SLML Project-2 Classification

Shun Zhang, student ID:15300180012

November 11, 2017

1 Logistic Regression

1.1 Bayes' Rule

Now we have a D-dimensional data vector $x = (x_1, \cdots, x_D)^T$ and an associated class variable $y \in \{0, 1\}$which is Bernoulli with parameter $\alpha(i.e.\ p(y = 1) = \alpha$ and $p(y = 0) = 1 - \alpha)$. Assume that the dimensions of x are conditionally independent given y, and that the conditional likelihood of each x_i is Gaussian with μ_{i0} and μ_{i1} as the means of the two classes and σ_i as their shared standard deviation, which means

$$p(x_i|y = 1) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x-\mu_{i1})^2}{2\sigma_i^2}}$$

$$p(x_i|y = 0) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x-\mu_{i0})^2}{2\sigma_i^2}}$$

then we can derive the form of $p(y = 1|x)$ by *Bayes' rule*

$$p(y = 1|x) = \frac{p(x|y = 1)p(y = 1)}{p(x)}$$

where $p(x)$ can also be calculated according to independence among x_i

$$\begin{aligned} p(x) &= p(x|y = 1)(y = 1) + p(x|y = 0)(y = 0) \\ &= \alpha \prod_{i=1}^{D} p(x_i|y = 1) + (1 - \alpha) \prod_{i=1}^{D} p(x_i|y = 0) \\ &= \alpha \frac{1}{(\sqrt{2\pi})^D \sigma_1 \cdots \sigma_D} e^{\sum_{i=1}^{D} -\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}} + (1 - \alpha) \frac{1}{(\sqrt{2\pi})^D \sigma_1 \cdots \sigma_D} e^{\sum_{i=1}^{D} -\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}} \end{aligned}$$

then we have

$$
\begin{aligned}
p(y=1|x) &= \frac{p(x|y=1)p(y=1)}{p(x)} \\
&= \frac{1}{1+\frac{p(x|y=0)p(y=0)}{p(x|y=1)p(y=1)}} \\
&= \frac{1}{1+\frac{1-\alpha}{\alpha}\frac{\prod_{i=1}^{D} p(x_i|y=0)}{\prod_{i=1}^{D} p(x_i|y=1)}} \\
&= \frac{1}{1+\frac{1-\alpha}{\alpha}e^{\sum_{i=1}^{D} -\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2} - \sum_{i=1}^{D} -\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}}} \\
&= \frac{1}{1+e^{ln(\frac{1-\alpha}{\alpha})}e^{\sum_{i=1}^{D}\{-\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}+\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}\}}} \\
&= \frac{1}{1+e^{ln(\frac{1-\alpha}{\alpha})}e^{\sum_{i=1}^{D}\{\frac{\mu_{i0}-\mu_{i1}}{\sigma_i^2}x_i-\frac{\mu_{i0}^2+\mu_{i1}^2}{2\sigma_i^2}\}}} \\
&= \frac{1}{1+e^{-\sum_{i=1}^{D}\frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}x_i-\sum_{i=1}^{D}\frac{\mu_{i0}^2+\mu_{i1}^2}{2\sigma_i^2}+ln(\frac{1-\alpha}{\alpha})}}
\end{aligned}
$$

which is exactly the same form as *Logistic Regression*

$$
p(y=1|x) = \sigma(\omega^T x + b) = \frac{1}{1+e^{-\sum_{i=1}^{D}\omega_i x_i - b}}
$$

where

$$
\omega_i = \frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}
$$

$$
b = \sum_{i=1}^{D}\frac{\mu_{i0}^2+\mu_{i1}^2}{2\sigma_i^2} - ln\Big(\frac{1-\alpha}{\alpha}\Big)
$$

1.2 Maximum Likelihood Estimation

Now suppose we are given a training set $D = \{(x^{(1)}, y^{(1)}), \cdots, (x^{(N)}, y^{(N)})\}$. Consider a binary logistic regression classifier of the same form as before, of which the negative

log-likelihood function should be

$$\begin{aligned} E(\omega, b) &= -ln\Big(\prod_{i=1}^{N} p(y^{(i)}|x^{(i)}, \omega, b)\Big) \\ &= -ln\Big(\prod_{i=1}^{N}\Big\{y^{(i)}\frac{1}{1+e^{-\omega^T x^{(i)}-b}} + (1-y^{(i)})\frac{e^{-\omega^T x^{(i)}-b}}{1+e^{-\omega^T x^{(i)}-b}}\Big\}\Big) \end{aligned}$$

since $y^{(i)}$ is binary

$$\begin{aligned} &= -\sum_{i=1}^{N}\Big\{y^{(i)}ln\Big(\frac{1}{1+e^{-\omega^T x^{(i)}-b}}\Big) + (1-y^{(i)})ln\Big(\frac{e^{-\omega^T x^{(i)}-b}}{1+e^{-\omega^T x^{(i)}-b}}\Big)\Big\} \\ &= -\sum_{i=1}^{N}\Big\{(y^{(i)}-1)(\omega^T x^{(i)}+b) - ln\Big(1+e^{-\omega^T x^{(i)}-b}\Big)\Big\} \\ &= \sum_{i=1}^{N}\Big\{(1-y^{(i)})(\omega^T x^{(i)}+b) + ln\Big(1+e^{-\omega^T x^{(i)}-b}\Big)\Big\} \end{aligned}$$

Then we have the derivatives

$$\begin{aligned} \frac{dE(\omega, b)}{d\omega} &= \sum_{i=1}^{N}\Big\{x^{(i)}(1-y^{(i)}) - \frac{x^{(i)}e^{-\omega^T x^{(i)}-b}}{1+e^{-\omega^T x^{(i)}-b}}\Big\} \\ &= \sum_{i=1}^{N}\Big(\frac{1}{1+e^{-\omega^T x^{(i)}-b}} - y^{(i)}\Big)x^{(i)} \end{aligned}$$

$$\begin{aligned} \frac{dE(\omega, b)}{db} &= \sum_{i=1}^{N}\Big\{(1-y^{(i)}) - \frac{e^{-\omega^T x^{(i)}-b}}{1+e^{-\omega^T x^{(i)}-b}}\Big\} \\ &= \sum_{i=1}^{N}\Big(\frac{1}{1+e^{-\omega^T x^{(i)}-b}} - y^{(i)}\Big) \end{aligned}$$

1.3 L2 Regularization

Now assume that a Gaussian prior is placed on each element of $\boldsymbol{\omega}$, and b such that $p(\omega_i) = N(\omega_i|0, \frac{1}{\lambda})$ and $p(b) = N(b|0, \frac{1}{\lambda})$. Thus, a proportion to the posterior $p(\boldsymbol{\omega}, b|\mathcal{D})$ should be

$$\begin{aligned} p(\boldsymbol{\omega}, b|\mathcal{D}) &\propto p(\mathcal{D}|\boldsymbol{\omega}, b)p(\boldsymbol{\omega}, b) \\ &\propto \left(\prod_{i=1}^{N} p(y^{(i)}|x^{(i)}, \omega, b)\right)\left(\frac{\sqrt{\lambda}}{\sqrt{2\pi}}\right)^{D} e^{-\frac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega}} \frac{\sqrt{\lambda}}{\sqrt{2\pi}} e^{-\frac{1}{2}b^2} \\ &\propto \left(\prod_{i=1}^{N} p(y^{(i)}|x^{(i)}, \omega, b)\right)\left(\frac{\sqrt{\lambda}}{\sqrt{2\pi}}\right)^{D+1} e^{-\frac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega}-\frac{1}{2}b^2} \end{aligned} \tag{1}$$

and we then take the negative log of (1)

$$\begin{aligned} L(\boldsymbol{\omega}, b) &= -ln\left\{\left(\prod_{i=1}^{N} p(y^{(i)}|x^{(i)}, \omega, b)\right)\left(\frac{\sqrt{\lambda}}{\sqrt{2\pi}}\right)^{D+1} e^{-\frac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega}-\frac{1}{2}b^2}\right\} \\ &= E(\boldsymbol{\omega}, b) + \frac{\lambda}{2}\boldsymbol{\omega}^T\boldsymbol{\omega} + \frac{\lambda}{2}b^2 - \frac{D+1}{2}ln\Big(\frac{\lambda}{2\pi}\Big) \\ &= E(\boldsymbol{\omega}, b) + \frac{\lambda}{2}\sum_{i=1}^{D}\omega_i^2 + \frac{\lambda}{2}b^2 + C(\lambda) \end{aligned}$$

where $C(\lambda) = -\frac{D+1}{2}ln\big(\frac{\lambda}{2\pi}\big)$. After that, we have the derivatives

$$\begin{aligned} \frac{dL(\boldsymbol{\omega}, b)}{d\boldsymbol{\omega}} &= \frac{dE(\boldsymbol{\omega}, b)}{d\boldsymbol{\omega}} + \lambda\boldsymbol{\omega} \\ &= \sum_{i=1}^{N}\left(\frac{1}{1+e^{-\boldsymbol{\omega}^T\boldsymbol{x}^{(i)}-b}} - y^{(i)}\right)\boldsymbol{x}^{(i)} + \lambda\boldsymbol{\omega} \end{aligned}$$

$$\begin{aligned} \frac{dL(\boldsymbol{\omega}, b)}{db} &= \frac{dE(\boldsymbol{\omega}, b)}{db} + \lambda b \\ &= \sum_{i=1}^{N}\left(\frac{1}{1+e^{-\boldsymbol{\omega}^T\boldsymbol{x}^{(i)}-b}} - y^{(i)}\right) + \lambda b \end{aligned}$$

2 Digit Classification

2.1 k-Nearest Neighbours

In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. In both cases, the input consists of the

k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression.

Here, to classify a new sample, we find the closest k samples according to *L2 distance* and then we label the new sample with the same label of the majority of the k closest samples. What's more, if k is even and the two classes have the same number of samples within the k closest samples, we just take the one we assumed at first. That is to say, when we have assumed class 1, then every sample encountered with the previous case will be labeled as class 1. So we always choose k to be an odd number to avoid such 'problem'.

Now we take the *minst_train* set as our training set and *mnist_valid* as the validation set. We then run kNN for different values of $k \in \{1, 3, 5, 7, 9\}$ and plot the classification rate as a function of k on the left of Figure 1.

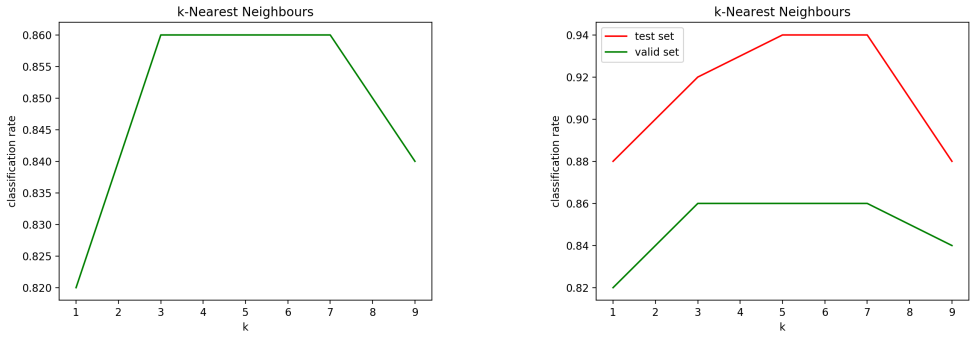

Figure 1: kNN classification rate on validation set and testing set

Since $k \in \{3, 5, 7\}$ shows the same classification rate, we first choose $k = 5$, of which the classification rate is about 86%. And the classification rate of $k = 3$ and $k = 7$ are also about 86%. The reason why $k = 5$ is chosen is that $k = 3$ is the first k that reaches the rate of 86% and $k = 7$ is the last according to the graph, which may implies that both $k = 3$ and $k = 7$ are not so stable as $k = 5$. On the other hand, too little k may leads to overfitting(considering the extreme case $k = 1$), while too large k may leads to bad performance(considering the extreme case of $k = the\ number\ of\ samples$ then every sample will be labeled as the same class). After that, we plot the classification rate of both validation set and testing set as a function of k on the right of Figure 1, from which we could see that the rates of $k = 3, 5, 7$ are about 92%, 94% and 94% respectively. Fortunately, the result proves part of my speculations before. That is, $k = 3$ may be somewhat overfitting so its rate is lower than the other two. To sum up, $k = 5$ seems to be a good choice.

2.2 Logistic Regression

To make it clear, the logistic regression here is a simple one without penalized term and a more complicated one will be covered in section 2.3.

2.2.1 Recall the logistic model

Now we consider the binary case with class 0/1. Instead of the linear regression, we turn to the *logistic function*

$$p(X) = \frac{e^{b+\omega^T X}}{1 + e^{b+\omega^T X}} \tag{2}$$

To fit the model (2), we use a method called *maximum likelihood*, which we have discussed in section 1.2. In section 1.2, we have already derived the *loss function* also called the *cross entropy* of logistic regression[Bishop. P206]

$$E(\omega, b) = \sum_{i=1}^{N} \left\{ (1 - y^{(i)})(\omega^T x^{(i)} + b) + ln\Big(1 + e^{-\omega^T x^{(i)} - b}\Big) \right\} \tag{3}$$

and also with the derivatives of it w.r.t the parameters ω and b

$$\frac{dE(\omega, b)}{d\omega} = \sum_{i=1}^{N} \left(\frac{1}{1 + e^{-\omega^T x^{(i)} - b}} - y^{(i)} \right) x^{(i)} \tag{4}$$

$$\frac{dE(\omega, b)}{db} = \sum_{i=1}^{N} \left(\frac{1}{1 + e^{-\omega^T x^{(i)} - b}} - y^{(i)} \right) \tag{5}$$

to make it simple, we concanate ω and b as one vector ω'

$$\omega' = (\omega_1, \omega_2, \cdots, \omega_D, b)^T$$

and also we can add a *dumb variable* to $x^{(i)}$

$$x'^{(i)} = ((x^{(i)})^T, 1)^T = (x_1^{(i)}, \cdots, x_D^{(i)}, 1)^T$$

then (4) and (5) can be re-written as one equation

$$\frac{dE(\omega')}{d\omega'} = \sum_{i=1}^{N} \left(\frac{1}{1 + e^{-\omega'^T x'^{(i)}}} - y^{(i)} \right) x'^{(i)} \tag{6}$$

With the derivative (6), we can then use *gradient descent* to fit the model (2).

2.2.2 Choose hyperparameters

First we complete the missing part in the *logistic code* .

Input:	ω', $train\_data$, $train\_targets$
require:	$x^{(i)} \leftarrow$ the i^{th} row of $train\_data$ $y_i \leftarrow$ the i^{th} value of $train\_targets$
loop(i)	$x'^{(i)} \leftarrow ((x^{(i)})^T, 1)^T$ $f_i \leftarrow (1 - y_i)(-\omega'^T x'^{(i)}) + ln(1 + e^{-\omega'^T x'^{(i)}})$ $df_i \leftarrow (\frac{1}{1+e^{-\omega'^T x'^{(i)}}} - y_i)x'^{(i)}$ $y_i \leftarrow sigmoid(\omega'^T x'^{(i)})$
end loop	
	$f \leftarrow \sum_{i=1}^{N} f_i$ $df \leftarrow \sum_{i=1}^{N} df_i$ $y \leftarrow (y_1, y_2, \cdots, y_N)^T$
return:	f, df, y

Then we move on to identify the parameters: **the learning rate**, **the number of iterations** and **the way of initializing the weights**.

After some experiments, we choose to initialize weights as random numbers between 0 and 0.1 in order to avoid that the loss function goes to infinity. Meanwhile, we set the random seed to be 0 for choosing the other two hyper-parameters with the same initial weights, which is

```
np.random.seed(0)
weights = np.random.rand(M + 1, 1) * 0.1
```
(in python3)

We then apply different learning rate $\in \{0.001,\ 0.01,\ 0.1,\ 1\}$ with also the same number of iterations (1000). The results are plots of correct classification rate as a function of the number of iterations in Figure 2 (where the percentage within the legends is the final classification rate).

So from the results in Figure 2 we could see that when learning rate is small, 0.001 for example, it takes quite a large number of iterations to converge as you can see in the top-left figure in Figure 2.

Note that when learning rate is 0.1 or 1, they once reached the rate 90.0% but they fall to 88.0% when the process goes on and the latter one falls much faster then the former one. So here we are, we now have learning rate 1, 0.1 and 0.01 to choose from, which is a typical trade-off between time (the number of iterations) and the performance. Finally, we choose the performance, which means **we choose the learning rate 0.01**.

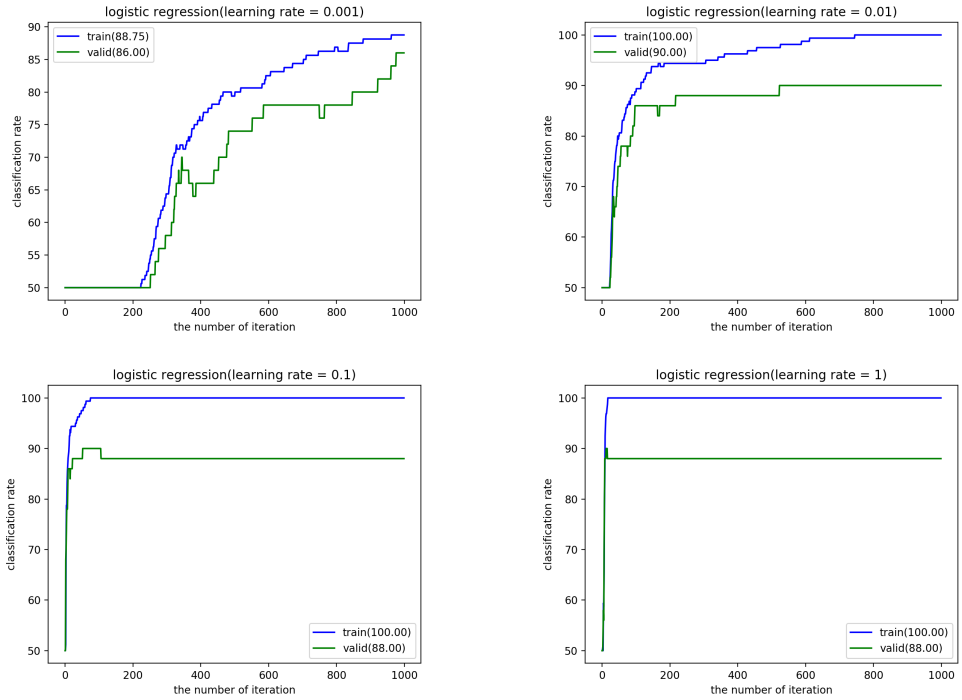

Figure 2: results for learning rate $\in \{0.001,\ 0.01,\ 0.1,\ 1\}$

Then we have to choose the certain number of iterations to ensure that the model converges when the process is over. To do this, we should run the model with different initial weights, so here we initialize the weights without random seed. (Figure 3)

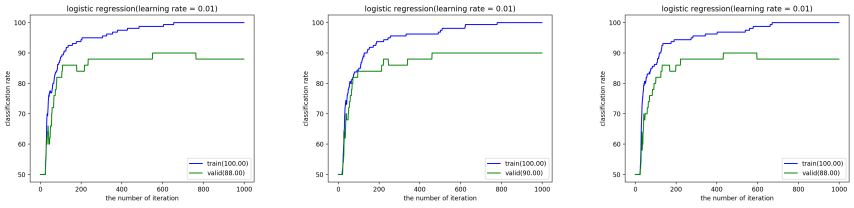

Figure 3: learning rate 0.01 with random initial weights

From the results in Figure 3, we can draw a tentative conclusion that the model converges before the 800^{th} iteration with learning rate 0.01. To be more conservative, we finally choose **the number of iterations to be 1000**, which is, by coincidence, the same as the one when we choosing the learning rate.

2.2.3 Final performance

Recall that in section 2.2.2, we've already chosen the hyper-parameters: learning rate = 0.01, the number of iterations = 1000 and we initialize the weights as random numbers between 0 and 0.1. Here we move on to have a look at the performance on testing set with different size of training set.

	cross entropy	classification error
training set	20.586	0.00%
validation set	12.205	10.00%
testing set	11.406	8.00%

Table 1: training with ***mnist_train***

	cross entropy	classification error
training set	0.243	0.00%
validation set	36.037	32.00%
testing set	32.715	26.00%

Table 2: training with ***mnist_train_small***

Note that we run the model with **random weights** between 0 and 0.1, so the results **may change a little every time**. But what we could say is that better performance on testing set depends on larger training set.

2.2.4 Cross entropy

With learning rate = 0.01, the number of iterations 1000 and random weights between 0 and 0.1, we run the model several times and draw the figure about how cross entropy changes as training processes.
Fortunately, regardless of the subtle change brought by the randomness of the weights' initialization, we could say that the results barely change, which may be a confirmation to our hyper-parameters in section 2.2.2 to some extent.

2.3 Penalized Logistic Regression

2.3.1 Penalized loss function

Recall that in section 1.3, we've derived the penalized loss function as $L(\boldsymbol{\omega}, b)$. However, here we do not want to penalize the *bias term* and we can omit the term $C(\lambda)$ because it

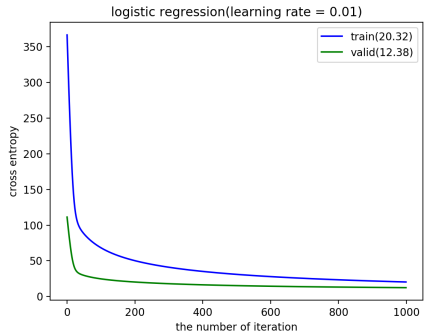

Figure 4: training with ***mnist_train***

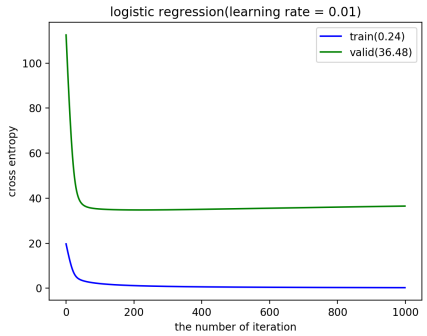

Figure 5: training with ***mnist_train_small***

is irrelevant to weights. So the new loss function should be

$$L^*(\boldsymbol{\omega}, b) = E(\boldsymbol{\omega}, b) + \frac{\lambda}{2}\sum_{i=1}^{D}\omega_i^2$$

along with the derivatives of loss function should be

$$\frac{dL^*(\boldsymbol{\omega}, b)}{d\boldsymbol{\omega}} = \sum_{i=1}^{N}\left(\frac{1}{1+e^{-\boldsymbol{\omega}^T\boldsymbol{x}^{(i)}-b}} - y^{(i)}\right)\boldsymbol{x}^{(i)} + \lambda\boldsymbol{\omega}$$

$$\frac{dL^*(\boldsymbol{\omega}, b)}{db} = \sum_{i=1}^{N}\left(\frac{1}{1+e^{-\boldsymbol{\omega}^T\boldsymbol{x}^{(i)}-b}} - y^{(i)}\right)$$

where the derivative of b is a bit different from the one in section 1.3 because it only controls the height of the function but not its complexity. With the new loss function and the new derivatives, we could move on to fit the penalized model.

2.3.2 Choose the penalty parameter λ

Before we start, we should make it clear that it is more rigorous to re-choose the previous three hyper-parameters for every λ. However, here we just simply keep the former ones and focus on choosing the best λ. To do the comparison between $\lambda \in \{0.001, 0.01, 0.1, 1.0\}$, we run two nested loops: The outer loop is over values of λ. The inner loop is over 10 re-runs. Finally, we take the average of the evaluation metrics, **final cross entropy and classification error** over the different re-runs. The results are shown as follow, along with the performance on testing set once we choose the λ.

- **training with *mnist_train***

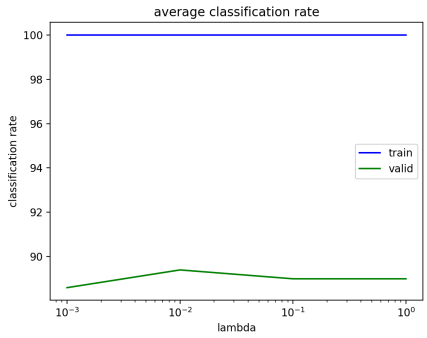

Figure 6: average classification rate

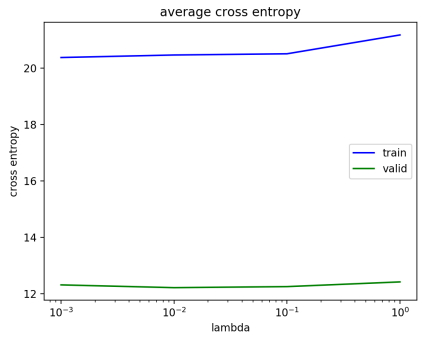

Figure 7: average cross entropy

From figure 6 we could see that the average classification rate of training set among different λ is about 100% (while for $\lambda = 1.0$, it's about 99.94%), which means the four models all finally converge. The average classification rate of validation set among different λ first goes up and then goes down. The reason why it first goes up is that larger λ implies better generalization and it then goes down may because too large λ induces a too generalized model with poor performance.
From figure 7 we could see that the average cross entropy of training set among different λ keeps going up when λ increases and there's a 'jump' from 0.1 to 1.0, which shares the same reason that when λ is too large, the model performs poorly. The average cross entropy of validation set among different λ first goes down and then goes up, for which the reason is also the same.
So, here with training set mnist_train, we would choose $\lambda = 0.01$ for its highest classification rate and also almost lowest cross entropy on validation set.

- **training with *mnist_train_small***

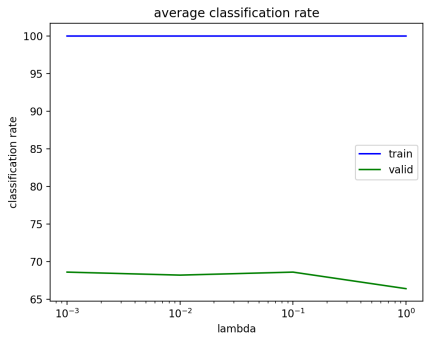

Figure 8: average classification rate

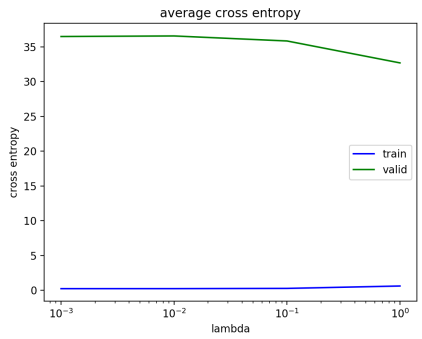

Figure 9: average cross entropy

From figure 8 we could see that the average classification rate of training set among different λ is also about 100%. The average classification rate of validation set among different λ drops greatly, by comparison, from 0.1 to 1.0, whose reason has already been discussed.
From figure 9 we could see that the average cross entropy of training set among different λ is quite close to 0. The average cross entropy of validation set among different λ first keeps going down, which is rare in the case of training with mnist_train.
So, here with training set mnist_train_small, we would choose $\lambda = 0.1$ for its highest classification rate and comparatively lower cross entropy on validation set.

- **performance on testing set**

Penalized	λ	test error
mnist_train	0.01	8.00%
mnist_train_small	0.1	24.00%
Unpenalized		test error
mnist_train		8.00%
mnist_train_small		26.00%

Table 3: test error with different training set and λ

From Table 3, we could see that penalized model training with mnist_train_small performs better on testing set. The reason may be that model with penalty compromises the problem of overfitting caused by the small size of training data, to some extent, and returns a better performance on testing set.

2.4 Naive Bayes

- **a quick review**

In this question we will experiment with a binary Naive Bayes classifier. In a Naive Bayes classifier, the conditional distribution for example $\boldsymbol{x} \in R^d$ to take on class c (out of K different classes) is defined by

$$p(c|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|c)p(c)}{\sum_{k=1}^{K} p(\boldsymbol{x}|k)p(k)}$$

where $p(\boldsymbol{x}|c) = \prod_{i=1}^{d} p(x_i|c)$ according to the *Naive Bayes assumption*. In this question, we model $p(x_i|c)$ as a Gaussian for each i as

$$p(x_i|c) = \mathcal{N}(x_i|\mu_{ic}, \sigma_{ic}^2) = \frac{1}{\sqrt{2\pi\sigma_{ic}^2}} exp\bigg(- \frac{(x_i - \mu_{ic})^2}{2\sigma_{ic}^2} \bigg)$$

The prior distribution $p(c)$ and parameters $\mu_c = (\mu_{1c}, \cdots, \mu_{dc})^T, \sigma_c^2 = (\sigma_{1c}^2, \cdots, \sigma_{dc}^2)$ for all c are learned on a training set using *maximum likelihood estimation.*

- **visualization**

  After completing the pipeline of training, testing a Naive Bayes classifier, we have the visualization of the mean and variance of the model.

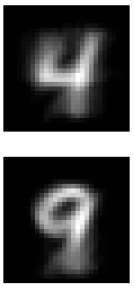

Figure 10: mean of the model

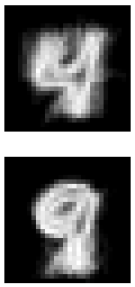

Figure 11: variance of the model

- **a brief comment**

  Considering the mathematical concept of mean, Figure 10 could be interpreted as that if a certain pixel appears to be 'white' for more times through out the training set then the same pixel in Figure 10 becomes 'more white' or 'more lightened'. Simply, mean of the model is a representative of the whole training set or, you can say, the image of how sure you are with a pixel. The brighter, more sure.

  Also considering the mathematical concept of variance, Figure 11 could be interpreted as that if a certain pixel appears to be 'the same white' or 'the same dark' for more times within the training set then the same pixel in Figure 11 becomes 'more dark' or 'less lightened'. So, by contrast, if a certain pixel appears to be 'white' or 'dark' almost equally, then the same pixel in Figure 11 appears to be 'more white' or 'more lightened'. Simply, variance of the model shows the 'edge' of the class, which is the edge between the almost sure 'white area' and the almost sure 'dark area'.

2.5 Compare k-NN, Logistic Regression, and Naive Bayes

Well, its really hard to say which model is better. Just as the saying goes, 'There's no free lunch.' So, here we simply compare the three models according to their performance on the testing data with different training data: ***mnist_train***(Table 4) and ***mnist_train_small***(Table 5).

Table 4: trainning data: ***mnist_train***

	test classification rate
k-Nearest Neighbours	94.0%
Logistic Regression	92.0%
Penalized Logistic Regression	92.0%
Naive Bayes	80.0%

Table 5: trainning data: ***mnist_train_small***

	test classification rate
k-Nearest Neighbours	66.0%
Logistic Regression	74.0%
Penalized Logistic Regression	76.0%
Naive Bayes	66.0%

From the tables above, we find that kNN performs much better when there is enough data for training. Logistic regression performs well when training set is small, of which the penalized version performs a little bit better than the unpenalized one. Naive Bayes shows a comparatively poor performance on both of the two cases.
To sum up, we'd better choose kNN or logistic regression when training set is large and choose penalized logistic regression when training set is small.

3 Stochastic Subgradient Methods

Stochastic gradient methods are an appealing training strategy when the number of training examples n is very large. They have appealing theoretical properties in terms of the training and testing objective, even when n is infinite. However, they can prove difficult to get working in practice, and are much more reliable than the methods above. In this question, we?ll explore how different techniques affect the performance of stochastic gradient method. We?ll explore using different step-size selection schemes and using the average of pervious iterations.

3.1 Averaging Strategy

Figure 12 is the process that reports the performance on the basis of current weights, while Figure 13 is based on averaging weights. Apparently, though Figure 13 is much more smooth than Figure 12, it performs even worse.

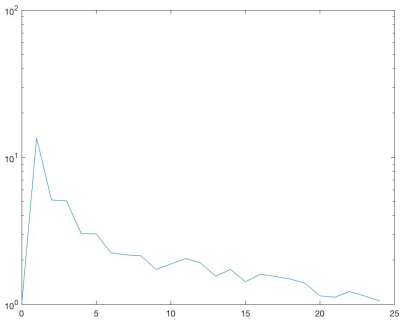

Figure 12: use current weights

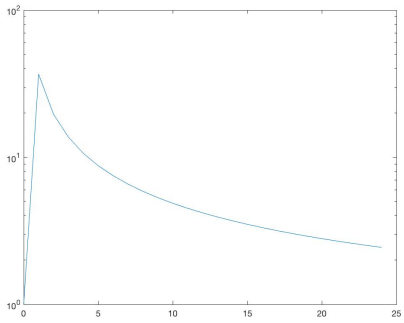

Figure 13: use **averaging** weights

3.2 'Second-half' Averaging Strategy

The reason why model with averaging strategy performs worse than the former one may be that it places just as much weight on the early iterations (w0, w1, and so on) as it does on the later iterations. A common variant is to exclude the early iterations from the averaging process. For example, we can start averaging once we have get half way to $maxIter$.

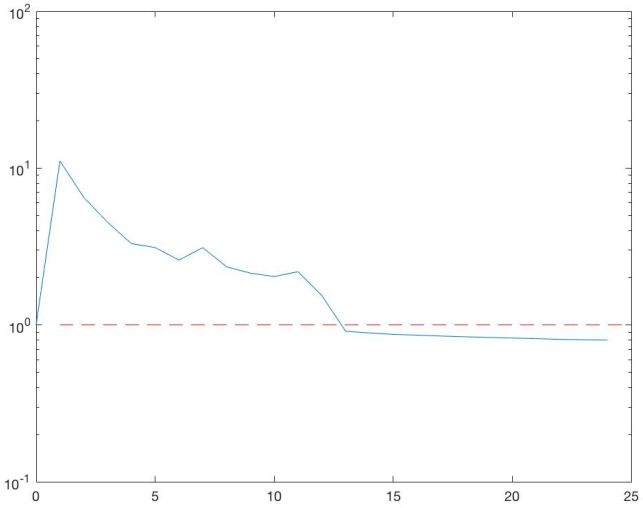

Figure 14: averaging half way to $maxIter$

The performance in Figure 14 looks much better than the previous two.