Fast Robust Logistic Regression for Large Sparse Datasets with Binary Outputs

Paul R. Komarek
Dept. of Mathematical Sciences
Carnegie Mellon University
Pittsburgh, PA 15213

Andrew W. Moore
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Although popular and extremely well established in mainstream statistical data analysis, logistic regression is strangely absent in the field of data mining. There are two possible explanations of this phenomenon. First, there might be an assumption that any tool which can only produce linear classification boundaries is likely to be trumped by more modern nonlinear tools. Second, there is a legitimate fear that logistic regression cannot practically scale up to the massive dataset sizes to which modern data mining tools are applied. This paper consists of an empirical examination of the first assumption, and surveys, implements and compares techniques by which logistic regression can be scaled to data with millions of attributes and records. Our results, on a large life sciences dataset, indicate that logistic regression can perform surprisingly well, both statistically and computationally, when compared with an array of more recent classification algorithms.

1 Introduction

This paper asks a simple question:

> Given the wide variety of recent classification algorithms (such as support vector machines) how does one of the oldest—logistic regression—measure up?

We investigate logistic regression's (LR) performance as a statistical method, in which the first concern is accurate modeling of the monomial PDF of binary-valued output conditioned on a set of inputs. We also investigate LR as a data mining method, in which the first concern is computational tractability on large datasets with millions of records and hundreds of thousands of attributes.

This paper concludes that there are circumstances in which LR equals or exceeds the predictive performance of more recent algorithms. We show how the application of well-established sparse conjugate gradient approaches in the inner loop allows the scaling of LR to sparse datasets with the aforementioned dimensions, as well as large dense datasets with thousands of dimensions.

2 Terminology

When discussing datasets in this paper, each record belongs to or is predicted to belong to either the positive or negative class. A *positive row* is a row belonging to, or predicted to belong to, the positive class. A similar definition holds for a *negative row*. The number of rows in a dataset is denoted R, while the number of attributes is M. The *sparsity factor* of a dataset is the proportion of nonzero entries, and is written F. The total number of nonzero elements in a dataset is the product MRF.

3 Logistic Regression

Logistic regression gets its name from the logistic curve it uses to model the expected value of zero-one outputs. Suppose our data X has R rows and M attributes. Let $\mathbf{x_i}$ denote row i from the dataset, β denote the $M+1$ model parameters, and let μ_i be the model's expected value for the output associated with row $\mathbf{x_i}$. Then the logistic curve may be written as

$$\mu_\mathbf{i} = \frac{\exp(\beta_0 + \beta_1 \mathbf{x_{i}}_1 + \cdots \beta_M \mathbf{x_{i}}_M)}{1 + \exp(\beta_0 + \beta_1 \mathbf{x_{i}}_1 + \cdots \beta_M \mathbf{x_{i}}_M)}$$

and in a single variable appears as in figure 1. Notice that μ_i is strictly between zero and one. Let $\mathbf{y}$ be the vector of R outputs. The maximum likelihood equations for this model are

$$\sum_{i=1}^{R} (\mathbf{y}_i - \mu_i) = 0 \quad (1)$$

$$\sum_{i=1}^{R} \mathbf{x_{i}}_j (\mathbf{y}_i - \mu_i) = 0, \; j = 1 \ldots M \quad (2)$$

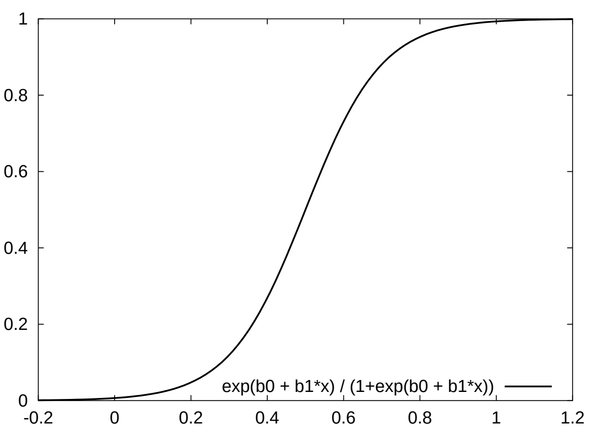

Figure 1: Logistic Model

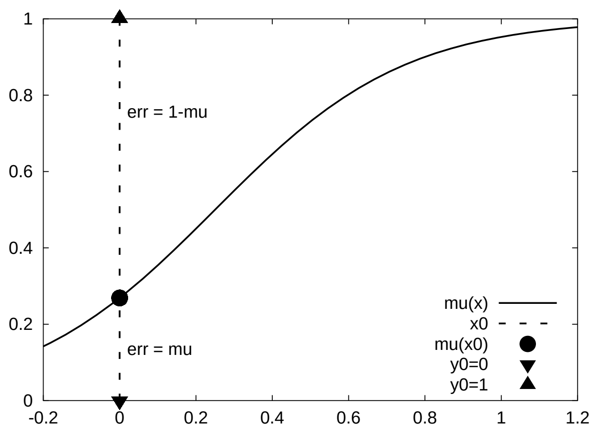

Figure 2: Logistic Model Residuals Vary With x

It is common to use an Iteratively Re-weighted Least Squares (IRLS) algorithm to solve the above maximum likelihood equations. In such an algorithm there is an outer loop which evaluates the current expected values $\mu = (\mu_1, \ldots, \mu_M)^T$ and prediction errors $(\mathbf{y} - \mu)$, and creates linear models which approximate the logistic model. These linear models have a diagonal weight matrix $\mathbf{W}$ with diagonal entries $\mu_i(1-\mu_i)$ and adjusted dependent variables

$$\eta_i = \mathbf{x_i}^T \beta + \frac{y_i - \mu_i}{\mu_i(1-\mu_i)},$$

where β is the previous guess for the parameters. From this model a new guess for β is computed using weighted linear regression, as shown in equation (3).

$$\beta = \left(\mathbf{X^T W X}\right)^{-1} \mathbf{X^T W} \eta \tag{3}$$

The new parameter vector β is then used to update the expected values μ and prediction errors $(\mathbf{y} - \mu)$. This is the end of the outer loop.

The residual errors in the logistic regression model are binomially distributed with variance changing throughout the domain. This may be seen in figure 2. Suppose the model assigns to a point x in the domain the probability μ of belonging to the positive class. The observed class for this point will be positive or negative, yielding a residual error of $(1-\mu)$ or μ respectively. Thus the residual error depends on x.

The outer loop of the algorithm described above may be terminated on any of several conditions, depending on numerical characteristics of the data and personal preference. Perhaps the most common convergence criterion is stabilization of an error measurement known as *deviance*. The deviance DEV may be computed as

$$\text{DEV} = \sum_{i=1}^{R} y_i \ln\left(\frac{y_i}{\mu_i}\right) + (1-y_i)\ln\left(\frac{1-y_i}{1-\mu_i}\right)$$

It may also be desirable to check for the stabilization of the expected values μ. With some datasets, it is possible for excessive iterations to create numerical problems (McIntosh 1982 [8], McCullagh 1989 [7]). For this reason one might create customized termination criteria for the logistic regression outer loop.

It is important to note the difference between logistic regression and least-squares methods with a sigmoid or logistic model. Logistic regression is a type of generalized linear model with an error function chosen to reflect the changing variance of the binomially-distributed residual errors. Applications which use a least-squares or similar criterion for such models ignore the nonuniform error variance across the domain. This may or may not matter for the performance of an ad-hoc function estimator like neural networks (Watrous 1986 [13], Kramer 1989 [6]). However, such an approach eschews the statistical foundations that give credibility to logistic regression.

Logistic regression is well-studied and is supported by a rich body of literature. For more details, one might wish to read McCullagh 1989 ([7]) or Hosmer 2000 ([4]). Both theoretical and numerical issues are covered in these works.

4 Fast Robust Logistic Regression

When applying logistic regression to large, sparse real-world data, we've encountered two principal problems. The first problem is numerical instability and the second is scalability or speed. Below we present four simple approaches to these problems.

4.1 Simple Algorithm

Solving the weighted linear model in the IRLS algorithm discussed in section 3 requires direct or indirect evaluation of $(\mathbf{X^T W X})^{-1}\mathbf{X^T W}\eta$. Because the weight matrix $\mathbf{W}$ is nonnegative and diagonal, if $\mathbf{X^T}$ has full column rank then

$\mathbf{X^TWX}$ is symmetric positive definite (SPD). This suggests an obvious method for computing the parameters β, which will be called the Simple Algorithm:

1. Remove linear dependencies from the data $\mathbf{X}$
2. Rewrite equation (3) as $\mathbf{X^TWX}\beta = \mathbf{X^TW}\eta$
3. In each outer loop iteration, compute the Cholesky factorization $\mathbf{LL^T}$ of $\mathbf{X^TWX}$
4. Solve for β using an efficient back-substitution.

This approach is attractive because of the stability of the Cholesky decomposition on a wide variety of matrices.

It is an assumption of this paper that the structure of the sparsity in the data cannot be exploited to prevent fill-in in the course of an algorithm such as Gaussian elimination. For this reason, preprocessing is likely to require approximately $O\left(\min(M^2R, R^2M)\right)$, using the notation from section 2.

Suppose the data has no linear dependencies and has dimensions $R \times M$. If the data is sparse, the complexity of this approach is $O\left(M^2RF + M^3\right)$. The second term, M^3, represents a standard Cholesky decomposition of $\mathbf{X^TWX}$. If the data is dense, the complexity of the Simple Algorithm is $O\left(M^2R + M^3\right)$. To use a standard Cholesky decomposition, the input data must have full column rank and hence $M < R$. Therefore the complexity is $O\left(M^2RF\right)$, with $F = 1.0$ for dense datasets.

Simple optimizations can be made which improve the speed but not the complexity. For instance, if the data is sparse then the construction of the $\mathbf{X^TWX}$ on each LR iteration can be made more efficient by pre-caching the intersections of all columns $\mathbf{x_i}, \mathbf{x_j}$ of the data $\mathbf{X}$. Such an "intersection cache" can require large amounts of storage; for a 5000 attribute dataset with an average of 25 common values between columns, the intersection cache will be over one gigabyte. This cache effectively eliminates the cost of constructing $\mathbf{X^TWX}$. Typically there are ten iterations of the LR outer loop per fold while doing 10-fold cross validation.

It should be clear that the Simple Algorithm will not scale well as dataset size grows. It does not fully exploit sparseness during the Cholesky decomposition. Use of a limited fill-in Cholesky decomposition which did not require construction of the $\mathbf{X^TWX}$ matrix would reduce the time complexity of the Simple Algorithm, though simultaneously making it an approximate algorithm. However, in our experiments numerical issues made the $\mathbf{X^TWX}$ matrix appear indefinite to the Cholesky decomposition, preventing solution of equation (3).

4.2 Robust Cholesky Algorithm

It is possible to combine linear dependency removal and Cholesky decomposition. Just as the Cholesky decomposition can be used to test if a matrix is SPD, a small modification allows it to 'find' a SPD submatrix of a symmetric input matrix. Suppose we are computing the Cholesky decomposition of a symmetric matrix $\mathbf{A}$ to produce the lower triangular Cholesky factor $\mathbf{L}$. For logistic regression, $\mathbf{A} = \mathbf{X^TWX}$. The off-diagonal entries of the Cholesky factor $\mathbf{L}$ may be written in closed form as

$$\mathbf{L}_{i,j} = \frac{\mathbf{A}_{i,j} - \sum_{k<j}\mathbf{L}_{i,k}\mathbf{L}_{j,k}}{\mathbf{L}_{j,j}},\ j < i$$

while the diagonal entries may be written as

$$\mathbf{L}_{i,i} = \sqrt{\mathbf{A}_{i,i} - \sum_{k<i}\mathbf{L}_{i,k}^2} \tag{4}$$

The input matrix $\mathbf{A}$ is SPD, if and only if all the diagonal entries of $\mathbf{L}$ are well-defined and positive.

Our modification to the standard Cholesky decomposition, above, is nearly as simple as discarding rows and columns from the input matrix $\mathbf{A}$ when the associated diagonal entry is zero or not defined. Note that discarding row i and column i of $\mathbf{A}$ or $\mathbf{L}$ is equivalent to discarding attribute i from the data $\mathbf{X}$. Suppose that $\mathbf{A}$ is SPD, but numerical problems cause the diagonal entry $\mathbf{L}_{i,i}$ to be nonpositive. This indicates that the numerator of the right-hand-side of equation (4) is negative, hence the summation term in the numerator is too large. Every element of $\mathbf{L}$ in this errant summation has been used successfully in previous rows, except the values from row $i-1$. Therefore we recompute row i of $\mathbf{L}$ as if row and column $i-1$ of $\mathbf{A}$ did not exist. If the newly computed diagonal entry is positive, we mark row and column $i-1$ of the input matrix as bad and continue. If the newly computed diagonal entry is negative, we assume that row i is bad and discard it instead.

We developed this robust Cholesky decomposition because removing linear dependencies from the columns of our datasets did not prevent numerical problems in a standard Cholesky decomposition of $\mathbf{X^TWX}$. We observed that this robust Cholesky decomposition removed nearly the same set of attributes as did Gaussian elimination, and the resulting predictions were better. The use of this robust Cholesky implementation inside our LR outer loop on the original unprocessed data will be called the Robust Cholesky Algorithm.

The time complexity of this method is similar to that of the Simple Algorithm in section 4.1. The only difference is that there is no need for preprocessing. However, the cost of removal of linear dependencies has only been shifted to the first iteration of LR when the robust Cholesky decomposition is run. In fact, this occurs in the first LR iteration of every fold. The Robust Cholesky Algorithm overcomes numerical issues but fails to exploit sparseness.

4.3 Stepwise Logistic Regression

Another approach to handling problematic datasets is *stepwise* logistic regression. This procedure wraps LR's outer loop with a basis-choosing loop. The basis-choosing step creates a subset $\tilde{\mathbf{X}}_\mathbf{0}$ of the data $\mathbf{X}$, runs the LR outer loop on this subset, and uses the result to choose a new data subset $\tilde{\mathbf{X}}_\mathbf{1}$. This repeats until the basis-choosing loop has met some termination criterion. This algorithm will be called the Stepwise Algorithm. The Stepwise Algorithm is attractive because no preconditioning of the data is needed, and numerical problems are handled dynamically by the basis-choosing loop. The basis-choosing step can make use of prior knowledge about the attributes, or use generic measures like information gain.

The complexity of this algorithm depends on the basis-choosing step and the number of basis-choosing iterations. Once a basis is chosen, the solution to the logistic regression problem for that basis must be found. We believe that attractiveness of logistic regression for large sparse datasets is not materially affected by the Stepwise Algorithm.

4.4 Conjugate Gradient as an Inner Loop

Nearly all of the time spent in LR iterations is on solving the linear regression. In particular, that time is spent constructing $\mathbf{A} = \mathbf{X}^\mathbf{T}\mathbf{W}\mathbf{X}$ and finding its Cholesky factors. By using a conjugate gradient (CG) iterative linear solver for equation (3), we avoid construction of $\mathbf{A} = \mathbf{X}^\mathbf{T}\mathbf{W}\mathbf{X}$ altogether. The algorithm based on this technique will be called the CG Algorithm.

A thorough explanation of conjugate gradient and related methods can be found in Greenbaum 1997 ([3]), Nash 1996 ([10]), or Bishop 1995 ([1]). Briefly, solving a linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is SPD, using conjugate gradient is similar to minimizing the quadratic expression

$$\frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{x} + \mathbf{b}^\mathbf{T}\mathbf{x} \tag{5}$$

using Newton's method with a specially chosen set of direction vectors. CG can be thought of as a specialization of Newton's method for quadratic functions with SPD matrices, but this ignores the features most important to our application. Those features are that conjugate gradient does not require explicit computation of the Hessian of expression (5), as Newton's method does, that CG can perform well when the matrix $\mathbf{A}$ is rank deficient and hence SPD, and that typically few CG iterations are necessary. With perfect arithmetic, the exact solution to a linear system with SPD matrix $\mathbf{A}$ having dimensions $M \times M$ can be found in exactly M iterations.

In practice, a termination parameter is used to control the quality of approximation. We will refer to this parameter later as *epsilon*. The CG algorithm runs until an error measurement is smaller than epsilon. This error measurement is the distance between the current solution and the optimal solution, under the assumption that one is finding minima of expression (5). Therefore this error measurement is also appropriate for the equivalent problem of solving a linear systems with SPD matrix $\mathbf{A}$.

Like Newton's method, CG is well-studied. A modified CG method known as the *biconjugate gradient method* is designed for matrices $\mathbf{A}$ in expression (5) which are not SPD. To reduce the number of iterations needed for convergence of the CG method, or speed up each iteration, a *preconditioning matrix* can be applied to both sides of the linear system $\mathbf{Ax} = \mathbf{b}$. There is a significant literature on choosing these preconditioners for use with the CG method, sometimes using the name *preconditioned conjugate gradient*. A good reference on the application of iterative methods to linear systems is Greenbaum 1997 ([3]). We have had significant success in our work with the basic CG algorithm, and hope for improved speed and predictive performance with greater sophistication in our inner loop.

The time complexity of the CG Algorithm is $O(MRF \cdot \texttt{maxiters})$, where `maxiters` is an upper-bound placed on the number of CG iterations used to approximate a solution to the linear system (3). In the experiments found in this paper, fewer than 500 CG iterations were used. If we fix `maxiters` independent of the number of attributes, the CG Algorithm scales linearly with the dimensions of the data. This is a tremendous improvement over the Robust Algorithm's $O\left(M^2RF\right)$ complexity.

5 Data

In this paper we report results on two large and significant datasets being used in the life sciences. Within this paper these dataset are referred to as `ds1` and `ds2`. `ds1` has approximately 6,000 binary-valued attributes and 27,000 rows, while `ds2` has approximately 1,000,000 binary-valued attributes and 90,000 rows. Both datasets are sparse.

At the time of submitting we do not have permission to disclose further details about these datasets. Qualitatively, one may think of these datasets as containing information similar to a combination of the publicly available Open Compound Database and associated biological test data, as provided by the National Cancer Institute ([11]).

We also present results on three additional datasets derived from `ds1` and `ds2`. Two datasets, `ds1.100pca` and `ds1.10pca` are linear projections of `ds1` to 100 and 10 dimensions, using principle component analysis (PCA). Because of the expense of PCA, the projection of `ds2` to 100 dimensions uses an approximate linear projection algorithm known as *Anchors* (Moore 2000 [9]). The resulting dataset is called `ds2.100anchors`. Note that in both PCA and Anchors we are reducing the number of attributes while the number of rows remains the same. These datasets allow comparison of predictive performance on the original large sparse datasets to the same on smaller dense datasets. These datasets further allow comparison of LR to more computationally expensive algorithms such as k-nearest neighbor (K-NN) which cannot reasonably be run on `ds1` and `ds2`.

The dataset `ds1` has sparsity factor $F = 0.0220$, while `ds2` has sparsity factor $F = 0.0003$. The PCA- and Anchors-based datasets are dense and can be imagined to have a sparsity factor $F = 1.0$. We assume that the sparsity of the input data has no exploitable structure.

6 Analysis

We have run experiments with all four algorithms mentioned in section 4. The Simple Algorithm always had numerical problems, and never produced results. The Stepwise Algorithm was used in early experiments on `ds1`. Because the basis-choosing step in our implementation added only one attribute at a time to the basis, it was very slow to accumulate even 500 attributes. The Robust Cholesky algorithm, on the other hand, was able to run on all approximately 6,000 attributes of `ds1` faster than the Stepwise Algorithm could accumulate 500. For these reasons, this section will focus on the Robust Cholesky Algorithm, abbreviated Robust LR, and the CG Algorithm, abbreviated CG LR.

To evaluate and compare the predictive performance of our LR implementations and other traditional machine learning algorithms, we used 10-fold cross-validation. To visualize the predictive performance, we use Receiver Operating Characteristic (ROC) curves (Duda 1973 [2]). To construct these curves, we sort the dataset rows according to the probability a row is in the positive class under the logistic model. Then, starting at the graph origin, we step through the rows in order of decreasing probability, moving up one unit if the row is positive and right one unit otherwise. Every point (x,y) on an ROC curve represents the learner's 'favorite' $x+y$ rows from the dataset. Out of these favorite rows, x are actually positive, and y are negative.

Suppose a dataset had P positive rows and $R-P$ negative rows. A perfect learner on this dataset would have an ROC curve starting at the origin, moving straight up to $(0,P)$, and then straight right to end at $(R-P,P)$. Random guessing would produce, on average, an ROC curve which started at the origin and moved directly to the termination point $(R-P,P)$. Note that all ROC curves will start at the origin and end at $(R-P,P)$ because R steps up or right must be taken, one for each row. As a crude summary of an ROC curve, we measure the area under the curve relative to area under a perfect learner's curve. The result is denoted AUC. A perfect learner has an AUC of 1.0, while random guessing produces an AUC of 0.5. In order to compute confidence intervals, we compute one AUC score for each fold of our 10-fold cross-validation. We report the mean of these scores and a 95% confidence interval. When comparing two learning algorithms we use the same set of 10-fold partitions for each. This allows a pairwise test to determine whether one algorithm is significantly better or worse than the other according to the AUC measure.

Table 1: Computer Specifications

Name	CPU	RAM
`liver`	Alpha EV67 667 MHz 4MB cache	4GB
`tux`	Athlon XP 1900+	768MB

The results presented below include several machine learning algorithms. Not every one of these algorithms is present in every comparison, due to computational complexity. CG LR, Bayes Classifier (BC), and a decision tree learner (D-Tree) can reasonably be run on the largest of our datasets, `ds2`. D-Tree utilizes chi-squared pruning with a threshold of 10%. Because of implementation details, D-Tree was not used on projected datasets. Special effort was required to make K-Nearest Neighbor (K-NN) fast on `ds1.100pca` and `ds2.100anchors` with $K = 9$. Support vector machine results used either a linear (Linear SVM) or radial basis function (RBF SVM) kernel, as implemented in SVM^{light} (Joachims 1999 [5], [12]). The data input format for SVM^{light} has a sparse format, and we believe that sparse instance vectors are exploited.

Our results were obtained on two different machines. Experiments with `ds1`, `ds1.100pca` and `ds1.10pca` were performed on `tux`, a somewhat typical modern desktop. Experiments with `ds2` and `ds2.100anchors` used `liver`, an old Alpha Server. We believe that no experiment ever allocated over two gigabytes of memory. The specifications of these machines can be found in table 1.

Our analysis begins by comparing several learners on the `ds1` dataset. The ROC curves for BC, D-Tree, Robust LR, CG LR, Linear SVM and RBF SVM can be seen in the rather crowded figure 3. Summarizing this graph is table 2. We see that the LR methods' predictive performance is competitive with more recent learning algorithms. While RBF SVM performed on par with CG LR, CG LR was over fifteen times faster. It is curious that CG LR performed better than Robust LR. The next experiment hints how that could happen.

Using PCA to reduce the dimensionality of the `ds1` dataset to 100 attributes changes the situation somewhat, as shown in table 3. Robust LR is doing better despite the reduction in available information. We interpret this as a sign that Robust LR was overfitting in the experiment on `ds1`. Perhaps the approximate solutions to equation (3) used by CG

Table 2: Predictive Performance for Several Learners, 10-Fold, `ds1`, Machine=`tux`

Learner	Time (sec.)	AUC
BC	18	0.805±0.021
D-Tree	355	0.893±0.011
Robust LR	25290	0.869±0.019
CG LR	234	0.931±0.012
Linear SVM	303	0.918±0.010
RBF SVM	3689	0.927±0.013

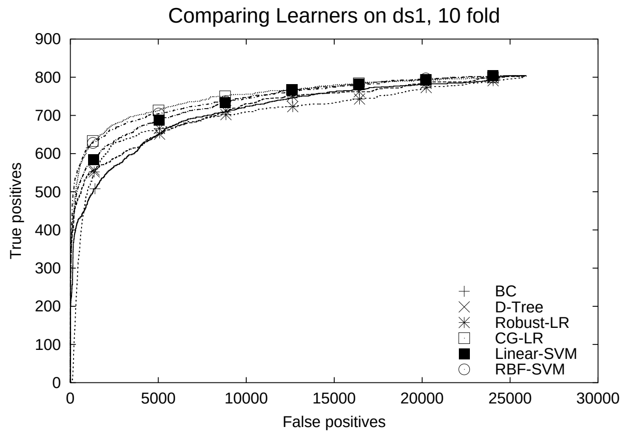

Figure 3: Predictive Performance for Several Learners, 10-Fold, `ds1`, Machine=`tux`

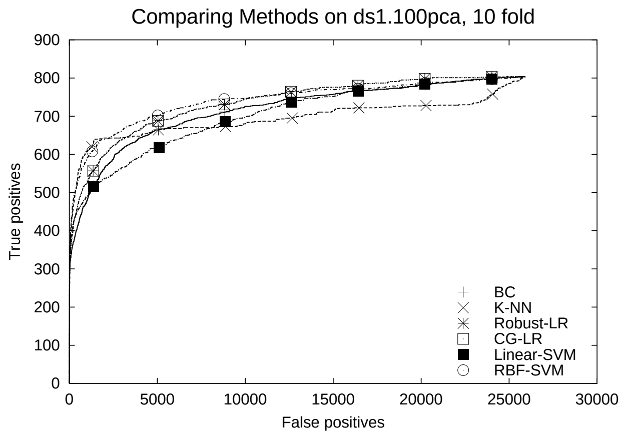

Figure 4: Predictive Performance for Several Learners, 10-Fold, `ds1.100pca`, Machine=`tux`

Table 4: Predictive Performance for Several Learners, 10-Fold, `ds1.10pca`, Machine=`tux`

Learner	Time (sec.)	AUC
BC	6	0.863±0.013
K-NN	155	0.817±0.028
Robust LR	13	0.795±0.015
CG LR	16	0.795±0.015
Linear SVM	90	0.569±0.076
RBF SVM	554	0.852±0.011

LR reduce overfitting. This explanation is similar in spirit to that used to support *early stopping* when training neural networks (e.g. Bishop 1995 [1]). Linear SVM suffer significantly with the projection, but BC, RBF SVM and the LR methods are less affected. From this we conclude that `ds1.100pca` contains useful information.

Though `ds1.100pca` is dense, it is now small enough for a fast K-NN algorithm to be run with $K = 9$. Note that CG LR required 255 seconds for this dataset. Given its complexity (c.f. section 4.4), we expect CG LR will scale to dense datasets with thousands of attributes and perhaps hundreds of thousands of rows.

Once we project the approximately 6000 attributes of `ds1` to the 10 attributes of `ds1.10pca`, the predictive performance of the algorithms spreads out as seen in table 4 and figure 5. The LR methods and RBF SVM fall significantly while Linear SVM plummets. BC and K-NN are still able to find useful information in the dataset. Almost certainly this is a special property of this dataset. Experiments on `ds2` and `ds2.100anchors` support this hypothesis. Although using PCA to project to the ten best dimensions is inexpensive and hence attractive, we must conclude that `ds1.10pca` contains too little information to be useful.

Table 3: Predictive Performance for Several Learners, 10-Fold, `ds1.100pca`, Machine=`tux`

Learner	Time (sec.)	AUC
BC	42	0.891±0.012
K-NN	386	0.862±0.017
Robust LR	707	0.914±0.010
CG LR	255	0.914±0.010
Linear SVM	288	0.874±0.010
RBF SVM	1528	0.920±0.014

Our final comparison uses the `ds2` and `ds2.100anchors` datasets. This data proved problematic for every algorithm we used. It is very sparse, has few positive rows, and of course is noisy. Because `ds2` has over one million attributes, only BC, D-Tree and CG LR were run on the full dataset. In addition to these algorithms, K-NN, RBF SVM and Linear SVM were run on the 100 dimensional Anchors projection `ds2.100anchors`. The results can be seen in table 5 and figure 6. While general performance is better than random guessing, only CG LR and D-Tree on the full dataset manage to step above the 60% AUC of the other algorithms, with CG LR doing significantly better than D-Tree. BC does not benefit from exposure to the full dataset, and K-NN did not distinguish itself with a steep ascent as it did in figures 4 and 5. This supports our hypothesis that

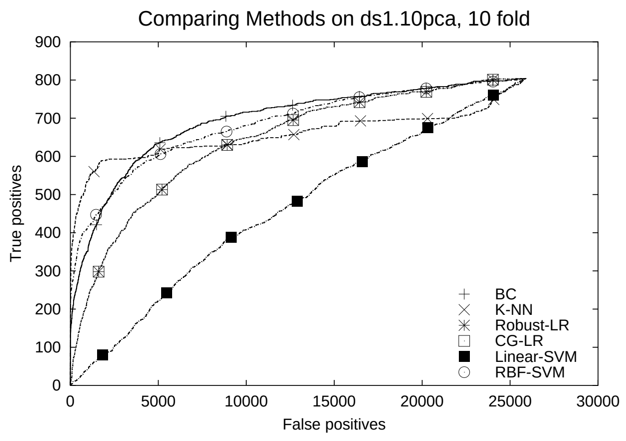

Figure 5: Predictive Performance for Several Learners, 10-Fold, `ds1.10pca`, Machine=`tux`

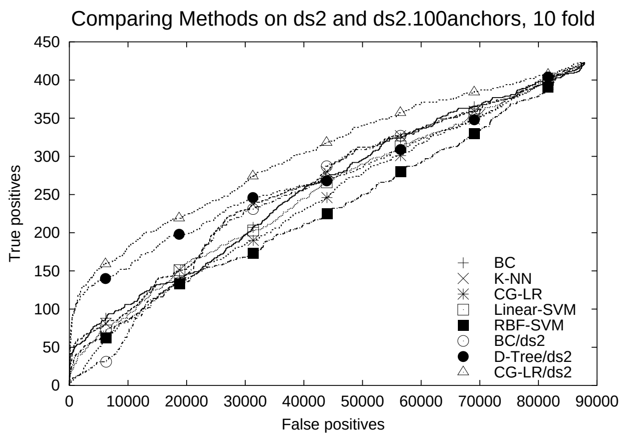

Figure 6: Predictive Performance for Several Learners, 10-Fold, `ds2` and `ds2.100anchors`, Machine=`liver`

Table 5: Predictive Performance for Several Learners, 10-Fold, `ds2.100anchors` except CG-ds2 and BC-ds2, which were run on the full `ds2` dataset; Machine=`liver`.

Learner	**Time (sec.)**	**AUC**
BC	151	0.614±0.030
K-NN	3905	0.620±0.028
CG LR	3795	0.572±0.021
Linear SVM	6364	0.583±0.029
RBF SVM	145781	0.561±0.030
BC-ds2	644	0.599±0.026
D-Tree-ds2	68224	0.643±0.032
CG-ds2	99738	0.712±0.028

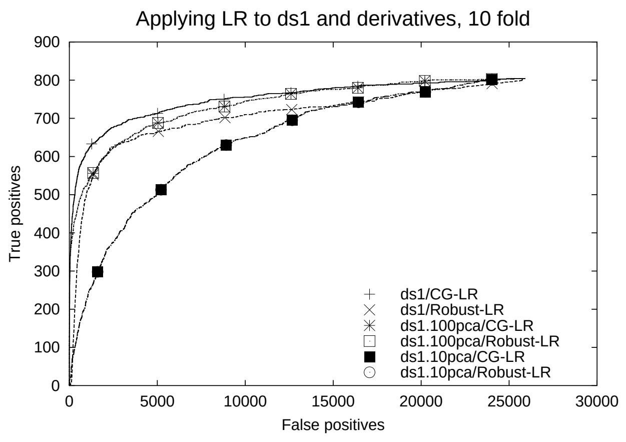

Figure 7: Comparing Robust LR and CG LR on `ds1`, `ds1.100pca`, and `ds1.10pca`.

BC and K-NN were exploiting peculiarities of `ds1.100pca` and `ds1.10pca` in a manner which may not generalize to other datasets.

Perhaps we should call attention to our concept of “scalable”. Although CG LR required approximately 27.7 hours of cpu time on an old 667MHz Alpha, or a guessed 12.25 hours on an Athlon XP 1900+, we believe the superior predictive performance more than justifies the expense.

We have established the potential of LR, and in particular CG LR, as a machine learning tool. Note that most of the ROC curves shown for CG LR are initially fairly steep, indicating that LR’s “favorite” rows are, by and large, members of the positive class. This property makes LR attractive in data mining scenarios where the goal is discovering “promising” rows in a dataset, with low false positive rate.

To summarize the performance of Robust and CG LR on `ds1`, `ds1.100pca`, and `ds1.10pca`, we provide figure 7. There is no new material here. It can be seen clearly here that CG LR is not a second-class LR implementation, especially given its superior performance over Robust LR. Although satisfactory and clearly competitive results can be obtained on the projected datasets, CG LR can be run on the original sparse datasets. Given CG LR’s speed and resistance to overfitting, and the expense of reducing dataset dimensionality, there is no reason *not* to run CG LR on the original sparse datasets.

Finally, it is interesting to explore the effects of the conjugate gradient epsilon parameter used to terminate conjugate gradient iterations. Earlier, we hypothesized that finding approximate solutions to equation (3), as done in CG LR, helped prevent overfitting. Figure 8 makes it clear that the best value for epsilon is not necessarily the smallest. While this is not enough to conclude that an epsilon of 0.01 causes less overfitting than an epsilon of 0.001, we must acknowledge that looser approximations when solving equation (3) can result in better LR predictions.

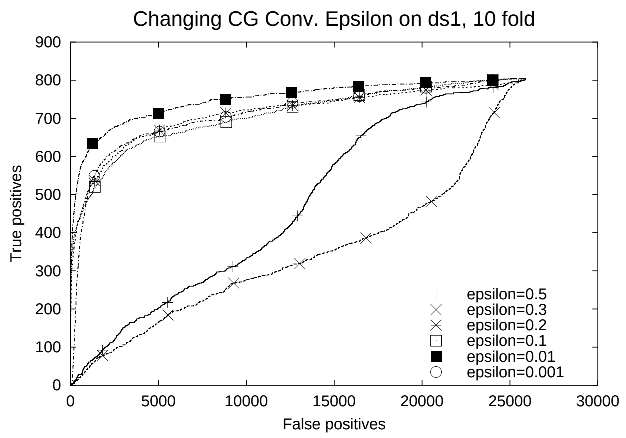

Figure 8: Varying CG Epsilon on `ds1`.

7 Conclusions

We have demonstrated that logistic regression, when accelerated by a conjugate gradient approximate linear solver, can surpass several well-known modern machine learning algorithms on two large sparse modern life sciences datasets. Our implementation provided superior predictive performance on a sparse dataset with over one million attributes and approximately 90,000 rows, and appears able to scale to millions of attributes with millions of rows. We have further demonstrated that conjugate gradient based logistic regression is suitable for use with dense datasets having thousands of attributes and tens of thousands of rows.

This is not the first paper to explore the combination of logistic regression and conjugate gradient algorithms. McIntosh (McIntosh 1982 [8]) clearly articulated and demonstrated the promise of this combination twenty years ago. We have independently rediscovered this combination, demonstrated its value, and are surprised that logistic regression does not have a more prominent place in the modern machine learning and data mining toolbox.

8 Future Work

Our chief aim is a larger empirical study of the techniques in this paper, including further exploration of CG and related methods. In particular, we will explore CG preconditioners and other performance enhancing techniques, such as those in McIntosh 1982 ([8]). We will also investigate semidefinite iterative methods, such as biconjugate gradient and others discussed in Greenbaum 1997 ([3]). We also intend to compare the performance of CG used in IRLS verus the more common technique of applying CG directly to the maximum likelihood equations (1) and (2).

Acknowledgements

The authors wish to thank Ting Liu of the School of Computer Science, Carnegie Mellon University, for her work implementing, debugging, and benchmarking the non-LR algorithms in this paper.

References

[1] C. M. Bishop. *Neural Netowrks for Pattern Recognition*. Oxford University Press, 1995.

[2] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973.

[3] A. Greenbaum. *Iterative Methods for Solving Linear Systems*, volume 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.

[4] D. W. Hosmer and S. Lemeshow. *Applied Logistic Regression*. Wiley-Interscience, 2 edition, 2000.

[5] T. Joachims. Making large-Scale SVM Learning Practical. In *Advances in Kernel Methods – Support Vector Learning*. MIT Press, 1999.

[6] A. H. Kramer and A. Sangiovanni-Vincentelli. Efficient Parallel Learning Algorithms for Neural Networks. In D. S. Touretzky, editor, *Neural information Processing Systems*, volume 1, pages 40–48. Morgan Kaufmann, San Mateo, 1989.

[7] P. McCullagh and J. A. Nelder. *Generalized Linear Models*, volume 37 of *Monographs on Statistics and Applied Probability*. Chapman & Hall, 2 edition, 1989.

[8] A. McIntosh. *Fitting Linear Models: An Application of Conjugate Gradient Algorithms*, volume 10 of *Lecture Notes in Statistics*. Springer-Verlag, New York, 1982.

[9] A. W. Moore. Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data. In *Proceedings of UAI-2000: The Sixteenth Conference on Uncertainty in Artificial Intelligence*, 2000.

[10] S. G. Nash and A. Sofer. *Linear and Nonlinear Programming*. McGraw-Hill, 1996.

[11] National Cancer Institute Open Compound Database, 2000. `http://cactus.nci.nih.gov/ncidb2`.

[12] SVM^{light}, 2002. `http://svmlight.joachims.org`.

[13] R. L. Watrous. Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization. Technical Report MS-CIS-87-51 LINC LAB 72, University of Pennsylvania, Philidelphia, June 1986.