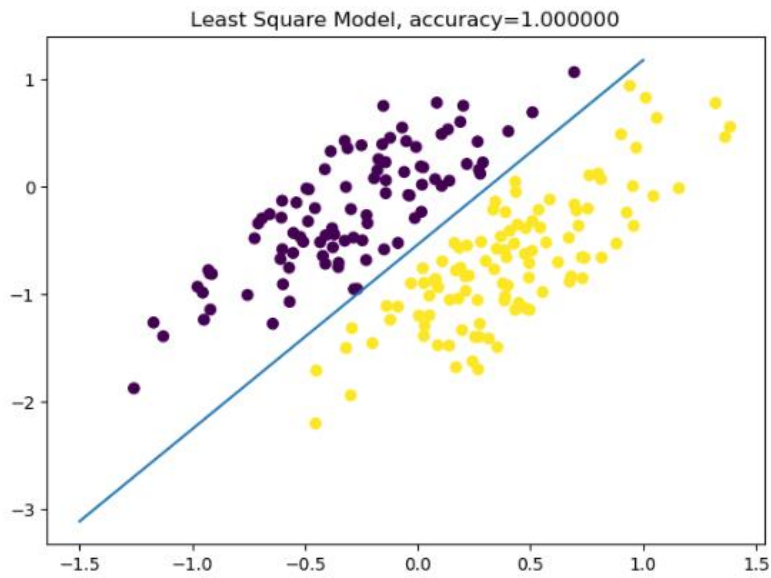


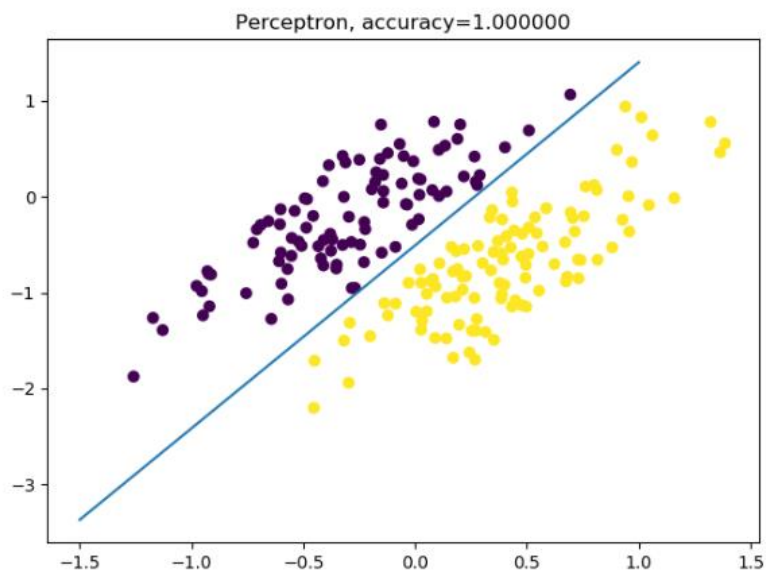
Assignment 2

Part 1

Least Square Model



Perceptron



Part 2

- The implementation of document pre-processing

```

1. # First, turn all the words into lowercase form, replace all the punctuations and whitespaces, and
2. # tokenize the train data
3. for line in self.text_train.data:
4.     line = line.lower()
5.     for c in string.punctuation:
6.         line = line.replace(c, "")
7.     for w in string.whitespace:
8.         line = line.replace(w, " ")
9.     line = line.split()
10.    self.train_vec.append(line)
11. # Next, construct a vocabulary in sorted order
12. self.vocab = sorted(list(set([w for line in self.train_vec for w in line])))
13. # Then, construct a dictionary which maps a word to its index
14. self.vocab_to_idx = {word: idx for idx, word in enumerate(self.vocab)}
15. multi_hot_vec = np.zeros((len(self.train_vec), len(self.vocab)))
16. # Finally, for all the words in vocabulary, if it presents in one sentence, mark it in its
17. # corresponding position
18. for i in range(len(self.train_vec)):
19.     for j in range(len(self.train_vec[i])):
20.         multi_hot_vec[i][self.vocab_to_idx[self.train_vec[i][j]]] = 1.0
21. # Turn the target into one-hot representation
22. target_vec = np.zeros((len(self.text_train.target), 4))
23. for i in range(len(target_vec)):
24.     target_vec[i][self.text_train.target[i]] = 1.0

```

All the vocabulary is included, without the min_count limitation.

- Calculate $\frac{\partial L}{\partial w_{ij}}, \frac{\partial L}{\partial b_i}$

In order to simplify the calculation of gradient, we can **augment the input vector x by adding a dummy input $x_0=1$** . In that case, b_i can be represented as w_{i0} .

According to 4.108 in PRML, $L = -\frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K t_{nk} \ln y_{nk}$ So $\frac{\partial L}{\partial y_{nk}} = -\frac{1}{N} \frac{t_{nk}}{y_{nk}}$

Where the target vectors are represented as an $N \times K$ matrix with elements t_{nk} , and $t_{nk} = 1$ if the n -th input vector x_n belongs to Class C_k . y_{nk} is the predicted probability of x_n in Class C_k

According to 4.106 in PRML, $\frac{\partial y_k}{\partial a_j} = y_k(I_{kj} - y_j)$

$$\frac{\partial L}{\partial a_{nj}} = \sum_{k=1}^K \frac{\partial L}{\partial y_{nk}} \frac{\partial y_{nk}}{\partial a_{nj}} = -\frac{1}{N} \sum_{k=1}^K \frac{t_{nk}}{y_{nk}} y_{nk} (I_{kj} - y_{nj}) = \frac{1}{N} (y_{nj} - t_{nj})$$

Where $a_{nj} = w_j^T x_n$ is an activation, and $\forall n: \sum_k t_{nk} = 1$

$$\frac{\partial L}{\partial w_i} = \sum_{n=1}^N \frac{\partial L}{\partial a_{ni}} \frac{\partial a_{ni}}{\partial w_i} = \frac{1}{N} \sum_{n=1}^N (y_{ni} - t_{ni}) x_n$$

We get the result: $\frac{\partial L}{\partial w_{ij}} = \frac{1}{N} \sum_{n=1}^N (y_{ni} - t_{ni}) x_{nj}$

if the regularization term is added, then the result is:

$$\frac{\partial L}{\partial w_{ij}} = \frac{1}{N} \sum_{n=1}^N (y_{ni} - t_{ni}) x_{nj} + 2\lambda w_{ij}$$

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial w_{i0}} = \frac{1}{N} \sum_{n=1}^N (y_{ni} - t_{ni}) x_{n0} = \frac{1}{N} \sum_{n=1}^N (y_{ni} - t_{ni}) \text{ because } x_{n0}=1$$

- Should the bias term be regularized?

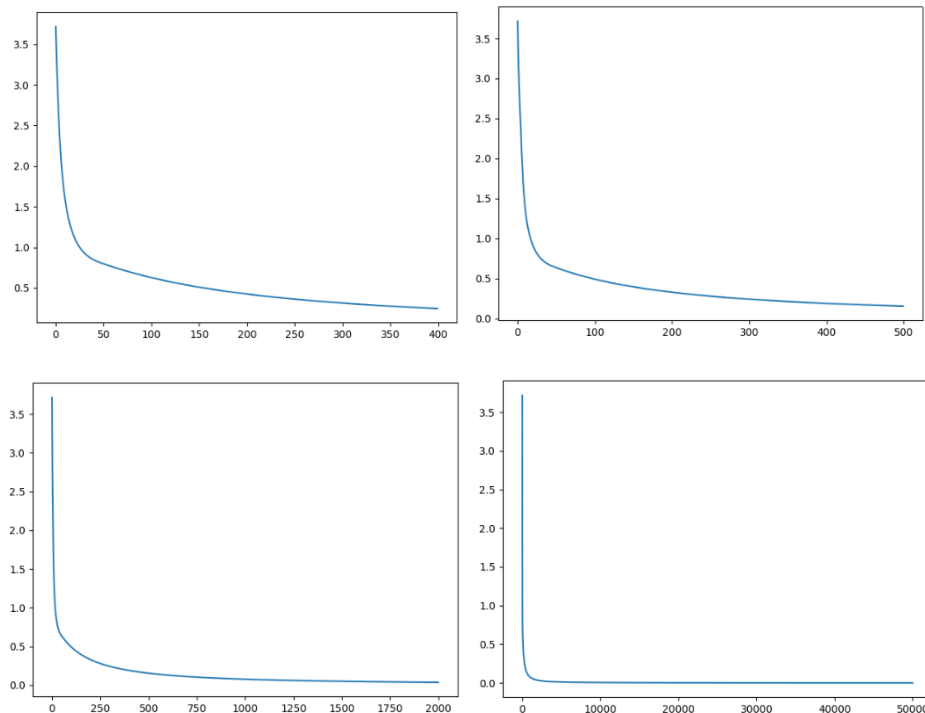
Regularization is based on the idea that overfitting on the target is caused by a being "overly specific" algorithm, which usually manifests itself by large values of the weight vectors.

The bias term merely offsets the relationship and its scale therefore is far less important to the problem. Moreover, **in case a large offset is needed for some reason**, regularizing it will prevent finding the correct relationship. In this problem, the calculated matrix W is not that large, so there is no need to add the regularization term to the gradient in order to simplify the matrix calculation.

- How to make sure the calculation of the gradient is correct?

We can use **gradient check** by calculating the **numerical gradient** to get the **approximate value of gradient** and then **compare the numerical gradient with the analytic gradient**. If the **difference between them is very small**, then we can assert our calculation is correct.

- Loss for logistic



The loss is decreasing, but still cannot converge to global optimum...

Step=50000, final loss=0.00127

- How to determine the learning rate?

Plot the loss curve as a function of the number of iterations and **make sure the loss is decreasing on every iteration**. If the learning rate is **too large**, it may be **hard for the loss to converge to global optimum**; but if the learning rate is **too small**, the **training time** may be too long. As a compromise, we can use **learning rate decay**, such as **exponential decay**. $\alpha = \alpha_0 \cdot 0.95^{epoch_num}$
When epoch_num is too large, simply assign $\alpha = 0.01$

- When to terminate the training procedure?

For the full batch gradient descent, we can terminate the training procedure when we find **the new-**

computed loss is bigger than the loss last time, which means overfitting and we already find the optimum. Or we can set a limit and **when the loss is smaller than the limit**, we terminate the training procedure.

- What do you observe by doing the other 2 different ways of gradient descent?

When we are doing **stochastic gradient descent**, we update the parameters using only **one sample**. It is much **faster**, but the loss curve tends to **vibrate** and we may **not be able to get to the global optimum**.

When we are doing **batched gradient descent**, we update the parameters using **several samples**. It is also **faster** than full batch gradient descent, and it can achieve **similar result as the full-batch way**.

- Pros and cons of each of the three different gradient update strategies?

Full batch gradient descent:

Pros:

- ① It updates the parameters using all the samples, so the calculation can be done in a **vectorized style** using numpy.
- ② The gradient descent direction is determined by the whole dataset so it can **better represent the samples**. When the object function is a convex function, it is bound to **get to a global optimum**.

Cons:

- ① When the sample size is large, every iteration will perform a whole calculation on the samples, which can be **very slow**.

Stochastic gradient descent

Pros:

- ① In every iteration, we randomly choose a training sample and optimize its loss function. So the **parameters can be updated much quicker**.

Cons:

- ① The **accuracy of the model may suffer**. Even if the target function is a convex function, SGD still **cannot converge to global optimum**.
- ② One sample cannot represent whole training dataset. We may **stuck in local optimum**.

Batched gradient descent

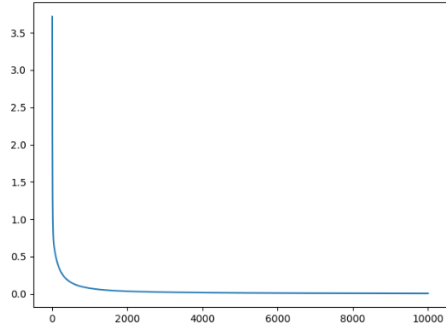
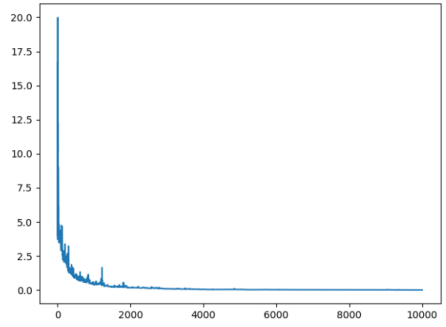
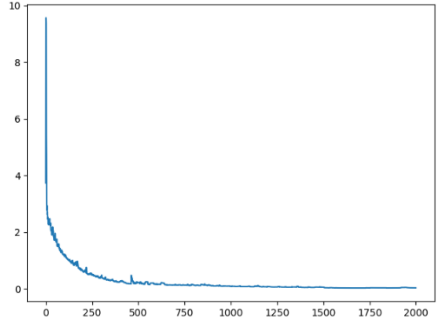
Pros:

- ① Through **matrix calculation**, BGD will not be too slow if batch size isn't too large.
- ② The **number of iterations can be greatly decreased**, and the **result is similar** to that of full batch gradient descent.

Cons:

- ① An improper batch size may result in some problems. We have to find the optimal batch size through trial.

● Result for three models

	Final loss	Loss curve	Accuracy on test set
GD	0.0063		0.860
SGD	0.0115		0.885
BGD(batch size=5)	0.0121		0.868