# Report for Assignment_2

## Part 1

This is a 2-D classification problem, which can be drawn on the plane. We can easily find that the dataset is linearly separative, so there are many methods to finish this part. Least square and perception algorithm are used to build the model for predict the classcification.

### Least Square

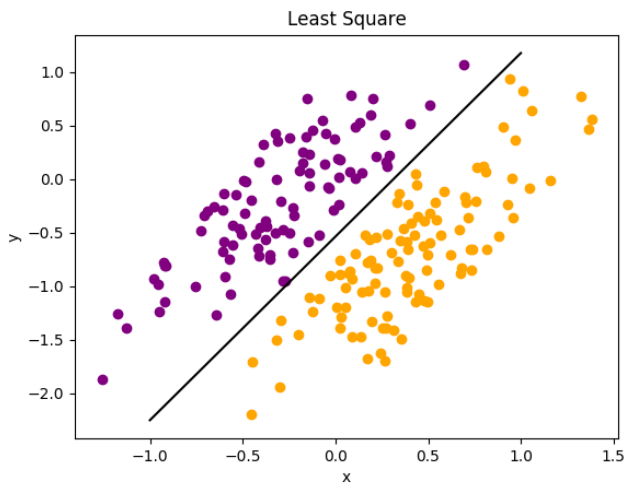
Least squares is first used in the definition of sum-of-squares error function, which lead to a closed-form solution. It has a brief fomula to calculate the parameters:

$$\widetilde{W} = (\widetilde{X}^T \widetilde{X})^{-1} \widetilde{X}^T T = \widetilde{X}^{\dagger} T$$

However, the matrix **T** (whose $n^{th}$ row is the vector $t_n^T$) is a 1-of-K binary coding scheme for where K denotes the number of classes. So **T** is a (N,2) matrix and W is a (3,2) matrix here. Because this is a binary classification problem, we can know the label of data through the first column of **T**. Consider a data and its label vector **t**, which is a row of **T**. If the first element is 1, it belongs to the first class. Otherwise, the first element is 0 and it belongs to the second class. So if we take the first element into consideration, it won't make mistakes. Further, if we define:

$$t(\vec{x}) = \begin{cases} 1 & \mathit{if}\, \vec{x} \,\epsilon\, \mathit{Class1} \\ -1 & \mathit{if}\, \vec{x} \,\epsilon\, \mathit{Class2} \end{cases}$$

It makes no difference if we make a projection from (0,1) to (-1,1). Then the label matrix **T** degenerate into a vector. Though most data can't satisfy the equality $W^T\tilde{x} = \pm 1$, the formula can indicate the class that the point belongs to. If $W^T\tilde{x}$ is positive the data is more likely belongs to Class 1, and vice verse. So the straight line $W^T\tilde{x} = 0$ divide the plain into two part and classify the data.



> Figure 1 The straight line is drawn using least square. It separates two kinds of points which have different color perfectly.

I take 80% data for training and the rest for testing. It surprising me that the precision rate is 100%. Further more, the model is stable and maintain 100% precision rate when I try to reduce the training set and increase the testing set. $W$ can be calculated quickly through vectorized data.

## Perception Algorithm

Perception algortithm corresponds to a two-class model and constructure a generalized linear model of the form:

$$y(\boldsymbol{x}) = f(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}))$$

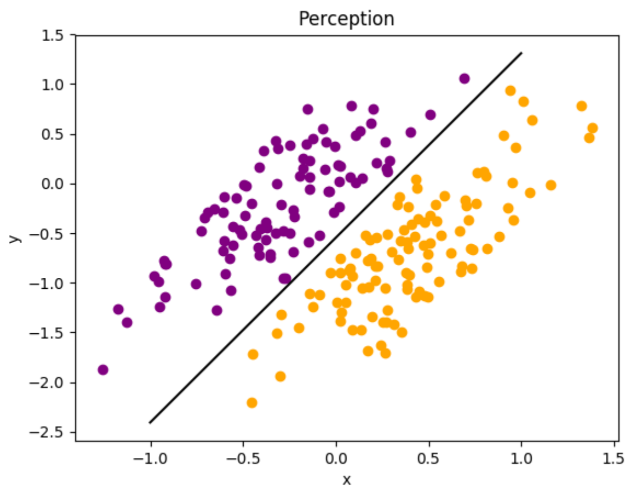
where the $\boldsymbol{\phi}(\boldsymbol{x})$ will typically include a bias component $\boldsymbol{\phi}_0(\boldsymbol{x}) = 1$ and the nonlinear activation function f(.) is given by a step function of the form

$$f(a) = \begin{cases} +1 & a \geqslant 0 \\ -1 & a < 0 \end{cases}$$

The formula above tell us that the different kinds of points. The point is misclassifying if $t(\boldsymbol{x})f(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})) < 0$. Because $f(\boldsymbol{x}^T\boldsymbol{\phi}(\boldsymbol{x}))$ and $\boldsymbol{x}^T\boldsymbol{\phi}(\boldsymbol{x})$ have the same signal, we can use $t(\boldsymbol{x})\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$ for simplicity. Here are the algorithm:

> **Perception Algorithm** *If the pattern is correctly classified, then the weighted vector remains unchanged, whereas if it is incorrectly classified, then for class $C_1$ we add the vector $\boldsymbol{\phi}(\boldsymbol{x}_n)$ onto the current estimate of weight vector* **w** *while for class $C_2$ we subtract the vector $\boldsymbol{\phi}(\boldsymbol{x}_n)$ from* **w**.

Considering the dataset is separated linearly, I choose $\phi(x) = x$ for simplicity. I set $\boldsymbol{w} = \mathbf{1}$ for initialization to avoid $\boldsymbol{w}^T\boldsymbol{x} \neq 0$, which guarantee data can be classified to either of a class and the gradient won't be equal to zero.



> Figure 2 The straight line is drawn using perception algorithm. It separates two kinds of points which have different color perfectly.

Similar to the precision test of least square, I take 80% data for training and the rest for testing. Though precision rate maintains 100%, it can't keep stable when I decrease the training set. Perception algorithm is relatively weaker than least square in this dataset.

# Part 2

## Pre-process pipeline

Calling function ***get_text_classification_datasets()***, I attain training set and test set which are `<class 'sklearn.utils.Bunch'>`. I find that the class has several attributes like data(which is the original news), filenames(which is the title or the classification of the news), target(which is the classification denoted by number) and etc. The **data** attribute is a list, whose elements are all raw text. It contains lots of whitespace and punctuation which disturb us. So I use ***str.maketrans()***, ***string.split()*** and ***string.lowcase()*** neglecting whitespace and punctuation and mapping strings into a list of words. Set **min_count=10**, then count the times that words occurs in text, and build a dictionary whose words in it appear frequently. Each word in dictionary has a unique label. Once we build a dictionary, we can map texts into multi-hot vector. If a word in dictionary appear in text, the corresponding element in vector is set to 1. If the dictionary has K words, then each text is mapped into a K-dimension vector. Combine vectors together and we attain an enormous matrix (denoted by $X$) whose element is binary. Also, the **target** attribute is the label of the text. There are 4 different class of news, which are denoted by (0,1,2,3). For each text, we build a 4-D one-hot vector $t_n$. Combine them together vertically and gain $T$. We feed text matrix $X$ and target matrix $T$ into our model for training.

## Loss function and gradient descent

If we combine those formula in index together and compute loss function:

$$
\begin{aligned}
L &= -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y} + \frac{1}{2}\lambda\|W\|^2 \\
&= -\frac{1}{N}\sum_{n=1}^{N} y_n \log softmax(z_n) + \frac{1}{2}\lambda\|W\|^2 \\
&= -\frac{1}{N}\sum_{n=1}^{N} y_n \log \frac{e^{z_n^i}}{\sum e^{z_n^i}} + \frac{1}{2}\lambda\|W\|^2 \\
&= -\frac{1}{N}\sum_{n=1}^{N} \begin{bmatrix} y_n^1 & y_n^2 & \dots & y_n^m \end{bmatrix} \begin{bmatrix} \log \frac{e^{z_n^1}}{\sum e^{z_n^i}} \\ \log \frac{e^{z_n^2}}{\sum e^{z_n^i}} \\ \dots \\ \log \frac{e^{z_n^m}}{\sum e^{z_n^i}} \end{bmatrix} + \frac{1}{2}\lambda\|W\|^2 \\
&= -\frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} y_n^m (z_n^m - \log \sum e^{z_n^m}) + \frac{1}{2}\lambda\|W\|^2 \\
&= -\frac{1}{N}\sum_{n=1}^{N} (y_n z_n - \log \sum e^{z_n^m}) + \frac{1}{2}\lambda\|W\|^2 \\
&= -\frac{1}{N}\sum_{n=1}^{N} \left[ y_n (W^\top x_n + b) - \log \sum e^{W^T x_n^m + b^m} \right] + \frac{1}{2}\lambda\|W\|^2
\end{aligned}
$$

It's we define $h_n = \frac{e^{W^T x_n + b}}{\sum e^{z_n^m}}$, and then we can have:

$$\begin{aligned}
L &= -\frac{1}{N}\sum_{n=1}^{N}\left[\, y_n^1 \quad y_n^2 \quad \dots \quad y_n^m \,\right]\begin{bmatrix}\log h_n^1 \\ \log h_n^2 \\ \dots \\ \log h_n^m\end{bmatrix} + \frac{1}{2}\lambda\|W\|^2 \\
&= -\frac{1}{N}\sum_{n=1}^{N} y_n h_n + \frac{1}{2}\lambda\|W\|^2 \qquad (1) \\
\frac{\partial L}{\partial W} &= -\frac{1}{N}\sum_{n=1}^{N}\left(y_n x_n - \frac{e^{W^T x_n + b}}{\sum e^{z_n^m}} x_n\right) + \lambda W \\
&= -\frac{1}{N}\sum_{n=1}^{N}(y_n - h_n)x_n + \lambda W \qquad (2) \\
\frac{\partial L}{\partial b} &= -\frac{1}{N}\sum_{n=1}^{N}\left(y_n - \frac{e^{W^T x_n + b}}{\sum e^{z_n^m}}\right) \\
&= -\frac{1}{N}\sum_{n=1}^{N}(y_n - h_n) \qquad (3)
\end{aligned}$$

Implement the gradient descent and then update the our parameters:

$$W \leftarrow W - \alpha\frac{\partial L}{\partial W} = W + \alpha\left[\frac{1}{N}\sum_{n=1}^{N}(y_n - h_n)x_n - \lambda W\right] \qquad (4)$$

$$b \leftarrow b - \alpha\frac{\partial L}{\partial b} = b + \alpha\frac{1}{N}\sum_{n=1}^{N}(y_n - h_n) \qquad (5)$$

We can use $(1)$~$(5)$ above updating our parameters. In practice, I combine $W$ and $b$ together, define $\hat{X} = [X, \mathbf{1}]$, and train parameters in vectorized style in numpy. It's important to note that we shouldn't regularize the bias, because the bias won't lead to over-fitting. Bias doesn't effect the model complexity, so there is no need to regularize it. There is an efficient method to check my gradient descent calculation, which was called **Gradient Check**. It is a numerical examination method through comparing back propagation computing gradient and approximate gradient.Here is the gradient check: First, we should approximate the gradient (denoted by *gradapprox*) and back propagation computing gradient (denoted by *grad*)

$$\begin{aligned}
&1.\theta^{+} = \theta + \varepsilon \\
&2.\theta^{-} = \theta + \varepsilon \\
&3.L^{+} = L(\theta^{+}) \\
&4.L^{-} = L(\theta^{-}) \\
&5.gradapprox = \frac{L^{+} - L^{-}}{2\varepsilon}
\end{aligned}$$

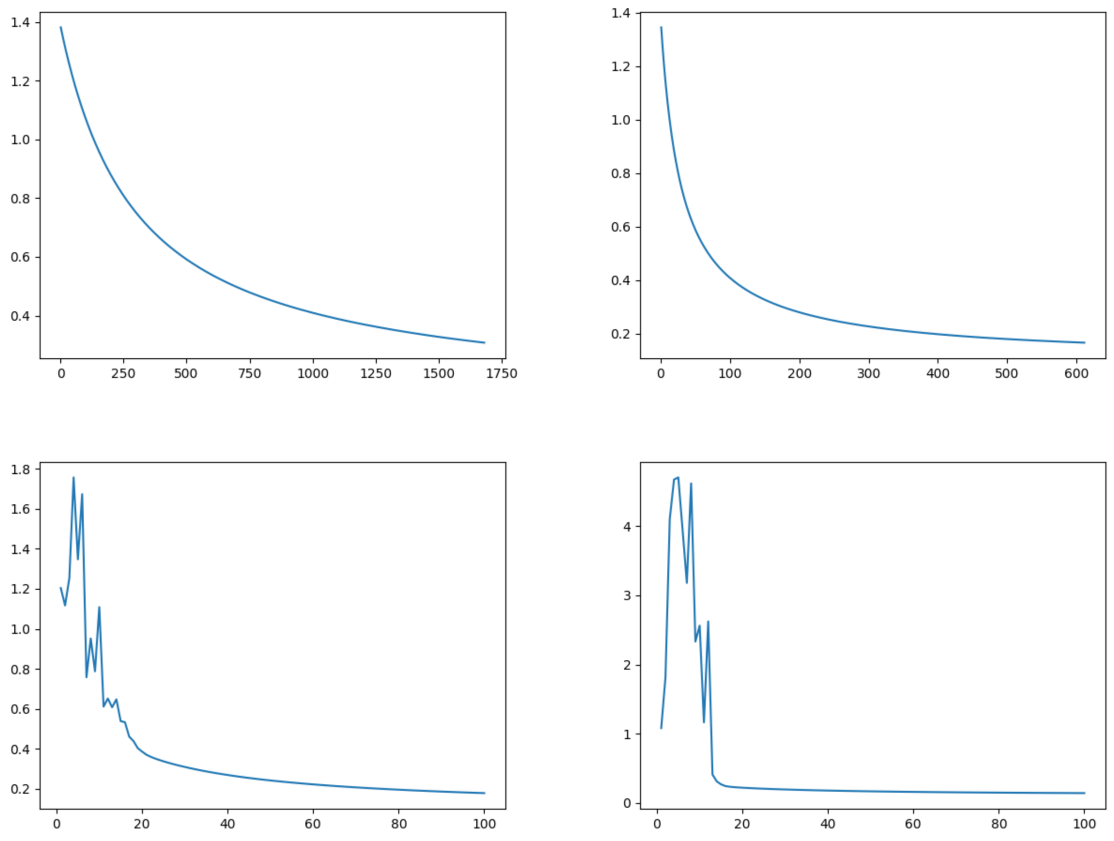
And then we can calculate the difference between them:

$$difference = \frac{\|grad - gradapprox\|^2}{\|grad\|^2 + \|gradapprox\|^2}$$

If the difference is small (for example,less than 1e-7), we can conclude the gradient is right.

## Learning rate and loss curve

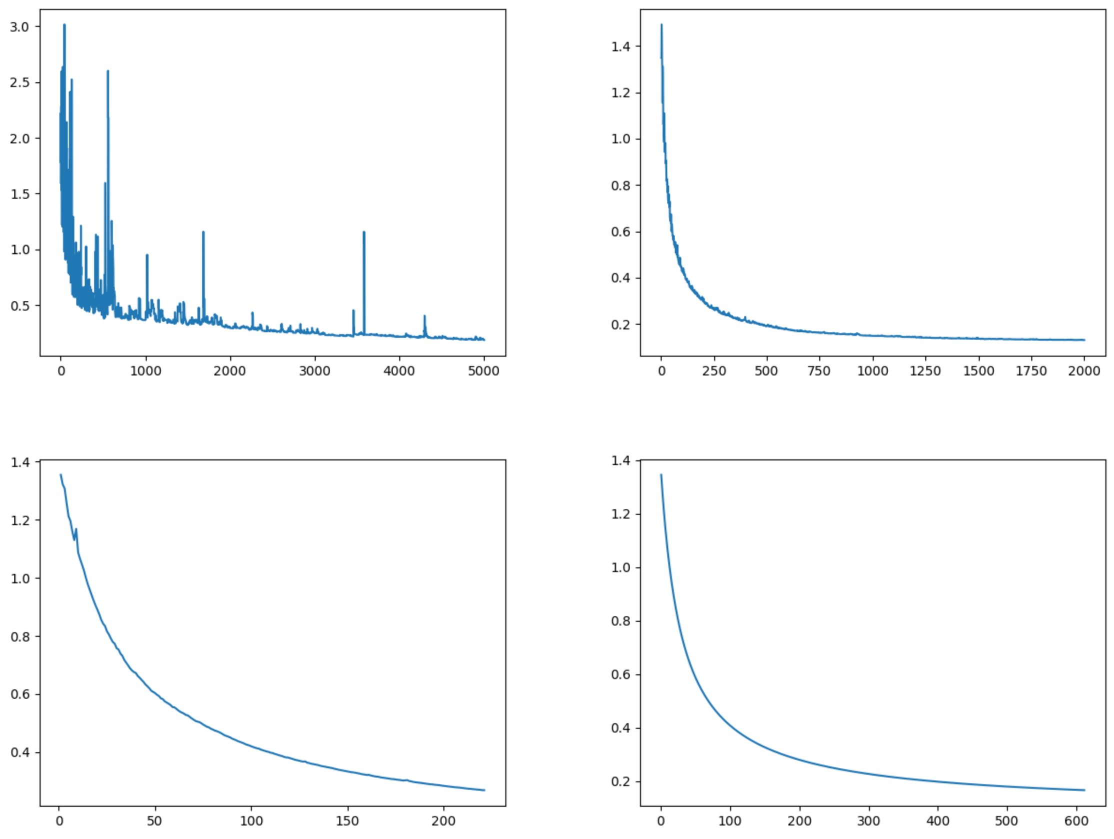
Learning rate is a hyper-parameter that guides us how to adjust the network weights through the gradient of the loss function. Generally, learning rate shouldn't be too big or too small.



> Figure 3 There are four different loss curve generated by different learning rate $\alpha$ = 0.01, 0.1, 0.5, 1. Though all loss converge finally, the curve fluctuate severely when $\alpha$ is big.

The figure above perfectly shows how learning rate affects the training process. On one hand, the loss curve will fluctuate when $\alpha$ is big. On the other hand, the curve converges slowly and it takes much time to train when $\alpha$ is small. In practice, I choose $\alpha = 0.1$, which is suitable in my view. Because loss function of softmax is a strictly convex function, it can be guaranteed that the function converge a certain value. Considering $[W, b]$ is near the global optimum point, the gradient is small and loss converge slowly, which is a waste of time. So I choose an approximate method, which we can terminate the training procedure when two consecutive train's loss is less than 1e-4.

## Stochastic gradient descent (SGD) and mini batched gradient descent (MBGD)



> Figure 4 There are four different loss curve generated by different method, the first is SGD, the second and third are MBGD which *batch size* are 10 and 100, and the fourth is full batch gradient descent (FBGD).

SGD and BGD are both efficient ways training model. They don't feed all training data into model once. Instead, they randomly pick a part of data, and the model updates like what they do in FBGD. I use ***np.random.randint()*** to choose the feed-in data, and draw the loss of whole dataset. It can be easy to find that the loss curve of SGD and MBGD are fluctuating. If we regard SGD and FBGD as special conditions of MBGD, we can conclude that the bigger batch size we set for training, the smoother loss curve it have.

### cons and pros

SGD: The stochastic gradient descent is iteratively updated by each sample. If the sample size is large, then it is possible to use only some of the samples to iterate the theta to the optimal solution. The disadvantage is that SGD has more noise than BGD, making SGD not the most optimized direction for each iteration. So although the training speed is fast, but the accuracy is reduced, it is not globally optimal. Although it contains a certain degree of randomness, it is equal to the correct derivative in terms of expectation.

MBGD: MBGD uses a small batch of samples, such as n samples, to calculate each time, so that it can reduce the variance of parameter updates, and the convergence is more stable. However, Mini-batch gradient descent does not guarantee good convergence. If the selection rate is too small, the convergence speed will be slow. If it is too large, the loss function will stop or even deviate at a minimum value.

FBGD: For convex functions, it can converge to the global minimum. The disadvantage is that in an update, the gradient is calculated for the entire data set, so it is very slow to calculate. And encountering a large number of data sets can be very tricky, and it is not possible to invest in new data to update the model in real time.

## Performance in test set

We can gain the following table when we train our model in different ways. If we set the optimization parameters $\lambda = 0.002$ :

| **Accuracy** | **SGD** | **MBGD10** | **MBGD100** | **FBGD** |
|---|---|---|---|---|
| Train(%) | 99.688 | 99.733 | 99.377 | 99.199 |
| Test(%) | 91.176 | 92.380 | 92.781 | 92.714 |

If we set $\lambda = 0.01$:

| **Accuracy** | **SGD** | **MBGD10** | **MBGD100** | **FBGD** |
|---|---|---|---|---|
| Train(%) | 95.950 | 98.887 | 99.154 | 98.575 |
| Test(%) | 85.828 | 92.714 | 92.714 | 92.580 |

We can find that the training accuracy is better when optimization parameters is small. It make sense, because $\lambda$ controls the complexity of the model, and the parameters has less limitation. We also can find that model has nearly the same accuracy predicting the class of text in test set when $\lambda$ takes different values (except SGD). In the other way, however, we can find that the accuracy of test set are around 92% to 93%, which is 7% lower than train set. The difference between two dataset contributes to the different accuracy. It indicate that it's hard to further improve the model performance.