Assignment 02

Linear classification

Anonymity

Department of Computer Science, Fudan University

April 2, 2019

1 Overview

1.1 Introduction

In Lecture 4, we have discussed several well-known linear classification methods, such as perceptron and logistic regression.

1.2 Assignment Description

In this assignment you are going to explore several well-know linear classification methods such as perceptron and logistic regression, and a realistic dataset will be provided for you to evaluate these methods.

1.3 Outline

The rest of the report is organized as follow: In Section 2, we start with the Part I of this assignment. Specifically, we will use **Least Square Model** in Sec. 2.2 and **Perceptron Algorithm** in Sec. 2.3 to classify a linearly seperable datasets in 2-dimension case. In Section 3, we will dive into a realistic scenerio, where we apply **Logistic Regression** to a text classification task. This gives us a deep insight into the practical use of machine learning. As usual, for the purpose of rating, you can only focus on sections answering the questions in requirements.

2 Part I: Linear Classification

2.1 Problem Description

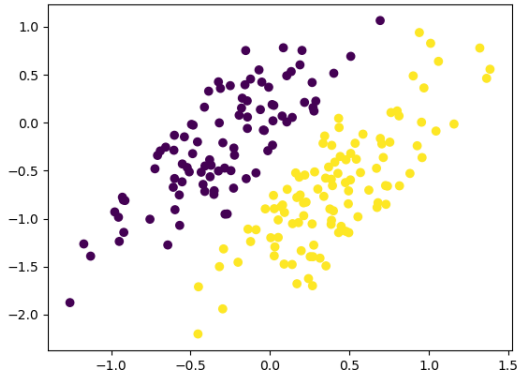
To start with, we consider the least square model as an extension for the linear regression model to the problem of classification, and also the perceptron algorithm. You are given a simple linearly separable dataset containing 2 classes of points on a 2D plane, and you should build a model to correctly classify them.

You are provided with a function **gen_linear_seperatable_2d_2c_dataset** in the handout file, you should import it to your solution as in the first assignment.

Requirements

Namely speaking, you should use least square model and the perceptron algorithm to learn two models to separate the dataset, and report the accuracy after you have learned the model.

Since the two models are linear, you should be able to draw a decision line on top of the dataset to visually show how can you separate the dataset. Include this plot in you report.



2.2 Least Square Model

2.2.1 Theory

Least Square Model is a standard approach in regression analysis, which aims to minimize the sum of squares of the residuals made in the results of every single point. It is usually credited to Carl Friedrich Gauss [1], but it was first published by Adrien-Marie Legendre [2].

We first consider a general classification D-dimensional problem with K classes, with a 1-of-K binary coding scheme for the target vector $\boldsymbol{t}$. Each class $\mathcal{C}_k$ is described by its own linear model so that

$$y(\boldsymbol{x}) = \boldsymbol{w}_k^\top \boldsymbol{x} + w_{k0} \tag{1}$$

where $k = 1, \cdots, K$. Typically, we group these together using vector notation so that

$$\boldsymbol{y}(\boldsymbol{x}) = \widetilde{\boldsymbol{W}}^\top \widetilde{\boldsymbol{x}} \tag{2}$$

where $\widetilde{\boldsymbol{W}}$ is a matrix whose k^{th} column comprises the $D+1$-dimensional vector $\widetilde{\boldsymbol{w}}_k = (w_{k0}, \boldsymbol{w}_k^\top)^\top$ and $\widetilde{\boldsymbol{x}}$ is the corresponding augmented input vector $(1, \boldsymbol{x}^\top)^\top$

with a dummy input $x_0 = 1$. A new input $\boldsymbol{x}$ is the assigned to the class for which the output $y_k = \widetilde{\boldsymbol{w}}_k^\top \widetilde{\boldsymbol{x}}$ is largest.

Now, we determine the weight matrix $\widetilde{\boldsymbol{W}}$ by minimizing a sum-of-squares error function,

$$\begin{aligned} E_D(\widetilde{\boldsymbol{W}}) =& \frac{1}{2}\mathrm{Tr}\left((\widetilde{\boldsymbol{X}}\widetilde{\boldsymbol{W}} - \boldsymbol{T})^\top(\widetilde{\boldsymbol{X}}\widetilde{\boldsymbol{W}} - \boldsymbol{T})\right) \\ =& \frac{1}{2}\left\|\widetilde{\boldsymbol{X}}\widetilde{\boldsymbol{W}} - \boldsymbol{T}\right\|_F^2, \end{aligned} \tag{3}$$

where $\|\cdot\|_F$ denotes the Frobenius norm. Note that Eq.(3) is just a generalization of least square error in high dimension.

Setting the dereivative with respect to $\widetilde{\boldsymbol{W}}$ to zero, we then obtain the solution for $\widetilde{\boldsymbol{W}}$ in the form

$$\widetilde{\boldsymbol{W}} = (\widetilde{\boldsymbol{X}}^\top\widetilde{\boldsymbol{X}})^{-1}\widetilde{\boldsymbol{X}}^\top\boldsymbol{T} = \widetilde{\boldsymbol{X}}^\dagger\boldsymbol{T} \tag{4}$$

and the discriminant function in the form

$$\boldsymbol{y}(\boldsymbol{x}) = \widetilde{\boldsymbol{W}}^\top\widetilde{\boldsymbol{x}} = \boldsymbol{T}^\top(\widetilde{\boldsymbol{X}}^\dagger)^\top\widetilde{\boldsymbol{x}}, \tag{5}$$

where $\widetilde{\boldsymbol{X}}$ is the pseudoinverse of the maxtrix $\widetilde{\boldsymbol{X}}$.

2.2.2 Two-Class Simplification

In this assignment, we deal with the 2-class case, which simplifies the original problem substantailly. As we shall see, the decision line can be determined directly from Eq.(4).

As shown in Eq.(4), we can first derive the two class weights in the form

$$\widetilde{\boldsymbol{w}}_1 = \widetilde{\boldsymbol{X}}^\dagger\boldsymbol{t}_1 \tag{6}$$

and

$$\widetilde{\boldsymbol{w}}_2 = \widetilde{\boldsymbol{X}}^\dagger\boldsymbol{t}_2 \tag{7}$$

Recall that we classify a new input $\boldsymbol{x}$ by selecting a class K with the largest discriminant function. On top of that, we can derive the decision function for two-class case in the form.

$$y(\boldsymbol{x}) = (\widetilde{\boldsymbol{w}}_2 - \widetilde{\boldsymbol{w}}_1)^\top\widetilde{\boldsymbol{x}} = (\boldsymbol{t}_2 - \boldsymbol{t}_1)^\top(\widetilde{\boldsymbol{X}}^\dagger)^\top\widetilde{\boldsymbol{x}}, \tag{8}$$

The decision surface is $y(\boldsymbol{x}) = 0$, which means an input vector is assigned to class $\mathcal{C}_2$ if $y(\boldsymbol{x}) \geq 0$ and to class $\mathcal{C}_1$ otherwise.

Hence, instead of evaluating two coefficient vector $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ respectively, we can simply define a new weight vector $\boldsymbol{w} = \widetilde{\boldsymbol{X}}^\dagger(\boldsymbol{t}_2 - \boldsymbol{t}_1)$ and use the decision fucntion $y(\boldsymbol{x}) = \boldsymbol{w}^\top\widetilde{\boldsymbol{x}}$ for classification.

2.2.3 Experiments

To substantiate the effectiveness of LSM, we conduct experiments on the given dataset, the result of which is displayed in Fig. 1.

The decision line is given by $y(\boldsymbol{x}) = 0$ and depicted in Fig. 1. The classification accuracy of LSM is 100%, as observed in the result plot.

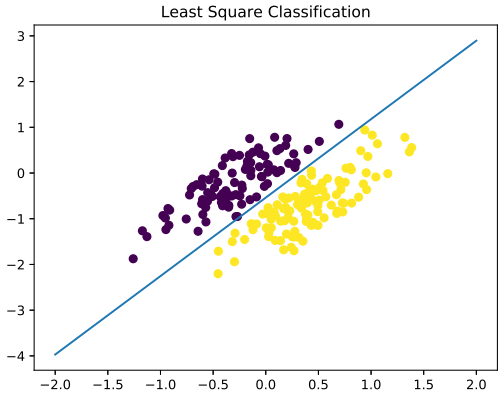


Figure 1: LSM Result: A Perfect Decision Line

2.2.4 Drawbacks

Despite the high accuracy of LSM in Sec. 2.2.3, it cannot apply to general classfication task due to the unappropriate error function. The sum-of-squares error function penalizes predictions that are 'too correct' in that they lie a long way on the correct side of the decision boundary. As a result, LSM lacks robustness to outliers.

2.3 Perceptron Algorithm

2.3.1 Theory

Peceptron Algorithm is an algorithm for supurvised learning of binary classifiers. The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt [3] and occupied an important place in the history of pattern recognition algorithms.

As a two-class model, it constructs a linear model of the form

$$y(\boldsymbol{x}) = f(\boldsymbol{w}^\top\boldsymbol{x}) \tag{9}$$

where $f(\cdot)$ is a nonlinear activation function, given by a step function of the form

$$f(\cdot) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases} \tag{10}$$

As usual, the bias component is included in the weight vector $\boldsymbol{w}$. For the sake of convenience, here we use target values $t = +1$ for class $\mathcal{C}_1$ and $t = -1$ for class $\mathcal{C}_2$, which matches the choice of activation function.

Subsequently, we should define an error function for the learning process. Here, we invoke the *perceptron criterion*, given by

$$E_P(\boldsymbol{w}) = -\sum_{n\in\mathcal{M}} \boldsymbol{w}^\top\boldsymbol{x}_n t_n \tag{11}$$

where $\mathcal{M}$ denotes the set of all misclassified patterns.

Applying the stochastic gradient descent algorithm, we can derive the change in the weight vector in the

formula

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E_P(\boldsymbol{w}) = \boldsymbol{w}^{(\tau)} + \eta \boldsymbol{x}_n t_n \quad (12)$$

where η is the learning rate parameter and τ is an integer that indexes the steps of the algorithm. For details of the gradient descent algorithm, I refer you to Sec. 3.4.

Since the stochastic gradient descent only considers one data per iteration, this learning rule is not guaranteed to reduce the total error function at each stage. Notwithstanding, the *perceptron convergence theorem* [4] substantiates that it always finds an exact solution for the linearly seperable case.

2.3.2 Experiments

We apply the perceptron algorithm on the given dataset and illustrate the result in Fig. 2.

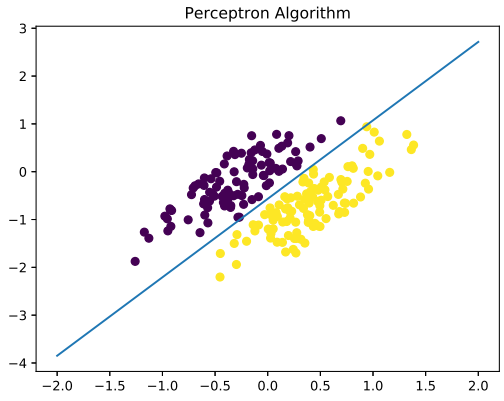


Figure 2: Perceptron Result: Another Perfect Decision Line

It can be observed that two classes is perfectly seperated by the decision line and the accuracy of this algorithm is 100%. This result is consistent with the *perceptron convergence theorem*, which guarantees the exact solution in the linearly seperable case.

2.3.3 Drawbacks

Although its remarkable performance in the linearly seperable case, there are still many drawbacks of the perceptron algorithm:

1. The perceptron cannot provide probabilistic outputs.
2. Generalization to $K > 2$ classes is tricky.
3. The algorithm will never converge on the non-linearly sperable data sets even in the simplest XOR case.

3 Part II: Text Classification

3.1 Problem Descrition

In this part of the assignment, you are required to use logistic regression to do a simple text classification task. A function **get_text_classification_datasets** is also provided, which returns the training and testing dataset, and you could explore the dataset by looking more closely to what it returns.

3.2 Requirement 1

Requirement 1: *Implement the preprocess pipeline to transform the documents into multi-hot vector representation, and also the targets into one-hot representation, indicate how you implemented them in your report.*

The main procedure of preprocessing is listed as follow:

1. **Prune Text Documents.** Firstly, we should delete all the characters in **string.punctation**, which is equivalent to a string delete process. This can be easily done by **str.maketrans()** and **text.translate()** methods. After that, we substitute all **string.whitespace** by a space, which is accomplished using **re.sub()** method. **However, be careful with the regular expression!** When attempting to delete the characters in **string.punction** using **re**, I have trouble with the annoying characters, such as '[' and '\'. After some efforts, I finally determine to avoid the *elegant* **re** module and turn to the **str** module.
2. **Extract Useful Words.** Since it is unneccesary to distinguish the letter case, we convert all characters into lowercase by **test.lower()** method. Then, we can easily split the document into word lists by spaces using **text.split()** method.
3. **Get Vocabulary.** Instead of recording all words in our dictionary, we only consider frequent words to avoid the excessive memory cost. Build a dictionary to count the occurence of words. Filter the words that occur at least 10 times. Finally, put all remaining words into our dictionary.
4. **Transform into Multi-Hot/One-Hot Vector.** This is a simple job. The only thing we should note is the efficiency of implementation. **Look up words in dictionary rather than in list!**

3.3 Logistic Regression

In this section, we begin our treatment of logistic regression. For the sake of convenience, we will use N to denote the size of data set, M to denote the number of features and K to denote the number of classes.

As a general linear model, we typically impose a linear combination of input vector $\boldsymbol{x}$ as

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}, \quad (13)$$

where $\boldsymbol{W}$ is a $M \times K$ weight matrix and $\boldsymbol{b}$ is a $K \times 1$ bias vector.

Then, we use a non-linear function as the activation function to enable probabilistic interpretation for the output.

For binary classification, the logistic sigmoid function

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \tag{14}$$

transforms the unbounded prediction $a \in (-\infty, +\infty)$ from the output of the linear model to a bounded intervel $(0, 1)$, which could be interpreted as the probability of the prediction.

The generalization of logistic sigmoid function to K-dimension is softmax function, which is given in the form

$$\hat{\boldsymbol{y}} = softmax(\boldsymbol{z}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} = \frac{\exp(\boldsymbol{z})}{\boldsymbol{1}^\top \exp(\boldsymbol{z})}, \tag{15}$$

where we have defined $\exp(\boldsymbol{z}) = [e^{z_1}, e^{z_2}, \cdots, e^{z_k}]^\top$.

We can then make prediction through selecting a class with the maximum associated probability:

$$\hat{\boldsymbol{y}}_{pred} = \arg \max_{j \in 1..K} [softmax(\boldsymbol{z})]. \tag{16}$$

For the training purpose, we need to find an object function to optimize. One straightforward idea is to maximize the likelihood, which is given in the form

$$p(\boldsymbol{Y}) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(\mathcal{C}_k)^{y_{nk}} = \prod_{n=1}^{N} \prod_{k=1}^{K} \hat{y}_{nk}^{y_{nk}}, \tag{17}$$

where y_{nk} is the target vector. Taking the negative logarithm then gives

$$\log(p(\boldsymbol{Y})) = -\sum_{n=1}^{N} \boldsymbol{y}_n^\top \log \hat{\boldsymbol{y}}_n, \tag{18}$$

where $\boldsymbol{y}_n$ denotes the target one-hot vector and $\hat{\boldsymbol{y}}_n$ is our prediction. Then, taking the average over the data set, we will obtain the cross entropy loss function

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^{N} \boldsymbol{y}_n^\top \log \hat{\boldsymbol{y}}_n. \tag{19}$$

In order to reduce overfitting, an additional term penalizing large weights is added to the loss function as:

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^{N} \boldsymbol{y}_n^\top \log \hat{\boldsymbol{y}}_n + \lambda \| \boldsymbol{W} \|_F^2, \tag{20}$$

where $\|\cdot\|_F$ denotes the Frobenius norm, which is defined as

$$\| \boldsymbol{W} \|_F := \sqrt{\sum_{i=1}^{M} \sum_{j=1}^{K} w_{ij}^2} = \sqrt{\mathrm{Tr}\left(\boldsymbol{W}^\top \boldsymbol{W} \right)}. \tag{21}$$

Finally, we resort to gradient descent approach to minimize the loss function $\mathcal{L}$ in order to get parameter matrix $\boldsymbol{W}$ and $\boldsymbol{b}$. This will be further discussed in Sec. 3.4.

3.4 Gradient Descent

Gradient Descent is a first-order iterative optimization algorithm for finding the minimum of a function. Gradient Descent is based on the observation that if the $F(\boldsymbol{x})$ is defined and differentiable in a neighborhood of a point $\boldsymbol{x}$, then $F(\boldsymbol{x})$ decreases fastest if one goes from $\boldsymbol{x}$ in the direction of the negative gradient of F at $\boldsymbol{x}$, $-\nabla F(\boldsymbol{x})$. It follows that, if

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \nabla F(\boldsymbol{x}_n) \tag{22}$$

for $\alpha > 0$ small enough, then $F(\boldsymbol{x}_n) \geq F(\boldsymbol{x}_{n+1})$. So hopefully the sequence $\{\boldsymbol{x}_n\}$ converges to the desired local minimum. When the function F is convex, all local minima are also global minima, so in this case gradient descent can converge to the global solution.

Full Batch Gradient Descent(FBGD) ,which invovles processing the entire training set in one go, can be computationally costly for large data sets and cannot apply to on-line problems. Thus, it is worthwhile to use sequential algorithms, such as **Stochastic Gradient Descent(SGD)**. If the error function comprises a sum over data points $E = \sum_n E_n$, then after presentation of pattern n, the stochastic gradient descent algorithm updates the point by

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \nabla E_n. \tag{23}$$

Mini-Batch Gradient Descent(miniBGD) is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent.

3.5 Requirement 2

Requirement 2: *Differentiate the loss function for logistic regression, write down how you compute* $\frac{\partial \mathcal{L}}{\partial W_{i,j}}$, $\frac{\partial \mathcal{L}}{\partial b_i}$, *and then implement the calculation in vectorized style in numpy (meaning you should not use any explicit loop to obtain the gradient). Answer the two questions in your report: (1) Sometimes, to overcome overfitting, people also use L2 regularization for logistic regularization, if you add the L2 regularization, should you regularize the bias term? (2) how do you check your gradient calculation is correct?*

3.5.1 Mathematics

To calculate the gradient of cross entropy loss function $\mathcal{L}$, we should take the derivative step by step, and then apply the chain rule.

Firstly, we calculate the derivative of $\boldsymbol{z}$ with respect to weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$:

$$\frac{\partial \boldsymbol{z}}{\partial W_{i,j}} = [0, 0, \cdots, x_i, \cdots, 0]^\top \tag{24}$$

$$\frac{\partial \boldsymbol{z}}{\partial b_j} = [0, 0, \cdots, 1, \cdots, 0]^\top \tag{25}$$

where $i = 1 \cdots M$ and $j = 1 \cdots K$. Note that only the j-th element is not zero.

Then, we evaluate the Jacobian matrix of our predictor $\hat{\boldsymbol{y}}$ with respect to $\boldsymbol{z}$:

$$\begin{aligned}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}} &= \frac{\frac{\partial \exp(\boldsymbol{z})}{\partial \boldsymbol{z}} \cdot (\boldsymbol{1}^\top \exp(\boldsymbol{z})) - \frac{\partial \boldsymbol{1}^\top \exp(\boldsymbol{z})}{\partial \boldsymbol{z}} \cdot \exp(\boldsymbol{z})^\top}{(\boldsymbol{1}^\top \exp(\boldsymbol{z}))^2} \\ &= \frac{(\boldsymbol{1}^\top \exp(\boldsymbol{z}))\, diag(\exp(\boldsymbol{z})) - \exp(\boldsymbol{z}) \exp(\boldsymbol{z})^\top}{(\boldsymbol{1}^\top \exp(\boldsymbol{z}))^2}\end{aligned}$$

where $diag(\exp(\boldsymbol{z}))$ denotes the diagonal matrix with i-th diagonal element equal to e^{z_i}.

Specifically, we conclude

$$\frac{\partial \hat{y}_k}{\partial z_j} = \hat{y}_k(I_{kj} - \hat{y}_j), \tag{26}$$

where I_{kj} are the elements of the identity matrix. This is the general form of derivative of sigmoid function in multi-class case.

In the end, we write the loss function in the form:

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \log \hat{y}_{nk} + \lambda \| \boldsymbol{W} \|_F^2 \tag{27}$$

and use the chain rule:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_{ij}} &= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \left(\frac{\partial z_{nj}}{\partial W_{ij}} \frac{\partial \hat{y}_{nk}}{\partial z_{nj}} \frac{1}{\hat{y}_{nk}} \right) + 2\lambda W_{ij} \\ &= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} x_{ni} (I_{kj} - \hat{y}_{nj}) + 2\lambda W_{ij} \\ &= -\frac{1}{N} \sum_{n=1}^{N} (y_{nj} - \hat{y}_{nj}) x_{ni} + 2\lambda W_{ij}\end{aligned}$$

where we have made use of $\sum_k y_{nk} = 1$.

Analogously, we have

$$\frac{\partial \mathcal{L}}{\partial b_j} = -\frac{1}{N} \sum_{n=1}^{N} (y_{nj} - \hat{y}_{nj}).$$

3.5.2 Regularization

In machine learning problems, regularization is the process of adding information in order to prevent overfitting. One notable phenomenon is that with the magnitude of the coefficients growing larger, the models become more complex and the coefficients have become finely tuned to the data by developing large positive and negative values.

A common regularization technique is to add a penalty term to the error function in order to discourage the coefficients from reaching large values. The simplest such penalty term takes the form of a sum of squares of all of the coefficients.

Since the bias term $\boldsymbol{b}$ merely plays a role of offset and its scale is less important to the complexity of models, **we should not regularize the bias**. Moreover, in case a large bias is needed, regularizing it will prevent finding the correct relationship. On top of this, we should only penalize the 'overly specific' coefficients matrix $\boldsymbol{W}$.

Note that the regularization coefficient λ is a hyperparameter. In practice, we need to spilt our data into training set and validation set. After that, we should train our model with distinct λ and select the one with highest accuracy on validation set. This procedure is the same as what we have done in Assignment 1, and is both tedious and time-consuming. **To make life easier, we directly set $\lambda = 0.001$.**

3.5.3 Gradient Checker

Recall that the mathematical definition of the derivative as

$$\frac{\partial J(\theta)}{\partial \theta} = \lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}. \tag{28}$$

Thus, at any specifical value of θ, we can numerically approximaate the derivative as follows:

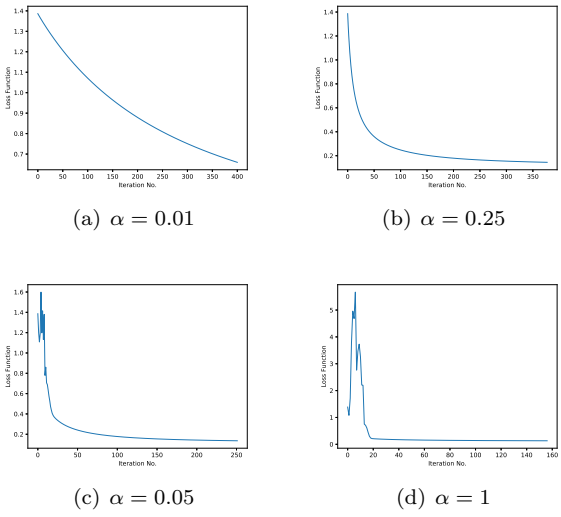
$$\frac{\partial J(\theta)}{\partial \theta} \approx \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}. \tag{29}$$

The degree to which these two values should approximate each other will depend on the details of J and ϵ. Here we set $\epsilon = 10^{-3}$ to check our gradient calculator. If the difference between two values is below the error threshold, we then conclude that our calculator is correct and thus verify our graident formula.

I have implemented the gradient checker in **PartII.Gradient_Checker()** function and checked my gradient.

3.6 Requirement 3

Requirement 3: *Finish the training for the logistic regression model on the training dataset, include the plot for the loss curve you obtained during the training of the model. Answer the questions: (1) how do you determine the learning rate? (2) how do you determine when to terminate the training procedure?*



(a) $\alpha = 0.01$ (b) $\alpha = 0.25$

(c) $\alpha = 0.05$ (d) $\alpha = 1$

Figure 3: **FBGD Loss Curve** with different learning rate α

After all preprocessing procedure, we implement our training method using **Full Batch Gradient Descent(FBGD)** approach where we have set the regularization coefficient $\lambda = 0.001$. During the training of the model, we record the value of loss function and plot the loss function in Fig. 3.

Learning rate is a hyperparameter that controls how much we are adjusting the weights of our network with respect to the loss gradient. As can be seen in Fig. 3, for small and moderate learning rate, such as $\alpha = 0.01, 0.25$, the loss function will decrease at every iteration but converge in hundreds of iterations. Large learning rate, however, may even increase the loss function.

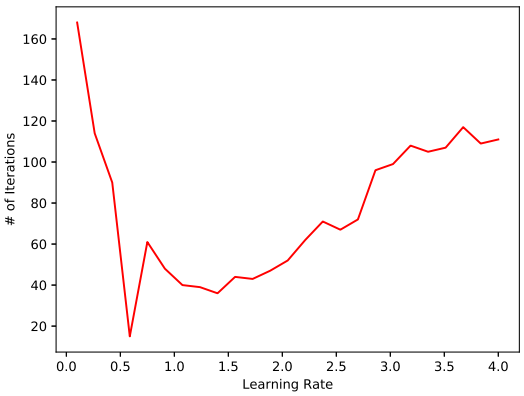


Figure 4: Number of iterations before convergence for various learning rate

To exemplify the effect of learning rate, we depict a curve with x-axis being learning rate and y-axis being the number of iterations needed for convergence. From the Fig. 4, it is evident that with lower learning rate, the convergence speed is rather slow. Whereas the large learning rate may cause this number unstable due to the unpredictable behavior in optimization process. And as shown in Fig. 3, it even leads to the unexpected increase of loss function. Thus, we conclude that there is a tradeoff between speed and stability of convergence.

Consequently, to determine the learning rate, we should first train our models on small datasets with several learning rates and plot their loss curve and then pick the moderate one. In this problem, $\alpha = 0.25$ is suitable.

Another common strategy is to decay the learning rate during the optimization process. For instance, you can divide the learning rate by 10 after several iterations aimming to avoid the oscillation around the optima.

As for **terminating condition**, it is straightforward for full batch gradient descent. The cost function will decrease every iteration until numerical instability sets in. The magnitude of the changes will become smaller and smaller and we can stop the iteration when the changes below our predefined thresholds.

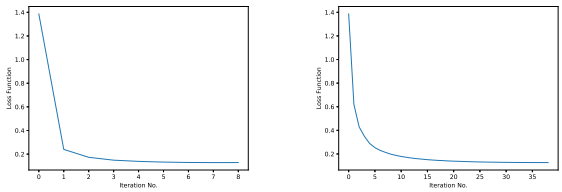
In the cases of **SGD** or **MiniBGD**, the loss function is not guaranteed to decrease at each iteration owing to the nosiy training process. In spite of this, the loss still tends to lower and converge to the optimal in the long run. Hence, we can evaluate the loss function after a block of iterations and check whether the changes is below the thresholds. **In our implementations, we examine the loss function per epoch and define the threshold as $\epsilon = 10^{-4}$.**

3.7 Requirement 4

Requirement 4: *Somethings, other than doing a full batch gradient descent (where you take into consideration all the training data during the calculation of the gradient), people use stochastic gradient descent or batched gradient descent (meaning only one sample or several samples per update), you should also experiment with the other 2 ways of doing gradient descent with logistic regression. Answer the questions: (1) what do you observe by doing the other 2 different ways of gradient descent? (2) can you tell what are the pros and cons of each of the three different gradient update strategies?*

3.7.1 Experiments

Aside from **FBGD**, we implement two other gradient descent approaches introduced in Sec. 3.4. The result loss curve is depicted in Fig. 5. Here, we set the batch size equal to 128 for **MiniBGD**.



(a) **SGD** (b) **MiniBGD**

Figure 5: Loss Curve of **SGD** and **MiniBGD**

As can be observed, the total number of epochs is significantly decreased with small batch sizes. In contrast with hundreds of epoches for **FBGD** training, **SGD** only need 8 epoches, which gives rise to its high computational efficiency. However, the time per epoch is increased due to the growing number of updates in each scanning of the entire dataset. **MiniBGD** balances the number of epochs and the time consuming per epoch, thus gives a trade-off solution to avoid two extremes.

3.7.2 Discussion

In this section, we breifly list some pros and cons of three different gradient update stratagies.

Upsides of SGD

- **SGD** updates the model for each training example, which enables its use in online machine learning problems.

- It can also significantly reduce the number of epoches before convergence.
- Since we only need one example per iteration, the space cost is rather low.
- The noisy update process may allow the model to avoid local minima in non-convex problems.

Downsides of SGD

- **SGD** updates so frequently that makes it more computationally expensive and take longer to train models with high update cost.
- It becomes harder to predict the error bound for optimization due to the unpredictable learning process.

Upsides of FBGD

- Although **FBGD** takes more epoches to converge, its total iterations is fewer, thus needs few updates than **SGD**.
- The terminating condition of **FBGD** is straightforward.

Downsides of FBGD

- Commonly, it requires the entire training dataset in memory and available to the algorithm.
- The computation of gradient may become very slow for large datasets.

Upsides of MiniBGD

- It's the balanced version of **SGD** and **FBGD**, which both save the memory and improve the training speed.
- The model update frenquency is higher than **FBGD**, which allows for a more robust convergence, avoiding local minima.

Downsides of MiniBGD

- **MiniBGD** requires the configuration of an additional 'mini-batch size' hyperparameter for the learning algorithm.

3.8 Requirement 5

Requirement 5: *Report your result for the three differently trained model on the test dataset.*

We have run our three implementaions on the test set. For all these methods, we have set regularization coefficient $\lambda = 0.001$ and convergence threshold $\epsilon = 0.001$. And for **FBGD** and **MiniBGD** with batch size = 128, we let the learning rate α be 0.25, while the learning rate for **SGD** is 0.01. This is because **SGD** is unstable near the optima.

The accuracy of three methods is nearly identical, which is 92.6% for **FBGD**, 92.7% for **MiniBGD**, 92.7% for **SGD**. In spite of this, **SGD** is far more efficient compared with the other two methods.

References

[1] Otto Bretscher. Linear algebra with applications.(2005).

[2] Stephen M Stigler. Gauss and the invention of least squares. *The Annals of Statistics*, pages 465–474, 1981.

[3] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para.* Cornell Aeronautical Laboratory, 1957.

[4] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, CORNELL AERONAUTICAL LAB INC BUFFALO NY, 1961.