## **L1-regularization - Theory**

We have discussed generalization and retraining in our lectures. We have seen that even when we add a column of random noise to our data, we can still improve our training rate. In general, we want the dimension of our X to be much smaller than the number of observations of N. Another possible problem is that the dimension is much larger than the number of observations for specific data sets.

One way to show this is to build a matrix X. We want our matrix X to be thin when N is large and D is small. In the opposite case, when the matrix X is wide, N is small, and D is large, and it is not very well, and we need to take some actions to avoid potential problems.

In this situation, we can select only a small number of the most important factors that set the trend from the whole array of factors, and remove all the others that are just noise. It is called sparsity, since most of the factors will be zero, and only a small number will not be zero. In this lecture, we discuss L1-regularization, which allows us to achieve sparsity.

The basis of L1-regularization is a relatively simple idea. As in the case of L2-regularization, we merely add a penalty to the original cost function. Just as with L2-regularization, we use L2- rationing for the correction of weighting coefficients, with L1-regularization we use special L1- rationing. L2-regularization is also called Ridge regression, and L1-regularization is called lasso regression.

J_{RIDGE} = - sum_{i=1}^{N} (t_n log; y_n + (1 - t_n) ;log (1 - y_n))+ lambda parallel wparallel^2_2, 

J_{LASSO} = - sum_{i=1}^{N} (t_n log; y_n + (1 - t_n) ;log (1 - y_n))+ lambda parallel wparallel_1. 

Before we tackle the problem, let’s consider the probability distribution. You know that the exponent of the negative square is the Gaussian distribution, so with L2-regularization, we had Gaussian likelihood and a Gaussian prior for w. In this case, we no longer have a Gaussian prior by w.

Which distribution has a negative absolute value in the exponent? This is the Laplace distribution.

p (w) = frac {lambda} {2} exp (- lambda |w|).

Thus, for L1-regularization, we have a Laplace distributed prior of weighting coefficients and find a solution for the posterior for w with the Laplace prior.

We find a gradient and move towards it:

J = - sum_{n=1}^{N} (t_n ;log;y_n + (1 - t_n); log (1 - y_n)) + lambda ||w||, 

frac {partial J} {partial w} = X^T (Y - T) + lambda sign (w). 

With L1-regularization, you have already known how to find the gradient of the first part of the equation. The second part is λ multiplied by the sign (x) function. The sign (x) function returns one if x> 0, minus one if x <0, and zero if x = 0.

**L1-regularization - The Code**

I suggest writing the code together to demonstrate the use of L1-regularization.

The plan is that we will generate some data, where the input variables are represented by a broad matrix, whereas Y will depend only on several factors, and the rest will be only noise. Then we use L1-regularization to find the sparse weighting coefficients that determine the useful dimensions of X.

If you do not want to write the code yourself, but just run it, the corresponding file is in the repository called l1\_regularization.py.

So, let’s start with importing the Numpy and Matplotlib libraries.

import numpy as np

import matplotlib.pyplot as plt

We also define our sigmoid function. You have already known it

*def* sigmoid(*z*):

return 1 / (1 + np.exp(-z))

Set N = 50 and D = 50 to make it a wide matrix. The values of X are uniformly distributed in the range from -5 to +5.

N = 50

D = 50

X = (np.random.random((N,D)) – 0.5)\*10

The true values of the weighting coefficients, which are defined by the variable true\_w, are set to 1, 0.5 and -0.5, so only the first three dimensions have values, and the remaining 47 dimensions are set equal to zero. This does not affect the result in any way.

true\_w = np.array([1, 0.5, -0.5] + [0]\*(D-3))

Now define our Y. It will be a sigmoid of X plus some random noise.

Y = np.round(sigmoid(X.dot(true\_w) + np.random.randn(N)\*0.5))

The next is the gradient descent. The training coefficient is set to 0.001, the penalty for L1-normalization is set to 2. I recommend you to experiment with different values of the penalty and see what happens. The number of iterations is set to 5000. Besides, we calculate the value of the cost function. We know how to do this.

costs[]

w = np.random.randn(D) / np.sqrt(D)

learning\_rate = 0.001

l1 = 2.0

for t in *xrange*(5000):

Yhat = sigmoid(X.dot(w))

delta = Yhat – Y

w = w – learning\_rate\*(X.T.dot(delta) + l1\*np.sign(w))

cost = -(Y\*np.log(Yhat) + (1-Y)\*np.log(1 – Yhat)). mean() + l1\*np.abs(w).mean()

costs.append(cost)

plt.plot(costs)

plt.show()

Finally, we draw the graphs of the true weighting coefficients with the calculated ones

plt.plot(true\_w, *label*=’true w’)

plt.plot(w, *label*=’w map’)

plt.legend()

plt.show()

Run the program and see what happens.

As you can see, the cost function converges quite quickly.

The value of the weighting coefficients has been calculated well, but it does not entirely coincide with the real ones.

We try again, but we set the penalty of L1-regularization equal to 10.

l1 = 10.0

We see a new cost function and weighting coefficients.

The coefficients are now much closer to zero since they are shifted into this area by regularization. Therefore, try to expose a smaller regularization penalty to achieve success. If it works out well, then keep its value as close as possible to zero.