in part 3 where the perceptron line had to be plotted. Since the y-intercept had to be non-zero in this case, starting the weights all at 0, or any whole number, would not work.So, I decided to make weights(1)(weights[0] for Python) being the first element of the weights variable to be 0.1, which is not a whole number and close to 0. This was how I experimented and played around with different values creating a solution that did what it was supposed to do. The reason it cannot be 0 is because any element added to weights(1) would be a 1 or -1 where most of the time, if not all of the time, they would cancel each other out to make weights(1) = 0.