Perceptron Lab Report

1 — Algorithm Implementation

As dictated by the lab requirements, I implemented a perceptron class with a “fit” method that uses the perceptron algorithm to determine how weights should be modified over an epoch. If weights are not provided to the “fit” method, I initialize them to an array of randomly generated uniform floats between -1 and 1. As a stopping criterion, I determined to end training when there were a significant number of epochs without improvement in accuracy. I arbitrarily chose 5 as a significant number of epochs, and to determine accuracy I used my perceptron class’s “score” method at the end of each epoch, keeping track of the best overall accuracy from all epochs. If the accuracy for a given epoch was higher than the highest previously found accuracy, the number of epochs without improvement was reset to zero and the new highest accuracy was set to the accuracy from this most recent epoch. Otherwise, if the algorithm iterates through 5 epochs without making any improvements to the accuracy, we terminate training. The “predict” method simply uses the weights calculated by the “fit” method to determine the output for the data given as a parameter and outputs a “1” if the output is greater than zero and outputs a “0” otherwise. The “score” method, in turn, uses the output from the “predict” method and compares it against an expected output to determine the overall accuracy, represented as $\frac{\textit{number correct}}{\textit{number of samples}}$. Unless specifically told not to, my perceptron class will also shuffle the data after each epoch by using the np.random.shuffle function, shuffling the input values and associated labels as one matrix. I also implemented a train/text split function using a similar method. The split function determines how many rows of data need to be considered “test” data based on the given test data proportion, then shuffles the data and, assuming it was determined that *n* rows are to be used for testing, picks the first *n* rows of the shuffled matrix as training data.

After training and evaluating my perceptron class on the provided “debug” data, I found that the implementation was outputting the expected values and felt confident it was correct. As a result, I trained the model on the “evaluation” data, which yielded an accuracy of [redacted], with the weights seen in Table 1.1 below:

Table 1.1

Feature	Variance Wavelet	Skewness Wavelet	Curtosis Wavelet	Entropy Wavelet	Bias
Weight	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

2 — Test Datasets

I created two test datasets in ARFF, each with 8 instances of data and 4 instances for each class. One dataset is linearly separable by design while the other is not. I accomplished this through mathematical reasoning and rough estimation (i.e. I made sure there was a significant difference between the x_2 values for instances of different classes where the x_1 values were relatively close to each other) and verified my results by plotting them. To create the non-linearly separable set, I simply kept the same x_1 and x_2 values and, since I knew these created a linearly separable set, I knew that by swapping some of the associated class values I would necessarily create a non-linearly separable set (because a data point belonging to one side in the linearly separable set would now be classified in the other group). The results of this effort are shown in tables 2.1 and 2.2 below:

Table 2.1 — Linearly Separable Data

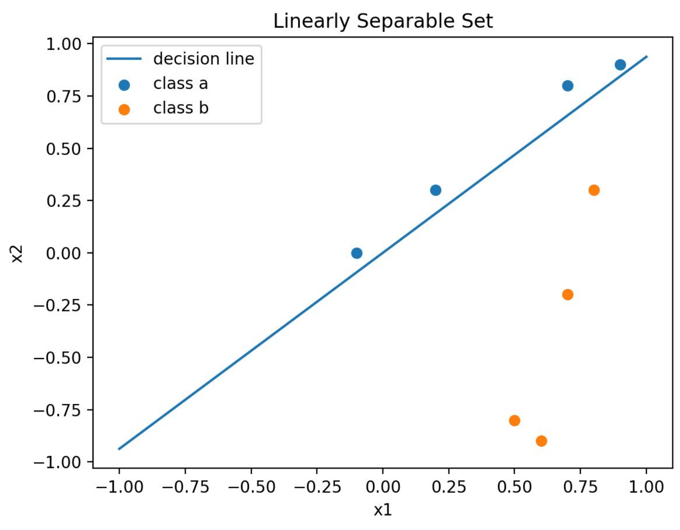
x_1	x_2	Target
0.9	0.9	1
[redacted]	[redacted]	1
[redacted]	[redacted]	1
[redacted]	[redacted]	1
[redacted]	[redacted]	0
[redacted]	[redacted]	0
[redacted]	[redacted]	0
0.6	-0.9	0

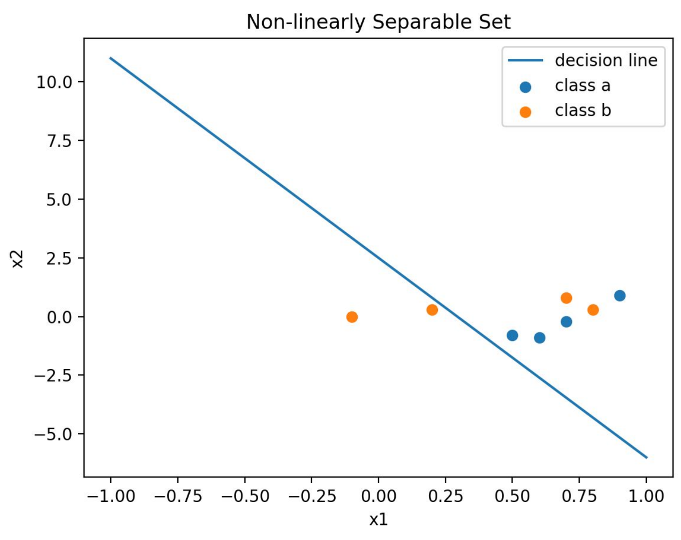
Table 2.2 — Non-linearly Separable Data

x_1	x_2	Target
0.9	0.9	1
[redacted]	[redacted]	0
[redacted]	[redacted]	0
[redacted]	[redacted]	0
[redacted]	[redacted]	0
[redacted]	[redacted]	1
[redacted]	[redacted]	1
0.6	-0.9	1

3 — Results of Training on Test Data

I trained the perceptron classifier on both the above sets, trying different learning rates but keeping the initial weights the same (all zeros) for consistency. I was surprised to find that, as suggested by the lab specification, the learning rate had very little effect on the number of epochs completed to reach the stopping criteria even though the final weights were significantly affected by the learning rate. Granted, this could be partially tied to my arbitrary choice to stop iterating after completing 5 epochs without improvement. Ultimately, however, the biggest surprise to me was seeing that no matter how big or small I made the learning rate, the number of epochs required to train the model on both sets was always the same or nearly the same. Below are depicted graphs of the instances and resulting decision line for both data sets when learning rate is set to 0.05:





Based on the decision line for the linearly separable set depicted above, the classifier seems to have been able to train to 100% accuracy for the training data set. Since the perceptron classifier can only truly separate linearly separable sets, this same accuracy did not hold for the non-linearly separable set of data but it was still able to classify with 75% accuracy, which suggests that even if data is not strictly-speaking linearly separable, the perceptron algorithm may still be a reasonable approximation in situations where the data can be mostly linearly separated.

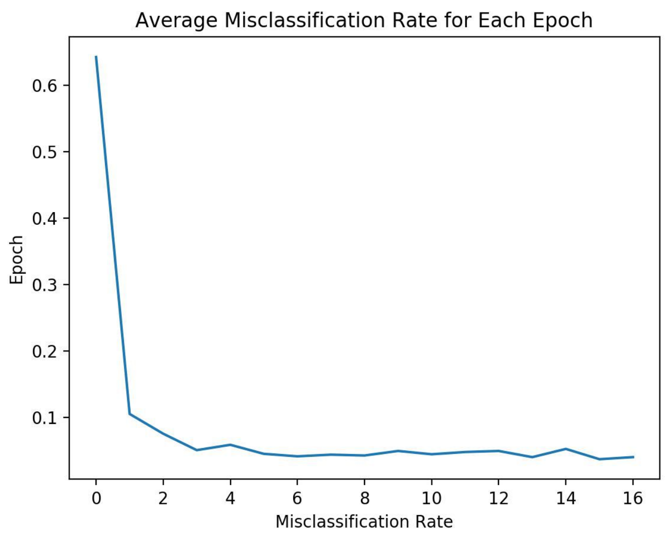
4 — Standard Voting Problem

After evaluating the test data described above, I trained the perceptron classifier on the voting data provided and found that the accuracy rates were quite high. The perceptron was trained five times, each time splitting all data randomly into 30% test data and 70% training data. The training data was also shuffled after each epoch. Below are recorded the accuracies for each trial for the training set, the test set, the number of epochs required to train, and the average for each value over the five trials.

Table 4.1

Trial	**1**	**2**	**3**	**4**	**5**	**Average**
Training Set Accuracy	0.90	0.97	0.97	0.91	0.92	0.93
Test Set Accuracy	0.94	0.96	0.95	0.88	0.93	0.93
# of Epochs	7	13	9	7	14	10

Looking at the final weights in each trial, some features proved more relevant to classification than others. The two features that seemed to have the biggest impact on output (as they consistently had higher weights across the five trials) were [redacted] [redacted] [redacted] [redacted] and [redacted] [redacted] The former two tended to indicate a likelihood of voting Republican while the latter two were more associated with voting Democrat. Some of the features that had small weights and therefore seemed to affect the outcome less included [redacted] [redacted] After graphing the average misclassification rates across all trials for each epoch (depicted to the right), it is evident that the most significant improvements in accuracy are made in the



first 2-3 epochs, with accuracy plateauing and occasionally worsening slightly after several epochs.

5 — Scikit-learn Perceptron

For this section of the lab, I trained the Scikit-learn perceptron classifier on the voting data set and on a set of data about diabetes patients from https://storm.cis.fordham.edu/~gweiss/data-mining/weka-data/diabetes.arff. Given basic initialization parameters (i.e. stopping criteria of 0.01 and a random state value) it produced similar results to my perceptron's results for the voting data (usually between 90-96% accuracy). Like the voter data, I split up the diabetes data randomly with 70% of the data being used for training and the other 30% being used for testing. The scikit-learn perceptron model usually achieved between 60-65% accuracy for the test set (with outlying accuracy being closer to 30% on some trials). It seems the scikit-learn perceptron uses the "lor" parameter as stopping criteria by stopping the training when the difference between loss of two consecutive iterations goes below the threshold "lor" value. The "eta0" parameter likewise seems to correspond to our learning rate parameter. In experimenting with some of the hyperparameters, the ones I found to have the greatest effect were certain combinations of the "penalty" parameter and relatively large values for "alpha," the multiplier for the regularization term specified by "penalty." I found that regularizing with an "L2" penalty and using a relatively large value for alpha (e.g. 1) resulted in significant loss of accuracy for the voting data, dropping from accuracy in the 90's to low 60's and 50's. This is likely because the "L2" penalty combined with a large multiplier didn't allow the weights to be tuned finely enough, resulting in the inability of the model to find the most optimal weights. Using these same parameters for the diabetes data, however, seemed to have little or no effect on the average accuracy of the model for predicting the test data set.