

# Project-2

Tianran Zhang 15300180104

November 19, 2017

## Contents

<b>1</b>	<b>Logistic Regression</b>	<b>2</b>
1.1	Bayes' Rule . . . . .	2
1.2	Maximum Likelihood Estimation . . . . .	3
1.3	L2 Regularization . . . . .	3
<b>2</b>	<b>Digit Classification</b>	<b>4</b>
2.1	k-Nearest Neighbours . . . . .	4
2.2	Logistic regression . . . . .	6
2.2.1	Implementation the missing part of logistic . . . . .	6
2.2.2	Hyperparameter Setting . . . . .	7
2.3	Penalized logistic regression . . . . .	13
2.4	Naive Bayes . . . . .	14
2.5	Compare k-NN, Logistic Regression, and Naive Bayes . . . . .	15
<b>3</b>	<b>Stochastic Sub Gradient Methods</b>	<b>16</b>
3.1	Averaging of $w^t$ . . . . .	16
3.2	Second-Half Averaging . . . . .	16
3.3	Modify of SVM . . . . .	17
<b>4</b>	<b>Codes for report</b>	<b>18</b>
4.1	knn.m . . . . .	18
4.2	chosenK.m . . . . .	18
4.3	logistic_predict.m . . . . .	20
4.4	logistic.m . . . . .	20
4.5	chosen_rate . . . . .	21
4.6	logistic_pen.m . . . . .	23
4.7	logistic_regression_template.m . . . . .	24
4.8	logistic_regression_penalized.m . . . . .	27
4.9	run_nb.m . . . . .	31
4.10	svmAvg.m . . . . .	31
4.11	svmAvg1.m . . . . .	33
4.12	svm_new.m . . . . .	35

# 1 Logistic Regression

## 1.1 Bayes' Rule

$$x = (x_1, \dots, x_D)^T \quad (1)$$

$$p(y = 1) = \alpha \quad (2)$$

$$p(y = 0) = 1 - \alpha \quad (3)$$

$$p(x_i|y = 0) = N(\mu_{i0}, \sigma_i^2) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left\{-\frac{(x - \mu_{i0})^2}{2\sigma_i^2}\right\} \quad (4)$$

$$p(x_i|y = 1) = N(\mu_{i1}, \sigma_i^2) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left\{-\frac{(x - \mu_{i1})^2}{2\sigma_i^2}\right\} \quad (5)$$

$$p(y = 1|x) = \frac{p(x|y = 1)p(y = 1)}{p(x|y = 1)p(y = 1) + p(x|y = 0)p(y = 0)} \quad (6)$$

$$= \frac{\alpha p(x|y = 1)}{\alpha p(x|y = 1) + p(x|y = 0)(1 - \alpha)} \quad (7)$$

$$= \frac{1}{1 + \frac{1-\alpha}{\alpha} \frac{p(x|y=0)}{p(x|y=1)}} \quad (8)$$

$$= \frac{1}{1 + \frac{1-\alpha}{\alpha} \prod_{i=1}^D \frac{p(x_i|y=0)}{p(x_i|y=1)}} \quad (9)$$

$$= \frac{1}{1 + \exp\left\{\log \frac{1-\alpha}{\alpha} + \sum_{i=1}^D \left[-\frac{1}{2\sigma_i^2} [(2\mu_{i1} - 2\mu_{i0})x_i + \mu_{i0}^2 - \mu_{i1}^2]\right]\right\}} \quad (10)$$

$$= \frac{1}{1 + \exp(-\sum_{i=1}^D w_i x_i - b)} \quad (11)$$

$$= \sigma(w^T x + b) \quad (12)$$

where

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

$$b = \sum_i \left[ \mu_{i0}^2 - \mu_{i1}^2 - \log \frac{1-\alpha}{\alpha} \right]$$

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

## 1.2 Maximum Likelihood Estimation

$$p(y|w, b) = \prod_{n=1}^N p(y = 1|x^{(n)}, w, b)^{y^{(n)}} [1 - p(y = 1|x^{(n)}, w, b)]^{(1-y^{(n)})} \quad (13)$$

$$E(w, b) = -\log(p(y|w, b)) \quad (14)$$

$$= -\left[\sum_{n=1}^N y^{(n)} \log \frac{1}{1 + \exp(w^T x^{(n)} + b)} + (1 - y^{(n)}) \log \frac{\exp(w^T x^{(n)} + b)}{1 + \exp(w^T x^{(n)} + b)}\right] \quad (15)$$

$$= -\sum_{n=1}^N \{y^{(n)}(w^T x^{(n)} + b) - \log[1 + \exp(w^T x^{(n)} + b)]\} \quad (16)$$

Then derive expressions for the derivatives of E with respect to w and b:

$$\frac{\partial E(w, b)}{\partial w} = -\sum_{n=1}^N [y^{(n)} x^{(n)} - \sigma(w^T x^{(n)} + b) x^{(n)}] \quad (17)$$

$$\frac{\partial E(w, b)}{\partial b} = -\sum_{n=1}^N [y^{(n)} - \sigma(w^T x^{(n)} + b)] \quad (18)$$

## 1.3 L2 Regularization

$$p(w_i) = \mathcal{N}(w_i|0, \frac{1}{\lambda}) = \frac{1}{\sqrt{2\pi\frac{1}{\lambda}}} \exp\{-\frac{\lambda}{2} w_i^2\} \quad (19)$$

$$p(b) = \mathcal{N}(b|0, \frac{1}{\lambda}) = \frac{1}{\sqrt{2\pi\frac{1}{\lambda}}} \exp\{-\frac{\lambda}{2} b^2\} \quad (20)$$

$$p(w, b|\mathcal{D}) = \frac{p(\mathcal{D}|w, b)p(w, b)}{p(\mathcal{D})} \quad (21)$$

$$= \frac{p(\mathcal{D}|w, b) \left(\sqrt{\frac{\lambda}{2\pi}}\right)^{D+1} \exp\{-\frac{\lambda}{2} \sum_{i=1}^D w_i^2\} \exp\{-\frac{\lambda}{2} b^2\}}{p(\mathcal{D})} \quad (22)$$

$$\propto p(\mathcal{D}|w, b) \left(\sqrt{\frac{\lambda}{2\pi}}\right)^{D+1} \exp\{-\frac{\lambda}{2} \sum_{i=1}^D w_i^2\} \exp\{-\frac{\lambda}{2} b^2\} \quad (23)$$

$$= p'(w, b|\mathcal{D}) \quad (24)$$

$$L(w, b) = -\log p'(w, b) \quad (25)$$

$$= E(w, b) + \frac{\lambda}{2} \sum_{i=1}^D w_i^2 + \frac{\lambda}{2} b^2 + C(\lambda) \quad (26)$$

where  $C(\lambda) = -\frac{D+1}{2} \log \frac{\lambda}{2\pi}$ .

The derivatives of  $L$  with respect to  $w$  and  $b$  are:

$$\frac{\partial L(w, b)}{\partial w} = \frac{\partial E(w, b)}{\partial w} + \lambda \sum_{i=1}^D w_i \quad (27)$$

$$\frac{\partial L(w, b)}{\partial b} = \frac{\partial E(w, b)}{\partial b} + \lambda b \quad (28)$$

## 2 Digit Classification

### 2.1 k-Nearest Neighbours

I write a script named 'knn.m' that runs KNN for different values of  $k \in \{1, 3, 5, 7, 9\}$  (the code is in section 4.1) and plot the classification rate on the validation set as a function of  $k$ , the plot is:

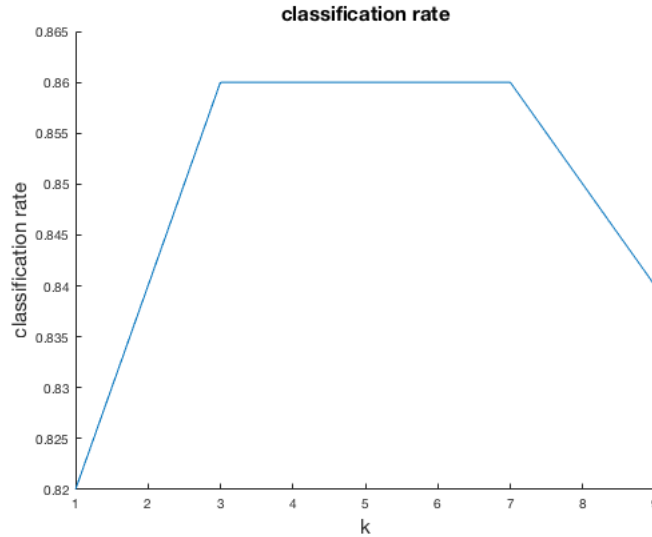


Figure 1: The classification rate for different  $k$

From the plot we can see that the classification rate is from 0.82 to 0.86, which is apparently depends on the values of  $k$ . As  $k$  is too small, the classification is more likely to be impacted by noisy, and as  $k$  is too large, when having many digits belongs to other types, then the classification rate will also be influenced.

Besides, as long as we get a new digits to classify, we have to calculate the L2-distance between the new digit and all training digits, which makes the classification to be much slowly when the data is large. But when dealing with small data or multi-model classification, KNN Classification works better for it's high classification rate and easy to realize.

The most important thing in KNN classification is to choose the value of  $k$ . To do this, I use cross-validation method: equally divide the training data into three parts, for each part,

set it as the new valid data and set the other parts together with the old valid data as the new training data(the code named 'chosenK.m' is in section 4.2).

The plots we got are shown below:

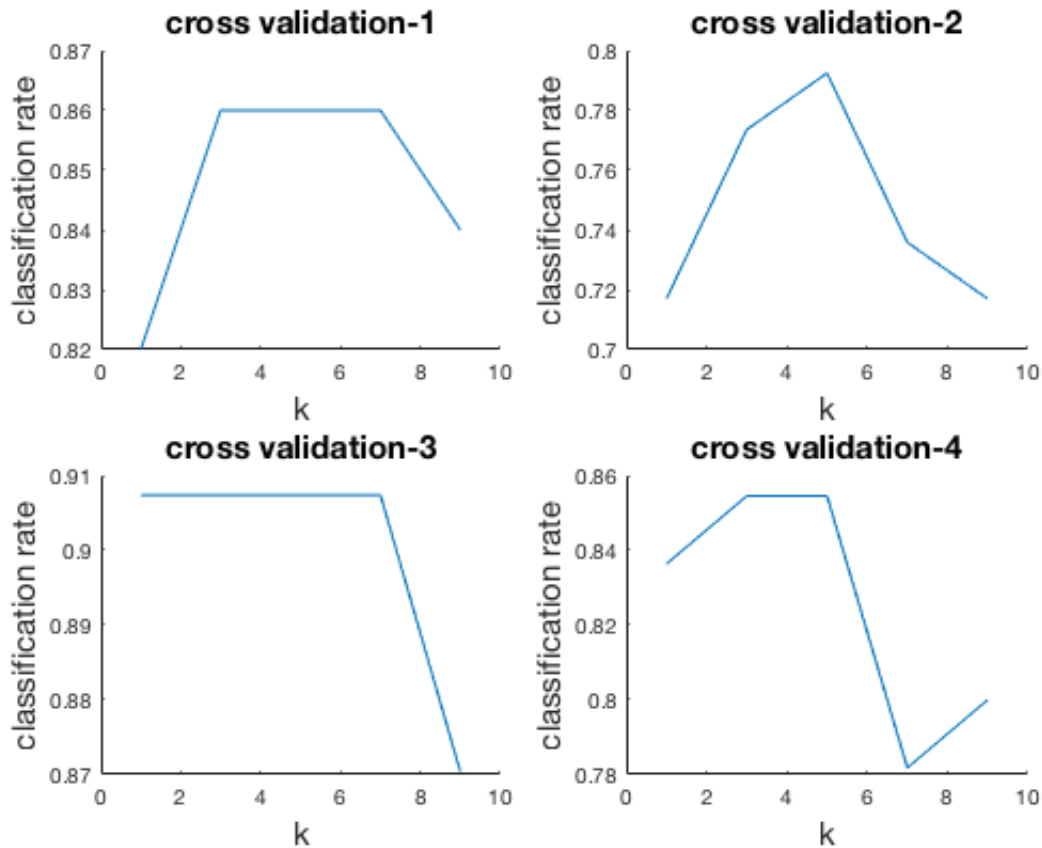


Figure 2: The classification rate against  $k$  in different cross-validation

From the plots we could find out that when  $k=5$ , the KNN Classification works better than other values of  $k$  in all cross validations. So I set  $k^*=5$  as the  $k$  I choose.

The rate for  $k^*+2$  and  $k^*-2$  is:

```

1 Codes in matlab:
2 kstar = 5;
3 krange = [kstar-2, kstar, kstar+2];
4 disp(r(krange));
5
6 The outputs are:
7     0.8600
8     0.8600
9     0.8600

```

Try test data to see the different classification rates of k. The plot is below:

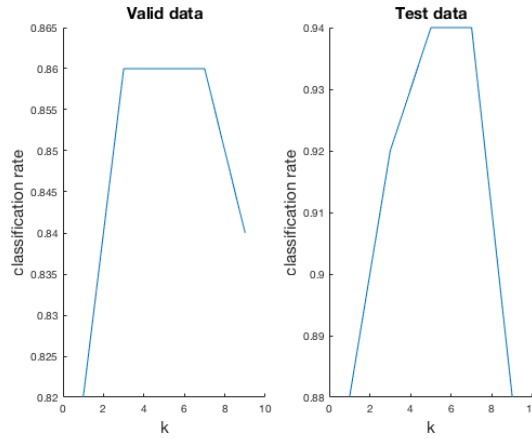


Figure 3: The classification against k for valid and test data

From the plots we could see that, the test performance for these values of k does correspond to the validation performance, which also works better when k=5. The reason for it is that the number of data is what really matters when choosing the k's value. And the valid data and test data are in the same scale and ranged similarly, so they both get the highly classification rate at k=5 and low in k=1 or larger than 8.

## 2.2 Logistic regression

### 2.2.1 Implementation the missing part of logistic

#### a. Function logistic\_predict

Function logistic\_predict is to calculate the estimate probabilities of y giving weights and data. The relationship between them is shown in section 1.1:

$$p(y = 1|x) = \sigma(w^T x + b) = \sigma(weights^T data)$$

where

$$weights^T = (w_1, w_2, \dots, w_n, b) \quad (w^T = (w_1, w_2, \dots, w_n))$$

$$data = (x_1, x_2, \dots, x_n, (1, \dots, 1))^T \quad (x = (x_1, x_2, \dots, x_n)^T)$$

The code named 'logistic\_predict' is in section 4.3

#### b. Function logistic

Function logistic is to calculate negative log likelihood and derivatives with respect to weights. The relationship between negative log likelihood and weights, data is shown in

section 1.2:

$$\begin{aligned}
E(w, b) &= - \sum_{n=1}^N \{y^{(n)}(w^T x^{(n)} + b) - \log[1 + \exp(w^T x^{(n)} + b)]\} \\
&= - \sum_{n=1}^N \{y^{(n)}(\text{weights}^T \text{data}^{(n)}) - \log[1 + \exp(\text{weights}^T \text{data}^{(n)})]\} \\
&= E(\text{weights})
\end{aligned}$$

and

$$\frac{\partial E(\text{weights})}{\partial \text{weights}} = - \sum_{n=1}^N [y^{(n)} \text{data}^{(n)} - \sigma(\text{weights}^T \text{data}^{(n)}) \text{data}^{(n)}]$$

where

$$\begin{aligned}
\text{weights}^T &= (w_1, w_2, \dots, w_n, b) \quad (w^T = (w_1, w_2, \dots, w_n)) \\
\text{data} &= (x_1, x_2, \dots, x_n, (1, \dots, 1))^T \quad (x = (x_1, x_2, \dots, x_n)^T)
\end{aligned}$$

The code of the function named 'logistic' is in section 4.4

After completing function `logistic_predict` and `logistic`, use function `chekgrad` to make sure that the gradients are correct by running section of *logistic\_regression\_template* and the output is much smaller than 1 (which means that the gradients are correct).

### 2.2.2 Hyperparameter Setting

a. learning rate

The hyperparameter `learning_rate` is very important, because if use a small learning rate, the model will take longer to converge and may overfit the data. On the other hand, if use a big learning rate, the model may not fit well with the test data.

Thus I wrote a script named 'chosen\_rate' to find out the best rate (the code named `chosen_rate` is in section 4.5). The plots shown below is the change of final cross entropy and final fraction of valid data classified correctly with different learning rate.

Plots when rate ranges from 0.001 to 0.1 (the step is 0.01):

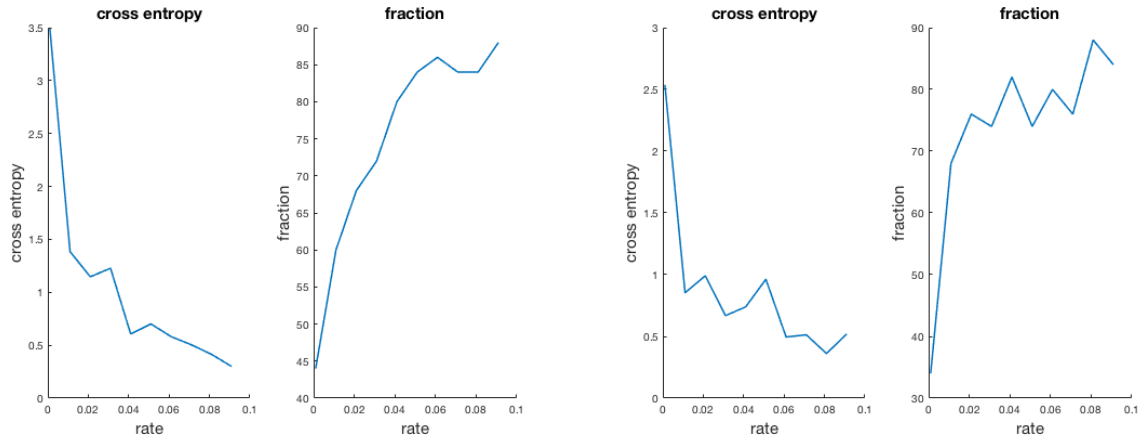


Figure 4: the cross entropy and fraction when rate is from 0.001 to 0.1(run the code for several times and choose two of them).

Plots when rate ranges from 0.1 to 1(the step is 0.01):



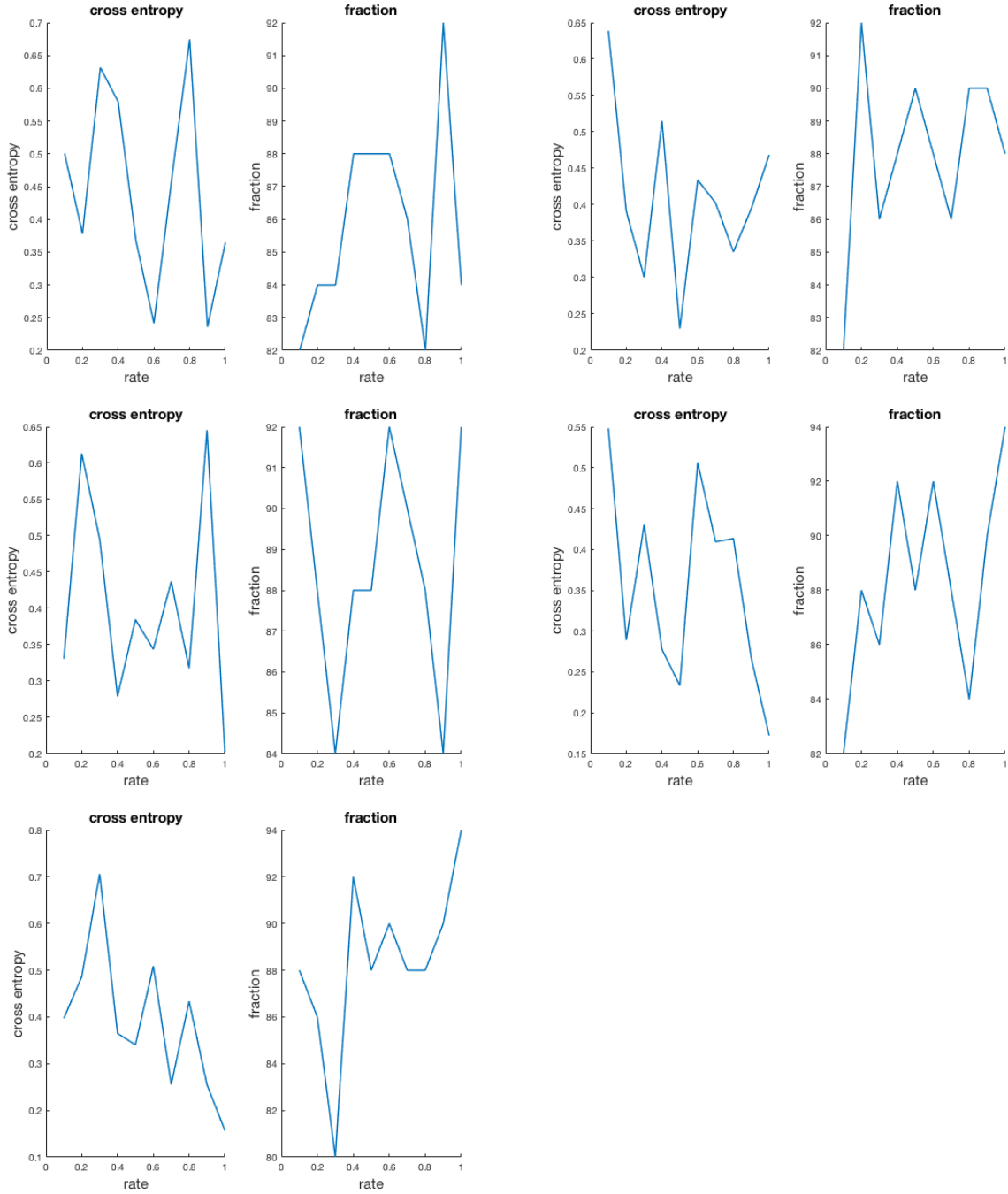


Figure 5: the cross entropy and fraction when rate is from 0.1 to 0.1(run the code for several times and choose five of them).

Plots when rate ranges from 1 to 10(the step is 1):

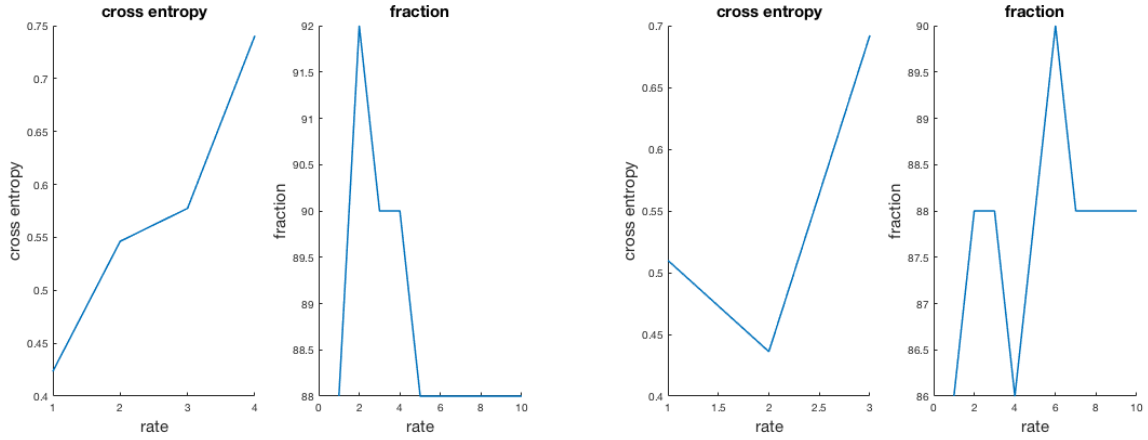


Figure 6: the cross entropy and fraction when rate is from 1 to 10(run the code for several times and choose two of them).

from the plots we could find out that the model trained better with rate ranges from 0.5 to 1.5, so we draw some figures more precisely.

Plots when rate ranges from 0.5 to 1.5(the step is 0.1):

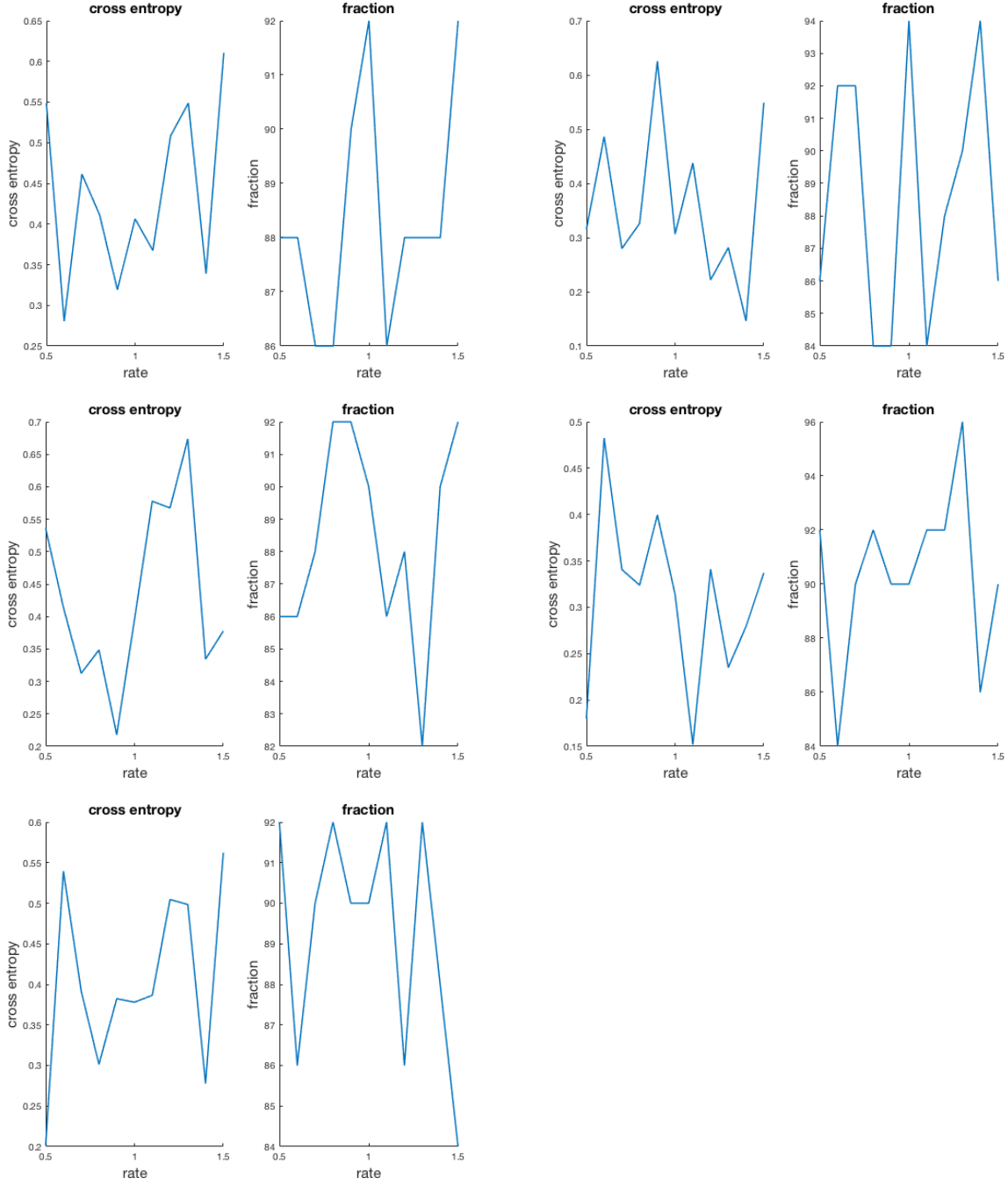


Figure 7: the cross entropy and fraction when rate is from 0.5 to 1.5(run the code for several times and choose five of them).

From those figures, we find the most effective rate and set the hyperparameter.learningrate=1, which means the actual learning rate is  $1/(N = 160) = 0.00625$ .

b. number of iteraions:

The number of iterations is set as 500 so the code won't take so much time to get the result.

c. weights:

The weights we initialize as random numbers satisfied normal distribution (which means  $w \sim N(0, 1)$ )

After setting the hyperparameter, the final cross entropy and classification error on the training, validation and test sets are reported below:

```

1 TRAIN CE: 0.004506      TRAIN FRAC:100.00
2 VALIC CE: 0.399154      VALID FRAC:88.00
3 TEST CE: 0.275091       TEST FRAC:96.00

```

The function 'evaluate' is used to compute cross entartropy and the fraction of inputs classified correctly,

Now look at how the cross entropy changes as training progress, the plots are below:

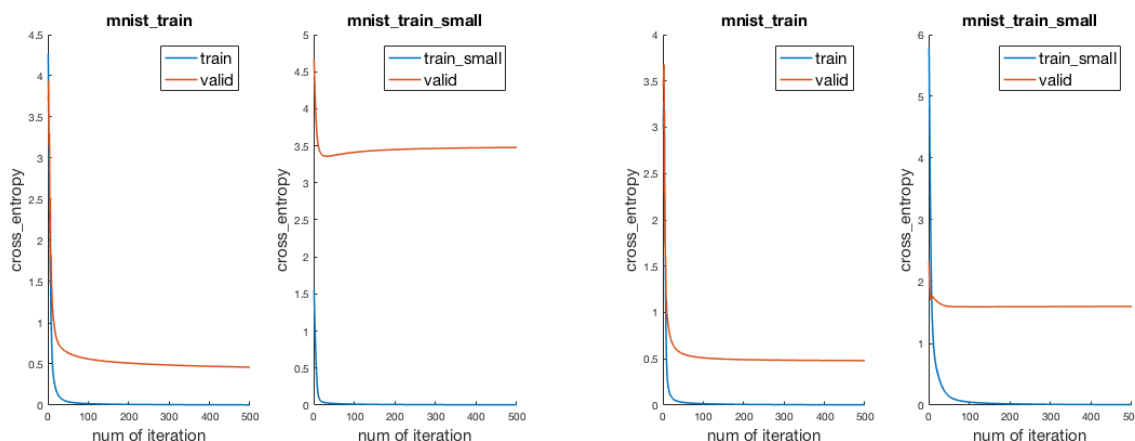


Figure 8: the cross entropy against the num of iteration (run the code for several times and choose two of them).

After running the codes several times, the results change, because the bigger the entratropy got, the more uncertainty the results are, and as the model been training, the entratropy goes down. so as the model been training, the cross entratropy becomes smaller.

## 2.3 Penalized logistic regression

As derived in section 1.3, after including a regularizer, the new likelihood function comes to (omit the  $C(\lambda)$  term in error computation):

$$\begin{aligned} p(w, b|\mathcal{D}) &= E(w, b) + \frac{\lambda}{2} \sum_{i=1}^D w_i^2 + \frac{\lambda}{2} b^2 \\ &= E(w, b) + \frac{\lambda}{2} \sum_{i=1}^D weights_i^2 \\ L(w, b) &= E(w, b) + \lambda \sum_{i=1}^D weights_i \end{aligned}$$

where

$$weights^T = (w_1, w_2, \dots, w_n, b) \quad (w^T = (w_1, w_2, \dots, w_n))$$

The code of the function named "logistic\_pen" is in section 4.6

Now for each value of  $\lambda \in \{0.001, 0.01, 0.1, 1.0\}$ , re-run logistic regression 10 times with the weights randomly initialized each time. Plot the average cross entropy and classification error against  $\lambda$  for both mnist\_train and mnist\_train\_small. The plots are shown below:

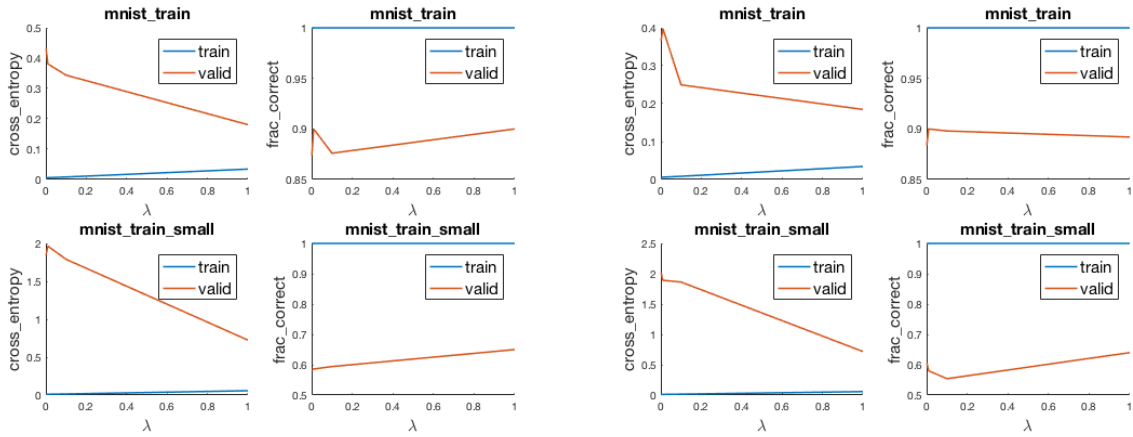


Figure 9: average cross entropy and classification error against  $\lambda$  (run the code for several times and choose two of them).

According to the plots, we observe the entropy and classification error change when  $\lambda$  increased:

the average cross entropy of valid data goes up, and then down

the average cross entropy of train data goes up  
the average classification error of valid data goes up, and then down, and then up  
the average classification error of train data keeps equal 1

The reason why they behave this way is that when  $\lambda$  is bigger, the restrain of weights is tighter thus make the classifier works better

Base on the discussion above, the best value of  $\lambda$  is 1, and the test error for the best value of  $\lambda$  is:

```
1 TEST CE:    0.235382          TEST FRAC:  93.20
2 TEST SMALL CE:  0.843465 TEST SMALL FRAC:  67.20
```

Compare the results with and without penalty, the test error in section 2.3 is:

```
1 TEST CE:    0.206912          TEST FRAC:  92.40
2 TEST SMALL CE:  0.611734 TEST SMALL FRAC:  73.00
```

Through the data, we can find out that the one with penalty performed better for both data set. I think the reason that penalty performed better is that:

We want to minimize  $E(\text{weights})$ , after adding a regularized term, it comes to the minimize of  $L(\text{weights}) = E(\text{weights}) + \frac{\lambda}{2} \sum_{i=1}^D \text{weights}_i^2$ . According to Lagrange multiplication operator, this question turns into a constrained minimum question: to obtain the minimize of  $L(\text{weights})$  subject to  $\sum_{i=1}^D \text{weights}_i \leq \frac{1}{\lambda}$ . So the  $E(\text{weights})$  gets smaller.

(The codes of logistic regression named '*logistic\_regression\_template.m*' is in section 4.7).  
(The codes of penalized regression named '*logistic\_regression\_penalized.m*' is in section 4.8).

## 2.4 Naive Bayes

To complete the pipeline of training, testing a native Bayes classifier and visualize learned models, I filled in the script '*run\_nb.m*' (The code named '*run\_nb.m*' is in section 4.9)

Thus the training and test accuracy using the naive Bayes model is :

```
1 training accuracy: 86.250000
2 test accuracy: 80.000000
```

And the visualization of the mean and variance vectors is :



Figure 10: the visualization of the mean and variance vectors

From the visualization results we could see that the mean and variance vectors are also belongs to two types but with low visualizations.

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

From the results we got from the above three training methods, we could compare these three models:

### a. KNN Classifier:

The easiest one to realize among the three, works better when it comes to small data or multi-model classification. But if the data is large, KNN Classifier needs too much time which makes it inefficiency. Besides, when the training data is not balanced(with one sample big and others small), the model may classified wrong which leads to a lower classification accuracy.

### b. Logistic Regression Classifier:

Logistic regression classifier works better with large dataset, but it may need a bit more time then the other two models.

### c. Naive Bayes Classifier:

Also easy to realize but the test accuracy is not high compared to the other two classifiers.

## 3 Stochastic Sub Gradient Methods

### 3.1 Averaging of $w^t$

First, in stochastic subgradient method, the average of  $w^t$  variables is convergence to final result of  $w$ . We can obtain convergence rates with decreasing steps(as mentioned in Sec.3.2.3):

if  $\alpha_t = \frac{1}{\mu_t}$ , we can show the convergence rate as  $O(\frac{1}{t})$ . Thus we make a new function named svmAvg that reports the performance based on running average of the  $w^t$  values rather than the current value.(The modified part of the cod named 'svmAvg' is in section 4.10)

The plot of the performance with averaging is shown below:

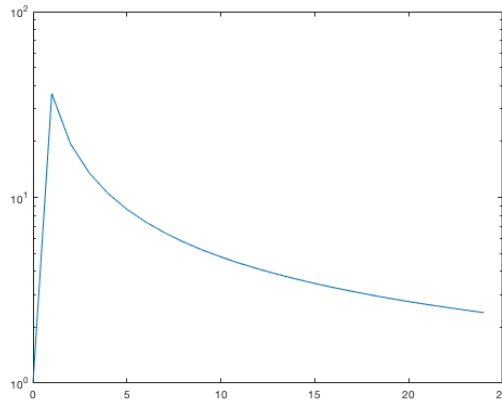


Figure 11: The performance with averaging

### 3.2 Second-Half Averaging

In this section, we make a little change to start averaging once we have get half way to maxIter instead of averaging at the beginning of the iteration.(The modied parts of function named'svmAvg1' is in section 4.11)

The plot of the performance "second-half" averaging is shown below:



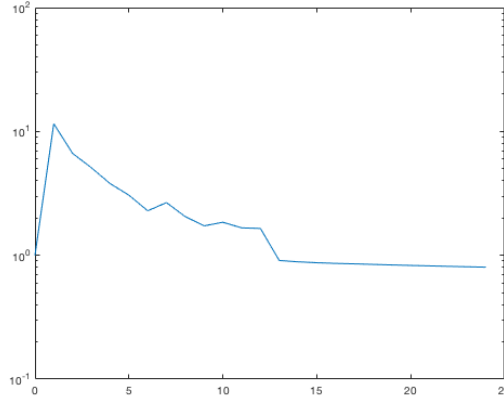


Figure 12: the performance of "second-half" averaging

### 3.3 Modify of SVM

Stochastic subgradient descent:

$w^{t+1} = w^t - \alpha_t g_{it} - \alpha_t \lambda w^t$ , where

$$|g_{it}| = \begin{cases} -y_i x_i & \text{if } 1 - y_i(w^T x_i) > 0, \\ 0 & \text{otherwise,} \end{cases}$$

the learning rate is low and the iteration cost is high because it could only change one item in one iteration, so we do a little change to make the model more effective by the following steps:

- keep in memory the gradients of all functions  $f_i$ ,  $i=1, \dots, n$
- Random selection  $i(t) \in 1, \dots, n$  with replacement
- For each iteration:  $w^{t+1} = w^t - \alpha_t \frac{\sum_{i=1}^n g_{it}}{n} - \alpha_t \lambda w^t$ ,  
where

$$|g_{it}| = \begin{cases} f'_i(w^t) & \text{if } i = i(t), \\ g_{i(t-1)} & \text{otherwise,} \end{cases}$$

and

$$f'_i(w^t) = \begin{cases} -y_i x_i & \text{if } 1 - y_i(w^T x_i) > 0, \\ 0 & \text{otherwise,} \end{cases}$$

The code of the new svm function named 'svm\_new' is in section 4.12 and the plot of the performance with the modifications is shown below:

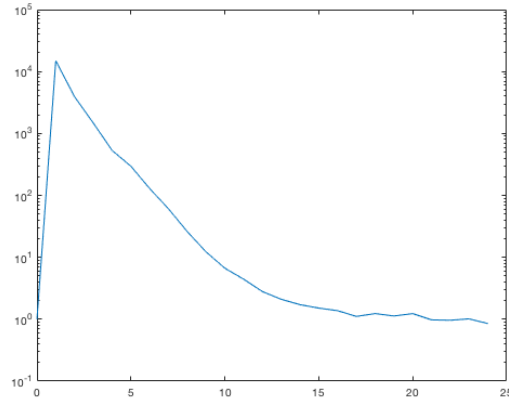


Figure 13: The performance of modified svm

## 4 Codes for report

### 4.1 knn.m

```

1 clear all;
2 close all;
3
4 load mnist_train;
5 load mnist_valid;
6
7 [n, m] = size(valid_inputs);
8
9 r = zeros(n,1);
10
11 for k = 1:2:9
12     valid_predict = run_knn(k, train_inputs, train_targets, valid_inputs);
13     r(k) = sum([valid_predict==valid_targets]) / n;
14 end
15
16 figure;
17 hold on;
18 title('Valid data', 'FontSize', 15);
19 plot(1:2:9, r(1:2:9));
20 xlabel('k', 'FontSize', 15);
21 ylabel('classification rate', 'FontSize', 15);

```

### 4.2 chosenK.m

```

1 % find the best k in knn
2 clear all;

```

```

3 close all;
4
5 load mnist_train;
6 load mnist_valid;
7
8 [n, m] = size(valid_inputs);
9 [n_train, m_train] = size(train_inputs);
10 t = floor(n_train/3);
11 step = [1, t, 2*t, n_train];
12 r = zeros(4,9);
13
14 for k = 1:2:9
15     valid_predict = run_knn(k, train_inputs, train_targets, valid_inputs);
16     r(1,k) = sum([valid_predict==valid_targets]) / n;
17 end
18
19 for t = 2:4
20     t1_inputs = [valid_inputs; train_inputs(1:(step(t-1)-1), : )];
21     train_inputs((step(t)+1):n_train, : );
22     t1_targets = [valid_targets; train_targets(1:(step(t-1)-1), : )];
23     train_targets((step(t)+1):n_train, : );
24     t2_inputs = train_inputs(step(t-1):step(t), : );
25     t2_targets = train_targets(step(t-1):step(t), : );
26     for k = 1:2:9
27         t_predict = []
28         t_predict = run_knn(k, t1_inputs, t1_targets, t2_inputs);
29         r(t,k) = sum([t_predict == t2_targets]) / (step(t)-step(t-1)+1);
30     end
31 end
32
33 %plot some features
34 figure;
35 subplot(2,2,1);
36 hold on;
37 title('cross validation-1', 'FontSize', 15);
38 plot(1:2:9,r(1, 1:2:9));
39 xlabel('k', 'FontSize', 15);
40 ylabel('classification rate', 'FontSize', 15);
41
42 subplot(2,2,2);
43 hold on;
44 title('cross validation-2', 'FontSize', 15);
45 plot(1:2:9,r(2, 1:2:9));
46 xlabel('k', 'FontSize', 15);
47 ylabel('classification rate', 'FontSize', 15);

```

```

48
49 subplot(2,2,3);
50 hold on;
51 title('cross validation-3', 'FontSize', 15);
52 plot(1:2:9,r(3, 1:2:9));
53 xlabel('k', 'FontSize', 15);
54 ylabel('classification rate', 'FontSize', 15);
55
56 subplot(2,2,4);
57 hold on;
58 title('cross validation-4', 'FontSize', 15);
59 plot(1:2:9,r(4, 1:2:9));
60 xlabel('k', 'FontSize', 15);
61 ylabel('classification rate', 'FontSize', 15);

```

### 4.3 logistic\_predict.m

```

1 function [y] = logistic_predict(weights, data)
2 %   Compute the probabilities predicted by the logistic classifier.
3 %
4 %   Note: N is the number of examples and
5 %         M is the number of features per example.
6 %
7 %   Inputs:
8 %       weights: (M+1) x 1 vector of weights, where the last element
9 %               corresponds to the bias (intercepts).
10 %       data: N x M data matrix where each row corresponds
11 %            to one data point.
12 %   Outputs:
13 %       y: :N x 1 vector of probabilities. This is the output of
14 %          the classifier.
15
16 [n,m] = size(data);
17 data = [data, ones(n,1)];
18 y = sigmoid(data * weights);
19 end

```

### 4.4 logistic.m

```

1 function [f, df, y] = logistic(weights, data, targets, hyperparameters)
2 % Calculate log likelihood and derivatives with respect to weights.
3 %
4 % Note: N is the number of examples and
5 %       M is the number of features per example.
6 %
7 % Inputs:

```

```

8 %      weights:      (M+1) x 1 vector of weights , where the last element
9 %                  corresponds to bias (intercepts).
10 %      data:        N x M data matrix where each row corresponds
11 %                  to one data point.
12 %      targets:     N x 1 vector of binary targets. Values should be either
13 %0 or 1.
14 %      hyperparameters: The hyperparameter structure
15 %
16 % Outputs:
17 %      f:           The scalar error value?i.e. negative log likelihood).
18 %      df:          (M+1) x 1 vector of derivatives of error w.r.t. weights.
19 %      y:           N x 1 vector of probabilities. This is the output of the
20 %                  classifier.
21
22 [n, m] = size(data);
23 x = [data , ones(n,1)];
24 w = x*weights;
25 f = -sum(x * weights .* targets)+ sum(log(1+ exp(x * weights)));
26 df = -x' * (targets-sigmoid(x * weights));
27 y = logistic_predict(weights , data);
28 end

```

## 4.5 chosen\_rate

```

1 %% Clear workspace.
2 clear all;
3 close all;
4
5 %% Load data.
6 load mnist_train;
7 load mnist_valid;
8 load mnist_test;
9
10 r0=0;
11 ce = [];
12 frac=[];
13 N = size(train_inputs ,1);
14 tic;
15 for r = 0.5:0.1:1.5
16     r0 = r0+1;
17     % Initialize hyperparameters.
18     % Learning rate
19     hyperparameters.learning_rate = r;
20     hyperparameters.weight_regularization = 0;
21     hyperparameters.num_iterations = 300;

```

```

22 weights = randn((size(train_inputs,2)+1),1);
23 weights_small = weights;
24
25 cross_entropy_train = zeros( hyperparameters.num_iterations , 1 );
26 cross_entropy_train_small = cross_entropy_train;
27 cross_entropy_valid = cross_entropy_train;
28 cross_entropy_valid_small = cross_entropy_train;
29
30 %% Begin learning with gradient descent.
31 for t = 1:hyperparameters.num_iterations
32     % Find the negative log likelihood and derivative w.r.t. weights.
33     [f, df, predictions] = logistic(weights, ...
34         train_inputs , ...
35         train_targets , ...
36         hyperparameters);
37
38     [cross_entropy_train(t), frac_correct_train] =
39     evaluate(train_targets , predictions);
40
41     if isnan(f) || isinf(f)
42         error('nan/inf error');
43     end
44
45     %% Update parameters.
46     weights = weights - hyperparameters.learning_rate .* df/ N;
47     predictions_valid = logistic_predict(weights, valid_inputs);
48     [cross_entropy_valid(t), frac_correct_valid] =
49     evaluate(valid_targets , predictions_valid);
50
51     predictions_test = logistic_predict(weights, test_inputs);
52     [cross_entropy_test, frac_correct_test] = evaluate(test_targets ,
53     predictions_test);
54
55     %% Print some stats.
56     fprintf(1, 'ITERATION:%4i    NLOGL:%4.2f\n TRAIN CE %.6f
57     TRAIN FRAC:%2.2f\t VALIC_CE %.6f VALID FRAC:%2.2f\t ' ,...
58         t, f/N, cross_entropy_train(t), frac_correct_train*100,
59         cross_entropy_valid(t), frac_correct_valid*100);
60     fprintf(1, 'TEST CE %.6f TEST FRAC:%2.2f
61     \n ',cross_entropy_test , frac_correct_test*100);
62
63 end
64 ce(r0)= cross_entropy_valid(t);
65 frac(r0) = frac_correct_valid*100;
66 end

```

```

67 toc;
68 % draw some features
69 x = 0.5:0.1:1.5;
70 figure;
71 subplot(1,2,1);
72 hold on;
73 title('cross entropy', 'FontSize', 15);
74 plot(x, ce, 'LineWidth', 1.5);
75 xlabel('rate', 'FontSize', 15);
76 ylabel('cross entropy', 'FontSize', 15);
77 subplot(1,2,2);
78 hold on;
79 plot(x, frac, 'LineWidth', 1.5);
80 title('fraction', 'FontSize', 15);
81 xlabel('rate', 'FontSize', 15);
82 ylabel('fraction', 'FontSize', 15);

```

## 4.6 logistic\_pen.m

```

1 function [f, df, y] = logistic_pen(weights, data, targets,
2 hyperparameters)
3 % Calculate log likelihood and derivatives with respect to weights.
4 %
5 % Note: N is the number of examples and
6 %       M is the number of features per example.
7 %
8 % Inputs:
9 %       weights: (M+1) x 1 vector of weights, where the last element
10 %               corresponds to bias (intercepts).
11 %       data: N x M data matrix where each row corresponds
12 %             to one data point.
13 %       targets: N x 1 vector of targets class probabilities.
14 %       hyperparameters: The hyperparameter structure
15 %
16 % Outputs:
17 %       f: The scalar error value.
18 %       df: (M+1) x 1 vector of derivatives of error w.r.t. weights.
19 %       y: N x 1 vector of probabilities. This is the output of the
20 %          classifier.
21 %
22
23 %TODO: finish this function
24
25 [n,m] = size(data);
26 y = sigmoid([data, ones(n,1)] * weights);

```

```

27
28 [f1, df1, y1] = logistic(weights, data, targets, hyperparameters);
29 weights(length(weights))=0;
30 h = hyperparameters.weight_regularization;
31 f = f1 + h/2 * sum(weights.^2);
32 df = df1 + h * weights;
33
34 end

```

## 4.7 logistic\_regression\_template.m

```

1  %% Clear workspace.
2  clear all;
3  close all;
4
5  %% Load data.
6  load mnist_train;
7  load mnist_valid;
8  load mnist_train_small;
9  load mnist_test;
10
11 tic;
12 % Learning rate
13 hyperparameters.learning_rate = 0.00625; %1 %...
14 % Weight regularization parameter
15 hyperparameters.weight_regularization = 0; %...
16 % Number of iterations
17 hyperparameters.num_iterations = 500; %...
18 % Logistics regression weights
19 weights = randn((size(train_inputs,2)+1),1);
20 weights_small = weights;
21
22 cross_entropy_train = zeros( hyperparameters.num_iterations, 1 );
23 cross_entropy_train_small = cross_entropy_train;
24 cross_entropy_valid = cross_entropy_train;
25 cross_entropy_valid_small = cross_entropy_train;
26
27 %% Verify that your logistic function produces the right gradient, diff
28 should be very close to 0
29
30 % this creates small random data with 20 examples and 10
31 dimensions and checks the gradient on
32 % that data.
33 nexamples = 20;
34 ndimensions = 10;

```



```

35 diff = checkgrad('logistic', ...
36                 randn((ndimensions + 1), 1), ... % weights
37                 0.001,... % perturbation
38                 randn(nexamples, ndimensions), ... % data
39                 rand(nexamples, 1), ... % targets
40                 hyperparameters) % other hyperparameters
41
42 N = size(train_inputs,1);
43 N_small = size(train_inputs_small,1);
44
45 %% Begin learning with gradient descent.
46 for t = 1:hyperparameters.num_iterations
47
48     % Find the negative log likelihood and derivative w.r.t. weights.
49     [f, df, predictions] = logistic(weights, ...
50                                     train_inputs, ...
51                                     train_targets, ...
52                                     hyperparameters);
53
54     [f_small, df_small, predictions_small] = logistic(weights_small, ...
55                                                         train_inputs_small, ...
56                                                         train_targets_small, ...
57                                                         hyperparameters);
58
59     [cross_entropy_train(t), frac_correct_train] = evaluate(train_targets,
60 predictions);
61 [cross_entropy_train_small(t), frac_correct_train_small] =
62 evaluate(train_targets_small, predictions_small);
63
64 if isnan(f) || isinf(f)
65     error('nan/inf error');
66 end
67
68
69 if isnan(f_small) || isinf(f_small)
70     error('nan/inf error');
71 end
72
73 %% Update parameters.
74 weights = weights - hyperparameters.learning_rate .* df;
75 predictions_valid = logistic_predict(weights, valid_inputs);
76 [cross_entropy_valid(t), frac_correct_valid] =
77 evaluate(valid_targets, predictions_valid);
78
79 weights_small = weights_small - hyperparameters.learning_rate .*

```

```

80     df_small;
81     predictions_valid_small = logistic_predict(weights_small,
82     valid_inputs);
83     [cross_entropy_valid_small(t), frac_correct_valid_small] =
84     evaluate(valid_targets, predictions_valid_small);
85
86     predictions_test = logistic_predict(weights, test_inputs);
87     [cross_entropy_test, frac_correct_test] = evaluate(test_targets,
88     predictions_test);
89
90     %% Print some stats.
91     fprintf(1, 'ITERATION:%4i    NLOGL:%4.2f\n TRAIN CE %.6f
92     TRAIN FRAC:%2.2f\t VALIC.CE %.6f VALID FRAC:%2.2f\t ',...
93     t, f/N, cross_entropy_train(t), frac_correct_train*100,
94     cross_entropy_valid(t), frac_correct_valid*100);
95     fprintf(1, 'TEST CE %.6f TEST FRAC:%2.2f\n', cross_entropy_test,
96     frac_correct_test*100);
97
98 end
99 toc;
100
101 figure;
102 subplot(1,2,1);
103 hold on;
104 title('mnist\_train', 'FontSize', 15);
105 plot(1:hyperparameters.num_iterations, cross_entropy_train,
106 'LineWidth', 1.5);
107 plot(1:hyperparameters.num_iterations, cross_entropy_valid,
108 'LineWidth', 1.5);
109 l = legend('train', 'valid');
110 set(l, 'FontSize', 15);
111 xlabel('num of iteration', 'FontSize', 15);
112 ylabel('cross\_entropy', 'FontSize', 15);
113
114 subplot(1,2,2);
115 hold on;
116 title('mnist\_train\_small', 'FontSize', 15);
117 plot(1:hyperparameters.num_iterations, cross_entropy_train_small,
118 'LineWidth', 1.5);
119 plot(1:hyperparameters.num_iterations, cross_entropy_valid_small,
120 'LineWidth', 1.5);
121 l = legend('train\_small', 'valid');
122 set(l, 'FontSize', 15);
123 xlabel('num of iteration', 'FontSize', 15);
124 ylabel('cross\_entropy', 'FontSize', 15);

```

## 4.8 `logistic_regression_penalized.m`

```
1  %% Clear workspace.
2  clear all;
3  close all;
4
5  %% Load data.
6  load mnist_train;
7  load mnist_valid;
8  load mnist_train_small;
9
10 lambda = [0.001, 0.01, 0.1, 1];
11 cross_entropy_train_lambda = zeros(4, 1);
12 cross_entropy_valid_lambda = zeros(4, 1);
13 frac_correct_train_lambda = zeros(4, 1);
14 frac_correct_valid_lambda = zeros(4, 1);
15
16 cross_entropy_train_lambda_small = zeros(4, 1);
17 cross_entropy_valid_lambda_small = zeros(4, 1);
18 frac_correct_train_lambda_small = zeros(4, 1);
19 frac_correct_valid_lambda_small = zeros(4, 1);
20
21 tic;
22 for la = 1:4
23     %% TODO: Initialize hyperparameters.
24     % Learning rate
25     hyperparameters.learning_rate = 0.00625;    %...
26     % Weight regularization parameter
27     hyperparameters.weight_regularization = lambda(la);    %...
28     % Number of iterations
29     hyperparameters.num_iterations = 300;    %...
30     % Logistics regression weights
31     % TODO: Set random weights.
32
33     % cross_entropy_train = zeros( hyperparameters.num_iterations,
34     10 );
35     % cross_entropy_valid = cross_entropy_train;
36     % frac_correct_train = cross_entropy_train;
37     % frac_correct_valid = cross_entropy_train;
38
39     % cross_entropy_train_small =
40     zeros( hyperparameters.num_iterations, 10 );
41     % cross_entropy_valid_small = cross_entropy_train_small;
42     % frac_correct_train_small = cross_entropy_train_small;
43     % frac_correct_valid_small = cross_entropy_valid_small;
```

```

44
45     for k = 1:10
46         weights = randn((size(train_inputs,2)+1),1);
47         weights_small = weights;
48         N = size(train_inputs,1);
49         N_small = size(train_inputs_small,1);
50
51         for t = 1:hyperparameters.num_iterations
52
53             % Find the negative log likelihood and derivative w.r.t.
54             weights.
55             [f, df, predictions] = logistic_pen(weights, ...
56                                                 train_inputs, ...
57                                                 train_targets, ...
58                                                 hyperparameters);
59             [f_small, df_small, predictions_small] =
60             logistic_pen(weights_small, ...
61                         train_inputs_small, ...
62                         train_targets_small, ...
63                         hyperparameters);
64
65             [cross_entropy_train, frac_correct_train] =
66             evaluate(train_targets, predictions);
67             [cross_entropy_train_small, frac_correct_train_small] =
68             evaluate(train_targets_small, predictions_small);
69             %cross_entropy_train
70             ????
71             %% Find the fraction of correctly classified validation
72             examples.
73             % [temp, temp2, frac_correct_valid] = logistic(weights, ...
74             % valid_inputs, ...
75             % valid_targets, ...
76             % hyperparameters);
77
78             % [temp, temp2, frac_correct_valid_small] =
79             logistic(weights, ...
80             % valid_inputs, ...
81             % valid_targets, ...
82             % hyperparameters);
83
84             if isnan(f) || isinf(f)
85                 error('nan/inf error');
86             end
87             if isnan(f_small) || isinf(f_small)
88                 error('nan/inf error');
89             end

```

```

89
90     %% Update parameters.
91     weights = weights - hyperparameters.learning_rate .* df; %/N;
92     predictions_valid = logistic_predict(weights, valid_inputs);
93     [cross_entropy_valid, frac_correct_valid] =
94     evaluate(valid_targets, predictions_valid);
95
96     weights_small = weights_small -
97     hyperparameters.learning_rate .* df_small; %/N_small;
98     predictions_valid_small = logistic_predict(weights_small,
99     valid_inputs);
100    [cross_entropy_valid_small, frac_correct_valid_small] =
101    evaluate(valid_targets, predictions_valid_small);
102
103    %% Print some stats.
104    fprintf(1, 'ITERATION:%4i    NLOGL:%4.2f TRAIN CE %.6f
105    TRAIN FRAC:%2.2f VALIC_CE %.6f VALID FRAC:%2.2f\n', ...
106            t, f/N, cross_entropy_train, frac_correct_train*100,
107            cross_entropy_valid, frac_correct_valid*100);
108
109    end
110    cross_entropy_train_lambda(la) =
111    cross_entropy_train_lambda(la) + cross_entropy_train;
112    cross_entropy_valid_lambda(la) =
113    cross_entropy_valid_lambda(la) + cross_entropy_valid;
114    frac_correct_train_lambda(la) = frac_correct_train_lambda(la) +
115    frac_correct_train;
116    frac_correct_valid_lambda(la) = frac_correct_valid_lambda(la) +
117    frac_correct_valid;
118
119    cross_entropy_train_lambda_small(la) =
120    cross_entropy_train_lambda_small(la) +
121    cross_entropy_train_small;
122    cross_entropy_valid_lambda_small(la) =
123    cross_entropy_valid_lambda_small(la) +
124    cross_entropy_valid_small;
125    frac_correct_train_lambda_small(la) =
126    frac_correct_train_lambda_small(la) + frac_correct_train_small;
127    frac_correct_valid_lambda_small(la) =
128    frac_correct_valid_lambda_small(la) + frac_correct_valid_small;
129
130    end
131    cross_entropy_train_lambda(la) = cross_entropy_train_lambda(la)/
132    10;
133    cross_entropy_valid_lambda(la) = cross_entropy_valid_lambda(la)/

```

```

134     10;
135     frac_correct_train_lambda(la) = frac_correct_train_lambda(la)/10;
136     frac_correct_valid_lambda(la) = frac_correct_valid_lambda(la)/10;
137
138     cross_entropy_train_lambda_small(la) =
139     cross_entropy_train_lambda_small(la)/10;
140     cross_entropy_valid_lambda_small(la) =
141     cross_entropy_valid_lambda_small(la)/10;
142     frac_correct_train_lambda_small(la) =
143     frac_correct_train_lambda_small(la)/10;
144     frac_correct_valid_lambda_small(la) =
145     frac_correct_valid_lambda_small(la)/10;
146 end
147
148 toc;
149
150 figure;
151 subplot(2,2,1);
152 hold on;
153 title('mnist\_train', 'FontSize', 15);
154 plot(lambda, cross_entropy_train_lambda, 'LineWidth', 1.5);
155 plot(lambda, cross_entropy_valid_lambda, 'LineWidth', 1.5);
156 l = legend('train', 'valid');
157 set(l, 'FontSize', 15);
158 xlabel('\lambda', 'FontSize', 15);
159 ylabel('cross\_entropy', 'FontSize', 15);
160
161 subplot(2,2,2);
162 hold on;
163 title('mnist\_train', 'FontSize', 15);
164 plot(lambda, frac_correct_train_lambda, 'LineWidth', 1.5);
165 plot(lambda, frac_correct_valid_lambda, 'LineWidth', 1.5);
166 l = legend('train', 'valid');
167 set(l, 'FontSize', 15);
168 xlabel('\lambda', 'FontSize', 15);
169 ylabel('frac\_correct', 'FontSize', 15);
170
171
172 subplot(2,2,3);
173 hold on;
174 title('mnist\_train\_small', 'FontSize', 15);
175 plot(lambda, cross_entropy_train_lambda_small, 'LineWidth', 1.5);
176 plot(lambda, cross_entropy_valid_lambda_small, 'LineWidth', 1.5);
177 l = legend('train', 'valid');
178 set(l, 'FontSize', 15);

```

```

179 xlabel( '\lambda', 'FontSize', 15);
180 ylabel( 'cross\_entropy', 'FontSize', 15);
181
182
183 subplot(2,2,4);
184 hold on;
185 title( 'mnist\_train\_small', 'FontSize', 15);
186 plot(lambda, frac_correct_train_lambda_small, 'LineWidth', 1.5);
187 plot(lambda, frac_correct_valid_lambda_small, 'LineWidth', 1.5);
188 l = legend( 'train', 'valid');
189 set(l, 'FontSize', 15);
190 xlabel( '\lambda', 'FontSize', 15);
191 ylabel( 'frac\_correct', 'FontSize', 15);

```

## 4.9 run\_nb.m

```

1 % Learn a Naive Bayes classifier on the digit dataset, evaluate its
2 % performance on training and test sets, then visualize the mean and
3 variance
4 % for each class.
5
6 load mnist_train;
7 load mnist_test;
8
9 [log_prior, class_mean, class_var] = train_nb(train_inputs,
10 train_targets);
11 [Trainprediction, TrainAccuracy] = test_nb(train_inputs, train_targets,
12 log_prior, class_mean, class_var);
13 [Testprediction, TestAccuracy] = test_nb(test_inputs, test_targets,
14 log_prior, class_mean, class_var);
15
16 fprintf( 'training accuracy: %f\n', TrainAccuracy*100);
17 fprintf( 'test accuracy: %f\n', TestAccuracy*100);
18 plot_digits(class_mean);
19 plot_digits(class_var);

```

## 4.10 svmAvg.m

```

1 function [model] = svmAvg1(X,y,lambda,maxIter)
2
3 % Add bias variable
4 [n,d] = size(X);
5 X = [ones(n,1) X];
6
7 % Matlab indexes by columns,
8 % so if we are accessing rows it will be faster to use the traspose

```

```

9  Xt = X';
10
11  % Initial values of regression parameters
12  w = zeros(d+1,1);
13  w_mean = w;
14
15  % Apply stochastic gradient method
16  for t = 1:maxIter
17      if mod(t-1,n) == 0
18          % Plot our progress
19          % (turn this off for speed)
20
21          objValues(1+(t-1)/n) = (1/n)*sum(max(0,1-y.*(X*w_mean))) +
22          (lambda/2)*(w_mean'*w_mean);
23          semilogy([0:t/n],objValues);
24          pause(.1);
25      end
26
27      % Pick a random training example
28      i = ceil(rand*n);
29
30      % Compute sub-gradient
31      [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i);
32
33      % Set step size
34      alpha = 1/(lambda*t);
35
36      % Take stochastic subgradient step
37      w = w - alpha*(sg + lambda*w);
38      if t > 1
39          w_mean = (w_mean*(t-1)+w)/t;
40      else
41          w_mean = w;
42      end
43  end
44
45  model.w = w;
46  model.predict = @predict;
47
48  end
49
50  function [yhat] = predict(model,Xhat)
51  [t,d] = size(Xhat);
52  Xhat = [ones(t,1) Xhat];
53  w = model.w;

```



```

54 yhat = sign(Xhat*w);
55 end
56
57 function [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i)
58
59 [d,n] = size(Xt);
60
61 % Function value
62 wtx = w'*Xt(:,i);
63 loss = max(0,1-y(i)*wtx);
64 f = loss;
65
66 % Subgradient
67 if loss > 0
68     sg = -y(i)*Xt(:,i);
69 else
70     sg = sparse(d,1);
71 end
72 end

```

#### 4.11 svmAvg1.m

```

1 function [model] = svmAvg2(X,y,lambda,maxIter)
2
3 % Add bias variable
4 [n,d] = size(X);
5 X = [ones(n,1) X];
6
7 % Matlab indexes by columns,
8 % so if we are accessing rows it will be faster to use the traspose
9 Xt = X';
10
11 % Initial values of regression parameters
12 w = zeros(d+1,1);
13 w_mean = w;
14 half = ceil(maxIter/2);
15 % Apply stochastic gradient method
16 for t = 1:maxIter
17     if mod(t-1,n) == 0
18         % Plot our progress
19         % (turn this off for speed)
20
21         objValues(1+(t-1)/n) = (1/n)*sum(max(0,1-y.*(X*w_mean))) +
22         (lambda/2)*(w_mean'*w_mean);
23         semilogy([0:t/n],objValues);

```

```

24         pause(.1);
25     end
26
27     % Pick a random training example
28     i = ceil(rand*n);
29
30     % Compute sub-gradient
31     [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i);
32
33     % Set step size
34     alpha = 1/(lambda*t);
35
36     % Take stochastic subgradient step
37     w = w - alpha*(sg + lambda*w);
38     if t > half
39         w_mean = (w_mean*(t-half)+w)/(t-half+1);
40     else
41         w_mean = w;
42     end
43 end
44
45 model.w = w;
46 model.predict = @predict;
47
48 end
49
50 function [yhat] = predict(model,Xhat)
51 [t,d] = size(Xhat);
52 Xhat = [ones(t,1) Xhat];
53 w = model.w;
54 yhat = sign(Xhat*w);
55 end
56
57 function [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i)
58
59 [d,n] = size(Xt);
60
61 % Function value
62 wtx = w'*Xt(:,i);
63 loss = max(0,1-y(i)*wtx);
64 f = loss;
65
66 % Subgradient
67 if loss > 0
68     sg = -y(i)*Xt(:,i);

```

```

69 else
70     sg = sparse(d,1);
71 end
72 end

```

#### 4.12 svm\_new.m

```

1 function [model] = svmAvg(X,y,lambda,maxIter)
2
3 % Add bias variable
4 [n,d] = size(X);
5 X = [ones(n,1) X];
6
7 % Matlab indexes by columns,
8 % so if we are accessing rows it will be faster to use the traspose
9 Xt = X';
10
11 % Initial values of regression parameters
12 w = zeros(d+1, 1);
13 sg1 = w;
14 for i = 1 : (d+1)
15     sg(i, :) = Xt(i, :) .* y(i);
16 end
17
18 % Apply stochastic gradient method
19 for t = 1:maxIter
20     if mod(t-1,n) == 0
21         % Plot our progress
22         % (turn this off for speed)
23         objValues(1+(t-1)/n) = (1/n)*sum(max(0,1-y.*(X*w))) + (lambda/
24         2)*(w'*w);
25         semilogy([0:t/n],objValues);
26         pause(.1);
27     end
28
29     % Pick a random training example
30     i = ceil(rand*n);
31
32     % Compute sub-gradient
33
34     if t == 1
35         sg1 = mean(sg,2);
36     else
37         sg1 = sg1 - sg(:, i)./n;
38         [f, sg(:, i)] = hingeLossSubGrad(w,Xt,y,lambda,i);

```

```

39         sg1 = sg1 + sg(:, i)./n;
40     end
41     % Set step size
42     alpha = 1/(lambda*t);
43
44     % Take stochastic subgradient step
45     w = w - alpha*(sg1 + lambda*w);
46
47 end
48
49 model.w = w;
50 model.predict = @predict;
51
52 end
53
54 function [yhat] = predict(model,Xhat)
55 [t,d] = size(Xhat);
56 Xhat = [ones(t,1) Xhat];
57 w = model.w;
58 yhat = sign(Xhat*w);
59 end
60
61
62 function [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i)
63
64 [d,n] = size(Xt);
65
66 % Function value
67 wtx = w'*Xt(:,i);
68 loss = max(0,1-y(i)*wtx);
69 f = loss;
70
71 % Subgradient
72 if loss > 0
73     sg = -y(i)*Xt(:,i);
74 else
75     sg = sparse(d,1);
76 end
77 end

```