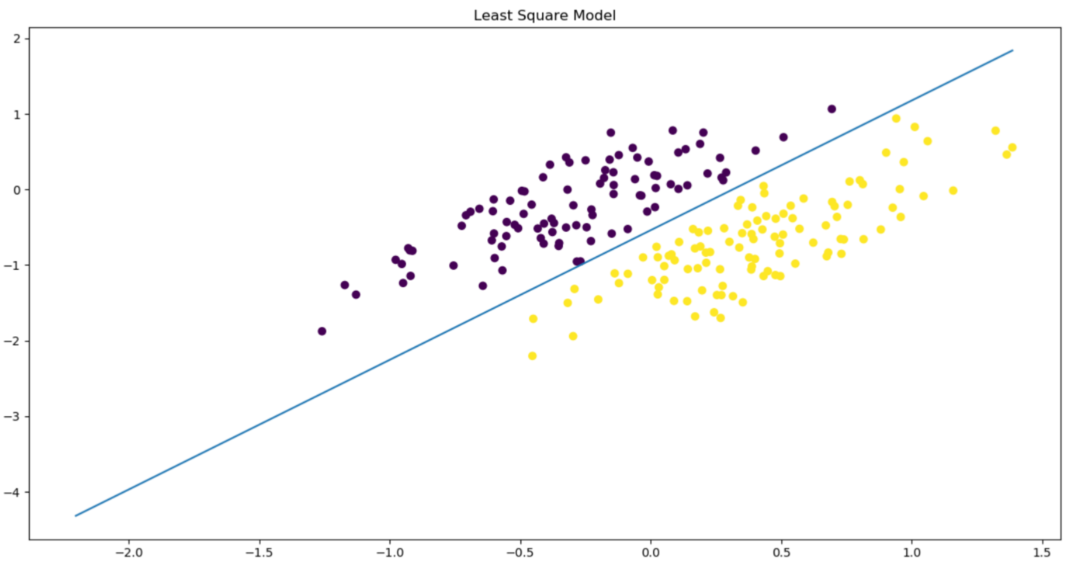
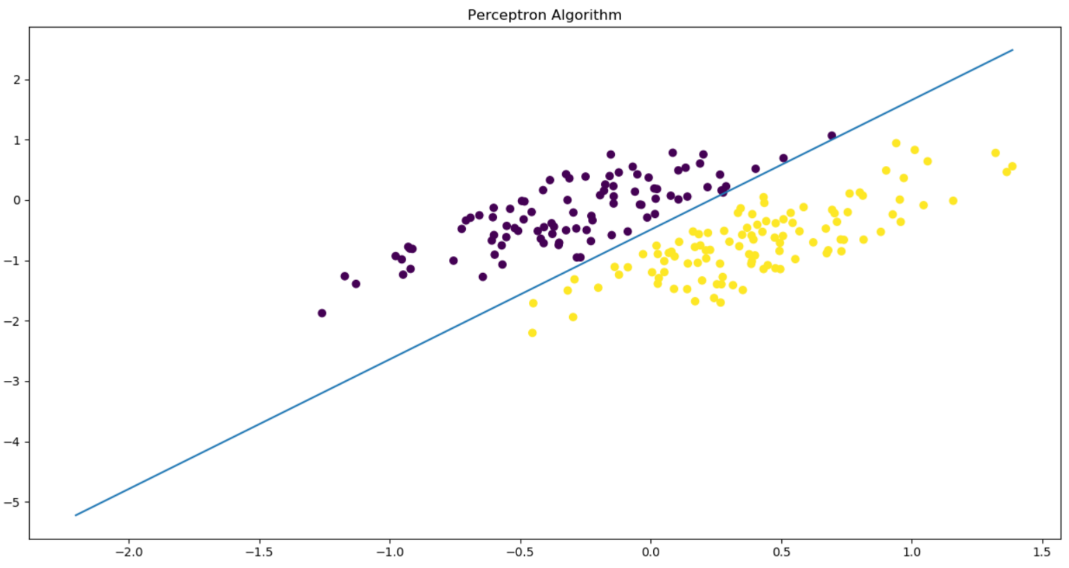
Assignment-2 Report

Part 1

The two figures are as follows.





Part 2

Requirements 1

1. Convert each character into it's lowercase and then replace each punctuation with *None* using ***str.maketrans*** .

```
trans = str.maketrans({key: None for key in string.punctuation})
train_X = [dat.lower().translate(trans) for dat in train.data]
test_X = [dat.lower().translate(trans) for dat in test.data]
```

2. Replace blank characters with spaces using **regular expression**.

```
train_X = [re.sub(pattern=r'\s', repl=' ', string=dat) for dat in train_X]
test_X = [re.sub(pattern=r'\s', repl=' ', string=dat) for dat in test_X]
```

3. Split the string with space.

```
train_X = [dat.split(' ') for dat in train_X]
test_X = [dat.split(' ') for dat in test_X]
```

3. Since sometimes a blank character is followed by another, there exist **empty strings ('')** in the split strings which need to be removed.

```
for i in range(len(train_X)):
    while '' in train_X[i]:
        train_X[i].remove('')
for i in range(len(test_X)):
    while '' in test_X[i]:
        test_X[i].remove('')
```

4. What we get now are arrays whose elements are meaningful words, so next we need to build a vocabulary. First we add all words which occur at least 10 times to a ***set*** in *Python* to remove the repeating words.

    ```
    vocab = set()
    word_count = dict()
    for i in range(len(train_X)):
        for j in range(len(train_X[i])):
            word_count[train_X[i][j]] = word_count.get(train_X[i][j], 0) + 1

        for i in range(len(train_X)):
            for j in range(len(train_X[i])):
                if word_count[train_X[i][j]] >= min_count:
                    vocab.add(train_X[i][j])
    ```

5. Then we sort the words in the *set* and we will get a array with ordered words.

    ```
    vocab = sorted(list(vocab))
    ```

6. Finally, we use ***dict*** in *Python* to map each word to it's index in the sorted array.

```
vocab_dict = dict()
for i in range(len(vocab)):
    vocab_dict[vocab[i]] = i
```

7. After building the vocabulary, we need to convert each array we get in step 4 to a multi-hot vector whose length equals to the number of distinct words in the vocabulary we built. A multi-hot vector is somehow a **sparse vector**, that is to say, only a few elements are 1 and the others are 0. The 1 elements' indexes are words' corresponding value in the vocabulary, where the words are elements of the multi-hot vector's corresponding array.

```
num_w = len(vocab)
train_X = [[0 for i in range(num_w)] for j in range(len(train_X))]
test_X = [[0 for i in range(num_w)] for j in range(len(test_X))]
for i in range(len(train_X)):
    for j in range(len(train_X[i])):
        train_X[i][vocab_dict[train_X[i][j]]] = 1

for i in range(len(test_X)):
    for j in range(len(test_X[i])):
        if vocab_dict.get(test_X[i][j]):
            test_X[i][vocab_dict[test_X[i][j]]] = 1
```

Requirements 2

1. The formula derivation is

$$x^{(i)} = [x_1 \ldots x_d, 1], X = [x^{(1)} \ldots x^{(m)}]^T$$

$$w^{(j)} = [w_{1j} \ldots w_{dj}, b_j]^T, \theta = [w^{(1)} \ldots w^{(K)}]$$

$$p(y^{(i)} = j | x^{(i)}; \theta) = \frac{\exp(x^{(i)}\theta_j)}{\sum_{k=1}^{K} \exp(x^{(i)}\theta_k)}$$

$$L(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{K} 1\{y^{(i)} = j\} \cdot \log(p(y^{(i)} = j | x^{(i)}; \theta))\right]$$

$$L(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{K} 1\{y^{(i)} = j\} \cdot (x^{(i)}\theta_j - \log(\sum_{k=1}^{K} \exp(x^{(i)}\theta_k)))\right]$$

$$\begin{aligned}\frac{\partial L(\theta)}{\partial \theta_j} &= -\frac{1}{m}\sum_{i=1}^{m}\left[\frac{\partial \sum_{j=1}^{K} 1\{y^{(i)}=j\}x^{(i)}\theta_j}{\partial \theta_j} - \frac{\partial \sum_{j=1}^{K} 1\{y^{(i)}=j\}\log(\sum_{k=1}^{K}\exp(x^{(i)}\theta_k))}{\partial \theta_j}\right]\\ &= -\frac{1}{m}\sum_{i=1}^{m}\left[1\{y^{(i)}=j\}x^{(i)} - \frac{\partial \sum_{j=1}^{K} 1\{y^{(i)}=j\}\sum_{k=1}^{K}\exp(x^{(i)}\theta_k)}{\partial \theta_j}\frac{1}{\sum_{k=1}^{K}\exp(x^{(i)}\theta_k)}\right]\\ &= -\frac{1}{m}\sum_{i=1}^{m}\left[1\{y^{(i)}=j\}x^{(i)} - \frac{x^{(i)}\exp(x^{(i)}\theta_j)}{\sum_{k=1}^{K}\exp(x^{(i)}\theta_k)}\right]\\ &= -\frac{1}{m}\sum_{i=1}^{m}x^{(i)}\left[1\{y^{(i)}=j\} - p(y^{(i)}=j|x^{(i)};\theta)\right]\end{aligned}$$

, Where *d* is the dimension of a multi-hot vector, *m* is the number of input data and *K* is the number of categories.

When using L2 regularization, just add the following term to the gradient of *W*.

$$2\lambda\sum_{i=1}^{d}\sum_{j=1}^{K}w_{ij}$$

2. Regularizing the bias term is incorrect.
3. The correctness can be proved mathematically. In addition, decrease of loss can also approve the correctness in some degree.

Requirements 3

1. If the learning rate is too large, the loss will only converge to a local optimal point, sometimes may fluctuate. If the learning rate is too small, the loss will converge very slowly.

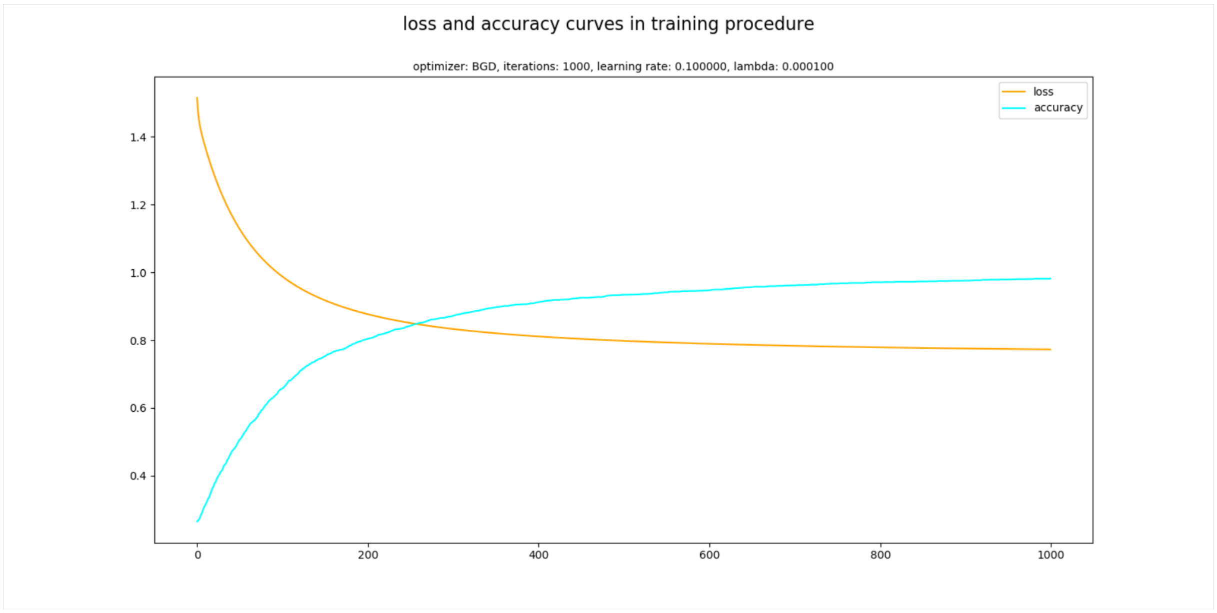
   In practice, we can set a small learning rate at first, if the loss converges too slowly, we can set the learning rate bigger by **tripling** it each time (An empirical method, e.g. 0.001, 0.003, 0.01, 0.03...). A more appealing method is that we can give the learning rate a **decay rate**, thus the learning rate can decrease slightly. That's because the rate of loss's descent is also dropping as iterations increase.

2. We can terminate the training procedure when the loss no longer decreases, more precisely, the training procedure can be terminated when the descent of loss is less than *1e-3* (The threshold is usually set empirically.).

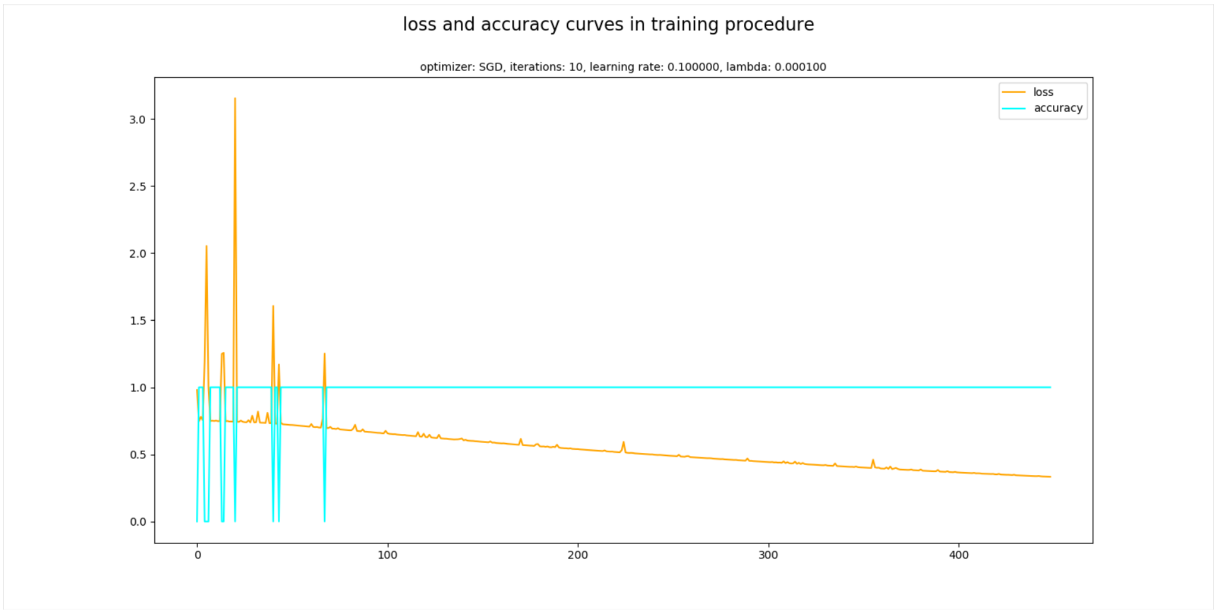
Requirements 4

** In all the following parts, (1) an **iteration** means all training data are taken as input.(e.g. In BGD, an iteration has 1 update of w. An iteration has m / batch size updates of w in MBGD and m updates of w in SGD, where m is the number of training data.), (2) the **speed of the convergence** of each method is measured by the number of w's updates.*

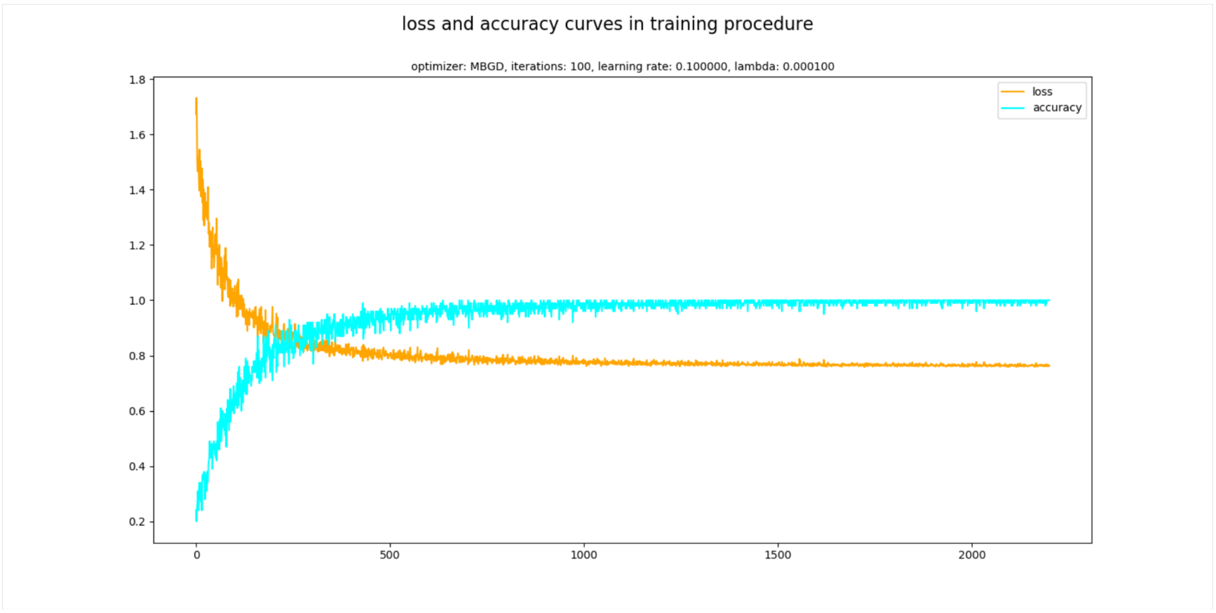
1. Descriptions of the three methods
   - For BGD strategy, the computation is of high cost especially when there are a large amount of training data. However, the BGD optimizer can converge to the global optimal for convex function.



- The training procedure of SGD is faster than BGD's. But SGD has more noise, so the training loss sometimes increases, which means it's convergence is slower. The SGD might fall into the local optimal.



- MBGD is a compromise of the BGD and the SGD. It's training procedure is faster than BGD's and it is more easily to converge when compared to SGD.



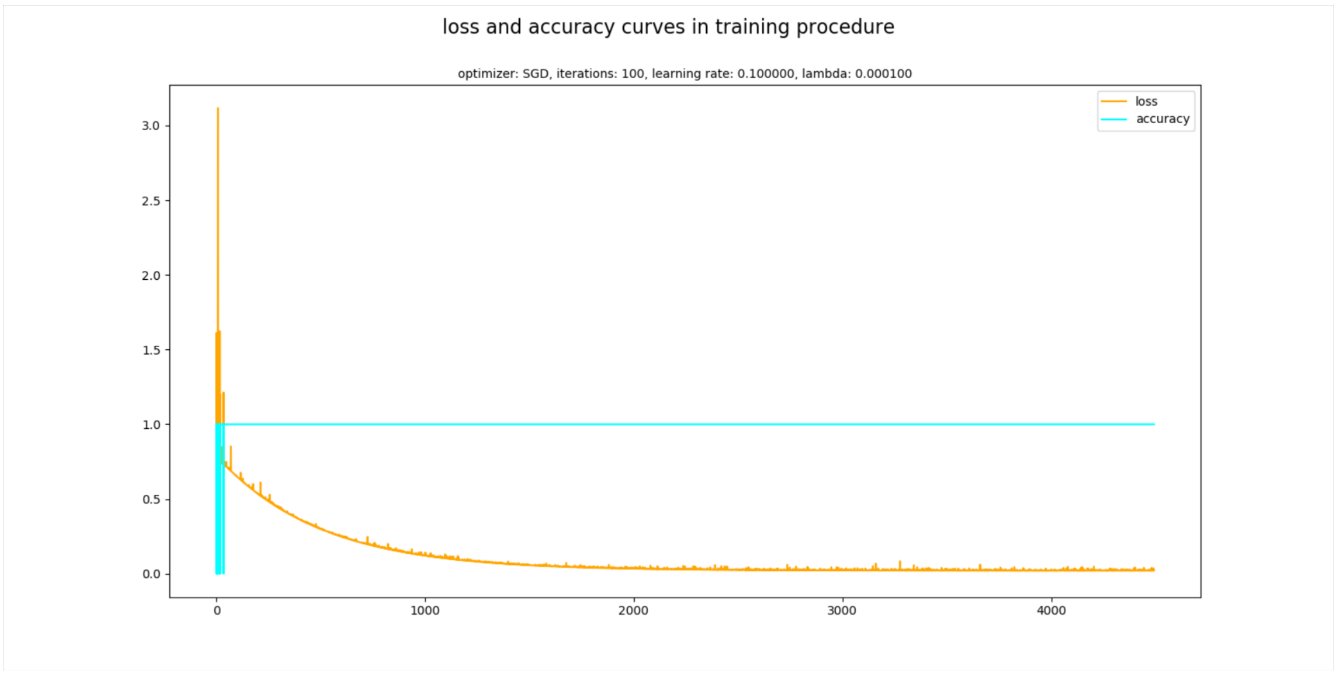
2. Pros and cons of the three methods

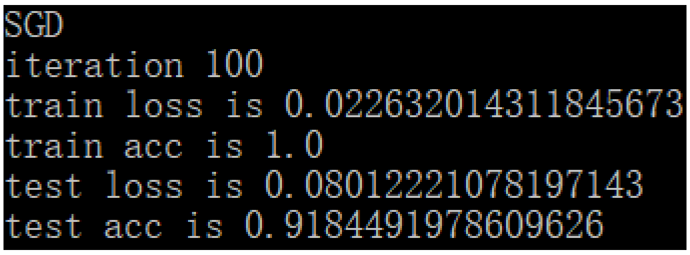
	pros	**cons**
BGD	1. can converge to global optimal when the underlying function is convex.	1. the training procedure is time-consuming. 2. sometimes the gradient has redundancy when there are similar samples in training data. 3. the model can't be updated real time.
SGD	1. the training procedure is fast. 2. there is no redundancy in the gradient. 3. the model can be updated real time by adding new training data.	1. the training loss may fluctuate because of the randomness and the frequent update of parameters. 2. might fall into local optimal or saddle point.
MBDG	1. the training procedure is fast. 2. the fluctuation of training loss is reduced.	1. the training loss fluctuates slightly. 2. might fall into local optimal or saddle point. 3. introducing a new hyper-parameter *batch size*.

Requirements 5

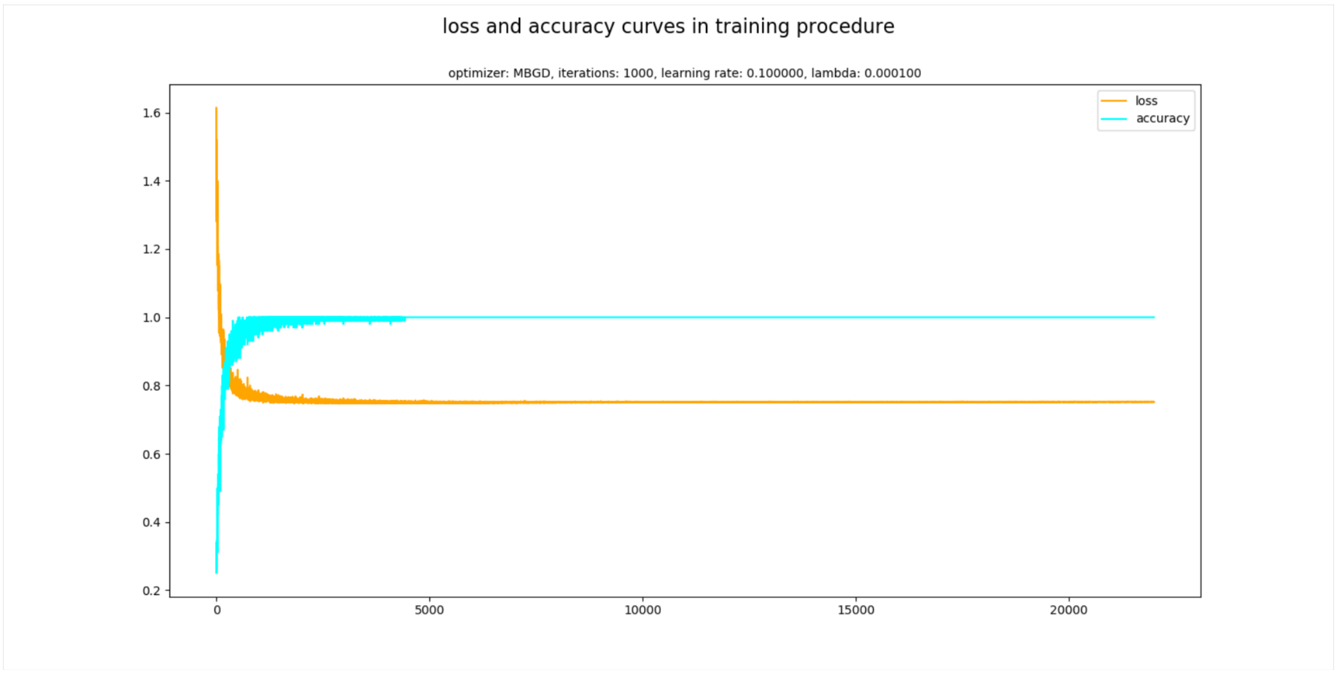
** Since the fine tuning of hyper-parameters is **not** the main purpose of this assignment, I just set the learning rate and lambda roughly and give the three methods same hyper-parameters. Sometimes the performance is influenced by the initialization of w, but it is not discussed in this part.*

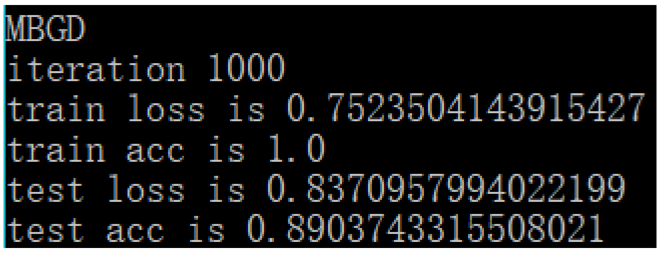
See following figures.



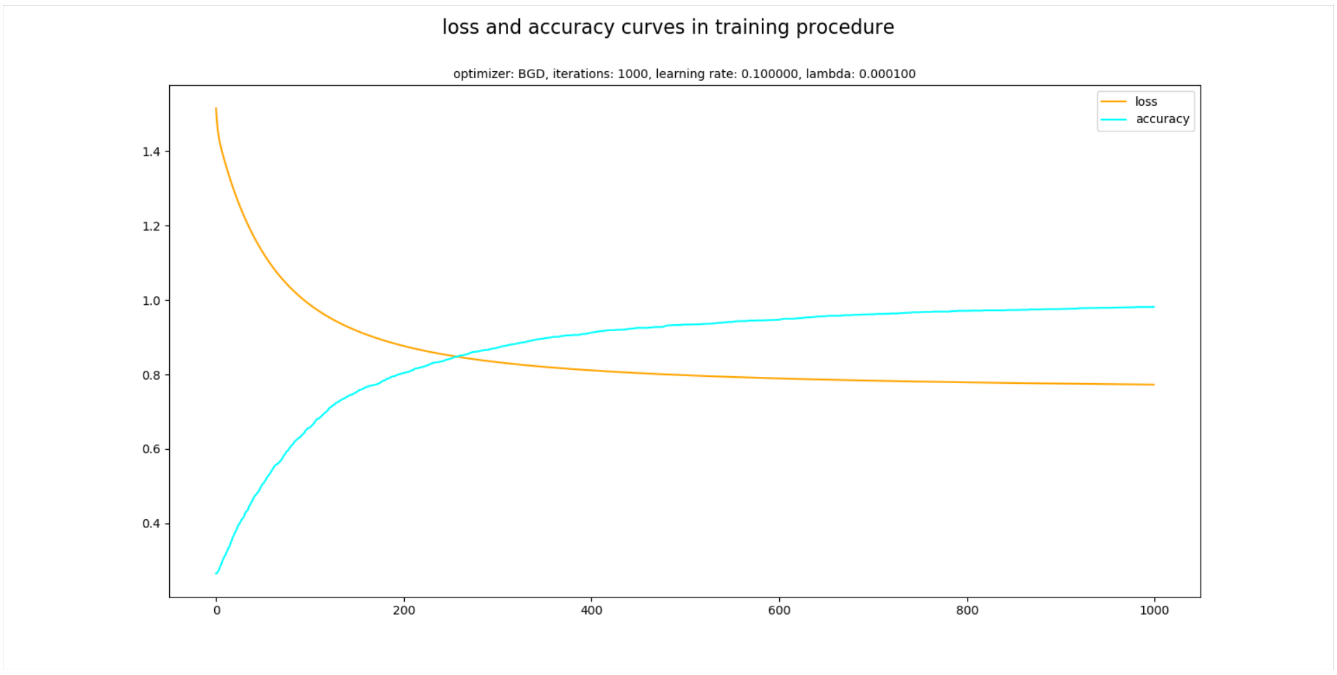


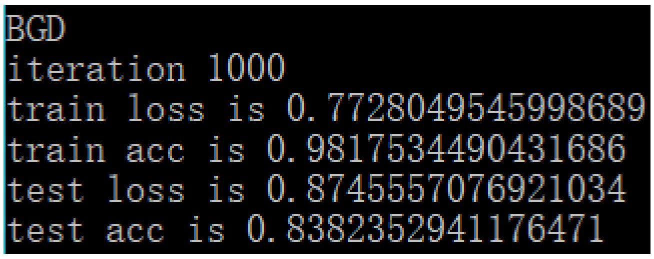
```
SGD
iteration 100
train loss is 0.022632014311845673
train acc is 1.0
test loss is 0.08012221078197143
test acc is 0.9184491978609626
```





```
MBGD
iteration 1000
train loss is 0.7523504143915427
train acc is 1.0
test loss is 0.8370957994022199
test acc is 0.8903743315508021
```





```
BGD
iteration 1000
train loss is 0.7728049545998689
train acc is 0.9817534490431686
test loss is 0.8745557076921034
test acc is 0.8382352941176471
```