Assignment 1: Adaline and Logistic Regression

Logan Nunno

February 2026

1 Task 1

The way that the bias was absorbed into the weight was that it was added as the first element of the weight vector. This changes the size of that vector so we need to account for that new value inside of every place where we do vector matrix multiplication or any other places where the size of W is needed to be the same as some other provided value. The first place this needs to be done is when calculating the net inputs. The original net inputs is $z = w_1x_1 + w_2x_2 + \cdots + w_mx_m + b$ and we want to keep the same value for this so we need to add a constant 1 to the start of X so that when we do the dot product the shape is valid and we still get the same result. So we add a 1 to the start of X to match with the bias being used as the first element in the vector W. The way I do this is by having the bias be another random number that is created at the same time as W and it is inside of W. I then make a variable for X with bias that is just X with a leading 1 in the first element. This new version of X can then be used any place where the normal X and the weight vector interact so in the net input function, when recalculating the value of the weights and when doing the predict with a new X. In the case with predict, I remake the X adding the leading one because in predict it is likely that the new values are change, so X is different.

The two different versions (bias absorbed and normal) give us the same value We can see this because our weight vector is now $W = w_0, w_1, \cdots, w_m$ where w_0 is the bias value. We do the dot product of the W and X we still get the same result because we get the equation $W \cdot X = (1 \cdot w_0) + (w_1 \cdot x_1) + \cdots$ we see that the dot product with give us 1 times w_0 and we know that w_0 is our bias. So, this is equal to $b + \sum_{j=1}^{m} w_j x_j$ and this is the original formula for the model. Doing this allows us to update the bias by the same gradient descent model that is used to update all of the other weights. This also saves computation time because now we only have to do a single operation of $W \cdot X$ when before it was $W \cdot X + b$.

2 Task 2

The test that I did for this was that the learning rate was 0.01 and I used 500 Epochs. This was a good balance between having too large of a learning rate and having too many epochs. I ran two different tests the first shown in figure 1 is the Iris data set with both Adaline and logistic regression. The second shown in figure 2 is a graph of the wine dataset. One thing to point out with the comparison of the two types of machine learning models is that Adaline uses the mean squared error and logistic regression uses Log loss. The data for both was also standardized.

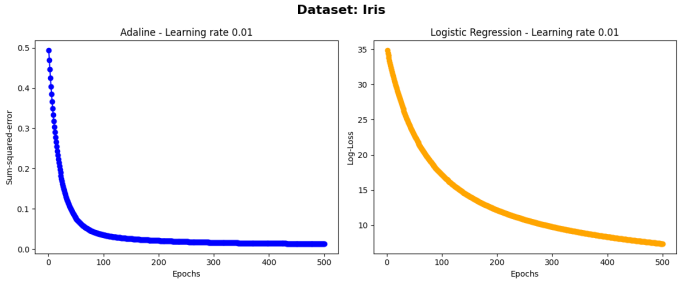

Figure 1: Iris Dataset Error Graph

In figure 1 we are comparing the Iris dataset using Adaline and Logistic Regression. With Adaline we can see the loss value gets to be very low and flatten out after only 100 Epochs and after that point there is not much work to be done so that part is a waste of work. For logistic regression we can see that the values of the loss are very different from Adaline this is because of the scale of the graph. The thing that is important is the shape of the graph. With that we can see that graph is a more gradual curve than Adaline. We can also see that the even after 500 Epochs we still have a large loss. We can see that at the end the slope is still slightly negative so there is still work that needs to be done to maximize the gap between the two classes.

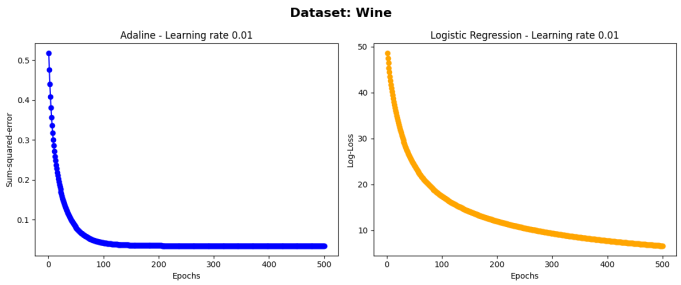

Figure 2: Wine Dataset Error Graph

In figure 2 we see a similar thing that we saw in figure 1. For Adaline we see a steep drop off in the loss and a flattening of the curve after 100 Epochs. This means that after 100 Epochs we see diminishing returns in the reduction of the loss. For Logistic regression we can see that in the beginning the slope is very negative and as the number of Epochs increase we see a reduction in the slope. After the 500th Epoch we still have a fairly negative slope so the number of Epochs or learning rate should be increased.

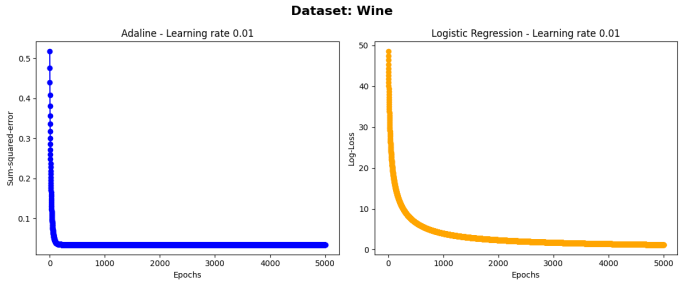

Figure 3: Wine Dataset Error Graph with increased Epochs

We can see in both of the data sets tested that Adaline has almost found the optimal separation between the two classes after only 100 Epochs. By comparison with Logistic regression after 500 Epochs we still have a negative slope so we either need to increase the learning rate or increase the number of Epochs we use. in Figure 3 we can see with an increased number of Epochs up to 5000 we finally see the slope get closer to zero than we did with 500 Epochs but with this number of Epochs it is very hard to see the slope of Adaline. In Adaline it is almost a straight line down to zero the a flat line down from there to 5000 Epochs. This exaggerates the effect of showing that Adaline does not need that number of Epochs when we have a higher learning rate. The graph not shown is the graph of Iris with the increased Epochs. In this graph still after over 5000 iterations we still do not have a flat line at the bottom of the graph while Adaline has been done for a long time.

The reason we see this difference is because that Adaline works very well when sets of data are linearly separable very clearly. In both wine and Iris data sets the data is linearly separable so Adaline works very well for this data. If the data were not as clearly linearly separable then we would not see the great performance that we see from Adaline. Another reason we see this difference so much is how the losses are being calculated. With Adaline we use mean squared error and this leads to when we see that the weights are close to the value needed the error because very small because of the square. With Logistic regression it uses a sigmoid function so even when the model correctly classifies the points it will keep updating the weights to push the values closer to 0 or 1. This results in the long curve that we see in the figures above leading to an increased number of epochs to get to the maximum values even when that might not be necessary.

3 Task 3

The way in which I developed the multiclass classification using Perceptron was by using hierarchical classification. Hierarchical classification is splitting the groups into different sets. The first model is used to classify if it is group 1 or something else. The second model is used to separate the class 1 and 2 from each other. The second model is only trained using class 1 and 2 while the first model has all of the data but to it class 1 and 2 are the same thing (or not class 0). That is how the data is trained using fit. When we go to predict data with a value we give it the value or values like we normally do. We first use model 1 to check to see the value is with class 0 or either class 1 or 2. If it is class 0 then we say it is normally. If it is not class 0 then we have to use the second model to see if it is class 1 or 2.

The other option that I found for multiclass classification using Perceptron was by using a one vs rest idea. The idea is that you have a model for each class of data and the values in that model are positive for that class and negative for all other ones. The way this is used to predict a value is when we get a new data point x we do not know what model would be best so we test all of the models and which ever one gives the highest value is the model it belongs to.

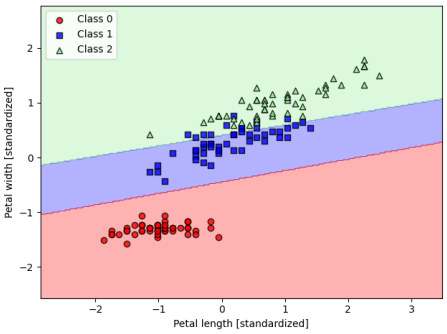

Figure 4: Hierarchical classification

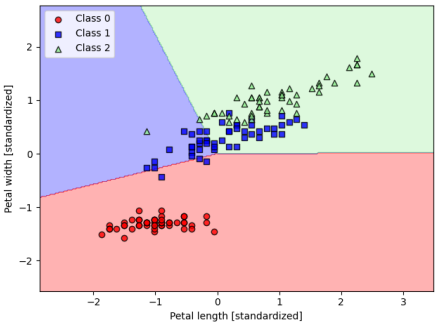

Figure 5: One V.S Rest Classification

The reason that I did hierarchical classification over one vs rest was because of how the data is structured and how the they compared to each other. I had originally done one vs rest but after looking at the output of the predication model and how it was graphed it did not look like it would be good. The original had the line for class 0 right next to classes 1 and 2 and the the line separating classes 1 and 2 was vertical. One-vs-rest forces each classifier to separate one class from the combination of the other two, which can lead to suboptimal boundaries when two classes are highly overlapping. When using hierarchical classification I was able to get it where the line reduced the number of miss classification by making it so the line would be either vertical or horizontal if needed. In this case we have a diagonal line that best fits the data. This change in the two versions can be seen in figures 4 and 5. This conflict in the

two types of models could be because of the learning rate and the number of iterations. When changing these hyperparameter it changes how the graphs will look and in turn how to model predicts any given value. After trying to get the One Vs Rest model to work I was unable to get it to look similar to when running the same data through a know library like sklearn. This suggested that my implementation or hyperparameter may not have been optimal, so I experimented with hierarchical classification instead. With some data using a one vs rest model might work well but in this case with the Iris dataset it did not achieve the accuracy that I was looking for.

4 Task 4

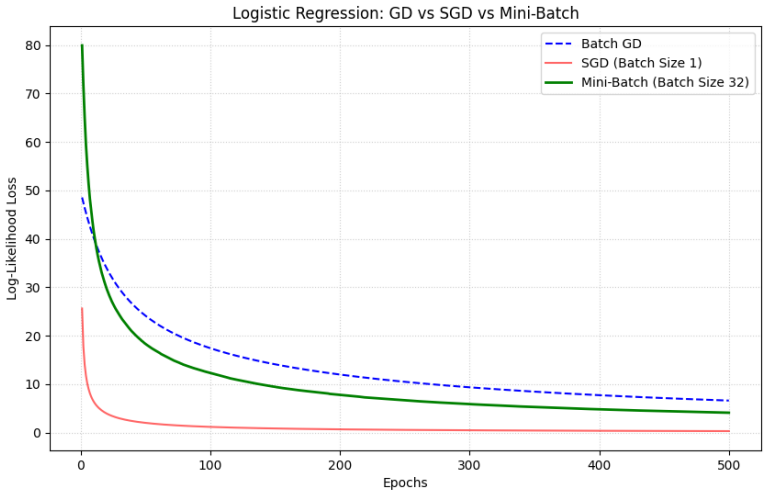

Figure 6: GD vs SGD vs Mini-Batch

The different versions of Logistic regression each have some tradeoffs that need to be accounted for. Shown in figure 6 we can see the loss calculations found for the three versions of Logistic regression. The first Logistic regression version is Batch. That means that all samples are used to update the weights during each epoch. The second type is pure SGD. Stochastic gradient descent (SGD) randomly selects a data point and uses that to update the weight. In our case it selects a single data point and updates the weights and does this for all data points for a single epochs. The last type is mini-batch SGD. This selects a batch of a given number of data points and uses those samples to update the weight. In our case we are using a batch size of 32 so it will select 32 random elements. It will use these elements to update the weights and then select the next elements until all of them have been seen once for one epoch.

Based on the graph that was produced when running the Wine dataset with a learning rate of 0.01 and using 500 Epochs we can see the loss convergence speed. Based on the graph we can see that the version with the best convergence speed is SGD. We know this because it reaches the lowest values of loss the fastest. It starts out with the steepest slope at the start so it lowers the loss very fast. The next best is the mini-batch at the end of the 500 epochs it is in the middle of the three versions of logistic regression. It starts with the highest loss but it passes batch GD relatively quickly. The worst version we see is the normal gradient descent. With batch GD we see it starts out better than mini-batch but as it gets going the slope becomes less before it reaches the bottom of the graph.

For the time comparison of each of the versions I used a timer that was started and stopped during the call to the fit functions. This allowed me to see in real time how long each of the models take in comparison to each other using the same data. After running the test the following times were given as output in table 1. This shows that in this case the fastest version was the Batch GD.

Algorithm	**Time (s)**
Batch GD	0.0272
SGD	3.4440
Mini-Batch	0.2589

Table 1: Performance Comparison of Gradient Descent Variants

The reason that the Batch GD was the fastest is because the way it was written and the size of the data. In our case it is written with the use of optimized linear algebra operations we also only perform 500 total operations it completes the fastest. The second best is mini-batch as we can see from the table. It performs more operations per epoch because of the amount of data that is used for each update of the weight. In Batch it uses all data but in mini batch it uses a subset of the data in our case 32 elements so to use all of the elements it needs to complete multiple iterations. This is why the SGD is the worst because it needs to do so many operations per epoch.

Given this information we can see that there are tradeoffs that need to be made when picking a model. If you are looking for speed then then the best is Batch but this also might not be the best because we would need to increase the number of epochs to decrease the losses. If you are looking for the one with the highest converge speed then you would use SGD. The best of both worlds is mini-batch and that is what the goal of mini-batch is.