OPTML programming homework: Logistic Regression

B01902004 資工四 蔡捷恩

0. Environment:

Matlab R2015b

1. Terms

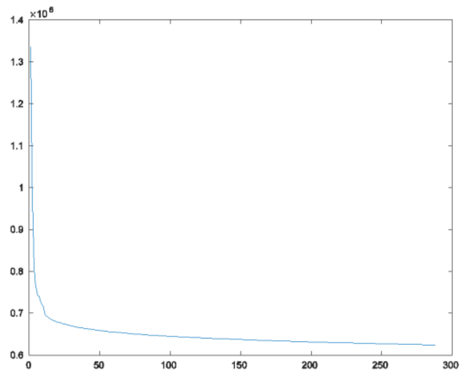
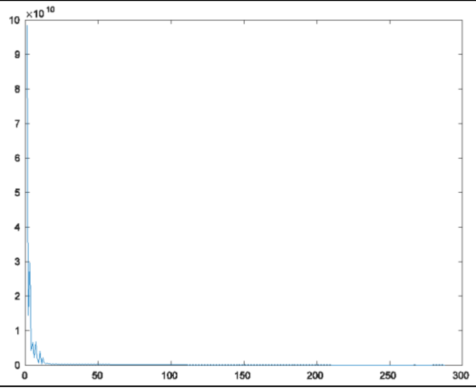
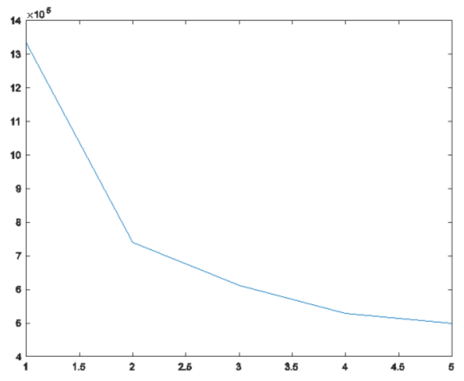
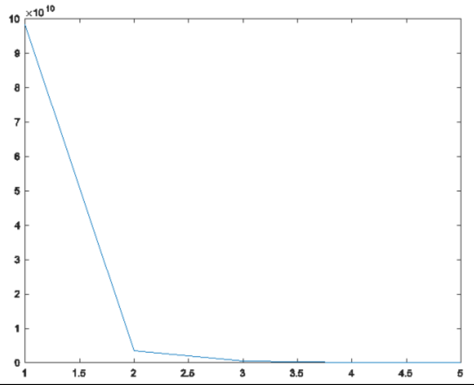
- Objective:	$f(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\log(1+e^{-y_i\boldsymbol{w}^T x_i})$
- GD:	Gradient Descent method
- NM+LS:	Newton's Method with conjugate gradient and backtracking Line-Search

2. Experiment:

GD vs. NM+LS

Using the settings of $[C = 0.1, \varepsilon = 0.01, \eta = 0.01]$, max Iteration = 1,000 and for NM+LS, $\xi = 0.1$, we compare GD & NM+LS in full KDD2010 dataset.

	GD	NM+LS
Initial objective ($\mathbf{w}^0 = \mathbf{0}$)	1.335e+06	1.335e+06
Iter. to convergence	289	5
Ending objective	6.240e+05	4.982e+05
Elapsed time	4 hr 26 min 45 sec	5 hr 5 min 56 sec
Training accuracy	86.78%	90.49%
Testing accuracy	88.98%	89.98%

	Interation-Objective	Iteration-$\|\nabla f(\boldsymbol{w})\|^2$
GD		
NM+LS		

By the chart above we can find that:

- The decreasing of objective is guaranteed by backtracking line search.
- By looking at the norm of the gradient ($\|\nabla f(\boldsymbol{w})\|^2$) at each iteration, we can find that by employing NM+LS (right), we can find a quite nice step at each iteration that leads to smaller $\|\nabla f(\boldsymbol{w})\|^2$. The value of $\|\nabla f(\boldsymbol{w})\|^2$ in chart of GD method (left) jitters a lot instead. This finding point out the main difference between GD and NM+LS: *NM+LS tends to cost less iteration than GD since it attempts to make the "gradient after taking the Newton step* $\boldsymbol{s}$*", namely,* $\nabla f(\boldsymbol{w}+\boldsymbol{s})$ *close to* $\boldsymbol{0}$ *by finding a* $\boldsymbol{s}$ *such that the second order Taylor's approximation* $\nabla\hat{f}(\boldsymbol{w}+\boldsymbol{s})=\boldsymbol{0}$. However, in GD method it does not guarantee this thing. This effect is especially obvious to see when current $\boldsymbol{w}$ is close to optimum $\boldsymbol{w}^*$. In GD method, in the last few iterations, the objective drop very slow (~0.01 every 30 iteration) but the $\nabla f(\boldsymbol{w})$ still jittering such that it cannot hit a small enough $\nabla f(\boldsymbol{w})$ that satisfies the stopping criterion. On the other hand, NM+LS converge quickly. (this effect is illustrated in figure 1.)

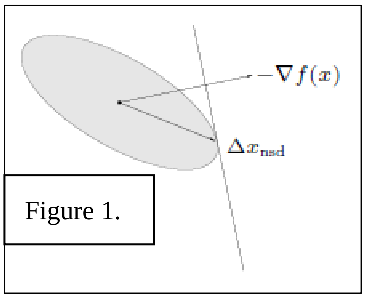


Figure 1.

- Observe that all the step size equals to 1 in the very begin. This may indicate that the NM+LS method is in the ***quadratically convergent phase*** early on. This also explain why NM+LS cost significantly less iteration then GD.

LIBLINEAR vs. MY IMPLEMENTATION

Here we compare the LIBLINEAR toolkit (including trust region Newton's Method (TR) and line search method (LS)) and my implementation. Notice that the stopping criterion of LIBLINEAR is slightly different from ours to avoid unbalanced label, it is actually:

$$\|\nabla f(\boldsymbol{w})\| \le \varepsilon \times \frac{min(pos, neg)}{l} \times \|\nabla f(\boldsymbol{w_0})\|$$

Where *pos* is *#positive_sample*, *neg* is *#negative_sample*, and *l* is *#all_samples*. To fairly compare each method, I change the stopping criterion to the same criterion of LIBLINEAR to conduct the following experiment on full dataset.

	LIBLINEAR-TRON	LIBLINEAR-LS	NM+LS
Initial objective	1.335e+06	1.335e+06	1.335e+06
Iter. to convergence	106	6	6
Ending objective	4.944e+05	4.940e+05	4.9332e+05
Elapsed time	2 hr 56 min 57 sec	51 min 46 sec	7 hr 55 min 55 sec
Training accuracy	90.26%	90.56%	90.54%
Testing accuracy	89.99%	89.99%	89.99%

※ By changing the stopping criterion, even if all the parameters remain unchanged (the only difference between NM+LS in last section and reported here is the stopping criterion), NM+LS converge a even smaller objective.

It is interesting to see trust region method actually cost more iteration to converge to a fair objective. I conclude the following points:

- The idea of trust region Newton method is try to find the next step $\boldsymbol{s}$ by solving the following problem:

$$\begin{aligned} &\text{minimize} \quad \frac{1}{2}\boldsymbol{s}^T\nabla^2 f(\boldsymbol{w})\boldsymbol{s} + \nabla f(\boldsymbol{w})^T\boldsymbol{s} \\ &\text{subject to} \quad \|\boldsymbol{s}\| \leq \Delta_k \end{aligned}$$

  the constraint $\|\boldsymbol{s}\| \leq \Delta_k$ makes the optimization process look for the best 2nd-order-approximating next step only within a trust region which the 2nd-order-approximation is trust to be precise enough. And Δ_k is to be updated if we find out that 2nd order does not approximate the actual function value properly.
- <u>What we do in NM+LS is first find a direction that approximately minimize $\nabla f(\boldsymbol{w}+\boldsymbol{s})$ then decide the step size by back-tracking line search. Trust region method goes the other way, it first decide the step size Δ_k to form a ellipse-shaped trust region (search space) $\|\boldsymbol{s}\| \leq \Delta_k$ within which we believe our second-order approximation is reasonable. Then, within the trust region, we find the step by solving the above convex-optimization problem.</u>
- In my experiment, TRON consume more total time then NM+LS. It is reasonable because with restricted step searching space, trust region method tends to take more step to converge then NM+LS.
- However, with restricted searching space, each iteration tends to consume less time than NM+LS. This explains why TRON cost 100sec/iter whereas NM+LS cost 518sec/iter in average.
- So, choosing between TRON & NM+LS is actually a trade-off between [TRON: Low cost per iteration, slow convergence] and [NM+LS: High cost per iteration, fast convergence]. In our task (i.e., logistic regression on kddb dataset), NM+LS is a more reasonable choice.
- Even though my implementation ended up with lower objective than LIBLINEAR-LS, the training accuracy is actually a little bit lower than LIBLINEAR-LS. This is the effect of regularization term in the objective, it prevent our model from overfitting to purely maximum likelihood of training data.

LIBLINEAR_LINESEARCH vs MY IMPLEMENTATION

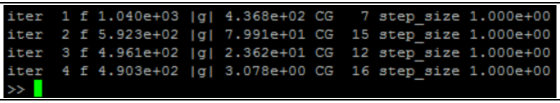
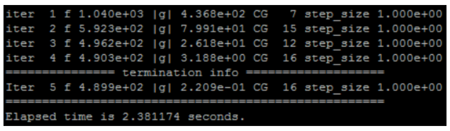
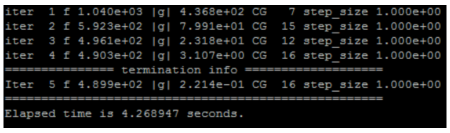
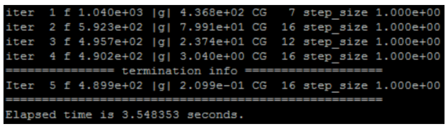
Numerical Issue:

- In Matlab R2015b, matrix multiplication (MM) operator (*) is set to call multithreaded MM algorithm of a highly optimized BLAS, even for sparse MM, it is default to do multithreaded operation too. As shown in the following snap shots, this kind of parallelism does cause a trouble. By starting Matlab with flag –singleCompThread, I force Matlab to do single threaded MM. Comparing MY-MULTI and MY-SINGLE on a small set of data (first 15,000 samples in the training set, I did not evaluate the scenario on full dataset since the time issue, however, it should act the same way in full dataset.), we see that even though the code is remained unchanged, there are still differences between two results. While multithreaded MM get some speed up, it sacrifices the numerical stability and does not guarantee the order of computing.

- Comparing LIBLINEAR and MY-SINGLE, there are still little differences between the results, especially for later iterations. The difference could be caused by the error-propagation issue. I heavily reuse the computation result to save times, every operation that would be called later is cached by temporary variable.
- As CJ Lin mentioned in class, log(1+x) could be very un-precise while x is very small. I changed all the log(1+x) to log1p(x) provided by Matlab to work around numerical problems, however, the results does not change in small dataset.

NOTICE: the following argument have nothing to do with the accuracy of optimizing procedure, I am just trying to reproduce the "exact" same result as LIBLINEAR

- CJ Lin also mentioned, the permutation of data could lead to different result since in numerical computing, (a+b)+c does not guaranteed to be equal to (b+c)+a, the order how operations are performed matters. So I re-permute the data and the gradient seems to be more alike to LIBLINEAR result.

	Snap shot
LIBLINEAR	
MY-MULTI	
MY-SINGLE	
MY-SINGLE (PERMUTED)	

LIBLINEAR:

```
iter  1 f 1.040e+03 |g| 4.368e+02 CG   7 step_size 1.000e+00
iter  2 f 5.923e+02 |g| 7.991e+01 CG  15 step_size 1.000e+00
iter  3 f 4.961e+02 |g| 2.362e+01 CG  12 step_size 1.000e+00
iter  4 f 4.903e+02 |g| 3.078e+00 CG  16 step_size 1.000e+00
>>
```

MY-MULTI:

```
iter  1 f 1.040e+03 |g| 4.368e+02 CG   7 step_size 1.000e+00
iter  2 f 5.923e+02 |g| 7.991e+01 CG  15 step_size 1.000e+00
iter  3 f 4.962e+02 |g| 2.618e+01 CG  12 step_size 1.000e+00
iter  4 f 4.903e+02 |g| 3.188e+00 CG  16 step_size 1.000e+00
=============== termination info ===================
Iter  5 f 4.899e+02 |g| 2.209e-01 CG  16 step_size 1.000e+00
====================================================
Elapsed time is 2.381174 seconds.
```

MY-SINGLE:

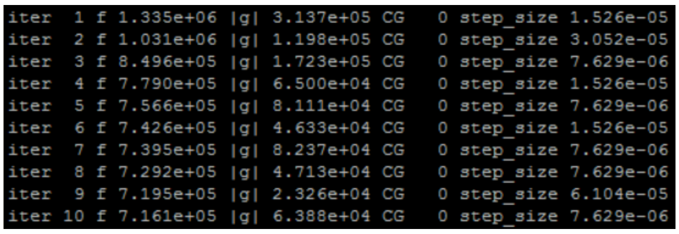
```
iter  1 f 1.040e+03 |g| 4.368e+02 CG   7 step_size 1.000e+00
iter  2 f 5.923e+02 |g| 7.991e+01 CG  15 step_size 1.000e+00
iter  3 f 4.961e+02 |g| 2.318e+01 CG  12 step_size 1.000e+00
iter  4 f 4.903e+02 |g| 3.107e+00 CG  16 step_size 1.000e+00
=============== termination info ===================
Iter  5 f 4.899e+02 |g| 2.214e-01 CG  16 step_size 1.000e+00
====================================================
Elapsed time is 4.268947 seconds.
```

MY-SINGLE (PERMUTED):

```
iter  1 f 1.040e+03 |g| 4.368e+02 CG   7 step_size 1.000e+00
iter  2 f 5.923e+02 |g| 7.991e+01 CG  16 step_size 1.000e+00
iter  3 f 4.957e+02 |g| 2.374e+01 CG  12 step_size 1.000e+00
iter  4 f 4.902e+02 |g| 3.040e+00 CG  16 step_size 1.000e+00
=============== termination info ===================
Iter  5 f 4.899e+02 |g| 2.099e-01 CG  16 step_size 1.000e+00
====================================================
Elapsed time is 3.548353 seconds.
```

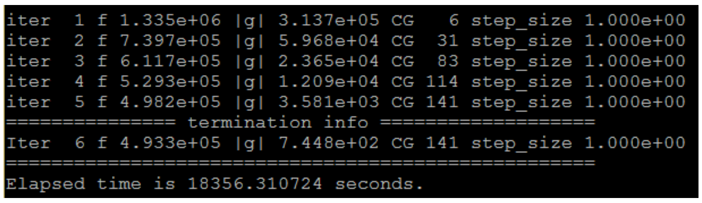
➢ I print out the final objective and $\|\nabla f(\boldsymbol{w}^{final})\|$ while in the implementation of LIBLINEAR does not. (ignore CG & step_size in termination info, we don't perform CG once stopping criterion is satisfied.)

==============================Gradient Descent=================================



```
iter  1 f 1.335e+06 |g| 3.137e+05 CG   0 step_size 1.526e-05
iter  2 f 1.031e+06 |g| 1.198e+05 CG   0 step_size 3.052e-05
iter  3 f 8.496e+05 |g| 1.723e+05 CG   0 step_size 7.629e-06
iter  4 f 7.790e+05 |g| 6.500e+04 CG   0 step_size 1.526e-05
iter  5 f 7.566e+05 |g| 8.111e+04 CG   0 step_size 7.629e-06
iter  6 f 7.426e+05 |g| 4.633e+04 CG   0 step_size 1.526e-05
iter  7 f 7.395e+05 |g| 8.237e+04 CG   0 step_size 7.629e-06
iter  8 f 7.292e+05 |g| 4.713e+04 CG   0 step_size 7.629e-06
iter  9 f 7.195e+05 |g| 2.326e+04 CG   0 step_size 6.104e-05
iter 10 f 7.161e+05 |g| 6.388e+04 CG   0 step_size 7.629e-06
```

=============================Newton's Method==================================



```
iter  1 f 1.335e+06 |g| 3.137e+05 CG   6 step_size 1.000e+00
iter  2 f 7.397e+05 |g| 5.968e+04 CG  31 step_size 1.000e+00
iter  3 f 6.117e+05 |g| 2.365e+04 CG  83 step_size 1.000e+00
iter  4 f 5.293e+05 |g| 1.209e+04 CG 114 step_size 1.000e+00
iter  5 f 4.982e+05 |g| 3.581e+03 CG 141 step_size 1.000e+00
=============== termination info ===================
Iter  6 f 4.933e+05 |g| 7.448e+02 CG 141 step_size 1.000e+00
====================================================
Elapsed time is 18356.310724 seconds.
```