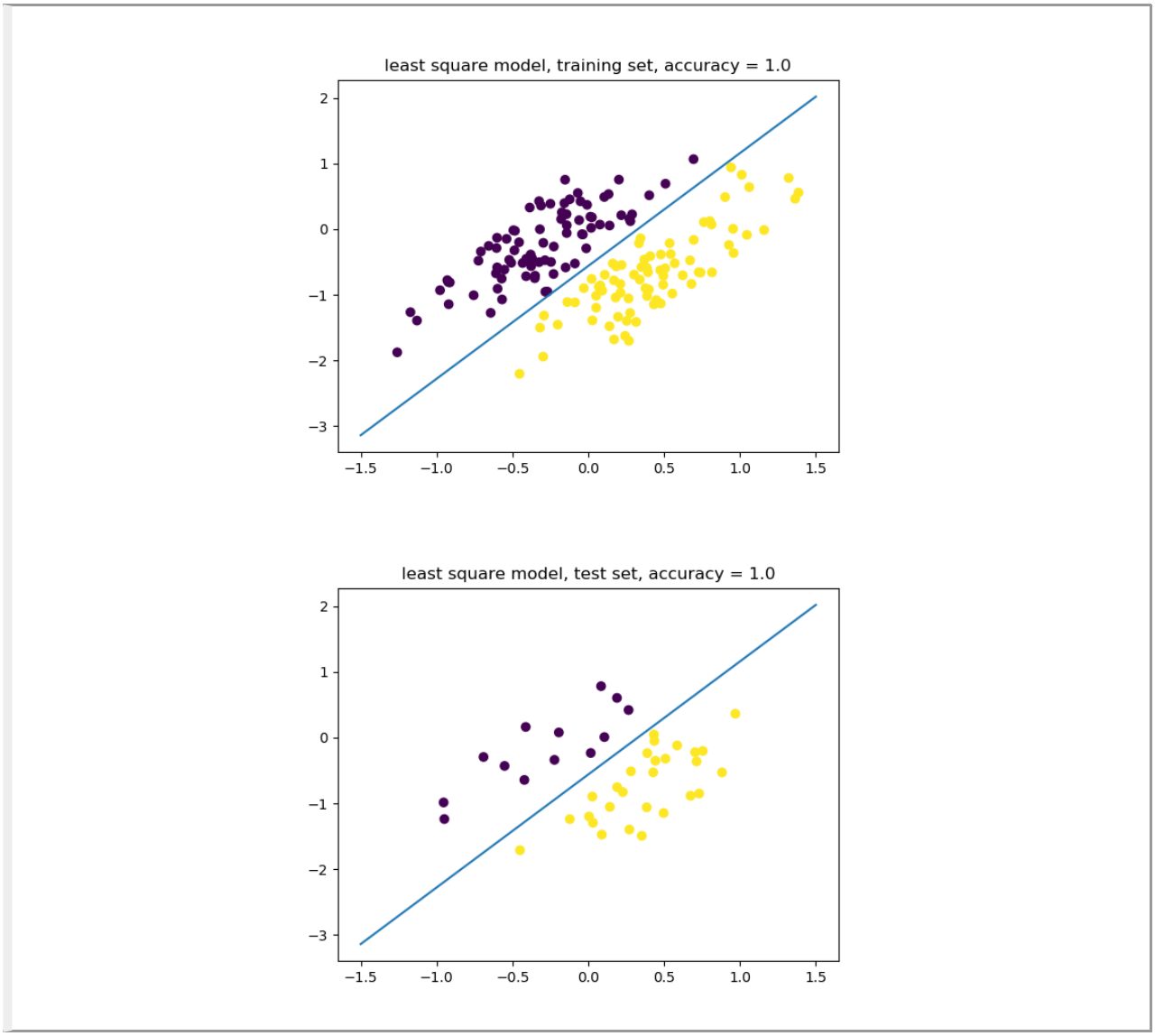
PRML Assignment-2 Report

PART 1

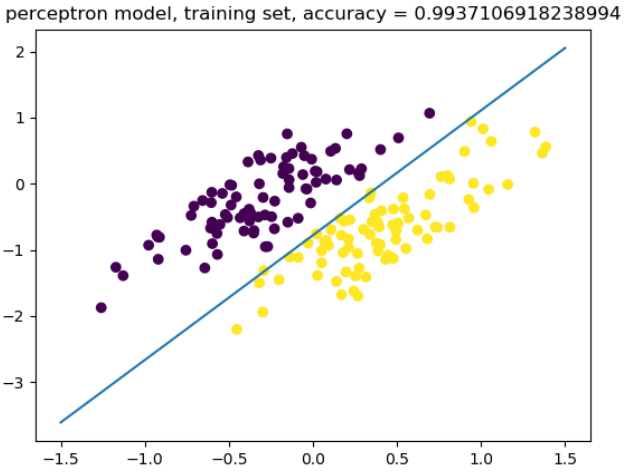
- In this part, we should firstly **preprocess** the input vectors, that is, **adding one dimension** for each vector and the value is 1.

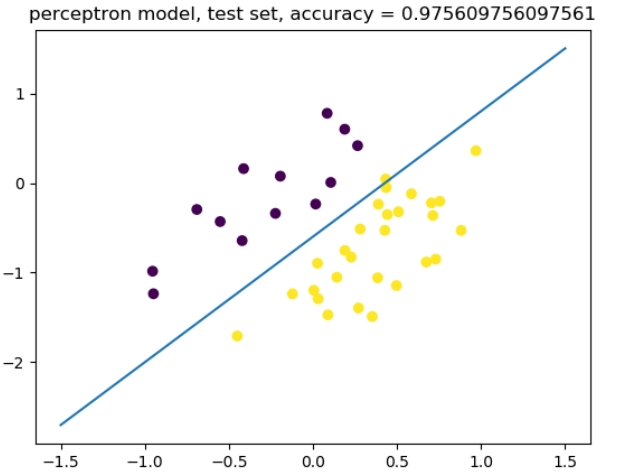
The Least Square Model



The Perceptron Model

- In the perceptron algorithm, we should firstly initialize a W, and then **modify it in every iteration** until convergence. In this assignment, I use the function `np.random.random_sample()` to initialize W, so the result of each runtime is different. As for the times of iteration, for simplification, only iterate over the training set once.





PART 2

QUESTION 1

- To transform the documents into multi-hot vector representation, I firstly preprocess the documents, including **delete punctuations and extra whitespace**, **convert uppercase to lowercase** and split every document into words. Just like the function `preprocess_dataset` .

```
124  # -----------  data preprocess  -------------
125
126  def split_string(doc):
127      new_doc = ""
128      for c in doc:
129          if string.punctuation.find(c) == -1:
130              if string.whitespace.find(c) != -1:
131                  c = ' '
132              new_doc += c
133      return new_doc.lower().split()
134
135  def preprocess_dataset(data_set):
136      return [split_string(item) for item in data_set]
137
```

- After preprocessing the documents, I can get a **vocabulary dictionary**, and then transform them into multi-hot vector representation. For simplification, I add an additional dimension for each vector, and value is 1.

```
137
138  # ----------  get multi-hot-vector  -------------
139
140  def handle_string(doc):
141    vector = [0]*(D+1)
142    vector[0] = 1
143    for item in doc:
144      if item in vocabulary_dict:
145        vector[vocabulary_dict[item]] = 1
146    return vector
147
148  def handle_dataset(data_set):
149    return np.array([handle_string(item) for item in data_set])
150
```

- As for transforming the targets into one-hot representation, I think it is not very useful for my rest operation, so I don't write it in the `source.py`.

```
286  def handle_target(target):
287    target_vector = [0]*4
288    target_vector[target] = 1
289    return target_vector
290
291  def transform_targets(input_targets):
292    return np.array([handle_target(item) for item in input_targets])
```

QUESTION 2

Compute $\frac{\partial L}{\partial W_{ij}}$, $\frac{\partial L}{\partial b_i}$

- Parameter declaration: K is the number of classifications, D is the dimension of input vector, N is the number of vectors in input dataset.
- For simplification, $W^T\vec{x}+\vec{b}$ can be equal to $\overline{W}^T\vec{\overline{x}}$, where W is a **D * K** matrix, $\vec{x}$ is a **D-dimensional** vector, $\vec{b}$ is a **K-dimensional** vector, $\overline{W}$ is a **D+1 * K** matrix, $\overline{W}_{0i}$ is equal to $\vec{b}_i$ and $\overline{W}_{ij}$ *(i > 0)* is equal to W_{i-1j}, $\vec{\overline{x}}$ is a **D+1-dimensional** vector and equal to $[1,\vec{x}^T]^T$.
- Based on 4.109 in PRML and my calculation,

$$\frac{\partial L}{\partial W_{ij}} = -\frac{1}{N}\sum_{n=1}^{N}(\overline{y}_{nj}-y_{nj})x_{ni}$$

$$\frac{\partial L}{\partial b_i} = -\frac{1}{N}\sum_{n=1}^{N}(\overline{y}_{ni}-y_{ni})$$

- For the convenience of the next calculation, in the assignment, I directly implement the **derivative of L with the transpose of W**. Codes is as follow.

```python
def L_derivative_to_Wtsp(Xi, Yi, Ti, N):
    temp_list = []
    for i in range(4):
        temp_list.append((Yi[i]-Ti[i]) * Xi)
    return (1/N) * np.array(temp_list)
```

Regularize the bias term?

- Overfitting usually requires the output of the model to be **sensitive to small changes** in the input data(i.e. to exactly interpolate the target values, you tend to need **a lot of curvature** in the fitted function). The bias parameter **b don't contribute to the curvature of the model**, so there is usually little point in regularizing them as well.
- Reference (https://stats.stackexchange.com/questions/153605/no-regularisation-term-for-bias-unit-in-neural-network/257658)

How to check the gradient calculation is correct?

- The main way to check the gradient calculation is by **comparing it against a finite-difference(FD) approximation to the gradient**. If these two answers are **close enough**, then the gradient calculation is correct.
- Reference (https://timvieira.github.io/blog/post/2017/04/21/how-to-test-gradient-implementations/)

QUESTION 3

How to determine the learning rate?

- Learning rate is one of the most important typer-parameter which tells **how far to move the weights** in the direction opposite of the gradient.
- If the learning rate is **low**, then training is more **reliable**, but optimization will **take a lot of time** because steps towards the minimum of the loss function are tiny. If the learning rate is **high**, then training **may not converge or even diverge**. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse.
- A naive approach to select a good learning rate is to **try a few different values** and see which one gives the best loss without sacrificing speed of traning.
- Another way is to **reduce learning rate as the traning progresses** by using pre-defined learning rate schedules including time-based decay, step decay and exponential decay. In this assignment, I use the **step decay**, that is droping the learning rate **every 10 iterations**. Codes are as follow.

```python
    if learning_rate > 0.01:
        learning_rate = initial_lrate * math.pow(0.95, math.floor((1+i)/10))
```

- Reference (https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1)

When to terminate the traning procedure?

- Because the loss function is a **convex function**, we can definitely find a W to **make loss the smallest**, that is the global optimum. Therefore, we can train the dataset until we find a loss that **smaller than the next loss**. Or for faster, we can just **limit a traning times** or a **loss limit**, when training enough times or the loss is smaller than the loss limit, terminate the traning procedure.

QUESTION 4

What is observed by doing the SGD and BGD?

- In SGD, we just use the cost gradient of 1 randomly input vector and **modify the W once at each iteration**. At the same times of iteration and learning rate compard to FBGD, it is **faster**, but the finally **loss is still large**, the loss curve is **vibrate** and we can't get the global optimum until we train it at more times.
- In BGD, we divide the input dataset into **some small batches**, use one of them in turn to **modify the W at each iteration**. At the same times of iteration and learning rate compard to FBGD, it is also **faster** and **get a better result** than SGD.

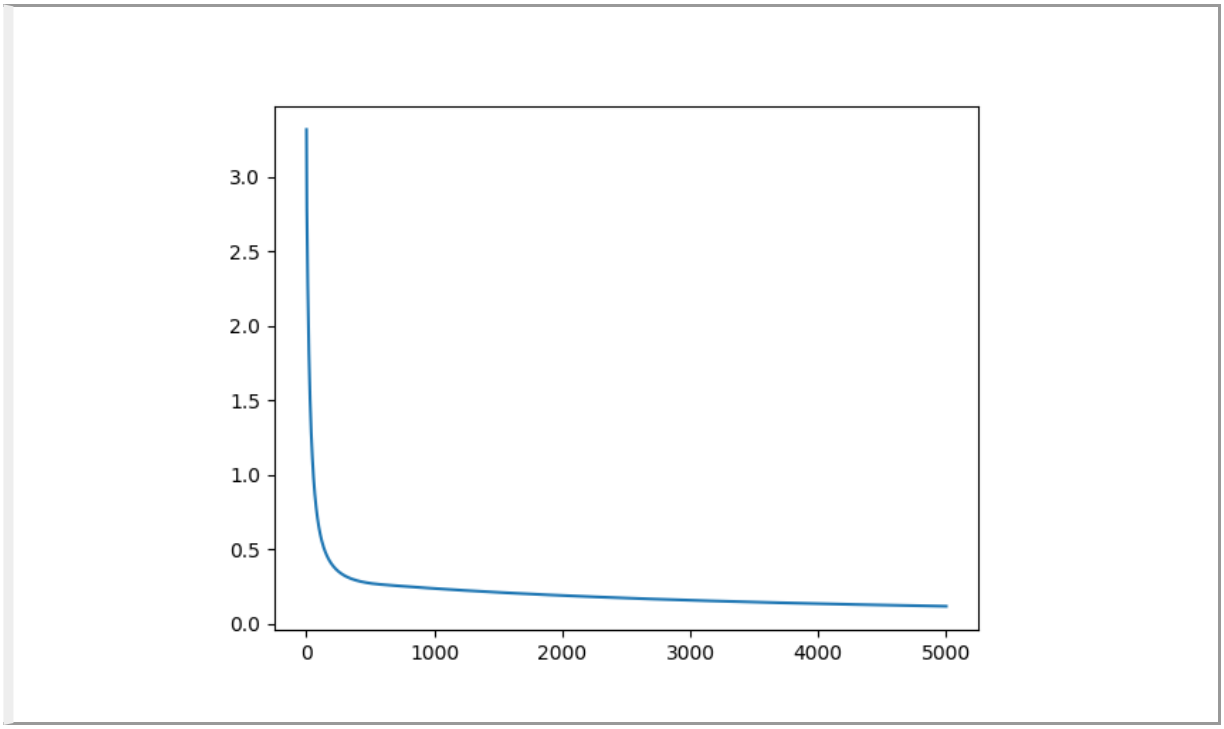
The props and cons

- **Full batch gradient descent**
  - Props: Through evaluating all traning vector, FBGD can produce a **stable loss gradient** and a **stable convergence**.
  - Cons: The stable loss gradient can sometimes result in a state of convergence that **isn't the best the model can achieve**, and when the number of input vectors is very large, the full batch gradient descent can be **very slow** to get the result.
- **Stochastic gradient descent**
  - Props: Because of just evaluating a randomly traning vector, the SGD is **faster than the others**.
  - Cons: The frequent updates are more **computationally expensive** than FBGD, also can result in **noisy gradients** and may cause the loss curve to **fluctuate** instead of slowly decrease.
- **Batched gradient descent**
  - Props: The BGD **create a balance** between the efficiency of BGD and the robustness of SGD, that is, it can be **faster at the same time guaranteed a better result**.
  - Cons: There are **no well-defined rules** to select a **proper batch**.
- Reference (https://medium.com/@ODSC/understanding-the-3-primary-types-of-gradient-descent-987590b2c36)

QUESTION 5

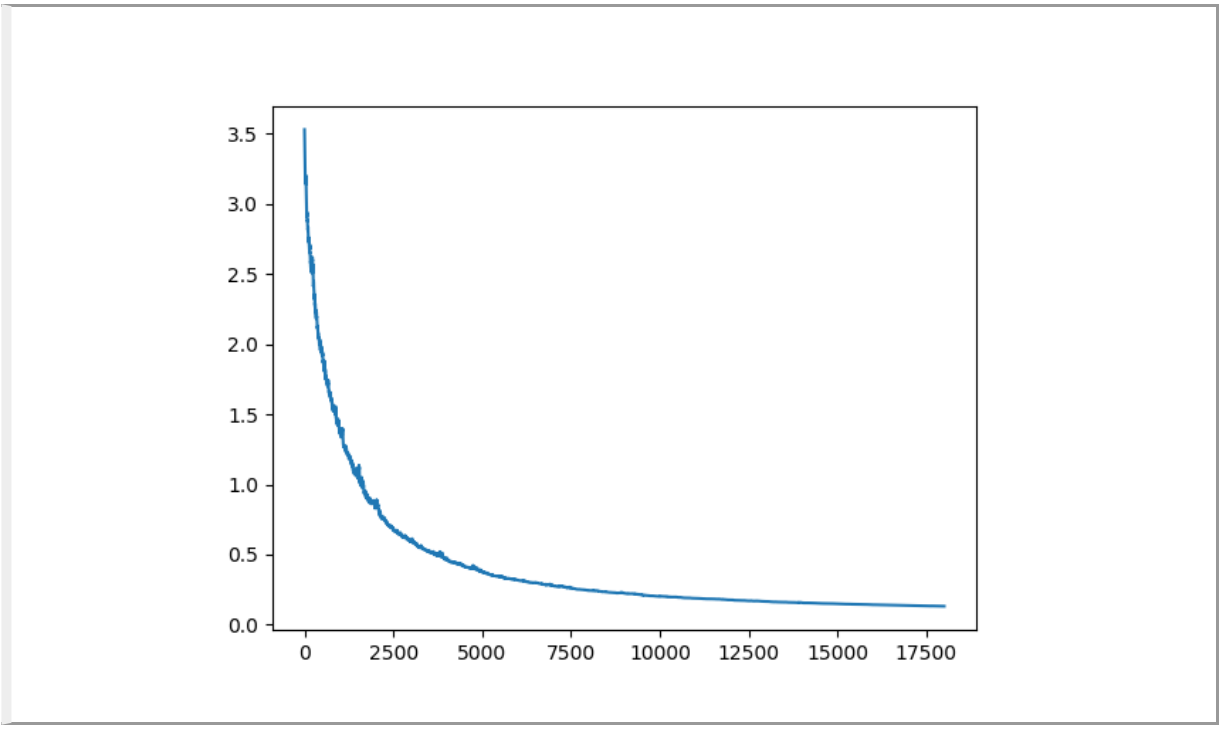
Full batch gradient descent

- **Loss of training set : 0.112**
- **Accuracy of test set : 0.81951871657754**



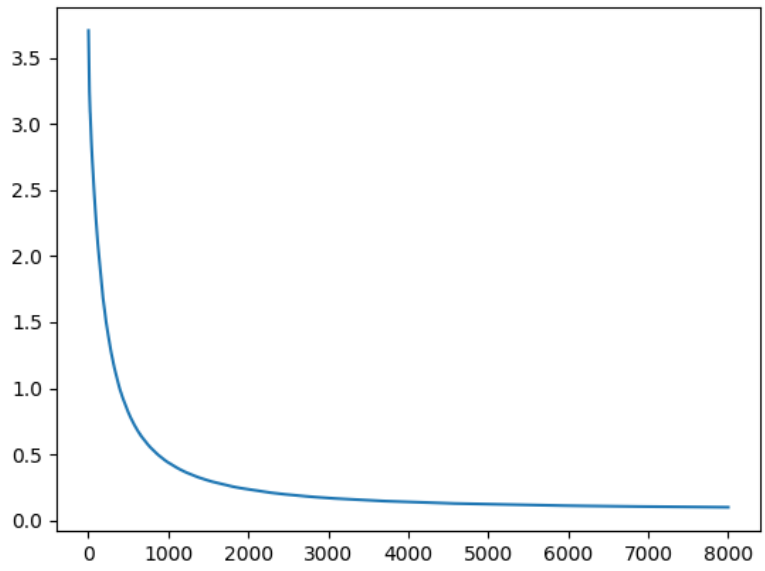
Full batch gradient descent

- **Loss of training set : 0.125**
- **Accuracy of test set : 0.8135026737967914**



Batched gradient descent

- **Loss of training set : 0.104**
- **Accuracy of test dataset : 0.8362299465240641**


3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
5000
6000
7000
8000