Notes on CG and LM-BFGS Optimization of Logistic Regression

Hal Daumé III
Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
hdaume@isi.edu

1 Introduction

It has been recognized that the typical iterative scaling methods [BDD96, Ber97] used to train logistic regression classification models (maximum entropy models) are quite slow. Goodman has suggested the use of a component-wise optimization of GIS [Goo02], which he has measured to be faster on many tasks. However, in general, the iterative scaling methods pale in comparison to conjugate gradient ascent (for binary problems) and limited memory BFGS for multiclass problems [Min01, Min03, Mal02]. Unfortunately, while these methods are typically algorithmically more efficient than the iterative scaling algorithms, they are also significantly more difficult to implement (especially LM-BFGS). This paper describes one particular implementation that is known to be quite fast, and has gone through several iterations of optimization. The actual implementation can be downloaded from `http://www.isi.edu/~hdaume/megam/` and may be used freely for any research purposes[1]. The intended audience of this paper is quite technical: I assume you know what logistic regression/maximum entropy models are, I assume you know what gradients and Hessians are, etc.

2 Notation

We will assume throughout that we have N many training iid data points, $\boldsymbol{x}_n$ and their corresponding classes y_n. Each data point will have F many components, denoted x_{nf}. For binary problems, we assume $y_n \in \{-1, +1\}$ and for multiclass problems, we assume $y_n \in \{1, 2, \dots, C\}$ where C is the total number of classes. We will use a Gaussian prior on weights with precision (inverse variance) λ, and will denote our weight vector $\boldsymbol{w}$ of length F. Dot products will be written using matrix notation, eg $\boldsymbol{w}^\top \boldsymbol{w}$ will be the 2-norm of $\boldsymbol{w}$; correspondingly, $\boldsymbol{w}\boldsymbol{w}^\top$ will denote an $F \times F$ matrix. We will typically denote gradients by $\boldsymbol{g}$, a vector of length F, and Hessians by $\mathbf{H}$, a square matrix with dimension F. In general, subscripts will the the lower-case version of their upper bound, and vectors will be indexed from 1 (i.e., $\sum_{f=1}^{F} x_{nf}$ or $\prod_{n=1}^{N} y_n$). In such cases, we will typically leave off the upper and lower

[1] Suitable acknowledgment is appreciated, either in the form of a footnote or a reference to this paper (a bibtex entry can be found on the web page); If you wish to use the software for commercial/non-research purposes, please contact me.

bounds from the sum or product to simplify notation. In general, if there is a vector $\boldsymbol{v}$ that changes over iterations, $\boldsymbol{v}'$ will refer to the value at the current iteration, and $\boldsymbol{v}$ will refer to the value at the previous iteration (though our algorithms will explicitly update these).

3 Conjugate Gradient Ascent

The basic idea in CG is to select our search direction so that it is perpendicular to the search direction from the previous iteration; see [Min03] for further details. In particular, if $\boldsymbol{u}$ is an arbitrary direction, we update $\boldsymbol{w}$ by:

$$\boldsymbol{w}' \leftarrow \boldsymbol{w} + \frac{\boldsymbol{g}^\top \boldsymbol{u}}{\lambda \boldsymbol{u}^\top \boldsymbol{u} + \sum_n \sigma\left(\boldsymbol{w}^\top \boldsymbol{x}_n\right) \sigma\left(-\boldsymbol{w}^\top \boldsymbol{x}_n\right) \left(\boldsymbol{u}^\top \boldsymbol{x}_n\right)^2} \boldsymbol{u}$$

and σ is the logistic function, $\sigma(a) = (1 + \exp -a)^{-1}$; the gradient is given by:

$$\boldsymbol{g} = -\lambda \boldsymbol{w} + \sum_n \sigma\left(-y_n \boldsymbol{w}^\top \boldsymbol{x}_n\right) y_n \boldsymbol{x}_n$$

We choose u according to $\boldsymbol{u}' \leftarrow \boldsymbol{g} - \beta \boldsymbol{u}$, where a good value of β is according to the Hestenes-Stiefel formula:

$$\beta = \frac{\boldsymbol{g'}^\top \left(\boldsymbol{g}' - \boldsymbol{g}\right)}{\boldsymbol{u}^\top \left(\boldsymbol{g}' - \boldsymbol{g}\right)}$$

As can be observed, the value $\boldsymbol{w}^\top \boldsymbol{x}_n$ appears quite frequently in all expressions. Implementationally, it is very important to *cache* this value and update it between iterations, rather than constantly recompute it. My recommended implementation is shown in Figure 1. In general with the problems we work with, each $\boldsymbol{x}_n$ will be sparse, and will typically be implemented by storing two arrays, one for indices and one for feature values (note that the indices need not be sorted). In the algorithm, we need to compute both dense dot products, and dot products between sparse vectors $\boldsymbol{x}_n$ and dense vectors. Both of these can be implemented efficiently, the first in a direct sum/loop, the latter in a loop over the indices of $\boldsymbol{x}_n$, summing the corresponding components of the dense vector. Note that in the computation of β in the algorithm, we do not need to actually construct a new vector $\boldsymbol{g'} - g$, but can rather take the dot product implicitly. Finally, it is recommended that the memory for g' be allocated outside the loop, so we do not waste time doing so inside.

4 Limited Memory BFGS

For multiclass problems, it becomes impossible to explicitly construct and invert the Hessian matrix. In the binary instance, the Hessian had a simple form the enabled simple analytic inversion; the alternative used in LM-BFGS is to use only an approximation to the true Hessian, and to build this approximation up iteratively. In particular, we will approximate the Hessian at iteration i using the previous M values of the weight vector and of the gradient (of course, when $i < M$, we only use the $i - 1$ most recent).

In order to facilitate the use of such memories, we introduce a new data structure that contains three arrays of length exactly M (doubly linked lists would also work),

Algorithm CG($\boldsymbol{x}, \boldsymbol{y}, \lambda$)
Initialize $\boldsymbol{w} \leftarrow \langle 0 \rangle_F$, $\mathbf{wtx} \leftarrow \langle 0 \rangle_N$, $\boldsymbol{g} \leftarrow \langle 0 \rangle_F$, $\boldsymbol{u} \leftarrow \langle 0 \rangle_F$
while not converged **do**
  $\boldsymbol{g}' \leftarrow -\lambda \boldsymbol{w}$
  for $n = 1 \dots N$ **do**
    $\boldsymbol{g}' \leftarrow \boldsymbol{g}' + \sigma\left(-y_n \text{ wtx}[n]\right) y_n \boldsymbol{x}_n$
  end for
  $\beta \leftarrow \left(\boldsymbol{g}'^\top \left(\boldsymbol{g}' - \boldsymbol{g}\right)\right) / \left(\boldsymbol{u}^\top \left(\boldsymbol{g}' - \boldsymbol{g}\right)\right)$
  $\boldsymbol{u} \leftarrow \boldsymbol{g} - \beta \boldsymbol{u}$
  $z \leftarrow \left(\boldsymbol{g}'^\top \boldsymbol{u}\right) / \left(\lambda \boldsymbol{u}^\top \boldsymbol{u} + \sum_n \sigma\left(\text{ wtx}[n]\right) \sigma\left(-\text{ wtx}[n]\right) \left(\boldsymbol{u}^\top \boldsymbol{x}_n\right)^2\right)$
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + z\boldsymbol{u}$
  for $n = 1 \dots N$ **do**
    $\text{wtx}[n] \leftarrow \text{ wtx}[n] + z \boldsymbol{u}^\top \boldsymbol{x}_n$
  end for
  $\boldsymbol{g} \leftarrow \boldsymbol{g}'$
end while
return $\boldsymbol{w}$

Figure 1: The full training algorithm for conjugate gradient ascent.

one for the old weight vectors, one for the old gradient vectors, and one for a scalar. The data structure should support a *push* operation that adds a new vector pair and scalar to the memory, overwriting the oldest if necessary. We also need to be able to iterate through the memory both from most recent back, and from least recent forward. This is implemented in the data structure `memory` in the implementation referenced in the introduction.

In LM-BFGS, we take steps according to:

$$\boldsymbol{w}' \leftarrow \boldsymbol{w} - \eta \mathbf{H} \boldsymbol{g}$$

where η is a step size parameter. For $\eta = 1$, this is a Newton step; however, step sizes < 1 are useful and so we use a line search algorithm [PFTV02] to find it (this is described later). The LM-BFGS trick is to be able to compute $\mathbf{H}\boldsymbol{g}$ in a reasonable amount of time, using our memory. This trick is described in general terms in [NN91, AM94] and will simply be used in our algorithm, specialized to the case of logistic regression.

As in CG, it is important to cache the dot product of the weights with the feature vectors. However, in this case, storing such values takes a matrix of size $N \times C$, since we need to store it for each2 class. Nevertheless, doing so is quite imperative to efficient optimization; otherwise, all execution time is spent on function evaluation. Additionally, we also compute a value qtx which stores the dot product of the change in weight by x, also of size $N \times C$. The algorithm for LM-BFGS optimization is depicted in Figure 5. This requires three subroutines, ComputeGradient, ComputePosterior and LineSearch. These are depicted in Figures 2, 3 and 4, respectively.

The main LM-BFGS algorithm essentially performs one one iteration without using any Hessian information (the part before the while loop) and then begins the loop, using previous iteration's gradients. At each iteration it finds a step parameter η by calling the LineSearch algorithm. In our experience, this algorithm terminates after zero, one or two iterations, and its computations are very inexpensive, so there is no need to use a more complex line search. The notation $\text{mem}_d[m]$ means the

Algorithm ComputeGradient($\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}$, wtx)
$\boldsymbol{g} \leftarrow -\lambda \boldsymbol{w}$
for $n = 1, \ldots, N$ **do**
  $z \leftarrow \bigoplus_c \mathrm{wtx}[n, c]$
  for $c = 1, \ldots, C$ **do**
    $g \leftarrow g + (\delta_{c,y_n} - \exp(\mathrm{wtx}[n, c] - z))\, \boldsymbol{x}_{nc}$
  end for
end for
return $\boldsymbol{g}$

Figure 2: The ComputeGradient subroutine required for LM-BFGS.

Algorithm ComputePosterior($\lambda, \boldsymbol{y}, \boldsymbol{w}, \boldsymbol{q}$, wtx, qtx, η)
$p \leftarrow -\lambda/2 \left(\boldsymbol{w}^\top \boldsymbol{w} + \eta^2 \boldsymbol{q}^\top \boldsymbol{q} + 2\eta \boldsymbol{q}^\top \boldsymbol{w}\right)$
for $n = 1, \ldots, N$ **do**
  $s \leftarrow -\infty$
  for $c = 1, \ldots, C$ **do**
    $\chi \leftarrow \mathrm{wtx}[n, c] + \eta \mathrm{qtx}[n, c]$
    $s \leftarrow s \oplus \chi$
    if $c = y_n$ **then**
      $p \leftarrow p + \mathrm{wtx}[n, c]$
    end if
  end for
  $p \leftarrow p - s$
end for
return p

Figure 3: The ComputePosterior subroutine required for LineSearch.

mth vector d to be pushed into memory, where $m = M$ means the most recent and $m = 1$ means the least recent; the other subscripts are the same. Again, it is advantageous to allocate all memory outside any loops and compute using only existing arrays. In fact, we can gain some memory savings by storing the vectors $\boldsymbol{d}$ and $\boldsymbol{u}$ in the approximate Hessian computation in place of $\boldsymbol{g}$, and then restoring it later (see my implementation for this slight trick).

The computation of the gradient and posterior make use of the $\oplus$ operator, which is defined to be addition of values in log-space. This can be implemented efficiently and each $\oplus$ operation requires a logarithm and exponentiation computation. $\bigoplus$ is simply a $\sum$-sum using $\oplus$ instead of $+$.

In the ComputePosterior algorithm, within one line search, the values $\boldsymbol{w}^\top \boldsymbol{w}$, $\boldsymbol{q}^\top \boldsymbol{q}$ and $\boldsymbol{q}^\top \boldsymbol{w}$ will always be the same, so it is recommended that these are cached outside of the ComputePosterior and passed in as arguments (this is done in my implementation as well).

Finally, the line search is a simple backtracking line search [PFTV02]. This uses the technique of modeling the (negative log) posterior by a cubic and explicitly maximizing it each time. Note that the maximization need not converge – we only need a value of η that attains sufficient decrease. In the full LM-BFGS algorithm, if the value η returned is ever zero, then the iterations need to stop (if this is not done, then on the next iteration things will blow up).

5 Summary

I have described efficient implementations of the conjugate gradient and limited memory BFGS methods for optimizing logistic regression classifiers. I have made available a public implementation of these methods to demonstrate their effectiveness on real world problems. The algorithms described herein are completely self-contained and require no digging through literature to find sub-components. This was done, perhaps, at a slight loss in generality, but the reader is directed to [Min03, AM94] for more general details. It is my sincere hope that these notes and the implementation are helpful to some users.

References

[AM94] Brett M. Averick and Jorge J. Moré. Evaluation of large-scale optimization problems on vector and parallel architectures. *SIAM Journal of Optimization*, 4, 1994.

[BDD96] Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[Ber97] A. Berger. The improved iterative scaling algorithm: A gentle introduction, 1997.

[Goo02] Joshua Goodman. Sequential conditional generalized iterative scaling. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, 2002.

[Mal02] Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of CoNLL*, 2002.

[Min01] Thomas P. Minka. Algorithms for maximum-likelihood logistic regression. Technical Report 758, Carnegie Mellon University, 2001.

[Min03] Thomas P. Minka. A comparison of numerical optimizers for logistic regression. `http://www.stat.cmu.edu/~minka/papers/logreg/`, 2003.

[NN91] S.G. Nash and J. Nocedal. A numerical study of the limited memory BFGS method and the truncated Newton method for large scale optimization. *SIAM Journal of Optimization*, 1:358–372, 1991.

[PFTV02] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, second edition, January 2002.

```
Algorithm LineSearch(λ, y, wtx, w, q, g, qtx)
τ ← min{1, 100/√(q⊤q)}
f_old ← ComputePosterior(λ, y, q, wtx, qtx, 0)
γ ← τ g⊤ q
if γ < 0 then
  return 0
end if
η_min ← 1e-10/ max_f ((τ abs q_f)/ max{(abs w_f), 1})
η ← 1, η_old ← 0, f_2 ← f_old
while true do
  if τη < η_min then
    return 0
  else
    f ← ComputePosterior(λ, y, q, wtx, qtx, ητ)
    if f ≥ f_old + 1e-4τηγ then
      return τη
    else if abs(η − 1) < 1e-20 then
      η_tmp ← γ/(2(f_old + γ − f))
      η_old ← η, η ← max{η_tmp, (η/10)}, f_2 ← f
    else
      r_1 ← f − f_old − γη
      r_2 ← f_2 − f_old − γη_old
      a ← (r_1/η^2 − r_2/η_old^2)/(η − η_old)
      b ← (ηr_2/η_old^2 − η_old r_1/η^2)/(η − η_old
      η_tmp ← 0
      if abs a < 1e-20 then
        η_tmp ← −γ/(2b)
      else
        d ← b^2 − 3aγ
        if d < 0 then
          η_tmp ← η/2
        else if b <= 0 then
          η_tmp ← (√d − b)/(3a)
        else
          η_tmp ← −γ/(b + √d)
        end if
      end if
      η_tmp ← min{η_tmp, η/2}
      η_old ← η, η ← max{η_tmp, (η/10)}, f_2 ← f
    end if
  end if
end while
return τη
```

Figure 4: The LineSearch subroutine required for LM-BFGS.

Algorithm LM-BFGS$(\boldsymbol{x}, \boldsymbol{y}, \lambda)$
Initialize $\boldsymbol{w} \leftarrow \langle 0 \rangle_F$, $\text{wtx} \leftarrow \langle 0 \rangle_{N \times C}$
$\boldsymbol{g} \leftarrow$ComputeGradient$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, \text{ wtx})$
$\boldsymbol{q} \leftarrow \boldsymbol{g} / \sqrt{\boldsymbol{g}^\top \boldsymbol{g}}$
$\text{qtx} \leftarrow \boldsymbol{q}^\top \boldsymbol{x}$
$\eta \leftarrow$LineSearch$(\lambda, \boldsymbol{y}, \text{ wtx}, \boldsymbol{w}, \boldsymbol{q}, \boldsymbol{g}, \text{ qtx})$
for $n = 1 \ldots N, c = 1 \ldots C$ **do**
  $\text{wtx}[n, c] \leftarrow \text{ wtx}[n, c] + \eta \text{ qtx}[n, c]^\top \boldsymbol{x}_{nc}$
end for
$\boldsymbol{w}' \leftarrow \boldsymbol{w} + \eta \boldsymbol{q}$
Initialize $\text{mem} \leftarrow \emptyset$
while not converged **do**
  $\boldsymbol{g}' \leftarrow$ComputeGradient$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, \text{ wtx})$
  $\alpha \leftarrow (\boldsymbol{g}' - \boldsymbol{g})^\top (\boldsymbol{w}' - \boldsymbol{w})$
  $\sigma \leftarrow (\boldsymbol{g}' - \boldsymbol{g})^\top (\boldsymbol{g}' - \boldsymbol{g})$
  Push $\boldsymbol{d} = (\boldsymbol{w}' - \boldsymbol{w})$, $\boldsymbol{u} = (\boldsymbol{g}' - \boldsymbol{g})$ and α onto mem
  $\boldsymbol{q} \leftarrow \boldsymbol{g}'$
  $\boldsymbol{\beta} \leftarrow \langle 0 \rangle_M$
  for $m = M, \ldots, 1$ **do**
    $\beta[m] \leftarrow (\text{mem}_d[m]) / (\text{mem}_\alpha[m])$
    $\boldsymbol{q} \leftarrow \boldsymbol{q} - \beta[m] (\text{mem}_u[m])$
  end for
  $\boldsymbol{q} \leftarrow \sigma \boldsymbol{q}$
  for $m = 1, \ldots, M$ **do**
    $\zeta \leftarrow (\text{mem}_u[m])^\top \boldsymbol{q}$
    for $f = 1, \ldots, F$ **do**
      $\xi \leftarrow (\text{mem}_d[m, f]) (\beta[m] - \zeta / (\text{mem}_a[m]))$
      $q[f] \leftarrow q[f] + \xi$
      $\zeta \leftarrow \zeta + \xi$
    end for
  end for
  $\boldsymbol{q} \leftarrow -q$
  $\text{qtx} \leftarrow \boldsymbol{q}^\top \boldsymbol{x}$
  $\eta \leftarrow$LineSearch$(\lambda, \boldsymbol{y}, \text{ wtx}, \boldsymbol{w}, \boldsymbol{q}, \boldsymbol{g}, \text{ qtx})$
  for $n = 1 \ldots N, c = 1 \ldots C$ **do**
    $\text{wtx}[n, c] \leftarrow \text{ wtx}[n, c] + \eta \text{ qtx}[n, c]^\top \boldsymbol{x}_{nc}$
  end for
  $\boldsymbol{w}' \leftarrow \boldsymbol{w} + \eta \boldsymbol{q}$
  $\boldsymbol{g} \leftarrow \boldsymbol{g}'$
end while
return $\boldsymbol{w}$

Figure 5: The full training algorithm for limited memory BFGS.