# CSCI-SHU 360 Machine Learning
# Solution to homework 3

Yufeng Xu yx3038@nyu.edu

March 28, 2024

## 1 Logistic Regression

### 1.1

Let $X = [X, 1] \in \mathbb{R}^{n\times(d+1)}$, weights $W = [W, b] \in \mathbb{R}^{(d+1)\times c}$. We denote $X_{i,\cdot}$ as $X_i$, $W_{\cdot,j}$ as $W_j$, then we have:

$$\begin{aligned}
F(W) &= \frac{1}{n}\sum_{i=1}^{n} -\log[Pr(y = y_i|x = X_i; W)] + \frac{\eta}{2}||W||_F^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} -\log[\frac{\exp(z_{y_i})}{\sum_{j=1}^{c}\exp(z_j)}] + \frac{\eta}{2}||W||_F^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}(-z_{y_i} + \sum_{i=1}^{n}\log[\sum_{j=1}^{c}\exp(z_{ij})]) + \frac{\eta}{2}||W||_F^2 \\
&= -\frac{1}{n}\sum_{i=1}^{n} z_{y_i} + \frac{1}{n}\sum_{i=1}^{n}\log[\sum_{j=1}^{c}\exp(z_{ij})] + \frac{\eta}{2}||W||_F^2 \\
&= -\frac{1}{n}\sum_{i=1}^{n}(X_i W_{y_i}) + \frac{1}{n}\sum_{i=1}^{n}\log[\sum_{j=1}^{c}\exp(X_i W_j)] + \frac{\eta}{2}\sum_{j=1}^{c}||W_j||_F^2
\end{aligned}$$

Therefore,

$$\begin{aligned}
\frac{\partial F(W)}{\partial W_j} &= -\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}\{y_i = j\}X_i^T + \frac{1}{n}\sum_{i=1}^{n}\frac{\exp(X_i W_j)X_i^T}{\sum_{k=1}^{c}\exp(X_i W_k)} + \eta W_j \\
&= \frac{1}{n}\sum_{i=1}^{n}(\frac{\exp(X_i W_j)}{\sum_{k=1}^{c}\exp(X_i W_k)} - \mathbb{1}\{y_i = j\})X_i^T + \eta W_j \\
&= \frac{1}{n}\sum_{i=1}^{n}(Pr(y = j|X_i; W) - \mathbb{1}\{y_i = j\})X_i^T + nWj \\
&= \frac{1}{n}X^T\begin{pmatrix} Pr(y = j|X_1; W) - \mathbb{1}\{y_1 = j\} \\ \vdots \\ Pr(y = j|X_n; W) - \mathbb{1}\{y_n = j\}\end{pmatrix} + \eta W_j
\end{aligned}$$

Let $Y \in \mathbb{R}^{n\times c}$ where $Y_{ij} = \mathbb{1}\{y_i = j\}$, $P \in \mathbb{R}^{n\times c}$ where $P_{ij} = Pr(y = j|X_i; W)$. Then,

$$\frac{\partial F(W)}{\partial W} = \begin{bmatrix}\frac{\partial F(W)}{\partial W_1} \cdots \frac{\partial F(W)}{\partial W_c}\end{bmatrix}$$

$$= \frac{1}{n} X^T \left( P_{\cdot,1} - Y_{\cdot,1} \dots P_{\cdot,c} - Y_{\cdot,c} \right) + \eta W$$
$$= \frac{1}{n} X^T (P - Y) + \eta W$$

## 1.2

After training after modifying the logits $z'$, we obtained the results as follows. The training result is quite desirable and no overflow occurred throughout the process.

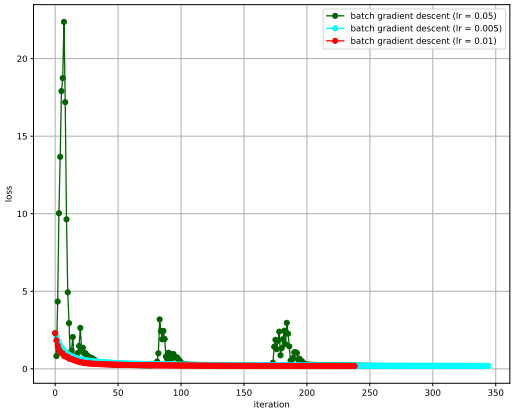


Figure 1: Loss-Iteration curve of vanilla logistic regression with different learning rates

Next we want to explain this modification is correct from a mathematical perspective. Suppose for one data sample $X_i$ we have $z'_i = z_i - \max_j z_{ij}$. Let $\max_j z_{ij} = z^*_i$, then $\Pr(y_i = k|X_i) = \frac{\exp(z_{ik})}{\sum_{j=1}^c \exp(z_{ij})}$, $\Pr'(y_i = k|X_i) = \frac{\exp(z'_{ik})}{\sum_{j=1}^c \exp(z'_{ij})} = \frac{\exp(z_{ik} - z^*_i)}{\sum_{j=1}^c \exp(z_{ij} - z^*_i)} = \frac{\exp(z_{ik}) / \exp(z^*_i)}{\left(\sum_{j=1}^c \exp(z_{ij})\right) / \exp(z^*_i)} = \frac{\exp(z_{ik})}{\sum_{j=1}^c \exp(z_{ij})} = \Pr(y_i = k|X_i)$., hence modifying the logits $z$ does not change the overall results.

## 1.3

By comparing the curves Figure 1, we have 2 observations on the pros and cons of large/small learning rates:
(1) **large learning rates accelerate the convergence of training.** When learning rate=0.05, the training terminates at 260 iterations, and when learning rate=0.01, the training terminates at 240 iterations. When learning rate=0.005, however, the training terminates at 350 iterations, which prolongs the training time. However, **large learning rates may also result in a noisier training process.** When learning rate=0.05, there are 3 sharp spikes in the loss curve, while the loss curve for learning rate=0.01 and 0.005 are much smoother. Sometimes when the learning rate is too large, the loss function may even fail to converge.
(2) In contrast to large learning rates, **small learning rates lead to smoother training process, but slower convergence.** The optimal strategy is to find a sweet spot between large and small learning rates. In this case, 0.01 is the best learning rate, which converges the fastest and has no "spikes" in the loss curve.

## 1.4

In this section, we investigate the performance of SGD, and our finetuning is focues on 4 aspects: (1) batch size; (2) learning rate; (3) weight decay; (4) learning rate annealing; (5) maximum epochs.

**Batch size:** we performed an automatic finetuning on 3 batch sizes - 10, 50, and 100. The moderate batch

size - 50, results in the fastest convergence of the loss function; when other hyper parameters are chosen properly, the smaller batch sizes always perform comparably or even outperform large batch sizes.

**Learning rate:**For most cases, learning rate=0.01 leads to a good performance of the model. However, when the batch size is very small, the learning rate needs to be cut down. For instance, when batch size=10, the learning rate needs to be cut down to 0.001 to maintain a comparable performance; when batch size = 1, the learning rate needs to be reduced to 0.0001. We suspect this is because smaller batch size leads to noisier gradients, and smaller learning rates help to mitigate this impact.

**Weight decay($\eta$):** When the weight decay is 0, there is no significant gap between train and test accuracy, which indicates generalizability is not the bottleneck that limits the model's performance. However, it takes much more epochs for the loss to converge when $\eta = 0$, which means a positive $\eta$ boosts the optimization of the model.

When weight decay is 0.01, both the train and test accuracy degrade, meaning 0.01 is too large. When weight decay is 0.005, the test accuracy is the best, 97.33%.

**Learning rate annealing($\alpha$):** We tried different annealing strategies, with annealing rate = 0.5/0.33/0.1, or no annealing at all(rate=1), the final precision is almost not affected.

**Max epoch:** Initially we set the max epoch to 500, but the training always terminates far before 500 epoch. When weight decay is 0, the training terminates at 175 epoch; when weight decay $\geq$ 0.05, the training terminates at 50-70 epoch. We decided max epoch is not an important factor in fine tuning.

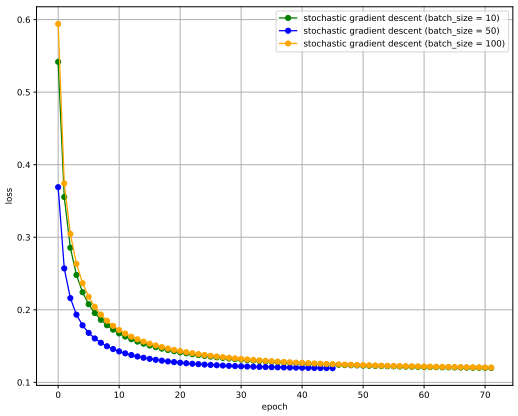


Figure 2: Loss-Iteration curve of minibatch logistic regression with different learning rate=0.001/0.01, weight decay=0.05, max epoch=500, annealing rate=0.5.

| batch size | 10 | 50 | 100 |
|---|---|---|---|
| learning rate | 0.001 | 0.01 | 0.01 |
| final F(w) | 0.1199 | 0.1195 | 0.1205 |
| train accuracy | 99.11% | 99.18% | 99.03% |
| test accuracy | 97.33% | 97.33% | 97.33% |

Table 1: final loss value, train accuracy, and test accuracy when learning rate=0.001/0.01, weight decay=0.05, max epoch=500, annealing rate=0.5.

Considering all the experiment results we obtained, we decide the set of optimal hyper parameters are: **batch size=50, learning rate=0.01, and weight decay=0.05. The impact of learning rate annealing and max epoch on the performance can be ignored.** The results of the optimal setting are displayed in Figure 2 and Table 1.

## 1.5

In order to investigate the convergence of the curves, we first fixed the learning rate to 0.01, then compared the convergence speeds corresponding to batch size=10/50/100. The results are as follows.
From Figure 3, we can see although the curves converge faster when batch size is small (10/50), the curve with batch size 10 ends up with a much larger final value of loss function.

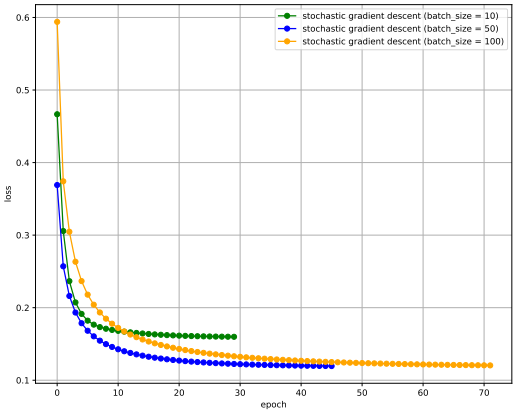


Figure 3: Loss-Iteration curve of minibatch logistic regression with earning rate=0.01, batch size=10/50/100.

Next, we scaled the learning rates linearly w.r.t the batch sizes. Because batch size=50 performs the best when learning rate=0.01, we assume 0.01 is the appropriate learning rate for batch size=50, and adjust the learning rates for batch size=10/100 based on 0.01: learning rate $= 0.01 \cdot \frac{10}{50} = 0.002$ for batch size=10, learning rate $= 0.01 \cdot \frac{100}{50} = 0.02$. The results are shown in Figure 4:

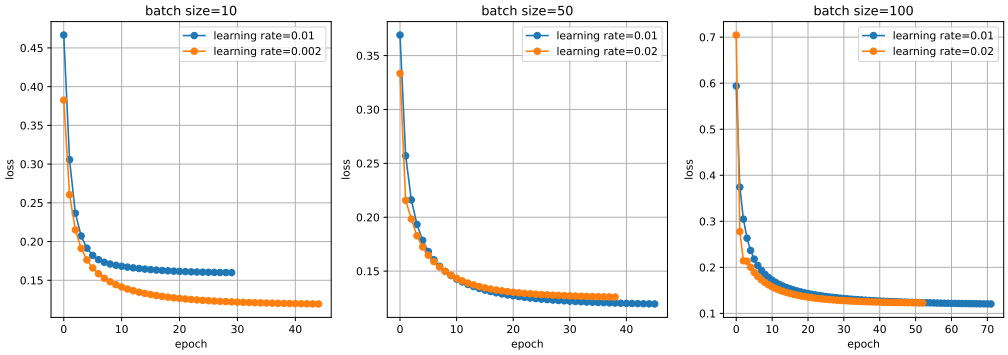


Figure 4: Loss-Iteration curve of minibatch logistic regression with earning rate=0.01, batch size=10/50/100.

As shown in the left and the right plot, scaling the learning linearly w.r.t the batch size boosts the training, either reducing the final value of the loss function or accelerating the convergence. Meanwhile, after changing learning rate from 0.01 to 0.02 for batch size=50, the performance becomes worse as the final loss value becomes larger, and test accuracy drops from 97.3% to 96.4%.
A mathematical explanation to this phenomenon is that: as the gradient is scaled by $\frac{1}{n}$, the stride of the gradient of each batch is almost constant regardless of the batch size. However, when batch size is smaller, the weights are updated more times for each epoch, resulting in faster convergence.

Meanwhile, due to the variance of the data samples, smaller batch size results in noisier gradient for each step. If the learning rate remains the same, the loss function is likely to oscillate around the minimum but fails to reach the minimum. Therefore, a smaller learning rate is adopted to make the loss function converge better at the price of slower convergence.
More specifically, # **times the weights are updated in each epoch** $\propto \frac{1}{\text{batch size}}$, $|$**gradient**$|$ **of each batch** is almost constant, so when **learning rate** $\propto$ batch size, the amount of update in each epoch $\approx$ #time updated $\times$ learning rate $\times$ gradient almost remains constant, resulting in comparable model performance.

# 2 Lasso

## 2.1

We performed coordinate descent on the generated dataset. The loss-iteration curve obtained is shown in Figure 5:

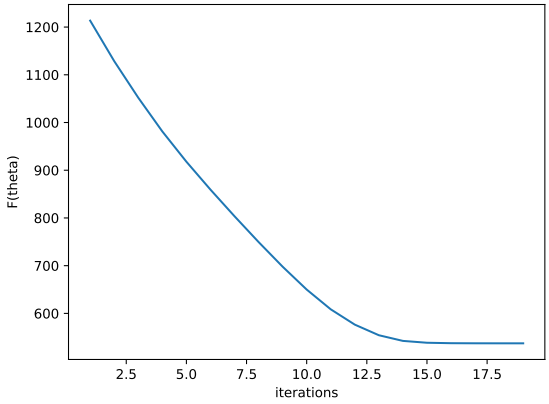


Figure 5: $F(\theta, \theta_0)$ v.s. coordinate descent step in coordinate descent. It is worth noticing the curve is non-increasing as the number of iterations increases, which corresponds to our expectations.

We also collected the indices of non-zero elements in $\theta^*$ and elements in $\hat{\theta}$ with absolute value $> 1.1 \times 10^{-1}$. The results are as follows:

- indices non-zero elements in $\theta^*$: (0, 1, 2, 3, 4)
- indices of elements with absolute value $> 1.1 \times 10^{-1}$ in $\hat{\theta}$: (0, 1, 2, 3, 4, 12, 52)

Given $1.1 \times 10^{-1}$ is much smaller than 10(the absolute value of non-zero elements in $\theta^*$), the prediction by lasso is consistent with the ground truth, which indicates our coordinate descent generates a reasonable result.

## 2.2

| precision | recall | rmse | sparsity |
|---|---|---|---|
| 38.46% | 100.0% | 0.8593 | 13 |

Table 2: Evaluation metrics obtained from $\theta$ calculated in 2.1

## 2.3

After experimenting with 50 different values of $\lambda$ evenly spaced from $\lambda_{min} = 0$ to $\lambda_{max} = ||(y - \bar{y})X||_\infty$, we obtained the precision&recall vs. $\lambda$ plot as shown in Figure 6:

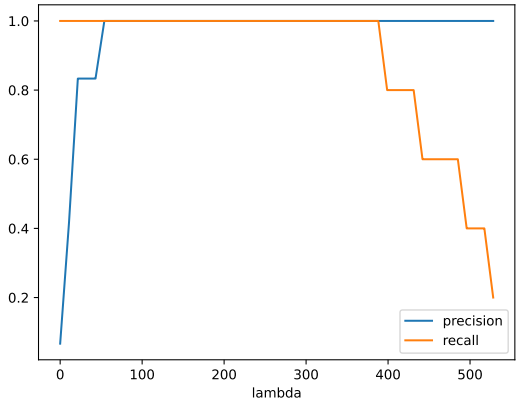


Figure 6: precision&recall vs. $\lambda$ plot

Some key observations from the plot are: (1) when $\lambda$ is very small, the precision is very low while the recall is very high. This is because the model only has a small penalty on the norms of the weights, and tends to make weights non-zero to fit the data better. As a result, the model is able to cover all the non-zero elements in $\theta^*$, while $|\{\text{non-zero indices in } \hat{\theta}\}|$ is large as well, leading to high recall and low precision.
(2) when $\lambda$ is very large, the precision is very high while the recall decreases and becomes very low. On the one hand, the model has strong penalty on the norms of the weights, hence $|\{\text{non-zero indices in } \hat{\theta}\}|$ becomes smaller and only the weights that are non-zero in $\theta^*$ as well will be kept, resulting in high precision. On the other hand, the penalty is so strong that the model fails to cover all the non-zero weights in $\theta^*$, resulting in low recall.
(3) when the value of $\lambda$ is neither too large nor too small, there is an interval where both precision and recall are 100%.
(4) when $\lambda$ is extremely large($> 600$), all the weights in $\hat{\theta}$ will become 0, resulting in a zero-division error in our program.

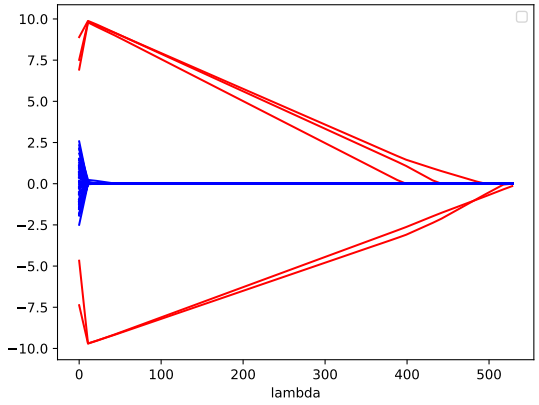


Figure 7: The value of different weights vs. $\lambda$ plot. Note the weights which are non-zero in $\theta^*$ are plotted in red, whereas weights that are zero in $\theta^*$ are plotted in blue.

We also inspected the change of the values of the weights as $\lambda$ increases. Results are shown in Figure 7. The weights that are non-zero in $\theta^*$ are plotted in red, whereas zero weights are plotted in blue. It is evident that as $\lambda$ grows, the values of the "zero-weights" converge to 0 very quickly, while the absolute values of the "non-zero weights" shrink much slower and eventually converge to 0 as well. The point where all the blue curves reach $0(\lambda \approx 50)$ corresponds to the point that precision reaches 100% in Figure 6($\lambda \approx 50$).

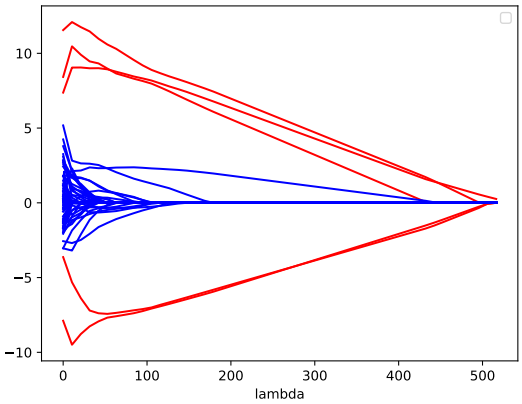


Figure 8: The value of different weights vs. $\lambda$ plot when $\sigma = 10.0$. Note the weights which are non-zero in $\theta^*$ are plotted in red, whereas weights that are zero in $\theta^*$ are plotted in blue.

Next, we changed the standard deviation of the data generator that generated $X$ and $y$ from 1 to 10, and obtained a new lasso solution path plot, as shown in Figure 8. Some key observations are:
(1) the blue curves converge much slower and randomly than their counterparts in Figure 7.
(2) the red curves converge at almost the same rate but much more randomly, with march larger gap between curves of the same color (the same for blue curves).
(3) There is one blue curve that converges to 0 after a red curve converge to 0. Consequently, there does not exist a $\lambda$ such that both precision and recall are equal to 100%. In other words, more variance in the data bring more difficulty for the optimization of the model, making it less likely to perform well on all metrics.

## 2.4

In this section, we tried different combinations of $n$ and $m$ - ($n = 50$,$m = 75$), ($n = 50$, $m = 150$), ($n = 50$, $m = 1000$), ($n = 100$, $m = 75$), ($n = 100$, $m = 150$), ($n = 100$, $m = 1000$) - to explore the robustness of the model under different settings. We inspected the values of $\lambda$ that result in both good precision and recall (Table 3).

| $n$ \ $m$ | 75 | 150 | 1000 |
|---|---|---|---|
| 50 | [50, 380] | [40, 420] | [200, 440] |
| 100 | [50, 900] | [40, 750] | [50, 640] |

Table 3: The intervals of "good $\lambda$'s" where both precision and recall are high. Note for n=50, m=1000, even in the optimal interval we only have precision and recall$\approx$ 0.8. For other combinations of $n$ and $m$, both precision and recall are 1 within the intervals shown above.

We also plotted the lasso solution path plots corresponding to these combinations of $n$ and $m$. The results are shown in Figure 9.

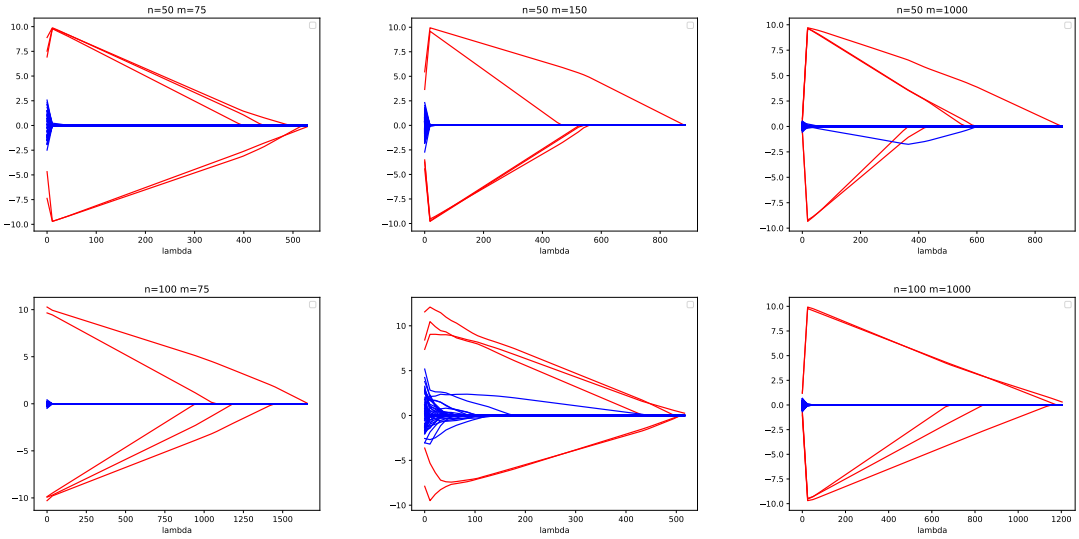


Figure 9: Lasso solution path plot corresponding to different $n$ and $m$.

## 2.5

**Implementation:** Compared to the previous dataset, this dataset has a much larger scale and demands a more efficient implementation of Lasso regression. We made two major changes in the code: (1) replaced all the numpy 2d and 1d arrays with scipy.sparse arrays to save space and time. (2) modified the way to compute $r$. Recall the definition $r_{i,j} = y_i - (\sum_{j' \neq j} w_{j'} x_{i,j'})$, hence $r_j = y - (\sum_{j' \neq j} w_{j'} x_{j'})$. Initially this was implemented by masking $w$ and $x$, but later we modified the method to $r_j = y - (\sum_{j'} w_{j'} x_{j'}) + w_j x_j$. Before the modification, the computation for $r, a, c$ took 0.008s, in contrast to 0.002s after the modification. Therefore, the efficiency of lasso regression is improved significantly. We compared the efficiency of the two implementations by checking the time for one iteration to run for the train set (n=30000, m=2500), and the time it takes to run a randomly generated data with n=30000 and m=2500. The modified version runs 5.60 sec for each iteration, whereas the old version took over 30 seconds to return the result.

**RMSE:** We modified $\lambda_{min}$ to $0.1 \cdot \lambda_{max}$ and number of $\lambda$'s to 20. The train & validation RMSE vs. $\lambda$ curve is shown in Figure 10:

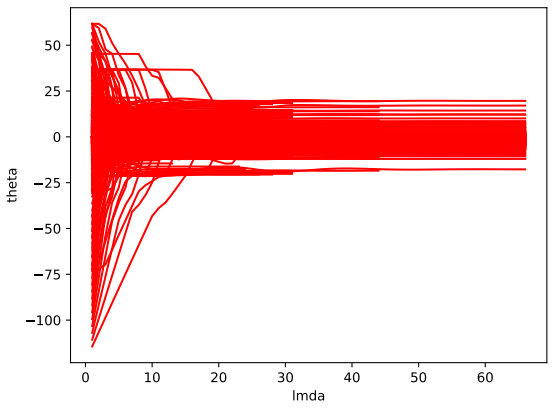


Figure 10: The train & validation RMSE vs. $\lambda$ curve we obtained.

**Lasso Solution Path:** The lasso solution path curve we obtained is shown in Figure 11:

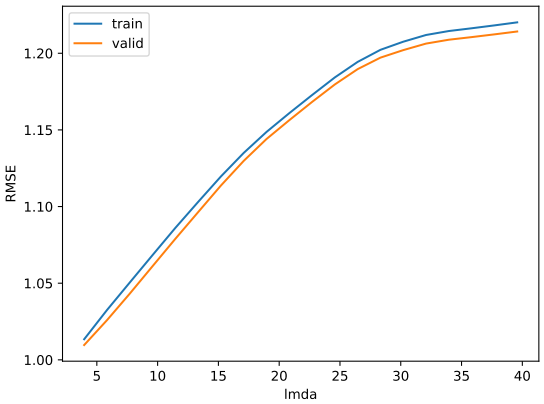


Figure 11: The lasso solution paths.

**Best $\lambda$:** By comparing the RMSE scores on the validation set, we selected the best $\lambda$ - **3.958**, and tested its performance on the test set. The test RMSE is **1.025**. The value is a little bit higher than what we have expected. One possible explanation is that $\lambda_{min}$ is set too large. With smaller $\lambda$ we may be able to obtain better RMSE performance.

**Top-10 features** we selected the top 10 features with the highest absolute values of weights based on the best $\lambda$(3.958) we obtained. The features and their weights are shown in Table 4:

| feature | great | not | best | amazing | love | delicious | rude | the worst | horrible | awesome |
|---|---|---|---|---|---|---|---|---|---|---|
| weight | 19.63 | -17.75 | 17.06 | 14.34 | 12.23 | 12.13 | -11.99 | -11.95 | -10.49 | 10.06 |

Table 4: The top 10 features with the highest absolute values of weights when using the best $\lambda$ value.

The result in this table makes pretty much sense because the words with large positive weights are often associated with positive ratings, whereas the words with small negative weights are relevant to negative remarks.