Report on Project 2

Zhunxuan Wang, 13300180086
School of Mathematical Sciences

November 19, 2017

1 Logistic Regression

1.1 Bayes' Rule

Based on the Bayes' theorem,

$$\Pr(y=1 \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid y=1)\Pr(y=1)}{p(\mathbf{x} \mid y=1)\Pr(y=1) + p(\mathbf{x} \mid y=0)\Pr(y=0)} = \frac{\alpha\, p(\mathbf{x} \mid y=1)}{\alpha\, p(\mathbf{x} \mid y=1) + (1-\alpha)\, p(\mathbf{x} \mid y=0)},$$

and all the components of $\mathbf{x}$ are conditionally independent given y, thus

$$\begin{aligned}\Pr(y=1 \mid \mathbf{x}) &= \frac{\alpha \prod_{i=1}^{D} p(x_i \mid y=1)}{\alpha \prod_{i=1}^{D} p(x_i \mid y=1) + (1-\alpha) \prod_{i=1}^{D} p(x_i \mid y=0)} \\ &= \frac{\alpha \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp -\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}}{\alpha \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp -\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2} + (1-\alpha) \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp -\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}}.\end{aligned} \tag{1}$$

Dividing the numerator and denominator by the numerator, we have

$$\begin{aligned}\Pr(y=1 \mid \mathbf{x}) &= \frac{1}{1 + \frac{1-\alpha}{\alpha} \exp\left[-\sum_{i=1}^{D} \frac{(x_i-\mu_{i0})^2 - (x_i-\mu_{i1})^2}{2\sigma_i^2}\right]} \\ &= \frac{1}{1 + \frac{1-\alpha}{\alpha} \exp\left[-\sum_{i=1}^{D} \frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2} \cdot x_i - \sum_{i=1}^{D} \frac{\mu_{i0}^2 - \mu_{i1}^2}{2\sigma_i^2}\right]}.\end{aligned} \tag{2}$$

Reforming the formula above, we have the form of a logistic function

$$\Pr(y=1 \mid \mathbf{x}) = \frac{1}{1 + \exp\left[-\sum_{i=1}^{D} \frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2} \cdot x_i - \left(\sum_{i=1}^{D} \frac{\mu_{i0}^2 - \mu_{i1}^2}{2\sigma_i^2} - \ln\frac{1-\alpha}{\alpha}\right)\right]}.$$

Therefore the weights $\mathbf{w} = (w_1, \ldots, w_D)^\intercal$ and the bias b will be subject to

$$\begin{cases} w_i = \dfrac{\mu_{i1}-\mu_{i0}}{\sigma_i^2} & (3) \\ b = \displaystyle\sum_{i=1}^{D} \frac{\mu_{i0}^2 - \mu_{i1}^2}{2\sigma_i^2} - \ln\frac{1-\alpha}{\alpha} & (4) \end{cases}$$

1.2 Maximum Likelihood Estimation

From the logistic function classifier we obtain

$$\Pr\left(y \mid \mathbf{x};\, \mathbf{w}, b\right) = \left(\Pr\left(y = 1 \mid \mathbf{x};\, \mathbf{w}, b\right)\right)^{y} \left(\Pr\left(y = 0 \mid \mathbf{x};\, \mathbf{w}, b\right)\right)^{1-y}.$$

Given $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}$, the negative log-likelihood of $y^{(1)}, y^{(2)}, \ldots, y^{(N)}$ is

$$E\left(\mathbf{w}, b\right) = -\log \Pr\left(y^{(1)}, y^{(2)}, \ldots, y^{(N)} \mid \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)};\, \mathbf{w}, b\right).$$

Due to the i.i.d. assumption,

$$\begin{aligned} E\left(\mathbf{w}, b\right) &= -\log \prod_{i=1}^{N} \Pr\left(y^{(i)} \mid \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)};\, \mathbf{w}, b\right) = -\log \prod_{i=1}^{N} \Pr\left(y^{(i)} \mid \mathbf{x}^{(i)};\, \mathbf{w}, b\right) \\ &= -\sum_{i=1}^{N} \left[y^{(i)} \log \Pr\left(y = 1 \mid \mathbf{x}^{(i)};\, \mathbf{w}, b\right) + \left(1 - y^{(i)}\right) \log \Pr\left(y = 0 \mid \mathbf{x}^{(i)};\, \mathbf{w}, b\right)\right] \\ &= \sum_{i=1}^{N} \left\{y^{(i)} \log\left[1 + \exp\left(-\mathbf{w} \cdot \mathbf{x}^{(i)} - b\right)\right] + \left(1 - y^{(i)}\right) \log\left[1 + \exp\left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right)\right]\right\}. \end{aligned} \tag{5}$$

And the derivatives of E are

$$\begin{cases} \dfrac{\partial E}{\partial \mathbf{w}} = \displaystyle\sum_{i=1}^{N} \left[y^{(i)} \dfrac{-\exp\left(-\mathbf{w} \cdot \mathbf{x}^{(i)} - b\right)}{1 + \exp\left(-\mathbf{w} \cdot \mathbf{x}^{(i)} - b\right)} + \left(1 - y^{(i)}\right) \dfrac{\exp\left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right)}{1 + \exp\left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right)}\right] \mathbf{x}^{(i)} & (6) \\ \dfrac{\partial E}{\partial b} = \displaystyle\sum_{i=1}^{N} \left[y^{(i)} \dfrac{-\exp\left(-\mathbf{w} \cdot \mathbf{x}^{(i)} - b\right)}{1 + \exp\left(-\mathbf{w} \cdot \mathbf{x}^{(i)} - b\right)} + \left(1 - y^{(i)}\right) \dfrac{\exp\left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right)}{1 + \exp\left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right)}\right] & (7) \end{cases}$$

1.3 L-2 Regularization

Assuming that $\mathbf{w}$ and b have the Gaussian prior distribution, the prior distribution $p\left(\mathbf{w}, b \mid \mathcal{D}\right)$ is

$$p\left(\mathbf{w}, b \mid \mathcal{D}\right) = \left[\prod_{i=1}^{D} \mathcal{N}\left(w_i \mid 0, \frac{1}{\lambda}\right)\right] \mathcal{N}\left(b \mid 0, \frac{1}{\lambda}\right).$$

Therefore the posterior distribution is

$$p\left(\mathbf{w}, \mathbf{b} \mid \left(y \mid \mathbf{x}\right)\right) \propto p\left(\mathbf{w}, b \mid \mathcal{D}\right) \Pr\left(y \mid x;\, \mathbf{w}, b\right)$$

and then the negative logarithm

$$\begin{aligned} L\left(\mathbf{w}, b\right) &= -\log p\left(\mathbf{w}, b \mid \mathcal{D}\right) \prod_{i=1}^{N} \Pr\left(y^{(i)} \mid \mathbf{x}^{(i)};\, \mathbf{w}, b\right) \\ &= -\log p\left(\mathbf{w}, b \mid \mathcal{D}\right) + E\left(\mathbf{w}, b\right) \\ &= E\left(\mathbf{w}, b\right) - \sum_{i=1}^{D} \log\left(\sqrt{\frac{\lambda}{2\pi}} \exp -\frac{\lambda}{2} w_i^2\right) - \log\left(\sqrt{\frac{\lambda}{2\pi}} \exp -\frac{\lambda}{2} b^2\right) \\ &= E\left(\mathbf{w}, b\right) + \frac{\lambda}{2} \sum_{i=1}^{D} w_i^2 + \frac{\lambda}{2} b^2 - \frac{D+1}{2} \log \frac{\lambda}{2\pi} \end{aligned} \tag{8}$$

where $C\left(\lambda\right) = -\dfrac{D+1}{2} \log \dfrac{\lambda}{2\pi}$. The derivatives of L are

$$\begin{cases} \dfrac{\partial L}{\partial \mathbf{w}} = \dfrac{\partial E}{\partial \mathbf{w}} + \lambda \mathbf{w} & (9) \\ \dfrac{\partial L}{\partial b} = \dfrac{\partial L}{\partial b} + \lambda b & (10) \end{cases}$$

2 Digit Classification

2.1 k-Nearest Neighbors

2.1.1 k-NN Classifier

Given a parameter k, for a incoming test point $\mathbf{x}$, find the k-nearest points in the training set. The most frequent class in the k points is the classifying result of the test point:

1. find the distance between the test point and each point in the training set

$$d_i = \rho\left(\mathbf{x}, \mathbf{x}_i\right), \; i = 1, 2, \ldots, N$$

   take euclidean distance for continuous data and discrete data (binarization preprocessing) respectively.

2. sort the indices of the training set by d_i in increasing order.

3. take the first k indices (the same as taking k minimums of d_i)

$$I_k = \{i_1, i_2, \ldots, i_k\}.$$

4. find the most frequent class in training label set $\mathbf{y}$ with indices I_k.

2.1.2 Model Performance

With the prediction model trained on `mnist_train`, tuning $k \in \{1, 3, 5, 7, 9\}$, we made predictions on the validation set and the test set (`mnist_valid` and `mnist_test` respectively). The accuracy plot of each set is shown as follows

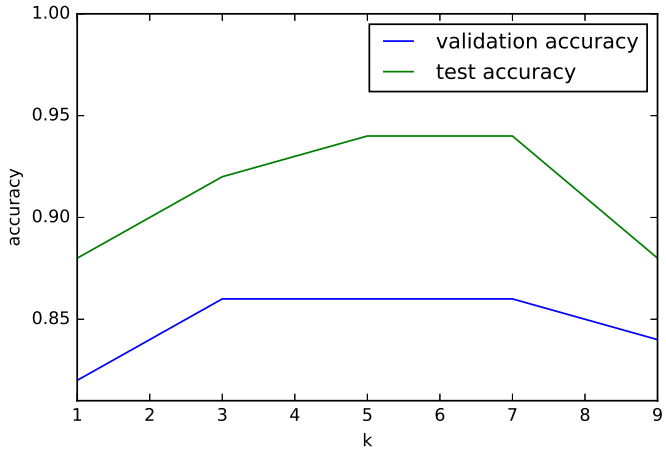

Figure 1: The Accuracy Plot against k

As we observe from the plot, the accuracies show incresing-then-decreasing trends on both validation set and test set as k grows. The highest accuracies in validation set and test set are 0.86 and 0.94 respectively, when $k \in \{3, 5, 7\}$ and $k \in \{5, 7\}$.

The optimal k I chose is $k^* = 5$, at which the validation accuracy and the test accuracy are 0.86 and 0.94 (both are peaks) respectively. The accuracies of $k^* - 2$ and $k^* + 2$ of the validation set are both 0.86, and that of the test set are 0.92 and 0.94. I chose $k^* = 5$ because the performances on $k^* - 2$ and $k^* + 2$ are both efficient on both sets, which makes k^* a relatively stable peak, and then it is more probable to perform more efficiently on other test sets.

The trends as k grows in each set are similar, but the performance on the test set are much better than the validation set, probably because the data in test set is more similar to the training set than the data in validation set (splitting training data to training set and validation set unevenly may cause this problem).

2.2 Logistic Regression

Based on the model deduced in section 1.2, we implemented the logistic regression on both `mnist_train` and `mnist_train_small`.

2.2.1 Hyperparameters Tuning

Before testing the model, we need to tune the hyperparameters based on the training set and the validation set, in order to get the optimal model. There are three key hyperparameters to tune

- Learning rate: I have tried multiple values in $\{0.001, 0.01, 0.1, 1, 10, 100\}$ as the learning rate and observed that when it came to 100, I got a `Nan` error, and when it came to 10, the oscillation of the cross entropy is too strong. And when it came too little, the convergence speed is relatively low, thus the chosen learning rate would be 1.
- Iteration times: as the learning rate being fixed at 1, the number iteration times when convergent is around 45 on `mnist_train` and 20 on `mnist_train_small`, thus I set the iteration times to 50 and 25 respectively.
- Weight initialization: I have tried multiple distributions (e.g. standard normal distribution, standard uniform distribution), and observed that the standard uniform distribution showed a better performance on accuracies, etc. Thus I chose the standard uniform distribution as the initial weight.

2.2.2 Model Performance

Fixing the hyperparameters and training the model, we have the final cross entropies and error rates as the follow tables show

	train	valid	test
CE	0.010785	0.045450	0.020982
Err	0	0.08	0.06

Table 1: Final Results on `mnist_train`

	train	valid	test
CE	0.001631	0.041588	0.017804
Err	0	0.32	0.24

Table 2: Final Results on `mnist_train_small`

As we observe from the tables above, the final CE and error rates showed efficient performances on all the three sets on `mnist_train`, and the test performance is better than the valid performance. However for `mnist_train_small`, the performance is limited by its small scale, and the test performance is also better than the valid performance.

The CE and accuracy plots against iteration times on both `mnist_train` and `mnist_train_small` are shown as follows

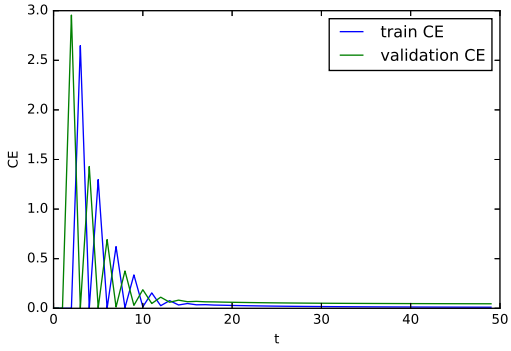

Figure 2: CE on `mnist_train`

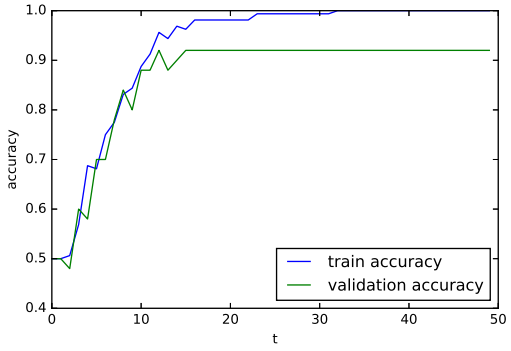

Figure 3: Accuracies on `mnist_train`

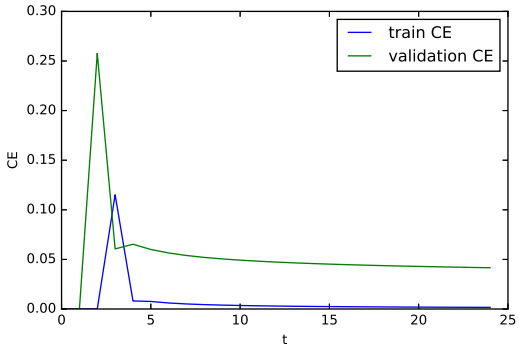

Figure 4: CE on `mnist_train_small`

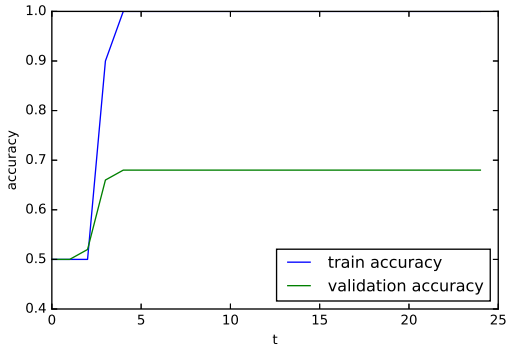

Figure 5: Accuracies on `mnist_train_small`

As we observe from the plots above, on `mnist_train`, the CE showed a state of strong oscillation and decreasing in early iterations, and then the state declined, gradually showed a convergence state. However the oscillation state on `mnist_train_small` is much weaker, and its convergence state came earlier. The performances on the both dateset showed a great convergence speed.

Running my codes several times, the results changed just a little, probably because the randomness of the initial weights, and the sensitivity of the model to initial weights which may be determined by hyperparameters.

If the results changed a lot, I would tuning the parameters, for instance,

- Choosing a distribution of small range or small variance which generating the initial weights randomly.
- Choosing a different model that is not as sensitive as before to the turbulance of initial weights.

2.3 Penalized Logistic Regression

Based on the model deduced in section 1.3, and abandoning the bias term and the constant in $L(\mathbf{w}, b)$, we implemented the penalized logistic regression on both `mnist_train` and `mnist_train_small`.

2.3.1 Model Performance

Tuning the $\lambda \in \{0.001, 0.01, 0.1, 1.0\}$, we have the average (through 10 runs) CE and errors plots against $\log \lambda$ as follows

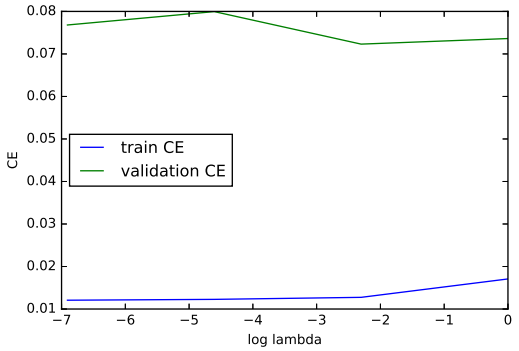

Figure 6: CE on `mnist_train`

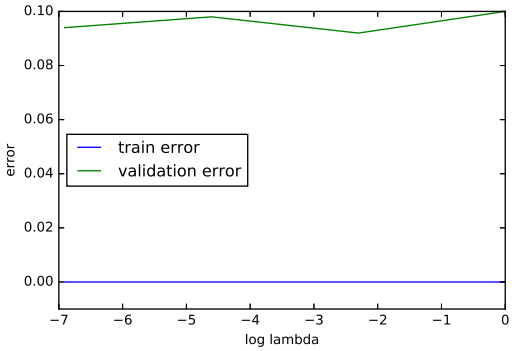

Figure 7: Errors on `mnist_train`

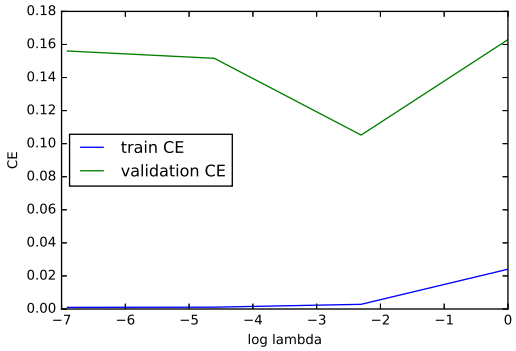

Figure 8: CE on `mnist_train_small`

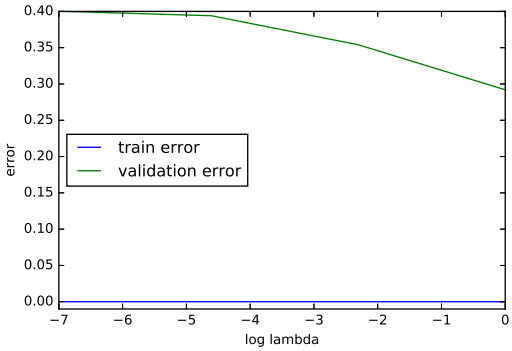

Figure 9: Errors on `mnist_train_small`

As we observe from the plots above, for `mnist_train`, the train CE showed an increasing tread though all λ, however the validation CE went up, and then down, and then up, of which the optimal λ with minimum validation CE is $\lambda = 0.1$, and the train error stayed at 0, and the validation error had the same trend as CE, of which the optimum is also $\lambda = 0.1$. The valid CE and the valid error had the same trend probably because of the positive correlation between CE and error rate (generally speaking).

For `mnist_train_small`, the train CE showed an increasing tread though all λ, however the validation CE went up, and then down, of which the optimal λ with minimum validation CE is $\lambda = 0.1$, and the train error stayed at 0, however the validation error had a decreasing trend, of which the optimum is $\lambda = 1$. The valid CE and the valid error had different status when $\lambda = 1$, of which overfitting may be one of the reasons.

In general, the performances between $\lambda = 0.001$ and $\lambda = 0.01$ are close, which is probably because they are both too little to affect the training process. And if the regularization weight is large, it may weaken the minimization process of the true loss function, which may cause CE greater. Thus we take $\lambda = 0.1$ as the regularization parameter.

The test performance when $\lambda = 0.1$ on `mnist_train` has a CE of 0.012254 and an error rate of 0.06. For `mnist_train_small`, the test performance has a CE of 0.050173 and an error rate of 0.24.

The test performance on `mnist_train` is a little better when introducing penalty, however that on `mnist_train_small` is almost the same, even in the contrary. The reason may be that penalties on small datasets can affect the loss minimization negatively, and, that the dataset is so small that it has a greater uncertainty on performances. On large datasets with less uncertainty, penalties can prevent overfitting, which may probably improve the model performance.

2.4 Naïve Bayes

2.4.1 Naïve Bayes Classifier

Assuming that $\mathbf{x}$ is the feature vector (with $M = 784$ features) of an image and label $c = 0, 1$ represents 2 classes respectively, the naïve bayes classifier can be written as

$$\begin{aligned} \Pr(c \mid \mathbf{x}) &= \frac{p(\mathbf{x} \mid c)\Pr(c)}{p(\mathbf{x})} = \frac{p(\mathbf{x} \mid c)\Pr(c)}{p(\mathbf{x} \mid c=0)\Pr(c=0) + p(\mathbf{x} \mid c=1)\Pr(c=1)} \\ &\propto p(\mathbf{x} \mid c)\Pr(c) = \Pr(c)\prod_{i=1}^{M} p(x_i \mid c) \end{aligned} \tag{11}$$

where prior $\Pr(c)$ and $p(x_i \mid c)$ will be obtained by frequency calculation and ML on the training dataset.

Based on the maximum a posteriori estimation (in spite of the parameters in $p(x_i \mid c)$), the classifier can be written as

$$c_{\mathrm{o}}(\mathbf{x}) = \arg\max_{c} \Pr(c)\prod_{i=1}^{M} p(x_i \mid c),\ c = 0, 1.$$

For the ML of $p(x_i \mid c)$, the Gaussian distribution has a ML of

$$\hat{\mu} = \frac{\sum X_i}{N},\ \hat{\sigma}^2 = \frac{\sum (X_i - \hat{\mu})^2}{N},$$

hence

$$\hat{\mu}_{jc} = \frac{\sum_{i=1}^{N} X_{ij} \mathbb{1}_{\{c\}}(c_i)}{\sum_{i=1}^{N} \mathbb{1}_{\{c\}}(c_i)},\ \hat{\sigma}^2_{jc} = \frac{\sum_{i=1}^{N} (X_i - \hat{\mu}_{jc})^2 \mathbb{1}_{\{c\}}(c_i)}{\sum_{i=1}^{N} \mathbb{1}_{\{c\}}(c_i)},$$

where $\mathbb{1}_{\{c\}}(c_i)$ is the indicator function.

2.4.2 Model Performance

After fitting the naïve Bayes model, the predictions has an accuracy of 0.8625 on training set and 0.8 on testing set. The image of the mean and variance vector $\boldsymbol{\mu}_c$ and $\boldsymbol{\sigma}_c^2$ for both models are shown as follows

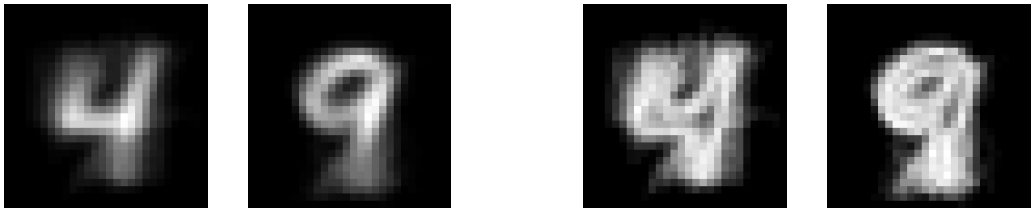

Figure 10: The Mean Images of Both Classes

Figure 11: The Variance Images of Both Classes

As we observe from the images above, the 2 classes are digit 4 and digit 9. In the mean images, the internal parts of the drawing lines of the digits are lighter than the contour lines, however the variance images is quite the opposite that the contour lines are much lighter. In other words, the mean image and the variance image are complementary in each class.

Based on the observation, we can infer that the internal parts of the drawing lines are lighter in mean images because the same digits are roughly in the same shapes, however the exact shapes are unstable, therefore the variances on the edges of the shapes are relatively greater.

2.5 Comparison

We have implemented k-NN, logistic regression (with or without penalty) and naïve Bayes model on `MNIST` data, and they have their own pros and cons

- k-NN [2] does not actually have a training process, the model parameters are the training data itself, which costs plenty of storage. However its performance is efficient.
- Logistic regression [1] has a training process which fit large dataset to model parameters, and it has many mutant formation (e.g. with penalty, with other loss functions). In this project, the loss function is negative log likelihood function.
- Naïve Bayes [3] is a classic statistic model, which fit the dataset to required probabilities or probability densities based on frequency.

The testing accuracies are shown as the following table below

	k-NN	LR	LR (with P)	NB
test	0.94	0.94	0.94	0.8

From the table we can conclude that k-NN, logistic regression (with or without penalty) has a efficient performance on `MNIST`, of which the testing accuracies are all 0.94. The naïve Bayes model is much less efficient on testing performance, only 0.8.

3 Stochastic Subgradient Methods

Considering the minimization of the SVM objective function

$$\min_{\mathbf{w}} f(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - y_i \mathbf{w}^\intercal \mathbf{x}_i\}$$

of whose $f(\mathbf{w})$ the sub-gradient is

$$\lambda\mathbf{w} + \begin{cases} -y_i\mathbf{x}_i, & \text{if } 1 - y_i(\mathbf{w}^\intercal\mathbf{x}) > 0 \\ 0, & \text{otherwise} \end{cases} \in \partial f(\mathbf{w}),$$

thus the stochastic subgradient method would be

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha_t \left(\lambda\mathbf{w}^t + \begin{cases} -y_{i_t}\mathbf{x}_{i_t}, & \text{if } 1 - y_{i_t}\left(\mathbf{w}^{t\intercal}\mathbf{x}_{i_t}\right) > 0 \\ 0, & \text{otherwise} \end{cases} \right)$$

3.1 Polyak-Ruppert Averaging

Define the running average $\bar{\mathbf{w}}^t = \sum_{i=0}^{t-1} \mathbf{w}^i$, and replace the current $\mathbf{w}^t$ by $\bar{\mathbf{w}}^t$ in each performance testing step [4]. The plot of the value of objective function against performance testing steps generated by `svmAvg` is shown as follows

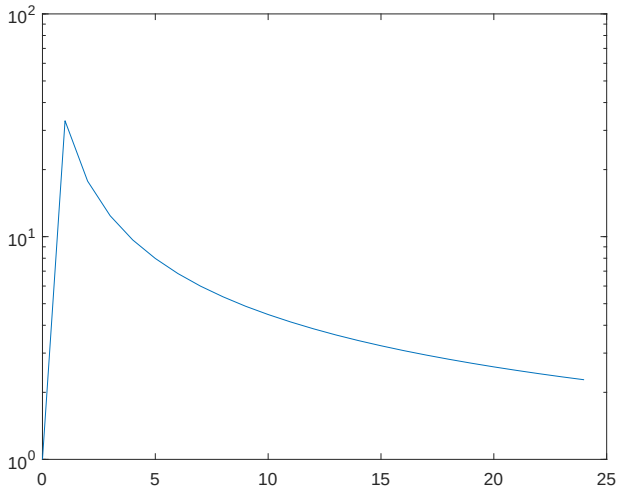

Figure 12: The Function Value Plot

As we observe from the plot, although averaging all the iterations smoothed the performance, the objective function was still increased compared to its initial value.

The modified part of the code is in appendix A.

3.2 Second-Half Averaging

Define the second-half running average $\bar{\mathbf{w}}^t = \sum_{i=\lceil \frac{maxIter}{2} \rceil}^{t-1} \mathbf{w}^i$, where $t > \lceil \frac{maxIter}{2} \rceil$. Remaining the first half of $\mathbf{w}^t$ where $t \leq \lceil \frac{maxIter}{2} \rceil$ and replacing the other half to $\bar{\mathbf{w}}^t = \sum_{i=\lceil \frac{maxIter}{2} \rceil}^{t-1} \mathbf{w}^i$, we have the plot of the value of objective function against performance testing steps generated by modified `svmAvg`

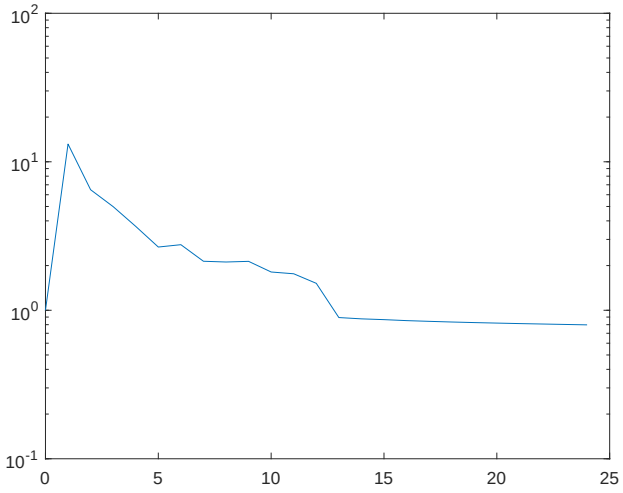

Figure 13: The Function Value Plot

As we observe from the plot, averaging the iterations of the second half smoothed the performance of the second half, the objective function value decreased to around 0.8, in a convergence state.

The modified part of the code is in appendix A.

3.3 Additional Optimization

I applied the modified SAG (Stochastic Average Gradient) method [5]

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \frac{\alpha_t}{n} \sum_{i=1}^{n} \mathbf{g}_i^{t+1},\ \mathbf{g}_i^{t+1} = \begin{cases} \nabla f_i\left(\mathbf{w}^t\right), & i = i_t \\ \mathbf{g}_i^t, & i \neq i_t \end{cases}$$

with another initial condition $\mathbf{g}_i^0 = \nabla f_i\left(\mathbf{w}^0\right)$. The main idea of this algorithm is to store the previously calculated gradients, along with the newly calculated stochastic gradient, taking the average of these gradients as the final gradient of the current step. The optimization plot is shown as follows

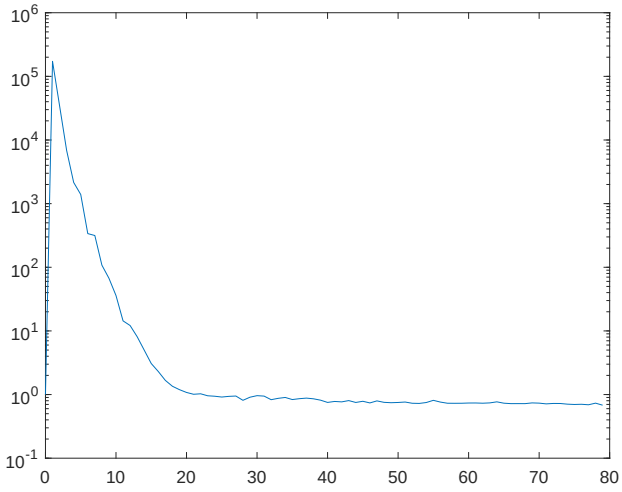

Figure 14: The Function Value Plot (modified SAG)

As we observe from the plot, in the early period of optimizing the function value reached beyond 10^5. It has a fast down trend before the 20-th iteration, and a slow down trend after. It has a faster convergence speed than averaging and a slower convergence speed than second-half averaging. However, the objective function value decreased to around 0.7 in a convergence state, which is better than the previous methods.

The function `svmSAG` is in appendix A.

References

[1] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[2] James M Keller, Michael R Gray, and James A Givens. A fuzzy k-nearest neighbor algorithm. *IEEE transactions on systems, man, and cybernetics*, (4):580–585, 1985.

[3] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Madison, WI, 1998.

[4] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

[5] Nicolas L Roux, Mark Schmidt, and Francis R Bach. A stochastic gradient method with an exponential convergence _rate for finite training sets. In *Advances in Neural Information Processing Systems*, pages 2663–2671, 2012.

A MATLAB Codes for Stochastic Subgradient Methods

Function `svmAvg` for averaging

```
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);
w_set(:, 1) = w;

% Apply stochastic gradient method
for t = 1 : maxIter
  if mod(t - 1, n) == 0
    % Plot our progress
    % (turn this off for speed)
    w_mean = mean(w_set, 2);
    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
  [~, sg] = hingeLossSubGrad(w, Xt, y, lambda, i);

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

  % Take stochastic subgradient step
  w = w - alpha * (sg + lambda * w);
  w_set(:, t + 1) = w;
end

model.w = mean(w_set, 2);
model.predict = @predict;

end
```

Function `svmAvg` for second-half averaging

```
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);
w_set = []
% Apply stochastic gradient method
for t = 1 : maxIter

    if t >= maxIter / 2
        w_set = [w_set, w];
    end

    if mod(t - 1, n) == 0
        % Plot our progress
        % (turn this off for speed)
        if t >= maxIter / 2
            w_ = mean(w_set, 2);
        else
            w_ = w
        end
        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
    [~, sg] = hingeLossSubGrad(w, Xt, y, lambda, i);

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

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

end

model.w = mean(w_set, 2);
model.predict = @predict;

end
```

Function `svmSAG` for modified SAG

```
function [model] = svmSAG(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);
gradient = zeros(d + 1, n);

for i = 1 : n
  [~, gradient(:, i)] = hingeLossSubGrad(w, Xt, y, lambda, i);
end

sum_gradient = sum(gradient, 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))) + (
        lambda / 2) * (w' * w);
    semilogy([0 : t / n], objValues);
    pause(.1);
  end

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

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

  sum_gradient = sum_gradient - gradient(:, i) + sg + lambda * w;
  gradient(:, i) = sg + lambda * w;

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

  % Take stochastic subgradient step
  w = w - (alpha / n) * sum_gradient;

end

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

end
```