Predict Diabetes Using Perceptron Algorithm

Lipin Guo
The University of Adelaide
Australia
a1865281@adelaide.edu.au

Abstract

Our task is to use perceptron to predict diabetes with provided diabetes dataset. The base model has 59.74% accuracy. After experiments, accuracy reaches to 74.03%. The performance increases 14.29% of accuracy and the model trained faster.

1. Introduction

The main task is to use perceptron to predict diabetes, which is one of the typical binary classification problems to determine if a patient has the disease or not.

1.1. Dataset

The Pima Indians Diabetes dataset is a small dataset, only has 768 examples in total, consisting of 8 medical predictor variables (features) and 1 target variable. The medical predictor variables include pregnancies the patient has had, glucose, blood pressure, skin thickness, insulin level, BMI, diabetes pedigree, and age [1].

This dataset has 268 diabetic (34.9%) and 500 healthy patients (65.1%) as outcomes [2]. It means the dataset is imbalanced as the target class has an uneven distribution of observations.

In this paper, we use pre-processed diabetes dataset (scaled) [3]. The scaled data makes training faster and prevents the optimization from getting stuck in local optimal [4]. It saves our time to perform data augmentation. From Figure 1.1[5], we can see that the histogram on the diagonal show the distribution of a single predictor variable while the scatter plots on the upper and lower triangles show the relationship of variables. We noticed that the dataset is not linear separable.

The target variable in first column of the dataset had already been labeled as +1 or -1. We can split the dataset to get inputs and y outputs/labels. By splitting the dataset according to a ratio of 6:2:2, there are 460 training, 154 validation, and 154 test examples.

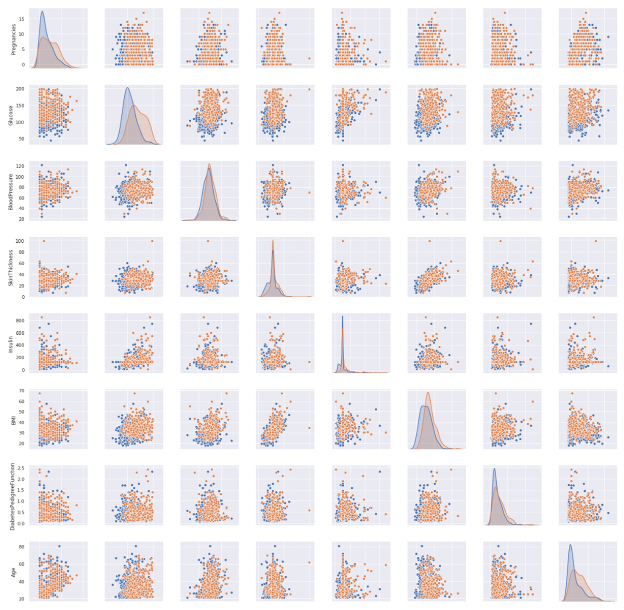

Figure 1.1: Pair Plot of predictor variables for clean data.

1.2. Perceptron

Although there are several methods to perform binary classification task, such has SVM and logistic regression [6], we use perceptron algorithm which was invented in 1943 by McCulloch and Pitts [7]. 'Perceptron is an algorithm for supervised learning of single layer binary linear classifiers [8].' In the context of neural networks, perceptron is a neural network link that contain computations used to track features and use artificial intelligence in the input data. The nerve is connected to artificial neurons using simple logic gates with binary outputs. In contrast to biological neurons, artificial neurons invoke mathematical functions and have nodes, inputs, weights, and outputs equivalent to nuclei, dendrites, synapses, and axons, respectively [9]. There are two kinds of perceptron, single-layer and multi-layer perceptron.

The aim of this paper is to build a single-layer perceptron model based on perceptron algorithm. The single-layer

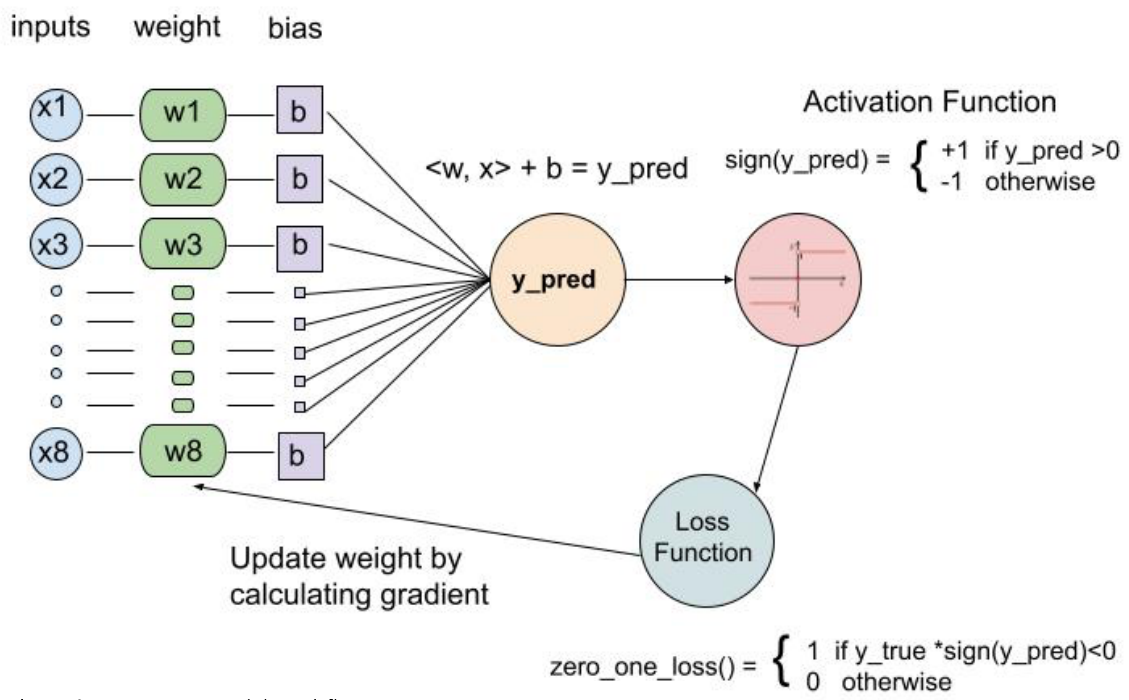
inputs
weight
bias
x1
x2
x3
x8
w1
w2
w3
w8
b
<w, x> + b = y_pred
y_pred
Activation Function
sign(y_pred) = { +1 if y_pred >0 ; -1 otherwise
Loss Function
Update weight by calculating gradient
zero_one_loss() = { 1 if y_true *sign(y_pred)<0 ; 0 otherwise

Figure 2: Perceptron model workflow.

perceptron can only learn linearly separable patterns. As the diabetes dataset is not linear separable, we assume that our perceptron model may not have a high performance. In further part, we will describe methodology of perceptron.

2. Methodology

The model we built is very simple. From the shape of training dataset, we know that each example contains 8 features, which can be represented as a vector **x** = [x1, x2, … , x8]. And we would like to use these inputs x1, x2, …, x8 to produce a binary output +1 or -1.

From Figure 2, we can see that at the beginning, the model gets a prediction result from a linear function y_pred = <**w**, **x**>+b. Then, put the prediction output to a activation function (in our model is sign function) to extract the sign. After that, the loss function compares the sign(y_pred) with y_true to get the loss/error. After using the loss to calculate gradient, we update weight.

2.1. Mathematics

To make binary predictions, perceptron algorithm base on a linear predictor function combining a set of weights, which can be represented as a column vector **w** = [w1, w2, …, w8], with the feature vector **x**, and adding a threshold or a bias b. And the linear function is y_pred = <**w**, **x**> + b, while <**w**, **x**> is the inner product of the summation of **w**i multiplied by **x**i.

2.2. Weights and Bias

Weights are learnable parameters inside the network. Weights affect how much a change in the input affects the output [10]. A lower weight value will make no change to the input, or a larger weight value will change the output more significantly. Before training, we initially set a set of random weights in discrete uniform distribution. In every iteration, the weight coefficient is automatically learned.

Bias b is initialled as 1. Bias represents the distance from their intended value to the predictions.

2.3. Activation Function

The activation function in the base model is sign function shown in Figure 2. If y_pred is greater than 0, return +1 else return -1. This sign function helps us to extracts the sign of the prediction value. It is efficient that separating our data into different hemisphere of the hyperplane. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.

2.4. Loss Function

Loss function computes the distance between the current output of the algorithm and the expected output. It’s a method to evaluate how algorithm models the data. From Figure 2.4 [11], depended on the decision function, different loss function will produce different result. In the base model, we use zero one loss, as Figure 2 shown. When we get correct prediction, it returns 0, otherwise, it returns 1. And we will test perceptron loss and hinge loss in Section 3 Experimental Analysis.

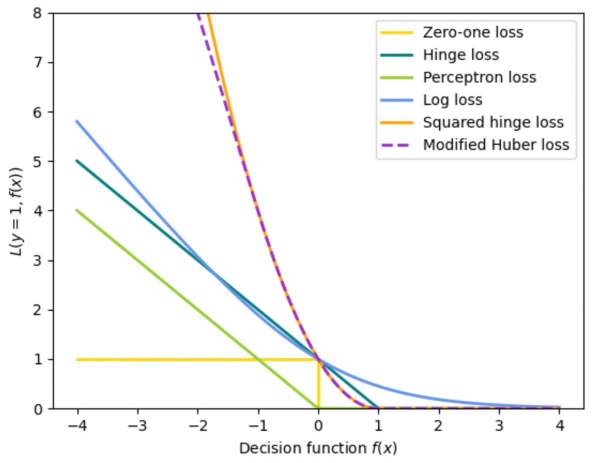

Figure 2.4: Various convex loss function supported by SGDClassifier.

2.5. Learning Rate

The learning rate is a hyperparameter that controls how much the model changes in response to estimation error each time the model weights are updated. Choosing a learning rate is challenging, as too small a value may cause the training process to be too long and may get stuck, while too large a value may result in learning a suboptimal set of weights too quickly or with an unstable training process. We initial 0.0001 as our learning rate.

2.6. Metrics

The outputs are measured by accuracy. Accuracy = number of correct predictions/ total number of predictions. The given accuracy computes the accuracy over the top-3 predictions out of 5 guesses. The higher the accuracy the better the output, while considering the accuracy curve.

3. Experimental Analysis

Initially, the base model has 59.74% test accuracy, with epoch = 20, learning rate = 0.0001, and loss function = zero one loss. It can be served as the benchmark and control model. We mainly have three parameters to tune in our model, including epoch, learning rate and loss function. The following experiments are performed with the base model with different combinations of parameters changed. The accuracy started from 59.74% and increased to 74.03%.

3.1. 1 Trick: Different Epochs

Experiment	Epoch	Accuracy
0	20	59.74%
1	50	74.03%
2	100	74.03%
3	150	74.03%
4	200	74.03%

Table 3.1: Base model, with learning rate = 0.0001, loss function = zero one loss.

From Table 3.1, we can see that the accuracy increases and reaches the highest point 74.03% when the epoch number raises to 50. After then, the accuracy curve converges.

3.2. 2 Trick: Different Epochs and Learning Rate

Experiment	Epoch	Learning rate	Accuracy
0	20	0.0001	59.74%
1	50	0.0001	74.03%
2	20	0.001	74.03%
3	50	0.001	74.03%
4	20	0.01	74.68%
5	50	0.01	75.32%

Table 3.2: Base model, with epoch = 20, loss function = zero one loss.

From Table 3.2, we can see that the learning rate is increasing. Although we got the highest accuracy 75.32% when the learning rate = 0.01, and epoch = 50, the accuracy curve is fluctuating, shown in Figure 3.2, which means the model does not perform well. Even we tuned epoch and learning rate at the same time, this situation doesn't change. To avoid such situation, we need to choose a smaller learning rate, such as 0.001. Compared to accuracy in experiment 2 in Table 3.2, we get the same accuracy of 74.03% when epoch = 20, and learning rate = 0.001, which means the model learns faster with higher learning rate and less iterations. Therefore, it is better to use learning rate = 0.001.

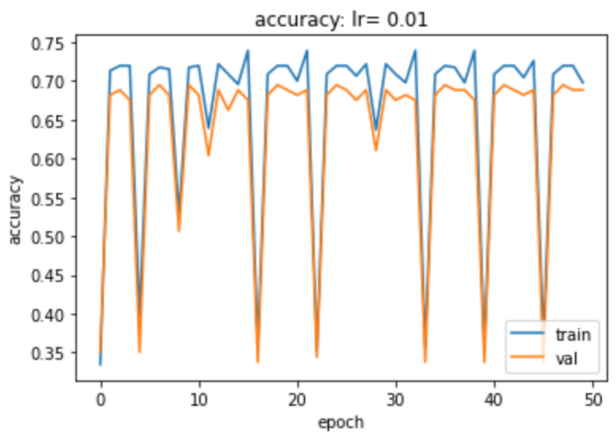

Figure 3.2: Accuracy curve of base model after tuning the learning rate to 0.01.

3.3. 2 Tricks: Different Epochs and Loss Function

Experiment	Epoch	Loss Function	Accuracy
0	20	Zero one loss	59.74%
1	50	Zero one loss	74.03%
2	100	Zero one loss	74.03%
3	20	Perceptron loss	59.74%
4	50	Perceptron loss	74.03%
5	100	Perceptron loss	74.03%
6	20	Hinge loss	74.68%
7	50	Hinge loss	74.03%
8	100	Hinge loss	74.03%

Table 3.3: Base model, with learning rate = 0.0001.

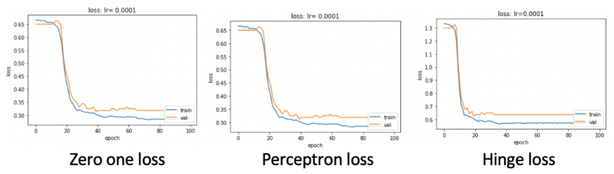

Figure 3.3: Base model, with learning rate = 0.0001, epoch =100

From Table 3.3, we noticed that zero one loss and perceptron loss have the same influence in the accuracy. Under the same condition, hinge loss help to get a higher accuracy 74.68% when the epoch = 20. From Figure 3.3, we can see that zero one loss and perceptron loss have the same loss curve, which decrease dramatically after 20 epochs, but hinge loss decreases dramatically before 20 epochs. From 50 epochs, these three kinds of loss perform similar.

3.4. 2 Tricks: Different Learning Rate and Loss Function

Experiment	Learning Rate	Loss Function	Accuracy
0	0.0001	Zero one loss	59.74%
1	0.001	Zero one loss	74.03%
2	0.01	Zero one loss	74.68%
3	0.0001	Perceptron loss	59.74%
4	0.001	Perceptron loss	74.03%
5	0.01	Perceptron loss	74.68%
6	0.0001	Hinge loss	74.68%
7	0.001	Hinge loss	74.03%
8	0.01	Hinge loss	61.69%

Table 3.4: Base model, with epoch = 20.

From Table 3.4, with the increase of learning rate, we can see that for zero one loss and perceptron loss have the same performance. However, the accuracy of hinge loss decreases when the learning rate increases. When the learning rate = 0.001, these three kinds of loss perform similar. This probably is influenced by the decision function as Figure 2.3 showed. When the learning rate = 0.01, these three kinds of loss have the same issues as we showed in Figure 3.2. This result also proved that it is the wise choice to choose learning rate = 0.001.

3.5. 3 Tricks: Different Epochs, Learning Rate and Loss Function

Experi-ment	Epoch	Learning Rate	Loss Function	Accuracy
0	20	0.0001	Zero one	59.74%
1	20	0.001	Zero one	74.03%
2	50	0.001	Zero one	74.03%
3	20	0.0001	Perceptron	59.74%
4	20	0.001	Perceptron	74.03%
5	50	0.001	Perceptron	74.03%
6	20	0.0001	Hinge loss	74.68%
7	20	0.001	Hinge loss	74.03%
8	50	0.001	Hinge loss	74.03%

Table 3.5: Base model.

If we choose the learning rate = 0.001, these loss function show that 74.03% is the beset accuracy we can achieve. This may be influenced by the non-linear separable dataset. Although when the learning rate = 0.0001, loss function = hinge loss, we can get 74.68% accuracy. The accuracy doesn't rise quickly from 74.03% to 74.68%, but the learning rate is much slower compared to 0.001. We won't change the loss function as previous comparison.

Therefore, after 40 experiments, we consider 74.03% as the best accuracy when the learning rate = 0.001, epoch = 20, loss function = zero one loss. Next, we will evaluate the model using metrics.

3.6. Model Evaluation

Only using accuracy to evaluate the model is not enough as our data is imbalanced (Section 1.1). Here, we use confusion matrix used to visualize the performance of perceptron algorithm.

		Perdition Condition	
	Total 154	Diabetes (PP)	Non-diabetes (PN)
Actual Condition	Diabetes (P) 99	(TP) 71	(FN) 28
	Non-diabetes (N) 55	(FP) 12	(TN) 43

Table 3.6: Confusion matrix for the base model with 74.03% accuracy, using test dataset with 154 examples. P: real positive. N: real negative. PP: prediction positive. PN: prediction negative. TP: true positive. FN: false negative. FP: false positive. TN: true

negative.

From Table 3.6 [12], we can use Precision = TP / (TP+FP) = 85.54% to know our model not to label a negative sample as positive. Recall = TP / (TP+FN) = 71.71%, which means that our model has 71.71% to find the positive samples. Therefore, our perceptron algorithm has a good performance.

3.7. Visualization Results

The graphs we compared here are the base model (before tunning parameters) as the benchmark and base model (after tunning parameters) as the best performer. The baseline test accuracy of base model returns 59.74% with epochs = 20, learning rate = 0.0001, loss function = zero one loss. The test accuracy from best performed base model returns 74.03% with the parameters of epoch = 20, learning rate = 0.001, loss function = zero one loss. As Figure 3.7.1 (b), the accuracy curve is converge.

3.7.1. Training and Validation Accuracy Curve

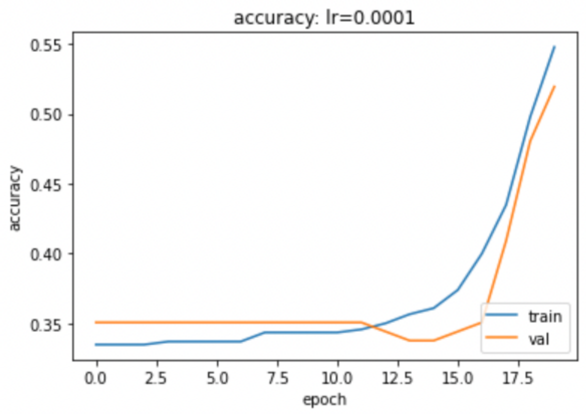

Figure 3.7.1 (a): Accuracy curve of base model (before tunning parameters) with learning rate = 0.0001, loss function = zero one loss.

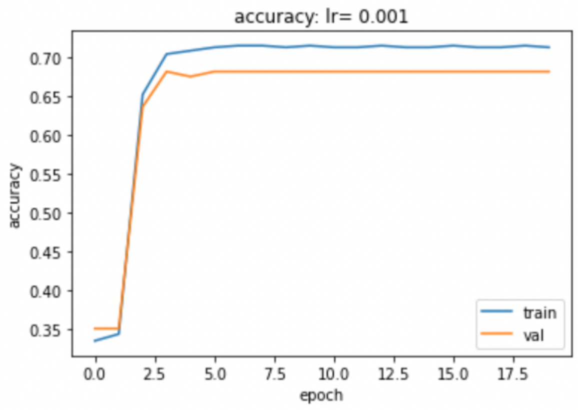

Figure 3.7.1 (b): Accuracy curve of base model (after tunning parameters) with learning rate = 0.001, loss function = zero one loss.

3.7.2. Training and Validation Loss Curve

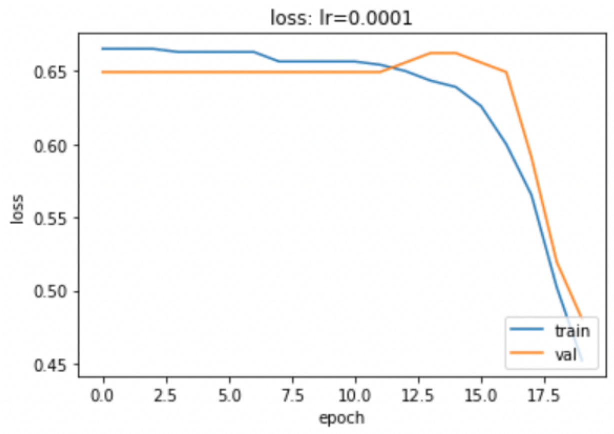

Figure 3.7.2 (a): Loss curve of base model (before tunning parameters) with learning rate = 0.001, loss function = zero one loss.

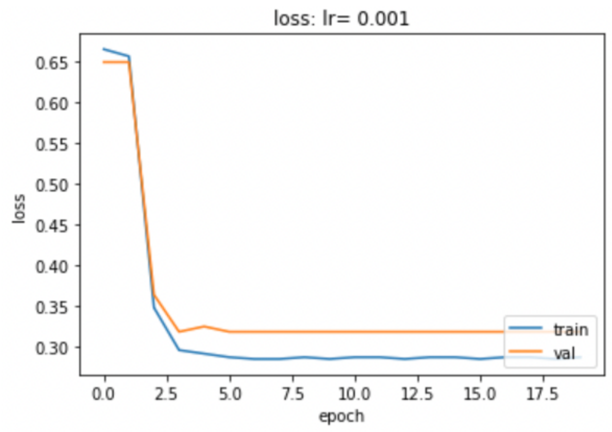

Figure 3.7.2 (b): Loss curve of base model (after tunning parameters) with learning rate = 0.001, loss function = zero one loss.

4. Code

The code detail please refer to https://github.com/CatherineKwok/perceptron.

The code includes several parts:

- Loading data
- Code implementation
- Training and Testing data: base model
- Experiments

5. Conclusion

In conclusion, we create a single-layer perceptron model. The baseline of this single-layer perceptron model is

59.74% accuracy when epoch = 20, learning rate = 0.0001, loss function is zero one loss. After tuning parameters such as epoch, learning rate and loss function, our finial accuracy is 74.03% when epoch = 20, learning rate = 0.001, loss function is zero one loss.

In future, we can do below experiments to increase the performance:

(1) Weight initialization. Weight initialization is important as it help to achieve better optimization, faster convergence, and feasible learning process [13]. We can change weight's distribution from discrete uniform distribution to normal distribution.
(2) Bias initialization. Bias initialization would influence the minimization of loss and decision boundary [14]. We can try different kinds of value to bias.
(3) Architectural change. The limitation of our single-layer perceptron is it can only use to perform binary classification with linear separable data. However, in real world, many classification problems are non-linear. To overcome this, the idea of the single-layer perceptron was expanded by adding some hidden layers between the input and output layer.
(4) Kernel trick. This also help to solve the non-linear classification problem.

References

[1]https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database
[2]https://www.kaggle.com/code/vincentlugat/pima-indians-diabetes-eda-prediction-0-906
[3]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html
[4]https://www.atoti.io/articles/when-to-perform-a-feature-scaling/
[5]https://www.kaggle.com/code/shrutimechlearn/step-by-step-diabetes-classification-knn-detailed
[6] https://en.wikipedia.org/wiki/Binary_classification
[7] McCulloch, W; Pitts, W (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5: 115–133.
[8]https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron
[9]https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron
[10]https://deepai.org/machine-learning-glossary-and-terms/weight-artificial-neural-network
[11]https://scikit-learn.org/dev/auto_examples/linear_model/plot_sgd_loss_functions.html
[12] https://en.wikipedia.org/wiki/Confusion_matrix
[13]https://towardsdatascience.com/why-better-weight-initialization-is-important-in-neural-networks-ff9acf01026d
[14]`https://medium.com/@glenmeyerowitz/bias-initialization-in-a-neural-network-2e5d26fed0f0