Project #2
MA797: Machine Learning

Ella Pavlechko
Due 10/2/19

1 This is a simple data set to illustrate the Perceptron Learning Model. Consider the following training data with two categories (labels):

$$C_1(1) : (0, 1.5)^T, (1, 1)^T, (2, 2)^T, (2, 0)^T$$

$$C_2(-1) : (0, 0)^T, (1, 0)^T, (0, 1)^T$$

That is, there are seven training data points, each data point has two features and a corresponding label.

(a) Plot these seven training data points and observe that they are separable.

(b) Starting with the weights $w_0 = (-2, 4, 1)^T$, where $w_0 = -2$ is the threshold with corresponding artificial coordinate $x_0 = 1$. Plot the linear model $w^T x = 0$. Note that, this linear model doesnt separate all data points.

(c) Using the Perceptron Learning Algorithm, update the weights so that the linear model will eventually separate all seven data points. At each iteration, plot the linear model on the same plot with the data points.

Please show all performed calculations at each iteration of the perceptron learning algorithm as well as the plots.

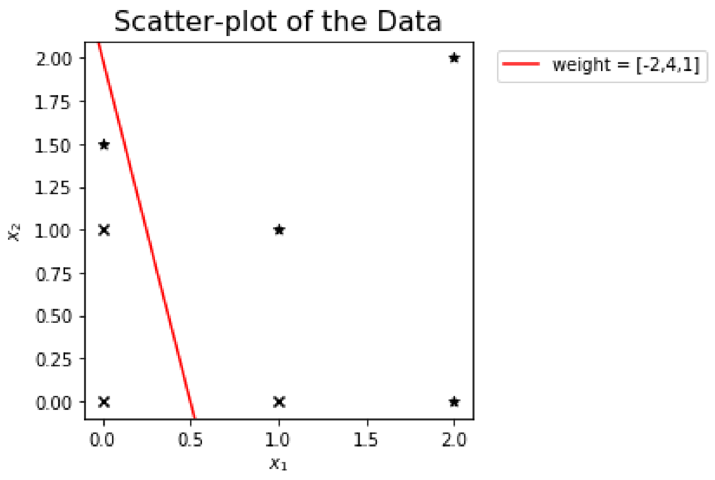
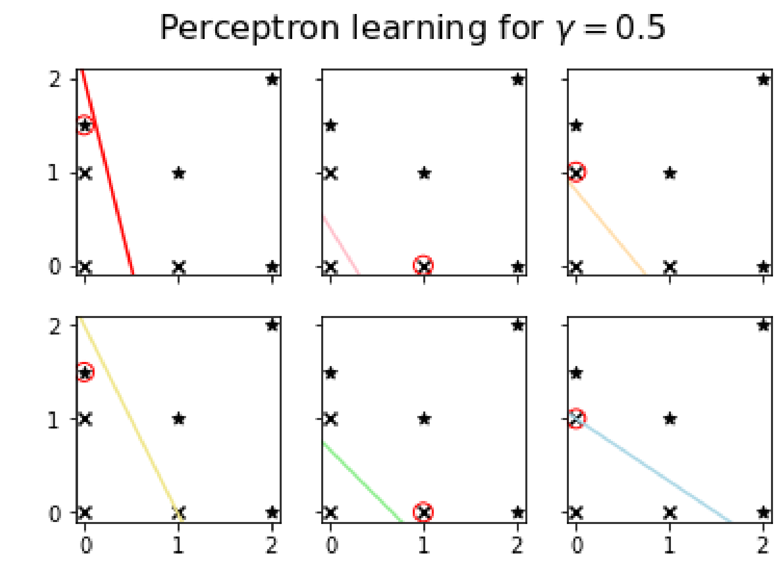


Figure 1: The seven data points and the line $w^T x = 0$

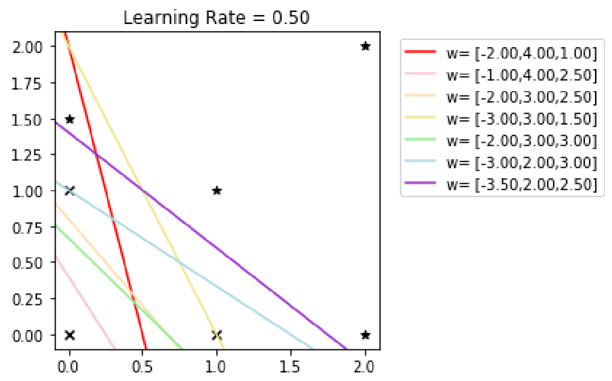
(a) & (b) Denote the data points from Class 1 with a star (★), and the points of Class 2 with a cross (×). We then generate the scatter plot seen in Figure 1 and observe the data will be separated for lines in the region bounded by $x_2 = -\frac{3}{4}x_1 + 1.5$ and $x_2 = -1x_1 + 1$. Combining these boundary conditions together, and re-expressing in vector notation, the data is separated by the line $w^T x = 0$ when the weight vector $w^T = [w_0, w_1, 1]$ has

$$-1.5 < w_0 < -1 \quad \text{and} \quad \frac{3}{4} < w_1 < 1$$

where $x = [1, x_1, x_2]^T$ (Note: the statement will also hold for all scalar multiples of the vector w). However, given this criteria we notice the linear model with weight $[-2, 4, 1]$ will not separate our data. This is further evidenced in Figure 1 where the points $(0, 1.5)$ and $(1, 0)$ are mis-classified by the red line, and thus $w_0^T x = 0$ doesn't separate the data.



(a) Updating the weights, where the next weight is computed based off the mis-labeling of the point circled in red.



(b) Putting all weights from (a) into one graph. The Perceptron's final weight is shown in purple.

Figure 2

(c) Given the initial weight vector w_0 we'd like to update it so that it separates the data. On the first EPOCH, we'll move down the list of points, and whenever we hit a point that is mis-classified we will update our weight using the equation

(1) $$w_{new} = w_{old} + \gamma(y(i) - sign(w_{old}^T\ x(i)))x(i)$$

we'll then continue down the list of points, updating if necessary. Once we reach the end of the list we'll go back to the beginning of the list, starting the second EPOCH, and go down the list. We may stop returning to the beginning of the list when we either reach the pre-set EPOCH value, or when we make a full run-through of the points and no updates of w are needed.

For demonstration purposes, we set $\gamma = 0.5$. A summary of all the updated weights can be seen in Figure 2b, and we will walk through it step-by-step, as in Figure 2a. Starting in the top left with our original weight vector, the first point that's mis-classified is $[1, 0, 1.5]^T$ (highlighted with a red circle). Our new weight becomes

$$w_{new} = \begin{bmatrix} -2 \\ 4 \\ 1 \end{bmatrix} + \frac{1}{2}(1 - sign(-2 + 0 + 1.5)) \begin{bmatrix} 1 \\ 0 \\ 1.5 \end{bmatrix} = \begin{bmatrix} -2 \\ 4 \\ 1 \end{bmatrix} + \frac{1}{2}(2) \begin{bmatrix} 1 \\ 0 \\ 1.5 \end{bmatrix} = \begin{bmatrix} -1 \\ 4 \\ 2.5 \end{bmatrix}$$

Plotting $w_{new}^T x = 0$ gives the pink line in Figure 2a (upper-middle). Since this pink line correctly classifies the rest of the points in Class 1, $data\_label = sign(w_{old}^T\, data)$ turning Equation 1 into $w_{new} = w_{old}$. The next point in the list that the pink line mis-classifies is $[1, 1, 0]^T$, so we must update.

$$w_{new} = \begin{bmatrix} -1 \\ 4 \\ 2.5 \end{bmatrix} + \frac{1}{2}(-1 - sign(-1 + 4 + 0)) \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 4 \\ 2.5 \end{bmatrix} + \frac{1}{2}(-2) \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \\ 2.5 \end{bmatrix}$$

Plotting this line gives the orange line in the top-right corner of Figure 2a. Moving down the list of points, the next point mis-classified is $[1, 0, 1]^T$.

$$w_{new} = \begin{bmatrix} -2 \\ 3 \\ 2.5 \end{bmatrix} + \frac{1}{2}(-1 - sign(-2 + 0 + 2.5)) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \\ 2.5 \end{bmatrix} + \frac{1}{2}(-2) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -3 \\ 3 \\ 1.5 \end{bmatrix}$$

Plotting this line gives the yellow line in the bottom-left corner of Figure 2a.

We have reached the end of the list, so this marks the end of the first EPOCH. We will now circle back to the first data point. The yellow line mis-classifies the first point, $[1, 0, 1.5]^T$, so we will update.

$$w_{new} = \begin{bmatrix}-3\\3\\1.5\end{bmatrix} + \frac{1}{2}(1 - sign(-3 + 0 + 1.25))\begin{bmatrix}1\\0\\1.5\end{bmatrix} = \begin{bmatrix}-3\\3\\1.5\end{bmatrix} + \frac{1}{2}(2)\begin{bmatrix}1\\0\\1.5\end{bmatrix} = \begin{bmatrix}-2\\3\\3\end{bmatrix}$$

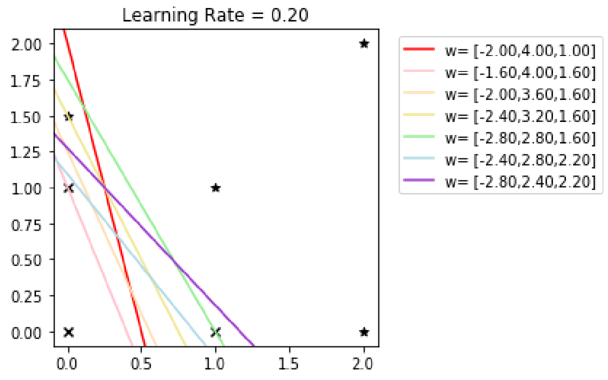
This gives the green line in the bottom-center of Figure 2a. The next point mis-identified is $[1, 1, 0]$ so

$$w_{new} = \begin{bmatrix}-2\\3\\3\end{bmatrix} + \frac{1}{2}(-1 - sign(-2 + 3 + 0))\begin{bmatrix}1\\1\\0\end{bmatrix} = \begin{bmatrix}-2\\3\\3\end{bmatrix} + \frac{1}{2}(-2)\begin{bmatrix}1\\1\\0\end{bmatrix} = \begin{bmatrix}-3\\2\\3\end{bmatrix}$$

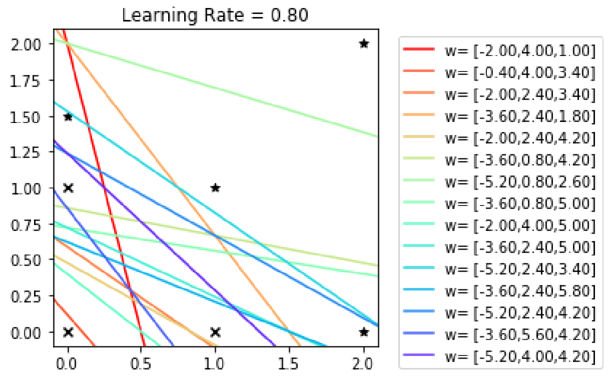
This plots the blue line in the bottom-right of Figure 2a. The last point mis-identified is $[1, 0, 1]$

$$w_{new} = \begin{bmatrix}-3\\2\\3\end{bmatrix} + \frac{1}{2}(-1 - sign(-3 + 0 + 3))\begin{bmatrix}1\\0\\1\end{bmatrix} = \begin{bmatrix}-3\\2\\3\end{bmatrix} + \frac{1}{2}(-2)\begin{bmatrix}1\\0\\1\end{bmatrix} = \begin{bmatrix}-3.5\\2\\2.5\end{bmatrix}$$

Plotting this gives the purple line in Figure 2b. Observe that this line separates the data, so it properly labels everything, and if we continued the Perceptron we would see no further updates occur. Thus, the hyperplane separating the data is $[-3.5, 2, 2.5]\ x = 0$.



(a)



(b)

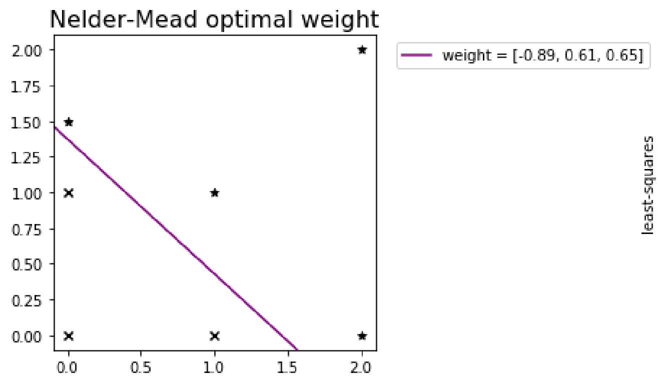
Figure 3: Changing the Learning Rate. The order of weight updates corresponds to their order in the rainbow (ie can be read down the legend)

Now the question arises, what would be the best learning rate, γ? Doing some prelimary testing with $\gamma = 0.2, 0.5,$ and 0.8 we can analyze the number of updates required and the final hyperplane separating the data. Comparing Figures 2b, 3a, and 3b we see that we get multiple different weight vectors separating the data (shown in purple on each plot). We would consider the best γ to be the one which minimizes the distance from the hyperplane to the points. Using the least-squares error, we can compute the best hyperplane $w_{op}^T x = 0$, where

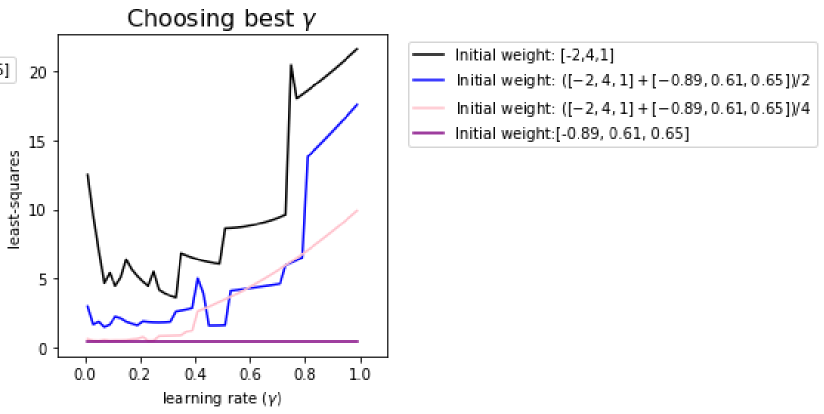
$$w_{op} = \min_{w} \left\{ \frac{1}{7} \sum_{i=1}^{7} (w \cdot x(i) - y_i)^2 \right\}$$

Using the Nelder-Mead solver, we find that $w_{op} = [-0.89, 0.61, 0.65]^T$ and gives a least-squares error of 0.455. The hyperplane defined by this weight is shown in Figure 4a. Thus, the optimal γ would be the one which realizes this weight in our Perceptron Learning Algorithm.

However, as Figure 4b shows, for all γ between 0 and 1 with initial weight $[-2, 4, 1]$, the optimal least-squares error is never attained. Instead, the minimum least-squares is 3.61, and is attained when $\gamma = 0.33$. Testing initial weights of w_{op}, w_0 and $\frac{w_{op}+w_0}{2}$ we find that it becomes possible to attain the optimal hyperplane the closer our intitial weight gets to w_{op}.



(a)



(b)

Figure 4: Optimizing hyperparameters for Problem 1

2 The MATLAB file, SampleCredit.mat, is the data set for the credit card applications. The pdf file, CreditApproval.pdf, has a description for this data set. It is noted the values in the data set was changed by the authors to protect the confidentiality of the data. Use the first 500 data for the training set and the rest for the testing set. Write your own code to implement the Perceptron Learning Algorithm for this data set. For the hyperparameters, set the learning rate to be 0.1 and the number of epochs to be 1000. For the output, report the accuracy for the training data as well as the testing data for the following two cases:

(a) Data is used unchanged

(b) Data is normalized by the maximum value of each feature

(a) After importing the given data and adding a column of one's, our *training_data* became a 500×16 matrix, and the *test_data* was 153×16. To find an initial guess of a weight vector, we set **random.seed(0)** and randomly chose 16 values from a uniform distribution on -1 to 1. Since the importance of choosing a good starting weight vector became apparent in our error analysis in Problem 1, we wanted to initialize our Perceptron Learning Algorithm with the weight that minimizes the least-squares error. Using the Nelder-Mead optimizer, we found our initial weight vector to be w_0,

$$w_0 = \begin{bmatrix} 0.16 \\ -0.31 \\ -0.18 \\ 0.50 \\ -0.28 \\ 0.54 \\ 0.08 \\ 0.50 \\ -0.03 \\ -0.03 \\ 0.13 \\ 0.11 \\ 0.16 \\ 2.15 \\ -0.01 \\ -0.00 \end{bmatrix} \xrightarrow{\textit{Perceptron Learning Algorithm}} w^* = \begin{bmatrix} -572.6 \\ -2586.11 \\ -1169.49 \\ 884.84 \\ -3397.08 \\ -3396.26 \\ -867.32 \\ -12119.3 \\ 12800.61 \\ -12447.83 \\ -3450.87 \\ 8655.11 \\ -2122.24 \\ 2195.75 \\ -1324.4 \\ 10229.6 \end{bmatrix}$$

Taking the initial guess w_0 and applying the same algorithm as Problem 1, after 1000 EPOCH's it found the weight vector w^*. (It's interesting to note that before the Perceptron, the highest weights corresponded to citizenship, ethnicity, bank, and amount of debt. But after the Perceptron they were employment, prior defaults, ethnicity, and income. This could indicate the Perceptron considers these four factors the most when deciding if an applicant is approved or denied.)

Testing to see if the calculated weight vector, w^*, separates the training data, we will take the hyperplane $(w^*)^T x = 0$ and calculate the amount of points this hyperplane mis-identified.

$$\begin{aligned}(\%\ error)_{training} &= \frac{\text{\# points mis-identified in training set}}{500} \\ &= \frac{\text{\# nonzero entries of } [(w^*\ training\_data^T) - training\_labels^T]}{500} \\ &= 35.2\%\end{aligned}$$

Applying the same concept to the testing data, we find the percent error is

$$\begin{aligned}(\%\ error)_{testing} &= \frac{\text{\# points mis-identified in testing set}}{153} \\ &= \frac{\text{\# nonzero entries of } [(w^*\ test\_data^T) - test\_labels^T]}{153} \\ &\approx 32.0\%\end{aligned}$$

Surprisingly, the Perceptron will mis-label credit applicants fairly frequently. Counting the number of positive and negative entries of $(w^*\ test\_data^T) - test\_labels^T$, we find there were 31 positives and 18 negatives. This implies the Perceptron gave 31 people the Credit Card when it should've been denied, and rejected 18 people who should've been approved. So we might consider this Perceptron to be overly kind and should reject more people.

(b) Using the same idea as part (a), after collecting the data, we found the maximum entry in each feature and divided that column of the data by that largest value. Since the features were all positive values, we didn't need to be concerned with the absolute values. Now having 'normalized' data, we took the random 16×1 vector from (a) and used the Nelder-Mead method to find a good first guess for our weight. We'll denote this weight as $\tilde{w}_0$, and after applying the Perceptron Learning Algorithm for 1000 EPOCH's we reach a weight vector of $\tilde{w}^*$

$$\tilde{w}_0 = \begin{bmatrix} 0.51 \\ -0.13 \\ 0.17 \\ -0.23 \\ -0.52 \\ 0.44 \\ -0.25 \\ 0.13 \\ -0.32 \\ -0.17 \\ -0.19 \\ -0.17 \\ 0.00 \\ -0.04 \\ -0.02 \\ -0.51 \end{bmatrix} \xrightarrow{Perceptron\ Learning\ Algorithm} \tilde{w}^* = \begin{bmatrix} 2.710 \\ -0.43 \\ -0.55 \\ -1.07 \\ 0.01 \\ 0.97 \\ -0.29 \\ 0.55 \\ 1.92 \\ -2.37 \\ -1.29 \\ -0.33 \\ -0.50 \\ -0.11 \\ -1.18 \\ 13.66 \end{bmatrix}$$

Here, the factors influencing the approval or rejection the most are the applicant's income, prior defaults, employment status, and years of employment.

Testing how well this Perceptron's hyperplane, $\tilde{w}^T x = 0$, categorizes the data,

$$(\%\ error)_{training} = \frac{\text{\# points mis-identified in training set}}{500} = 23.6\%$$

$$(\%\ error)_{testing} = \frac{\text{\# points mis-identified in testing set}}{153} \approx 18.3\%$$

Upon further investigation, we find that this normalized Perceptron gave 4 people approval when it shouldn't have, but it denied 24 people who the banker approved. Thus, the normalized Perceptron is in general more accurate, but also more strict than the un-normalized version.

Note: Altering the hyperparameters γ and EPOCH for part (a) and (b) don't seem to make significant differences in the percent error. However, when the random seed isn't set to zero at the beginning of the problem, it effects the Nelder-Mean initial guess and we see a change in the accuracy. Running a few simulations the best $(\%\ error)_{testing}$ on the unseen data that I found for (a) was 18.95, and for (b) 13.73. The worst was (a)33.3, (b)29.4 (these were very mean and rejected a lot of people).