Assignment 2 of PRML

Introduction

In this assignment, three requirements are listed below:

- Use least square model to separate a given dataset, and draw the decision line.
- Use the perceptron algorithm to separate a given dataset, and draw the decision line
- Use Logistic Regression to finish a simple text classification task based on softmax function.
  - Implement a preprocess method to turn given documents to one-hot representations, which useless punctuations and blanks cleared.
  - Consider how to compute the differential. **Implement this in vector style.** Also consider: (1). About the regularization. Should we use L2 regularization for the **bias term**. (2). How to check the **correctness** of the trained results?
  - Draw a loss curve. And answer: (1). How to decide the learning rate? (2). How to decide the termination of training.

As the second part of assignment 2 is quite different from the first one, we will not list the detailed tasks and methods here and leave them when needed. And they will be solved case by case. Fake codes for them will be presented when discussed.

Part One

Least Square Model

Least Square Model is a classical classification method, based on the idea that the distance between the predictions and the facts should be as less as possible. And the model recognize the distance but not the truth or not, which means it tries to get close to the facts but not focus on making a nice classification. However, we could classify an input instance by computing its predicted value. For convenience, parameters are listed below:

parameters	**represent**	**shape**
X	the input of the train dataset	N*2
$\hat{X}$	the augment matrix of the dataset	N*(2+1)
y	the target for the train dataset	N*1

parameters	represent	shape
W	the trained parameters	2*1
$\hat{W}$	the augment of W	(1+2)*1

Select the Model

Our separator here is the function:

$$f(x;W,b)=W^Tx+b$$

or say:

$$XW+b=y$$

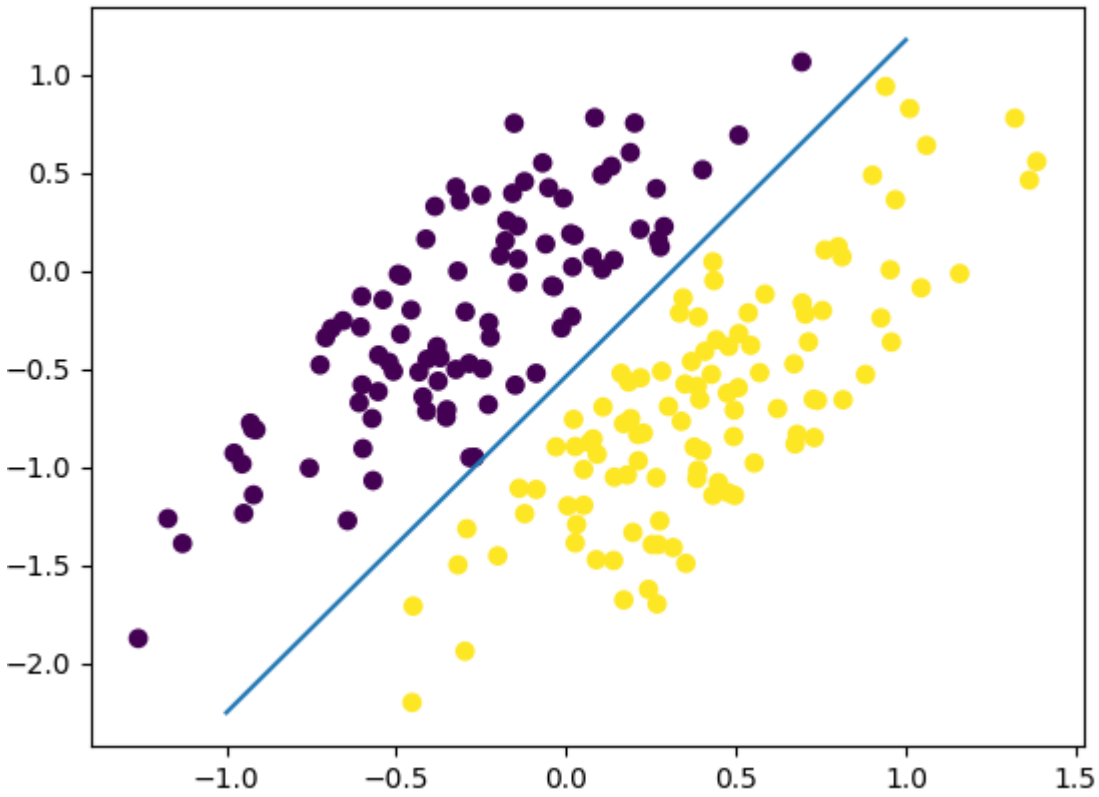
Our target has only 1 or 0, so we assume that when $f>0.5$, it is classified as A, while the other as B. And our goal is to get a suitable W and b to compute f, and predicts the class. For convenience, we use the augment matrix to compute f by adding a constant 1 to all vectors of dataset. So the third parameter in W means b in the function.And the formula can be written as $f(\hat{x})=\hat{W}^T\hat{x}$.

Minimize the Loss

We use Least Square Estimation here, which has an analytical solution:

$$\hat{W}=(\hat{X}^T\hat{X})^{-1}\hat{X}^Ty$$

So, we simply apply this formula with $\hat{X}$ and y as input, the trained parameter will be got immediately. And we could plot the decision line in the dataset plot.



And it turns out to be perfect on my training set with *accuracy=100%* . Besides, the decision line has proper distances with training data.

Perceptron Algorithm

After Least Square Model, we could go on into the Perceptron Algorithm. Perceptron Algorithm is a typical passive learning algorithm. It updates its parameter only when the classifier makes a wrong prediction.

Select the Model

Different from the setting of Least Square Model, we focus on the correctness of the prediction and ignore the process of the prediction data. In other words, the classifier updates its parameter by making predictions and compare the predictions with test data. When a right prediction shows up, we simply ignore it and move on; while for a wrong prediction, we update the parameter.

As we have done in Least Square Model, we have an augment dataset and a parameter matrix to be trained. For convenience, we consider class A as 1 and B as -1. When we use the parameter matrix to make a prediction, the sample will be classified as A when the output is larger than 0. Under this setting, no matter what class it is, a correct prediction always has $y\hat{W}^T\hat{x} > 0$, and a wrong prediction : $y\hat{W}^T\hat{x} \leq 0$. So according to Perception Algorithm, we only updates the value when $y\hat{W}^T\hat{x} \leq 0$.

Update the Parameter

According to discussions above, we can get the Loss function:

$$L(x;\hat{W}) = max(0, -y\hat{W}\hat{x})$$

And we get its differential and get the update formula:

$$\hat{W} = \hat{W} + \alpha y\hat{x}\ I(y\hat{W}^{T}\ \hat{x} \leq 0)$$

α refers to the learning rate parameter, which is a super-parameter justifying the speed of the learning process.

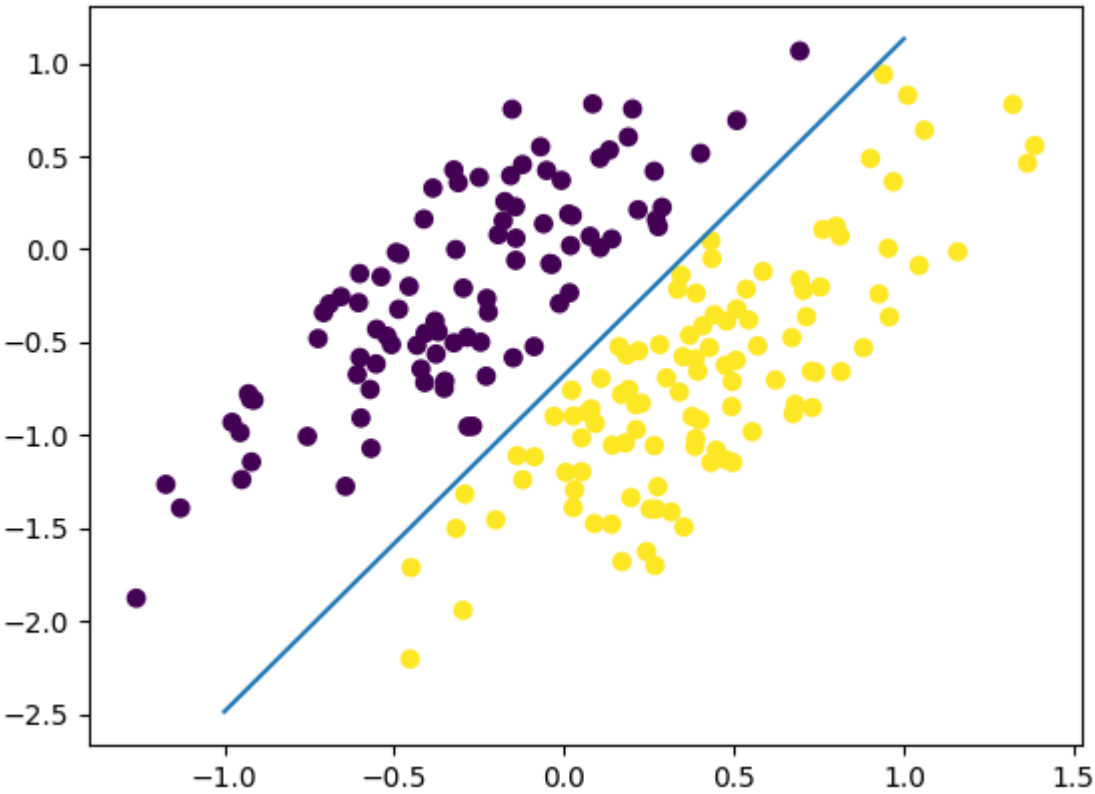
Fake code here:

```
init alpha, W, T
for i = 0 to T do:
    shuffle the dataset:
    for j = 0 to N do:
        get a sample data
        compute the output and make a prediction
        if wrong:
            update the parameter
        else:
            do nothing
        end
    end
end
return W
```

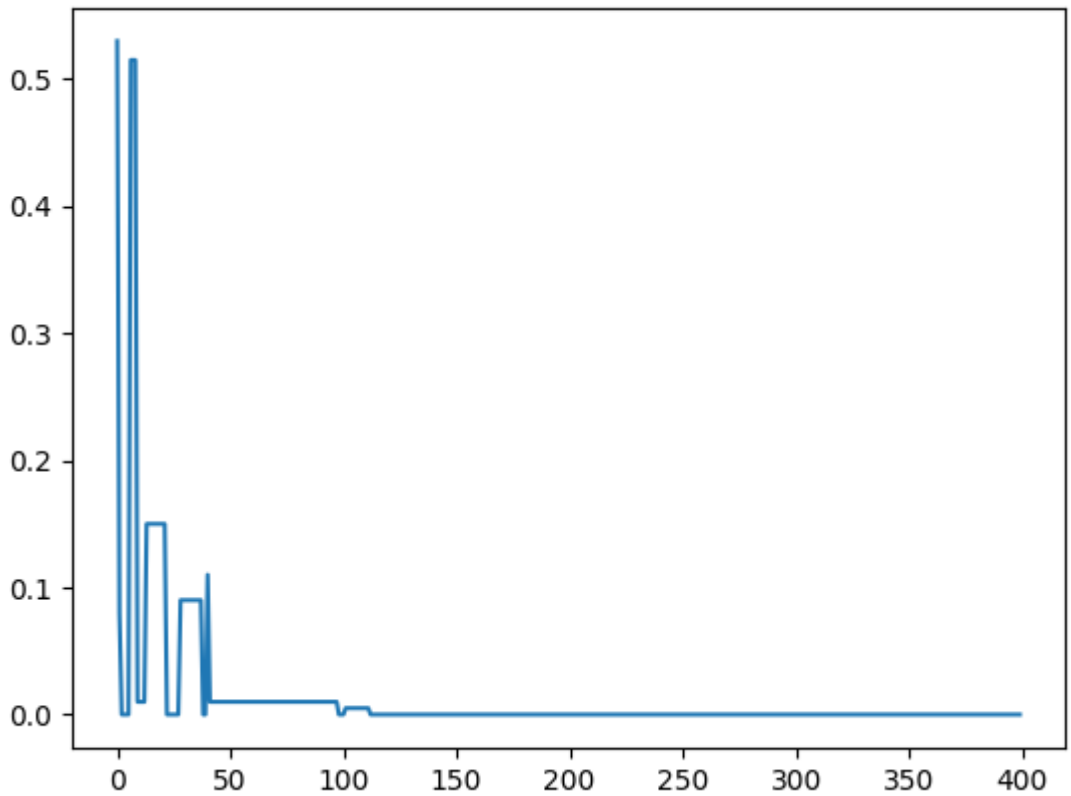
T in the fake code refers to the rounds of training, which will be given a more detailed discussion later.

Discuss on the Result

Before going deep into other discussions, we draw the decision line on the given dataset first to have a general vision of the feature of the algorithm.

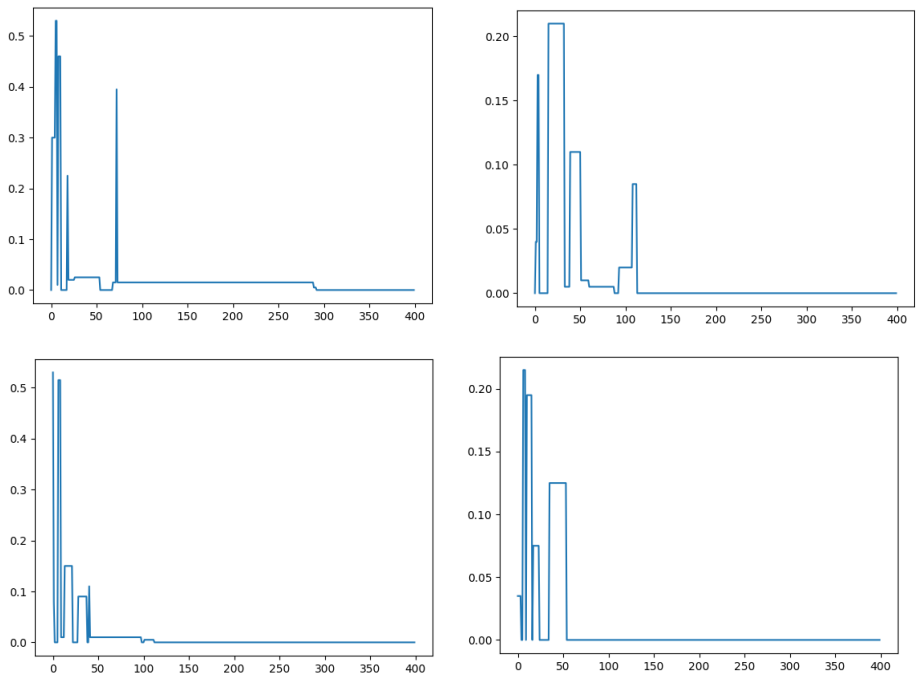


According to Perceptron Convergence Theorem, the perceptron algorithm can always find a solution when a solution exists. We will not focus on the proof of that, but we'd like to have a vision of the process of convergence.

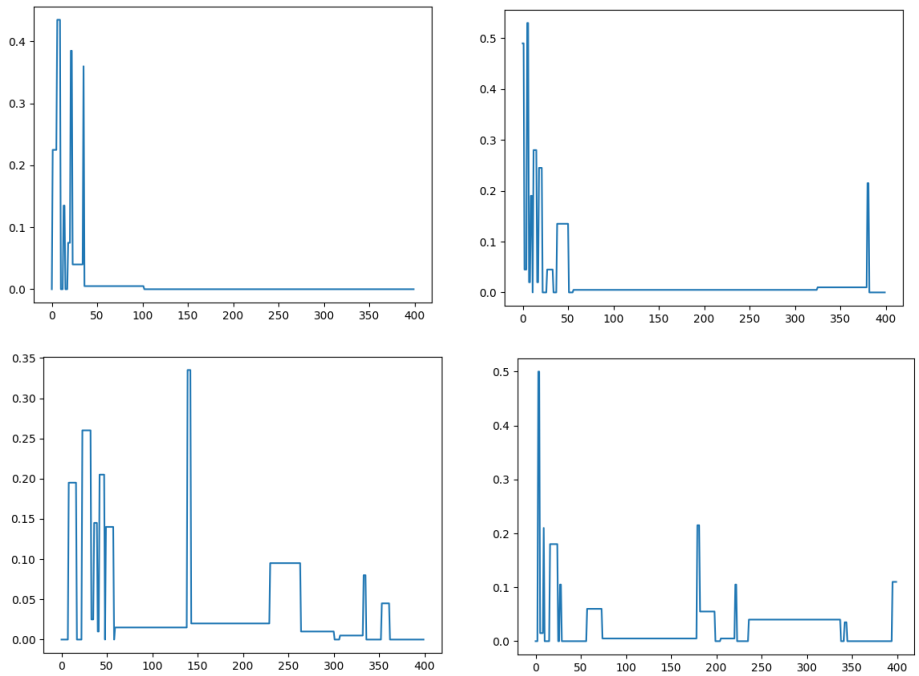


We could see that the accuracy varies greatly. A peak appears when a sample takes the parameter far away from the right solution. But we could also see that the height of the peaks drop as the turns goes up, and finally reaches 0 at about the 110-th turn.

And we have left the discussion of the learning rate. In order to visually understand the effects of learning rate, we now draw the accuracy curves and the loss curves with different learning rate.



We can find out that the larger the learning rate the earlier to get converged. But in fact, this method **does not** show a convincing picture of the procedure. We have another sets of figures with fixed learning rate as 1 which means they are actually the same in every aspects except the random dataset:



So the process of perceptron actually largely depends on the order of training set, which is totally in random order. In fact, it's hard to get an intuitive vision of the learning rate's influences on the convergence speed. But anyway, a quite small or large learning rate actually slows down the rate for it will spend a lot of time to reach a convergence when the step is small, and easily miss the target when it's too large.

Part Two

In this part, we are asked to implement a simple task on text classifying based on the word bag model and Logistic Regression. Several basic requirements will be solved case by case. And the topics in this part will be listed as below.

- How to turn normal documents to one-hot vectors.
- How to implement the derivative the loss function we used to update the parameters.
- The topic about L2 regularization.
- How to validate the trained result, or in other words, the correctness.
- How to choose the learning rate.
- How to decide the determination of training process.
- How about the performance of the normal logistic regression, the stochastic gradient descent and batched gradient descent. Also pros and cons will be discussed.
- Check the result with the test dataset.

As the tasks have a thoughtful consideration of the whole process of the task, we'd like to strictly follow the order of tasks and try to give satisfying explorations of them all. But before that, I will spend some time explaining the model.

Build the Model

For the simple implement of text classification, we will use the bag of words to solve this question. Every document will be mapped into a one-hot (or multi-hot) vector. Using the softmax function:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

And we build the target as a hot-hot matrix, too. We use the parameter to predict the class of the document by choose the largest predicted value as its class. We use the cross entropy loss function to finish the gradient descent. We could also involve the augment in the update function and we get:

$$W = W - \alpha \frac{\partial L}{\partial W}$$

with :

$$L = -\frac{1}{N} \sum y_n log\hat{y}_n + \lambda ||W||^2$$

And we will leave the strategy of normal gradient descent, stochastic gradient descent and batched gradient descent in later part.

Prepossess of Documents

As we get the documents in their original form, they are constructed with punctuations and blanks. And we need to remove all the useless information. Then we need to collect all the words appearing in these documents and build a dictionary which holds all the words in it. As the training set is quite large, we will ignore the words which only appear for 10 times or less and constructs a more concise dictionary. And I will present a simple construct and step of my prepossess of the documents.

build_env()

This is the structure function links the producing of documents, prepossessing of documents and return multi-hot matrix.

```
dataset,testdataset=get-text-classification-datasets()
turn test dataset to one-hot matrix
one-hot-collections, dictionary=build-dict()
X=augment(one-hot-collections).T
Y=target.T
save X and Y in .txt
```

build_dict()

This is the kernel function to convert documents to one-hot matrix.

```
INPUT: a list of documents
OUTPUT: a one-hot matrix and a built dictionary

build vacant words-set list-documents
for paper in documents:
    paper=remove-punctuations(paper)
    words=split paper into words
    words=remove-stopwords()
    add paper to list-document
    add words to words-set

# build the dictionary
dictionary=remove-duplicate-words(words-set)
count the words in words-set
dictionary=remove-rare-words(dictionary)

#build one-hot documents
make all the documents in the form of one-hot according to the dictionary

return
```

Until now, we have turn the original papers into a one-hot matrix according to a dictionary built by the training set.When we remove the punctuations, we choose to remove all the stop-words, too, as they have a large count but present little use.

How to Implement the Derivative?

We present this part in two procedures. The first is the derivative of Loss Function and the other the regularization term.

The Loss Function

The loss function here is the cross entropy loss:

$$R(W) = -\frac{1}{N}\sum (y^n)^T log\ \hat{(}y)^n$$

Where the $\hat{y}$ is the prediction made by out model.

The derivative work is learned in the text book Chapter Three, and I try to make the proof on my own here based on my comprehension. To make the conduction, two formulas are essential:

$$y = softmax(Z) \quad \Rightarrow \quad \frac{\partial y}{\partial x} = diag(y) - yy^T$$

$$z = W^T x \quad \Rightarrow \quad \frac{\partial z}{\partial w_c} = [0, 0, \ldots, x, \ldots, 0]$$

Then we can start our deduction. For convenience, we start by the situation of vector, and naturally promote it to matrix case. And we now start our deduction.

$$\begin{aligned}
\frac{\partial L}{\partial w_c} &= -\frac{\partial z^n}{\partial w_c}\frac{\partial \hat{y}^n}{\partial z^n}\frac{\partial log\hat{y}^n}{\partial \hat{y}^y}y^n \\
&= -M_c(x)(diag(\hat{y}^n) - \hat{y}^n(\hat{y}^n)T)(diag(\hat{y}))^{-1}y^n \\
&= -M_c(x^n)(I - \hat{y}^n 1^T)y^n \\
&= -M_c(x^n)(y^n - \hat{y}^n) \\
&= -x^n[y^n - \hat{y}^n]
\end{aligned}$$

The first step is the application of chain rule, and the first to the second line applies the two formulas listed before. And the rest several lines base on basic matrix computation. They will be present in simple computation. And apply this into matrix form and we get:

$$\sum_{n=1}^{N} x^n(y^n - \hat{y}^n_{W_t})^T$$

And in Python, above computations are restricted to matrix operations only. And we give the source code of this part:

```
def cross_entropy_gradient(W_c,X,Y): #W_c=(d+1)*c X=(d+1)*N Y=c*N when usual train
    #W_c=(d+1)*c x=(d+1)*1 y=c*1 when stochastic gradient descent train
    # Y_tilta=get_onehot(W_c,X) #Y_tilta=c*N
    Y_tilta=softmax(W_c,X) #Y_tilta=c*N
    Y_sub=(Y-Y_tilta).T
    return np.dot(X,Y_sub) #X_Y=d*c
```

The Regularization Term

And we use L2 regularization term here, which means: $\frac{1}{2}||W||^2$. This can be done immediately using the basic rule. And we get W. And after we add the term to the update formula, we could find that this term simply add an attenuation to the original parameter which keeps the parameter in comparative small form.

And finally we have:

$$W = (1 - \frac{\lambda\alpha}{N})W - \frac{\alpha}{N}\sum_{n=1}^{N} x^n(y^n - \hat{y}^n_{W_t})^T$$

About L2 Regularization

Should we regularize the bias term? I think maybe not. As the bias applies to all the prediction in the same degree. In other words, the bias itself will not affect the single prediction on one specific sample, thus it has no direct connection with over-fitting. However, compared to the size of dictionary, the little difference of bias do not affect the result that much. And an augment matrix is easier for computation and represent the formula. Since taking the bias into the regularization will not affect the result greatly, we will involve it in the regularization for convenience.

How to Check the Calculation

One simple idea is to directly compute the derivative by its definition. For a specific point, we could use the definition formula to compute derivative and compare it with the output of the normal result of derivative.

$$\frac{\partial L}{\partial W_{ij}} = limit_{\delta_{ij} \to 0} \frac{L(W + \delta_{ij}) - L(W)}{\delta_{ij}}$$

or another formula:

$$\frac{\partial L}{\partial W_{ij}} = limit_{\delta \to 0} \frac{L(W + \delta_{ij}) - L(W - \delta_{ij})}{2\delta_{ij}}$$

And we can know the correctness of the normal derivative.

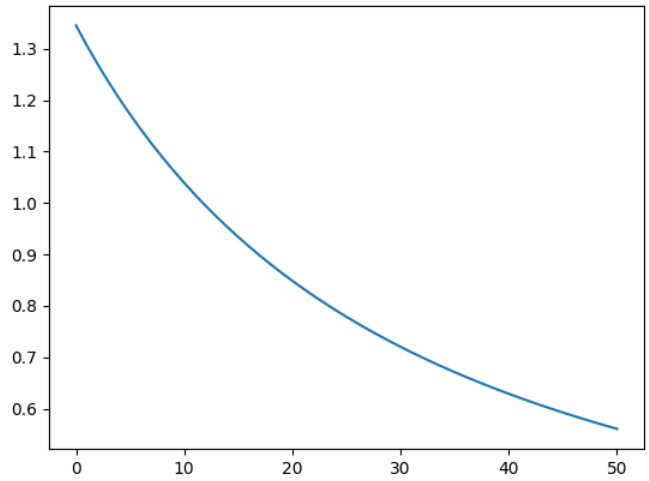
About the Loss Curve

It will be easy to draw a loss curve. We here mainly focus on the Stochastic Gradient Descent and Normal Gradient Descent. We plot the value of loss function every step or epoch. And we get the following plots.

$$R(W) = -\frac{1}{N}\sum (y^n)^T log\ \hat{(}y)^n$$

We have shown this formula before, but for convenience, we put it here again.a

Here we use normal gradient descent as the method to draw the figure and fix the epochs to 50. And λ is fixed to 5, with α is fixed to 0.05.



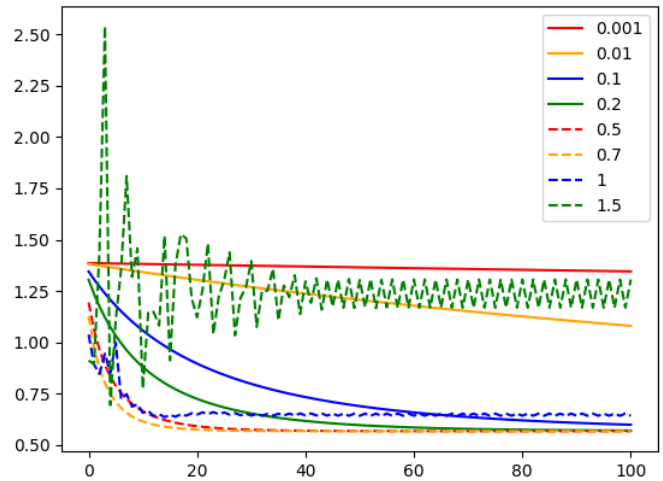
As we could see from this figure, it was extremely smooth. That' s probably because of the choice of learning rate which we are just ready to have an exploration.

Choose the Learning Rate

As we all know, a small learning rate results in a slow convergence while a large one makes it hard to reach the target, easily miss the target and keep regular waves at last.

We can simply plot different plots with different learning rate and choose one which reaches the convergence first. And another method is to vary the learning rate depending on the epoch of training. Maybe it's intuitive that a large learning rate is preferred at beginning while a smaller one is preferred later.

We draw several loss curves with different learning rate and picks one which convergence first and in a smooth curve. For a better intuition, we fixed the max epochs as 100, and λ as 300. (300 is still a small parameter compared with N). We choose α as 0.001, 0.01, 0.1, 0.2, 0.5, 0.7, 1, 1.5 in order to have a look at their performance.

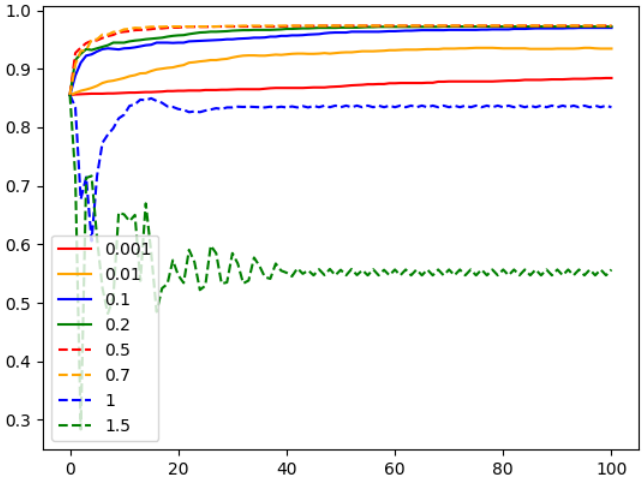


From the figure above, we can find that the speed of descent is larger when the learning rate is bigger. But as we have a special learning rate as 1.5 which presents drastic fluctuations and ends up in a high loss. And the learning rate 1 reaches a relative situation but holds a larger loss than 0.7 sample. And 0.5, 0.7 sample reaches convergence successfully.

From the formula, we could figure out that the best learning rate is decided by the model itself and λ. First, it affects the weight of update term $\sum_{n=1}^{N} x^n(y^n - \hat{y}_{W_t}^n)^T$. A lager λ makes the influences of this term increases. Second, λ affects the convergent point of loss directly. And if we deceases λ, we would probably have a larger learning rate with more accuracy on training set but less generalization.

Based on the discussions above, choose learning rate as 0.5 for this question would be nice, for it needs almost the same time as 1 to get convergence and small enough to prevent unstable situation.

We could also have a look at the accuracy of the accuracy curve:



And this coordinates with the result of loss. The performance of $\alpha = 0.7$ is also good in accuracy test.

And we could also try to vary learning rate version during the learning process. One simple strategy is to take the learning rate as *1/T*, where *T* is the number of executed epochs.

When to Determinate the process

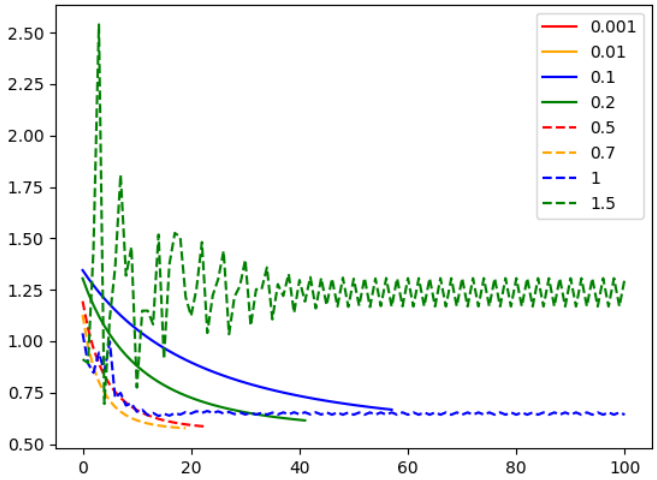
We could use three strategies. The first is to draw a least line for the loss function. When two continuing epochs have difference in the loss less than the line, we stop the training. Even though, we have more options to determine the ending. For example, we could set a max number for difference. When the change of loss is less than ϵ, we stops.

$$||\Delta loss||^2 < \epsilon$$

And the other method is to focus on the accuracy. When we find that the accuracy stops to change greatly, we stops the training.

The third one is to compare the difference of W directly. But the change of W is decided by the derivative of loss function, which is similar to the first method. So we could simply try these first two methods. We apply the first method and plot again. However, both of these two methods depends on learning rate and regularization term. But from another aspect, when the change is too slow, we could just stop the training, for it will cost a lot time to reach convergence, so the method would be just fine.

And the following is a figure which is controlled by $\epsilon = 0.0002$. We could find that the 0.07 curve stops first. 1 and 1.5 curve never reaches a determination.



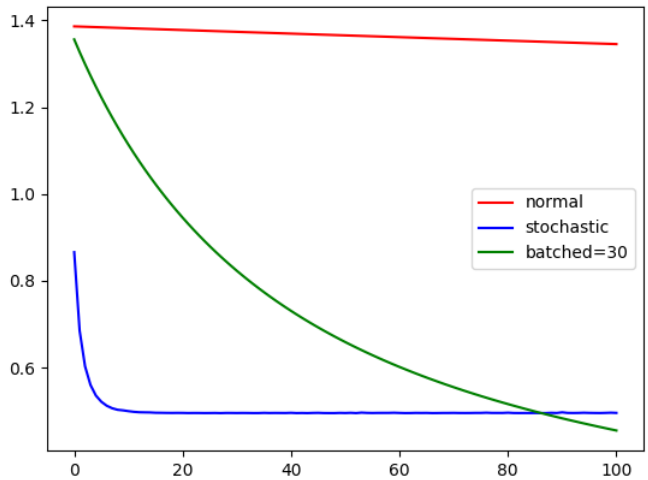
In fact, the first method focuses on the process of the training, while the second one focuses on the result. And the problem of deciding the determination is converted to the decision of ϵ.

About Stochastic Gradient Descent and Batched Gradient Descent

Basic Loss Curve

Stochastic Gradient Descent (SGD) is a special form of Batched Gradient Descent (BGD). To have a intuition of these methods, we don not use the $\epsilon - determination$ to make the plot with $\alpha = 0.01$, $\epsilon = 0$, $\lambda = 0.1$. And for convenience, we choose batched parameter *k* as 30. We will have an exploration on that later.

From the formula of SGD and BGD, we know that they are more easily affected by λ, because they compute much more times of W_c than the normal gradient descent. So we cut down λ to go on test.

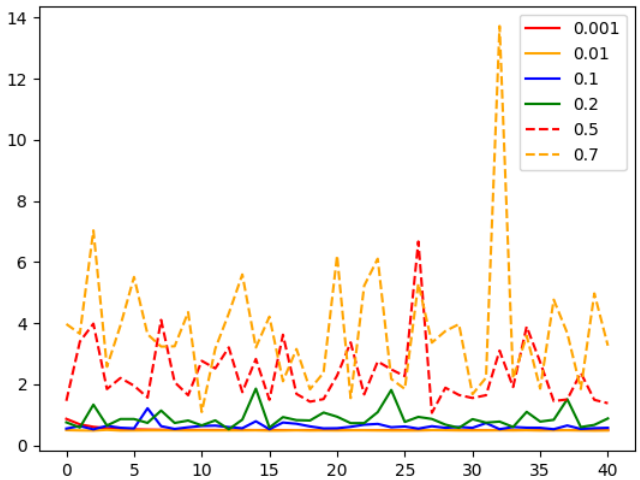


From the Loss curve and Accuracy curve, we know that the SGD and BGD convergent with great speed, but keep steady fluctuations. But in the same time, they spend more time to finish one epoch, for much more computations needed. Of course, BGD show less time need than SGD and less epochs than normal gradient descent to reach convergence. And we could see that the loss of BGD is less than that of SGD, as it better shows the global features of the dataset.

Further Explorations on SGD and BGD

About SGD

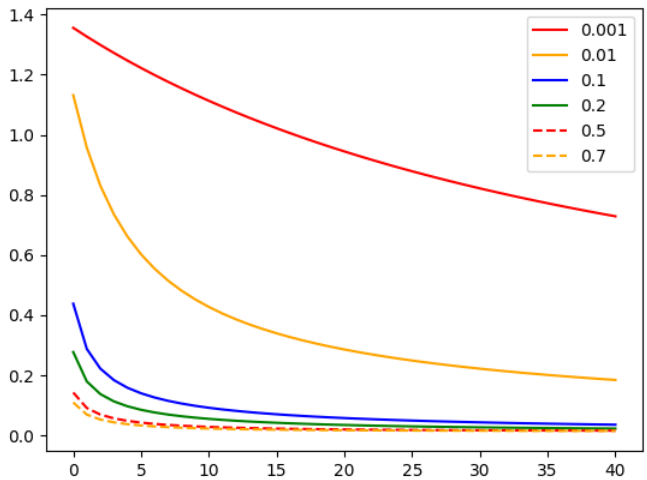
The problem of SGB is to choose a good learning rate. For it updates parameter quickly, it requires a smaller α than that of NGB. For a learning rate of *0.01*, it finishes its process in just two epochs. BUT it has a larger loss even though it reaches its convergence in a shorter time. So the main point of SGD is not on the time for convergence but the loss.



We plots some loss curves based on different α, and our predictions match the results. And once α is larger than a certain point, it starts to fluctuate greatly and never reaches a convergence. And 0.7 here is no longer a suitable learning rate.

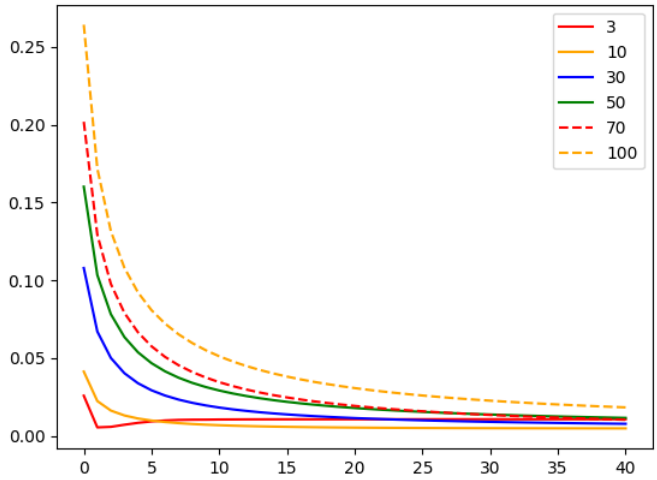
About BGD

The parameters of BGD are k and α. As BGD is half-NGD and half-SGD, the differences of the BGD mainly depends on the spacial parameter k. A larger k makes it better shows the global dataset, while a smaller one reaches convergence faster. Like SGD, we plot a same figure based on same parameters. k is fixed as 30.



Unlike SGB, BGD bears a bigger learning rate but cannot convergent as fast as SGD. But interestingly, it has much less loss compared to SGD.

And another important parameter is k. We fixed α as 0.7 and varies k.



And just like the difference of SGD and NGD, a larger *k* uses more time to reach convergence. And a smaller *k* has larger loss, and of course spends more execution time.

Pros and Cons

Abviously, SGB reacts fast to the data set, and reaches the convergence in a quite short time, but it spends too much time computing. It uses less learning rate to study, and causes larger loss on the training set.

NGB reacts much slowly than than SGD, but it finishes computing using much less time. It needs about tens of times of the epochs of SGD.

And BGD is just between the two. It uses less time computing and reaches the convergence in a suitable time.

Check the result with the test dataset.

Using $epoch = 40$, $\lambda = 100$ and $\alpha = 1$, we get the $accuracy = 0.9291$ on NGD model. With a larger epoch, it can be increased to 0.9298.

Using $epoch = 40$, $\lambda = 0.2$ and $\alpha = 0.01$, we get $accuracy = 0.915$ on SGD model. And the model fully reaches its convergence.

Using $epoch = 40$, $\lambda = 2.5$, $k = 100$ and $\alpha = 0.1$, we get $accuracy = 0.934$ on BGD model.

So although NGD can get the best estimation, it needs too much time before convergence. And SGD model reaches the convergence in a short time, but it lacks accuracy. So balancing α and k in BGD provides a good solution to reach a nice accuracy in a relative short time. For example, in BGD with $\lambda = 5$, $\alpha = 0.4$, and $k = 200$, the accuracy reaches 0.936 in 100 epochs. The only problem is the time we have to train the classifier.

Conclusions

In this paper, we have discussed three different kinds of classifiers and their performance case by case. For LSM, it performances perfectly in the given dataset in a neat and quick way. And from the figure, we could find that the classifier just keeps suitable distance with the dataset, which is just what we what. However, the perceeptron algorithm performances a little worse. Although it also has a 100% accuracy, but some data lies quite close to the classifying line. Besides the learning rate of

perceptron shows no significant use in this problem.

And about NGD, also SGD and BGD, there are much more interesting things. We have discussed the deduction, the choice of learning rate and regularization term. We have a close look at the performance of each model. And the most interesting thing is that, the SGD reaches convergence in a short time but lacks accuracy; NGD reaches convergence in a long time, but has higher accuracy. And adjust k in BGD, we could balance the time we spend and the accuracy. So a nice picked parameter for BGD shows the best performance.

References

1. Christopher M.Bishop, *Recognizing and Machine Learning*.
2. index
3. 《神经网络与深度学习》