CSE 250A. Assignment 5

Hao-en Sung (A53204772)
wrangle1005@gmail.com

November 1, 2016

5.1 Gradient-based learning

(a) Show that the gradient of the conditional log-likelihood $L = \sum_t^T \log P(y_t|\vec{x}_t$ is given by

$$\frac{\partial L}{\partial w_i} = \sum_{t=1}^{T} \left[\frac{y_t - p_t}{p_t(1-p_t)} \right] g'(\vec{w} \cdot \vec{x}_t) x_{it},$$

where $g'(z)$ denotes the first derivative of $g(z)$. Intuitively, this result shows that the differences between observed values y_t and predictions p_t appear as error signals for learning.

Sol. I can rewrite L as follows.

$$\begin{aligned} L &= \sum_t^T \log P(y_t|\vec{x}_t) \\ &= \sum_t^T \log \left[(p_t)^{y_t} \cdot (1-p_t)^{1-y_t} \right] \\ &= \sum_t^T \left[y_t \log p_t + (1-y_t) \log(1-p_t) \right] \end{aligned}$$

Then, I have

$$\begin{aligned} \frac{\partial L}{\partial w_i} &= \sum_t^T \left[y_t \cdot \frac{1}{p_t} \cdot g'(\vec{w} \cdot \vec{x}) \cdot x_{it} + (1-y_t) \cdot \frac{-1}{-p_t} \cdot g'(\vec{w} \cdot \vec{x}) \cdot x_{it} \right] \\ &= \sum_t^T \frac{(1-p_t) + (y_t - 1)}{p_t \cdot (1-p_t)} \cdot g'(\vec{w} \cdot \vec{x}) \cdot x_{it} \\ &= \sum_t^T \frac{y_t - p_t}{p_t \cdot (1-p_t)} \cdot g'(\vec{w} \cdot \vec{x}) \cdot x_{it}. \end{aligned}$$

□

(b) Show that the general result in part (a) reduces to the result in lecture when $g(z) = [1 + e^{-z}]^{-1}$ is the sigmoid function. Again, this result shows that the differences between observed values $y_i t$ and predictions $p_i t$ appear as error signals for learning.

Sol. If I use $g_z = \frac{1}{1+e^{-z}}$ to express p_t and $g(\vec{w} \cdot \vec{x})$, and use $g'_z = g_z \cdot (1 - g_z)$, I have

$$\begin{aligned} \frac{\partial L}{\partial w_i} &= \sum_t^T \frac{y_t - g(\vec{w} \cdot \vec{x})}{g(\vec{w} \cdot \vec{x}) \cdot (1 - g(\vec{w} \cdot \vec{x}))} \cdot g(\vec{w} \cdot \vec{x}) \cdot (1 - g(\vec{w} \cdot \vec{x})) \cdot x_{it} \\ &= \sum_t^T (y_t - g(\vec{w} \cdot \vec{x})) \cdot x_{it}. \end{aligned}$$

□

5.2 Multinomial logistic regression

Show that the gradient of the conditional log-likelihood $L = \sum_t \log P(y_t|\vec{x}_t)$ is given by:

$$\frac{\partial L}{\partial \vec{w}_i} = \sum_{t=1}^{T} (y_{it} - p_{it}) \vec{x}_t.$$

Sol. I can write L as follows.

$$\begin{aligned}
L &= \sum_t^T \log P(y_t|\vec{x}_t) \\
&= \sum_t^T \log \left[\prod_{l=1}^k (p_{lt})^{y_{lt}}\right] \\
&= \sum_t^T \sum_{l=1}^k y_{lt} \log p_{lt} \\
&= \sum_t^T \sum_{l=1}^k \left[y_{lt} \log(e^{\vec{w}_l \cdot \vec{x}_t}) - y_{lt} \log\left(\sum_{j=1}^k e^{\vec{w}_j \cdot \vec{x}_t}\right)\right] \\
&= \sum_t^T \left[\sum_{l=1}^k y_{lt} \log(e^{\vec{w}_l \cdot \vec{x}_t}) - \log\left(\sum_{j=1}^k e^{\vec{w}_j \cdot \vec{x}_t}\right) \sum_{l=1}^k y_{lt}\right] \\
&= \sum_t^T \left[\sum_{l=1}^k y_{lt} \log(e^{\vec{w}_l \cdot \vec{x}_t}) - \log\left(\sum_{j=1}^k e^{\vec{w}_j \cdot \vec{x}_t}\right)\right]
\end{aligned}$$

Then, I have

$$\begin{aligned}
\frac{\partial L}{\partial \vec{w}_i} &= \sum_t^T \frac{y_{it}}{e^{\vec{w}_l \cdot \vec{x}_t}} \cdot e^{\vec{w}_l \cdot \vec{x}_t} \cdot \vec{x}_t - \frac{e^{\vec{w}_l \cdot \vec{x}_t}}{\sum_{j=1}^k e^{\vec{w}_l \cdot \vec{x}_t}} \cdot \vec{x}_t \\
&= \sum_t^T (y_{it} - p_{it}) \cdot \vec{x}_t.
\end{aligned}$$

□

5.3 Convergence of gradient descent

(a) Consider minimizing the function $g(x) = \frac{\alpha}{2}(x - x_*)^2$ by gradient descent, where $\alpha > 0$. Derive an expression for the error $\epsilon_n = x_n - x_*$ at the n^{th} iteration in terms of the initial error ϵ_0 and the step size $\eta > 0$.

Sol. From $g(x)$ I know that $\frac{\partial g(x)}{\partial x} = \alpha \cdot (x - x_*)$. Then I can apply it with general gradient framework and list several updates of x as follows.

$$\begin{aligned}
x_1 &= x_0 - \eta \cdot \alpha(x_0 - x_*) \\
&= (1 - \eta \cdot \alpha) \cdot x_0 + \eta \cdot \alpha \cdot x_* \\
x_2 &= x_1 - \eta \cdot \alpha(x_1 - x_*) \\
&= (1 - \eta \cdot \alpha) \cdot x_1 + \eta \cdot \alpha \cdot x_* \\
&\vdots
\end{aligned}$$

One can observe the pattern of it, consider $\epsilon_0 = x_0 - x_*$, and derive

$$\begin{aligned}
x_n &= (1 - \eta \cdot \alpha)^n \cdot x_0 + \eta \cdot \alpha \cdot x_* \cdot \sum_{i=0}^{n-1} (1 - \eta \cdot \alpha)^i \\
&= (1 - \eta \cdot \alpha)^n \cdot (\epsilon_0 + x_*) + \eta \cdot \alpha \cdot x_* \cdot \frac{(1 - \eta \cdot \alpha)^n - 1}{-\eta \cdot \alpha} \\
&= (1 - \eta \cdot \alpha)^n \cdot (\epsilon_0 + x_*) - x_* \cdot [(1 - \eta \cdot \alpha)^n - 1] \\
&= (1 - \eta \cdot \alpha)^n \cdot \epsilon_0 + x_* \\
\epsilon_n = x_n - x_* &= (1 - \eta \cdot \alpha)^n \cdot \epsilon_0.
\end{aligned}$$

□

(b) For what values of the step size η does the update rule converge to the minimum at x_*? What step size leads to the fastest convergence, and how is it related to $g''(x_n)$?

Sol. If it converges, after n iterations ϵ_n needs to be 0. Then I have

$$\begin{aligned}
0 = \epsilon_n &= (1 - \eta\alpha)^n \cdot \epsilon_0 \\
\eta \cdot \alpha &= 1 \\
\eta &= \frac{1}{\alpha}.
\end{aligned}$$

For this problem, designing step size $\eta = \frac{1}{\alpha}$ has the fastest convergence — $g(x)$ will converge at $n = 1$. It is also noticeable that

$$\eta = \frac{1}{\alpha} = \frac{1}{g''(x_n)}.$$

□

(c) Consider minimizing the quadratic function in part (a) by gradient descent with a momentum term. Again, let $\epsilon_n = x_n - x_*$ denote the error at the nth iteration. Show that the error in this case satisfies the recursion relation:

$$\epsilon_{n+1} = (1 - \alpha\eta + \beta)\epsilon_n - \beta\epsilon_{n-1}.$$

Sol. One can prove it by writing down the update equations and simply them.

$$\begin{aligned}
x_{n+1} &= x_n - \eta \cdot \alpha \cdot (x_n - x_*) + \beta \cdot (x_n - x_{n-1}) \\
x_{n+1} &= (1 - \eta \cdot \alpha + \beta) \cdot x_n + \eta \cdot \alpha \cdot x_* - \beta \cdot x_{n-1} \\
\epsilon_{n+1} + x_* &= (1 - \eta \cdot \alpha + \beta) \cdot (\epsilon_n + x_*) + \eta \cdot \alpha \cdot x_* - \beta \cdot (\epsilon_{n-1} + x_*) \\
\epsilon_{n+1} &= (1 - \eta \cdot \alpha + \beta) \cdot \epsilon_1 - \beta \cdot \epsilon_{n-1}
\end{aligned}$$

□

(d) Suppose that the second derivative $g''(x^*)$ is given by $\alpha = 1$, the learning rate by $\eta = \frac{4}{9}$, and the momentum parameter by $\beta = \frac{1}{9}$. Show that one solution to the recursion in part (c) is given by:

$$\epsilon_n = \gamma^n \epsilon_0,$$

where ϵ_0 is the initial error and γ is a numerical constant to be determined. (Other solutions are also possible, depending on the way that the momentum term is defined at time $t = 0$; do not concern yourself with this.) How does this rate of convergence compare to that of gradient descent with the same learning rate ($\eta = \frac{4}{9}$) but no momentum parameter ($\beta = 0$)?

Sol. By setting $\epsilon_1 = \epsilon_0$, I can write down some of the values of them as follows.

$$\begin{aligned}
\epsilon_2 &= \frac{2}{3}\epsilon_1 - \frac{1}{9}\epsilon_0 = \frac{5}{9}\epsilon_0 \\
\epsilon_3 &= \frac{2}{3}\epsilon_2 - \frac{1}{9}\epsilon_1 = \frac{7}{27}\epsilon_0 \\
\epsilon_4 &= \frac{2}{3}\epsilon_3 - \frac{1}{9}\epsilon_2 = \frac{9}{81}\epsilon_0
\end{aligned}$$

One can easily find out the pattern that $\epsilon_n = \frac{2n+1}{3^n}\epsilon_0$. In this setting, $\gamma = \frac{(2n+1)^{\frac{1}{n}}}{3}$.

If one sets $\beta = 0$, then one has $\epsilon_n = \left(\frac{2}{3}\right)^n$, whose convergence rate is much worse than the previous setting ($\beta = \frac{1}{9}$). □

5.4 Newton's method

(a) Consider the polynomial function $g(x) = (x - x_*)^{2p}$ for positive integers p, whose minimum occurs at $x = x_*$. Suppose that Newton's method is used to minimize this function, starting from some initial estimate x_0. Derive an expression for the error $\epsilon_n = |x_n - x_*|$ at the nth iteration in terms of the initial error ϵ_0.

Sol. After derivation, I have

$$\begin{aligned}
\frac{\partial g(x)}{\partial x} &= 2p \cdot (x - x_*)^{2p-1} \\
\frac{\partial^2 g(x)}{\partial x^2} &= 2p \cdot (2p-1) \cdot (x - x_*)^{2p-2},
\end{aligned}$$

and $x_{n+1} = x_n - \frac{1}{2p-1} \cdot (x_n - x_*)$.

After applying it to Newton's method and regarding $x_n = x_* \pm \epsilon_n$, I can have the general formula for x_n as follows.

$$\begin{aligned}
x_n &= \left(1 - \frac{1}{2p-1}\right)^n \cdot x_0 + \frac{1}{2p-1} \cdot x_* \cdot \sum_{i=0}^{n-1} \left(1 - \frac{1}{2p-1}\right)^i \\
&= \left(1 - \frac{1}{2p-1}\right)^n \cdot x_0 + \frac{1}{2p-1} \cdot x_* \cdot \frac{\left(1 - \frac{1}{2p-1}\right)^n - 1}{-\frac{1}{2p-1}} \\
x_* \pm \epsilon_n &= \left(1 - \frac{1}{2p-1}\right)^n \cdot (x_* \pm \epsilon_0) - x_* \cdot \left[\left(1 - \frac{1}{2p-1}\right)^n - 1\right] \\
\epsilon_n &= \pm\left(1 - \frac{1}{2p-1}\right)^n \cdot \epsilon_0 = \left(1 - \frac{1}{2p-1}\right)^n \cdot \epsilon_0 \qquad (\epsilon_n \text{ is always positive})
\end{aligned}$$

□

(b) For the function in part (a), how many iterations of Newton's method are required to reduce the initial error by a constant factor $\delta < 1$, such that $\epsilon_n \leq \delta\epsilon_0$? Starting from your previous answer, show that $n \geq (2p-1)\log(1/\delta)$ iterations are sufficient. (Hint: use the inequality that $\log z \leq z-1$ for $z > 0$.)

Sol. From $\epsilon_n = \left(1 - \frac{1}{2p-1}\right)^n \cdot \epsilon_0$ and $\epsilon_n \leq \delta \cdot \epsilon_0$, it can be derived that

$$\begin{aligned}
\left(1 - \frac{1}{2p-1}\right)^n &\leq \delta \\
n \log \left(1 - \frac{1}{2p-1}\right)^n &\leq \log \delta \\
n \log \frac{2p-1}{2p-2} &\geq \log \frac{1}{\delta} \\
n \cdot \left(\frac{2p-1}{2p-2} - 1\right) &\geq \log \frac{1}{\delta} \\
n \cdot \frac{1}{2p-2} &\geq \log \frac{1}{\delta} \\
n &\geq (2p-2)\log \frac{1}{\delta}
\end{aligned}$$

It is interesting to notice that the fact "$n \geq (2p-2)\log \frac{1}{\delta}$ iterations are sufficient" directly implies "$n \geq (2p-1)\log \frac{1}{\delta}$ iterations are sufficient". Thus, it is proved. □

(c) Consider the function $g(x) = x_* \log(x_*/x) - x_* + x$, where $x_* > 0$. Show that the minimum occurs at $x = x_*$, and sketch the function in the region $|x - x_*| < x_*$.

Sol. One can derive its first-order derivative as

$$\begin{aligned}
\frac{\partial g}{\partial x} &= x_* \cdot \frac{x}{x_*} \cdot \frac{-x_*}{x^2} + 1 \\
&= -\frac{x*}{x} + 1.
\end{aligned}$$

When minimum occurs, $\frac{\partial g}{\partial x} = -\frac{x*}{x} + 1 = 0$. Then I have $x = x_*$.

To plot the formula $g(x) = x_* \log(x_*/x) - x_* + x$, I take $x_* = 1$ as an example and have $g(x) = \log(1/x) - 1 + x, 0 < x < 2$. I later use *Wolfram Alpha* to draw the figure for me, as shown in Figure 1.

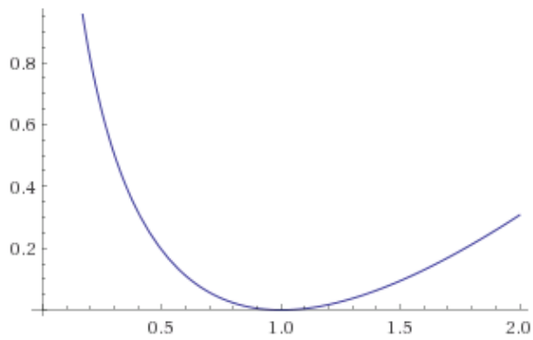

Figure 1: $g(x) = \log(1/x) - 1 + x, 0 < x < 2$, when $0 < x < 2, x_* = 1$

□

(d) Consider minimizing the function in part (c) by Newton's method. Derive an expression for the relative error $r_n = (x_n - x_*)/x_*$ at the nth iteration in terms of the initial relative error r_0. Note the rapid convergence (which is typical of Newton's method). For what range of initial values (for x_0) does Newton's method converge to the correct answer?

Sol. One can derive its second-order derivative as

$$\frac{\partial^2 g}{\partial x^2} = \frac{x*}{x^2}.$$

Then, the update formula becomes

$$\begin{aligned}
x_{n+1} &= x_n - \frac{-\frac{x_*}{x_n} + 1}{\frac{x^*}{x_n^2}} \\
&= x_n - (-x_n + \frac{x_n^2}{x_*}) \\
&= 2x_n - \frac{x_n^2}{x_*} \\
&= -\frac{1}{x_*}(x_n - x^*)^2 + x_*.
\end{aligned}$$

Replace this formula into r_n, I have

$$\begin{aligned} r_n &= \frac{x_{n-1} - x_*}{x_*} \\ &= -\frac{1}{(x^*)^2}(x_{n-1} - x^*)^2 \\ &= -\left(\frac{x_{n-1} - x^*}{x^*}\right)^2 \\ &= -(r_{n-1})^2 \\ &= -(r_0)^{2^n}. \end{aligned}$$

To make relative error closed to 0 when n gets larger enough, the constraint is that

$$\begin{aligned} -1 < r_0 < 1 \\ -1 < \frac{x_0 - x_*}{x_*} < 1 \\ 0 < x_0 < 2 \cdot x_*. \end{aligned}$$

□

5.5 Stock market prediction

(a) Linear coefficients

Sol. Assume data is given as zero-based and there are overall n time points, the log-likelihood can be written as

$$\begin{aligned} L &= \sum_{t=4}^{n-1} \log\left\{\frac{1}{\sqrt{2\pi}} \cdot \exp\left[-\frac{1}{2}\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right)^2\right]\right\} \\ &= \sum_{t=4}^{n-1}\left[\log\frac{1}{\sqrt{2\pi}} - \frac{1}{2}\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right)^2\right]. \end{aligned}$$

I then can find out the derivatives of α_1, α_2, α_3, α_4 as follows.

$$\begin{aligned} \frac{\partial L}{\partial \alpha_1} &= \sum_{t=4}^{n-1}\left[\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right) \cdot x_{t-1}\right] \\ &= \sum_{t=4}^{n-1} x_t \cdot x_{t-1} - \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-1} - \alpha_2 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-2} - \alpha_3 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-3} - \alpha_4 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-4} \\ \frac{\partial L}{\partial \alpha_2} &= \sum_{t=4}^{n-1}\left[\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right) \cdot x_{t-2}\right] \\ &= \sum_{t=4}^{n-1} x_t \cdot x_{t-2} - \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-2} - \alpha_2 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-2} - \alpha_3 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-3} - \alpha_4 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-4} \\ \frac{\partial L}{\partial \alpha_3} &= \sum_{t=4}^{n-1}\left[\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right) \cdot x_{t-3}\right] \\ &= \sum_{t=4}^{n-1} x_t \cdot x_{t-3} - \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-3} - \alpha_2 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-3} - \alpha_3 \sum_{t=4}^{n-1} x_{t-3} \cdot x_{t-3} - \alpha_4 \sum_{t=4}^{n-1} x_{t-3} \cdot x_{t-4} \\ \frac{\partial L}{\partial \alpha_4} &= \sum_{t=4}^{n-1}\left[\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right) \cdot x_{t-4}\right] \\ &= \sum_{t=4}^{n-1} x_t \cdot x_{t-4} - \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-4} - \alpha_2 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-4} - \alpha_3 \sum_{t=4}^{n-1} x_{t-3} \cdot x_{t-4} - \alpha_4 \sum_{t=4}^{n-1} x_{t-4} \cdot x_{t-4} \end{aligned}$$

When at optimal, it is clear that $\frac{\partial L}{\partial \alpha_1} = \frac{\partial L}{\partial \alpha_2} = \frac{\partial L}{\partial \alpha_3} = \frac{\partial L}{\partial \alpha_4} = 0$. Then, I can rewrite above equations as follows.

$$\begin{aligned} \sum_{t=4}^{n-1} x_t \cdot x_{t-1} &= \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-1} + \alpha_2 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-2} + \alpha_3 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-3} + \alpha_4 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-4} \\ \sum_{t=4}^{n-1} x_t \cdot x_{t-2} &= \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-2} + \alpha_2 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-2} + \alpha_3 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-3} + \alpha_4 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-4} \\ \sum_{t=4}^{n-1} x_t \cdot x_{t-3} &= \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-3} + \alpha_2 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-3} + \alpha_3 \sum_{t=4}^{n-1} x_{t-3} \cdot x_{t-3} + \alpha_4 \sum_{t=4}^{n-1} x_{t-3} \cdot x_{t-4} \\ \sum_{t=4}^{n-1} x_t \cdot x_{t-4} &= \alpha_1 \sum_{t=4}^{n-1} x_{t-1} \cdot x_{t-4} + \alpha_2 \sum_{t=4}^{n-1} x_{t-2} \cdot x_{t-4} + \alpha_3 \sum_{t=4}^{n-1} x_{t-3} \cdot x_{t-4} + \alpha_4 \sum_{t=4}^{n-1} x_{t-4} \cdot x_{t-4} \end{aligned}$$

Since training data is given, and every value of x_t are known, I can regard the elements on the left-side of equal sign as vector b, coefficients of α on the right-side of equal sign as matrix A, and $[\alpha_1, \alpha_2, \alpha_3, \alpha_4]$ as vector x. By doing so, I am able to find out α_1, α_2, α_3, and α_4 by solving

$$A \cdot x = b \Rightarrow x = A^{-1} \cdot b.$$

After solving it through 1, I have

$$\begin{aligned}\alpha_1 &= +0.945200\\ \alpha_2 &= +0.019742\\ \alpha_3 &= -0.013645\\ \alpha_4 &= +0.046781\end{aligned}$$

□

(b) Mean squared prediction error

Sol. To infer the best x_t, I can find out the optimal x_t by solving

$$\begin{aligned}0 &= \frac{\partial P(x_t|x_{t-1}, x_{t-2}, x_{t-3}, x_{t-4})}{\partial x_t}\\ &= \frac{1}{\sqrt{2\pi}} \cdot \exp\left[-\frac{1}{2}\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right)^2\right]\\ &\qquad \cdot \left[-\left(x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\right)\right]\\ &= x_t - \alpha_1 x_{t-1} - \alpha_2 x_{t-2} - \alpha_3 x_{t-3} - \alpha_4 x_{t-4}\\ x_t &= \alpha_1 x_{t-1} + \alpha_2 x_{t-2} - \alpha_3 x_{t-3} + \alpha_4 x_{t-4}.\end{aligned}$$

The error for training and testing data evaluated by MSE are listed as in Table 1

Train	13918.632921
Test	3018.267841

Table 1: MSE evaluation for Training and Testing data

□

(c) Source code

Sol. Code is appended as 1 in 5.6. □

5.6 Handwritten digit classification

(a) Training

Sol. I can write down the logistic regression L with $sigma(z) = \frac{1}{1+\exp(-z)}$as follows.

$$\begin{aligned}L &= \sum_{t=1}^{T} \log P(Y = y_t|\vec{X} = \vec{x}_t)\\ &= \sum_{t=1}^{T} \log\left[\sigma(\vec{w} \cdot \vec{x}_t)^{y_t} + \sigma(-\vec{w} \cdot \vec{x}_t)^{1-y_t}\right]\\ &= \sum_{t=1}^{T} \left[y_t \log \sigma(\vec{w} \cdot \vec{x}_t) + (1 - y_t) \log \sigma(-\vec{w} \cdot \vec{x}_t)\right].\end{aligned}$$

One can derive its first-order derivative $\frac{\partial L}{\partial w_i}$ as

$$\begin{aligned}\frac{\partial L}{\partial w_i} &= \sum_{t=1}^{T}\left[y_t \cdot \frac{1}{\sigma(\vec{w} \cdot \vec{x}_t)} \cdot \sigma(\vec{w} \cdot \vec{x}_t) \cdot \sigma(-\vec{w} \cdot \vec{x}_t) \cdot x_{it}\right.\\ &\qquad \left.+(1 - y_t)\frac{1}{\sigma(-\vec{w} \cdot \vec{x}_t)} \cdot \sigma(-\vec{w} \cdot \vec{x}_t) \cdot \sigma(\vec{w} \cdot \vec{x}_t) \cdot -x_{it}\right]\\ &= \sum_{t=1}^{T} \left[y_t \cdot (1 - \sigma(\vec{w} \cdot \vec{x}_t)) \cdot x_{it} - (1 - y_t) \cdot \sigma(\vec{w} \cdot \vec{x}_t) \cdot x_{it}\right]\\ &= \sum_{t=1}^{T} \left[(y_t - \sigma(\vec{w} \cdot \vec{x}_t)) \cdot x_{it}\right],\end{aligned}$$

and its second-order derivative $\frac{\partial^2 L}{\partial w_i w_j}$ as

$$\frac{\partial L}{\partial w_i w_j} = \sum_{t=1}^{T} \left[(-\sigma(\vec{w} \cdot \vec{x}_t) \cdot \sigma(-\vec{w} \cdot \vec{x}_t)) \cdot x_{it} \cdot x_{jt} \right].$$

On top of these derivations, I can actually apply matrix arithmetic and reduce the coding complexity and runtime complexity by rewriting updating component as follows.

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{X}^\intercal \cdot (\vec{y} - \vec{p})$$
$$\frac{\partial^2 L}{\partial \boldsymbol{w} \partial \boldsymbol{w}^\intercal} = -\boldsymbol{X}^\intercal \cdot \boldsymbol{W} \cdot \boldsymbol{X},$$

where

$$\begin{aligned}
\boldsymbol{X} &\in \mathbb{R}^{N \times (d+1)} \\
\vec{y} &\in \{0,1\}^N \\
\vec{w} &\in \mathbb{R}^{d+1} \\
\vec{p} &\in [0,1]^n, \vec{p} = \sigma(\vec{w} \cdot \vec{x}_t) \\
W &\in \mathbb{R}^{N \times N}, W_{t,t} = \sigma(\vec{w} \cdot \vec{x}_t) \cdot \sigma(-\vec{w} \cdot \vec{x}_t).
\end{aligned}$$

I finally use Gradient Descent with learning rate $\frac{0.02}{|\text{Training set}|} = \frac{0.02}{1400}$ and $|\text{obj}_i - \text{obj}_{i-1}| < 10^{-5} \cdot \text{obj}_{i-1}$ as terminal condition, where objective function here means log-likelihood evaluation.

Logs for error rate and log-likelihood with model learned by Gradient Descent and Newton Method are recorded every 1000 iterations as in Table 2 and 4, respectively. Convergent results are also included with star quotes.

Iteration	Accuracy	Likelihood
0	0.500000	-970.406050
1000	0.060714	-296.208211
2000	0.051429	-246.622976
3000	0.050000	-225.465064
4000	0.049286	-213.189377
5000	0.048571	-204.982818
6000	0.047143	-199.020753
7000	0.044286	-194.444074
8000	0.044286	-190.790444
9000	0.041429	-187.787061
10000	0.041429	-185.261784
11000	0.041429	-183.100136
11718	0.041429	-181.727052

Table 2: Gradient Descent Performance on Training Data

Learned weighting matrix by Gradient Descent and Newton Methods are recorded as in Table 3 and 5, respectively.

-0.689834	-0.922944	-0.953424	-0.871996	-0.791724	-0.295043	+0.798828	+1.301020
+0.101432	+0.079168	+0.067038	+0.004588	-0.044236	+0.376032	-0.730701	-1.005583
+1.720352	+1.235978	+1.052528	+0.382260	+0.205880	-1.310565	-2.183798	-1.815942
+0.696372	+0.573136	+0.306146	-0.216073	-0.555927	-1.277633	-0.030231	-0.126762
+0.352614	+0.620029	+0.029492	-0.184650	-0.516168	-0.394536	-0.139146	-0.383889
+0.749953	-0.069832	-0.198536	+0.082580	+0.130382	-0.325018	+0.387953	-0.853773
+0.888573	-0.203736	+0.829273	+0.348762	+0.182165	-0.098498	+0.262937	-0.663952
+0.211428	+0.178077	+0.477476	+0.647015	+0.302790	+0.498590	+0.075188	-0.303156

Table 3: Gradient Descent Learned Weighting Matrix

Iteration	Accuracy	Likelihood
0	0.500000	-970.406050
8	0.037143	-155.066227

Table 4: Newton Method Performance on Training Data

□

-0.842915	-1.750337	-1.392669	-1.773758	-0.687147	-1.305795	+0.493975	+1.689067
-0.422957	-0.131999	+0.300544	+0.121530	-0.841001	+0.852241	-1.477303	-1.806474
+4.393941	+1.148980	+1.579018	+0.020254	+0.727953	-2.503780	-2.645475	-3.187053
+0.599188	+0.416479	+0.740025	-0.429995	-0.819841	-2.925849	+0.351872	-0.226664
+0.548289	+1.228903	+0.092584	-0.616438	-0.800033	-0.008204	-0.638959	-0.057343
+1.196199	-0.280616	-0.545331	-1.044893	-0.175635	-1.182336	+0.930696	-2.001428
+1.758006	-0.558850	+1.281434	+0.692415	+0.379191	-0.456029	+0.280092	-1.876704
+0.640071	+0.210965	-0.365669	+1.019392	+0.310134	+0.605055	+0.736704	-1.079109

Table 5: Newton Method Learned Weighting Matrix

(b) Testing

Sol. With learned weighting matrix, I can easily predict the label for testing data. The error rate for testing data predicted by Gradient Descent and Newton Method learned weighting matrices are 0.046250 and 0.062500, respectively. It can be interpreted that Newton Method has stronger fitting power in terms of training data. □

(c) Source code

Sol. Code is appended as 2 in 5.6. □

Appendix

```cpp
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <vector>
#include <algorithm>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

double square(double x) {
    return x * x;
}

double calErr(vector<double>& dat, VectorXd& x) {
    int size = dat.size();
    double ret = 0;
    for (int l = 4; l < size; l++) {
        double pred = 0;
        for (int i = 0; i < 4; i++) {
            pred += x(i) * dat[l-1-i];
        }
        ret += square(dat[l] - pred);
    }
    return ret / (size - 4);
}

int main() {
    vector<double> tn;
    { // read training data: nasdaq00.txt
        FILE *pf = fopen("../dat/nasdaq00.txt", "r");
        if (pf == NULL) {
            fprintf(stderr, "cannot open nasdaq00.txt\n");
            exit(EXIT_FAILURE);
        }

        double v;
        while (fscanf(pf, "%lf", &v) != EOF) {
            tn.emplace_back(v);
        }

        fclose(pf);
    }

    vector<double> tt;
    { // read testing data: nasdaq01.txt
        FILE *pf = fopen("../dat/nasdaq01.txt", "r");
        if (pf == NULL) {
            fprintf(stderr, "cannot open nasdaq01.txt\n");
            exit(EXIT_FAILURE);
        }

        double v;
        while (fscanf(pf, "%lf", &v) != EOF) {
            tt.emplace_back(v);
        }

        fclose(pf);
    }

    // 5.5 (a) Prepare needed info from training set
    // to calculate \sum_{l=4}^{n-1} x_{l-i} x_{l-j}
    vector<vector<double>> vct(5, vector<double>(5, 0));
    int size = tn.size();
    for (int l = 4; l < size; l++) {
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                vct[i][j] += tn[l-i] * tn[l-j];
            }
        }
    }

    // 5.5 (a) Prepare matrix a and vector b
    // to solve a x = b => x = a^{-1} b
    MatrixXd a(4, 4);
    for (int i = 1; i < 5; i++) {
        for (int j = 1; j < 5; j++) {
            a(i-1, j-1) = vct[i][j];
        }
    }
    VectorXd b(4);
    for (int j = 1; j < 5; j++) {
        b(j-1) = vct[0][j];
    }
    VectorXd x(4);
    x = a.inverse() * b;
    printf("5.5 (a)\n");
    for (int i = 0; i < 4; i++) {
```

```
        printf("a_%d: %+f\n", i+1, x(i));
    }
    printf("\n\n");

    // 5.5 (b) Calculate MSE for training and testing
    printf("5.5 (b)\n");
    double tnErr = calErr(tn, x);
    printf("Training RMSE: %f\n", tnErr);
    double ttErr = calErr(tt, x);
    printf("Testing RMSE:  %f\n", ttErr);
    printf("\n\n");
}
```

Listing 1: Code for 5-5

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

const int NUM_TN = 700;
const int NUM_TT = 400;
const int NUM_FT = 64;

const double eps = 1e-9;
const double rel_eps = 1e-5;
const double LN_RATE = 0.02 / (2 * NUM_TN); //0.0002;

void readData(MatrixXf& x, const int nrow, const int ncol, const string str) {
    string fn = "../dat/" + str;
    FILE *pf = fopen(fn.c_str(), "r");
    if (pf == NULL) {
        fprintf(stderr, "cannot open %s\n", str.c_str());
        exit(EXIT_FAILURE);
    }

    int v;
    for (int i = 0; i < nrow; i++) {
        for (int j = 0; j < ncol; j++) {
            fscanf(pf, "%d", &v);
            x(i, j) = v;
        }
        x(i,ncol) = 1;
    }

    fclose(pf);
}

double smd(double x) {
    return 1.0 / (1 + exp(-x));
}

double calLLH(const VectorXf& y, const VectorXf& p, const int size) {
    double res = 0;
    for (int l = 0; l < size; l++) {
        res += y(l) * log(eps+p(l)) + (1-y(l)) * log(eps+1-p(l));
    }
    return res;
}

double calACC(const VectorXf& y, const VectorXf& p, const int size) {
    double res = 0;
    for (int l = 0; l < size; l++) {
        res += y(l) == (p(l) > 0.5);
    }
    return res / size;
}

int main() {
    // 5.6 (a): Training
    printf("\n\n\n[[ TRAINING PHASE ]]\n");

    // Read training data
    printf("\nRead training data\n");
    MatrixXf tn3_x(NUM_TN, NUM_FT+1);
    readData(tn3_x, NUM_TN, NUM_FT, "train3.txt");
    VectorXf tn3_y = VectorXf::Zero(NUM_TN);
    MatrixXf tn5_x(NUM_TN, NUM_FT+1);
    readData(tn5_x, NUM_TN, NUM_FT, "train5.txt");
    VectorXf tn5_y = VectorXf::Ones(NUM_TN);
    MatrixXf tn_x(2*NUM_TN, NUM_FT+1);
    tn_x << tn3_x,
            tn5_x;
    VectorXf tn_y(2*NUM_TN);
    tn_y << tn3_y,
            tn5_y;

```

```
77      // Prepare weighting matrix
78      VectorXf w = VectorXf::Zero(NUM_FT+1);
79
80      // Calculate initial error
81      VectorXf tn_p = (tn_x * w).unaryExpr(&smd);
82      double tn_acc = calACC(tn_y, tn_p, 2*NUM_TN);
83      double tn_llh = calLLH(tn_y, tn_p, 2*NUM_TN);
84      double lt_llh = tn_llh;
85      printf("#%4d TN Error (LLH): %f (%f)\n", 0, 1 - tn_acc, tn_llh);
86
87      // Start training process
88      for (int iter = 1; ; iter++) {
89          // g'(x)
90          VectorXf nw = VectorXf::Zero(64);
91          nw = tn_x.transpose() * (tn_y - tn_p);
92
93          /*{ // update w by Newton Method
94              // g''(x)
95              VectorXf tn_q = (-tn_x * w).unaryExpr(&smd);
96              MatrixXf pq = tn_p.cwiseProduct(tn_q).asDiagonal();
97              MatrixXf hs = -tn_x.transpose() * pq * tn_x;
98              w = w - hs.inverse() * nw;
99          }*/
100
101         { // update w by Gradient Descent
102             w = w + LN_RATE * nw;
103         }
104
105         tn_p = (tn_x * w).unaryExpr(&smd);
106         tn_acc = calACC(tn_y, tn_p, 2*NUM_TN);
107         tn_llh = calLLH(tn_y, tn_p, 2*NUM_TN);
108
109         if (abs(tn_llh - lt_llh) < rel_eps * abs(lt_llh)) {
110             printf("\nConverge!\n");
111             printf("#%4d TN Error (LLH): %f (%f)\n", iter, 1 - tn_acc, tn_llh);
112             break;
113         } else if (iter % 1000 == 0) {
114             printf("#%4d TN Error (LLH): %f (%f)\n", iter, 1 - tn_acc, tn_llh);
115         }
116
117         lt_llh = tn_llh;
118     }
119
120     // Print out learned weighting matrix
121     printf("\nLearned Weighting Matrix\n");
122     for (int i = 0; i < 8; i++) {
123         for (int j = 0; j < 8; j++) {
124             printf("%+f ", w(i*8+j));
125         }
126         puts("");
127     }
128
129
130     // 5.6 (a): Testing
131     printf("\n\n\n[[ TESTING PHASE ]]\n");
132
133     // Read testing data
134     printf("\nRead testing data\n");
135     MatrixXf tt3_x(NUM_TT, NUM_FT+1);
136     readData(tt3_x, NUM_TT, NUM_FT, "test3.txt");
137     VectorXf tt3_y = VectorXf::Zero(NUM_TT);
138     MatrixXf tt5_x(NUM_TT, NUM_FT+1);
139     readData(tt5_x, NUM_TT, NUM_FT, "test5.txt");
140     VectorXf tt5_y = VectorXf::Ones(NUM_TT);
141     MatrixXf tt_x(2*NUM_TT, NUM_FT+1);
142     tt_x << tt3_x,
143             tt5_x;
144     VectorXf tt_y(2*NUM_TT);
145     tt_y << tt3_y,
146             tt5_y;
147
148     // Start prediction process
149     VectorXf tt_p = (tt_x * w).unaryExpr(&smd);
150     double tt_acc = calACC(tt_y, tt_p, 2*NUM_TT);
151     printf("TT Error: %f\n", 1 - tt_acc);
152 }
```

Listing 2: Code for 5-6