Machine Learning

Homework 1

B02901143 楊筑雅

2016/10/14

I. Linear regression function by gradient descent
Here is linear regression function:

$$y = b + \sum_{i}^{n} w_i \cdot x_i + w_{n+i} \cdot x_i^2 + \ldots + w_{n(m-1)+i} \cdot x_i^m$$

After training and validation, I choose the model with order = 1(m = 1), which is:

$$y = b + \sum_{i}^{n} w_i \cdot x_i$$

The loss function, which implies the estimation error, is:

$$L(w, b) = \sum_{k}^{\|} \left(\hat{y}^k - \left(b + \sum_{i}^{n} w_i \cdot x_i^k \right) \right)^2$$

The gradient is:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix} \qquad \begin{aligned} \frac{\partial L}{\partial w_i} &= \sum_{k}^{\|} 2 \left(\hat{y}^k - \left(b + \sum_{i}^{n} w_i \cdot x_i^k \right) \right) (-x_i^k) \\ \frac{\partial L}{\partial b} &= \sum_{k}^{\|} 2 \left(\hat{y}^k - \left(b + \sum_{i}^{n} w_i \cdot x_i^k \right) \right) \end{aligned}$$

To get smoother function, I also use regularization. Then the loss function is:

$$L(w, b) = \sum_{k}^{\|} \left(\hat{y}^k - \left(b + \sum_{i}^{n} w_i \cdot x_i^k \right) \right)^2 + \lambda \sum_{i}^{n} (w_i)^2$$

The gradient is:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix} \qquad \begin{aligned} \frac{\partial L}{\partial w_i} &= \sum_{k}^{\|} 2 \left(\hat{y}^k - \left(b + \sum_{i}^{n} w_i \cdot x_i^k \right) \right) (-x_i^k) + 2\lambda w_i \\ \frac{\partial L}{\partial b} &= \sum_{k}^{\|} 2 \left(\hat{y}^k - \left(b + \sum_{i}^{n} w_i \cdot x_i^k \right) \right) \end{aligned}$$

In order to find the proper parameters (w, b) to make the loss the lowest, use gradient descent to update the parameters:

$$w^{t+1} = w^t - \eta \frac{\partial L}{\partial w} \qquad b^{t+1} = b^t - \eta \frac{\partial L}{\partial b}$$

Here is my code:

```
for it in range(100001):
```

```
#regularization
tr_L = lam * np.sum(W ** 2)
val_L = lam * np.sum(W ** 2)
dW = lam * 2 * W
db = 0
for i in range(tr_X.shape[0]):
    tr_L += (tr_y[i] - (np.sum(tr_X[i] * W) + b)) ** 2
    dW += 2 * (tr_y[i] - (np.sum(tr_X[i] * W) + b)) \
            * (-tr_X[i])
    db += 2 * (tr_y[i] - (np.sum(tr_X[i] * W) + b))
#validation
for i in range(val_X.shape[0]):
    val_L += (val_y[i] - (np.sum(val_X[i] * W) + b)) ** 2
#adagrad update
dW_adagrad += dW ** 2
db_adagrad += db ** 2
W -= eta * dW / np.sqrt(dW_adagrad)
b -= eta * db / np.sqrt(db_adagrad)
```

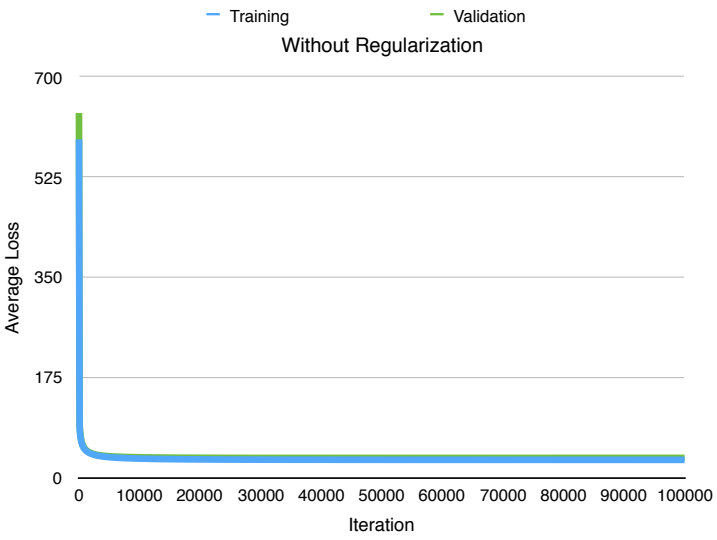
II. My method

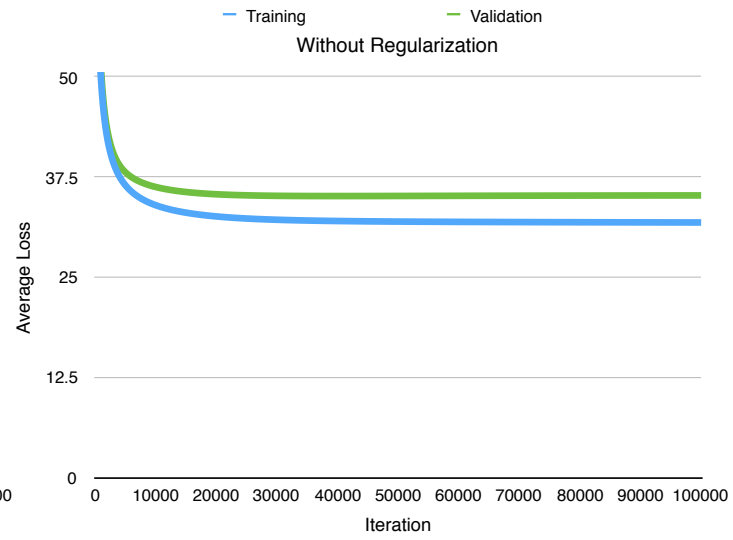
1. Feature extraction
   I use all data of 18 subjects in 9 hours to be feature X. That is, there are totally 162 x. Feature y is the PM2.5 of the tenth hour. I extract all continuous 10-hours data, even those across two days. Then I shuffle the data to avoid learning wrong feature. I use 2/3 data to be training data and 1/3 data to be validation data, so I can choose the best model among several ones and notice if overfitting occurs.

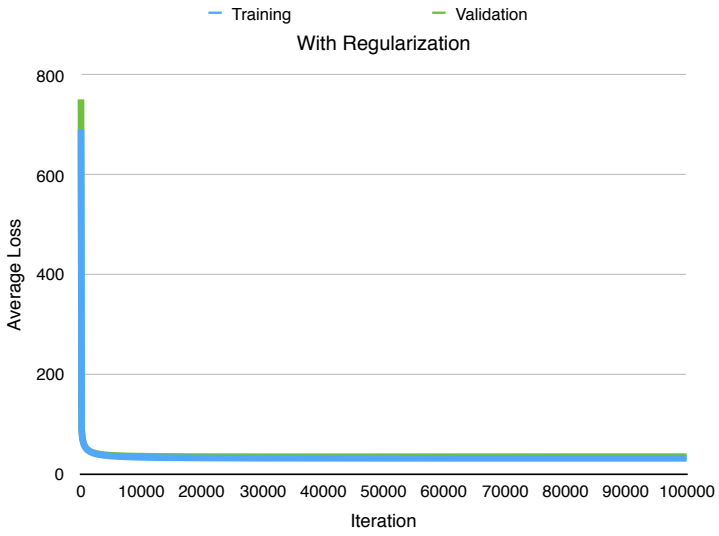
2. Cost computing
   I use linear regression with order = 1 and regularization with $\lambda = 100$ to calculate the loss of training data and validation data.

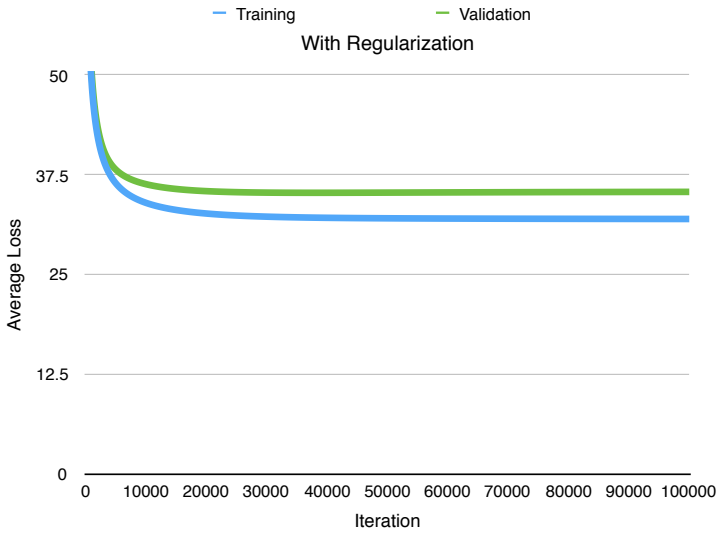
3. Parameters update
   I use gradient descent to update parameters. However, vanilla update, which use fixed learning rate, is not efficient. If learning rate is large, the loss may not reach the minimum; if learning rate is small, it takes a lot of time for the loss to converge. Therefore, I have tried Adagrad and Adam. After comparing, I chose Adagad in the end.

III. Discussion on regularization
Regularization makes the weights closer to 0, that is, makes the function smoother. From the figures below, the results with regularization is slightly better than the one without regularization. The reason why the results are so close may be that initial values of weights are already close to 0 or that λ is not large enough to see the difference.







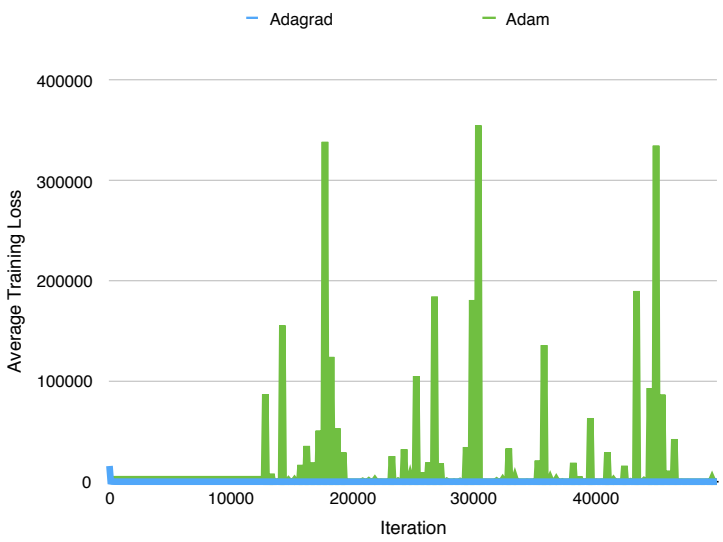


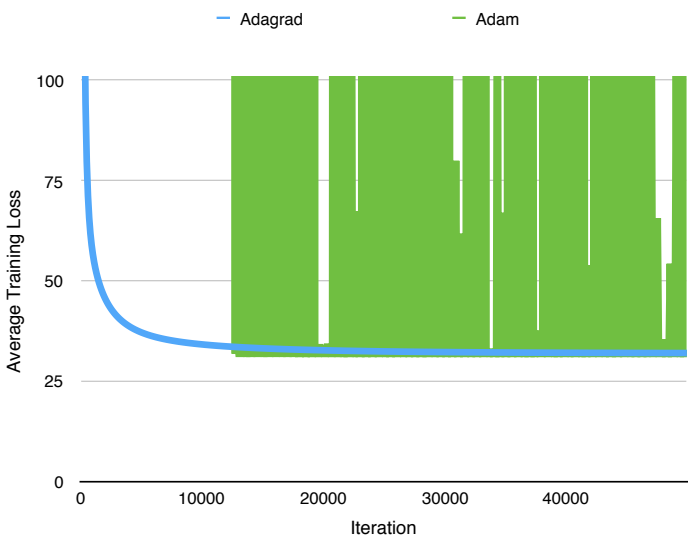
IV. Discussion on learning rate

Here is the code about Adam:

```
m_W = beta1 * m_W + (1 - beta1) * dW
v_W = beta2 * v_W + (1 - beta2) * (dW ** 2)
m_b = beta1 * m_b + (1 - beta1) * db
v_b = beta2 * v_b + (1 - beta2) * (db ** 2)
#bias correction
m_W_hat = m_W / (1 - beta1 ** (it + 1))
v_W_hat = v_W / (1 - beta2 ** (it + 1))
m_b_hat = m_b / (1 - beta1 ** (it + 1))
v_b_hat = v_b / (1 - beta2 ** (it + 1))
W -= eta * m_W_hat / (np.sqrt(v_W_hat) + eps)
b -= eta * m_b_hat / (np.sqrt(v_b_hat) + eps)
```

Figures below show the average training loss (training loss / number of training data) in 50000 iterations using Adagrad and Adam. That the loss of Adagrad drops quickly at first and keep deceasing slowly is quite obvious since the learning rate of Adagrad gets smaller as the number of iterations gets more. Unlike Adagrad, Adam also keeps a momentum. There are some extremely large losses in the middle of iterations, so it tells that learning rate sometimes is too large. However, it is mostly going on the right way and the losses of Adagrad and Adam are almost the same in the end.


Adagrad
Adam
400000
300000
200000
100000
0
Average Training Loss
0
10000
20000
30000
40000
Iteration


Adagrad
Adam
100
75
50
25
0
Average Training Loss
0
10000
20000
30000
40000
Iteration