HW1 Project PM2.5 Report

Student number : r05921012
Name : 吳宗澤

1. Input process

First, let me introduce how I process the input data from the train.csv . Because I just want the numeral data from the input data , I just jump the first line in the data and pop out the first three elements in every lines that is splited by comma and append into a list(dimensions : 4320 x 24). However, to multiply the weight matrix and training feature matrix conveniently, I transfer the matrix that its columns represent the 18 kinds of features and its rows represent the time from the first day number 0 hour to the last day number 23 hour in order(dimensions : 5760 x 18).

2. Method and regression function

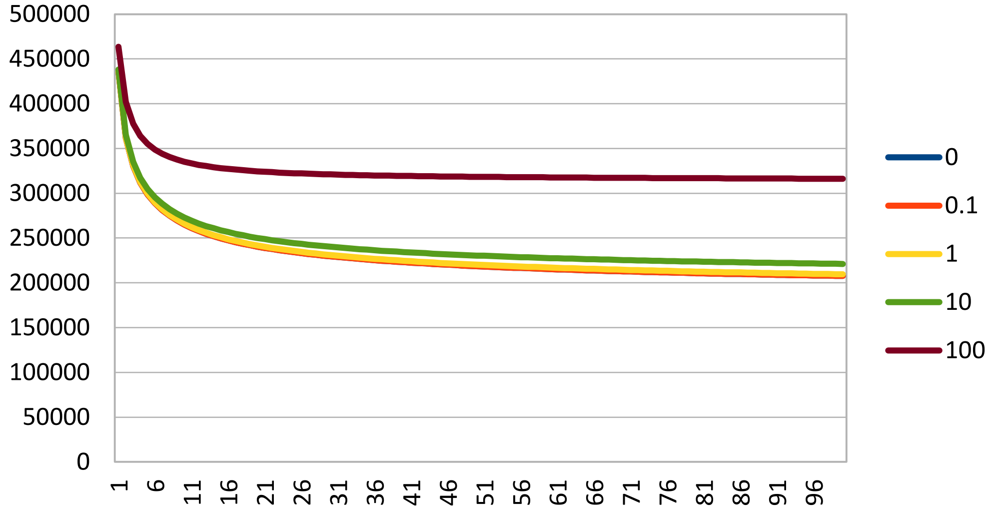
I try two kinds of regression functions that they are $y = b + w1 \cdot x1$ and$y = b + w1 \cdot x1 + w2 \cdot x2$. x1 is the training feature that I pick 10 kinds and 9 hours data into a numpy array(dimensions : 9 x 10) , and x2 is only the square of PM2.5 data for 9 hours that I place into a numpy array(dimensions : 9 x 1). The reason I choose those 10 features is the elements that really affect PM2.5. After comparing two kinds of regression function performance, $y = b + w1 \cdot x1$ is the one I choose. The loss function I use are the similar one mentioned in the class with the regularization. In addition, I try two kinds of adaptive learning rate methods that are adam and adagrad, but what really surprise is that adagrad has a better and stable performance than adam.

Additionally, I try two different methods to update my weight and bias. One is updating weight and bias per every training data, and another is updating them per every 20 training data. At beginning, updating every time loss function will fall dramatically than updating every 20 data, but after a while, the second method will have better performance that loss function drops much faster. At last, I choose the one which updates per every data, because it can converges to a lower loss.

Moreover, I also try two kinds of method in the learning process. The first one is taking features from 0~8 hours to estimate number 9 hour and then form 1~9 hours to estimate the number 10 hours in sequence in every months. Another one is randomly picking 9 hours to estimate number 10 hour in every months. Comparing this two performances, the process in time sequence has a much higher performance. (Code is at the bottom)

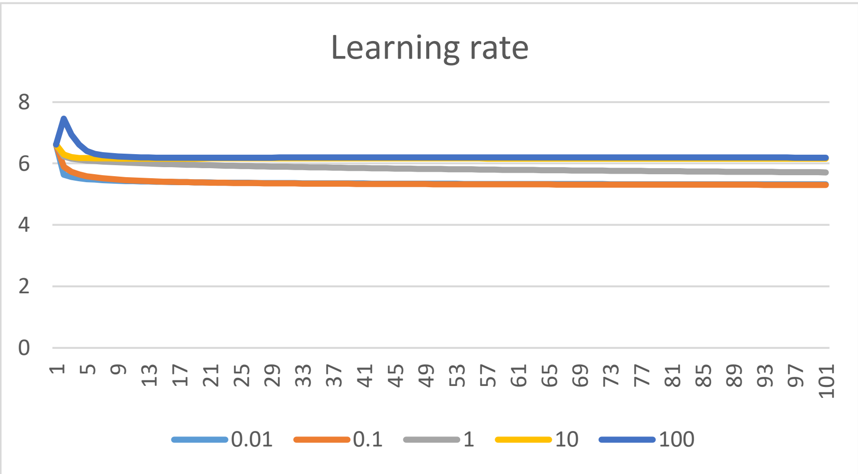
3. Regularization

I try several regularization delta into the differential equation of the loss function. With the regularization delta bigger, the loss will converge at a bigger number and decrease at a very low velocity. The picture below is the graph of iteration and loss with different delta. The regularization isn't work well in our cases, because our regression function isn't a high order function.



4. Learning rate

Although I am using adaptive learning rate, the learning rate we assume still affect the loss. We can find out that a bigger learning rate is stuck at a bigger loss, and learning rate equals to 0.1 has the best performance in this cases. (the loss is after I take a log10)



CODE(main part of linear regression)

```python
def gradient_descent(b_starting,w_starting,learning_rate,regulation_delta,num_iteration,final_data):
    b = b_starting
    w = w_starting
    acceleration_b = 0
    acceleration_w = 0
    prev_loss =0
    loss = 41651
    for u in range(num_iteration):
        for i in range(12):
            differential_b = 0
            differential_w = np.zeros((9,10))
            for a in range(471):
                #input matrix
                x = final_data[a+i*480:a+9+i*480,0:10]
                #weight times input matrix
                w_multiply_data  = np.multiply(w,x)
                w_multiply_data = np.sum(w_multiply_data)
                #differential equation
                diff_b = -2*(final_data[9+a+i*480,9] - (b + w_multiply_data))
                diff_w = -2*x*(final_data[9+a+i*480,9] - (b + w_multiply_data)) + 2*w*regulation_delt
                differential_b += diff_b
                differential_w += diff_w
                #adagrad the differential square sum
                acceleration_b += diff_b**2
                acceleration_w += np.square(diff_w)
                gradient_b = differential_b
                gradient_w = differential_w
                #updating
                b =  b - (learning_rate*gradient_b/math.sqrt(0.00000000001+ acceleration_b))
                w = w - (learning_rate*gradient_w/np.sqrt(0.00000000001+acceleration_w))
                differential_b = 0
                differential_w = np.zeros((9,10))
        #if loss varies only a little , stop
        prev_loss = loss
        loss = compute_loss_function(b,w,regulation_delta,final_data)
        if abs(prev_loss - loss) < 0.15:
            break
        print loss
    return b,w
```