Akhil Chilumuru
UIN: 633002192
Sep 11, 2022

CSCE 636: Deep Learning
Assignment 1

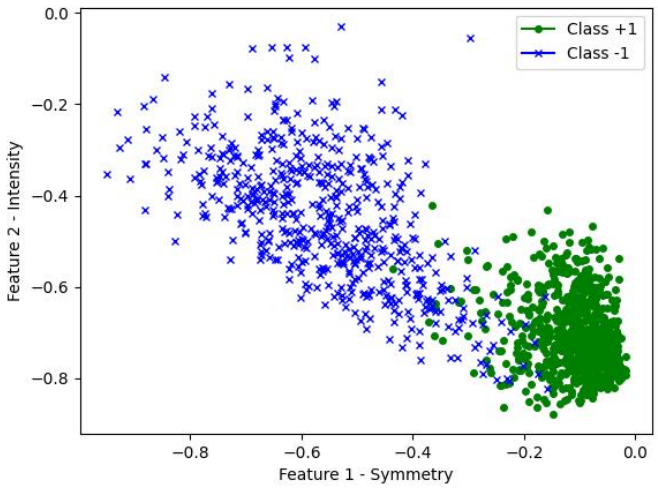
1 Answers to the non-programming part

1. (a) **Explain what the function train_valid_split does and why we need this step.**
The function *train_valid_split* divides the given dataset into train set and validation set. The split index value is 2300, meaning that indexes 0 to 2299 belong to the train set and the rest belong to the validation set. The reason we split the data is that we need to tune the hyperparameters of the model based on the validation data. We cannot evaluate the model on the data that it has been trained on. So, we compare different models using the performance on validation data.

   (b) **Before testing, is it correct to re-train the model on the whole training set? Explain your answer.**
Generally during training, the more data the model sees, the better the training accuracy would be. But that doesn't necessarily mean an improvement in test accuracy. Because it could run the risk of over-fitting to the training data. There is no guarantee that the model will perform better on unseen data. Moreover, since our training split size (1350) is sufficiently large for the number of parameters (3) we have, there is no need to re-train on the whole training set.

   (d) **In the function prepare X, there is a third feature which is always 1. Explain why we need it.**
In the function *prepare_X*, we add a feature with constant 1 because that is the bias term. The bias term represents *no dependence on any input feature*. For example, if you consider the output to be constant regardless of any input, then without bias term it is not possible to model it.

   (f) **Test your code in "code/main.py" and visualize the training data from class 1 and 2 by implementing the function visualize_features. The visualization should not include the third feature. Therefore it is a 2-D scatter plot. Include the figure in your submission.**
We plot the two features as a 2D scatter plot:



2. (a) **Write the loss function E(w) for one training data sample (x,y).**

The loss function considering a single data point (X, y) would be as follows:

$$E(w) = ln(1 + e^{-y\mathbf{w^T x}})$$

(b) **Compute the gradient $\nabla E(w)$. Please provide intermediate steps of derivation.**

$$\begin{aligned}
\nabla E(w) &= \frac{\partial}{\partial \mathbf{w}} \ln\left(1 + e^{-y\mathbf{w^T x}}\right) \\
&= \frac{1}{1 + e^{-y\mathbf{w^T x}}} \frac{\partial}{\partial \mathbf{w}} e^{-y\mathbf{w^T x}} \\
&= \frac{1}{1 + e^{-y\mathbf{w^T x}}} e^{-y\mathbf{w^T x}} \frac{\partial}{\partial \mathbf{w}} (-y\mathbf{w^T x}) \\
&= \frac{1}{1 + e^{y\mathbf{w^T x}}} (-y\mathbf{x}) \\
Therefore, \nabla E(w) &= \frac{-y\mathbf{x}}{1 + e^{y\mathbf{w^T x}}}
\end{aligned}$$

(c) **When will we need to use the sigmoid function in prediction?**

The Sigmoid function is a strictly increasing function i.e, monotonic. Because of this nature, the prediction criteria can be effectively written as a linear decision boundary as follows for positive class:

$$\theta(w^T x) >= 0.5 \iff w^T x >= 0$$

and similarly for negative class:

$$\theta(w^T x) < 0.5 \iff w^T x < 0$$

Therefore, during the inference stage, it is computationally inefficient to make predictions by calculating the sigmoid of the linear transformation $\theta(w^T x)$. Instead, we could just stop at the linear transformation $w^T x$ and make predictions.

However, we may still need to use the sigmoid function if we either (1) care about the confidence of our predictions like we do during the training phase or (2) want to compute metrics such as the area under the ROC curve to determine the proper threshold for classification.

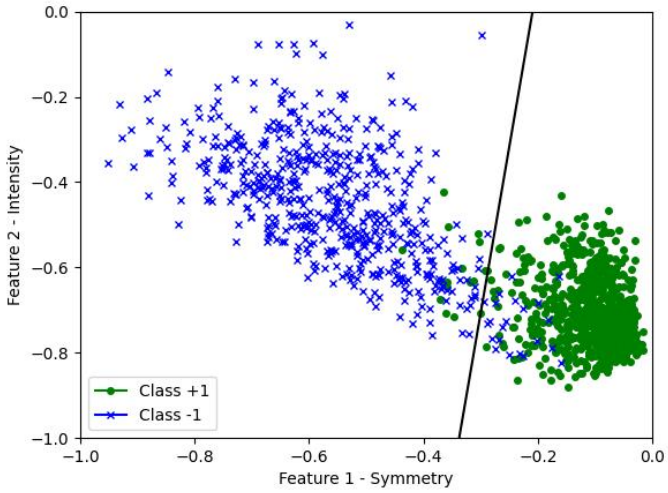
(d) **Is the decision boundary still linear if the prediction rule is changed to the following? Justify briefly.**

Technically it is a linear model even with a high threshold because even with a threshold of 0.9, sigmoid is still monotonic and there exists a unique mapping between input $w^T x$ and threshold 0.9. The decision boundary will then be $w^T x >= inverse\_sigmoid(0.9)$ for positive class and $w^T x < inverse\_sigmoid(0.9)$ for negative class.

(e) **In light of your answers to the above two questions, what is the essential property of logistic regression that results in the linear decision boundary?**

The essential property of logistic regression that results in a linear decision boundary is the sigmoid loss function which is a strictly monotonic function.

3. (d) **Test your code in *"code/main.py"* and visualize the results after training by using the function *visualize_results*. Include the figure in your submission.**



(e) **Implement the testing process and report the test accuracy of your best logistic regression model.**
The test accuracy of the best logistic regression model is 93.07%.
The training accuracy of the best logistic regression model is 97.25%.
The validation accuracy of the best logistic regression model is 97.88%.

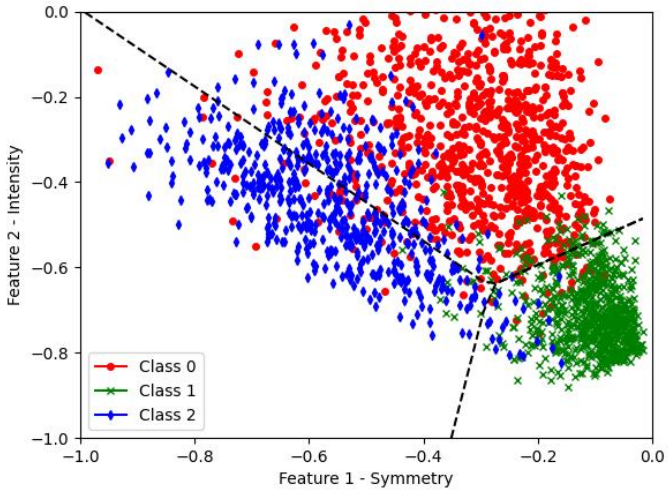
The best model was obtained by setting a learning rate of 0.6 and training with mini-batches of size 10 for up to 500 epochs.

4. (a) **Based on the course notes, implement the function _gradient.**

$$\frac{\partial E}{\partial \mathbf{w_k}} = (p_k - y_k)\mathbf{x}$$

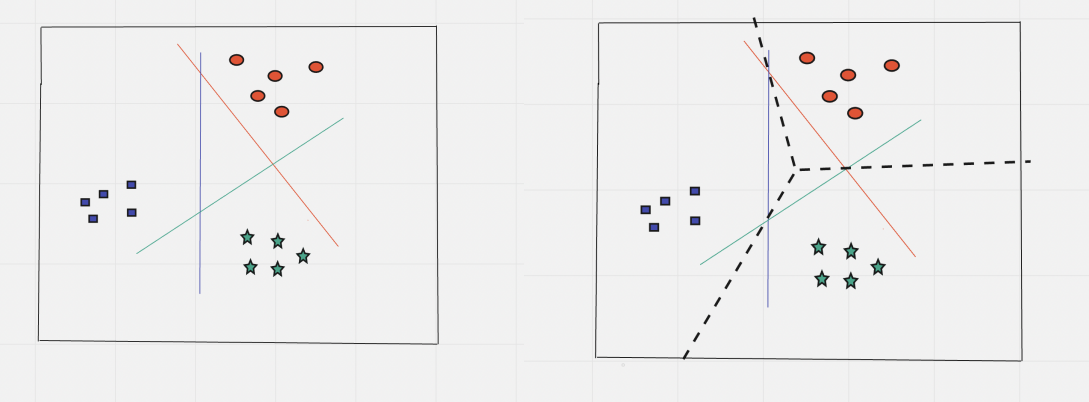
where w_k is the weight vector for class k, p_k is the output probability, y_k is the k^{th} component of the one-hot vector

(d) **Test your code in *"code/main.py"* and visualize the results after training by using the function *visualize_results_multi*. Include the figure in your submission.**



Although the decision boundary looks quite simple in the plot, computing it is not straightforward. In the softmax classifier, we have a one vs all classification rule. Ideally, we should have k decision boundaries for k classifiers, with k=3 in this case. They do not necessarily have to converge.

For example, consider the following data and the decision boundaries:



(1)

The image on the left shows the one–vs–all decision boundaries of the classes. The image on the right shows, the fused decision boundary of all three classes. The idea behind fused decision boundary is that, a point is assigned to a class that has the farthest distance from the one–vs–all boundary to the point. The fused boundary is actually the angular bisector of the original one–vs–all boundary. The plotting equations used in the code are the angular bisector computations.

(e) **Implement the testing process and report the test accuracy of your best logistic regression model.**

The test accuracy of the best multi-class logistic regression model is 87.08%.
The training accuracy of the best multi-class logistic regression model is 89.52%.
The validation accuracy of the best multi-class logistic regression model is 87.46%.

The best model was obtained by setting a learning rate of 0.3 and training with mini-batches of size 10 for up to 500 epochs.

5. (a) **Train the softmax logistic classifier and the sigmoid logistic classifier using the same data until convergence. Compare these two classifiers and report your observations and insights.**

After training both softmax and logistic classifiers with same $learning\_rate = 0.5$ and convergence critera as $max\_iter = 10000$, both the classifiers achieve about the same accuracy of 93.07% on the test data.

```
softmax classififer acc on train data with 2 classes = 0.9718518518518519
softmax classififer acc on validation data with 2 classes = 0.9788359788359788
softmax classififer acc on test data with 2 classes = 0.9307359307359307

logistic classififer acc on train data with 2 classes = 0.9718518518518519
logisitic classififer acc on validation data with 2 classes = 0.9761904761904762
logistic classififer acc on test data with 2 classes = 0.9307359307359307
```

The weights of both the classifiers also roughly obey the equation $w2 - w1 = w$, where $[w1, w2]$ are the weights of the softmax classifier and w is the weights of the logistic classifier.

```
softmax classifier weights= [[ -5.17104689   5.17104689]
                             [-15.35632706  15.35632706]
                             [ -1.10162941   1.10162941]]

logistic classifier weights= [10.27898314 30.6065897   2.21038575]
```

(b) **Explore the training of these two classifiers and monitor the graidents/weights. How can we set the learning rates so that $w1 - w2 = w$ holds for all training steps?**
Let us first analytically derive relation between the two learning rates.

Consider $w_s(t) = [w_s1(t), w_s2(t)]$ to be the weights of the softmax classifier at time t (or iteration). Also, consider $w_l(t)$ to be the weights of the logistic classifier at the same time t.

Assume that the relation $w_s1(t) - w_s2(t) = w_l(t)$ holds at time t.

Now, we want to find out the relation between learning rates of softmax (η_s) and logistic (η_l) classifiers such that the equation holds at time $t + 1$.

In other words, what should the relation between η_s and η_l be so that $w_s1(t+1) - w_s2(t+1) = w_l(t+1)$ holds.

First, we expand each individual term in the equation.

$$w_s(t+1) = w_s(t) - \eta_s * \frac{\partial E(w)}{\partial w_s(t)} \text{ where } E(w) \text{ is the cross-entropy loss function}$$

Now, in problem 2(a) we already evaluated $\nabla E(w)$ to be $[(p1 - y1) * \vec{x}, (p2 - y2) * \vec{x}]$ where $\vec{x}$ is a data sample, $p = [p1, p2]$ are the predicted probabilities of classes 1,2 and $y = [y1, y2]$ are the ground truth one hot vector components.

If we consider our data sample to be of class 0, then $y1 = 1$ and $y2 = 0$;

Now,

$$\begin{aligned}\frac{\partial E(w)}{\partial w_s(t)} &= [(p1 - y1) * \vec{x}, (p2 - y2) * \vec{x}] \\ &= [(p1 - 1)\vec{x}, (p2)\vec{x}] \\ &= [(p1 - 1)\vec{x}, (1 - p1)\vec{x}]\end{aligned}$$

Which means,

$$w_s(t+1) = w_s(t) - \eta_s * [(p1 - 1)\vec{x}, (1 - p1)\vec{x}]$$
$$[w_s1(t+1), w_s2(t+1)] = [w_s1(t) - w_s2(t)] - \eta_s * [(p1 - 1)\vec{x}, (1 - p1)\vec{x}]$$

and,

$$\begin{aligned}
w_s1(t+1) - w_s2(t+1) &= w_s1(t) - w_s2(t) - \eta_s * (p1-1)\vec{x} + \eta_s * (1-p1)\vec{x} \\
&= w_s1(t) - w_s2(t) - 2 * \eta_s * (p1-1)\vec{x} \\
&= w_l(t) - 2 * \eta_s * (p1-1)\vec{x} \\
&= w_l(t) - 2 * \eta_s * (\frac{e^{w_s1(t)^T x}}{e^{w_s1(t)^T x} + e^{w_s2(t)^T x}} - 1)\vec{x} \\
&= w_l(t) - 2 * \eta_s * (\frac{-e^{w_s2(t)^T x}}{e^{w_s1(t)^T x} + e^{w_s2(t)^T x}})\vec{x} \\
&= w_l(t) - 2 * \eta_s * (\frac{-1}{e^{(w_s1(t) - w_s2(t))^T x} + 1})\vec{x} \\
&= w_l(t) - (2 * \eta_s) * (-1) * (\frac{1}{1 + e^{w_l(t)^T x}})\vec{x}
\end{aligned}$$

In logistic regression, class 0 has $y = -1$. Now, if you notice the weight update rule for logistic regression, we have:

$$\begin{aligned}
w_l(t+1) &= w_l(t) - \eta_l * \frac{\partial E(w)}{\partial w_l(t)} \\
&= w_l(t) - \eta_l * y * (\frac{1}{1 + e^{w_l(t)^T x}})\vec{x} \\
&= w_l(t) - \eta_l * (-1) * (\frac{1}{1 + e^{w_l(t)^T x}})\vec{x}
\end{aligned}$$

Therefore, in order to have $w_s1(t+1) - w_s2(t+1) = w_l(t+1)$, we need to have $\eta_l = 2 * \eta_s$

We now try to empirically verify the same result. Setting the learning rate for softmax as 0.1 and logistic as 0.2, we get

```
after epoch=0, logistic weights=[ 0.21372375  2.23904578 -1.40672694]
after epoch=0, softmax weights=[[-0.09026854  0.09026854]
                                [-1.12503803  1.12503803]
                                [ 0.69229899 -0.69229899]]

after epoch=1, logistic weights=[ 0.25706493  3.72335692 -2.19907967]
after epoch=1, softmax weights=[[-0.13287781  0.13287781]
                                [-1.86416451  1.86416451]
                                [ 1.10207212 -1.10207212]]
```

Notice that the equation $w2 - w1 = w$ roughly holds after each training stage above.

2 Results and Analysis of the programming part

1. **Sigmoid Logistic regression**

The sigmoid logistic regression model was initially trained with all three gradient descent algorithms. Here's a summary of the training accuracy's of each algorithm when trained with $learning\_rate = 0.5$ and $max\_iter = 100$:

algorithm	train accuracy	parameters
Batch GD	0.94444	[0.18761413 3.55944904 -2.06327285]
mini Batch GD batch size=full	0.94444	[0.18761413 3.55944904 -2.06327285]
Stochastic GD	0.97259	[10.65600751 32.03642942 2.32110916]
mini Batch GD batch size=1	0.96814	[11.2914667 31.68848348 1.47082129]
mini Batch GD batch size=10	0.97259	[4.98803971 23.59239049 -3.07969263]

The above results show that mini batch gradient descent is a blend of stochastic and batch gradient descent. In mini batch GD, on one extreme we have stochastic GD when $batch\_size = 1$ and on the other extreme we have the batch GD when $batch\_size = full$. Based on these results, I have chosen to train with mini batch GD with $batch\_size = 10$.
Now, we evaluate the model with different hyper parameter values. The hyper parameter we have is learning rate. The following are the results when experimenting with $batch\_size = 10$, $max\_iter = 500$ and different learning rates:

learning_rate	**validation accuracy**
0.01	0.9735
0.03	0.9761
0.06	0.9761
0.09	0.9761
0.3	0.9788
0.6	0.9788
0.9	0.9761

The best model is with *learning_rate* either 0.3 or 0.6. Either of them is fine because they both have the same validation accuracy. Using the best model, when we run inference on the test data, the test accuracy obtained was 93.07%.

As it appears, the model seems to perform well on unseen data in the validation set, but it drops in performance on test data. This indicates that the training data is perhaps not representative of the test data. In other words, I suspect that the test data might not be from the same distribution as the train data.

2. **Softmax Logistic regression**

The softmax logistic regression model was initially trained with mini batch gradient descent algorithm with a $batch\_size = 10$, $learning\_rate = 0.5$, $max\_iter = 100$ and number of classes $k = 3$. The following was the result:

training accuracy	**parameters**
0.8952	[[6.77018021 -1.03477425 -5.73540597] [0.23562711 15.710986 -15.94661311] [10.39851542 -8.55965185 -1.83886357]]

Now, we evaluate the model with different hyperparameter values. The hyperparameter we have is the learning rate. The following are the results when experimenting with $batch\_size = 10$, $max\_iter = 500$ and different learning rates:

learning_rate	**validation accuracy**
0.01	0.8714
0.03	0.8746
0.09	0.8714
0.3	0.8746
0.6	0.8714
0.9	0.8682

The best model is with *learning_rate* either 0.3 or 0.03. Using the best model, when we run inference on the test data, the test accuracy obtained was 87.08%.

Although the validation and test accuracies are consistent, the train accuracy itself is bad at 89%. Therefore, to improve the training accuracy we either need to (1) have better input features i.e, perhaps symmetry and intensity are not sufficient to separate all the 3 classes or (2) introduce non-linearity into the model by using complex architectures such as neural networks or SVM with a non-linear kernel.