Large-scale Logistic Regression

R05922002 廖建棋

Environment / How to Run

I intended to implement algorithms in python. However, I found that the data loading function is not efficient, taking too long time to load into the memory. Therefore, I transfer to Matlab(R2017b) and train the model on the workstation of CMLab, which has Intel Xeon E5-2650 v2 CPU and 128GB RAM installed. I assume the function *libsvmread* is available in TA's machine. The scripts are *train.m* and *predict.m*, and we can invoke the primary functions in the project directory with matlab command. Run the following example command to train the model:

matlab -r "train('./kddb', 'gd', 0.1);"

The first argument of train is the path to the training file, and the second argument specifies the optimization algorithm. Use *'gd'* to run gradient descent with line search. Other strings (e.g. 'nt') will use the approximated Newton method to determine the descent direction. The third argument sets the coefficient C. Other parameters are specified in the source code, which follows the slides. This function will save a *weights.mat* file in the current directory.

Run the following example command to predict the test data:

matlab -r "predict('./weights.mat', './kddb.t');"

The first argument specifies the path to the model's weights. The second argument represents the path to the testing file. This function will print the testing accuracy in the end.

Computational Analysis

I think the computational bottleneck is the matrix-vector multiplication between data matrix X and a vector. It took around 2 seconds to finish, and it is inevitable. Therefore, I tried to avoid this operation and cache the result for later use. I also cached some frequently used vector for saving the recomputations. For example, I cached e^{-ywx} to save the exponent operation and element-wise vector products. I also found a wired scenario when doing the matrix-vector multiplication on the sparse matrix X. The Hessian-free equation caused some problem in Matlab if we directly translate it into code. As I mentioned previously, the matrix-vector product between X and a vector took 2 seconds, so in theory, the equation $X^T(D(Xs))$ should take around 4 seconds since it multiplies X twice. However, I found that it took 44 to 50 seconds when doing X^T times a vector. I think it was caused by the inner sparse representation of X, but it didn't happen when I tried it in the Matlab shell. Anyway, I swapped

the multiplication order of X^T and the vector by transposing them, so the X is avoided, and the issue is solved.

According to the previous bottleneck analysis, the computation cost of each method can be clarified. The line search of gradient descent must updates Xw, and it usually takes 17 iterations to find α. Therefore, each iteration of gradient descent at least took 34 to 45 seconds. Besides, gradient descent was hard to achieve the stopping criteria that the norm of gradient must be less or equal to 0.01 times the initial norm. Thus, I set a maximum of 100 iterations which took 1 hour and 11 mins to finish.

On the other hand, the bottleneck of Newton method is the Hessian free computation, and the stopping condition of the conjugate gradient(CG) was hard to fulfill in later iterations of the outer descending loop, so I set a limited 50 iterations of CG. However, the line search didn't perform since α=1 was good enough after CG. Unlike gradient descent method, The stopping criteria for descending can be achieved in 5 to 6 iterations.

By the way, I suspected the data loading function of Matlab version is not as efficient as of LIBLINEAR version since it took around 2.5 minutes to load the training data into Matlab. Other configuration details and enhancements are not clear. Therefore, I think the comparison of Newton method between LIBLINEAR and my Matlab version can only be in rough.

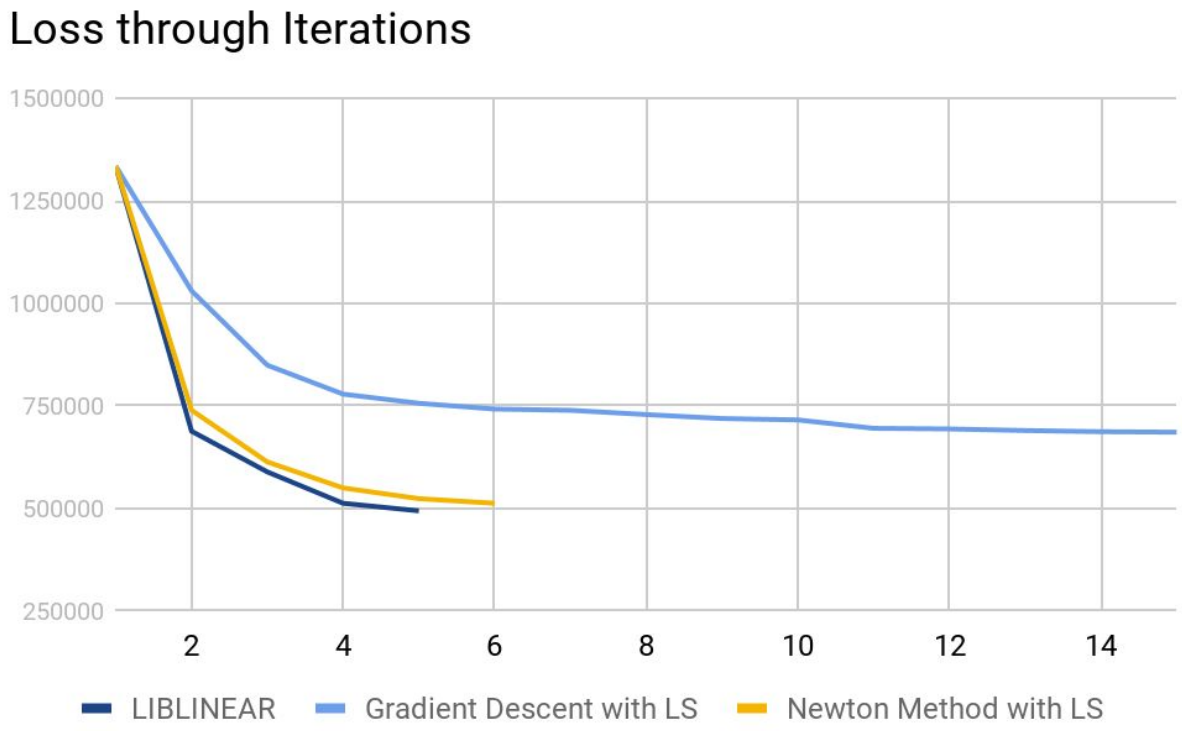
Comparison

I Compared 3 algorithms in the following table and the figure. I omitted the losses of gradient descent after 15 iterations in the figure since they didn’t change too much. The loss values of LIBLINEAR was copied from the logs, so they weren’t exact.

Method	LIBLINEAR	Gradient Descent	Newton Method
Time	10m 51s	1hr 11m	14m 43s
Descent Iterations	5	100	6
Total CG Iterations	118	NO CG	237
Final Loss	around 494200	644587.341802	512626.762094
Testing Accuracy	89.9973%	88.893387 %	89.859447 %

We can see that Newton methods are very efficient, taking only 5 to 6 iterations to reach the optimal and having lower losses in the last iteration than gradient descent, but my CG spent more iterations than LIBLINEAR's. However, The running time and the testing accuracy of my Newton method was still comparable with LIBLINEAR, and the learning curves were similar.

There must have some special techniques in LIBLINEAR such as trust region instead of line search to improve later convergence of CG.



Summary

Through this project, I implemented two logistic regression algorithms solving a large-scale classification problem. Thanks to the sparse matrix data structure, efficient matrix operations in Matlab, and sufficient memory from the workstation, otherwise it is impossible to solve this problem in such huge feature dimension space. The bottleneck is on the matrix-vector multiplication on dataset X, taking 2 seconds to complete. Newton method is a lot more efficient than gradient descent, which becomes very slow after a few iterations, but the computation of CG in the Newton method is the trade-off, so sophisticated techniques should be introduced to reduce the CG iterations.