Project-2

Tianran Zhang 15300180104

November 19, 2017

Contents

1 Logistic Regression

1.1 Bayes´ Rule

$$x = (x_1, ..., x_D)^T \tag{1}$$
$$p(y = 1) = \alpha \tag{2}$$
$$p(y = 0) = 1 - \alpha \tag{3}$$
$$p(x_i|y = 0) = N(\mu_{i0}, \sigma_i^2) = \frac{1}{\sqrt{2\pi\sigma_i^2}} exp\{-\frac{(x - \mu_{i0})^2}{2\sigma_i^2}\} \tag{4}$$
$$p(x_i|y = 1) = N(\mu_{i1}, \sigma_i^2) = \frac{1}{\sqrt{2\pi\sigma_i^2}} exp\{-\frac{(x - \mu_{i1})^2}{2\sigma_i^2}\} \tag{5}$$

$$\begin{aligned} p(y = 1|x) &= \frac{p(x|y = 1)p(y = 1)}{p(x|y = 1)p(y = 1) + p(x|y = 0)p(y = 0)} && (6) \\ &= \frac{\alpha p(x|y = 1)}{\alpha p(x|y = 1) + p(x|y = 0)(1 - \alpha)} && (7) \\ &= \frac{1}{1 + \frac{1-\alpha}{\alpha}\frac{p(x|y=0)}{p(x|y=1)}} && (8) \\ &= \frac{1}{1 + \frac{1-\alpha}{\alpha}\prod_{i=1}^{D}\frac{p(x_i|y=0)}{p}(x_i|y = 1)} && (9) \\ &= \frac{1}{1 + exp\{log\frac{1-\alpha}{\alpha} + \sum_{i=1}^{D} -\frac{1}{2\sigma_i^2}[(2\mu_{i1} - 2\mu_{i0})x_i + \mu_{i0}^2 - \mu_{i1}^2]\}} && (10) \\ &= \frac{1}{1 + exp(-\sum_{i=1}^{D} w_i x_i - b)} && (11) \\ &= \sigma(w^T x + b) && (12) \end{aligned}$$

where

$$w_i = \frac{\mu_{i1} - \mu_{i0}}{\sigma_i^2}$$
$$b = \sum i = 1^D \frac{\mu_{i0}^2 - \mu_{i1}^2}{2\sigma_i^2} - log\frac{1 - \alpha}{\alpha}$$
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

1.2 Maximum Likelihood Estimation

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

$$E(w,b) = -log(p(y|w,b)) \tag{14}$$

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

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

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

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

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

1.3 L2 Regularization

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

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

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

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

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

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

$$L(w,b) = -logp'(w,b) \tag{25}$$

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

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

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

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

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

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:

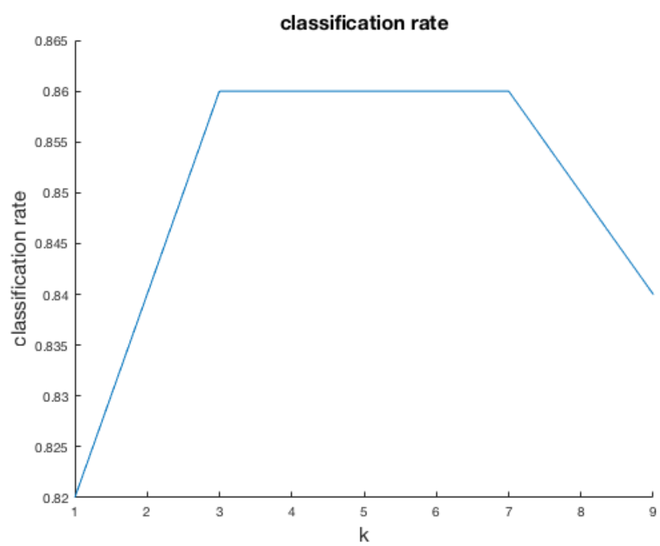

Figure 1: The classification rate for different k

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:

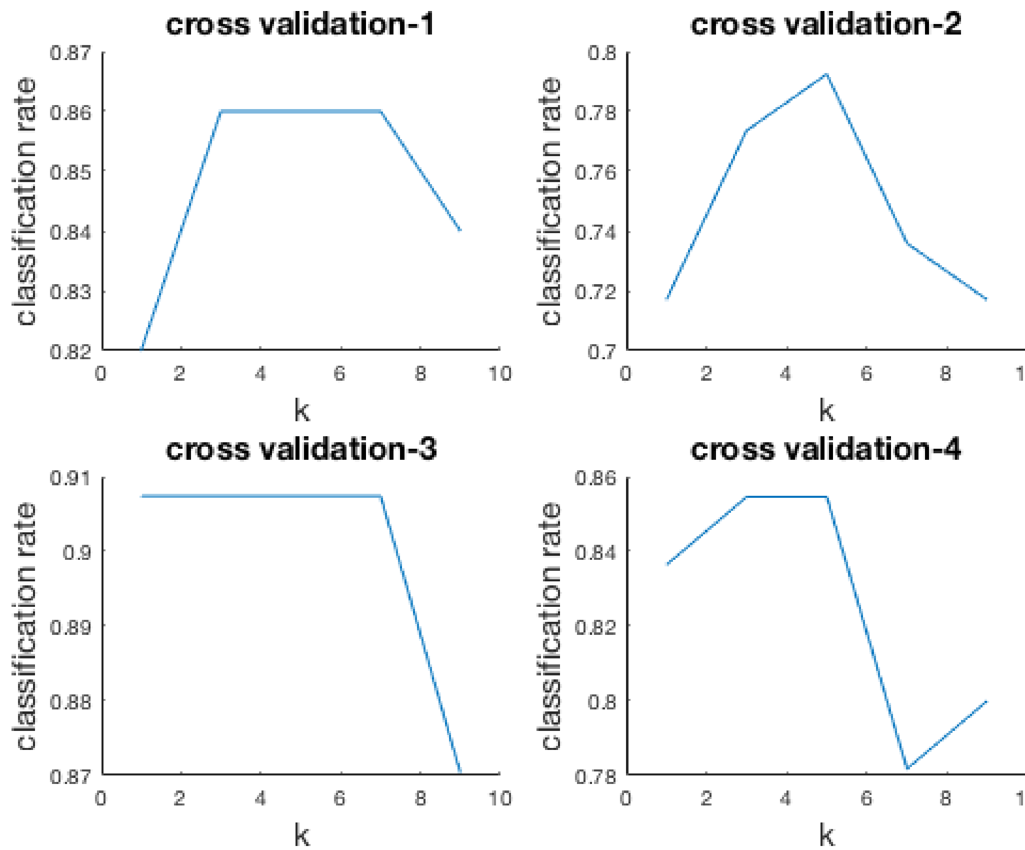

Figure 2: The classification rate against k in different cross-validation

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:

```
Codes in matlab:
  kstar = 5;
  krange = [kstar-2, kstar, kstar+2];
  disp(r(krange));

The outputs are:
    0.8600
    0.8600
    0.8600
```

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

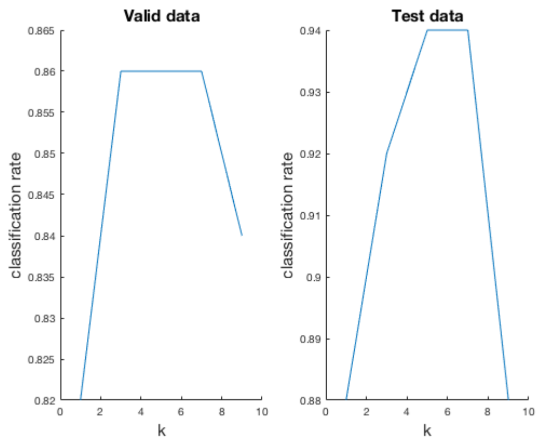

Figure 3: The classification against k for valid and test data

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=1or 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

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

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:

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

and

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

where

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

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):

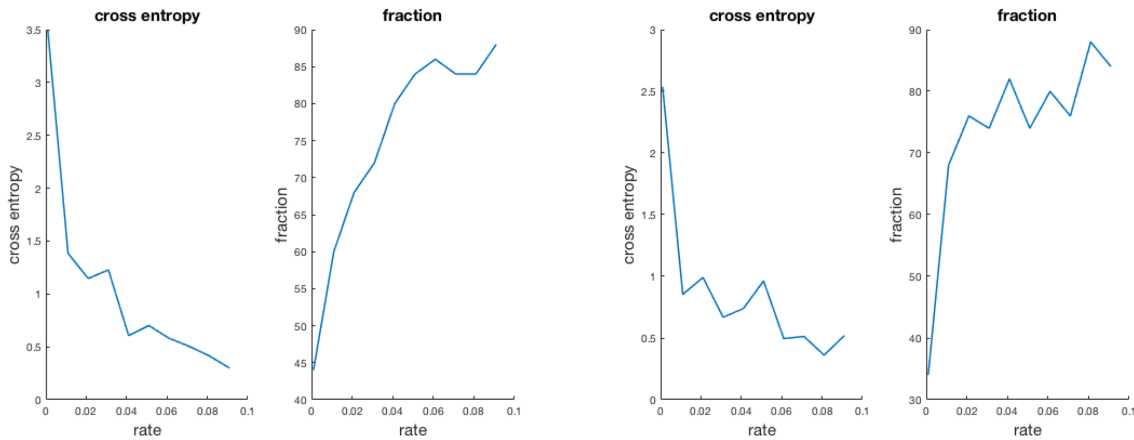

Figure 4: the cross entropy and fraction when rate is from 0.001 to 0.1(run the code for several times and choose two of them).

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

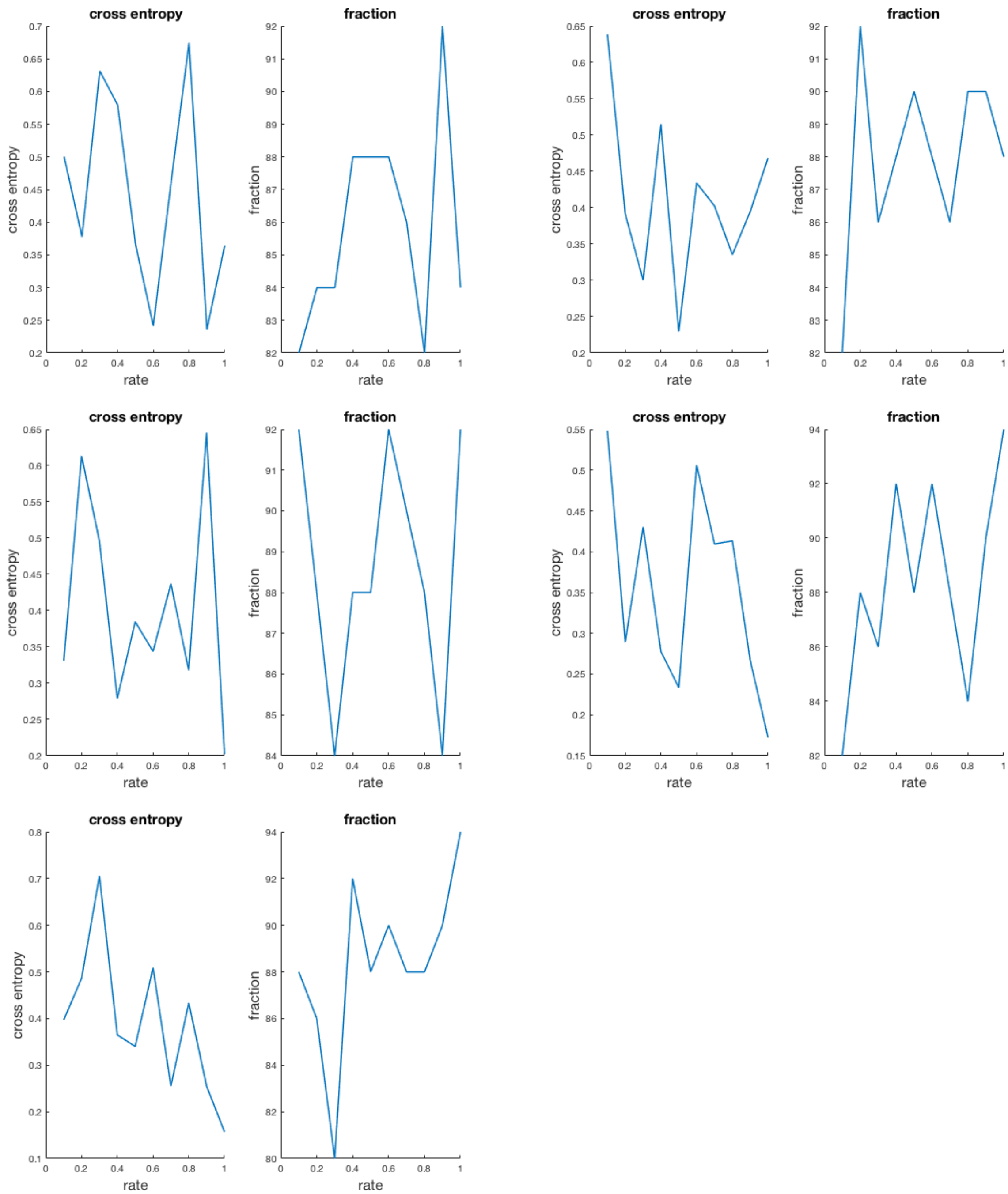

Figure 5: the cross entropy and fraction when rate is from 0.1 to 0.1(run the code for several times and choose five of them).

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

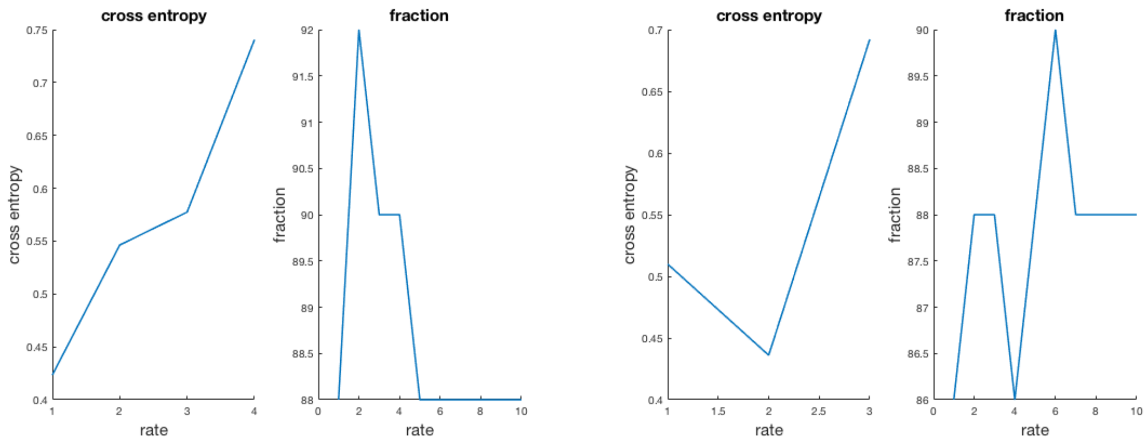

Figure 6: the cross entropy and fraction when rate is from 1 to 10(run the code for several times and choose two of them).

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):

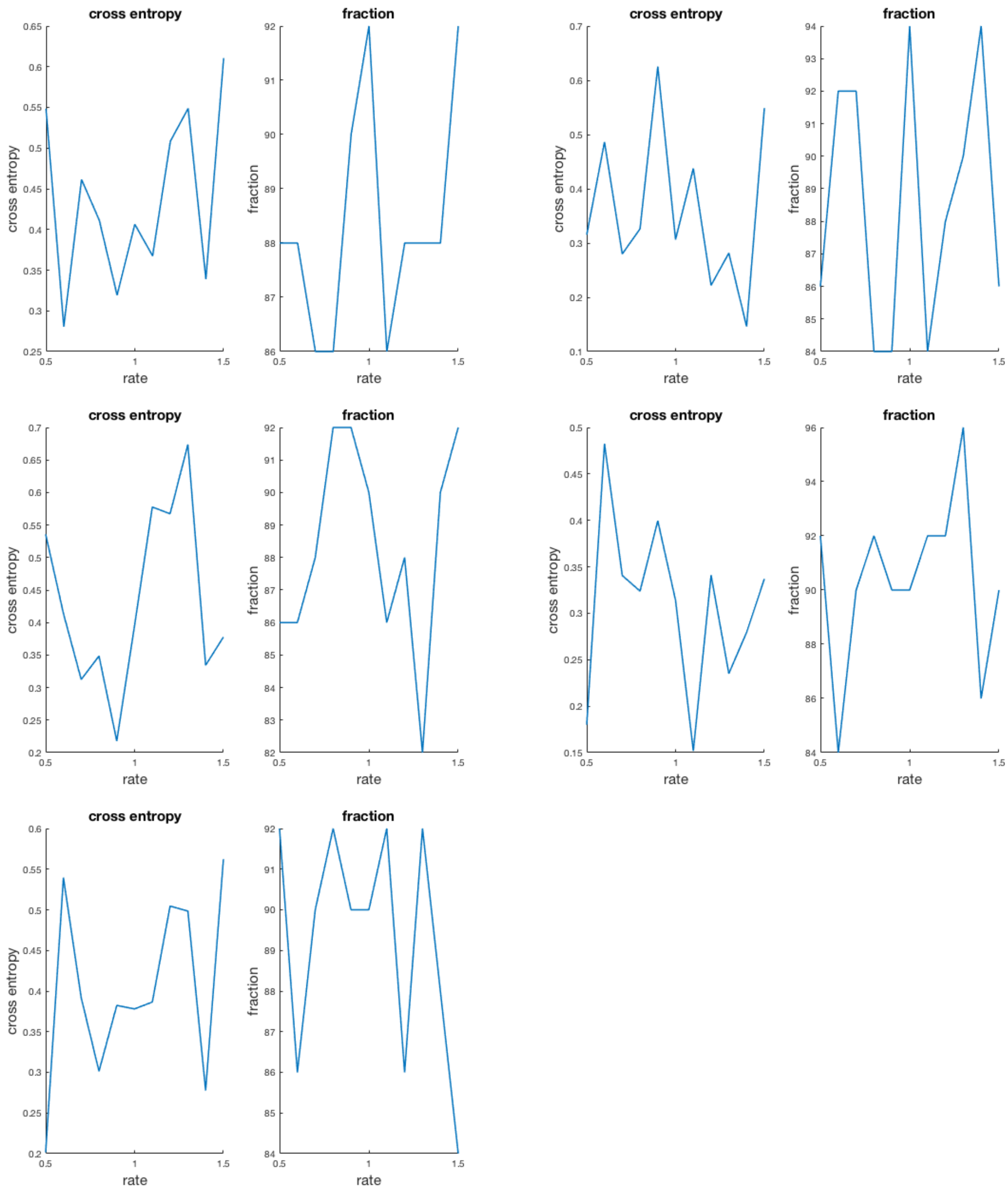

Figure 7: the cross entropy and fraction when rate is from 0.5 to 1.5(run the code for several times and choose five of them).

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:

```
TRAIN CE:  0.004506        TRAIN FRAC:100.00
VALIC CE:  0.399154        VALID FRAC:88.00
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:

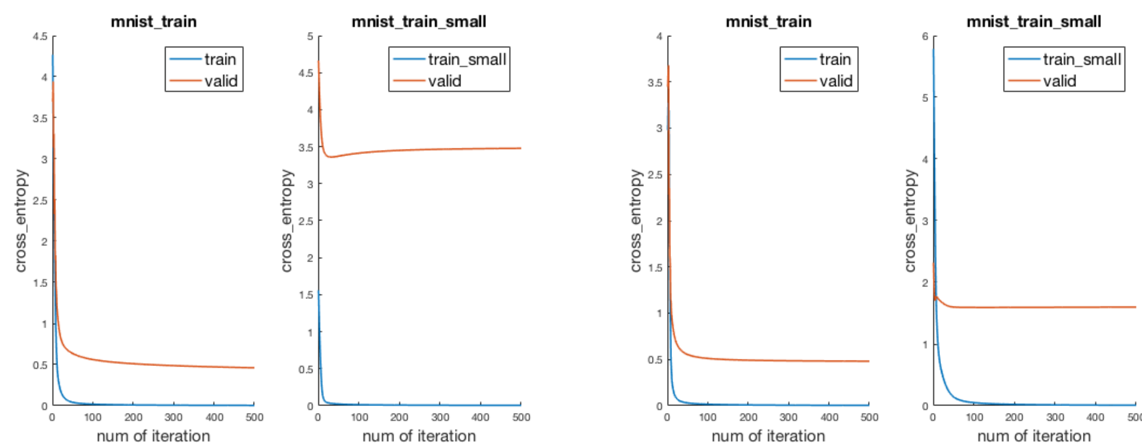

Figure 8: the cross entropy against the num of iteration (run the code for several times and choose two of them).

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

$$
weights^T = (w_1, w_2, ...w_n, b)(w^T = (w_1, w_2, ...w_n))
$$

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 λ for both mnist_train and mnist_train_small. The plots are shown below:

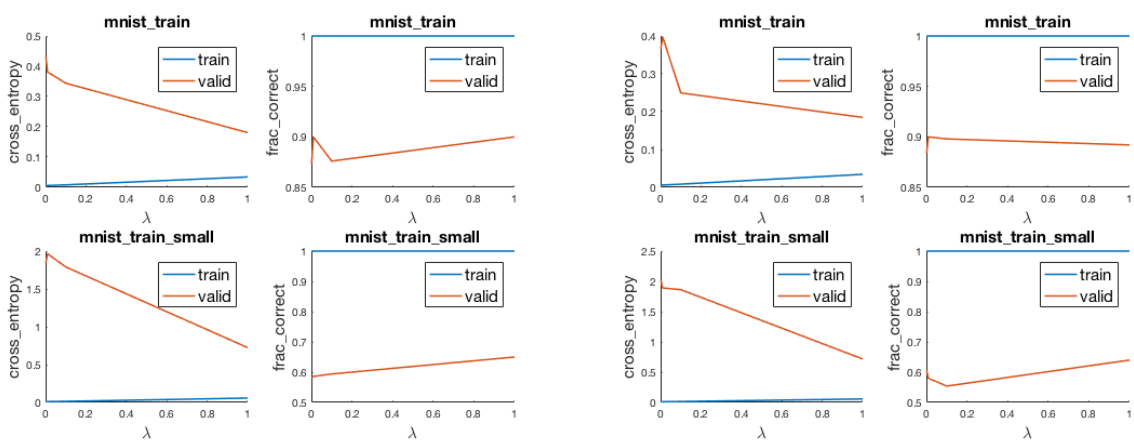

Figure 9: average cross entropy and classification error against λ (run the code for several times and choose two of them).

According to the plots, we observe the entropy and classification error change when λ 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 λ is bigger, the restrain of weights is tighter thus make the classifier works better

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

```
TEST CE:    0.235382          TEST FRAC:  93.20
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:

```
TEST CE:    0.206912          TEST FRAC:  92.40
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 the minimize of E(wights), after adding a regularized term, it comes to the minimize of $L(wights) = E(wights) + \frac{\lambda}{2}\sum_{i=1}^{D} wights_i^2$. According to Lagrange multiplication operator, this question turns into a constrained minimum question: to obtain the minimize of L(wights) subject to $\sum_{i=1}^{D} wights_i \leq \frac{1}{\lambda}$. So the E(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 :

```
training accuracy: 86.250000
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:

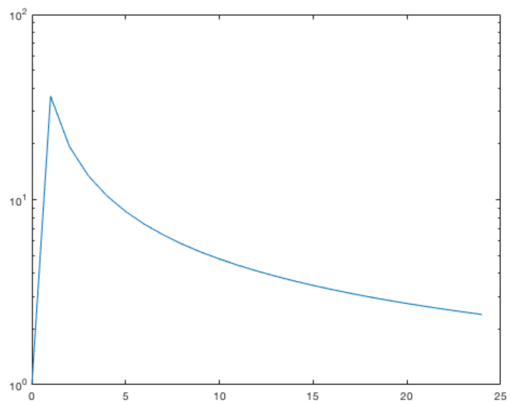

Figure 11: The performance with averaging

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:

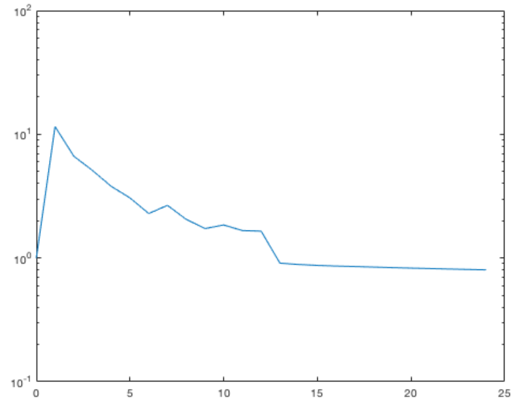

Figure 12: the performance of "second-half" averaging

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 iteraition, so we do a little change to make the model more effective by the following steps:
a. keep in memory the gradients of all functions f_i, i=1,...,n
b. Random selection $i(t) \in 1, ..., n$ with replacement
c. For each iteration: $w^{t+1} = w^t - \alpha_t \frac{\sum_{i=1}^n g_{it}}{n} - \alpha \lambda w^t$,
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:

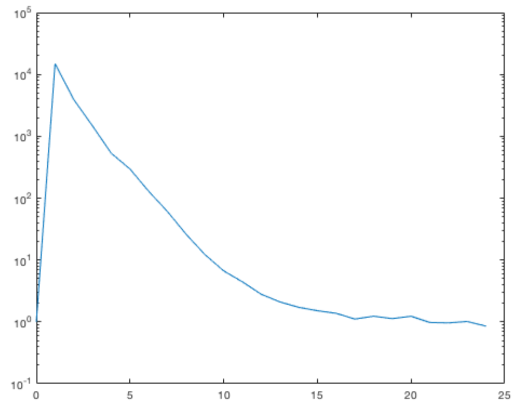

Figure 13: The performance of modified svm

4 Codes for report

4.1 knn.m

```
clear all;
close all;

load mnist_train;
load mnist_valid;

[n, m] = size(valid_inputs);

r = zeros(n,1);

for k = 1:2:9
    valid_predict = run_knn(k, train_inputs, train_targets, valid_inputs);
    r(k) = sum([valid_predict==valid_targets]) / n;
end

figure;
hold on;
title('Valid data', 'FontSize', 15);
plot(1:2:9,r(1:2:9));
xlabel('k', 'FontSize', 15);
ylabel('classification rate', 'FontSize', 15);
```

4.2 chosenK.m

```
% find the best k in knn
clear all;
```

```
close all;

load mnist_train;
load mnist_valid;

[n, m] = size(valid_inputs);
[n_train, m_train] = size(train_inputs);
t = floor(n_train/3);
step = [1, t, 2*t, n_train];
r = zeros(4,9);

for k = 1:2:9
    valid_predict = run_knn(k, train_inputs, train_targets, valid_inputs);
    r(1,k) = sum([valid_predict==valid_targets]) / n;
end

for t = 2:4
    t1_inputs = [valid_inputs; train_inputs(1:(step(t-1)-1), : );
        train_inputs((step(t)+1):n_train, : )];
    t1_targets = [valid_targets; train_targets(1:(step(t-1)-1), : );
    train_targets((step(t)+1):n_train, : )];
    t2_inputs = train_inputs(step(t-1):step(t), : );
    t2_targets = train_targets(step(t-1):step(t), : );
    for k = 1:2:9
        t_predict = []
        t_predict = run_knn(k, t1_inputs, t1_targets, t2_inputs);
        r(t,k) = sum([t_predict == t2_targets]) / (step(t)-step(t-1)+1);
    end
end

%plot some features
figure;
subplot(2,2,1);
hold on;
title('cross validation-1', 'FontSize', 15);
plot(1:2:9,r(1 , 1:2:9));
xlabel('k', 'FontSize', 15);
ylabel('classification rate', 'FontSize', 15);

subplot(2,2,2);
hold on;
title('cross validation-2', 'FontSize', 15);
plot(1:2:9,r(2 , 1:2:9));
xlabel('k', 'FontSize', 15);
ylabel('classification rate', 'FontSize', 15);
```

```matlab

subplot(2,2,3);
hold on;
title('cross validation-3', 'FontSize', 15);
plot(1:2:9,r(3 , 1:2:9));
xlabel('k', 'FontSize', 15);
ylabel('classification rate', 'FontSize', 15);

subplot(2,2,4);
hold on;
title('cross validation-4', 'FontSize', 15);
plot(1:2:9,r(4 , 1:2:9));
xlabel('k', 'FontSize', 15);
ylabel('classification rate', 'FontSize', 15);
```

4.3 logistic_predict.m

```matlab
function [y] = logistic_predict(weights, data)
%    Compute the probabilities predicted by the logistic classifier.
%
%    Note: N is the number of examples and
%          M is the number of features per example.
%
%    Inputs:
%        weights:    (M+1) x 1 vector of weights, where the last element
%                    corresponds to the bias (intercepts).
%        data:       N x M data matrix where each row corresponds
%                    to one data point.
%    Outputs:
%        y:          :N x 1 vector of probabilities. This is the output of
%                       the classifier.

[n,m] = size(data);
data = [data , ones(n,1)];
y = sigmoid(data * weights);
end
```

4.4 logistic.m

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

```
%        weights:     (M+1) x 1 vector of weights, where the last element
%                 corresponds to bias (intercepts).
%        data:        N x M data matrix where each row corresponds
%                 to one data point.
%        targets:     N x 1 vector of binary targets. Values should be either
%0 or 1.
%   hyperparameters: The hyperparameter structure
%
% Outputs:
%        f:             The scalar error value?i.e. negative log likelihood).
%        df:          (M+1) x 1 vector of derivatives of error w.r.t. weights.
%       y:          N x 1 vector of probabilities. This is the output of the
%                         classifier.

[n, m] = size(data);
x = [data , ones(n,1)];
w = x*weights;
f = -sum(x * weights .* targets)+ sum(log(1+ exp(x * weights)));
df = -x' * (targets-sigmoid(x * weights));
y = logistic_predict(weights, data);
end
```

4.5 chosen_rate

```
%% Clear workspace.
clear all;
close all;

%% Load data.
load mnist_train;
load mnist_valid;
load mnist_test;

r0=0;
ce = [];
frac=[];
N = size(train_inputs,1);
tic;
for r = 0.5:0.1:1.5
    r0 = r0+1;
    % Initialize hyperparameters.
    % Learning rate
    hyperparameters.learning_rate = r;
    hyperparameters.weight_regularization = 0;
    hyperparameters.num_iterations = 300;
```

```
    weights = randn((size(train_inputs,2)+1),1);
    weights_small = weights;

    cross_entropy_train = zeros( hyperparameters.num_iterations, 1 );
    cross_entropy_train_small = cross_entropy_train;
    cross_entropy_valid = cross_entropy_train;
    cross_entropy_valid_small = cross_entropy_train;

    %% Begin learning with gradient descent.
    for t = 1:hyperparameters.num_iterations
        % Find the negative log likelihood and derivative w.r.t. weights.
        [f, df, predictions] = logistic(weights, ...
            train_inputs, ...
            train_targets, ...
            hyperparameters);

        [cross_entropy_train(t), frac_correct_train] =
        evaluate(train_targets, predictions);

        if isnan(f) || isinf(f)
            error('nan/inf error');
        end

        %% Update parameters.
        weights = weights - hyperparameters.learning_rate .* df/ N;
        predictions_valid = logistic_predict(weights, valid_inputs);
        [cross_entropy_valid(t), frac_correct_valid] =
        evaluate(valid_targets, predictions_valid);

        predictions_test = logistic_predict(weights, test_inputs);
        [cross_entropy_test, frac_correct_test] = evaluate(test_targets,
        predictions_test);

        %% Print some stats.
        fprintf(1, 'ITERATION:%4i   NLOGL:%4.2f\n TRAIN CE %.6f
        TRAIN FRAC:%2.2f\t VALIC_CE %.6f VALID FRAC:%2.2f\t ',...
            t, f/N, cross_entropy_train(t), frac_correct_train*100,
            cross_entropy_valid(t), frac_correct_valid*100);
        fprintf(1, 'TEST CE %.6f TEST FRAC:%2.2f
        \n',cross_entropy_test, frac_correct_test*100);

    end
    ce(r0)= cross_entropy_valid(t);
    frac(r0) = frac_correct_valid*100;
end
```

```
toc;
% draw fome features
x = 0.5:0.1:1.5;
figure;
subplot(1,2,1);
hold on;
title('cross entropy', 'FontSize', 15);
plot(x, ce, 'LineWidth', 1.5);
xlabel('rate', 'FontSize', 15);
ylabel('cross entropy', 'FontSize', 15);
subplot(1,2,2);
hold on;
plot(x, frac, 'LineWidth', 1.5);
title('fraction', 'FontSize', 15);
xlabel('rate', 'FontSize', 15);
ylabel('fraction', 'FontSize', 15);
```

4.6 logistic_pen.m

```
function [f, df, y] = logistic_pen(weights, data, targets,
hyperparameters)
% Calculate log likelihood and derivatives with respect to weights.
%
% Note: N is the number of examples and
%       M is the number of features per example.
%
% Inputs:
%       weights:    (M+1) x 1 vector of weights, where the last element
%                   corresponds to bias (intercepts).
%       data:       N x M data matrix where each row corresponds
%                   to one data point.
%   targets:    N x 1 vector of targets class probabilities.
%   hyperparameters: The hyperparameter structure
%
% Outputs:
%       f:              The scalar error value.
%       df:         (M+1) x 1 vector of derivatives of error w.r.t. weights.
%      y:         N x 1 vector of probabilities. This is the output of the
%                           classifier.
%

%TODO: finish this function

[n,m] = size(data);
y = sigmoid([data,ones(n,1)] * weights);
```

```
[f1, df1, y1] = logistic(weights, data, targets, hyperparameters);
weights(length(weights))=0;
h = hyperparameters.weight_regularization;
f = f1 + h/2 * sum(weights.^2);
df = df1 + h * weights;

end
```

4.7 logistic_regression_template.m

```
%% Clear workspace.
clear all;
close all;

%% Load data.
load mnist_train;
load mnist_valid;
load mnist_train_small;
load mnist_test;

tic;
% Learning rate
hyperparameters.learning_rate = 0.00625; %1   %...
% Weight regularization parameter
hyperparameters.weight_regularization = 0;    %...
% Number of iterations
hyperparameters.num_iterations = 500;   %...
% Logistics regression weights
weights = randn((size(train_inputs,2)+1),1);
weights_small = weights;

cross_entropy_train = zeros( hyperparameters.num_iterations, 1 );
cross_entropy_train_small = cross_entropy_train;
cross_entropy_valid = cross_entropy_train;
cross_entropy_valid_small = cross_entropy_train;

%% Verify that your logistic function produces the right gradient, diff
should be very close to 0

% this creates small random data with 20 examples and 10
dimensions and checks the gradient on
% that data.
nexamples = 20;
ndimensions = 10;
```

```
diff = checkgrad('logistic', ...
                     randn((ndimensions + 1), 1), ...   % weights
                 0.001,...                              % perturbation
                 randn(nexamples, ndimensions), ...     % data
                 rand(nexamples, 1), ...                % targets
                 hyperparameters)                       % other hyperparameters

N = size(train_inputs,1);
N_small = size(train_inputs_small,1);

%% Begin learning with gradient descent.
for t = 1:hyperparameters.num_iterations

        % Find the negative log likelihood and derivative w.r.t. weights.
        [f, df, predictions] = logistic(weights, ...
                                        train_inputs, ...
                                        train_targets, ...
                                        hyperparameters);

    [f_small, df_small, predictions_small] = logistic(weights_small, ...
                                                      train_inputs_small, ...
                                                      train_targets_small, ..
                                                      hyperparameters);

    [cross_entropy_train(t), frac_correct_train] = evaluate(train_targets,
    predictions);
    [cross_entropy_train_small(t), frac_correct_train_samll] =
    evaluate(train_targets_small, predictions_small);

    if isnan(f) || isinf(f)
            error('nan/inf error');
    end


    if isnan(f_small) || isinf(f_small)
            error('nan/inf error');
        end

    %% Update parameters.
    weights = weights - hyperparameters.learning_rate .* df;
    predictions_valid = logistic_predict(weights, valid_inputs);
    [cross_entropy_valid(t), frac_correct_valid] =
    evaluate(valid_targets, predictions_valid);

    weights_small = weights_small - hyperparameters.learning_rate .*
```

```
    df_small;
    predictions_valid_small = logistic_predict(weights_small,
    valid_inputs);
    [cross_entropy_valid_small(t), frac_correct_valid_small] =
    evaluate(valid_targets, predictions_valid_small);

    predictions_test = logistic_predict(weights, test_inputs);
    [cross_entropy_test, frac_correct_test] = evaluate(test_targets,
    predictions_test);

        %% Print some stats.
        fprintf(1, 'ITERATION:%4i   NLOGL:%4.2f\n TRAIN CE %.6f
        TRAIN FRAC:%2.2f\t VALIC_CE %.6f VALID FRAC:%2.2f\t ',...
                t, f/N, cross_entropy_train(t), frac_correct_train*100,
                cross_entropy_valid(t), frac_correct_valid*100);
    fprintf(1, 'TEST CE %.6f TEST FRAC:%2.2f\n',cross_entropy_test,
    frac_correct_test*100);

end
toc;

figure;
subplot(1,2,1);
hold on;
title('mnist\_train', 'FontSize', 15);
plot(1:hyperparameters.num_iterations, cross_entropy_train,
'LineWidth', 1.5);
plot(1:hyperparameters.num_iterations, cross_entropy_valid,
'LineWidth', 1.5);
l = legend('train', 'valid');
set(l, 'FontSize', 15);
xlabel('num of iteration', 'FontSize', 15);
ylabel('cross\_entropy', 'FontSize', 15);

subplot(1,2,2);
hold on;
title('mnist\_train\_small', 'FontSize', 15);
plot(1:hyperparameters.num_iterations, cross_entropy_train_small,
'LineWidth', 1.5);
plot(1:hyperparameters.num_iterations, cross_entropy_valid_small,
'LineWidth', 1.5);
l = legend('train\_small', 'valid');
set(l, 'FontSize', 15);
xlabel('num of iteration', 'FontSize', 15);
ylabel('cross\_entropy', 'FontSize', 15);
```

4.8 logistic_regression_penalized.m

```
%% Clear workspace.
clear all;
close all;

%% Load data.
load mnist_train;
load mnist_valid;
load mnist_train_small;

lambda = [0.001, 0.01, 0.1, 1];
cross_entropy_train_lambda = zeros(4, 1);
cross_entropy_valid_lambda = zeros(4, 1);
frac_correct_train_lambda = zeros(4, 1);
frac_correct_valid_lambda = zeros(4, 1);

cross_entropy_train_lambda_small = zeros(4, 1);
cross_entropy_valid_lambda_small = zeros(4, 1);
frac_correct_train_lambda_small = zeros(4, 1);
frac_correct_valid_lambda_small = zeros(4, 1);

tic;
for la = 1:4
    %% TODO: Initialize hyperparameters.
    % Learning rate
    hyperparameters.learning_rate = 0.00625;   %...
    % Weight regularization parameter
    hyperparameters.weight_regularization = lambda(la);   %...
    % Number of iterations
    hyperparameters.num_iterations = 300;  %...
    % Logistics regression weights
    % TODO: Set random weights.

 %   cross_entropy_train = zeros( hyperparameters.num_iterations,
 10 );
  %  cross_entropy_valid = cross_entropy_train;
  %  frac_correct_train = cross_entropy_train;
  %  frac_correct_valid = cross_entropy_train;

  %  cross_entropy_train_small =
  zeros( hyperparameters.num_iterations, 10 );
  %  cross_entropy_valid_small = cross_entropy_train_small;
  %  frac_correct_train_small = cross_entropy_train_small;
  %  frac_correct_valid_small = cross_entropy_valid_small;
```

```
    for k = 1:10
        weights = randn((size(train_inputs,2)+1),1);
        weights_small = weights;
        N = size(train_inputs,1);
        N_small = size(train_inputs_small,1);

        for t = 1:hyperparameters.num_iterations

            % Find the negative log likelihood and derivative w.r.t.
            weights.
            [f, df, predictions] = logistic_pen(weights, ...
                                                train_inputs, ...
                                                train_targets, ...
                                                hyperparameters);
            [f_small, df_small, predictions_small] =
            logistic_pen(weights_small, ...
                                    train_inputs_small, ...
                                    train_targets_small, ...
                                    hyperparameters);

            [cross_entropy_train, frac_correct_train] =
            evaluate(train_targets, predictions);
            [cross_entropy_train_small, frac_correct_train_small] =
            evaluate(train_targets_small, predictions_small);
            %cross_entropy_train
            %????
            %% Find the fraction of correctly classified validation
            examples.
        %    [temp, temp2, frac_correct_valid] = logistic(weights, ...
        %                                      valid_inputs, ...
        %                                       valid_targets, ...
        %                                        hyperparameters);

%            [temp, temp2, frac_correct_valid_small] =
logistic(weights, ...
%                                                    valid_inputs, ...
%                                                     valid_targets, ...
%                                                     hyperparameters);
            if isnan(f) || isinf(f)
                error('nan/inf error');
            end
            if isnan(f_small) || isinf(f_small)
                error('nan/inf error');
            end
```

```
            %% Update parameters.
            weights = weights - hyperparameters.learning_rate .* df;%/N;
            predictions_valid = logistic_predict(weights, valid_inputs);
            [cross_entropy_valid, frac_correct_valid] =
            evaluate(valid_targets, predictions_valid);

            weights_small = weights_small -
            hyperparameters.learning_rate .* df_small;%/N_small;
            predictions_valid_small = logistic_predict(weights_small,
            valid_inputs);
            [cross_entropy_valid_small, frac_correct_valid_small] =
            evaluate(valid_targets, predictions_valid_small);

            %% Print some stats.
             fprintf(1, 'ITERATION:%4i   NLOGL:%4.2f TRAIN CE %.6f
             TRAIN FRAC:%2.2f VALIC_CE %.6f VALID FRAC:%2.2f\n',...
                    t, f/N, cross_entropy_train, frac_correct_train*100,
                    cross_entropy_valid, frac_correct_valid*100);

        end
        cross_entropy_train_lambda(la) =
        cross_entropy_train_lambda(la) + cross_entropy_train;
        cross_entropy_valid_lambda(la) =
        cross_entropy_valid_lambda(la) + cross_entropy_valid;
        frac_correct_train_lambda(la) = frac_correct_train_lambda(la) +
        frac_correct_train;
        frac_correct_valid_lambda(la) = frac_correct_valid_lambda(la) +
        frac_correct_valid;

        cross_entropy_train_lambda_small(la) =
        cross_entropy_train_lambda_small(la) +
        cross_entropy_train_small;
        cross_entropy_valid_lambda_small(la) =
        cross_entropy_valid_lambda_small(la) +
        cross_entropy_valid_small;
        frac_correct_train_lambda_small(la) =
        frac_correct_train_lambda_small(la) + frac_correct_train_small;
        frac_correct_valid_lambda_small(la) =
        frac_correct_valid_lambda_small(la) + frac_correct_valid_small;

    end
    cross_entropy_train_lambda(la) = cross_entropy_train_lambda(la)/
    10;
    cross_entropy_valid_lambda(la) = cross_entropy_valid_lambda(la)/
```

```
    10;
    frac_correct_train_lambda(la) = frac_correct_train_lambda(la)/10;
    frac_correct_valid_lambda(la) = frac_correct_valid_lambda(la)/10;

    cross_entropy_train_lambda_small(la) =
    cross_entropy_train_lambda_small(la)/10;
    cross_entropy_valid_lambda_small(la) =
    cross_entropy_valid_lambda_small(la)/10;
    frac_correct_train_lambda_small(la) =
    frac_correct_train_lambda_small(la)/10;
    frac_correct_valid_lambda_small(la) =
    frac_correct_valid_lambda_small(la)/10;
end

toc;

figure;
subplot(2,2,1);
hold on;
title('mnist\_train', 'FontSize', 15);
plot(lambda, cross_entropy_train_lambda, 'LineWidth', 1.5);
plot(lambda, cross_entropy_valid_lambda, 'LineWidth', 1.5);
l = legend('train', 'valid');
set(l, 'FontSize', 15);
xlabel('\lambda', 'FontSize', 15);
ylabel('cross\_entropy', 'FontSize', 15);

subplot(2,2,2);
hold on;
title('mnist\_train', 'FontSize', 15);
plot(lambda, frac_correct_train_lambda, 'LineWidth', 1.5);
plot(lambda, frac_correct_valid_lambda, 'LineWidth', 1.5);
l = legend('train', 'valid');
set(l, 'FontSize', 15);
xlabel('\lambda', 'FontSize', 15);
ylabel('frac\_correct', 'FontSize', 15);


subplot(2,2,3);
hold on;
title('mnist\_train\_small', 'FontSize', 15);
plot(lambda, cross_entropy_train_lambda_small, 'LineWidth', 1.5);
plot(lambda, cross_entropy_valid_lambda_small, 'LineWidth', 1.5);
l = legend('train', 'valid');
set(l, 'FontSize', 15);
```

```
xlabel('\lambda', 'FontSize', 15);
ylabel('cross\_entropy', 'FontSize', 15);


subplot(2,2,4);
hold on;
title('mnist\_train\_small', 'FontSize', 15);
plot(lambda, frac_correct_train_lambda_small, 'LineWidth', 1.5);
plot(lambda, frac_correct_valid_lambda_small, 'LineWidth', 1.5);
l = legend('train', 'valid');
set(l, 'FontSize', 15);
xlabel('\lambda', 'FontSize', 15);
ylabel('frac\_correct', 'FontSize', 15);
```

4.9 run_nb.m

```
% Learn a Naive Bayes classifier on the digit dataset, evaluate its
% performance on training and test sets, then visualize the mean and
variance
% for each class.

load mnist_train;
load mnist_test;

[log_prior, class_mean, class_var] = train_nb(train_inputs,
train_targets);
[Trainprediction, TrainAccuracy] = test_nb(train_inputs, train_targets,
log_prior, class_mean, class_var);
[Testprediction, TestAccuracy] = test_nb(test_inputs, test_targets,
log_prior, class_mean, class_var);

fprintf('training accuracy: %f\n', TrainAccuracy*100);
fprintf('test accuracy: %f\n', TestAccuracy*100);
plot_digits(class_mean);
plot_digits(class_var);
```

4.10 svmAvg.m

```
function [model] = svmAvg1(X,y,lambda,maxIter)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];

% Matlab indexes by columns,
%  so if we are accessing rows it will be faster to use  the traspose
```

```
Xt = X';

% Initial values of regression parameters
w = zeros(d+1,1);
w_mean = w;

% Apply stochastic gradient method
for t = 1:maxIter
    if mod(t-1,n) == 0
        % Plot our progress
        % (turn this off for speed)

        objValues(1+(t-1)/n) = (1/n)*sum(max(0,1-y.*(X*w_mean))) +
        (lambda/2)*(w_mean'*w_mean);
        semilogy([0:t/n],objValues);
        pause(.1);
    end

    % Pick a random training example
    i = ceil(rand*n);

    % Compute sub-gradient
    [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i);

    % Set step size
    alpha = 1/(lambda*t);

    % Take stochastic subgradient step
    w = w - alpha*(sg + lambda*w);
    if t >1
        w_mean = (w_mean*(t-1)+w)/t;
    else
        w_mean = w;
    end
end

model.w = w;
model.predict = @predict;

end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);
Xhat = [ones(t,1) Xhat];
w = model.w;
```

```
yhat = sign(Xhat*w);
end

function [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i)

[d,n] = size(Xt);

% Function value
wtx = w'*Xt(:,i);
loss = max(0,1-y(i)*wtx);
f = loss;

% Subgradient
if loss > 0
    sg = -y(i)*Xt(:,i);
else
    sg = sparse(d,1);
end
end
```

4.11 svmAvg1.m

```
function [model] = svmAvg2(X,y,lambda,maxIter)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];

% Matlab indexes by columns,
%  so if we are accessing rows it will be faster to use  the traspose
Xt = X';

% Initial values of regression parameters
w = zeros(d+1,1);
w_mean = w;
half = ceil(maxIter/2);
% Apply stochastic gradient method
for t = 1:maxIter
    if mod(t-1,n) == 0
        % Plot our progress
        % (turn this off for speed)

        objValues(1+(t-1)/n) = (1/n)*sum(max(0,1-y.*(X*w_mean))) +
        (lambda/2)*(w_mean'*w_mean);
        semilogy([0:t/n],objValues);
```

```
        pause(.1);
    end

    % Pick a random training example
    i = ceil(rand*n);

    % Compute sub-gradient
    [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i);

    % Set step size
    alpha = 1/(lambda*t);

    % Take stochastic subgradient step
    w = w - alpha*(sg + lambda*w);
    if t >half
        w_mean = (w_mean*(t-half)+w)/(t-half+1);
    else
        w_mean = w;
    end
end

model.w = w;
model.predict = @predict;

end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);
Xhat = [ones(t,1) Xhat];
w = model.w;
yhat = sign(Xhat*w);
end

function [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i)

[d,n] = size(Xt);

% Function value
wtx = w'*Xt(:,i);
loss = max(0,1-y(i)*wtx);
f = loss;

% Subgradient
if loss > 0
    sg = -y(i)*Xt(:,i);
```

```
else
    sg = sparse(d,1);
end
end
```

4.12 svm_new.m

```
function [model] = svmAvg(X,y,lambda,maxIter)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];

% Matlab indexes by columns,
%  so if we are accessing rows it will be faster to use  the traspose
Xt = X';

% Initial values of regression parameters
w = zeros(d+1, 1);
sg1 = w;
for i = 1 : (d+1)
    sg(i, :) = Xt(i, :) .* y(i);
end

% Apply stochastic gradient method
for t = 1:maxIter
    if mod(t-1,n) == 0
        % Plot our progress
        % (turn this off for speed)
        objValues(1+(t-1)/n) = (1/n)*sum(max(0,1-y.*(X*w))) + (lambda/
        2)*(w'*w);
        semilogy([0:t/n],objValues);
        pause(.1);
    end

    % Pick a random training example
    i = ceil(rand*n);

    % Compute sub-gradient

    if t == 1
        sg1 = mean(sg,2);
    else
        sg1 = sg1 - sg(:, i)./n;
        [f, sg(:  , i)] = hingeLossSubGrad(w,Xt,y,lambda,i);
```

```
            sg1 = sg1 + sg(:, i)./n;
        end
        % Set step size
        alpha = 1/(lambda*t);

        % Take stochastic subgradient step
        w = w - alpha*(sg1 + lambda*w);

end

model.w = w;
model.predict = @predict;

end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);
Xhat = [ones(t,1) Xhat];
w = model.w;
yhat = sign(Xhat*w);
end


function [f,sg] = hingeLossSubGrad(w,Xt,y,lambda,i)

[d,n] = size(Xt);

% Function value
wtx = w'*Xt(:,i);
loss = max(0,1-y(i)*wtx);
f = loss;

% Subgradient
if loss > 0
    sg = -y(i)*Xt(:,i);
else
    sg = sparse(d,1);
end
end
```