Logistic Regression for Data Mining and High-Dimensional Classification

Paul Komarek
Dept. of Math Sciences
Carnegie Mellon University
komarek@cmu.edu

Advised by Andrew Moore
School of Computer Science
Carnegie Mellon University
awm@cs.cmu.edu

Committee

Alain Calvet
CADD
PGRD
Ann Arbor Laboratories

Alan Frieze, Chair
Dept. of Math Sciences
Carnegie Mellon University
af1p@andrew.cmu.edu

Bob Nichol
Dept. of Physics
Carnegie Mellon University
nichol@andrew.cmu.edu

To my father, for curiosity.
To my mother, for discipline.

Contents

I Introduction **6**

1 Introduction **7**
1.1 Science In This Century . . . 7
1.2 Focus Of Thesis . . . 7
1.3 Organization of Thesis . . . 8

II Background **9**

2 Conjugate Gradient **10**
2.1 Minimizing Quadratic Forms . . . 10
2.1.1 The Wrong Iterative Method: Steepest Descent . . . 11
2.1.2 The Right Iterative Method: Conjugate Gradient . . . 12
2.2 Minimizing Arbitrary Convex Functions With Conjugate Gradient . . . 15

3 Linear Regression **18**
3.1 Normal Model . . . 18
3.2 Ridge Regression . . . 20

4 Logistic Regression **22**
4.1 Logistic Model . . . 22
4.2 Maximum Likelihood Estimation . . . 23
4.3 Iteratively Re-weighted Least Squares . . . 24
4.4 Logistic Regression for Classification . . . 25

III Logistic Regression For Fast Classification **28**

5 Logistic Regression Computation **29**
5.1 Preliminaries . . . 29
5.1.1 Why LR? . . . 29
5.1.2 Our Approach . . . 29
5.1.3 Datasets . . . 32
5.1.4 Scoring . . . 33
5.1.5 Computing Platform . . . 35

5.1.6 Scope . . . 36
5.2 IRLS Parameter Evaluation and Elimination . . . 36
5.2.1 Indirect (IRLS) Stability . . . 36
5.2.2 Indirect (IRLS) Termination And Optimality . . . 53
5.2.3 Indirect (IRLS) Speed . . . 58
5.2.4 Indirect (IRLS) Summary . . . 60
5.3 CG-MLE Parameter Evaluation and Elimination . . . 62
5.3.1 Direct (CG-MLE) Stability . . . 62
5.3.2 Direct (CG-MLE) Termination And Optimality . . . 75
5.3.3 Direct (CG-MLE) Speed . . . 77
5.3.4 BFGS-MLE . . . 81
5.3.5 Direct (CG-MLE) Summary . . . 82

6 Characterizing Logistic Regression **85**
6.1 Classifiers . . . 85
6.1.1 SVM . . . 86
6.1.2 KNN . . . 87
6.1.3 BC . . . 88
6.2 Synthetic Datasets . . . 89
6.2.1 Number of Rows . . . 89
6.2.2 Number of Attributes . . . 90
6.2.3 Sparsity . . . 91
6.2.4 Attribute Coupling . . . 92
6.3 Real-world Datasets . . . 92
6.4 Conclusions . . . 95

IV Conclusion **103**

7 Related Work **104**

8 Conclusions **106**

9 Contributions **108**

10 Future Work **109**

A Acknowledgements **111**

B Software Documentation **112**
B.1 Introduction . . . 113
B.2 Actions . . . 113
B.2.1 Action Summary . . . 113
B.2.2 Action Details . . . 113
B.3 Data . . . 120
B.3.1 Dataset files . . . 120
B.3.2 Sparse Datasets . . . 120

B.3.3 Dataset naming . . . 121
B.4 Learners . . . 122
B.4.1 bc . . . 122
B.4.2 dtree . . . 123
B.4.3 lr . . . 123
B.4.4 lr_cgmle . . . 125
B.4.5 newknn . . . 127
B.4.6 oldknn . . . 128
B.4.7 super . . . 129
B.4.8 svm . . . 130
B.5 Examples . . . 132
B.5.1 Simple **roc** examples on several datasets . . . 132
B.5.2 Examples with various actions . . . 133
B.5.3 Examples with various learners . . . 134

C Miscellanea **135**

Part I

Introduction

Chapter 1

Introduction

1.1 Science In This Century

Modern researchers are facing an explosion in data. This data comes from immense investment in automated terrestrial and space-based telescopes, roboticized chemistry as used in the life sciences, network-based content delivery, and the computerization of business processes to name just a few sources. Several such datasets are described and used later in this thesis. Only through the creation of highly scalable analytic methods can we make the discoveries that justify this immense investment in data collection.

We believe these methods will rely fundamentally on algorithms and data structures for high-performance computational statistics. These modular building blocks would be used to form larger "discovery systems" functioning as industrious, intelligent assistants that autonomously identify and summarize interesting data for a researcher; adapt to the researcher's definition of interesting; and test the researcher's hypotheses about the data.

For a discovery system to be genuinely useful to the research community, we believe it should make analyses as quickly as a scientist can formulate queries and describe their hypotheses. This requires scalable data structures and algorithms capable of analyzing millions of data points with tens or tens-of-thousands of dimensions in seconds on modern computational hardware. The ultimate purpose of a discovery system is to allow researchers to concentrate on their research, rather than computer science.

1.2 Focus Of Thesis

The focus of this thesis is fast and robust adaptations of logistic regression (LR) for data mining and high-dimensional classification problems. LR is well-understood and widely used in the statistics, machine learning, and data analysis communities. Its benefits include a firm statistical foundation and a probabilistic model useful for "explaining" the data. There is a perception that LR is slow, unstable, and unsuitable for large learning or classification tasks. Through fast approximate numerical methods, regularization to avoid numerical instability, and an efficient implementation we will show that LR can outperform modern algorithms like Support Vector Machines (SVM) on a variety of learning tasks. Our novel implementation, which uses a modified iteratively re-weighted least squares estimation procedure, can compute model parameters for sparse binary datasets with hundreds of thousands of rows and attributes, and millions or tens of millions

of nonzero elements in just a few seconds. Our implementation also handles real-valued dense datasets of similar size. We believe LR is a suitable building block for the discovery systems as described above.

1.3 Organization of Thesis

This thesis is divided into four parts. Following this introductory part is Part II, which presents background information on the algorithms we use. Chapter 2 describes the linear and nonlinear Conjugate Gradient algorithms, Chapter 3 describes linear regression, and Chapter 4 describes logistic regression. After this, Part III discusses the use of LR for fast classification. This part is divided into Chapter 5, which discusses computational challenges, approaches, and conclusions as well as the datasets, scoring methods, and computing platform used for experiments; and Chapter 6 wherein we characterize our three LR variants, support vector machines, k-nearest-neighbor, and Bayes' classifier.

Part IV contains four short chapters. These begin with Chapter 7, which covers research related to this thesis. Chapter 8 is a summary of this thesis, including the conclusions we have drawn from our research. Chapter 9 describes our contributions, and Chapter 10 suggests several ideas for related future work. Following these concluding chapters are several appendices. We acknowledge several important people and organizations in Appendix A. Appendix B reproduces the documentation for the Auton Fast Classifier software [19] used in this thesis. Appendix C contains some light-hearted, miscellaneous information which didn't fit elsewhere. Concluding this thesis is the bibliography.

Part II

Background

Chapter 2

Conjugate Gradient

Conjugate gradient (CG) is an iterative minimization algorithm. We employ two versions of CG to accelerate our implementation of LR, and to overcome various numerical problems. These versions will be distinguished by the names *linear CG* and *nonlinear CG*. Before describing these methods and explaining the remarkable efficiency of linear CG, we motivate our discussion using *quadratic forms* and a simpler iterative minimizer known as the *method of Steepest Descent*.

2.1 Minimizing Quadratic Forms

A quadratic form is any function $f : \mathbb{R}^n \to \mathbb{R}$ of the form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}\mathbf{x} + c \tag{2.1}$$

where $\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{x}$ and $\mathbf{b}$ are vectors of length n, and c is a scalar. We will assume $\mathbf{A}$ is symmetric and positive semidefinite, that is

$$\mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0,\ \forall x \in \mathbb{R}^n \tag{2.2}$$

Any quadratic form with a symmetric positive semidefinite $\mathbf{A}$ will have a global minimum, where the minimum value is achieved on a point, line or plane. Furthermore there will be no local minima. The gradient and Hessian of the quadratic form f in Equation 2.1 are

$$f'(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b},\ f''(\mathbf{x}) = \mathbf{A} \tag{2.3}$$

Therefore algorithms which find the minimum of a quadratic form are also useful for solving linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$.

The minimum of a quadratic form f can be found by setting $f'(x) = 0$, which is equivalent to solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. Therefore, one may use Gaussian elimination or compute the inverse or left pseudo-inverse of $\mathbf{A}$ [23; 44]. The time complexity of these methods is $\mathrm{O}\left(n^3\right)$, which is infeasible for large values of n. Several possibilities exist for inverting a matrix asymptotically faster than $\mathrm{O}\left(n^3\right)$, with complexities as low as approximately $\mathrm{O}\left(n^{2.376}\right)$ [21; 37]. Due to constant factors not evident in the asymptotic complexity of these methods, n must be very large before they outperform standard techniques such as an LU decomposition or Cholesky factorization [44; 37]. However, for very large n even $\mathrm{O}\left(n^2\right)$ is infeasible, and hence these approaches are not practical. An alternative is to find an approximate solution $\mathbf{x}$ using an iterative method.

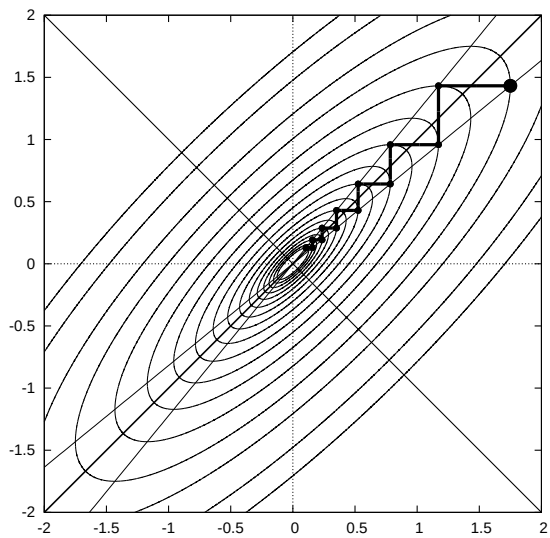


Figure 2.1: Steepest Descent worst-case path. The large dot is $\mathbf{x}_0$, and the smaller dots are the iterates $\mathbf{x}_i$. Steepest Descent follows $f'(\mathbf{x}_0)$ until it finds a perpendicular gradient. Then it repeats the process with the new gradient.

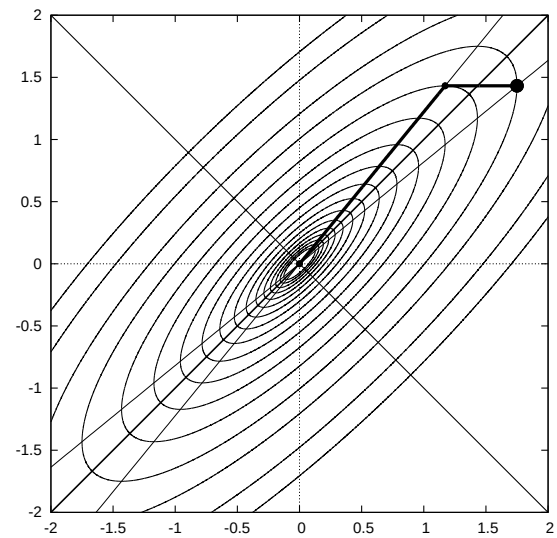


Figure 2.2: CG worst-case path. The large dot is $\mathbf{x}_0$, and the smaller dots are the iterates $\mathbf{x}_i$. Note that CG stops its first iteration such that $f'(\mathbf{x}_1) \perp f'(\mathbf{x}_0)$, but it does not follow the gradient for its second iteration.

2.1.1 The Wrong Iterative Method: Steepest Descent

Steepest Descent is a well-known iterative minimization method. It can be applied to any surface for which the gradient may be computed. The method of Steepest Descent computes the gradient at its current location, then travels in the opposite direction of the gradient until it reaches a minimum in this direction. Elementary calculus shows that at this minimum the new gradient is perpendicular to the previous gradient. When applied to a quadratic form f with initial guess $\mathbf{x}_0$, Steepest Descent may be summarized as

$\mathbf{d}_i = -f'(\mathbf{x}_i) = \mathbf{b} - \mathbf{A}\mathbf{x}_i$	Choose search direction.
$\alpha = \frac{\mathbf{d}_i^T \mathbf{d}_i}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i}$	Compute distance to minimum for line search.
$\mathbf{x}_{i+1} = \mathbf{x} + \alpha \mathbf{d}_i$	Compute next location.

Because we are only concerned with quadratic forms having a global minimum and no local minima, Steepest Descent is guaranteed to converge. However there is no guarantee that Steepest Descent will converge quickly. For example, Figure 2.1 shows Steepest Descent's path when started from a worst-case starting point $\mathbf{x}_0$ [41].

The minimum point in Figure 2.1 is $\mathbf{x}^* = (0,0)^T$, which by definition satisfies $\mathbf{A}\mathbf{x}^* = \mathbf{b}$. Let $\mathbf{e}_i = \mathbf{x}^* - \mathbf{x}_i$ represent the error vector at iteration i. We cannot compute $\mathbf{e}_i$ since we do not know $\mathbf{x}^*$; this requires the

inverse or pseudo-inverse of $\mathbf{A}$. However we can easily compute the image of $\mathbf{e}_i$ under $\mathbf{A}$, since

$$\begin{aligned} \mathbf{A}\mathbf{e}_i &= \mathbf{A}(\mathbf{x}^* - \mathbf{x}_i) && (2.4) \\ &= \mathbf{A}\mathbf{x}^* - \mathbf{A}\mathbf{x}_i && (2.5) \\ &= \mathbf{b} - \mathbf{A}\mathbf{x}_i && (2.6) \end{aligned}$$

Define the residual vector $\mathbf{r}_i = \mathbf{b} - \mathbf{A}\mathbf{x}_i = \mathbf{A}\mathbf{e}_i$, and note that Equation 2.3 implies $\mathbf{r}_i = -f'(\mathbf{x}_i)$. Since this negative gradient is the direction Steepest Descent will travel next, we may characterize Steepest Descent as greedily reducing the size of the residual.

Because Steepest Descent follows the gradient $-f'(\mathbf{x}_i) = \mathbf{r}_i$ until it reaches a point $\mathbf{x}_{i+1}$ for which $\mathbf{r}_{i+1} \perp \mathbf{r}_i$, the algorithm can only make square turns. When applied to a two dimensional quadratic form with symmetric positive semidefinite Hessian $\mathbf{A}$, this property implies that search directions must be repeated if more than two iterations are necessary. The only way to guarantee two or fewer iterations is to choose a starting point $\mathbf{x}_0$ for which $f'(x_0) \parallel \mathbf{e}_0$ or $f'(\mathbf{x}_0) \perp \mathbf{e}_0$. This cannot be done without computing $\mathbf{e}_0$, which requires knowing the answer $\mathbf{x}^*$.

2.1.2 The Right Iterative Method: Conjugate Gradient

The most important difference between CG and Steepest Descent is that CG will find the minimum of an n-dimensional quadratic form in n or fewer steps. Because a quadratic form's Hessian is constant it is possible to plan a set of search directions which avoid redundant calculations. CG takes advantage of this property, making it more suitable than Steepest Descent for minimizing quadratic forms. We explain below how the CG search directions and step lengths are chosen.

We would like to have a set of n linearly independent direction vectors $\mathbf{d_0}, \ldots, \mathbf{d_{n-1}}$ and scalars $\alpha_0, \ldots, \alpha_{n-1}$ for which

$$\mathbf{e}_0 = \sum_{j=0}^{n-1} \alpha_j \mathbf{d}_j \tag{2.7}$$

This would allow us to travel exactly the right distance in each direction, reducing the error to zero and arriving at a minimizer of the quadratic form f in n or fewer steps. However this is not enough. Suppose we knew $\mathbf{e_0}$ and could solve Equation 2.7. The solution would require $\mathrm{O}\left(n^3\right)$ operations to compute, a time complexity we dismissed as infeasible in Section 2.1. However, if the vectors $\mathbf{d_0}, \ldots, \mathbf{d_{n-1}}$ were mutually orthogonal, the solution to this equation could be found in $\mathrm{O}(n)$ time per α_j.

Because we cannot know $\mathbf{e_0}$ without knowing the solution, we cannot solve Equation 2.7 in any amount of time. However, we can compute $\mathbf{A}\mathbf{e_0}$, as shown in Equation 2.6. Left multiplying Equation 2.7 by $\mathbf{A}$ produces

$$\mathbf{A}\mathbf{e_0} = \sum_{j=0}^{n-1} \alpha_j \mathbf{A}\mathbf{d_j} \tag{2.8}$$

We have returned to our $\mathrm{O}\left(n^3\right)$ problem, and this time orthogonality of $\mathbf{d_0}, \ldots, \mathbf{d_{n-1}}$ will not help because we have left-multiplied $\mathbf{d_j}$ by $\mathbf{A}$. Instead we want mutual $\mathbf{A}$-*orthogonality*, wherein we require that $\mathbf{d_i}^T \mathbf{A}\mathbf{d_j} = 0,\ i \neq j$.

If we assume we can find a set of linearly independent directions $\mathbf{d_j}$ which are mutually $\mathbf{A}$-orthogonal, we

may solve for α_k by left-multiplying Equation 2.8 by $\mathbf{d_k}^T$, since

$$\mathbf{d_k}^T\mathbf{A}\mathbf{e_0} = \sum_{j=0}^{n-1}\alpha_j\mathbf{d_k}^T\mathbf{A}\mathbf{d_j} \tag{2.9}$$

$$= \alpha_k\mathbf{d_k}^T\mathbf{A}\mathbf{d_k} \tag{2.10}$$

by **A**-orthogonality and hence

$$\alpha_k = \frac{\mathbf{d_k}^T\mathbf{A}\mathbf{e_0}}{\mathbf{d_k}^T\mathbf{A}\mathbf{d_k}} = -\frac{\mathbf{d_k}^T\mathbf{r_0}}{\mathbf{d_k}^T\mathbf{A}\mathbf{d_k}} \tag{2.11}$$

Note that we further require $\mathbf{A}\mathbf{d_k} \neq 0$ for α_k to be well-defined. Our strategy, if a set of linearly independent **A**-orthogonal vectors $\mathbf{d_j}$ not in the null space of **A** exist, is to

1. Choose $\mathbf{x_0}$
2. Compute $\mathbf{r_i} = \mathbf{b} - \mathbf{A}\mathbf{x_i}$
3. Compute $\alpha_i = -\mathbf{d_i}^T\mathbf{r_0}/\mathbf{d_i}^T\mathbf{A}\mathbf{d_i}$
4. Compute $\mathbf{x_{i+1}} = \mathbf{x_i} - \alpha_i\mathbf{d_i}$
5. Increment i and repeat from step 2 until $||\mathbf{r_i}|| = 0$

This ensures we travel $-\sum_{j=0}^{n-1}\alpha_j\mathbf{d_j}$, thus eliminating all of $\mathbf{e_0}$ to arrive at the solution $\mathbf{x}^*$ in no more than n iterations.

Not only does there exist a set of linearly independent mutually **A**-orthogonal direction vectors $\mathbf{d_j}$ not in the null space of **A**, but each vector can be computed incrementally. Let $\mathbf{d_0} = \mathbf{r_0}$. This allows computation of $\mathbf{x_1}$ and $\mathbf{r_1}$, and all remaining direction vectors can be computed as needed via

$$\mathbf{d_i} = \mathbf{r_i} + \beta_i\mathbf{d_{i-1}} \tag{2.12}$$

where

$$\beta_i = \frac{\mathbf{r_i}^T\mathbf{r_i}}{\mathbf{r_{i-1}}^T\mathbf{r_{i-1}}} \tag{2.13}$$

[41; 31]. Another name for **A**-orthogonality is conjugacy, and it is from the conjugacy of its search directions that CG gets its name. Our final linear CG algorithm is shown in Algorithm 1. Note that line 1.1 in this algorithm can be equivalently written as $\alpha_i := -\mathbf{r_i}^T\mathbf{r_i}/(\mathbf{r_i}^T\mathbf{A}\mathbf{r_i})$ due to properties described below. This formulation eliminates the need to store $\mathbf{r_0}$. See Shewchuk [41] for details.

It is worth emphasizing that each direction vector is a linear combination of the current residual and the previous direction vector. Because the direction vectors are linearly independent, a simple induction implies that $\mathbf{d_i}$ is a linear combination of $\mathbf{r_0}, \ldots, \mathbf{r_i}$ and

$$\mathcal{S}_i = \text{span}\,(\mathbf{d_0}, \ldots, \mathbf{d_i}) = \text{span}\,(\mathbf{r_0}, \ldots, \mathbf{r_i}) \tag{2.14}$$

The space searched so far, $\mathcal{D}_i$, may also be written as the Krylov subspaces:

$$\mathcal{S}_i = \mathbf{d_0}, \mathbf{A}\mathbf{d_0}, \ldots, \mathbf{A}^{\mathbf{d_i}} = \mathbf{r_0}, \mathbf{A}\mathbf{r_0}, \ldots, \mathbf{A}^{\mathbf{r_i}} \tag{2.15}$$

See Shewchuk [41] and Nash and Sofer [31] for proofs of these properties.

input : $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x_0}$
output : $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$

$\mathbf{r_0} := \mathbf{b} - \mathbf{Ax_0}$
i := 0
while $||\mathbf{r_i}|| > 0$ **do**
  if i=0 **then** $\beta_i := 0$
  else $\beta_i := \mathbf{r_i}^T\mathbf{r_i}/(\mathbf{r_{i-1}}^T\mathbf{r_{i-1}})$
  $\mathbf{d_i} := \mathbf{r_i} + \beta_i\mathbf{d_{i-1}}$
1.1 $\alpha_i := -\mathbf{d_i}^T\mathbf{r_0}/(\mathbf{d_i}^T\mathbf{Ad_i})$
  $\mathbf{x_{i+1}} := \mathbf{x_i} - \alpha_i\mathbf{d_i}$
  $\mathbf{r_{i+1}} := \mathbf{b} - \mathbf{Ax_i}$
  $i := i + 1$

Algorithm 1: Linear CG.

Figure 2.2 shows the conjugate directions used by CG when started from the same place as Steepest Descent was in Figure 2.1. Because CG will converge within n iterations, this is a worst-case starting point for CG requiring $n = 2$ steps. CG does so much better than Steepest Descent in our example because it assumes, correctly, that f is a quadratic form. If f is not a quadratic form then adjustments must be made. The resulting algorithm is often referred to as *nonlinear CG*, and is discussed briefly in Section 2.2. Predictably, the CG algorithm discussed so far is called *linear CG*.

We have omitted demonstration of the existence of a linearly independent set of conjugate directions which are not in the null space of $\mathbf{A}$, and also omitted many interesting properties of CG. The value of α in Equation 2.11 would be the same had we defined it as the step length for minimizing line-search in direction $\mathbf{d_j}$ starting at $\mathbf{x_j}$. The minimizer is $\mathbf{x_{j+1}}$. This is where the gradient, and hence the residual, is perpendicular to $\mathbf{d_j}$. Thus r_{j+1} is perpendicular to $\mathbf{d_j}$. Equation 2.14 then implies that each new residual is perpendicular to the space already searched. It also follows that the residuals are mutually orthogonal and that $||\mathbf{e_i}||_{\mathbf{A}}^2 = \mathbf{e_i}^T\mathbf{Ae_i}$ is minimized over $\mathcal{S}_i$. These properties help explain the efficiency per iteration of CG.

One may also observe that the computations needed in each CG iteration depend only on $\mathbf{A}$ and the previous iteration, with the exception of α_k as shown in Equation 2.11. In fact α_k can also be written in terms of the previous iteration [41; 31]. Therefore we only need to store a few vectors and $\mathbf{A}$. Because only matrix-vector computations are needed, storage and computation may exploit the structure or sparsity of $\mathbf{A}$.

It was demonstrated above that CG should arrive at the exact solution $\mathbf{x}^*$ in no more than n iterations when $\mathbf{A}$ is $n \times n$. This property does not hold when rounded arithmetic is used, for instance when CG is performed on a computer. Errors during computations can reduce the conjugacy of the direction vectors and decrease efficiency so that many more than n iterations are required. [31]

The convergence rate for CG is not entirely straightforward. In Shewchuk [41] this rate is described as

$$\frac{||\mathbf{e_i}||_{\mathbf{A}}}{||\mathbf{e_0}||_{\mathbf{A}}} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i \tag{2.16}$$

where κ is *condition number* $\lambda_{\max}/\lambda_{\min}$, and $\lambda_{\max}$ and $\lambda_{\min}$ are the largest and smallest eigenvalues of $\mathbf{A}$ in absolute value. If CG is terminated when the error under $\mathbf{A}$ is sufficiently small relative to the initial error, that is $||\mathbf{e_i}||_{\mathbf{A}}/||\mathbf{e_0}||_{\mathbf{A}} < \varepsilon$, then CG has a worst-case complexity of $O(m\kappa)$ where m is the number of nonzero

entries in $\mathbf{A}$. The convergence rate and worse-case complexity occur when the eigenvalues of $\mathbf{A}$ are evenly distributed between λ_{min} and λ_{max} and are distinct. Significantly better convergence is possible when the eigenvalues are clustered or duplicated. [41]

There are many possibilities for deciding when to terminate CG besides the relative size of the error. Furthermore the relation between the true error and the computed error is complex, and the subject of ongoing numerical analysis research. Termination alternatives include minimizing the absolute size of the error, or relative or absolute size of the residual, or context-dependent quantities such as the relative change of the likelihood when computing a maximum likelihood estimate.

Finding a good termination criteria is very important to minimize wasted time while ensuring sufficient optimality. When we discuss our use of CG later in this thesis, the most important issue will be proper termination. Because the worst-case time complexity of CG depends on the spectrum of $\mathbf{A}$, our CG termination methods should to adapt to each dataset. We will also examine how the starting point $\mathbf{x_0}$ affects our methods in the one case where an informed choice is possible. Overall, we aim hide all of these issues from the users of our algorithms.

2.2 Minimizing Arbitrary Convex Functions With Conjugate Gradient

Setting the first derivative of a quadratic form equal to zero creates a system of linear equations. For this reason we refer to CG as *linear CG* when it is applied to a quadratic form. For differentiable convex functions in general the first derivative is not a linear system, and in this context we use *nonlinear CG*.

Let f be a quadratic form and g be an arbitrary differentiable convex function. Both have a symmetric positive semidefinite Hessian, which for f is constant and for g is not constant. Just like linear CG, nonlinear CG attempts to find the minimum of g by evaluating the gradient, computing a conjugate direction and finding the minimum in that direction. However there are many differences between the two methods. Everywhere we used the matrix $\mathbf{A}$ for linear CG, we must now use $g''(\mathbf{x})$ or make a different computation. Where we were able to compute the minimum point on the line search in direction $\mathbf{d_i}$ in Equation 2.11, we must now make a true search for the minimum. All methods we might have tried to compute the linear combination of Equation 2.12 would have resulted in the same coefficient β_i seen in Equation 2.13, but for nonlinear CG there are several reasonable formulas for β_i.

These changes make nonlinear CG more expensive than linear CG. We can no longer define the residual $\mathbf{r}$ as $\mathbf{b} - \mathbf{A}\mathbf{x}$, and must rely on the more general definition $\mathbf{r} = -g'(x)$. This definition requires gradient evaluations, but avoids computation of the Hessian. There are many ways to search for the minimum along a direction $\mathbf{d_i}$, but almost every method requires evaluation or approximation of the gradient at several points along the line $\mathbf{x_i} - \hat{\alpha}\mathbf{d_i}$ to find the optimal value for α. Some line searches require computation or estimation of g'', which may be expensive.

For this thesis we use the Secant method for line searches in nonlinear CG. This method uses directional derivatives at two endpoints along the line to fit a quadratic and moves the next line search iteration to the minimum of that quadratic. The actual slope is computed at this point and is used to determine which endpoint should be eliminated. Since g is quadratic the true minimum will always be found between the endpoints, so long as the endpoints are initialized with the minimizer between them. This condition itself requires a search along the negative gradient, looking for a point with positive directional derivative.

Terminating the line search can be done in many ways. Many methods have been suggested such as checking the relative change of g at the line search iterates, using the relative difference of the slopes, ob-

serving the slopes themselves, or even stopping after a fixed number of iterations [41; 31]. We have tried all of these approaches except the last. In our application of nonlinear CG to the LR maximum likelihood equations, we have selected a combination of relative and absolute slope checking.

There are several ways to compute conjugate search directions. In all cases a linear combination between the gradient and the previous search direction is used, but the value of the coefficient β in Equation 2.12 changes. Three methods of computing β are

$$\beta_i^{(\mathrm{FR})} = \mathbf{r_i}^T \mathbf{r_i} / \mathbf{r_{i-1}}^T \mathbf{r_{i-1}} \tag{2.17}$$

$$\beta_i^{(\mathrm{PR})} = \mathbf{r_i}^T (\mathbf{r_i} - \mathbf{r_{i-1}}) / \mathbf{r_{i-1}}^T \mathbf{r_{i-1}} \tag{2.18}$$

$$\beta_i^{(\mathrm{HS})} = \mathbf{r_i}^T (\mathbf{r_i} - \mathbf{r_{i-1}}) / d_{i-1}^T (\mathbf{r_i} - \mathbf{r_{i-1}}) \tag{2.19}$$

The first of these, $\beta_i^{(\mathrm{FR})}$, is known as Fletcher-Reeves and is the same formula used in linear CG. The second, $\beta_i^{(\mathrm{PR})}$, is called Polak-Ribiére and the remaining formula is Hestenes-Stiefel. Use of Fletcher-Reeves guarantees convergence so long $\mathbf{x_0}$ is sufficiently close to $\mathbf{x}^*$. Though functions have been devised for which Polak-Ribiére and Hestenes-Stiefel perform very poorly, use of these formulas often improves performance [41; 31].

With nonlinear CG the Hessian changes with every iteration. While we can ensure $\mathbf{d_i}$ and $\mathbf{d_{i+1}}$ are conjugate using the Hessian at $\mathbf{x_{i+1}}$, we cannot in general ensure that $\mathbf{d_i}$ is conjugate to $\mathbf{d_{i+2}}$. During initial iterations of nonlinear CG we can assume that we are not near $\mathbf{x}^*$, and hence large steps are taken which create large differences in the Hessian. Once we are near $\mathbf{x}^*$ we might assume that the Hessian will not change much between iterations, and conjugacy will again become meaningful. For this reason *restarts* are periodically employed.

A nonlinear CG restart at iteration i simply means setting $\mathbf{d_i} = -g'(\mathbf{x_i})$ as if we were starting from scratch with initial guess $\mathbf{x_i}$. There are several popular methods for deciding when to restart. The simplest criterion derives from the observation that there cannot be more than n conjugate directions in $\mathbb{R}^n$, and hence a restart should occur after every n iterations. Another popular restart criterion is *Powell restarts*, which checks the orthogonality of successive residuals. A restart is triggered when

$$\frac{\mathbf{r_i}^T \mathbf{r_{i-1}}}{\mathbf{r_{i-1}}^T \mathbf{r_{i-1}}} > \nu \tag{2.20}$$

for some small value of ν. One choice of ν commonly used is $||\mathbf{r_i}||^2 / ||\mathbf{r_{i-1}}^2||$, which corresponds to restarting when $\beta_i^{(\mathrm{PR})} < 0$. Note that Powell restarts can be easily incorporated into Polak-Ribiére direction updates by setting $\beta_i = \max(\beta_i^{(\mathrm{PR})}, 0)$. We will refer to this combination of direction update and restart criterion as "modified Polak-Ribiére". [41; 31]

Algorithm 2 summarizes nonlinear CG. A line search method must be chosen for 2.1 of this algorithm, to replace the step length computation 1.1 of Algorithm 1. The direction update function must be selected from several possibilities, and a criteria must be chosen for when to perform a CG restart. Finally, note that the residual requires a generic gradient computation.

All of the issues discussed here will be raised again in Section 5.3, where we apply nonlinear CG to the convex but non-quadratic LR likelihood function described in Section 4.2. We will continue to use the line search and restart criteria described here, and will compare performance using all four direction update formulas.

input : g($\mathbf{x}$), $\mathbf{x_0}$, direction update function $du(\mathbf{r_i}, \mathbf{r_{i-1}}, \mathbf{d_{i-1}})$, restart indicator $ri(\mathbf{r_i}, \mathbf{r_{i-1}}, \mathbf{d_{i-1}}, i)$, line search $ls(\mathbf{x_i}, \mathbf{d_i})$

output : minimizer $\mathbf{x}$ of g

$\mathbf{r_0} := -g'(\mathbf{x_0})$
i := 0
while $||\mathbf{r_i}|| > 0$ **do**
  if i=0 **then** $\beta_i := 0$
  else $\beta_i :=$ du$(\mathbf{r_i}, \mathbf{r_{i-1}}, \mathbf{d_{i-1}})$
  if ri$(\mathbf{r_i}, \mathbf{r_{i-1}}, \mathbf{d_{i-1}}, i)$ **then** $\beta_i := 0$
  $\mathbf{d_i} := \mathbf{r_i} + \beta_i \mathbf{d_{i-1}}$
2.1 $\mathbf{x_{i+1}} :=$ ls$(\mathbf{x_i}, \mathbf{d_i})$
  $\mathbf{r_{i+1}} := -g'(\mathbf{x_{i+1}})$
  $i := i + 1$

Algorithm 2: Nonlinear CG.

Chapter 3

Linear Regression

3.1 Normal Model

Regression is a collection of statistical function-fitting techniques. These techniques are classified according to the form of the function being fit to the data. In linear regression a linear function is used to describe the relation between the independent variable or vector $\mathbf{x}$ and a dependent variable y. This function has the form

$$f(\mathbf{x},\beta) = \beta_0 + \beta_1 x_1 + \ldots + \beta_M, x_M \tag{3.1}$$

where β is the vector of unknown parameters to be estimated from the data. If we assume that $\mathbf{x}$ has a zeroth element $x_0 = 1$, we include the constant term β_0 in the parameter vector **beta** and write this function more conveniently as

$$f(\mathbf{x},\beta) = \beta^T \mathbf{x} \tag{3.2}$$

Because f is linear, the parameters $\beta_1, \ldots, \beta_M$ are sometimes called the slope parameters while β_0 is the offset at the origin. Due to random processes such as measurement errors, we assume that y does not correspond perfectly to $f(\mathbf{x},\beta)$. For this reason a statistical model is created which accounts for randomness. For linear regression we will use the linear model

$$y = f(\mathbf{x},\beta) + \varepsilon \tag{3.3}$$

where the error-term ε is a Normally-distributed random variable with zero mean and unknown variance σ^2. Therefore the expected value of this linear model is

$$E(y_i) = f(\mathbf{x},\beta) + E(\varepsilon) \tag{3.4}$$

$$= f(\mathbf{x},\beta) \tag{3.5}$$

and for this reason f is called the *expectation function*. We assume that σ^2 is constant and hence does not depend on $\mathbf{x}$. We also assume that if multiple experiments $\mathbf{x_i}$ are conducted, then the errors ε_i are independent.

Suppose $\mathbf{X}$ is an $R \times M$ matrix whose rows $\mathbf{x_i}$ represent R experiments, each described by M variables. Let $\mathbf{y}$ be an $R \times 1$ vector representing the outcome of each experiment in $\mathbf{X}$. We wish to estimate values for the parameters β such that the linear model of Equation 3.3 is, hopefully, a useful summarization of the data $\mathbf{X}, \mathbf{y}$. One common method of parameter estimation for linear regression is *least squares*. This method finds

a vector $\hat{\beta}$ which minimizes the *residual sum of squares* (RSS), defined as

$$\begin{aligned} \mathrm{RSS}(\hat{\beta}) &= \sum_{i=1}^{R} \left(y_i - \hat{\beta}^T \mathbf{x_i}\right)^2 && (3.6) \\ &= (\mathbf{y} - \hat{\beta}^T \mathbf{X})^T (\mathbf{y} - \hat{\beta}^T \mathbf{X}) && (3.7) \end{aligned}$$

where $\mathbf{x_i}$ is the ith row of $\mathbf{X}$. Note that β is the "true" parameter vector, while $\hat{\beta}$ is an informed guess for β. Throughout this thesis a variable with a circumflex, such as $\hat{\beta}$, is an estimate of some quantity we cannot know like β. The parameter vector is sometimes called a *weight vector* since the elements of β are multiplicative weights for the columns of $\mathbf{X}$. For linear regression the *loss function* is the RSS. Regression techniques with different error structure may have other loss functions.

To minimize the RSS we compute the partial derivative at $\hat{\beta}_i$ with respect to β_i, for $i = 1, \ldots, M$, and set the partials equal to zero. The result is the *score equations* $\mathbf{X^T}(\mathbf{X}\hat{\beta} - \mathbf{y}) = 0$ for linear regression, from which we compute the *least squares estimator*

$$\hat{\beta} = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{y} \tag{3.8}$$

The expected value of $\hat{\beta}$ is β, and hence $\hat{\beta}$ is unbiased. The covariance matrix of $\hat{\beta}$, $\mathrm{cov}(\hat{\beta})$, is $\sigma^{-2}(\mathbf{X}^T\mathbf{X})^{-1}$. The variances along the diagonal are the smallest possible variances for any unbiased estimate of β. These properties follow from our assumption that the errors ε_i for predictions were independent and Normally distributed with zero mean and constant variance σ^2 [30].

Another method of estimating β is *maximum likelihood estimation*. In this method we evaluate the probability of encountering the outcomes $\mathbf{y}$ for our data $\mathbf{X}$ under the linear model of Equation 3.3 when $\beta = \hat{\beta}$. We will choose as our estimate of β the value $\hat{\beta}$ which maximizes the *likelihood function*

$$\begin{aligned} \mathbb{L}(\mathbf{y}, \mathbf{X}, \hat{\beta}, \sigma^2) &= P(y_1 | \mathbf{x_1}, \hat{\beta}, \sigma^2) \cdot \ldots \cdot P(y_R | \mathbf{x_R}, \hat{\beta}, \sigma^2) && (3.9) \\ &= (2\pi\sigma^2)^{-n/2} \prod_{i=1}^{R} \exp\left(-\frac{1}{2\sigma^2}(y_i - \hat{\beta}^T \mathbf{x_i})^T (y_i - \hat{\beta}^T \mathbf{x_i})\right) && (3.10) \end{aligned}$$

over $\hat{\beta}$. We are interested in maximization of $\mathbb{L}$ and not its actual value, which allows us to work with the more convenient log-transformation of the likelihood. Since we are maximizing over $\hat{\beta}$ we can drop factors and terms which are constant with respect to $\hat{\beta}$. Discarding constant factors and terms that will not affect maximization, the *log-likelihood function* is

$$\begin{aligned} \ln \mathbb{L}(\mathbf{y}, \mathbf{X}, \hat{\beta}, \sigma^2) &= \sum_{i=1}^{R} -\frac{1}{2\sigma^2}(y_i - \hat{\beta}^T \mathbf{x_i})^T (y_i - \hat{\beta}^T \mathbf{x_i}) && (3.11) \\ &= -(\mathbf{y} - \hat{\beta}^T \mathbf{X})^T (\mathbf{y} - \hat{\beta}^T \mathbf{X}) && (3.12) \end{aligned}$$

To maximize the log-likelihood function we need to minimize $(y_i - \hat{\beta}^T \mathbf{x_i})^T (y_i - \hat{\beta}^T \mathbf{x_i})$. This is the same quantity minimized in Equation 3.7, and hence it has the same solution. In general one would differentiate the log-likelihood and set the result equal to zero, and the result is again the linear regression score equations. In fact these equations are typically defined as the derivative of the log-likelihood function. We have shown that the maximum likelihood estimate (MLE) for $\hat{\beta}$ is identical to the least squares estimate under our assumptions that the errors ε_i are independent and Normally distributed with zero mean and constant variance σ^2.

If the variance σ_i^2 for each outcome y_i is different, but known and independent of the other experiments, a simple variation known as *weighted least squares* can be used. In this procedure a *weight matrix* $\mathbf{W} =$

$\text{diag}((\sigma_1^2, \ldots, \sigma_R^2)^T)$ is used to standardize the unequal variances. The score equations become $\mathbf{X^T W}(\mathbf{X}\hat{\beta} - \mathbf{y}) = 0$, and the *weighted least squares estimator* is

$$\hat{\beta} = \left(\mathbf{X}^T \mathbf{W} \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{W} \mathbf{y} \tag{3.13}$$

It is also possible to accommodate correlated errors when the covariance matrix is known. [30]

3.2 Ridge Regression

Suppose a linear regression model for average daily humidity contains attributes for the day-of-month and the temperature. Suppose further that in the data $\mathbf{X}$ these attributes are correlated, perhaps because the temperature rose one degree each day data was collected. When computing the expectation function $f(\mathbf{x_i}, \hat{\beta}) = \hat{\beta}^T \mathbf{x_i}$, overly large positive estimates of $\hat{\beta}_{\text{temperature}}$ could be transparently offset by large negative values for $\hat{\beta}_{\text{day-of-month}}$, since these attributes change together. More complicated examples of this phenomenon could include any number of correlated variables and more realistic data scenarios.

Though the expectation function may hide the effects of correlated data on the parameter estimate, computing the expectation of new data may expose ridiculous parameter estimates. Furthermore, ridiculous parameter estimates may interfere with proper interpretation of the regression results. Nonlinear regressions often use iterative methods to estimate $\hat{\beta}$. Correlations may allow unbound growth of parameters, and consequently these methods may fail to converge.

The undesirable symptoms of correlated attributes can be reduced by restraining the size of the parameter estimates, or preferring small parameter estimates. Ridge regression is a linear regression technique which modifies the RSS computation of Equation 3.7 to include a penalty for large parameter estimates for reasons explained below. This penalty is usually written as $\lambda \tilde{\beta}^T \tilde{\beta}$ where $\tilde{\beta}$ is the vector of slope parameters $\beta_1, \ldots, \beta_M$. This penalty is added to the RSS, and we now wish to minimize

$$\text{RSS}_{\text{ridge}} = \text{RSS} + \lambda \tilde{\beta}^T \tilde{\beta} \tag{3.14}$$

Unreasonably large estimates for $\tilde{\beta}$ will appear to have a large sum-of-squares error and be rejected. Consider two datasets which differ only in the scale of the outcome variable, for example $\mathbf{X}, \mathbf{y}$ and $(10^6 \times \mathbf{X}), (10^6 \times \mathbf{y})$. Because we do not penalize the offset parameter β_0, the same slope parameters $\tilde{\beta}$ will minimize Equation 3.14 for both datasets.

Ridge regression was originally developed to overcome singularity when inverting $\mathbf{X}^T\mathbf{X}$ to compute the covariance matrix. In this setting the ridge coefficient λ is a perturbation of the diagonal entries of $\mathbf{X}^T\mathbf{X}$ to encourage non-singularity [10]. The covariance matrix in a ridge regression setting is $(\mathbf{X}^T\mathbf{X} - \lambda\mathbf{I})^{-1}$. Using this formulation, Ridge regression may be seen as belonging to an older family of techniques called *regularization* [34].

Another interpretation of Ridge regression is available through Bayesian point estimation. In this setting the belief that β should be small is coded into a *prior distribution*. In particular, suppose the parameters $\beta_1, \ldots, \beta_M$ are treated as independent Normally-distributed random variables with zero mean and known variance τ^2. Suppose further that the outcome vector $\mathbf{y}$ is distributed as described in Section 3.1 but with known variance σ^2. The prior distribution allows us to weight the likelihood $\mathbb{L}(\mathbf{y}, \mathbf{X}, \hat{\beta}, \sigma^2)$ of the data by the probability of $\hat{\beta}$. After normalization, this weighted likelihood is called the *posterior distribution*. The posterior distribution of $\hat{\beta}$ is

$$f_{\text{posterior}}(\beta \mid \mathbf{y}, \mathbf{X}, \sigma^2, \tau^2) = (\mathbf{y} - \hat{\beta}^T\mathbf{X})^T(\mathbf{y} - \hat{\beta}^T\mathbf{X}) + \frac{\sigma^2}{\tau^2}\hat{\beta}^T\hat{\beta} \tag{3.15}$$

The Bayesian point estimator for β is the $\hat{\beta}$ with highest probability under the posterior distribution. Since $f_{\text{posterior}}(\beta \mid \mathbf{y}, \mathbf{X}, \sigma^2, \tau^2) = -\text{RSS}_{\text{ridge}}$ with $\lambda = \sigma^2/\tau^2$, the Bayesian point estimator for $\hat{\beta}$ is the same as the least squares estimator given loss function $\text{RSS}_{\text{ridge}}$ and an appropriately chosen prior variance τ^2. [10; 43].

Chapter 4

Logistic Regression

Linear regression is useful for data with linear relations or applications for which a first-order approximation is adequate. There are many applications for which linear regression is not appropriate or optimal. Because the range of the linear model in Equation 3.3 is all of $\mathbb{R}$, using linear regression for data with continuous outcomes in $(0,1)$ or binary outcomes in $\{0,1\}$ may not be appropriate. Logistic regression (LR) is an alternative regression technique naturally suited to such data.

4.1 Logistic Model

Let $\mathbf{X},\mathbf{y}$ be a dataset with binary outcomes. For each experiment $\mathbf{x_i}$ in $\mathbf{X}$ the outcome is either $y_i = 1$ or $y_i = 0$. Experiments with outcome $y_i = 1$ are said to belong to the *positive class*, while experiments with $y_i = 0$ belong to the *negative class*. We wish to create a regression model which allows *classification* of an experiment $\mathbf{x_i}$ as *positive* or *negative*, that is, belonging to either the positive or negative class. Though LR is applicable to datasets with outcomes in $[0,1]$, we will restrict our discussion to the binary case.

We can think of an experiment in $\mathbf{X}$ as a Bernoulli trial with mean parameter $\mu(\mathbf{x_i})$. Thus y_i is a Bernoulli random variable with mean $\mu(\mathbf{x_i})$ and variance $\mu(x_i)(1-\mu(x_i))$. It is important to note that the variance of y_i depends on the mean and hence on the experiment $\mathbf{x_i}$. To model the relation between each experiment $\mathbf{x_i}$ and the expected value of its outcome, we will use the logistic function. This function is written as

$$\mu(\mathbf{x},\beta) = \frac{\exp(\beta^T\mathbf{x})}{1+\exp(\beta^T\mathbf{x})} \tag{4.1}$$

where β is the vector of parameters, and its shape may be seen in Figure 4.1. We assume that $x_0 = 1$ so that β_0 is a constant term, just as we did for linear regression in Section 3.1. Thus our regression model is

$$y = \mu(\mathbf{x},\beta) + \varepsilon \tag{4.2}$$

where ε is our error term. It may be easily seen in Figure 4.2 that the error term ε can have only one of two values. If $y = 1$ then $\varepsilon = 1-\mu(\mathbf{x})$, otherwise $\varepsilon = \mu(\mathbf{x})$. Since y is Bernoulli with mean $\mu(\mathbf{x})$ and variance $\mu(\mathbf{x})(1-\mu(\mathbf{x}))$, the error ε has zero mean and variance $\mu(\mathbf{x})(1-\mu(\mathbf{x}))$. This is different than the linear regression case where the error and outcome had constant variance σ^2 independent of the mean.

Because the LR model is nonlinear in β, minimizing the RSS as defined for linear regression in Section 3.1 is not appropriate. Not only is it difficult to compute the minimum of the RSS, but the RSS minimizer will

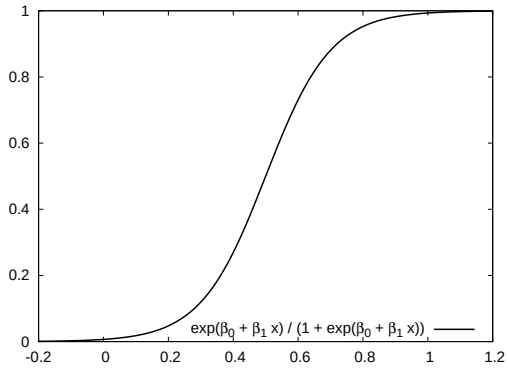


Figure 4.1: Logistic function with one attribute x.

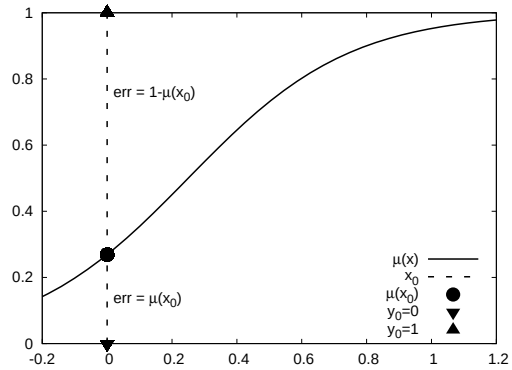


Figure 4.2: The logistic model's binomial error variance changes with the mean $\mu(x)$.

not correspond to maximizing the likelihood function. It is possible to transform the logistic model into one which is linear in its parameters using the *logit* function $g(\mu)$, defined as

$$g(\mu) = \frac{\mu}{1-\mu} \tag{4.3}$$

We apply the logit to the outcome variable and expectation function of the original model in Equation 4.2 to create the new model

$$g(y) = g(\mu(\mathbf{x})) + \tilde{\varepsilon} = \beta^T \mathbf{x} + \tilde{\varepsilon}(\mathbf{x}) \tag{4.4}$$

However, we cannot use linear regression least squares techniques for this new model because the error $\tilde{\varepsilon}$ is not Normally distributed and the variance is not independent of the mean. One might further observe that this transformation is not well defined for $y = 0$ or $y = 1$. Therefore we turn to parameter estimation methods such as maximum likelihood or *iteratively re-weighted least squares*.

It is important to notice that LR is a linear classifier, a result of the linear relation of the parameters and the components of the data. This indicates that LR can be thought of as finding a hyperplane to separate positive and negative data points. In high-dimensional spaces, the common wisdom is that linear separators are almost always adequate to separate the classes.

4.2 Maximum Likelihood Estimation

Recall that the outcome y is a Bernoulli random variable with mean $\mu(\mathbf{x}, \beta)$ in the LR model. Therefore we may interpret the expectation function as the probability that $y = 1$, or equivalently that $\mathbf{x_i}$ belongs to the positive class. Thus we may compute the probability of the ith experiment and outcome in the dataset $\mathbf{X}, \mathbf{y}$ as

$$\begin{aligned} \mathrm{P}(\mathbf{x_i}, y_i | \beta) &= \begin{cases} \mu(\mathbf{x}, \beta) & \text{if } y = 1, \\ 1 - \mu(\mathbf{x}, \beta) & \text{if } y = 0, \end{cases} && (4.5) \\ &= \mu(\mathbf{x}, \beta)^y (1 - \mu(\mathbf{x}, \beta))^{1-y} && (4.6) \end{aligned}$$

From this expression we may derive likelihood and log-likelihood of the data $\mathbf{X}, \mathbf{y}$ under the LR model with parameters β as

$$\mathbb{L}(\mathbf{X}, \mathbf{y}, \beta) = \prod_{i=1}^{R} \mu(\mathbf{x_i}, \beta)^{y_i} (1 - \mu(\mathbf{x_i}, \beta))^{1-y_i} \tag{4.7}$$

$$\ln \mathbb{L}(\mathbf{X}, \mathbf{y}, \beta) = \sum_{i=1}^{R} \Big(y_i \ln(\mu(\mathbf{x_i}, \beta)) + (1 - y_i) \ln(1 - \mu(\mathbf{x_i}, \beta)) \Big) \tag{4.8}$$

The likelihood and log-likelihood functions are nonlinear in β and cannot be solved analytically. Therefore numerical methods are typically used to find the MLE $\hat{\beta}$. CG is a popular choice, and by some reports CG provides as good or better results for this task than any other numerical method tested to date [27]. The time complexity of this approach is simply the time complexity of the numerical method used.

4.3 Iteratively Re-weighted Least Squares

An alternative to numerically maximizing the LR maximum likelihood equations is the *iteratively re-weighted least squares* (IRLS) technique. This technique uses the Newton-Raphson algorithm to solve the LR score equations. This process is explained in more detail below.

To compute the LR score equations we first differentiate the log-likelihood function with respect to the parameters

$$\frac{\partial}{\partial \beta_j} \ln \mathbb{L}(\mathbf{X}, \mathbf{y}, \beta) = \sum_{i=1}^{R} \left(y_i \frac{\frac{\partial}{\partial \beta_j} \mu(\mathbf{x_i}, \beta)}{\mu(\mathbf{x_i}, \beta)} - (1 - y_i) \frac{\frac{\partial}{\partial \beta_j} \mu(\mathbf{x_i}, \beta)}{1 - \mu(\mathbf{x_i}, \beta)} \right) \tag{4.9}$$

$$= \sum_{i=1}^{R} \left(y_i \frac{x_{ij}\, \mu(\mathbf{x_i}, \beta)\, (1 - \mu(\mathbf{x_i}, \beta))}{\mu(\mathbf{x_i}, \beta)} - (1 - y_i) \frac{x_{ij}\, \mu(\mathbf{x_i}, \beta)\, (1 - \mu(\mathbf{x_i}, \beta))}{1 - \mu(\mathbf{x_i}, \beta)} \right) \tag{4.10}$$

$$= \sum_{i=1}^{R} x_{ij} (y_i - \mu(\mathbf{x_i}, \beta)) \tag{4.11}$$

where $j = 1, \ldots, M$ and M is the number of parameters. We then set each of these partial derivatives equal to zero. If we extend the definition of μ such that $\mu(\mathbf{X}, \beta) = (\mu(\mathbf{x_1}, \beta), \ldots, \mu(\mathbf{x_R}, \beta))^T$, we may write the score equations as

$$\mathbf{X}^T (\mathbf{y} - \mu(\mathbf{X}, \beta)) = 0 \tag{4.12}$$

Define $\mathbf{h}(\beta)$ as the vector of partial derivatives shown in Equation 4.11. Finding a solution to the LR score equations is equivalent to finding the zeros of $\mathbf{h}$. Because the LR likelihood is convex [27], there will be only one minimum and hence exactly one zero for $\mathbf{h}$. The Newton-Raphson algorithm finds this optimum through repeated linear approximations. After an initial guess $\hat{\beta}_0$ is chosen, each Newton-Raphson iteration updates its guess for $\hat{\beta}_k$ via the first-order approximation

$$\hat{\beta}_{i+1} = \hat{\beta}_i + \mathrm{J}_{\mathbf{h}}(\hat{\beta}_i)^{-1} \mathbf{h}(\hat{\beta}_i) \tag{4.13}$$

where $\mathrm{J}_{\mathbf{h}}(\hat{\beta})$ is the *Jacobian* of $\mathbf{h}$ evaluated as $\hat{\beta}$. The Jacobian is a matrix, and each row is simply the transposed gradient of one component of $\mathbf{h}$ evaluated at $\hat{\beta}$. Hence the ij-element of $\mathrm{J}_{\mathbf{h}}(\hat{\beta})$ is $-\sum_{i=1}^{R} x_{ij} x_{ik} \mu(\mathbf{x_i}, \beta)(1 -$

$\mu(\mathbf{x_i}, \beta))$. If we define $w_i = \mu(\mathbf{x_i}, \beta)(1 - \mu(\mathbf{x_i}, \beta))$ and $\mathbf{W} = \text{diag}(w_1, \ldots, w_R)$, then the Jacobian may be written in matrix notation as $-\mathbf{X}^T\mathbf{W}\mathbf{X}$. Using this fact and Equation 4.12 we may rewrite the Newton-Raphson update formula as

$$\hat{\beta}_{i+1} = \hat{\beta}_i + (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{y} - \mu(\mathbf{X}, \hat{\beta})) \tag{4.14}$$

Since $\hat{\beta}_i = (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}\mathbf{X}\hat{\beta}_i$ we may rewrite Equation 4.14 as

$$\begin{aligned} \hat{\beta}_{i+1} &= (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{W}\mathbf{X}\hat{\beta}_i + (\mathbf{y} - \mu(\mathbf{X}, \hat{\beta}))) && (4.15) \\ &= (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}\mathbf{z} && (4.16) \end{aligned}$$

where $\mathbf{z} = \mathbf{X}\hat{\beta}_i + \mathbf{W}^{-1}(\mathbf{y} - \mu(\mathbf{X}, \hat{\beta}_i))$ [10; 30; 25]. The elements of vector $\mathbf{z}$ are often called the *adjusted dependent covariates*, since we may view Equation 4.16 as the weighted least squares problem from Section 3.1 with dependent variables, or *covariates*, $\mathbf{z}$. Newton-Raphson iterations may be stopped according to any of several criteria such as convergence of $\hat{\beta}$ or the LR log-likelihood from Equation 4.8.

Solving Equation 4.16 required a matrix inversion or Cholesky back-substitution, as well as several matrix vector multiplications. As a result, IRLS as described has time complexity $\mathrm{O}\left(M^3 + MR\right)$. We have commented several times in this thesis that $\mathrm{O}\left(M^3\right)$ is unacceptably slow for our purposes. A simple alternative to the computations of Equation 4.16 will be presented in Chapter 5.

4.4 Logistic Regression for Classification

Among the strengths of LR is the interpretability of the parameter estimates. Because the expectation function $\mu(\mathbf{x}, \beta)$ is the mean of a Bernoulli random variable, μ is a probability. A deeper analysis of fitted LR models is possible if the model's attributes were chosen carefully to avoid certain types of interactions and correlations. Hosmer and Lemeshow [13] and Myers et al. [30] explain in detail how to create LR models and measure the significance of a fitted LR model. Hosmer and Lemeshow [13] covers in depth the interpretation of a fitted model's coefficients in a wide variety of settings.

We are interested in LR for data-mining and high-dimensional classification. In these settings it is unlikely that the dataset's attributes were carefully chosen, or that one can sift through all combinations of hundreds-of-thousands of attributes to avoid harmful interactions and correlations. For this reason we must assume that our datasets will have correlated or even identical variables. Our experiment matrix $\mathbf{X}$ will have linear dependencies and $\mathbf{X}^T\mathbf{X}$ will only be positive semidefinite. The columns of $\mathbf{X}$ will be badly scaled. The outcomes $\mathbf{y}$ used for estimating β may have as many false positives as true positives. Furthermore, all of these shortcomings should be handled without human intervention.

For the reasons stated above, we will not explore model adequacy, significance or interpretability. Among the many interesting statistics available for LR model analysis, we are only interested in the loss function. This function will be part of the termination criterion for our parameter estimation methods described in Sections 4.2 and 4.3. Both LR and linear regression are members of a family of models called *generalized linear models*, and the loss functions for members of this family are defined as a scaled likelihood ratio known as the *deviance*. In particular, the deviance is defined as

$$\text{DEV}(\hat{\beta}) = 2\ln\left(\frac{\mathbb{L}_{\text{max}}}{\mathbb{L}_{\hat{\beta}}}\right)\phi \tag{4.17}$$

where $\mathbb{L}_{\text{max}}$ is the maximum likelihood achievable, $\mathbb{L}_{\hat{\beta}}$ is the likelihood for the actual model and parameter estimate being used, and ϕ is the *dispersion parameter* [25]. For the linear model in Equation 3.5 the dispersion parameter is defined as σ^2, and for the logistic model of Equation 4.2 it is 1. Using these definitions

we can compute the deviance for a linear model to be exactly the RSS. For logistic regression with binary outputs the deviance expands to

$$\begin{aligned} \mathrm{DEV}(\hat{\beta}) &= -2\ln\mathbb{L}(\mathbf{X},\mathbf{y},\hat{\beta}) && (4.18) \\ &= -2\sum_{i=1}^{R}\Big(y_i\ln(\mu(\mathbf{x_i},\beta)) + (1-y_i)\ln(1-\mu(\mathbf{x_i},\beta))\Big) && (4.19) \end{aligned}$$

[13]. Just as minimizing the RSS for linear regression was equivalent to maximizing the linear regression likelihood function, minimizing the LR deviance is equivalent to maximizing the LR likelihood function. For the remainder of this thesis, deviance should be understood to mean the LR deviance.

Since the deviance is just the negative log-likelihood, using either quantity to terminate iterative parameter estimation techniques will result in the same $\hat{\beta}$. Because deficiencies in the data matrix could cause the LR score equations shown in Equation 4.12 to have multiple solutions $\hat{\beta}$, waiting until $\hat{\beta}$ converges to terminate iterative methods may not be efficient. We prefer to terminate when the relative difference of the deviance $(\mathrm{DEV}(\hat{\beta}_i) - \mathrm{DEV}(\hat{\beta}_{i+1}))/\mathrm{DEV}(\hat{\beta}_{i+1})$ is sufficiently small. Determining what constitutes sufficiently small will be discussed later in this thesis.

LR is susceptible to the same parameter estimation problems caused by correlated attributes as is linear regression [13; 27]. In Section 3.1 we discussed a technique called ridge regression to prevent overly large parameter estimates. A similar technique can be used when estimating β for LR. The simplest way to apply a ridge regression penalty to LR maximum likelihood estimation is to add it to the loss function, which is the deviance. This corresponds to subtracting it from the log-likelihood function, allowing generic convergence criteria to be applied for numerical techniques. When using IRLS for parameter estimation there is a weighted least squares linear regression subproblem to which traditional ridge regression penalties may be applied. Unfortunately this is not equivalent to performing the IRLS derivation of Section 4.3 with the penalty applied to the LR log-likelihood function. Instead of deriving a custom IRLS which incorporates the penalty in the likelihood function, we will use the simpler approach of using traditional ridge regression when solving the weighted least squares subproblem.

Now that we have defined a termination criterion and described methods of applying ridge regression, we may summarize our two methods of find the LR MLE. Algorithm 3 describes the direct maximum likelihood estimation approach of Section 4.2, while Algorithm 4 describes the IRLS approach of of Section 4.3. Lines 3.1 and 4.1 in these algorithms include the regularization methods described above. While the direct method appears much simpler, it can be difficult to find a good iterative method for the nonlinear objective function shown in line 3.1. Despite the additional lines of computation for IRLS, the weighted least squares subproblem in line 4.1 is simply a linear system of equations.

We will be discussing various implementations of LR throughout this thesis. We have created a tree-figure depicting the relationship of our methods and modifications. Figure 4.3 shows the beginning of this tree. Currently there is only a mention of LR and the nonlinear deviance minimization problem, followed by a choice of generic nonlinear methods on the right and IRLS on the left. Additional nodes will be added to this tree in in Chapter 5.

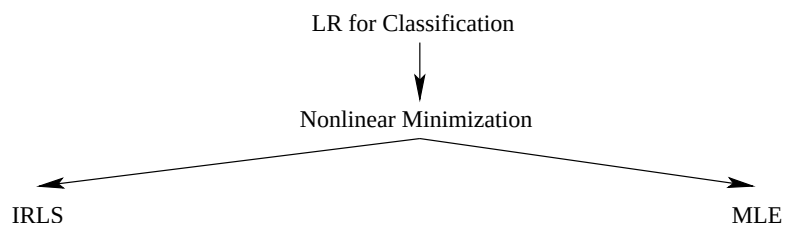


Figure 4.3: Tree showing relationships choices when implementing LR. This tree will be updated in later chapters when we describe LR implementation details or modifications.

input : $\mathbf{X}$, $\mathbf{y}$, $\hat{\beta}_0$, regularization parameter λ, termination parameter ε

output : $\hat{\beta}$ which maximizes LR likelihood

i := 0
$\mathrm{DEV}_{-1} := \infty$
$\mathrm{DEV}_0 := \mathrm{DEV}(\hat{\beta}_0)$

while $|(\mathrm{DEV}_{i-1} - \mathrm{DEV}_i)/\mathrm{DEV}_i| < \varepsilon$ **do**

3.1 Compute $\hat{\beta}_{i+1}$ using iterative method on $\ln \mathbb{L}(\mathbf{X}, \mathbf{y}, \hat{\beta}) - \lambda \hat{\beta}^T \hat{\beta}$
$\mathrm{DEV}_{i+1} = \mathrm{DEV}(\hat{\beta}_{i+1})$
$i := i + 1$

Algorithm 3: Finding the LR MLE directly.

input : $\mathbf{X}$, $\mathbf{y}$, $\hat{\beta}_0$, regularization parameter λ, termination parameter ε

output : $\hat{\beta}$ which maximizes LR likelihood

i := 0
$\mathrm{DEV}_{-1} := \infty$
$\mathrm{DEV}_0 := \mathrm{DEV}(\hat{\beta}_0)$

while $|(\mathrm{DEV}_{i-1} - \mathrm{DEV}_i)/\mathrm{DEV}_i| < \varepsilon$ **do**

$w_{ij} := \mu(\mathbf{x_j}, \hat{\beta}_i)(1 - \mu(\mathbf{x_j}, \hat{\beta}_i),\ j = 1..R$
$\mathbf{W_i} := \mathrm{diag}(w_{i1}, \ldots, w_{iR})$
$\mathbf{z_i} := \mathbf{X}\hat{\beta}_i + \mathbf{W}^{-1}(\mathbf{y} - \mu(\mathbf{X}, \hat{\beta}_i))$

4.1 Compute $\hat{\beta}_{i+1}$ as solution to weighted linear regression $(\mathbf{X}^T \mathbf{W_i} \mathbf{X} + \lambda \mathbf{I})\hat{\beta}_{i+1} = \mathbf{X}^T \mathbf{W_i} \mathbf{z_i}$
$\mathrm{DEV}_{i+1} = \mathrm{DEV}(\hat{\beta}_{i+1})$
$i := i + 1$

Algorithm 4: Finding the LR MLE via IRLS.

Part III

Logistic Regression For Fast Classification

Chapter 5

Logistic Regression Computation

5.1 Preliminaries

5.1.1 Why LR?

A wide variety of classification algorithms exist in the literature. Probably the most popular and among the newest is support vector machines (SVM). Older learning algorithms such as k-nearest-neighbor (KNN), decision trees (DTREE) or Bayes' classifier (BC) are well understood and widely applied. One might ask why we are motivated to use LR for classification instead of the usual candidates.

That LR is suitable for binary classification is made clear in Chapter 4. Our motivation for exploring LR as a fast classifier to be used in data mining applications is its maturity. LR is already well understood and widely known. It has a statistical foundation which, in the right circumstances, could be used to extend classification results into a deeper analysis. We believe that LR is not widely used for data mining because of an assumption that LR is unsuitably slow for high-dimensional problems. In Zhang and Oles [49], the authors observe that many information retrieval experiments with LR lacked regularization or used too few attributes in the model. Though they address these deficiencies, they still report that LR is "noticeably slower" than SVM. We believe we have overcome the stability and speed problems reported by other authors.

5.1.2 Our Approach

We have tried several LR techniques in our quest for reliability and speed. These include

- IRLS using Cholesky decomposition and back-substitution to solve the weighted least squares subproblem (WLS)
- IRLS with Modified Cholesky and back-substitution for WLS
- Stepwise IRLS with Modified Cholesky and back-substitution for WLS
- Divide-and-conquer IRLS with Modified Cholesky and back-substitution for WLS
- IRLS with linear CG for WLS
- Direct minimization of deviance using nonlinear CG

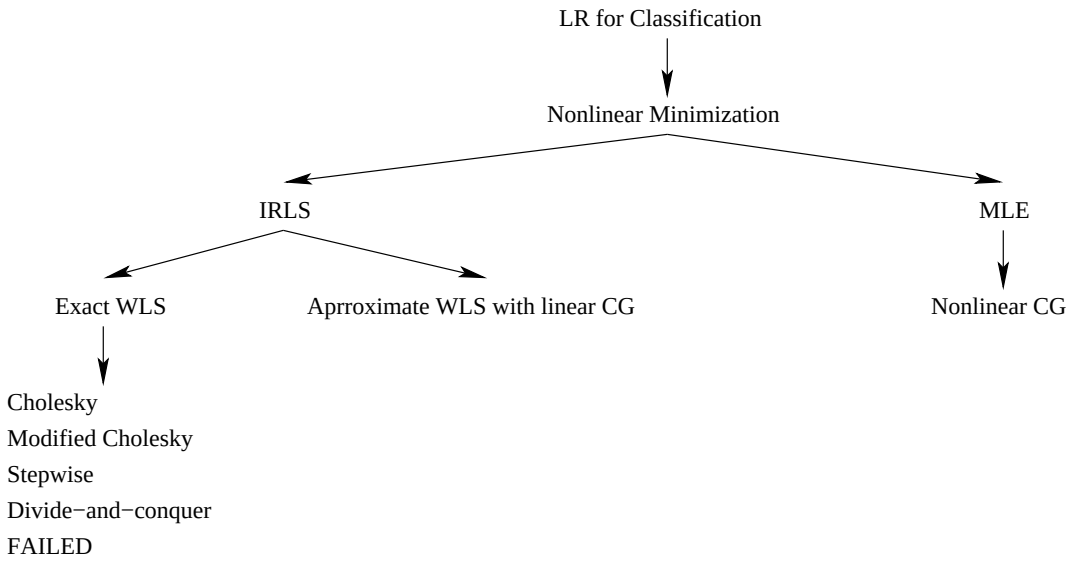


Figure 5.1: Tree showing overview of our LR implementation efforts. As this chapter progresses, more information will be added to this tree. The left-most branch lists several initial implementations that were rejected.

The relationship of these methods to MLE and IRLS are depicted in Figure 5.1, and are described below.

Initial experiments with unregularized IRLS implemented using a Cholesky decomposition and triangular back-substitution failed due to linear dependencies among the columns of our data. The linear dependencies cause the matrix $\mathbf{A} = \mathbf{X}^T\mathbf{W}\mathbf{X}$ to be positive semidefinite instead of positive definite, and the Cholesky decomposition fails. We tried a column-elimination strategy to remove dependencies. In our first attempt, we removed dependencies from sparse datasets using an exact version of Gaussian elimination. Later, we combined the column elimination with the Cholesky decomposition. This Modified Cholesky technique is described in Komarek and Moore [20]. While the latter approach was faster than the former, it was still very slow. We also encountered stability and overfitting problems. In particular, we encountered the model and weight saturation described in Section 5.2.1.1.

Stepwise logistic regression is a model-selection technique where attributes are added one-at-a-time to the model. The model quality is assessed after addition, and the addition is accepted or rejected. Stepwise LR is described in Hosmer and Lemeshow [13]. We implemented this using our Modified Cholesky technique. There are two principal problems with Stepwise LR. The first is deciding in which order attributes should be added to the model, which affects which attributes survive for the final model. The second problem is the continual reduction of performance as attributes are added, and the fact that you must estimate the LR parameters for many models. Later we attempted a tree-based divide-and-conquer strategy. The dataset attributes were recursively partitioned and many two-attribute LR problems were solved. The result was no faster than our existing techniques, and therefore we did not explore optimal methods for recombination of the partitioned results.

Instead of abandoning IRLS, we chose to explore the effects of approximate solutions to the weighted least squares linear regression subproblem. Because this only required solving a linear system, CG appeared

appropriate. This solved many of our numerical and speed problems, but introduced new complexity. We also tried using nonlinear CG to find the deviance minimizers directly, as suggested by McIntosh [26] and Minka [27]. While this removed some of the IRLS complexity and instability, it required *nonlinear* CG and introduced different complexities. To address numerical instability problems in both methods, we tried a variety of LR model modifications as well as ridge regression or similar coefficient shrinkage techniques.

The time complexity of IRLS with CG is simply the time complexity of CG since Equation 4.16 is a linear equation, times the maximum number of IRLS iterations allowed. According to Shewchuk [41] the time complexity of CG is $\mathrm{O}(m)$, where m is the number of nonzero entries in the linear system's matrix $\mathbf{A}$. For IRLS, $\mathbf{A} = \mathbf{X}^T\mathbf{W}\mathbf{X}$. We are ignoring the condition number κ described in Section 2.1.2. This time complexity is a reflection of the time to compute α_i, whose formula is shown in Equation 2.11. Because we can compute $\mathbf{A}\mathbf{d_k}$ as a series of sparse matrix-vector products with dimension M, it is not necessary to incur the M^2 cost of computing $\mathbf{A}\mathbf{x}$ directly. Therefore CG is $\mathrm{O}(m)$. A quick way to understand this complexity is by assuming that the number of CG iterations is essentially constant relative to the size of our data, which has always been true in our experiments, and the most expensive part of CG is the α_i computation.

As we will discuss later, the number of IRLS iterations this number is typically small, around five or ten. If we bound the maximum number of IRLS iterations by thirty, for example, the worst-case complexity is $\mathrm{O}(30 \times m) = \mathrm{O}(m)$. Perhaps it is deceptive to hide the number of IRLS iterations in the asymptotic complexity. On the other hand, we show in Chapter 6 that our IRLS implementation is empirically linear in the number of dataset rows, when the sparsity and number of attributes is fixed. That is, the number of IRLS iterations remains relatively constant across a wide range of dataset sizes. Therefore, we feel it is less misleading to ignore the small number of IRLS iterations when considering asymptotic complexity, especially when the number of IRLS iterations is bounded.

Our combination of IRLS and CG is very similar to *truncated-Newton* methods. Let $\mathbf{p}$ be the vector $\mathrm{J}_{\mathbf{h}}(\hat{\beta}_i)^{-1}\mathbf{h}(\hat{\beta}_i)$, as defined in Section 4.3, Equation 4.13. The vector $\mathbf{p}$ is called the *Newton direction*, and it is often written as the solution to the linear system $\mathrm{J}_{\mathbf{h}}(\hat{\beta}_i)\mathbf{p} = -\mathbf{h}(\hat{\beta}_i)$. This system can be solved with an iterative method. Because $\hat{\beta}$ is itself an approximation, it is not necessary to compute $\mathbf{p}$ exactly and the iterative method may be stopped early. Truncated-Newton methods get their name from this early stopping of the iterative approximation to the Newton direction [31]. The combination of IRLS and CG described earlier does not approximate the Newton direction. Instead, CG is applied to the weighted least squares regression that arises from the combination of the current position and the Newton direction. This linear system is different than that which describes the Newton direction alone, and we do not know whether the behavior of a truncated-Newton method would be identical to our IRLS and CG combination. Furthermore, IRLS is not equivalent to the Newton-Raphson method for all generalized linear models [25]. Applying CG to IRLS may have a better statistical foundation than truncated-Newton methods when generalizing beyond LR.

In the sections below are reports of stability, termination criterion and speed experiments with LR. These experiments will identify features important for successful use of LR in data mining and other high-dimensional classification tasks. The following chapter contains characterization and comparison experiments for LR, SVM, KNN and BC. In that chapter we will empirically demonstrate the utility of LR for these tasks.

Throughout the remainder of this thesis, references to IRLS imply use of CG for the weighted least squares linear regression subproblem. All references to maximum likelihood estimation, which will share the abbreviation MLE with maximum likelihood estimates, use nonlinear CG to solve the LR score equations. Exceptions to these conventions will be explicitly noted. The linear and nonlinear CG implementations are our own. We tried the CG implementation in the GNU Scientific Library [8] and found it was somewhat slower and more difficult to use.

5.1.3 Datasets

We will illustrate the strengths and weaknesses of LR for data mining and high-dimensional classification through life sciences and link detection datasets. We will characterize the performance of LR and other classification algorithms using several synthetic datasets. When discussing datasets in this thesis, each record belongs to, or is predicted to belong to, 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*. R is the number of rows in the dataset, and M is the number of attributes. The *sparsity factor* F is the proportion of nonzero entries. Thus the total number of nonzero elements in a dataset is the product MRF. Our datasets are summarized in Tables 5.1 and 5.2.

5.1.3.1 Life sciences

The life sciences data mining was performed on two datasets similar to the publicly available Open Compound Database and associated biological test data, as provided by the National Cancer Institute [32]. Both `ds1` and `ds2` are sparse and have binary features. Their dimensions and sparsity are shown in Table 5.1. Also in this table are two datasets derived from `ds1`, denoted `ds1.10pca` and `ds1.100pca`. These datasets are linear projections of `ds1` to 100 and 10 dimensions, using principal component analysis (PCA), and hence are dense with real-valued inputs.

5.1.3.2 Link detection

This thesis uses two publicly available datasets for *link detection* experiments. A *link* is a list of objects related to one another, for instance a group of names that co-occur in a newspaper article. In the link detection problem we analyze links to determine if a target object is related to each given link. The classifier is trained on data in which the target object has been removed, and every link which formerly included the target will have a positive outcome variable $y_i = 1$. For the links which did not include the target, we set $y_i = 0$. We assume that our datasets are noisy: links may contain unrelated objects and omit related objects.

The first link detection dataset used in this thesis is `citeseer`. This dataset is derived from the CiteSeer web site [4] and lists the names of collaborators on published materials. We extracted the most common name, `J_Lee`, to use as the target. The goal is to predict whether `J_Lee` was a collaborator for each work based on others listed for that work. The second dataset is derived from the Internet Movie Database [14] and is denoted `imdb`. Each row represents a work of entertainment, and the attributes represent people associated with that work. Again we chose the most frequently appearing attribute, `Blanc__Mel`, for the target.

We have previously published related link analysis work using LR. Kubica et al. [22] discusses the problem of *link completion*. In this task we are asked to sort a group of objects according their likelihood of having been removed from a link. This is equivalent to *collaborative filtering* [2]. Link completion is a multiclass problem and is somewhat different than the binary classifications problems we discuss in this thesis. While it is not common for LR to be used in this domain, our implementation is fast enough to make it possible.

5.1.3.3 Synthetic Datasets

We use four groups of synthetic datasets for characterization of LR and other learning algorithms. The datasets are described further below. To create these datasets, the method of [36] is employed. A random tree is generated with one node per dataset attribute. Each row of the dataset is generated independently of the other rows. Two parameters, the *coupling* c and the *sparsity* s, allow control over the properties of the generated dataset. To create one row the following steps are taken:

Table 5.1: Datasets.

Name	Attributes	Rows	Sparsity	Num Nonzero	Num Pos Rows
ds1	6,348	26,733	0.02199	3,732,607	804
ds1.100pca	100	26,733	1.00000	2,673,300	804
ds1.10pca	10	26,733	1.00000	267,330	804
ds2	1,143,054	88,358	0.00029	29,861,146	423
citeseer	105,354	181,395	0.00002	512,267	299
imdb	685,569	167,773	0.00002	2,442,721	824

1. Set the parent node (attribute) equal to 1 with probability s.
2. Traverse the remaining tree nodes in topological order. For each remaining node,
   (a) If the parent node has value 1, set this node's value to 1 with probability $\min(s+2c,1)$.
   (b) If the parent node has value 0, set this node's value to 0 with probability $\max(s-2c,0)$.

We require that $s \in [0,1]$ and $c \in [0,0.5]$. Setting the sparsity s to one implies all rows will be all ones, and setting $s = 0$ implies all rows will be all zeros. If $s = 0.5$ and the coupling c is one-half, then on average half of the rows will be all ones and half will be all zeros. If $c = 0$ then each attribute is independent of the other attributes, and values of c strictly between 0.0 and 0.5 create probabilistic dependencies between attributes.

Once the attributes have been generated, we generate the outcomes $\mathbf{y}$ such that a specified number N are 1. To do this we create a random vector of M weights $\mathbf{b}$ between -1.0 and 1.0. This vector represents the true linear weights if a linear model were used. For each row i, the inner product $\mathbf{b}^T\mathbf{x_i}$ is computed and stored. Finally y_i is set to 1 if $\mathbf{b}^T\mathbf{x_i}$ is among the top N inner products. While this noiseless linear weighting scheme does a poor job of simulating real-world datasets, it is adequate. Because the synthetic data is linearly separable, all classifiers we are interested in should be able to represent the data. The algorithms will have to accommodate a truly linearly separable dataset, which may require some regularization or other numerical controls. Finally, common wisdom tells us that with enough dimensions most real-world datasets are linearly separable anyway.

Our synthetic datasets were designed to test how algorithm performance varies with four factors. These factors are the number of rows, the number of attributes, the sparsity and the coupling. Each of the four groups of synthetic datasets fixes the value for three of these properties and varies the forth. These datasets are summarized in Table 5.2. In this table each dataset group is represented by one line. For the "numrows" group, the "Rows" column shows the range "1,000-1,000,000". There are ten datasets in this group, having one-thousand, two-thousand, five-thousand, ..., one-million rows. The same progression is used for the seven datasets in the "numatts" group. The sparse_x datasets also use a 1,2,5 progression. The coupled_x datasets have couplings of $0.00, 0.05, \ldots, 0.45$.

5.1.4 Scoring

All experiments in this thesis are ten-fold cross-validations. The predictive performance of the experiments is measured using the Area Under Curve (AUC) metric, which is described below. All times are reported in seconds, and details about timing measurements may be found in Section 5.1.5.

Before we describe AUC scores, we must first describe Receiver Operating Characteristic (ROC) curves [5]. We will use the ROC and AUC description we published in [20]. To construct an ROC curve, the dataset

Table 5.2: Synthetic Datasets.

Name	Attributes	Rows	Sparsity s	Coupling c	Num Datasets	Num Pos Rows
`numrows_x`	10^3	$10^3 - 10^6$	0.01	0.00	10	50%
`numatts_x`	$10^3 - 10^5$	10^5	0.01	0.15	7	10,000
`sparse_x`	10^3	10^5	0.0001 - 0.1	0.00	10	10,000
`coupled_x`	10^3	10^5	0.01	0.00 - 0.45	10	10,000

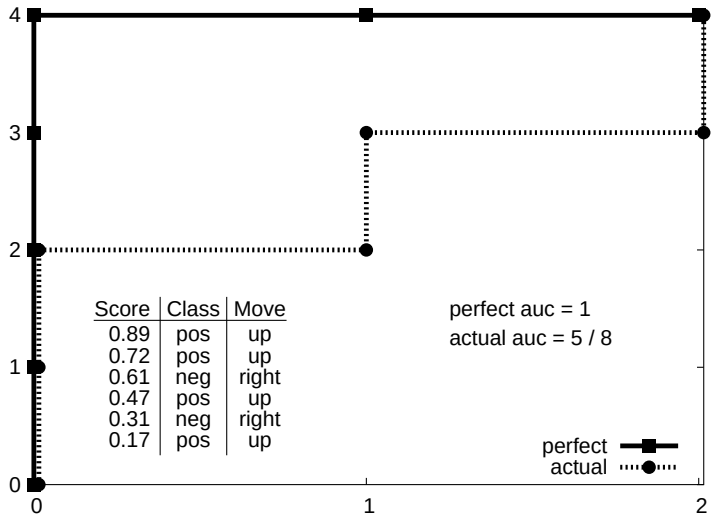


Figure 5.2: Example ROC curve.

rows are sorted according to the probability a row is in the positive class under the learned logistic model. Starting at the graph origin, we examine the most probable row. If that row is positive, we move up. If it is negative, we move right. In either case we move one unit. This is repeated for the remaining rows, in decreasing order of probability. 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.

Figure 5.2 shows an example ROC curve. Six predictions are made, taking values between 0.89 down to 0.17, and are listed in the first column of the table in the lower-left of the graph. The actual outcomes are listed in the second column. The row with highest prediction, 0.89, belongs to the positive class. Therefore we move up from the origin, as written in the third column and shown by the dotted line in the graph moving from (0,0) to (1,0). The second favorite row was positive, and the dotted line moves up again to (2,0). The third row, however, was negative and the dotted line moves to the right one unit to (2,1). This continues until all six predictions have been examined.

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 $(R-P,P)$. The solid line in Figure 5.2 illustrates the path of a perfect learner in our example with six

predictions. 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 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 the example shown in Figure 5.2, the dotted line representing the real learner encloses an area of 5. The solid line for the perfect learner has an area of 8. Therefore the AUC for the real learner in our example is 5/8.

Whereas metrics such as precision and recall measure true positives and negatives, the AUC measures the ability of the classifier to correctly rank test points. This is very important for data mining. We often want to discover the most interesting galaxies, or the most promising drugs, or the products most likely to fail. When presenting results between several classification algorithms, we will compute confidence intervals on AUC scores. For this we compute one AUC score for each fold of our 10-fold cross-validation, and report the mean and a 95% confidence interval using a T distribution.

5.1.5 Computing Platform

A goal of this thesis is to provide a reasonable and repeatable baseline measurement of classification with LR. To meet this goal, we use a single computing platform for all of our experiments. This platform consists of dual AMD Opteron 242 processors with 4GB or 8GB of memory running GNU/Linux. For machines of either memory size the memory modules are split across the two processors, due to the Opteron's NUMA configuration. The BIOS is set to interleave accesses within each local memory bank, and to interleave memory accesses between each processor's local memory. This technique prevents dramatic discrepancies in process performance when the operating system kernel runs the process on one cpu but allocates memory from the other processor's local bank.

Linux kernel version 2.4.23 or later is used in 64-bit mode. Executables are compiled with the GNU C compiler version 3.2.2 or later, using the `-O2` flag alone for optimization. All of our experiments fit within 4GB of ram and machines of either memory size behave identically.

There is some uncertainty in the timing of any computer process. In general we are unconcerned with variations in time below ten percent. Although it is common to use "user" time or "cpu" time to measure experiment speed, we use the "real" time required for computations within each fold, neglecting fold preparation and scoring overhead. When making performance comparisons, we run only one experiment per machine and the elapsed algorithm execution time should not differ significantly from the "cpu" time. This strategy should provide a good assessment of how long a user will wait for computations to finish.

Our test machines were purchased at two different times. A discrepancy between BIOS versions, discovered while writing the final chapter of this thesis, caused inconsistent timings for some experiments. We report over two thousand results in this thesis, and are unable to repeat all of the affected experiments. Instead, we have repeated a subset of the affected experiments such that all results in Chapter 5 are for the old, slower BIOS, and all results in Chapter 6 use the new, faster BIOS.

We use a Postgresql 7.3 database to store the programs, datasets, input and output for each experiment. This database is available for inspection of all experiments performed for this document, as well as many experiments not included here.

5.1.6 Scope

In the sections and subsections that follow, we explore many ways to improve the stability, accuracy and speed of LR computations. This presentation is broken into two sections for our two LR parameter estimation methods. Section 5.2 discusses variations on IRLS with CG. We will refer to this combination simply as IRLS. Section 5.3 discusses variations on MLE, where CG is the numerical method used to find the optimum parameter estimate. This combination will be called CG-MLE. We will be using the datasets, scoring method, and computing platform described above.

For the rest of this chapter, "parameters" will no longer refer to the LR parameters β. The parameters discussed below are implementation parameters that control which variations of LR computations are being used or how the computations proceed. For example, the `modelmax` parameter makes an adjustment to the LR expectation function, while the `cgeps` parameter is an error bound used for termination of CG iterations. Our goal in exploring these variations is to choose an implementation which is stable, correct, fast and autonomous. Since an autonomous classifier cannot require humans to micro-manage run-time parameters, we will seek default settings which meet our stability, correctness and speed goals on a wide variety of datasets. The six real-world datasets described in Section 5.1.3 will be used to evaluate our implementation and support our decisions.

In the IRLS and CG-MLE sections we divide the implementation parameters into three categories, according to their proposed purpose. These categories are

1. controlling the stability of computations
2. controlling termination and optimality of the final solution
3. enhancing speed

Many of the parameters belong to multiple categories. For example, proper termination of CG requires numerically stable iterations, and hence depends on stability parameters. Each parameter will be discussed in the context that motivated its inclusion in our experiments.

The parameters in each category will be thoroughly tested for effectiveness. Parameters which consistently enhance performance for all of the datasets will have default values assigned. These defaults will be chosen after further empirical evaluation, with optimality of the AUC score preferred over speed. Each section ends with a summary of the useful techniques and the default values chosen for the corresponding parameters. Our final LR implementations will be characterized and compared in Chapter 6.

5.2 IRLS Parameter Evaluation and Elimination

5.2.1 Indirect (IRLS) Stability

In the following sections we will describe the stability challenges we encountered while developing our IRLS algorithm, along with our proposed solutions. This is followed by an evaluation of these solutions, and a summary of our conclusions. For convenience we have summarized the proposed stability parameters in Table 5.3. Figure 5.3 shows our LR description tree, with a box indicating the branch relevant to this section.

5.2.1.1 Stability Parameter Motivation

The earliest problems we encountered in our implementations were floating point underflow and overflow. Because the datasets that concern us often have ten-thousand or more attributes, the discriminant $\eta = \beta_0 +$

Table 5.3: IRLS Parameters

Parameter	Description
`modelmin`	Lower threshold for $\mu = \exp(\eta)/(1+\exp(\eta))$
`modelmax`	Upper threshold for $\mu = \exp(\eta)/(1+\exp(\eta))$
`wmargin`	Symmetric threshold for weights $w = \mu(1-\mu)$
`margin`	Symmetric threshold for outcomes y
`binitmean`	Initialize β_0 to $E(\mathbf{y})$
`rrlambda`	Ridge-regression parameter λ
`cgwindow`	Number of non-improving iterations allowed
`cgdecay`	Factor by which deviance may decay during iterations
`lreps`	Deviance epsilon for IRLS iterations
`cgeps`	Residual epsilon for CG iterations
`cgdeveps`	Deviance epsilon for CG iterations
`lrmax`	Maximum number of IRLS iterations
`cgmax`	Maximum number of CG iterations
`cgbinit`	CG iterations for IRLS iteration i start where IRLS iteration $i-1$ stopped

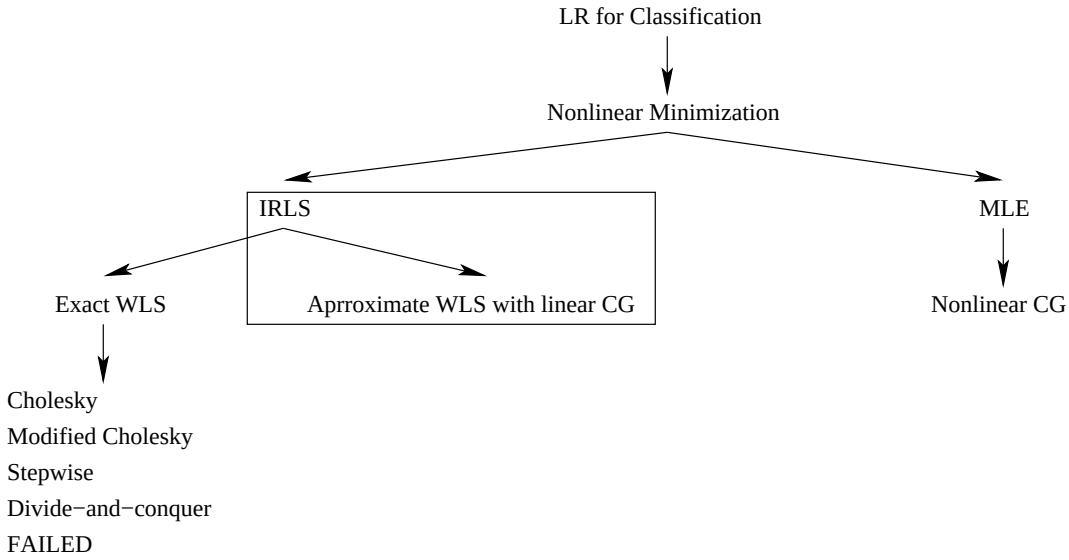


Figure 5.3: LR tree with rectangle marking the beginning of our IRLS implementation.

$\beta_1 x_1 + \ldots + \beta_M x_M$ can easily grow large. When computing $\exp(\eta)$ in the LR expectation function of Equation 4.1, a discriminant greater than 709 is sufficient to overflow IEEE 754 floating point arithmetic, the standard for modern computer hardware. The logistic function $\mu = \exp(\eta)/(1+\exp(\eta))$ becomes indistinguishable from 1 when the discriminant reaches 37. IEEE 754 denormalized floating point values allow discriminants as low as -745 to be distinguished from zero; however, denormalized values have a decreased mantissa which compromises accuracy. When the logit reaches 0 or 1, the weights $w = \mu(1-\mu)$ become 0 and $\mathbf{X}^T\mathbf{W}\mathbf{X}$ is guaranteed to be positive semidefinite. Worse, the IRLS adjusted dependent covariates $z = \eta + (y-\mu)/w$ become undefined. [11]

A further problem arises for binary outputs. Because the logit function approaches but does not reach the output values, it is possible that model parameters are adjusted unreasonably high or low. This behavior causes scaling problems because not all parameters will necessarily grow together. McIntosh [26] solved a similar problem by removing rows whose Binomial outcomes were saturated at 100%. This technique is not appropriate for Bernoulli outcomes that are identically one or zero.

We will explore several approaches to solving the stability challenges inherent in IRLS computations on binary datasets. We will introduce our stability parameters in the following paragraphs, with in-depth test results for each presented further below. The stability parameters are `modelmin`, `modelmax`, `wmargin`, `margin`, `binitmean`, `rrlambda`, `cgwindow` and `cgdecay`.

The `modelmin` and `modelmax` parameters are used to threshold the logistic function above and below, guaranteeing μ will always be distinguishable from zero and one. The `wmargin` parameter imposes a similar symmetric thresholding for the weights. The `margin` parameter shrinks the outputs toward one-half by the specified amount, allowing them to fall within the range achievable by the logit. Since non-binary outputs can be interpreted in LR as the observed mean of a binomial experiment, the `margin` parameter can be viewed as adding uncertainty to each experiment's outcome. All of these parameters have the unfortunate side-effect of changing the shape of the optimization surface. In particular, the `modelmin` and `modelmax` parameters destroy convexity and create a plane once β becomes large.

Normally the LR parameter vector β_0 is initialized to zero before IRLS iterations begin. McIntosh [26] reports that initializing the offset β_0 component to the mean of the outcomes $\mathbf{y}$ eliminated overflow errors during CG-MLE computations. To test whether this technique affects IRLS we have the `binitmean` flag. Also motivated by literature is the standard regularization technique of ridge-regression, discussed in Section 3.2. The ridge-regression weight λ is set using the `rrlambda` parameter. Larger `rrlambda` values impose a greater penalty on oversized estimates $\hat{\beta}$.

All of the previous parameters attempt to control the behavior of the model and optimizer. The two remaining stability parameters exist to stop optimization when all else fails. If `cgwindow` consecutive iterations of CG fail to improve the deviance, CG iterations are terminated. Similarly if a CG iteration increases the deviance by more than the parameter `cgdecay` times the best previous deviance, then CG iterations are terminated.

Though not evaluated in this section, we cannot avoid mentioning the termination parameters `lreps` and `cgeps`. A full description of these will wait until Section 5.2.2. In short these parameters are used to control when IRLS quits searching for the optimal LR parameter vector β. The smaller these values are, the more accurate our estimate of β. Smaller values also result in more IRLS and CG iterations. On the other hand if `lreps` and `cgeps` are set too small and we are not using any regularization, then we may find an LR parameter estimate $\hat{\beta}$ which allows perfect prediction of the training data, including any noisy or incorrect outcomes. This will result in poor prediction performance. One further note regarding the CG termination parameter `cgeps`. It is compared to the Euclidean norm of the CG residual vector $\mathbf{r}_i$, as defined in Section 2.1.2. Because this norm is affected by the dimensionality of the dataset, we originally scaled `cgeps` by the number of attributes. This scaling is not optimal, and will be discussed in detail in Section 5.2.2.

Table 5.4: Binarized stability parameters `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`. The - and x characters are used in later tables to represent these values.

Parameter(s)	"Off" (-) values	"On" (x) values
`modelmin`, `modelmax`	0.0, 1.0	1e-100, 0.99999998
`margin`	0.0	0.001
`rrlambda`	0.0	10.0
`cgwindow`, `cgdecay`	1000, 1000	3, 2

Tuning all of these stability parameters is unreasonable. The following analysis will examine the impact of each of these on six datasets, ultimately ascertaining which are beneficial and whether reasonable default values can be identified.

5.2.1.2 Stability Parameter Tables

Tables 5.5 through 5.8 summarize the majority of our stability experiments on sparse binary datasets. Each of these organizes results for ten-fold cross-validation experiments for combinations of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`. Because the tests are cross-validations, the testing sets for each fold come from the same distribution as the training set for each fold. Each parameter can have one of two values, as shown in Table 5.4, where one value effectively disables it and the other is chosen to illustrate that parameter's effect on computations. The asymmetry seen in the `modelmin` and `modelmax` "on" values is due to the asymmetry of the IEEE 756 floating point representation in which denormalized values allow greater resolution near zero. Note that `cgwindow` and `cgdecay` are disabled by making them very large. Unless stated otherwise, the `binitmean` is disabled and `wmargin` is zero.

The columns of the stability experiment tables are arranged in four groups. The first group has "-" and "x" symbols for each of the binarized parameters, or pairs of parameters. A "-" indicates the parameter or pair of parameters were set to their "off" state as defined in Table 5.4, while "x" indicates the "on" state from the same table. The `mm` column represents the state of the pair `modelmin` and `modelmax`, the `mar` column represents `margin`, `rrl` represent `rrlambda`, and `cgw` represents the pair `cgwindow` and `cgdecay`.

The second, third and forth groups of columns represent the performance attained when the stability parameters are set as indicated by the first group of columns. The title "Loose Epsilon" above the second group indicates that `cgeps` and `lreps` were set to the rather large values 0.1 and 0.5, respectively. The third group uses moderate epsilons, with `cgeps` set to 0.001 and `lreps` set to 0.1. The fourth group has "tight" epsilons, with `cgeps` set to 0.000001 and `lreps` set to 0.0001. The sub-columns of each group represent the AUC score, whether NaN values were encountered during computation, the minimum average deviance achieved during the ten folds of the cross-validation, and the number of real seconds elapsed during computation. We do not provide confidence intervals for the scores because the focus is on stability and not on optimal performance or speed. Our indication of stabile computations is a good score and a good speed, as judged against other results in the same table.

The purpose of the Loose Epsilon, Moderate Epsilon and Tight Epsilon groups is to explore how well stability parameters compensate for different optimality criteria. Once we have analyzed the stability parameters using their binarized value we can explore optimal settings. After this work with the stability parameters is finished, Section 5.2.2 will focus on finding widely-applicable termination criteria which balance optimality and speed.

Table 5.5: IRLS stability experiments for `ds1`. `binitmean` is disabled and `wmargin` is 0. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.897	-	3746	14	0.896	x	821	534	0.894	x	812	542
x	-	-	-	0.897	-	3746	16	0.896	x	821	534	0.894	x	812	560
-	x	-	-	0.897	-	3564	16	0.895	x	759	559	0.894	x	803	562
x	x	-	-	0.897	-	3564	15	0.895	x	759	558	0.894	x	803	570
-	-	x	-	0.897	-	3755	16	0.948	-	2087	111	0.948	-	2037	399
x	-	x	-	0.897	-	3755	16	0.948	-	2087	106	0.948	-	2037	400
-	x	x	-	0.897	-	3572	14	0.948	-	1990	110	0.948	-	1961	373
x	x	x	-	0.897	-	3572	16	0.948	-	1990	107	0.948	-	1961	374
-	-	-	x	0.897	-	3746	16	0.932	x	1417	79	0.932	x	1247	90
x	-	-	x	0.897	-	3746	16	0.932	x	1417	83	0.932	x	1247	89
-	x	-	x	0.897	-	3564	15	0.925	x	1214	96	0.926	x	1126	100
x	x	-	x	0.897	-	3564	16	0.925	x	1214	95	0.926	x	1126	101
-	-	x	x	0.897	-	3755	15	0.948	-	2087	80	0.949	-	2033	270
x	-	x	x	0.897	-	3755	16	0.948	-	2087	81	0.949	-	2033	271
-	x	x	x	0.897	-	3572	16	0.948	-	1991	85	0.948	-	1959	217
x	x	x	x	0.897	-	3572	16	0.948	-	1991	82	0.948	-	1959	217

5.2.1.3 Basic Stability Test: `ds1`

The first table to consider is Table 5.5. For this set of experiments on `ds1` there are several easy conclusions. All of the times and scores in the Loose Epsilon group are very close to one another, indicating little effect by the tested stability parameters. This is not true in the Moderate Epsilon or Tight Epsilon groups. By comparing pairs of rows one and two, three and four, etc., we see that the `modelmin` and `modelmax` parameters had little or no effect. Comparing these pairs of rows to one another shows that the `margin` parameter reduces the average minimum deviance, but by less than ten percent. Recall that the deviance is the LR loss function, and hence smaller is better. Furthermore `margin` made no significant change in LR's ability to correctly rank the test rows, judging by the small change in the AUC score. Recall from Section 5.1.4 that an AUC of one is the best possible score, and an AUC of zero is the worst possible score.

Comparing pairs of four rows shows the effect of the ridge-regression weight parameter `rrlambda`. This parameter makes a significant difference in the AUC score, the presence of NaN values in computations, the average minimum deviance and the speed. Though the deviance went up, the AUC improved. This suggests that the large coefficient penalty from the `rrlambda` parameter is preventing over-fitting of the training data. This will be discussed in greater detail after all of the IRLS stability charts are presented.

Finally we may compare the first and last halves of the table to see the effect of the `cgwindow` and `cgdecay` parameters. In the second half we see similar AUC scores, NaN occurrence, and deviance to the first half when `rrlambda` is used. However a clear improvement has been made when `rrlambda` isn't used. This suggests that experiments with `rrlambda` active never needed the `cgwindow` or `cgdecay` protection. With `cgwindow` and `cgdecay` active the non-`rrlambda` and `rrlambda` AUC scores are much closer than before, as are the times. The deviances still appear to dip too low without `rrlambda`, if our hypothesis of over-fitting is

Table 5.6: IRLS stability experiments for `imdb`. `binitmean` is disabled and `wmargin` is 0. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.965	-	9745	120	0.972	-	9236	146	0.908	x	1910	7576
x	-	-	-	0.965	-	9745	120	0.972	-	9236	145	0.904	x	1655	7840
-	x	-	-	0.965	-	8776	120	0.971	-	8366	148	0.911	x	1347	7570
x	x	-	-	0.965	-	8776	121	0.971	-	8366	133	0.907	x	1154	7844
-	-	x	-	0.965	-	9745	120	0.972	-	9236	149	0.983	-	3469	932
x	-	x	-	0.965	-	9745	121	0.972	-	9236	133	0.983	-	3469	930
-	x	x	-	0.965	-	8776	120	0.971	-	8367	135	0.983	-	2915	873
x	x	x	-	0.965	-	8776	121	0.971	-	8367	135	0.983	-	2915	874
-	-	-	x	0.965	-	9745	117	0.972	-	9236	134	0.971	x	2947	693
x	-	-	x	0.965	-	9745	120	0.972	-	9236	148	0.961	x	1762	922
-	x	-	x	0.965	-	8776	120	0.971	-	8366	146	0.973	x	2292	789
x	x	-	x	0.965	-	8776	121	0.971	-	8366	149	0.968	x	1362	985
-	-	x	x	0.965	-	9745	119	0.972	-	9236	148	0.983	-	3469	889
x	-	x	x	0.965	-	9745	120	0.972	-	9236	148	0.983	-	3469	892
-	x	x	x	0.965	-	8776	121	0.971	-	8367	147	0.983	-	2915	834
x	x	x	x	0.965	-	8776	119	0.971	-	8367	150	0.983	-	2915	841

correct.

Our conclusions from Table 5.5 are that `modelmin`, `modelmax` and `margin` aren't useful, while regularization through `rrlambda` and constant-improvement checks like `cgwindow` and `cgdecay` do appear useful. These conclusions apply only to experiments on `ds1` with the `wmargin` and `binitmean` parameters disabled. We will continue our analysis on the remaining three sparse datasets, though more briefly than for this first example, and summarize our findings at the end.

5.2.1.4 Basic Stability Test: `imdb`

The next dataset to consider is `imdb`, and the experiments are summarized in Table 5.6. The `modelmin` and `modelmax` parameters make some difference in the Tight Epsilon group, where enabling these parameters decreases the deviance and decreases the AUC . We can see that `margin` affects the deviance in every epsilon range, but produces only small variations in the AUC scores and speed. The `rrlambda` parameter only affects the tight epsilon experiments, apparently preventing over-fitting. In the tight epsilon range we also see `cgwindow` and `cgdecay` improving AUC and time significantly for experiments with `rrlambda` deactivated, and a decrease in time where `rrlambda` is activated. Only in the Tight Epsilon group are any NaN values present during computation, and activating `rrlambda` eliminates NaN values.

Our conclusions are that `modelmin` and `modelmax` should not be enabled for the `imdb` dataset, that `margin` is not effective, and that `rrlambda`, `cgwindow` and `cgdecay` are useful in preventing out-of-control calculations. It appears that the moderate epsilon experiments are better behaved with the `imdb` dataset than they were with `ds1`. Considering that the `cgeps` parameter is multiplied by the number of attributes, and that `imdb` has over twenty-five times as many attributes as `ds1`, we might surmise that this scaling of `cgeps` is too much.

Table 5.7: IRLS stability experiments for `citeseer`. `binitmean` is disabled and `wmargin` is 0. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.885	-	4113	35	0.928	-	3983	37	0.848	x	758	1508
x	-	-	-	0.885	-	4113	36	0.928	-	3983	40	0.848	x	758	1513
-	x	-	-	0.876	-	3466	36	0.923	-	3400	40	0.846	x	423	1526
x	x	-	-	0.876	-	3466	35	0.923	-	3400	40	0.846	x	423	1497
-	-	x	-	0.885	-	4113	36	0.928	-	3984	39	0.945	-	3478	103
x	-	x	-	0.885	-	4113	35	0.928	-	3984	39	0.945	-	3478	102
-	x	x	-	0.876	-	3466	36	0.923	-	3400	40	0.945	-	2910	101
x	x	x	-	0.876	-	3466	36	0.923	-	3400	41	0.945	-	2910	102
-	-	-	x	0.885	-	4113	35	0.928	-	3983	40	0.867	x	878	287
x	-	-	x	0.885	-	4113	38	0.928	-	3983	36	0.868	x	839	293
-	x	-	x	0.876	-	3466	36	0.923	-	3400	37	0.868	x	557	272
x	x	-	x	0.876	-	3466	36	0.923	-	3400	39	0.864	x	521	284
-	-	x	x	0.885	-	4113	36	0.928	-	3984	39	0.945	-	3478	102
x	-	x	x	0.885	-	4113	36	0.928	-	3984	38	0.945	-	3478	101
-	x	x	x	0.876	-	3466	35	0.923	-	3400	38	0.945	-	2910	101
x	x	x	x	0.876	-	3466	36	0.923	-	3400	40	0.945	-	2910	104

5.2.1.5 Basic Stability Test: `citeseer`

Table 5.7 covers stability experiments on the `citeseer` dataset. Again we see `modelmin` and `modelmax` having little influence, except for non-`rrlambda` experiments with tight epsilons where the influence is slightly more than nothing. The `margin` parameter is detrimental almost everywhere, while `rrlambda` is only helpful for tight epsilons. The `cgwindow` and `cgdecay` parameters are helpful for non-`rrlambda` experiments with tight epsilons. Our conclusions from the `citeseer` experiments are essentially the same as those for the `imdb` experiments in Table 5.6.

5.2.1.6 Basic Stability Test: `ds2`

The final sparse dataset is `ds2`, and stability experiments for this dataset may be found in Table 5.8. Again `modelmin` and `modelmax` make little difference, and the same can be said for `margin`. The large time difference between the first two pairs of rows is due to NaN values terminating IRLS iterations. The application of `rrlambda` does help AUC scores by apparently reducing overfitting in experiments with tight epsilons, as usual. There is one new twist with this dataset. When `rrlambda` is combined with `cgwindow` and `cgdecay` for experiments with tight epsilons we observe a dramatic speed improvement. It is possible that choosing a different `rrlambda` value than 10 may achieve the same speed improvement, and we will address this issue later.

Table 5.8: IRLS stability experiments for `ds2`. `binitmean` is disabled and `wmargin` is 0. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.478	-	8898	290	0.684	-	4378	731	0.669	x	41	21477
x	-	-	-	0.478	-	8898	293	0.684	-	4378	665	0.669	x	41	21486
-	x	-	-	0.477	-	8541	293	0.688	-	3963	641	0.649	-	45	56435
x	x	-	-	0.477	-	8541	294	0.688	-	3963	643	0.649	-	45	57015
-	-	x	-	0.478	-	8903	292	0.689	-	4377	681	0.720	-	963	10804
x	-	x	-	0.478	-	8903	296	0.689	-	4377	684	0.720	-	963	10799
-	x	x	-	0.477	-	8546	294	0.688	-	3966	658	0.722	-	879	10800
x	x	x	-	0.477	-	8546	295	0.688	-	3966	657	0.722	-	879	10805
-	-	-	x	0.478	-	8898	292	0.684	-	4378	738	0.681	x	254	3200
x	-	-	x	0.478	-	8898	292	0.684	-	4378	738	0.682	x	253	3250
-	x	-	x	0.477	-	8541	292	0.688	-	3963	694	0.683	x	218	3647
x	x	-	x	0.477	-	8541	293	0.688	-	3963	647	0.686	x	206	4502
-	-	x	x	0.478	-	8903	292	0.689	-	4377	742	0.720	-	961	6034
x	-	x	x	0.478	-	8903	295	0.689	-	4377	747	0.720	-	961	5707
-	x	x	x	0.477	-	8546	295	0.688	-	3966	716	0.722	-	877	6312
x	x	x	x	0.477	-	8546	296	0.688	-	3966	657	0.722	-	877	5852

5.2.1.7 Basic Stability Test: Sparse Conclusions

This concludes our basic stability tests with the `modelmin`, `modelmax`, `margin`, `rrlambda`, `cgwindow` and `cgdecay` parameters on sparse datasets. The most obvious conclusions are that `modelmin`, `modelmax` and `margin` have little beneficial effect, and sometimes are detrimental. It appears that `rrlambda` is very effective in preventing over-fitting, and in avoiding the wasted calculations and seconds that accompany over-fitting. The `cgwindow` and `cgdecay` parameters are especially helpful when `rrlambda` is not used, and may also be helpful when `rrlambda` is used. In no case were `rrlambda`, `cgwindow` or `cgdecay` particularly detrimental. We have raised the question as to whether multiplying `cgeps` by the number of attributes is appropriate. Perhaps more sensible would be multiplying `cgeps` by a different factor, as discussed later in Section 5.2.2.

5.2.1.8 Basic Stability Test: `ds1.100pca`

We now consider the same stability experiments as above on two dense datasets, `ds1.100pca` and `ds1.10pca`. As discussed in Section 5.1.3, these datasets are PCA projections of `ds1` down to 100 and 10 dimensions, respectively. Table 5.9 summarizes the results on `ds1.100pca`. As in the sparse experiments, all values in the Loose Epsilon group are within ten percent of one another. The `modelmin` and `modelmax` parameters are familiarly ineffective. The `margin` parameter seems to increase speed and decrease deviance in some experiments, but those same experiments show a decreased AUC . Regularization through `rrlambda` may have some effect, but that effect is even less than that of `margin`. The `cgwindow` and `cgdecay` parameters clearly reduce the time required by the computations, and have a small and probably insignificant effect on the scores. No overfitting is apparent anywhere in Table 5.9. We have run additional experiments with `cgeps`

Table 5.9: IRLS stability experiments for `ds1.100pca`. `binitmean` is disabled and `wmargin` is 0. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.906	-	3736	34	0.916	-	3356	66	0.919	-	3275	177
x	-	-	-	0.906	-	3736	36	0.916	-	3356	66	0.919	-	3275	177
-	x	-	-	0.906	-	3553	35	0.914	-	3246	60	0.919	-	3154	173
x	x	-	-	0.906	-	3553	35	0.914	-	3246	62	0.919	-	3154	173
-	-	x	-	0.906	-	3745	34	0.915	-	3402	62	0.919	-	3291	163
x	-	x	-	0.906	-	3745	34	0.915	-	3402	62	0.919	-	3291	164
-	x	x	-	0.906	-	3561	36	0.914	-	3259	61	0.918	-	3166	163
x	x	x	-	0.906	-	3561	32	0.914	-	3259	61	0.918	-	3166	164
-	-	-	x	0.905	-	3754	27	0.919	-	3295	61	0.919	-	3275	149
x	-	-	x	0.905	-	3754	27	0.919	-	3295	59	0.919	-	3275	150
-	x	-	x	0.905	-	3572	28	0.916	-	3213	53	0.919	-	3154	151
x	x	-	x	0.905	-	3572	28	0.916	-	3213	53	0.919	-	3154	152
-	-	x	x	0.905	-	3760	27	0.917	-	3342	55	0.919	-	3291	137
x	-	x	x	0.905	-	3760	28	0.917	-	3342	55	0.919	-	3291	137
-	x	x	x	0.903	-	3586	28	0.914	-	3244	47	0.918	-	3166	137
x	x	x	x	0.903	-	3586	27	0.914	-	3244	47	0.918	-	3166	138

as low as 1e-9 and `lreps` as low as 1e-7. While the time was approximately doubled, the minimum average deviance per fold and AUC scores were unchanged.

5.2.1.9 Basic Stability Test: `ds1.10pca`

Table 5.10 summarizes results for dataset `ds1.10pca`. New in this table is that `margin` in combination with `rrlambda` decrease the deviance by more than ten percent. However no improvement in AUC or speed is seen from this combination. There is a somewhat startling jump in time on some of the experiments with `cgwindow` and `cgdecay` enabled. Comparison of these experiments to their counterparts lacking `cgwindow` and `cgdecay` reveals that `cgwindow` and `cgdecay` may have terminated CG iterations too quickly. In one dramatic example the `cgwindow` and `cgdecay` experiment ran thirty IRLS iterations, wherein the first two IRLS iterations showed CG stage termination after six CG iterations. The counterpart experiment ran CG for twelve iterations each for the same two IRLS iterations. In this example the `cgwindow` and `cgdecay` experiment achieved lower overall deviance for the fold, but not significantly and there was no clear sign of overfitting in either case. The AUC score remained essentially the same, but the speed decreased significantly. Note that the decrease in speed is especially pronounced when `margin` is in effect.

These experiments on `ds1.10pca` suggest our `cgwindow` and `cgdecay` parameters may be too tight. None of the tested stability parameters has significant effect on AUC score, and none increased speed. If the experiments in which `margin` is enabled are ignored, the negative effects of `cgwindow` and `cgdecay` are less disturbing.

Table 5.10: IRLS stability experiments for `ds1.10pca`. `binitmean` is disabled and `wmargin` is 0. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.829	-	6255	4	0.842	-	5064	6	0.846	-	4985	11
x	-	-	-	0.829	-	6255	4	0.842	-	5064	6	0.846	-	4985	11
-	x	-	-	0.831	-	5679	4	0.841	-	4892	7	0.846	-	4828	10
x	x	-	-	0.831	-	5679	4	0.841	-	4892	6	0.846	-	4828	12
-	-	x	-	0.829	-	6256	4	0.842	-	5066	7	0.846	-	4985	12
x	-	x	-	0.829	-	6256	4	0.842	-	5066	7	0.846	-	4985	11
-	x	x	-	0.831	-	5679	4	0.841	-	4894	7	0.846	-	4828	12
x	x	x	-	0.831	-	5679	4	0.841	-	4894	7	0.846	-	4828	11
-	-	-	x	0.826	-	6333	3	0.841	-	5079	6	0.846	-	4985	12
x	-	-	x	0.826	-	6333	4	0.841	-	5079	7	0.846	-	4985	12
-	x	-	x	0.828	-	5570	4	0.841	-	4905	7	0.846	-	4837	27
x	x	-	x	0.828	-	5570	4	0.841	-	4905	6	0.846	-	4837	29
-	-	x	x	0.826	-	6333	2	0.841	-	5081	6	0.846	-	4985	18
x	-	x	x	0.826	-	6333	3	0.841	-	5081	6	0.846	-	4985	16
-	x	x	x	0.828	-	5571	4	0.841	-	4907	6	0.845	-	4837	31
x	x	x	x	0.828	-	5571	4	0.841	-	4907	6	0.845	-	4837	33

5.2.1.10 Basic Stability Test: Dense Conclusions

Our main conclusion for dense datasets is that stability is less of an issue than for our sparse datasets. One reasonable hypothesis is that stability problems are correlated with data having a large number of nonzero attributes in some rows. Having many nonzero values in a single record may lead to some rows exceeding the numerical bounds described at the beginning of Section 5.2.1.1. Another hypothesis, overlapping the first, is that datasets with a widely varying number of nonzero attributes per row are more susceptible to stability problems. Such a dataset could easily cause the entries in the $\mathbf{X}^T\mathbf{W}\mathbf{X}$ matrix to be badly scaled. In any case, we believe our data suggests that `modelmin`, `modelmax`, `margin` and `rrlambda` are unnecessary for dense data, while `cgwindow` and `cgdecay` may be harmful or at least require careful attention.

5.2.1.11 Stability Test: `wmargin`

Turning our attention to the `wmargin` parameter, we examine Table 5.11. These experiments enable the `wmargin` parameter for moderate epsilon experiments on the `ds1` dataset. We chose to use the `ds1` dataset and the Moderate Epsilon group for these tests because this combination showed adequate sensitivity to parameter changes in Section 5.2.1.3. The `wmargin` parameter was added to our software before `rrlambda` was implemented in an effort to eliminate NaN values. When this table is compared with the Moderate Epsilon group of Table 5.5, we observe the elimination of NaN values in all non-`rrlambda` experiments for which either `modelmin` and `modelmax`, or `margin`, are enabled. This same subset of experiments have similar scores. We note that `wmargin` has an undesirable effect on speed, especially in the second row of

Table 5.11: IRLS stability experiments for `ds1` with `wmargin`=1e-13. `binitmean` is disabled.

				Moderate Epsilon			
`mm`	`mar`	`rrl`	`cgw`	AUC	NaN	DEV	Time
-	-	-	-	0.890	x	479	585
x	-	-	-	0.893	-	307	1153
-	x	-	-	0.894	-	592	659
x	x	-	-	0.893	-	595	647
-	-	x	-	0.948	-	2087	127
x	-	x	-	0.948	-	2087	128
-	x	x	-	0.948	-	1990	128
x	x	x	-	0.948	-	1990	129
-	-	-	x	0.934	x	1367	164
x	-	-	x	0.933	-	1376	183
-	x	-	x	0.927	-	1204	151
x	x	-	x	0.927	-	1204	150
-	-	x	x	0.948	-	2087	95
x	-	x	x	0.948	-	2087	94
-	x	x	x	0.948	-	1990	94
x	x	x	x	0.948	-	1990	95

the table. In previous experiments, the occurrence of NaN values caused immediate termination of IRLS iterations. With `wmargin` eliminating the NaN values, training continues longer, deviance is reduced, and times increase. Since the holdout performance, judged by AUC , decreased we can assume the `wmargin` helps IRLS overfit the training data. Our conclusion is that using `wmargin` is inferior to using combinations of `rrlambda`, `cgwindow` and `cgdecay`.

In Table 5.12 we repeat the `wmargin` experiments for the dense dataset `ds1.100pca`. Since `wmargin` was intended to remove some NaN values, and that `ds1.100pca` experiments had no NaN values, it is not surprising that this table is nearly identical to the Moderate Epsilon group of Table 5.9. As a result, these experiments leave our previous assessment of `wmargin` in tact.

5.2.1.12 Stability Test: `binitmean`

Our final stability parameter is `binitmean`. This parameter was inspired by McIntosh [26] where it was used for direct LR CG-MLE. In Table 5.13 we test the effect of enabling `binitmean` on moderate epsilon experiments using dataset `ds1`. The Moderate Epsilon group of Table 5.5 can be used to identify the effects of `binitmean`. It is easy to see that `binitmean` does not eliminate the occurrence of NaN values, nor does it affect the scores significantly. It does appear to hurt the speed slightly.

Table 5.14 summarizes `binitmean` experiments on dataset `ds1.100pca` using moderate epsilons. The results can be compared to Table 5.9. These two tables are nearly identical. Our conclusion from Table 5.13 and Table 5.14 is that `binitmean` is not useful for our version of IRLS.

Table 5.12: IRLS stability experiments for `ds1.100pca` with `wmargin`=1e-13. `binitmean` is disabled.

				Moderate Epsilon			
`mm`	`mar`	`rrl`	`cgw`	AUC	NaN	DEV	Time
-	-	-	-	0.916	-	3356	65
x	-	-	-	0.916	-	3356	65
-	x	-	-	0.914	-	3246	60
x	x	-	-	0.914	-	3246	59
-	-	x	-	0.915	-	3402	59
x	-	x	-	0.915	-	3402	60
-	x	x	-	0.914	-	3259	59
x	x	x	-	0.914	-	3259	59
-	-	-	x	0.919	-	3295	60
x	-	-	x	0.919	-	3295	60
-	x	-	x	0.916	-	3213	53
x	x	-	x	0.916	-	3213	52
-	-	x	x	0.917	-	3342	54
x	-	x	x	0.917	-	3342	54
-	x	x	x	0.914	-	3244	47
x	x	x	x	0.914	-	3244	47

Table 5.13: IRLS stability experiments for `ds1` with `binitmean` enabled. `wmargin` is 0.

				Moderate Epsilon			
`mm`	`mar`	`rrl`	`cgw`	AUC	NaN	DEV	Time
-	-	-	-	0.896	x	822	523
x	-	-	-	0.896	x	822	536
-	x	-	-	0.895	x	758	546
x	x	-	-	0.895	x	758	567
-	-	x	-	0.948	-	2087	118
x	-	x	-	0.948	-	2087	118
-	x	x	-	0.948	-	1990	118
x	x	x	-	0.948	-	1990	117
-	-	-	x	0.930	x	1327	83
x	-	-	x	0.930	x	1327	84
-	x	-	x	0.928	x	1323	90
x	x	-	x	0.928	x	1323	90
-	-	x	x	0.948	-	2087	86
x	-	x	x	0.948	-	2087	86
-	x	x	x	0.948	-	1990	83
x	x	x	x	0.948	-	1990	83

Table 5.14: IRLS stability experiments for `ds1.100pca` with `binitmean` enabled. `wmargin` is 0.

				Moderate Epsilon			
`mm`	`mar`	`rrl`	`cgw`	AUC	NaN	DEV	Time
-	-	-	-	0.916	-	3356	68
x	-	-	-	0.916	-	3356	69
-	x	-	-	0.914	-	3246	62
x	x	-	-	0.914	-	3246	61
-	-	x	-	0.915	-	3402	63
x	-	x	-	0.915	-	3402	63
-	x	x	-	0.914	-	3259	60
x	x	x	-	0.914	-	3259	62
-	-	-	x	0.919	-	3295	62
x	-	-	x	0.919	-	3295	62
-	x	-	x	0.916	-	3213	55
x	x	-	x	0.916	-	3213	54
-	-	x	x	0.916	-	3342	56
x	-	x	x	0.916	-	3342	56
-	x	x	x	0.914	-	3259	47
x	x	x	x	0.914	-	3259	45

Table 5.15: IRLS stability experiments for `ds1` comparing `cgwindow` and `cgdecay`. Note the extra column so that enabling `cgwindow` is independent of enabling `cgdecay`.

					Moderate Epsilon			
`mm`	`mar`	`rrl`	`cgw`	`cgd`	AUC	NaN	DEV	Time
-	-	-	x	-	0.932	x	1417	84
x	-	-	x	-	0.932	x	1417	84
-	x	-	x	-	0.925	x	1214	95
x	x	-	x	-	0.925	x	1214	94
-	-	x	x	-	0.948	-	2087	85
x	-	x	x	-	0.948	-	2087	83
-	x	x	x	-	0.948	-	1990	83
x	x	x	x	-	0.948	-	1990	84
-	-	-	-	x	0.896	x	821	536
x	-	-	-	x	0.896	x	821	532
-	x	-	-	x	0.895	x	759	563
x	x	-	-	x	0.895	x	759	560
-	-	x	-	x	0.948	-	2087	112
x	-	x	-	x	0.948	-	2087	113
-	x	x	-	x	0.948	-	1990	114
x	x	x	-	x	0.948	-	1990	113

Table 5.16: IRLS stability experiments for `ds1.100pca` comparing `cgwindow` and `cgdecay`. Note the extra column so that enabling `cgwindow` is independent of enabling `cgdecay`.

					Moderate Epsilon			
mm	mar	rrl	cgw	cgd	AUC	NaN	DEV	Time
-	-	-	x	-	0.919	-	3295	62
x	-	-	x	-	0.919	-	3295	62
-	x	-	x	-	0.916	-	3213	54
x	x	-	x	-	0.916	-	3213	53
-	-	x	x	-	0.917	-	3342	55
x	-	x	x	-	0.917	-	3342	55
-	x	x	x	-	0.914	-	3244	48
x	x	x	x	-	0.914	-	3244	49
-	-	-	-	x	0.916	-	3356	69
x	-	-	-	x	0.916	-	3356	68
-	x	-	-	x	0.914	-	3246	61
x	x	-	-	x	0.914	-	3246	61
-	-	x	-	x	0.915	-	3402	62
x	-	x	-	x	0.915	-	3402	63
-	x	x	-	x	0.914	-	3259	61
x	x	x	-	x	0.914	-	3259	61

5.2.1.13 Stability Test: `cgwindow` Versus `cgdecay`

In previous sections we have seen the utility of enabling `cgwindow` and `cgdecay`. In Tables 5.15 and 5.16 we show the effects of enabling *only* `cgwindow` and `cgdecay`. From these tables we hope to determine whether both parameters are necessary. These tables replace the `cgw` column with separate columns for `cgwindow` and `cgdecay`. These columns are labeled `cgw` and `cgd`, respectively. Moderate epsilons were used, `binitmean` was disabled, and `wmargin`=0.

Table 5.15 shows results for dataset `ds1` with moderate epsilons. Many of the times are comparable to those of the Moderate Epsilon group in Table 5.5, but keep in mind that the rows are not equivalent. Enabling `cgwindow` is clearly more effective at preventing overfitting than enabling `cgdecay`. When `rrlambda` is enabled the deviances are the same, but the `cgwindow` experiments require less time. Comparing the `cgwindow` and `cgdecay` experiments to the lower half of the Moderate Epsilon group in Table 5.5 reveals discrepancies in the deviances obtained. This implies occasional interaction between `cgwindow` and `cgdecay`, rather than one always terminating before the other. Our conclusion from Table 5.15 is that `cgwindow` is superior to `cgdecay` and is effective by itself.

The experiments in Table 5.16 were run on the dense dataset `ds1.100pca` with moderate epsilons. There is less variation in deviance and AUC , but the times still indicate that `cgwindow` is terminating CG iterations at a more appropriate time than `cgdecay`. The slightly negative interaction with `rrlambda`, observed previously in Table 5.9, is still present. In contrast with the `ds1` experiments we just analyzed, the DEV achieved when `cgwindow` is enabled exactly matches that of the combined `cgwindow` and `cgdecay` experiments in Table 5.9. This suggests that in those experiments there was little or no interaction between `cgwindow` and `cgdecay`. Again we conclude that `cgdecay` is unnecessary when `cgwindow` is enabled.

Different values for `cgdecay` may improve its performance. In the best scenario for `cgdecay`, the

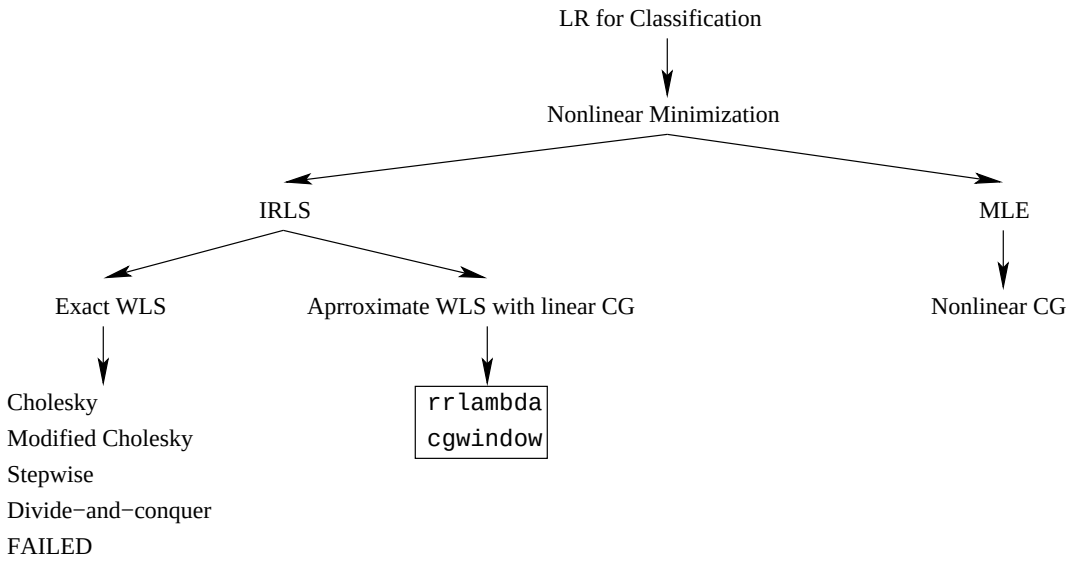


Figure 5.4: LR tree showing stability parameters selected for IRLS.

`cgwindow` could allow a few extra CG iterations. For our datasets, those iterations are not expensive relative to the total cost of the optimization. Extra CG iterations may contribute to overfitting if they reduce the training data deviance too far; in this case neither `cgwindow` or `cgdecay` will have any effect since these parameters catch deviance increases. Thus a finely tuned `cgdecay` is unlikely to do much better than `cgwindow`. For this reason we will not explore `cgdecay` further.

5.2.1.14 Stability Tests: Conclusions

Reviewing the conclusions of Sections 5.2.1.7, 5.2.1.10 and 5.2.1.12 we may identify which stability parameters are useful and safe. For sparse datasets we concluded that `rrlambda`, `cgwindow` and `cgdecay` were helpful. For dense datasets it wasn't clear that any stability parameters were needed, and we noticed that `cgwindow` and `cgdecay` may need some tuning to prevent premature CG termination. While `wmargin` is successful in reducing occurrences of NaN values for non-`rrlambda` experiments, it becomes irrelevant if `rrlambda` is used. Our `binitmean` experiments suggest that `binitmean` should not be used with our IRLS implementation. Finally we noted that `cgdecay` is not necessary when `cgwindow` is enabled.

For the remaining IRLS experiments in this thesis, we will activate only `rrlambda` and `cgwindow` from the stability parameters. The values of these parameters will be chosen empirically in Section 5.2.1.15. The parameters have been added to our LR description tree in Figure 5.4.

5.2.1.15 Stability Tests: Default Parameters

In this section we present two tables which motivate our selection of default stability parameter values. Our goal is to find a set of parameters that does well on all datasets, though our choices might not be optimal for any dataset. All later experiments will use the default values, hoping to encourage the idea that IRLS can be

Table 5.17: Moderate Epsilon

		Time						AUC					
		rrlambda						rrlambda					
Dataset	cgw	1	5	10	50	100	500	1	5	10	50	100	500
ds1	1	37	35	38	28	28	20	0.944	0.945	0.945	0.939	0.932	0.898
	2	74	50	45	39	33	24	0.943	0.947	0.946	0.941	0.932	0.909
	3	105	93	81	72	58	32	0.939	0.947	0.948	0.941	0.934	0.906
	5	160	126	106	77	57	39	0.941	0.948	0.948	0.941	0.934	0.906
imdb	1	135	134	135	133	133	133	0.972	0.972	0.972	0.972	0.972	0.972
	2	135	134	134	133	135	135	0.972	0.972	0.972	0.972	0.972	0.972
	3	134	132	134	133	135	134	0.972	0.972	0.972	0.972	0.972	0.972
	5	131	134	135	134	133	136	0.972	0.972	0.972	0.972	0.972	0.972
citeseer	1	36	37	34	34	35	35	0.928	0.928	0.928	0.928	0.928	0.928
	2	37	35	36	34	36	36	0.928	0.928	0.928	0.928	0.928	0.928
	3	34	34	36	35	37	34	0.928	0.928	0.928	0.928	0.928	0.928
	5	34	34	34	34	34	36	0.928	0.928	0.928	0.928	0.928	0.928
ds2	1	663	656	655	688	705	670	0.682	0.677	0.681	0.689	0.687	0.668
	2	668	679	680	708	707	697	0.682	0.689	0.689	0.687	0.690	0.667
	3	671	679	681	704	710	697	0.682	0.689	0.689	0.689	0.690	0.667
	5	667	680	681	701	710	698	0.682	0.689	0.689	0.689	0.690	0.667
ds1.100pca	1	31	31	31	27	25	21	0.916	0.916	0.916	0.912	0.907	0.891
	2	48	47	50	31	32	28	0.916	0.917	0.917	0.911	0.910	0.894
	3	57	58	52	29	29	38	0.918	0.918	0.917	0.911	0.908	0.893
	5	56	54	48	42	45	41	0.917	0.917	0.915	0.912	0.909	0.893
ds1.10pca	1	4	5	4	5	4	4	0.808	0.808	0.808	0.809	0.809	0.815
	2	5	4	5	5	6	5	0.831	0.830	0.829	0.839	0.839	0.842
	3	7	7	6	5	6	6	0.841	0.841	0.841	0.841	0.840	0.840
	5	7	7	6	7	7	7	0.842	0.842	0.842	0.842	0.841	0.840

used as an autonomous classifier with good results.

To make the times in this section's experiments comparable, all experiments were run as the only job on the computing platform described in Section 5.1.5. The experiments are broken into two sets, one with moderate epsilons and one with tight epsilons. Loose epsilons were not tested because of the lack of information the loose epsilon columns provided in previous sections. Each of the six datasets used in previous stability experiments are used again here. The IRLS stability parameters in question are `rrlambda` and `cgwindow`. For `rrlambda` the values 1, 5, 10, 50, 100 and 500 were tested, while for `cgwindow` the values 1, 2, 3 and 5. These test values are chosen based on previous experience, the nature of the parameters, and the desire to avoid excessive computation.

Table 5.17 shows results for moderate epsilons, while Table 5.18 shows results for tight epsilons. It has two halves, with times on the left and AUC scores on the right. Within each half, the columns represent the six `rrlambda` values. The table is broken into six sections horizontally, one for each dataset. Within each section are four rows representing the four `cgwindow` values. Labels for the `rrlambda` and `cgwindow` values

Table 5.18: Tight Epsilon

		Time						AUC					
		rrlambda						rrlambda					
Dataset	cgw	1	5	10	50	100	500	1	5	10	50	100	500
ds1	1	128	137	137	155	124	46	0.944	0.946	0.946	0.940	0.932	0.907
	2	169	196	210	113	96	90	0.943	0.948	0.948	0.940	0.932	0.909
	3	119	281	234	155	211	66	0.941	0.948	0.949	0.940	0.932	0.904
	5	226	304	283	183	150	83	0.941	0.948	0.948	0.941	0.932	0.904
imdb	1	1398	688	580	434	395	291	0.980	0.982	0.983	0.983	0.983	0.983
	2	955	844	728	482	417	292	0.980	0.982	0.983	0.983	0.983	0.983
	3	1290	981	791	481	413	292	0.980	0.982	0.983	0.983	0.983	0.983
	5	1499	982	791	482	415	289	0.980	0.982	0.983	0.983	0.983	0.983
citeseer	1	184	98	82	71	65	60	0.934	0.945	0.945	0.946	0.946	0.946
	2	211	126	96	74	65	58	0.935	0.945	0.945	0.946	0.946	0.946
	3	213	125	94	71	65	60	0.935	0.945	0.945	0.946	0.946	0.946
	5	211	126	95	72	64	59	0.935	0.945	0.945	0.946	0.946	0.946
ds2	1	2199	1791	2979	3738	4139	3959	0.706	0.705	0.713	0.724	0.727	0.712
	2	2130	6023	8089	4278	3127	3023	0.705	0.715	0.720	0.723	0.727	0.717
	3	4090	6312	5534	3942	3488	2580	0.705	0.714	0.720	0.726	0.726	0.723
	5	7158	8771	7300	5543	4269	2573	0.705	0.714	0.720	0.726	0.726	0.723
ds1.100pca	1	152	149	133	129	107	50	0.918	0.918	0.918	0.913	0.909	0.895
	2	104	118	104	108	277	211	0.919	0.919	0.919	0.913	0.910	0.893
	3	141	135	132	275	135	75	0.919	0.919	0.919	0.914	0.909	0.893
	5	147	139	132	111	99	82	0.919	0.919	0.919	0.914	0.909	0.893
ds1.10pca	1	34	33	34	32	27	26	0.845	0.845	0.845	0.842	0.826	0.839
	2	27	31	35	38	38	10	0.846	0.846	0.846	0.846	0.845	0.842
	3	11	11	17	40	11	10	0.845	0.846	0.846	0.846	0.845	0.842
	5	12	10	11	11	11	10	0.846	0.846	0.846	0.846	0.845	0.842

are listed above or to the left of their columns or rows, respectively.

We will employ a single-elimination strategy for finding good values of `rrlambda` and `cgwindow`. Any parameter value which causes uniformly poor AUC scores for a dataset will be removed from consideration. After all remaining AUC scores are adequate, the same procedure will be applied to the run times. With some luck, at least one combination of `rrlambda` and `cgwindow` will survive.

Examination of AUC scores in Tables 5.17 and 5.18 immediately eliminates `rrlambda` values 100 and 500. A closer look at the Tight Epsilon table eliminates `rrlambda`=1 because of `citeseer` and `ds2`. Similarly `ds1` eliminates `rrlambda`=50. Several rows with `cgwindow`=1 show poor scores in the remaining `rrlambda` columns, and in one case these experiments are slower as well.

When times are considered we see that setting `cgwindow`=5 is no better than `cgwindow`=3, and sometimes much worse. Our candidates are now `cgwindow`=2 or `cgwindow`=3, and `rrlambda`=5 or `rrlambda`=10. Within these restrictions `cgwindow`=3 and `rrlambda`=10 produce consistently good AUC scores and times,

while other combinations occasionally excel and occasionally plummet.

It is clear that we would benefit from tuning our parameters for every experiment, but we feel this is not reasonable. For all remaining IRLS experiments in this thesis we set `rrlambda` to 10 and `cgwindow` to 3. While this static assignment may handicap IRLS in experiments with alternative algorithms in Chapter 6, we believe a small penalty is less important than the benefit of eliminating experiment-by-experiment tuning.

5.2.2 Indirect (IRLS) Termination And Optimality

Many of the parameters in Section 5.2.1 affect termination and optimality, but were aimed at enhancing stability and providing a robust foundation for the remaining parameters. The parameters discussed in this section are primarily concerned with achieving optimal scores without wasted computation. These include the parameters `lreps`, `cgeps` and `cgdeveps`. There are an additional two parameters, `lrmax` and `cgmax`, used to set upper limits on the number of CG and IRLS iterations. These parameters are summarized in Table 5.3.

IRLS is terminated when the relative difference of the deviance falls below `lreps` or the number of IRLS iterations reaches `lrmax`. See Section 4.4 for details. Deviance-based termination is common, with the relative difference suggested by McIntosh [26]. Because relative deviances are scale-independent, it is not difficult to find a value for `lreps` which works well for most datasets.

Several options for CG termination were discussed in 2.1.2. Our focus is understanding whether and when LR is suitable for classification, especially in data mining applications, and we have examined very few termination criteria. The `cgeps` parameter is used for the traditional residual termination criteria. For linear CG inside IRLS, the residual vector is

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x} = \mathbf{X}^T\mathbf{W}\mathbf{z} - \mathbf{X}^T\mathbf{W}\mathbf{X}\beta \tag{5.1}$$

and, `cgwindow` aside, we terminate when the Euclidean norm $(\mathbf{r}^T\mathbf{r})^2$ of this vector is smaller than `cgeps`. It is difficult to find a problem-independent value for `cgeps` which provides optimal AUC scores and speed. Early in our research we choose to scale `cgeps` by the number of attributes in the dataset, such that the residual would be compared to $M \times$ `cgeps`. In this thesis we have questioned whether that scaling was appropriate, and we examine this question below.

The top half of Figure 5.5 shows how the AUC score varies as a function of `lreps` and `cgeps` for datasets `ds2` and `ds1.10pca`. Each line on these plots represents a different `lreps` value, and the horizontal axis represents `cgeps`. For `ds2` there is little difference between `lreps` lines so long as `cgeps` is not made too large. For `ds1.10pca` there is again little difference between `lreps` values, and no change for `cgeps` values. The `ds1.10pca` plot demonstrates that an absurdly low value of `lreps`, such as 0.5, can be somewhat harmful. This `lreps` sensitivity is dwarfed by the effect of too large a `cgeps` for `ds2`, and we will be very careful choosing a default `cgeps` value. The bottom half of Figure 5.5 illustrates how speed changes with `lreps` and `cgeps`. As one would expect, the shortest times occurring for the largest `lreps` and `cgeps` values. These results are representative of experiments on all six datasets used in our previous experiments.

The largest, and hence fastest, "safe" value of `lreps` appears to be 0.1. To be somewhat conservative, when using `cgeps` to terminate CG we will choose `lreps`=0.05. Choosing a value for `cgeps` is more difficult. Figure 5.6 shows how AUC and speed vary with `cgeps` in all six datasets used in previous experiments with `lreps` set to 0.05. Note that the vertical "Time" axis is logarithmic. It appears that any small value for `cgeps` will produce the same AUC score, but we incur a time penalty if `cgeps` is too small. This time penalty is severe for `ds1`, `imdb` and `ds2`. The largest sensible `cgeps` values for `ds1`, with respect to AUC, are quite different than those for `imdb` and `ds2`. Any conservative choice of `cgeps` will cause unduly slow results for `ds1`, which leads to a reconsideration of `cgeps` scaling.

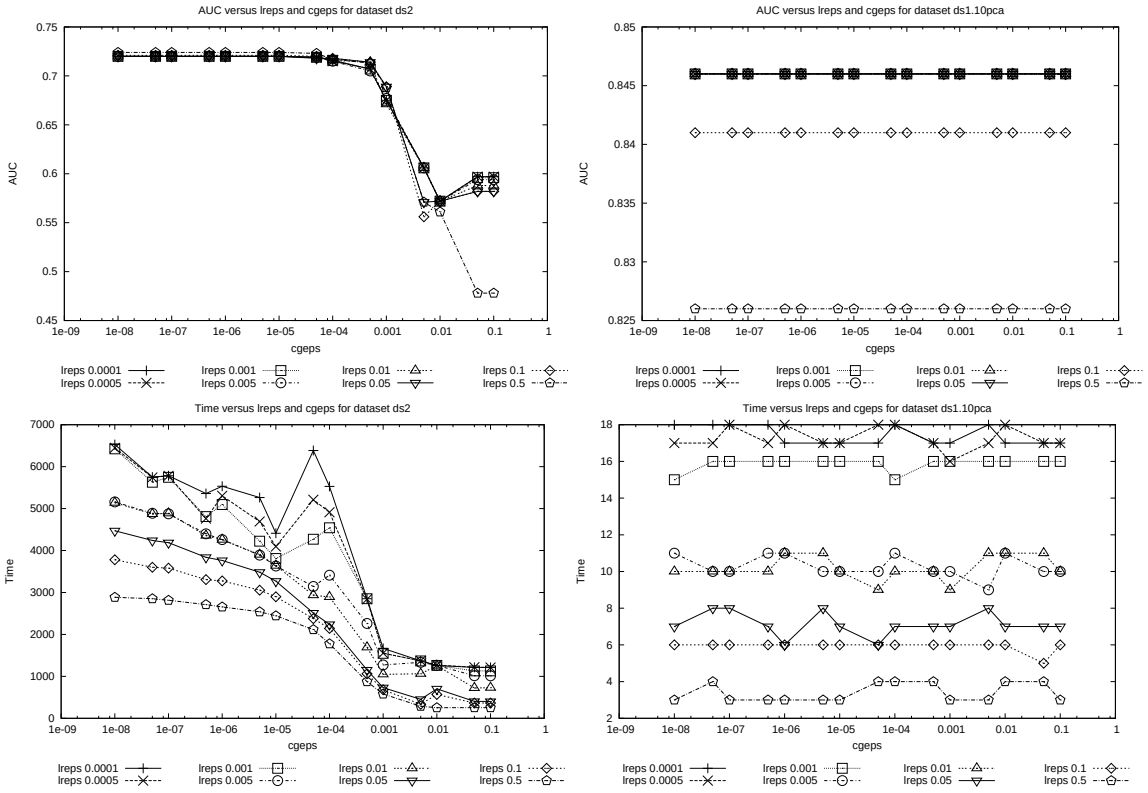


Figure 5.5: AUC and time over several `lreps` and `cgeps` values on datasets `ds2` and `ds1.10pca`.

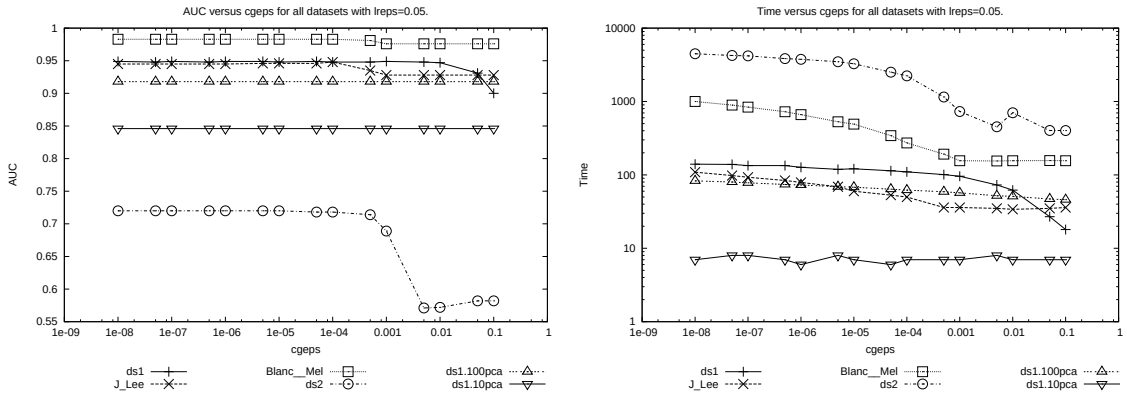


Figure 5.6: AUC and time versus `cgeps` on several datasets with `lreps`=0.05. Note that the time axis is logarithmic.

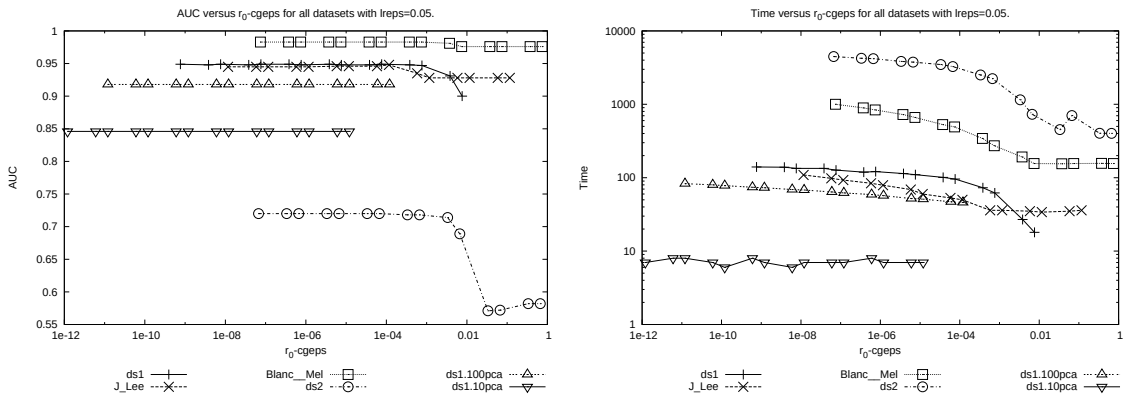


Figure 5.7: AUC and time versus $\mathbf{r}_0$-`cgeps` on several datasets with `lreps`=0.05. Note that the time axis is logarithmic.

The scaling used in every previous experiment multiplied the reported `cgeps` by M, the number of attributes in the dataset. This choice was made using the naive argument that the CG residual bound should depend on its dimensionality. Perhaps the most obvious change is to multiply by the square root of the dimensionality, since a Euclidean norm is used. Another option is multiplying `cgeps` by the value of the initial CG residual before any IRLS iterations, so that problems which start with a large residual terminate CG iterations sooner. A similar termination criterion is suggested in Shewchuk [41] and discussed briefly in Section 2.1.2. When the `binitmean` parameter is disabled, the LR parameter vector β is initialized to zero and the initial CG residual $\mathbf{r}_0$ of Equation 5.1 simplifies to $\mathbf{X}^T\mathbf{W}\mathbf{z} = \mathbf{X}^T(\mathbf{y} - [0.5,\ldots,0.5]^T)$.

Figure 5.6 suggests that the most flattering value of the M-scaled `cgeps` is 0.01 for `ds1`, 0.0005 for `imdb`, and somewhere near 0.0001 for `ds2`. The table below shows how these values scale under the number of attributes M, the square root of the number of attributes $\sqrt{M}$, and the Euclidean norm of the initial residual vector $\mathbf{r}_0$.

Dataset	M	M-`cgeps`	$\sqrt{M}$	$\sqrt{M}$-`cgeps`	$\mathbf{r}_0$	$\mathbf{r}_0$-`cgeps`
`ds1`	6,348	0.01	79.67	0.79778	83,227	0.00076
`imdb`	685,569	0.0005	827.99	0.41399	92,503	0.00370
`ds2`	1,143,054	0.0001	1069.14	0.10691	171,694	0.00066

Both $\sqrt{M}$-scaling and $\mathbf{r}_0$-scaling reduce the ratio of the high and low `cgeps` values from 100 to 7.5 and 5.6, respectively. Since the $\mathbf{r_0}$-scaling incorporates both the dimensionality-awareness of the $\sqrt{M}$ with dataset-specific information, we favor this scaling for residual-based CG termination. Figure 5.7 transforms the M-scaled `cgeps` values of Figure 5.6 into $\mathbf{r}_0$-scaled values. Table 5.19 shows AUC scores and times for several $\mathbf{r}_0$-`cgeps` values with `lreps`=0.05.

We conclude that a $\mathbf{r}_0$-`cgeps` of 0.005 is adequate for all datasets, while 0.001 is safe for all datasets. Choosing $\mathbf{r}_0$-`cgeps`=0.001 increases the time significantly. Nonetheless we prefer $\mathbf{r}_0$-`cgeps` $= 0.001$ for autonomous applications and will use this setting for later experiments in this thesis.

The size of the CG residual $\mathbf{r}$ is influenced by many factors including the number of positive dataset rows and the sparsity of the data, and hence there are many reasonable scalings for `cgeps`. For completeness we have performed experiments for other scale factors including the number of nonzero elements MRF, the

Table 5.19: AUC and Times For Several $\mathbf{r}_0$-cgeps Values

$\mathbf{r}_0$-cgeps	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
0.05	0.925	23	0.965	139	0.928	36	0.673	843	0.889	22	0.788	4
0.01	0.947	42	0.970	193	0.947	41	0.717	1590	0.912	26	0.846	6
0.005	0.948	47	0.983	279	0.947	37	0.720	2372	0.917	31	0.846	5
0.001	**0.949**	**86**	**0.983**	**370**	**0.946**	**44**	**0.720**	**3417**	**0.918**	**44**	**0.846**	**7**
0.0005	0.949	93	0.983	429	0.946	45	0.720	3548	0.918	46	0.846	7
0.0001	0.949	106	0.983	589	0.945	58	0.720	3838	0.918	50	0.846	7
0.00005	0.949	112	0.983	638	0.945	61	0.720	3936	0.918	52	0.846	7
0.00001	0.949	117	0.983	752	0.945	70	0.720	4211	0.918	56	0.846	7

Table 5.20: AUC and Times For Several cgdeveps Values

cgdeveps	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
0.01	0.945	65	0.983	440	0.946	67	0.721	2520	0.914	38	0.821	10
0.005	**0.946**	**78**	**0.983**	**514**	**0.946**	**74**	**0.718**	**2923**	**0.916**	**43**	**0.846**	**12**
0.001	0.948	111	0.983	622	0.945	84	0.722	3570	0.916	63	0.846	13

average number of nonzero elements per dataset row, the dataset sparsity F, the "dense" size MR, and the number of positive dataset rows. While $\mathbf{r}_0$-scaling is both logically and empirically attractive, the need to scale cgeps motivates exploration of CG termination criteria which do not depend on the CG residual.

One alternative to residual-based termination methods for CG is to use a context-sensitive measure such as the deviance. The cgdeveps parameter tests the relative difference of the deviance between CG iterations, just as lreps tests this quantity for IRLS iterations. While the CG residual is a natural part of CG computations, the LR deviance is not. The deviance requires $\mathrm{O}(MRF)$ time to compute. However, as seen with lreps the value of cgdeveps is adaptive, and may be more suitable for an autonomous classifier. We do not allow cgeps and cgdeveps to be used simultaneously, though nothing precludes this possibility.

The top half of Figure 5.8 illustrates AUC versus cgdeveps for ds2 and citeseer. The imdb dataset behaves something like citeseer on a smaller scale, while the others mimic ds2. The improvement in AUC for citeseer with looser cgdeveps comes from avoiding the overfitting we witnessed in earlier experiments with this dataset. Examining the deviances we see that the overfit experiments have reduced deviance an extra 1.5%, unlike the 500+% reductions seen in Table 5.7, and we are not concerned with the small AUC penalty in these experiments. It is possible that different stability parameter defaults would be appropriate for cgdeveps, but for this work we will keep our current stability parameters and avoid over-tight lreps and cgdeveps values.

Figure 5.8 suggests that an lreps of 0.1 is adequate and 0.05 is safe so long as cgdeveps is sufficiently small. As with cgeps, larger lreps and cgdeveps cause earlier termination and hence better speed. The bottom half of Figure 5.8 shows the relation between lreps, cgdeveps and time for datasets ds2 and citeseer. For these datasets and the others, the cost of lreps=0.05 is always very similar to the cost of lreps=0.1. Preferring to error on the side of safety, we will use lreps=0.05 for all cgdeveps experiments.

The relation between AUC, time and cgdeveps when lreps=0.05 is shown for all six datasets in Fig-

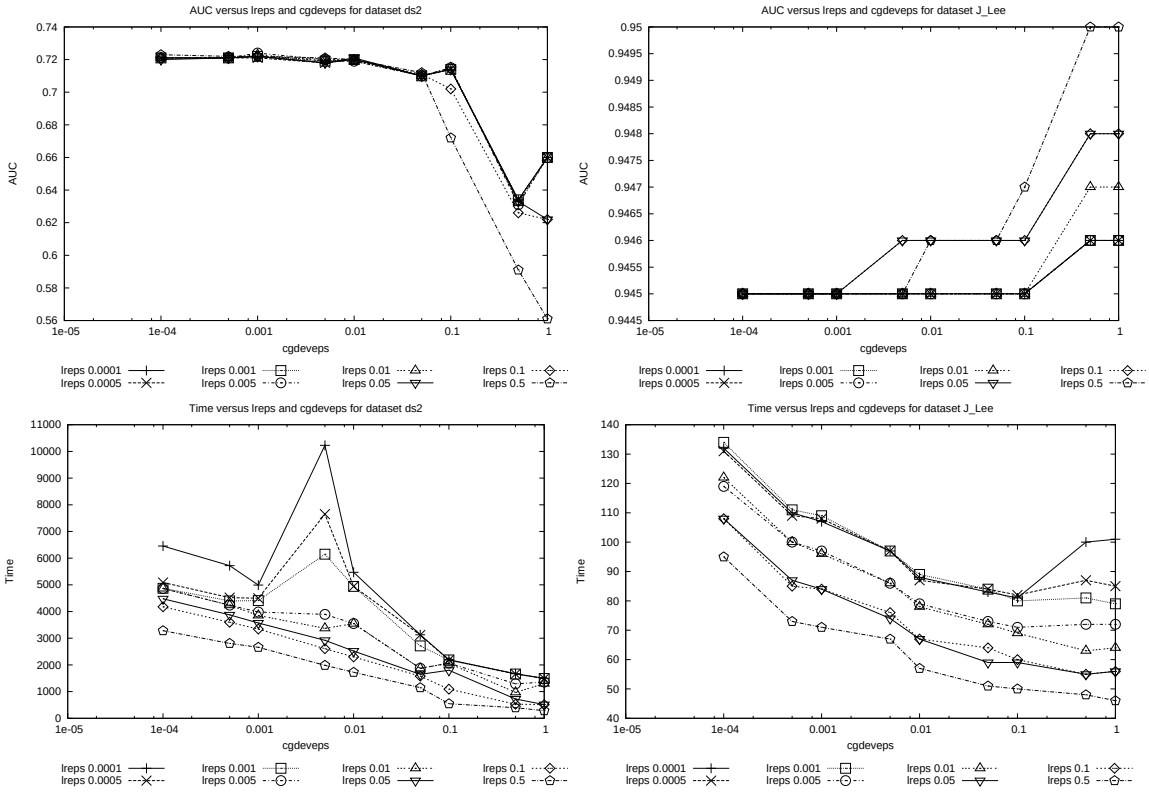


Figure 5.8: AUC and time over several `lreps` and `cgdeveps` values on datasets `ds2` and `ds1.10pca`.

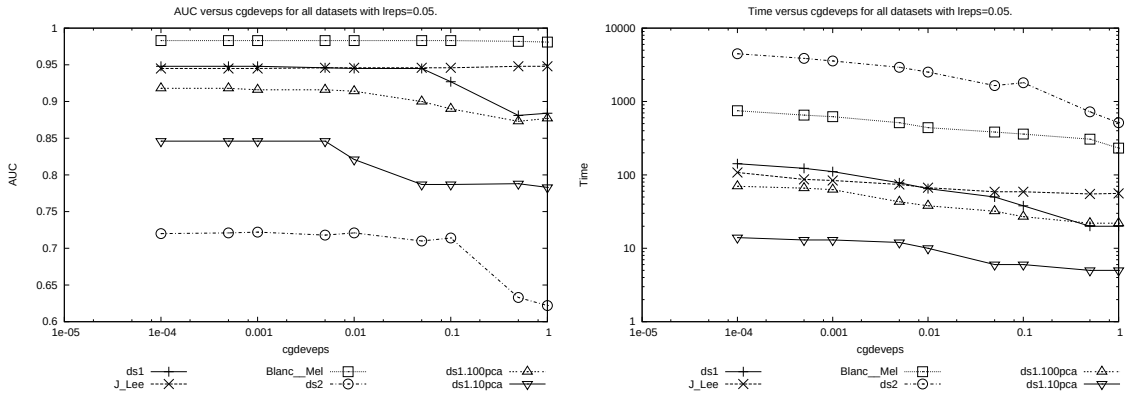


Figure 5.9: AUC and time versus `cgdeveps` on several datasets with `lreps`=0.05. Note that the time axis is logarithmic.

Table 5.21: Maximum LR and CG iteration counts across all folds of `cgeps` and `cgdeveps` experiments.

	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
	LR	CG	LR	CG	LR	CG	LR	CG	LR	CG	LR	CG
cgeps	6	34	7	7	6	3	8	55	5	20	5	9
cgdeveps	7	22	7	8	6	4	9	31	5	10	5	9

ure 5.9. Note that the time axis is logarithmic. The most interesting `cgdeveps` values are 0.01, 0.005 and 0.001. Table 5.20 shows AUC and times for these values. If not for dataset `ds1.10pca`, setting `cgdeveps`=0.01 would be adequate, perhaps even safe. While `cgdeveps`=0.001 appears safe for everyone, it comes with a 10% to 30% speed penalty over `cgdeveps`=0.005. Though 0.005 produces only adequate results for `ds1.10pca`, the neighboring decrease in AUC is only 3%. Given these trade-offs and since this only seems to affect one of our datasets, we accept the risk of data mining on-the-edge and choose `cgdeveps`=0.005. We ran each experiment from the `cgdeveps`=0.005 row of Table 5.20, just as we did in the $\mathbf{r}_0$-`cgeps`=0.001 row of Table 5.19. There was little deviance and the original experiments were representative of the mean.

The paragraphs above describe two very different strategies for terminating CG iterations. Because of the sizeable difference, and because of the challenges and trade-offs made when selecting default values for `cgeps` and `cgdeveps`, we will show results for both `cgeps` and `cgdeveps` throughout most of our later experiments.

Our experiments agree with the assertion in McCullagh and Nelder [25] that few IRLS iterations are needed to achieve optimality. Table 5.21 shows the maximum number of LR iterations for folds of the experiments reported above using default parameter values. For these experiments the maximums and minimums were always similar. In both the `cgeps` and `cgdeveps` experiments, no more than thirteen LR iterations were ever used. Therefore it is reasonable to set `lrmax` at thirty to prevent runaway experiments.

Table 5.21 also shows CG iteration maximums, taken within and then over the folds. From these counts we see that the number of CG iterations never exceeded fifty-five for our experiments with the default parameters. Because of the variability between CG iteration counts, we feel that setting `cgmax` to two-hundred is a reasonable restraint. While the counts in Table 5.21 are the maximum counts over the folds of each experiment, the minimum counts are not much different. We have added `lreps`, both CG termination parameters, `lrmax` and `cgmax` to our LR description tree, shown in Figure 5.10.

5.2.3 Indirect (IRLS) Speed

Most of the speed-enhancing techniques we have tried also belong to the stability or termination sections above. One technique not yet discussed aims at reducing the number of CG iterations by selecting better CG starting points. In all previous experiments the β parameter vector is set to zero for each IRLS iteration, excluding the constant term if `binitmean` is used. If we enable the speed parameter `cgbinit`, then β will only be set to zero for the first IRLS iteration. Subsequent iterations will start CG from the previous stopping point. This parameter may be found at the bottom of Table 5.3.

Table 5.22 shows the AUC and times achieved before and after `cgbinit` was added to the default parameters of the previous section. The `cgb` column indicates whether or not `cgbinit` was enabled. We observe little change in `cgeps` experiments, and it is not always to our benefit to enable `cgbinit`. For experiments using `cgdeveps` it is clear that `cgbinit` causes quicker termination with nearly identical scores.

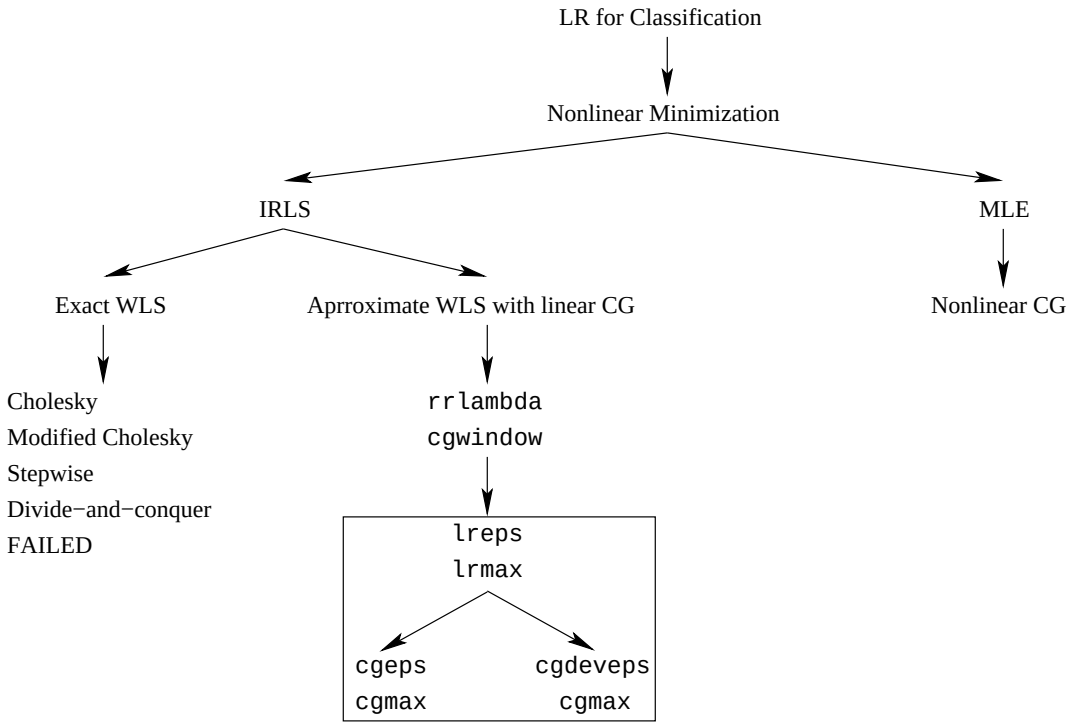


Figure 5.10: LR tree showing all termination parameters used for IRLS, including the safeguard parameters `lrmax` and `cgmax`.

Table 5.22: IRLS AUC and speed with and without `cgbinit` enabled

eps-type	cgb	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
cgeps	-	0.949	86	0.983	370	0.946	44	0.720	3417	0.918	44	0.846	7
cgeps	x	0.949	87	0.983	407	0.945	50	0.720	3264	0.918	48	0.845	8
cgdeveps	-	0.946	78	0.983	514	0.946	74	0.718	2923	0.916	43	0.846	12
cgdeveps	x	0.948	63	0.983	402	0.945	70	0.722	1978	0.913	36	0.842	9

Table 5.23 shows the mean IRLS and CG iteration over the ten folds of the experiments of Table 5.22. When `cgeps` is used, the number of IRLS and CG iterations are similar whether or not `cgbinit` is enabled. Comparison of the point-by-point progress of CG for the `ds1` experiments having `cgbinit` disabled or enabled revealed that IRLS iterations were finishing in nearly the same place. This is despite the difference in starting points for the second and later IRLS iterations when `cgbinit` is enabled. It appears to take as long from either starting point to reach the termination region for the objective function, both in time and the number of iterations. The final deviances are similar, as are the AUC scores. In summary, the `cgeps` `cgbinit` experiments with run at most five percent faster, run slower by up to thirteen percent, and arrive at nearly the same place. We see no reason to enable `cgbinit` for `cgeps` experiments.

For `cgdeveps` experiments, Table 5.23 suggests something very different is happening when `cgbinit` is enabled. The number of CG iterations per fold is different. Since our CG iteration counts represent the number of directions searched, we can conclude that different paths are taken toward the global optimum. Because the relative difference of the deviances is used to terminate CG iterations, there is no reason to believe that these paths will end in the same place; we only know that the relative improvement per CG iteration had decreased below a threshold. Point-by-point comparisons of CG and IRLS progress for the `cgdeveps` `ds1` experiments confirm that the distance between the terminating parameter estimates is as much as one-third the mean of the two parameter vectors' norms. Regardless of this discrepancy, the final parameter estimates with `cgbinit` produce similar AUC scores to those of previous experiments, and they arrive at those scores sooner. Therefore we will use `cgbinit` with future `cgdeveps` experiments.

Figure 5.11 shows an updated version of our LR description tree. Note that `cgbinit` only appears on the `cgdeveps` branch.

We are still faced with the question of why the number of CG iterations does not change when `cgbinit` is used with `cgeps`, but does change when `cgbinit` is used with `cgdeveps`. One explanation is that without `cgbinit`, terminating on `cgeps` and `cgdeveps` with our default parameters forces too many CG iterations for some datasets. When `cgeps` is used, linear CG will not stop until the gradient of the quadratic objective function is sufficiently small. If this termination region is particularly tiny, then the starting point might not materially affect how many iterations are needed to reach it, regardless of the path taken. This would prevent `cgbinit` from reducing the number of CG iterations needed to reach termination.

On the other hand, with `cgdeveps` the position of termination can change. Adding `cgbinit` to `cgdeveps` experiments changes the path and the termination point, allowing a change in the number of CG iterations. In particular, consider what happens once IRLS iterations are within a neighborhood of the optimum in which the Hessian changes relatively little. The optimum of the weighted least squares subproblem will be near the LR MLE. Continuation of CG iterations where the previous set left off may be similar to allowing CG to continue longer in previous iteration, since the new Hessian is similar to the old Hessian. Combining this observation with the properties of linear CG discussed in Section 2.1.2, it is unsurprising to find that the total number of CG iterations needed to reach the optimum are less when `cgbinit` is used.

5.2.4 Indirect (IRLS) Summary

Throughout this section we have evaluated many parameters for stability, termination and speed. Most of the variations on IRLS were rejected. We have retained `rrlambda` for regularization, `cgwindow` to assure continued improvement, and `cgbinit` was kept for `cgdeveps` experiments due to a speed improvement. Due to the interesting and different behaviors of the `cgeps` and `cgdeveps` termination methods, we will treat each as a different classifier for the remainder of this thesis. In the following chapter we will compare IRLS performance to that of other classification algorithms and see if our fixed parameters are adequate.

Table 5.23: Mean LR and CG iteration counts across all folds of `cgeps` and `cgdeveps` experiments, with and without `cgbinit` enabled.

	cgb	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
		LR	CG	LR	CG	LR	CG	LR	CG	LR	CG	LR	CG
cgeps	-	6.0	29.3	7.0	5.3	6.0	2.2	8.0	32.9	5.0	15.5	5.0	7.6
cgeps	x	6.0	28.9	7.0	5.3	6.0	2.0	8.0	30.4	5.0	16.3	5.0	8.6
cgdeveps	-	5.8	15.5	6.8	5.9	6.0	3.5	7.9	19.3	5.0	8.8	5.0	8.2
cgdeveps	x	6.0	10.2	6.8	3.6	6.0	2.5	7.0	13.6	5.0	6.2	5.0	5.2

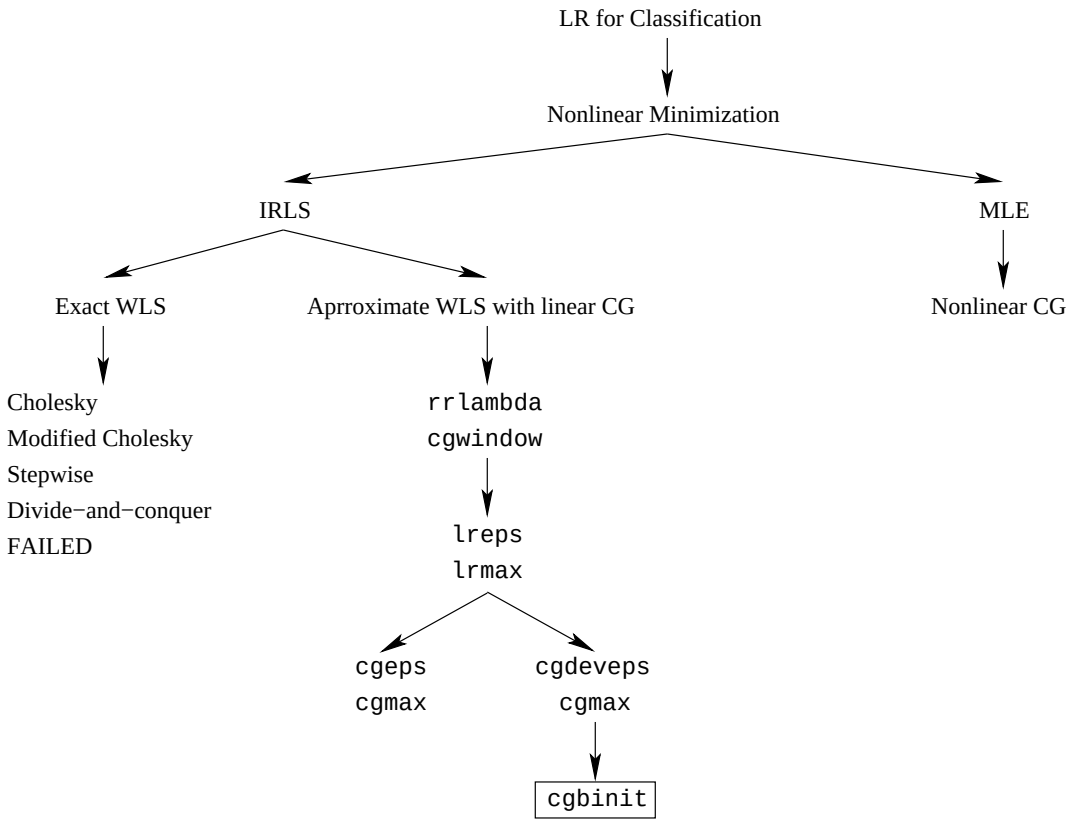


Figure 5.11: LR tree showing the `cgbinit` speed parameter used for IRLS with `cgdeveps`.

Algorithm 5 shows our final version of IRLS. The `lrmax`, `rrlambda`, and `lreps` parameters are embedded in the pseudocode to emphasize that they are fixed at their default values. There are two important changes from Algorithm 4. The first is at line 5.1, where we call CG to find an approximate solution to the linear weighted least squares subproblem. We have have written this as a function call to **CUSTOMCG**, since the details vary according to whether we are using `cgeps` or `cgdeveps`. The second change is at line 5.2. Instead of returning the value of $\hat{\beta}$ at the final iteration, we choose the value of $\hat{\beta}$ which had the minimum deviance among IRLS iterations.

When using `cgeps`, we modify CG as shown in Algorithm 6. To prevent confusion, we have replaced the scalar β used in the original CG of Algorithm 1 with γ. The primary difference is the addition of `cgwindow` termination. This may be seen at line 6.1 and the lines beginning at 6.2. Note that this version of CG ignores the $\hat{\beta}_{\text{old}}$ parameter sent by IRLS.

Algorithm 7 shows our modification to CG when using `cgdeveps`. Again we use γ in place of the original scalar β of Algorithm 1. Another difference is the addition of the `cgwindow` technique, seen at or around lines 7.2 and 7.3. The final difference is at line 7.4, where we choose the best $\mathbf{x}$ according to deviance. When compared to the CG algorithm used with `cgeps`, described in the previous paragraph, there are two important differences. The first is the nonzero initialization of $\mathbf{x_0}$ at line 7.1, which makes use of the $\hat{\beta}_{\text{old}}$ parameter passed by our IRLS algorithm. This is the result of the `cgbinit` parameter we have chosen to use with `cgdeveps`. A side-effect of the nonzero $\mathbf{x_0}$ is a change in the computation of $\mathbf{r_0}$, shown in the following line. The second important difference with the `cgeps` version of CG is the selection of the best $\mathbf{x}$ at line 7.4.

The lack of CG position history in Algorithm 6 is regrettable, since it complicates comparison of `cgeps` and `cgdeveps`. However, this difference is unlikely to have had any material effect. Had CG iterations caused the final iterate to be much worse than the best iterate, then `cgdecay` would have had greater positive effect in the experiments of Section 5.2.1.13. Instead, those experiments suggested that the deviance grew slowly after reaching a minimum. Since optimizing the likelihood does not necessarily correspond to maximizing the AUC score [46], it is not clear that choosing a slightly non-optimal iterate should have any negative effect at all.

We have neglected many interesting possibilities for IRLS with CG. It is possible to use CG with a *preconditioning matrix*, which may contribute to stability and speed if some or all of the covariance matrix can be estimated [41; 31; 7]. We made one experiment with a simple diagonal preconditioner [41]. The result was unsatisfactory but the failure was not investigated. Alternate techniques for IRLS or CG termination could be tried, including the use of *validation sets* [10]. Again we made an unsuccessful foray, and did not pursue a remedy. Though CG is designed for positive definite matrices, it has performed well on a variety of ill-conditioned data matrices arising from datasets such as `ds1` with linearly dependent or duplicate columns. A version of CG known as *biconjugate gradient* is designed for positive semidefinite matrices, and this might further accelerate IRLS. Besides biconjugate gradient, Greenbaum [7] lists several more iterative methods for solving linear systems. Truncated-Newton methods should be investigated for finding the zeros of the LR score equations, and the result compared to our combination of IRLS and CG.

5.3 CG-MLE Parameter Evaluation and Elimination

5.3.1 Direct (CG-MLE) Stability

The LR maximum likelihood equations are not quadratic forms as defined in Section 2.1. Therefore nonlinear CG is required instead of the linear CG used in our IRLS, above. We have chosen to use the Polak-Ribiére direction update due to its consistently good performance in experiments with our implementation of CG.

input : $\mathbf{X}$, $\mathbf{y}$

output : $\hat{\beta}$ which maximizes LR likelihood

/*Initialization. */

$\hat{\beta}_0 = \mathbf{0}$

$\mathrm{DEV}_0 := \mathrm{DEV}(\hat{\beta}_0)$

/*Iterations. */

for i := 0 ... lrmax-1 **do**

  /*Compute weights and adjusted dependent covariates. */

  $w_{ij} := \mu(\mathbf{x_j}, \hat{\beta}_i)(1 - \mu(\mathbf{x_j}, \hat{\beta}_i),\ j = 1..R$

  $\mathbf{W_i} := \mathrm{diag}(w_{i1}, \ldots, w_{iR})$

  $\mathbf{z_i} := \mathbf{X}\hat{\beta}_i + \mathbf{W}^{-1}(\mathbf{y} - \mu(\mathbf{X}, \hat{\beta}_i))$

  /*Approximate solution to linear weighted least squares subproblem. */

  $\hat{\beta}_{\mathrm{old}} := \hat{\beta}_i$

  $\mathbf{A} := (\mathbf{X}^T\mathbf{W_i}\mathbf{X} + \texttt{rrlambda} \cdot \mathbf{I})$

  $\mathbf{b} := \mathbf{X}^T\mathbf{W_i}\mathbf{z_i}$

5.1 $\hat{\beta}_{i+1} :=$ **CUSTOMCG**$(\hat{\beta}_{\mathrm{old}}, \mathbf{A}, \mathbf{b})$

  /*Termination. */

  $\mathrm{DEV}_{i+1} := \mathrm{DEV}(\hat{\beta}_{i+1})$

  if $|(\mathrm{DEV}_i - \mathrm{DEV}_{i+1})/\mathrm{DEV}_{i+1}| < \texttt{lreps}$ **then break**

/*Find best $\hat{\beta}$. */

5.2 $i^* := \mathrm{argmin}_{j=0\ldots i}\ \mathrm{DEV}_j$

$\hat{\beta} = \hat{\beta}_{i^*}$

Algorithm 5: Our IRLS implementation.

Table 5.24: CG-MLE Parameters

Parameter	Description
`modelmin`	Lower threshold for $\mu = \exp(\eta)/(1+\exp(\eta))$
`modelmax`	Upper threshold for $\mu = \exp(\eta)/(1+\exp(\eta))$
`margin`	Symmetric threshold for outcomes y
`binitmean`	Initialize β_0 to E($\mathbf{y}$)
`rrlambda`	Ridge-regression parameter λ
`cgwindow`	Number of non-improving iterations allowed
`cgdecay`	Factor by which deviance may decay during iterations
`cgeps`	Deviance epsilon for CG iterations
`cgmax`	Maximum number of CG iterations
`dufunc`	Nonlinear CG direction update selection

```
input  : A, b
output : x such that —Ax - b— is minimized within tolerance.

/*Initialization. */
x_0 := 0
r_0 := b
rsqrmin := ∞
window := cgwindow

/*Iterations. */
for i := 0 ...cgmax-1 do
    /*Termination. */
    if rsqrmin < cgeps then break
6.1  if window ≤ 0 then break

    /*Regular Linear CG. */
    if i=0 then γ_i := 0
    else γ_i := r_i^T r_i / (r_{i-1}^T r_{i-1})
    d_i := r_i + γ_i d_{i-1}
    α_i := -d_i^T r_0 / (d_i^T A d_i)
    x_{i+1} := x_i - α_i d_i
    r_{i+1} := b - A x_i

    /*cgwindow */
6.2  if (||r_{k+1}|| ≤ rsqrmin) then
        window := cgwindow
        rsqrmin := ||r_{k+1}||
    else window := window - 1
x := x_i
```

Algorithm 6: Modified CG for our IRLS implementation when using `cgeps`.

As discussed in Section 2.2, nonlinear CG needs occasional restarts of the search direction to the current gradient. For all of our CG-MLE experiments we use two restart criteria. The simplest is restarting after M iterations are performed, where M is the number of attributes in our dataset. This is only likely to occur for our narrowest dataset, `ds1.10pca`. The second is Powell restarts, described in Section 2.2. Powell restarts are incorporated into the Polak-Ribiére direction update formula, a combination we refer to as modified Polak-Ribiére direction updates. The proposed stability parameters for our CG-MLE implementation are explained in sections that follow, and summarized in Table 5.24. We will continue updating our LR description tree, and Figure 5.12 indicates the branch related to CG-MLE.

We recommend that the reader not compare results in this section to those of 5.2. Chapter 6 compares times and AUC scores for the final version of our IRLS implementation, using both `cgeps` and `cgdeveps`, the final version of our CG-MLE implementation, and three other popular classifiers.

input : $\hat{\beta}_{\text{old}}$, **A**, **b**
output : **x** such that —**Ax - b**— is minimized within tolerance.

/*`Initialization.` */
7.1 $\mathbf{x_0} := \hat{\beta}_{\text{old}}$
$\mathbf{r_0} := \mathbf{b} - \mathbf{A}\,\mathbf{x_0}$
$\text{DEV}_{-1} := \infty$
$\text{DEV}_0 := \text{DEV}(\mathbf{x_0})$
devmin := ∞
window := `cgwindow`

/*`Iterations.` */
for i := 0 ...`cgmax`-1 **do**
  /*`Termination.` */
  if $|(\text{DEV}_{i-1} - \text{DEV}_i)/\text{DEV}i| <$ `cgdeveps` **then break**
7.2 **if** window ≤ 0 **then break**

  /*`Regular Linear CG.` */
  if i=0 **then** $\gamma_i := 0$
  else $\gamma_i := \mathbf{r_i}^T\mathbf{r_i}/(\mathbf{r_{i-1}}^T\mathbf{r_{i-1}})$
  $\mathbf{d_i} := \mathbf{r_i} + \gamma_i\mathbf{d_{i-1}}$
  $\alpha_i := -\mathbf{d_i}^T\mathbf{r_0}/(\mathbf{d_i}^T\mathbf{A}\mathbf{d_i})$
  $\mathbf{x_{i+1}} := \mathbf{x_i} - \alpha_i\mathbf{d_i}$
  $\mathbf{r_{i+1}} := \mathbf{b} - \mathbf{A}\mathbf{x_i}$

  /*`cgwindow` */
7.3 $\text{DEV}_{i+1} := \text{DEV}(\mathbf{x_{i+1}})$
  if $\text{DEV}_{i+1} \leq$ devmin **then**
    window := `cgwindow`
    devmin := DEV_{i+1}
  else window := window - 1

/*`Find best x.` */
7.4 $i^* := \text{argmin}_{j=0\ldots i}\ \text{DEV}_j$
$\mathbf{x} := \mathbf{x_{i^*}}$

Algorithm 7: Modified CG for our IRLS implementation when using `cgdeveps`.

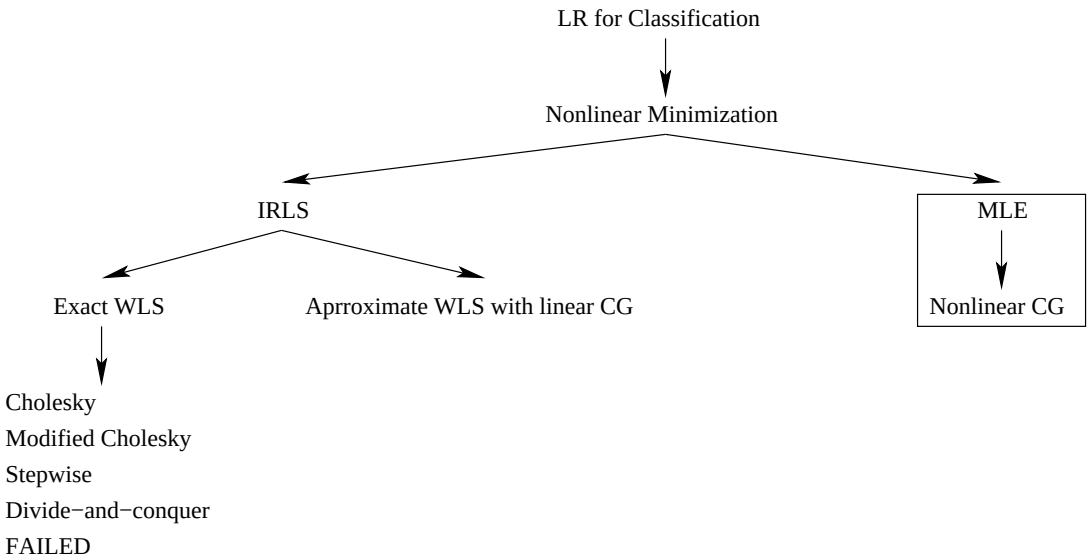


Figure 5.12: LR tree with rectangle marking the beginning of our CG-MLE implementation.

5.3.1.1 Stability parameter motivation

The MLE approach to LR uses the same logistic model as IRLS, and the current predictions from the model are used for log-likelihood computations. It may be assumed throughout this discussion that likelihood refers to the log-likelihood. The model output is μ in Equation 4.2. Therefore we anticipated similar underflow and overflow problems as we experienced in our early IRLS implementations. We also anticipated problems with binary outputs and parameters growing unreasonably large, based on our early experience with IRLS. These problems are described in more detail in Section 5.2.1.1. For these reasons we implemented the same `modelmin`, `modelmax` and `margin` stability parameters as we did for IRLS.

Also mentioned in Section 5.2.1.1 was a suggestion by McIntosh [26] to initialize the β_0 parameter to the mean of the outputs when using CG to find the MLE. We can enable this option using the `binitmean` parameter. A penalty similar to ridge regression may be added to the LR likelihood, as described in Section 4.4. The ridge regression parameter λ is controlled by the `rrlambda` parameter in our software. If `rrlambda` is useful in the experiments that follow, we will search for an optimal value of `rrlambda` and hence need not worry about scale.

Our final parameters for this stability section are `cgwindow` and `cgdecay`. These are described in Section 5.2.1.1 for the CG subproblem in IRLS. Here these parameters are also used for CG, though CG is now the entire solver. Although our previous experience with the CG-MLE technique has not suggested a clear need for these parameters, they have been added for symmetry with our IRLS experiments.

5.3.1.2 Stability Parameter Tables

The tables in the following sections describe our sparse stability experiments on the `modelmin`, `modelmax`, `margin`, `binitmean`, `rrlambda`, `cgwindow` and `cgdecay` parameters. These tables are arranged identically to

those in Section 5.2.1. Section 5.2.1.2 covers the symbols and construction of the tables, and Section 5.2.1.3 steps through our analysis technique.

The values used for the `modelmin`, `modelmax`, `margin`, `rrlambda`, `cgwindow` and `cgdecay` parameters are listed in Table 5.4. The IRLS tables of Section 5.2.1 showed results for three pairs of epsilon values, where the epsilons applied to the termination of both the IRLS and CG iterations. In CG-MLE experiments there is only one epsilon used to test for convergence. This parameter is named `cgeps` because it is used to terminate CG iterations. In the CG-MLE experiment tables, the Loose Epsilon group of columns has `cgeps`=0.1. The Moderate Epsilon group has `cgeps`=0.001, and the Tight Epsilon group has `cgeps`=0.000001. Unless otherwise stated, the `binitmean` parameter is disabled.

The CG-MLE `cgeps` parameter is described in more detail in Section 5.3.2. It is worth noting that it is not scaled by the number of parameters like the IRLS `cgeps` parameter. It is more similar to the IRLS `lreps` parameter because it is compared to the relative change of deviances. That the range of `cgeps` values used for the stability tests is different should not be surprising, since the algorithms are different. We believe the `cgeps` values used in the experiments below are sufficient for exploring stability.

As described in Section 5.2.1.1, the `modelmin` and `modelmax` parameters change the shape of LR expectation function, and hence the likelihood surface. Because these two parameters clip the model values, the likelihood looses its convexity where `modelmin` and `modelmax` take effect. This affects the line search used in CG, since it becomes possible to overshoot the convex region. Our line search implementation is aware of this and takes appropriate precautions to avoid failure cases.

5.3.1.3 Basic Stability Test: `ds1`

Table 5.25 summarizes our experiments for CG-MLE on `ds1`. The `modelmin` and `modelmax` parameters accomplish little, occasionally hurting the speed. The exception is that application of `modelmin`, `modelmax` or `margin` removes NaN values during computation. Even though `margin` both lowers and raises deviance, it has little effect on AUC scores and speed. Enabling `rrlambda` creates a dramatic improvement in score and speed for a moderate epsilon. For a tight epsilon the combination of `rrlambda`, `cgwindow` and `cgdecay` is required for a similar improvement. Enabling `rrlambda` raises the deviance but improves the AUC score, suggesting that CG-MLE is subject to the same overfitting problems as IRLS. We expect that CG-MLE and IRLS will share many such traits, since they are both maximizing the LR likelihood.

5.3.1.4 Basic Stability Test: `imdb`

In Table 5.26 we see `modelmin` and `modelmax` making a significant difference for experiments on dataset `imdb`. When `rrlambda` is disabled and a moderate or tight epsilon is used, these parameters reduce deviance by nearly one-half. In some cases times decrease, while in others the times triple. The AUC is decreased in every case. When `rrlambda` is enabled, `modelmin` and `modelmax` make no difference. The same volatility is observed when `margin` is enabled and `rrlambda` is disabled. While `margin` appears less detrimental than `modelmin` and `modelmax`, it still appears dangerous.

These experiments on `imdb` score higher and run faster when `rrlambda` is enabled for a moderate or tight epsilon. Enabling `cgwindow` and `cgdecay` decreases times for a moderate epsilon when `rrlambda` is disabled. As we have seen before, a tight epsilon benefits from the combination of enabling `rrlambda`, `cgwindow` and `cgdecay`. Unlike our experiments on `ds1` in Section 5.3.1.3, the top scores may be found with a loose epsilon.

Table 5.25: CG-MLE stability experiments for `ds1`. `binitmean` is disabled. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.883	-	3883	40	0.931	x	736	368	0.931	x	736	367
x	-	-	-	0.883	-	3883	39	0.930	-	670	430	0.930	-	670	402
-	x	-	-	0.883	-	3736	40	0.926	-	1038	415	0.926	-	1038	383
x	x	-	-	0.883	-	3736	40	0.926	-	1038	409	0.926	-	1038	384
-	-	x	-	0.883	-	3876	39	0.946	-	2313	164	0.946	-	2309	324
x	-	x	-	0.883	-	3876	40	0.946	-	2313	166	0.946	-	2309	324
-	x	x	-	0.883	-	3728	40	0.946	-	2228	165	0.947	-	2214	337
x	x	x	-	0.883	-	3728	40	0.946	-	2228	164	0.947	-	2214	338
-	-	-	x	0.883	-	3883	39	0.931	x	736	368	0.931	x	736	368
x	-	-	x	0.883	-	3883	40	0.930	-	670	429	0.930	-	670	401
-	x	-	x	0.883	-	3736	39	0.926	-	1038	409	0.926	-	1038	383
x	x	-	x	0.883	-	3736	39	0.926	-	1038	408	0.926	-	1038	384
-	-	x	x	0.883	-	3876	39	0.946	-	2313	174	0.946	-	2309	177
x	-	x	x	0.883	-	3876	40	0.946	-	2313	165	0.946	-	2309	176
-	x	x	x	0.883	-	3728	39	0.946	-	2228	163	0.947	-	2214	198
x	x	x	x	0.883	-	3728	39	0.946	-	2228	164	0.947	-	2214	197

Table 5.26: CG-MLE stability experiments for `imdb`. `binitmean` is disabled. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.982	-	3753	518	0.966	x	2166	1033	0.966	x	2166	1030
x	-	-	-	0.982	-	3753	520	0.946	-	1223	3724	0.945	-	1184	4339
-	x	-	-	0.982	-	3241	505	0.961	-	1778	1247	0.961	-	1778	1215
x	x	-	-	0.982	-	3241	505	0.946	-	1175	3941	0.947	-	1156	4301
-	-	x	-	0.983	-	4132	510	0.983	-	4090	666	0.983	-	4074	981
x	-	x	-	0.983	-	4132	507	0.983	-	4090	675	0.983	-	4074	977
-	x	x	-	0.983	-	3803	456	0.983	-	3453	749	0.983	-	3453	1174
x	x	x	-	0.983	-	3803	458	0.983	-	3453	771	0.983	-	3453	1178
-	-	-	x	0.982	-	3753	511	0.966	x	2166	1018	0.966	x	2166	1033
x	-	-	x	0.982	-	3753	509	0.946	-	1223	3844	0.945	-	1184	4329
-	x	-	x	0.982	-	3241	498	0.961	-	1778	1282	0.961	-	1778	1210
x	x	-	x	0.982	-	3241	497	0.946	-	1175	4005	0.947	-	1156	4300
-	-	x	x	0.983	-	4132	500	0.983	-	4090	688	0.983	-	4074	787
x	-	x	x	0.983	-	4132	502	0.983	-	4090	667	0.983	-	4074	789
-	x	x	x	0.983	-	3803	453	0.983	-	3453	746	0.983	-	3453	854
x	x	x	x	0.983	-	3803	461	0.983	-	3453	747	0.983	-	3453	856

Table 5.27: CG-MLE stability experiments for `citeseer`. `binitmean` is disabled. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.946	-	2544	73	0.891	x	578	580	0.887	x	574	630
x	-	-	-	0.946	-	2544	74	0.846	-	348	978	0.836	-	335	1076
-	x	-	-	0.929	-	1791	149	0.798	-	510	825	0.791	-	508	779
x	x	-	-	0.929	-	1791	151	0.804	-	457	1149	0.797	-	454	1086
-	-	x	-	0.947	-	3742	52	0.946	-	3725	114	0.946	-	3725	121
x	-	x	-	0.947	-	3742	51	0.946	-	3725	114	0.946	-	3725	120
-	x	x	-	0.947	-	3191	91	0.946	-	3148	168	0.946	-	3148	191
x	x	x	-	0.947	-	3191	90	0.946	-	3148	168	0.946	-	3148	192
-	-	-	x	0.946	-	2544	75	0.891	x	578	579	0.887	x	574	630
x	-	-	x	0.946	-	2544	75	0.846	-	348	978	0.836	-	335	1074
-	x	-	x	0.929	-	1791	150	0.798	-	510	821	0.791	-	508	778
x	x	-	x	0.929	-	1791	149	0.804	-	457	1152	0.797	-	454	1087
-	-	x	x	0.947	-	3742	52	0.946	-	3725	113	0.946	-	3725	118
x	-	x	x	0.947	-	3742	51	0.946	-	3725	114	0.946	-	3725	117
-	x	x	x	0.947	-	3191	90	0.946	-	3148	166	0.946	-	3148	191
x	x	x	x	0.947	-	3191	93	0.946	-	3148	168	0.946	-	3148	193

5.3.1.5 Basic Stability Test: `citeseer`

Our CG-MLE stability experiments on the `citeseer` dataset, summarized in Table 5.27, respond poorly to enabling `modelmin` and `modelmax`. The reaction to `margin` is even worse. Enabling `rrlambda` improves scores and times significantly, simultaneously raising the deviance six-fold or more. It appears that this dataset is acutely susceptible to overfitting. It is not clear that the `cgwindow` and `cgdecay` parameters are helpful. As we observed in Section 5.3.1.4, Table 5.26, top scores are achieved with a loose epsilon.

5.3.1.6 Basic Stability Test: `ds2`

Dataset `ds2` is the last of our sparse datasets. In Table 5.28 we see the same contrasting deviance between `rrlambda` and non-`rrlambda` experiments as we saw in Table 5.27. As before, we conclude that overfitting is a key factor for this dataset. As expected we see that enabling `rrlambda` raises the deviance, raises the AUC , and improves speed. Every other parameter has no effect, or causes both benefit and harm. Once more we see the best scores for loose values of epsilon.

5.3.1.7 Basic Stability Test: Sparse Conclusions

It seems clear from the sparse CG-MLE stability experiments above that `modelmin`, `modelmax` and `margin` should not be used. Not only are they ineffective, they are often detrimental. The `rrlambda` parameter made a large difference in most experiments with a moderate or tight epsilon. In some cases the addition of the `cgwindow` and `cgdecay` parameters dramatically reduced times compared to plain `rrlambda` experiments.

Table 5.28: CG-MLE stability experiments for ds2. binitmean is disabled. The first four columns represent the state of modelmin and modelmax, margin, rrlambda, and cgwindow and cgdecay.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.681	-	4255	1481	0.676	x	35	8513	0.676	x	35	8607
x	-	-	-	0.681	-	4255	1481	0.676	-	28	9170	0.676	-	28	8541
-	x	-	-	0.675	-	4077	1600	0.648	-	146	9847	0.648	-	146	9092
x	x	-	-	0.675	-	4077	1611	0.648	-	146	9831	0.648	-	146	9099
-	-	x	-	0.681	-	4263	1484	0.723	-	1395	3964	0.723	-	1395	7184
x	-	x	-	0.681	-	4263	1483	0.723	-	1395	3974	0.723	-	1395	7185
-	x	x	-	0.675	-	4075	1601	0.726	-	1252	4266	0.726	-	1251	7241
x	x	x	-	0.675	-	4075	1593	0.726	-	1252	4270	0.726	-	1251	7232
-	-	-	x	0.681	-	4255	1499	0.676	x	35	8509	0.676	x	35	8594
x	-	-	x	0.681	-	4255	1500	0.676	-	28	9159	0.676	-	28	8551
-	x	-	x	0.675	-	4077	1626	0.648	-	146	10097	0.648	-	146	9091
x	x	-	x	0.675	-	4077	1626	0.648	-	146	9983	0.648	-	146	9105
-	-	x	x	0.681	-	4263	1506	0.723	-	1395	4071	0.723	-	1395	3894
x	-	x	x	0.681	-	4263	1481	0.723	-	1395	3970	0.723	-	1395	3901
-	x	x	x	0.675	-	4075	1610	0.726	-	1252	4295	0.726	-	1251	4238
x	x	x	x	0.675	-	4075	1595	0.726	-	1252	4271	0.726	-	1251	4241

Though we observed the best scores in the Loose Epsilon group for three of our four datasets, the ds1 dataset had the best scores with a tight epsilon. This indicates we cannot ignore tight epsilon values, and the accompanying overfitting problems, when choosing our stability parameters and their values. Our conclusion is that the rrlambda, cgwindow and cgdecay parameters are useful for sparse CG-MLE computations.

5.3.1.8 Basic Stability Test: ds1.100pca

We have run stability experiments for CG-MLE on two dense datasets. The first of these is ds1.100pca, and experiments on this dataset are summarized in Table 5.29. There are no NaN values in any computations. The moderate and tight epsilon experiments show no signs of overfitting and little variance in score and deviance. Only in the tight epsilon columns do the times vary by more than ten percent. For these experiments we see rrlambda improving times by up to seventeen percent when cgdecay and cgwindow are not used. When cgdecay and cgwindow are used, enabling rrlambda improves times by up to forty-four percent. Using cgdecay and cgwindow without rrlambda yields little improvement in speed. Using a tight epsilon produced the best scores.

5.3.1.9 Basic Stability Test: ds1.10pca

Our second dense dataset is ds1.10pca. The experiments for this dataset, summarized in Table 5.30, have no NaN values. For any fixed epsilon, all scores and deviances are within ten percent of one another. As with the ds1.100pca experiments, overfitting does not appear to be an issue. The rrlambda parameter makes little difference, but cgwindow and cgdecay account for a speed increase of nearly twenty percent with a tight

Table 5.29: CG-MLE stability experiments for `ds1.100pca`. `binitmean` is disabled. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.879	-	4003	153	0.917	-	3317	464	0.919	-	3275	1108
x	-	-	-	0.879	-	4003	140	0.917	-	3317	461	0.919	-	3275	1112
-	x	-	-	0.877	-	3881	148	0.916	-	3201	467	0.919	-	3154	1189
x	x	-	-	0.877	-	3881	151	0.916	-	3201	469	0.919	-	3154	1196
-	-	x	-	0.879	-	3997	145	0.917	-	3328	499	0.918	-	3310	992
x	-	x	-	0.879	-	3997	152	0.917	-	3328	502	0.918	-	3310	990
-	x	x	-	0.876	-	3903	148	0.917	-	3208	480	0.918	-	3181	988
x	x	x	-	0.876	-	3903	148	0.917	-	3208	482	0.918	-	3181	987
-	-	-	x	0.879	-	4003	146	0.917	-	3317	453	0.919	-	3275	1050
x	-	-	x	0.879	-	4003	151	0.917	-	3317	455	0.919	-	3275	1125
-	x	-	x	0.877	-	3881	146	0.916	-	3201	462	0.919	-	3154	1201
x	x	-	x	0.877	-	3881	148	0.916	-	3201	461	0.919	-	3154	1208
-	-	x	x	0.879	-	3997	150	0.917	-	3328	486	0.918	-	3310	669
x	-	x	x	0.879	-	3997	152	0.917	-	3328	490	0.918	-	3310	666
-	x	x	x	0.876	-	3903	149	0.917	-	3208	473	0.918	-	3181	676
x	x	x	x	0.876	-	3903	149	0.917	-	3208	473	0.918	-	3181	680

Table 5.30: CG-MLE stability experiments for `ds1.10pca`. `binitmean` is disabled. The first four columns represent the state of `modelmin` and `modelmax`, `margin`, `rrlambda`, and `cgwindow` and `cgdecay`.

				Loose Epsilon				Moderate Epsilon				Tight Epsilon			
mm	mar	rrl	cgw	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time	AUC	NaN	DEV	Time
-	-	-	-	0.783	-	5914	13	0.839	-	5108	38	0.847	-	4985	90
x	-	-	-	0.783	-	5914	13	0.839	-	5108	38	0.847	-	4985	89
-	x	-	-	0.782	-	5743	14	0.843	-	4891	48	0.846	-	4828	87
x	x	-	-	0.782	-	5743	13	0.843	-	4891	47	0.846	-	4828	87
-	-	x	-	0.783	-	5918	14	0.840	-	5104	40	0.847	-	4987	86
x	-	x	-	0.783	-	5918	13	0.840	-	5104	41	0.847	-	4987	87
-	x	x	-	0.782	-	5746	13	0.841	-	4897	48	0.846	-	4829	92
x	x	x	-	0.782	-	5746	13	0.841	-	4897	48	0.846	-	4829	91
-	-	-	x	0.783	-	5914	13	0.839	-	5108	42	0.847	-	4985	82
x	-	-	x	0.783	-	5914	12	0.839	-	5108	41	0.847	-	4985	82
-	x	-	x	0.782	-	5743	13	0.843	-	4891	53	0.846	-	4828	80
x	x	-	x	0.782	-	5743	13	0.843	-	4891	53	0.846	-	4828	80
-	-	x	x	0.783	-	5918	14	0.840	-	5104	44	0.847	-	4987	75
x	-	x	x	0.783	-	5918	13	0.840	-	5104	45	0.847	-	4987	75
-	x	x	x	0.782	-	5746	13	0.841	-	4897	51	0.846	-	4829	74
x	x	x	x	0.782	-	5746	13	0.841	-	4897	52	0.846	-	4829	75

Table 5.31: CG-MLE stability experiments for `ds1` with `binitmean` enabled.

				Moderate Epsilon			
`mm`	`mar`	`rrl`	`cgw`	AUC	NaN	DEV	Time
-	-	-	-	0.933	x	821	349
x	-	-	-	0.931	-	698	382
-	x	-	-	0.927	-	1044	374
x	x	-	-	0.927	-	1044	381
-	-	x	-	0.946	-	2313	153
x	-	x	-	0.946	-	2313	153
-	x	x	-	0.946	-	2226	156
x	x	x	-	0.946	-	2226	156
-	-	-	x	0.933	x	821	349
x	-	-	x	0.931	-	698	388
-	x	-	x	0.927	-	1044	373
x	x	-	x	0.927	-	1044	374
-	-	x	x	0.946	-	2313	152
x	-	x	x	0.946	-	2313	152
-	x	x	x	0.946	-	2226	152
x	x	x	x	0.946	-	2226	153

epsilon. For a moderate epsilon, these parameters decrease speed by approximately ten percent. The best AUC scores come from a tight epsilon.

5.3.1.10 Basic Stability Test: Dense Conclusions

The dense datasets `ds1.100pca` and `ds1.10pca` showed fewer stability problems than the sparse datasets, and no overfitting was apparent. This is similar to the difference between our conclusions for sparse and dense IRLS experiments in Sections 5.2.1.7 and 5.2.1.10. For both datasets a tight epsilon produced the best scores. Together with the absence of overfitting, this suggests that even tighter epsilons may improve scores. This will be explored in Section 5.3.2.

The `modelmax`, `modelmin` and `margin` parameters should not be used. The `rrlambda` parameter was most useful in conjunction with the `cgdecay` and `cgwindow` parameters. Although `cgdecay` and `cgwindow` caused a small slowdown for moderating epsilon experiments on `ds1.10pca`, they created a medium to large speedup for tight epsilon experiments on `ds1.100pca` and `ds1.10pca`. Our conclusion is that we should use `rrlambda`, `cgwindow` and `cgdecay` together.

5.3.1.11 Stability Test: `binitmean`

Our final stability parameter is `binitmean`, which was suggested by McIntosh [26] for CG-MLE. We will identify the effects of this parameter using Table 5.31, corresponding to dataset `ds1`, and Table 5.32, corresponding to dataset `ds1.100pca`. In both tables a moderate epsilon was used.

Dataset `ds1` showed some stability and overfitting problems in experiments without `binitmean` enabled. See Section 5.2.1.3 for details about those experiments. In Table 5.31 we see that the `binitmean` parameter

Table 5.32: CG-MLE stability experiments for `ds1.100pca` with `binitmean` enabled.

				Moderate Epsilon			
`mm`	`mar`	`rrl`	`cgw`	AUC	NaN	DEV	Time
-	-	-	-	0.917	-	3326	441
x	-	-	-	0.917	-	3326	444
-	x	-	-	0.916	-	3198	475
x	x	-	-	0.916	-	3198	478
-	-	x	-	0.917	-	3328	497
x	-	x	-	0.917	-	3328	463
-	x	x	-	0.917	-	3209	463
x	x	x	-	0.917	-	3209	441
-	-	-	x	0.917	-	3326	442
x	-	-	x	0.917	-	3326	443
-	x	-	x	0.916	-	3198	474
x	x	-	x	0.916	-	3198	471
-	-	x	x	0.917	-	3328	499
x	-	x	x	0.917	-	3328	491
-	x	x	x	0.917	-	3209	472
x	x	x	x	0.917	-	3209	474

did not eliminate the NaN values encountered during computation. The times are up to thirteen percent lower than those in Section 5.2.1.3.

The dense dataset `ds1.100pca` showed no overfitting or stability problems in Section 5.3.1.8 when `binitmean` was not in use. As expected, enabling the `binitmean` parameter does not improve computations for this dataset, as verified by Table 5.32. Without `rrlambda` enabled, these `binitmean` experiments are slightly faster than their counterparts in Table 5.29 when `margin` is disabled, and slightly slower when `margin` is enabled. When `rrlambda` is enabled, and `cgwindow` and `cgdecay` are disabled, the `binitmean` experiments are faster by up to eight percent. When `rrlambda`, `cgwindow` and `cgdecay` are enabled, the `binitmean` experiments are slightly slower.

For both datasets `ds1` and `ds1.100pca` it appears that enabling the `binitmean` parameter improves speed without affecting stability. We concluded in Section 5.3.1.10 that `rrlambda`, `cgwindow` and `cgdecay` should be used together. With those three parameters enabled along with `binitmean`, and the other stability parameters disabled, we find a thirteen percent speed-up for `ds1` and a slight slowdown for `ds1.100pca`. We conclude that the `binitmean` parameter should be enabled for CG-MLE experiments.

5.3.1.12 Stability Test: `cgwindow` Versus `cgdecay`

Tables 5.33 and 5.34 use a tight epsilon on datasets `ds1` and `ds1.100pca`, respectively, since a moderate epsilon shows little difference in the `cgwindow` and `cgdecay` experiments of Tables 5.25 and 5.29. Just as seen in the IRLS `cgwindow` versus `cgdecay` experiments of Section 5.2.1.13, `cgdecay` does not appear to have made a significant contribution to the speed of the algorithm. It does prevent overfitting, and would certainly help prevent out-of-control CG iterations that go beyond overfitting.

Comparing Tables 5.33 and 5.34 to the tight epsilon columns of Tables 5.25 and 5.29 we see that the

Table 5.33: CG-MLE stability experiments for `ds1` comparing `cgwindow` and `cgdecay`. Note the extra column so that enabling `cgwindow` is independent of enabling `cgdecay`.

					Moderate Epsilon			
mm	mar	rrl	cgw	cgd	AUC	NaN	DEV	Time
-	-	-	x	-	0.931	x	736	375
x	-	-	x	-	0.930	-	670	388
-	x	-	x	-	0.926	-	1038	369
x	x	-	x	-	0.926	-	1038	370
-	-	x	x	-	0.946	-	2309	171
x	-	x	x	-	0.946	-	2309	171
-	x	x	x	-	0.947	-	2214	190
x	x	x	x	-	0.947	-	2214	192
-	-	-	-	x	0.931	x	736	376
x	-	-	-	x	0.930	-	670	388
-	x	-	-	x	0.926	-	1038	370
x	x	-	-	x	0.926	-	1038	370
-	-	x	-	x	0.946	-	2309	312
x	-	x	-	x	0.946	-	2309	312
-	x	x	-	x	0.947	-	2214	324
x	x	x	-	x	0.947	-	2214	324

Table 5.34: CG-MLE stability experiments for `ds1.100pca` comparing `cgwindow` and `cgdecay`. Note the extra column so that enabling `cgwindow` is independent of enabling `cgdecay`.

					Moderate Epsilon			
mm	mar	rrl	cgw	cgd	AUC	NaN	DEV	Time
-	-	-	x	-	0.919	-	3275	1116
x	-	-	x	-	0.919	-	3275	1107
-	x	-	x	-	0.919	-	3154	1191
x	x	-	x	-	0.919	-	3154	1196
-	-	x	x	-	0.918	-	3310	659
x	-	x	x	-	0.918	-	3310	658
-	x	x	x	-	0.918	-	3181	672
x	x	x	x	-	0.918	-	3181	672
-	-	-	-	x	0.919	-	3275	1112
x	-	-	-	x	0.919	-	3275	1122
-	x	-	-	x	0.919	-	3154	1208
x	x	-	-	x	0.919	-	3154	1205
-	-	x	-	x	0.918	-	3310	1000
x	-	x	-	x	0.918	-	3310	997
-	x	x	-	x	0.918	-	3181	997
x	x	x	-	x	0.918	-	3181	996

scores are unchanged. The cgwindow times are similar to the cgwindow and cgdecay combination experiments, while the cgdecay times are similar to experiments in which both cgwindow and cgdecay were disabled. This suggests that termination is occurring before cgdecay takes effect, but after cgwindow does. Because cgwindow does not hurt scores in our experiments, we conclude that cgwindow should remain part of our stability parameters. We will not search for better cgdecay values, as explained in the arguments of Section 5.3.1.12. We will eliminate cgdecay based on its ineffectiveness and our desire to minimize the number of parameters.

5.3.1.13 Stability Tests: Conclusions

Sections 5.3.1.7, 5.3.1.10 and 5.3.1.11 summarize our empirical findings for the effectiveness of our stability parameters for CG-MLE experiments. Those results suggest discarding modelmin, modelmax and margin while retaining rrlambda, cgwindow, cgdecay and binitmean. Section 5.3.1.12 examined the separate utility of cgwindow and cgdecay, concluding that cgdecay was unnecessary. It appears that all of the remaining parameters should be used simultaneously.

For the remainder of the CG-MLE experiments in this thesis, we will activate rrlambda, cgwindow and binitmean. Figure 5.13 shows our updated LR description tree. Values for the rrlambda and cgwindow parameters will be decided in Section 5.3.1.14.

5.3.1.14 Stability Tests: Default Parameters

We have chosen to use rrlambda, cgwindow and binitmean in all of our later experiments. While binitmean is boolean and needs no optimization, rrlambda and cgwindow should be tested at various values and reasonable but fixed values chosen. Our experiments are similar to those of Section 5.2.1.15, as is our presentation and the format of Tables 5.35 and 5.36.

Perhaps the most notable aspect of Tables 5.35 and 5.36 is that the scores are more stable over the ranges of cgwindow and rrlambda than those in Section 5.2.1.15. The ds1 and ds1.100pca experiments of both tables show worse performance for rrlambda values 50, 100 and 500, while ds2 shows worse performance for rrlambda=1, and to some extent for rrlambda=5. The other three datasets show little variation across rrlambda values. Thus rrlambda=10 is our first choice, while rrlambda=5 is our second choice. Since rrlambda=10 is usually faster than rrlambda=5, we will use rrlambda=10.

In the rrlambda=10 column there is no variation of AUC score across cgwindow values, with one uninteresting exception. In Section 5.3.1.13 we concluded that cgwindow sometimes helped speed tremendously when used in concert with rrlambda. One would expect that cgwindow=1 would yield maximum speed so long as the score is unchanged, and Tables 5.35 and 5.36 verify this. A cgwindow of one will terminate iterations as soon as the deviance fails to improve. While experiments with rrlambda=50 are usually faster, there are occasional glitches in the AUC scores for such experiments.

For the remainder of our experiments, CG-MLE experiments will use rrlambda=10 and cgwindow=1. Our overall impression is that CG-MLE is less sensitive to its stability parameters than our version of IRLS is.

5.3.2 Direct (CG-MLE) Termination And Optimality

While it is tempting to think of the CG solver in CG-MLE as similar to the CG solver in IRLS, the two are very different. In IRLS we use CG to solve a linear system. In that context CG is an iterative exact method with numerical error. When using CG to minimize the non-quadratic convex LR log-likelihood surface, the

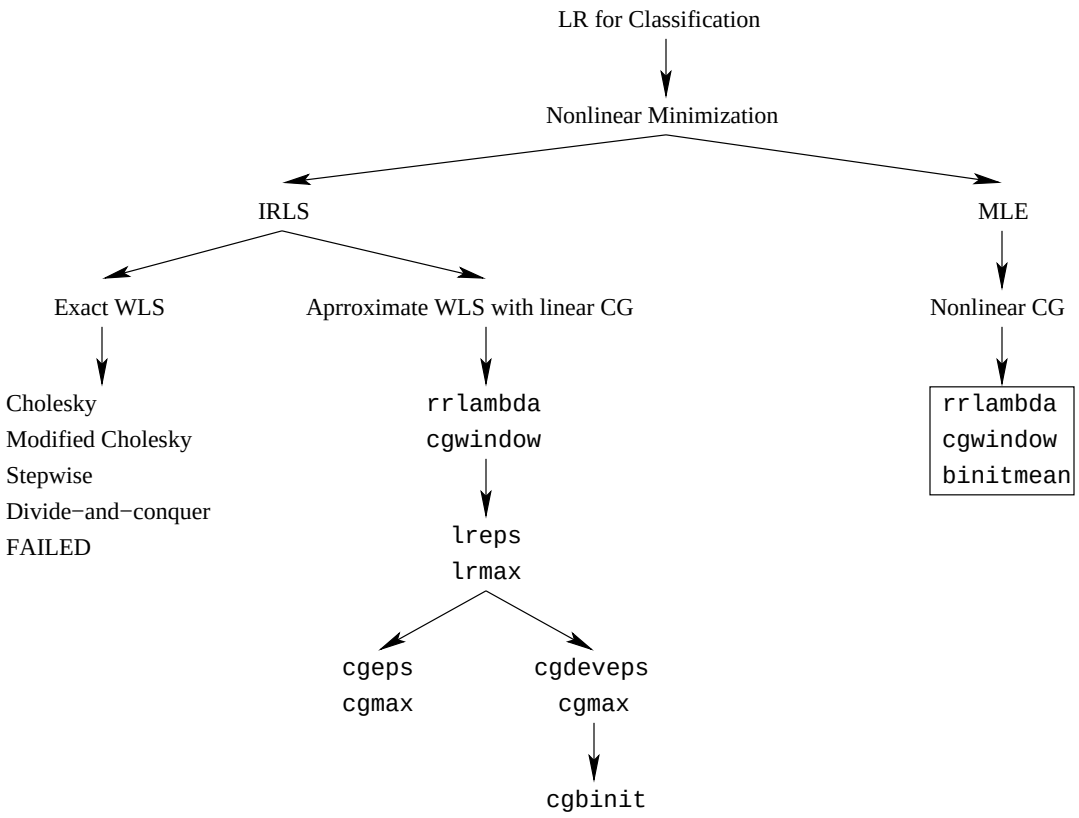


Figure 5.13: LR tree showing stability parameters selected for CG-MLE.

likelihood surface is approximated locally as a quadratic surface. The CG line search in IRLS is not actually a search; the step length is computed exactly and only one step is made. The CG line search in CG-MLE is a true search, and there are even competing criteria for choosing the direction of the line search, as explained in Section 2.2.

Unlike our IRLS implementation, the `cgeps` parameter for our CG-MLE implementation is not multiplied by the number of columns. As noted above, the CG-MLE `cgeps` parameter is the analogue of the IRLS `lreps` parameter. From an implementation view, it is very similar to `cgdeveps` since it is a non-residual termination test for CG. To ensure termination of CG iterations, we again have the `cgmax` parameter. These parameters are summarized in Table 5.24.

Figure 5.14 shows the relation between AUC and `cgeps` on the left, and between time and `cgeps` on the right. Note that the time axis is logarithmic. As `cgeps` decreases these graphs have the flat tails seen with `cgdeveps` in Figure 5.9. The AUC graph suggests that adequate convergence is obtained when `cgeps`=0.001. Just as with `cgdeveps` the `ds1.10pca` dataset is the most troublesome when choosing score versus time. Unlike the `cgdeveps` case, the tails in the time graph are already nearly flat by this point. We may choose the safer setting `cgeps`=0.0005 with little penalty. Table 5.37 shows the numerical results for `cgeps` values around 0.0005.

Table 5.35: Moderate Epsilon

		Time						AUC					
		rrlambda						rrlambda					
Dataset	cgw	1	5	10	50	100	500	1	5	10	50	100	500
ds1	1	268	181	144	84	73	42	0.944	0.947	0.946	0.930	0.919	0.883
	2	266	178	144	87	74	43	0.944	0.947	0.946	0.930	0.919	0.883
	3	269	181	144	88	74	44	0.944	0.947	0.946	0.930	0.919	0.883
	5	267	180	144	88	74	50	0.944	0.947	0.946	0.930	0.919	0.887
imdb	1	1084	704	470	513	495	427	0.980	0.983	0.983	0.983	0.983	0.987
	2	1083	748	503	543	517	427	0.980	0.983	0.983	0.983	0.983	0.987
	3	1082	752	608	578	525	428	0.980	0.983	0.983	0.983	0.983	0.987
	5	1087	751	608	579	526	426	0.980	0.983	0.983	0.983	0.983	0.987
citeseer	1	158	105	97	63	57	59	0.940	0.946	0.946	0.946	0.946	0.886
	2	160	105	98	61	59	61	0.940	0.946	0.946	0.946	0.946	0.886
	3	159	106	99	63	58	61	0.940	0.946	0.946	0.946	0.946	0.886
	5	166	107	100	59	59	59	0.940	0.946	0.946	0.946	0.946	0.886
ds2	1	4880	4058	3728	2683	2426	1065	0.693	0.717	0.723	0.722	0.719	0.636
	2	4871	4054	3712	2685	2426	1576	0.693	0.717	0.723	0.722	0.719	0.711
	3	4877	4062	3728	2709	2451	1592	0.693	0.717	0.723	0.722	0.719	0.711
	5	4880	4055	3729	2738	2482	1593	0.693	0.717	0.723	0.722	0.719	0.711
ds1.100pca	1	447	466	471	328	298	162	0.917	0.918	0.917	0.907	0.901	0.876
	2	442	466	470	326	297	177	0.917	0.918	0.917	0.907	0.901	0.876
	3	442	466	470	328	297	194	0.917	0.918	0.917	0.907	0.901	0.876
	5	444	465	470	328	298	304	0.917	0.918	0.917	0.907	0.901	0.890
ds1.10pca	1	40	44	46	29	30	24	0.839	0.839	0.841	0.837	0.843	0.832
	2	40	43	46	29	33	24	0.839	0.839	0.841	0.837	0.843	0.832
	3	40	44	46	29	32	24	0.839	0.839	0.841	0.837	0.843	0.832
	5	40	42	46	29	34	24	0.839	0.839	0.841	0.837	0.843	0.832

To prevent runaway CG-MLE iterations, we have the `cgmax` parameter. Table 5.38 shows the maximum number of CG iterations per fold of the experiments above using the default parameters. These maximums are not much different from the minimums for each fold. There is no evidence that any reasonable experiment will fail to terminate with these parameters, but for good measure we set `cgmax` to 100. The `cgeps` and `cgmax` parameters are shown in Figure 5.15 as part CG-MLE in our LR description tree.

5.3.3 Direct (CG-MLE) Speed

The simplicity of the CG-MLE approach leaves little to accelerate once termination is cared for. In this section we explore two techniques which do not fit into previous sections. The first technique modifies the CG direction update formula, and is controlled by the `dufunc` parameter. This parameter is summarized in Table 5.24, and described in more detail below. The second technique replaces CG with a different nonlinear minimizer. As will be seen, the experiments below do not prompt any change to our CG-MLE implementa-

Table 5.36: Tight Epsilon

		Time						AUC					
		rrlambda						rrlambda					
Dataset	cgw	1	5	10	50	100	500	1	5	10	50	100	500
ds1	1	349	212	159	83	75	42	0.945	0.948	0.946	0.930	0.919	0.883
	2	347	214	161	87	78	44	0.945	0.948	0.946	0.930	0.919	0.883
	3	351	217	164	91	82	54	0.945	0.948	0.946	0.930	0.919	0.884
	5	354	224	171	97	89	79	0.945	0.948	0.946	0.930	0.919	0.899
imdb	1	1109	727	476	512	496	429	0.980	0.982	0.983	0.983	0.983	0.987
	2	1145	897	559	540	522	457	0.980	0.983	0.983	0.983	0.983	0.987
	3	1182	949	726	580	553	461	0.980	0.983	0.983	0.983	0.983	0.987
	5	1256	1010	790	650	569	459	0.980	0.983	0.983	0.983	0.983	0.987
citeseer	1	176	115	99	64	58	61	0.940	0.946	0.946	0.946	0.946	0.886
	2	183	122	106	67	63	64	0.940	0.946	0.946	0.946	0.946	0.886
	3	199	125	111	67	63	64	0.940	0.946	0.946	0.946	0.946	0.886
	5	216	133	119	67	63	65	0.940	0.946	0.946	0.946	0.946	0.886
ds2	1	4941	4253	3975	2703	2623	1062	0.693	0.718	0.723	0.722	0.724	0.636
	2	4987	4294	4031	2757	2678	1717	0.693	0.718	0.723	0.722	0.724	0.711
	3	5073	4382	4110	2850	2765	1787	0.693	0.718	0.723	0.722	0.724	0.711
	5	5200	4498	4249	2985	2908	1918	0.693	0.718	0.723	0.722	0.724	0.711
ds1.100pca	1	951	673	606	329	317	162	0.919	0.918	0.918	0.907	0.902	0.876
	2	944	681	611	338	330	175	0.919	0.918	0.918	0.907	0.902	0.876
	3	955	701	630	356	342	195	0.919	0.918	0.918	0.907	0.902	0.876
	5	956	723	651	488	401	315	0.919	0.918	0.918	0.907	0.901	0.890
ds1.10pca	1	84	81	68	29	31	27	0.847	0.846	0.847	0.837	0.843	0.836
	2	84	82	72	33	32	29	0.847	0.846	0.847	0.837	0.843	0.835
	3	85	83	79	61	34	30	0.847	0.846	0.847	0.845	0.843	0.835
	5	83	83	84	64	38	33	0.847	0.846	0.847	0.845	0.843	0.835

Table 5.37: AUC and Times For Several CG-MLE cgeps Values

cgeps	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
0.001	0.946	144	0.983	470	0.946	97	0.723	3728	0.917	471	0.841	46
0.0005	**0.946**	**152**	**0.983**	**469**	**0.946**	**98**	**0.724**	**3784**	**0.916**	**506**	**0.844**	**50**
0.0001	0.946	158	0.983	471	0.946	99	0.723	3872	0.918	585	0.847	63

Table 5.38: Maximum CG iteration counts for all folds of CG-MLE experiments.

ds1	imdb	citeseer	ds2	ds1.100pca	ds1.10pca
41	5	4	42	34	34

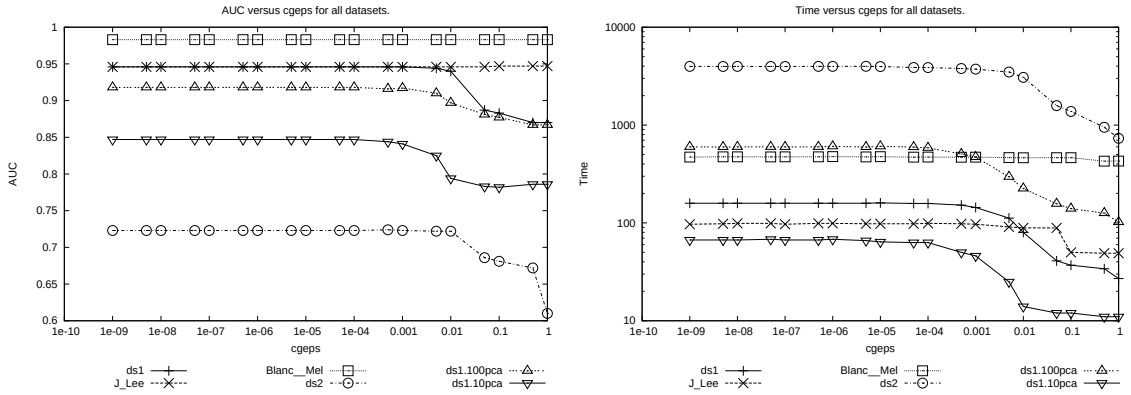


Figure 5.14: AUC and time versus `cgeps` on several datasets with `lreps`=0.05 for CG-MLE. Note that the time axis is logarithmic.

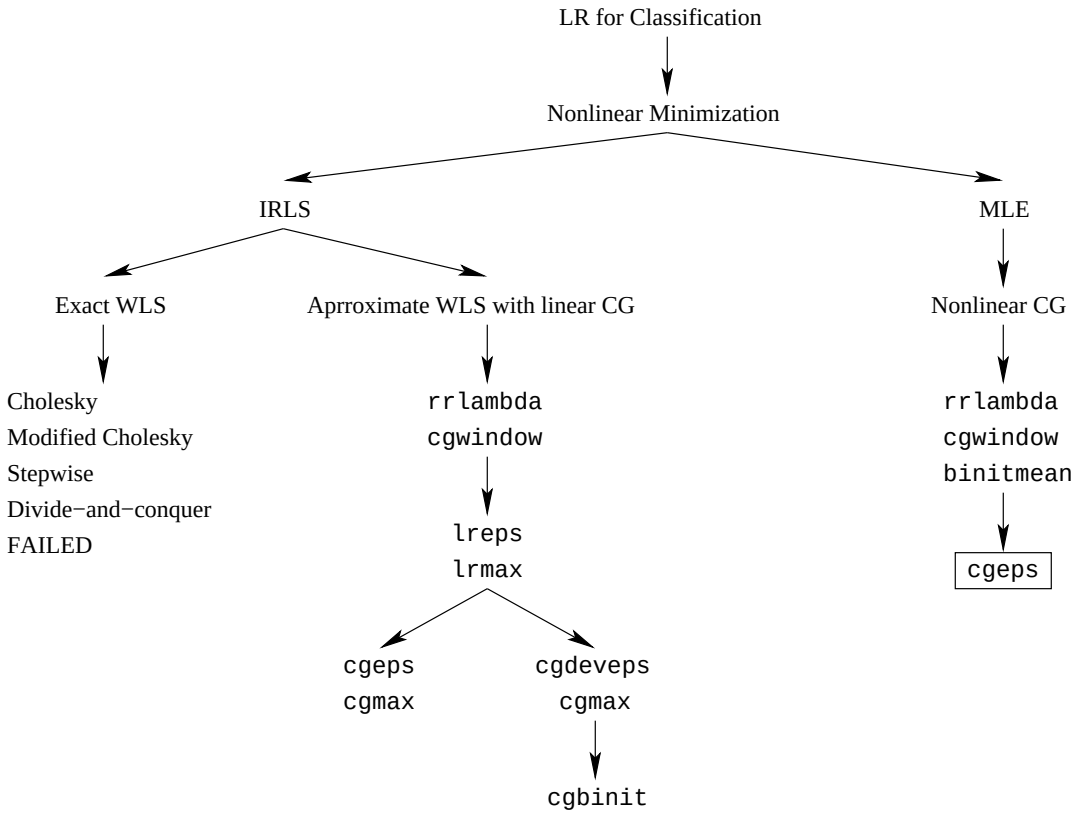


Figure 5.15: LR tree showing single termination parameter used with CG-MLE, along with the safeguard parameter `cgmax`.

Table 5.39: CG-MLE performance experiments for `ds1` with `binitmean` disabled while varying the `dufunc` parameter. The moderate epsilon was used.

				mod Polak-Ribiére			Polak-Ribiére			Hestenes-Stiefel			Fletcher-Reeves		
`mm`	`mar`	`rrl`	`cgw`	AUC	DEV	Time	AUC	DEV	Time	AUC	DEV	Time	AUC	DEV	Time
-	-	-	-	0.931	736	368	0.931	736	368	0.912	3254	192	0.933	658	460
x	-	-	-	0.930	670	430	0.930	670	430	0.912	3254	193	0.933	658	461
-	x	-	-	0.926	1038	415	0.926	1038	414	0.913	3096	188	0.926	934	471
x	x	-	-	0.926	1038	409	0.926	1038	409	0.913	3096	193	0.926	934	470
-	-	x	-	0.946	2313	164	0.946	2313	166	0.914	3262	183	0.946	2333	165
x	-	x	-	0.946	2313	166	0.946	2313	166	0.914	3262	184	0.946	2333	165
-	x	x	-	0.946	2228	165	0.946	2228	163	0.907	3227	177	0.946	2234	167
x	x	x	-	0.946	2228	164	0.946	2228	164	0.907	3227	178	0.946	2234	168
-	-	-	x	0.931	736	368	0.931	736	367	0.912	3254	199	0.933	658	460
x	-	-	x	0.930	670	429	0.930	670	430	0.912	3254	199	0.933	658	460
-	x	-	x	0.926	1038	409	0.926	1038	410	0.913	3096	205	0.926	934	467
x	x	-	x	0.926	1038	408	0.926	1038	411	0.913	3096	196	0.926	934	468
-	-	x	x	0.946	2313	174	0.946	2313	167	0.914	3262	186	0.946	2333	166
x	-	x	x	0.946	2313	165	0.946	2313	164	0.914	3262	194	0.946	2333	165
-	x	x	x	0.946	2228	163	0.946	2228	170	0.907	3227	179	0.946	2234	167
x	x	x	x	0.946	2228	164	0.946	2228	163	0.907	3227	176	0.946	2234	166

tion, and hence the LR description tree shown in Figure 5.15 is still valid.

5.3.3.1 Nonlinear CG Direction Updates

There is only one suitable direction update for the linear CG used in our IRLS implementation. The nonlinear CG used in CG-MLE has several reasonable direction update formulas, all of which are identical when applied to objective functions with constant Hessian. Our CG-MLE implementation uses the modified Polak-Ribiére direction update described in Section 2.2, based on experience and early CG-MLE experiments. The other direction updates we considered were Polak-Ribiére, Hestenes-Stiefel, and Fletcher-Reeves. These are also described in Section 2.2. Table 5.39 presents the results of stability experiments with several direction updates and a moderate epsilon on dataset `ds1`. In this table Hestenes-Stiefel has AUC scores too low to be considered, while Fletcher-Reeves has good scores but erratic times. Because modified Polak-Ribiére has a built-in orthogonality check, it was preferred over the otherwise identical Polak-Ribiére.

It is interesting to re-examine these direction updates now that default parameters have been chosen for CG-MLE. Because we are no longer burdened with a large set of parameter combinations, we can succinctly compare performance on all six datasets used in this section. Table 5.40 contains the results. For compactness we have abbreviated modified Polak-Ribiére as MPR, Polak-Ribiére as PR, Hestenes-Stiefel as HS and Fletcher-Reeves as FR. While Polak-Ribiére is again virtually identical to modified Polak-Ribiére and Hestenes-Stiefel scores poorly, we see that Fletcher-Reeves is a strong performer. Revisiting Table 5.39 we observe that Fletcher-Reeves behaved similarly to modified Polak-Ribiére when regularization was used. This is now the default. We may also observe that Fletcher-Reeves is occasionally faster than modified Polak-Ribiére. Though forty-one percent slower on `imdb`, Fletcher-Reeves is thirty-two percent faster on `citeseer`.

Table 5.40: AUC and Times For Several CG Direction Update Formulas

dufunc	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
MPR	**0.946**	**152**	**0.983**	**469**	**0.946**	**98**	**0.724**	**3784**	**0.916**	**506**	**0.844**	**50**
PR	0.946	153	0.983	470	0.946	99	0.724	3778	0.916	486	0.844	50
HS	0.923	238	0.983	1282	0.946	95	0.722	7181	0.895	631	0.800	43
FR	0.946	155	0.983	659	0.946	67	0.724	3369	0.916	463	0.845	52

On datasets `ds2` and `ds1.100pca`, Fletcher-Reeves is a more moderate eleven percent and eight percent faster than modified Polak-Ribiére. As a result it is difficult to choose between modified Polak-Ribiére and Fletcher-Reeves. Because of modified Polak-Ribiére's winning performance in earlier experiments, we will continue using modified Polak-Ribiére.

5.3.4 BFGS-MLE

The second technique is an alternative to CG-MLE. Instead of using CG for nonlinear minimization we use a Newton-like method which approximates the Hessian matrix without forming it explicitly. Our choice of Hessian approximations is the BFGS update, named for authors Broyden, Fletcher, Goldfarb and Shanno. We will call this LR variant BFGS-MLE. We rely on the GNU Scientific Library [8], or GSL, for our BFGS-MLE experiments. While they call their method `gsl_multimin_fdfminimizer_vector_bfgs()`, the usual name for such algorithms is limited-memory quasi-Newton methods. Further information on these methods can be found in Nash and Sofer [31]. For the remainder of this section BFGS will mean a limited memory quasi-Newton method with a BFGS Hessian approximation.

We will use the term BFGS to refer to both the BFGS matrix update, and to a quasi-Newton method using the BFGS matrix update. Because BFGS is a quasi-Newton method, it is somewhat similar to IRLS in the context of LR. See Section 4.3 for further explanation of the relation between IRLS and Newton's method. Where our IRLS approximates the solution to linear subproblems using expressions involving the true Hessian, BFGS approximates the Hessian and finds true solutions given that approximation. Because BFGS is applied directly to the MLE system of equations, it is reminiscent of CG-MLE. Note that all three of these algorithms will follow a different path on the MLE surface and hence can be expected to perform differently.

BFGS can be terminated when the size of the estimated gradient is near zero, or using the relative difference of the deviances. The GSL BFGS implementation returns after every line search and after every line search iteration. Computing the deviance each time is too expensive and the change in deviance is too small to make the relative change approach work well. The alternatives are to identify line search completion by examining the sequence of points generated along the line search, or to use the gradient size for termination. The GSL documentation seems to encourage gradient-based termination, and hence this is our chosen termination method.

We do not intend to investigate BFGS-MLE thoroughly in this thesis, and only wish to make a rough comparison of the time it takes to converge to a competitive AUC . To this end we have run twelve experiments on each of our six datasets. These twelve experiments vary three parameters of GSL's BFGS implementation. The first of these is the gradient-termination epsilon `gradeps` for which we try values 0.01, 0.001 and 0.0001. The second parameter is `initalpha`, the initial step-length used by the line search. We do not anticipate that BFGS will be particularly sensitive to this parameter since our optimization surface is convex, and we only try

Table 5.41: Best BFGS AUC

	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
BFGS-MLE	0.943	258	0.983	2775	0.947	755	0.716	4651	0.912	403	0.831	89
CG-MLE	0.946	152	0.983	469	0.946	98	0.724	3784	0.916	506	0.844	50

Table 5.42: Fastest BFGS time for "good" AUC scores.

	ds1		imdb		citeseer		ds2		ds1.100pca		ds1.10pca	
BFGS-MLE	0.940	166	0.982	2628	0.947	740	0.716	4651	0.911	396	0.826	81
CG-MLE	0.946	152	0.983	469	0.946	98	0.724	3784	0.916	506	0.844	50

values 0.01 and 0.000001 to be certain. The final parameter is `lineeps`, the tolerance applied to the relative orthogonality of the search direction and the gradient for the current line-search iteration. We were unsure how to tune this parameter, and tested its effect by setting it to 0.01 or 0.000001. All of these parameters are part of the GSL BFGS minimizer. We used the same `cgwindow`, `rrlambda` and `binitmean` settings chosen in our CG-MLE tests.

Our experiments confirm that the value of `initalpha` had little effect on time or AUC. The same can be said of `lineeps`, with the exception of experiments on dataset `ds1`. It seems likely that a self-tuning version of `lineeps` similar to $\mathbf{r}_0$-`cgeps` of Section 5.2.2 would perform adequately. For `gradeps` the value 0.01 always produced poor AUC scores, while there was no obvious pattern to whether 0.001 or 0.0001 would produce the optimal AUC or speed. Thankfully there was little variation between AUC and speed for `gradeps`=0.001 and `gradeps`=0.0001.

The best results of our experiments with BFGS-MLE are summarized in Tables 5.41 and 5.42. The first of these tables shows the best AUC value obtained over the twelve parameter combinations for each dataset. The second table shows the fastest experiment of the twelve which produces an AUC similar to the best AUC values of the first table. BFGS achieved similar AUC scores to those found by IRLS in Tables 5.19 and 5.20 and CG-MLE in Table 5.37. The fastest times are not competitive with the other algorithms. The times for BFGS experiments with `gradeps`=0.01 are much faster than those shown in Table 5.42, but some AUC scores in that table are approaching sub-optimal and all scores are miserable once `gradeps`=0.01.

Our BFGS-MLE experiments lead us to believe that proper tuning and careful attention to detail will not produce significant gains over the other LR methods we have described. We will not explore BFGS-MLE further in this thesis.

5.3.5 Direct (CG-MLE) Summary

We have found that CG-MLE requires many of the same stability parameters, with nearly identical default values, as IRLS did. Notably different was the ease with which the termination epsilon `cgeps` was found in Section 5.3.2. Perhaps the most interesting results are in Section 5.3.3, which examined the effects of different CG direction update formulas and the BFGS-MLE.

Our final CG-MLE method is summarized in Algorithm 8. Starting at line 8.1 we set our nonlinear CG parameters as we have described above. Line 8.2 shows how the `binitmean` parameter changes the value of

$\hat{\beta}_0$. Several lines related to `cgwindow` are present. As with our IRLS implementation, shown in Algorithm 5, we return the value of $\hat{\beta}$ which minimized the deviance. This may be seen in line 8.3. As in Algorithm 5, we have embedded the parameters `cgmax`, `cgeps`, and `cgwindow` to emphasize that we have fixed their values.

It is worth emphasizing the difference between CG-MLE and IRLS with `cgdeveps`. Both use the relative difference as a termination criterion. However, IRLS is a different nonlinear optimization method than nonlinear CG. The first IRLS iteration starts from the same place as the first CG-MLE iteration. In IRLS, linear CG is applied to a linear weighted least squares problem. In CG-MLE, nonlinear CG is applied to the score equations. Termination of linear CG for the first IRLS iteration is very unlikely to occur at the same place as termination occurs for nonlinear CG applied to the LR score equations, though linear CG should terminate with far fewer computations. At this point there are more IRLS iterations to run, but CG-MLE is finished. While both algorithms should ultimately arrive at similar parameter estimates, there is no reason to believe they will take the same path to get there, or require the same amount of computation. That both algorithms apply the same termination criteria to their version of CG is at best a superficial similarity.

We do not have many new ideas to propose for optimizing the LR MLE using numerical methods. Regularization was already suggested by Zhang and Oles [49]. Minka [27] made a brief comparison of several numerical methods, including a quasi-Newton method algorithm called *Böhning's method*, in a short technical report. Minka mentioned the need for regularization, and in two of his three datasets found that CG outperformed other algorithms. Zhang et al. [48] preferred Hestenes-Stiefel direction updates when using CG for nonlinear convex optimization, which is somewhat at odds with the conclusions of this chapter. We do not have an explanation for this contradiction. In fact, the most promising line of exploration opened by this section is the possibility that Fletcher-Reeves updates, usually ignored for nonlinear CG, might work well in our environment.

input : $\mathbf{X}$, $\mathbf{y}$

output : $\hat{\beta}$ which maximizes LR likelihood

/*`Initialization.` */

8.1 $g(\hat{\beta}) = \ln \mathbb{L}(\mathbf{X}, \mathbf{y}, \hat{\beta}) - \lambda \hat{\beta}^T \hat{\beta}$

du = modified Polak-Ribiére

ri$(\mathbf{r_i}, \mathbf{r_{i-1}}, \mathbf{d_{i-1}}, i)$ = $I_{\text{ mod }(i,R)=0}$

ls = Secant method with relative and absolute slope checking (Section 2.2)

8.2 $\hat{\beta}_0 := ((\sum_{i=1}^{R} y_i)/R, 0, \ldots, 0)^T$

$\mathbf{r_0} := -g'(\hat{\beta}_0)$

$\text{DEV}_{-1} := \infty$

$\text{DEV}_0 := \text{DEV}(\hat{\beta}_0)$

devmin := ∞

window := `cgwindow`

/*`Iterations.` */

for i := 0 ...`cgmax`-1 **do**

  /*`Termination.` */

  if $|(\text{DEV}_{i-1} - \text{DEV}_i)/\text{DEV}i| <$ `cgeps` **then break**

  if window ≤ 0 **then break**

  /*`Regular Nonlinear CG.` */

  if i=0 **then** $\gamma_i := 0$

  else $\gamma_i :=$ du$(\mathbf{r_i}, \mathbf{r_{i-1}}, \mathbf{d_{i-1}})$

  if ri$(\mathbf{r_i}, \mathbf{r_{i-1}}, \mathbf{d_{i-1}}, i)$ **then** $\gamma_i := 0$

  $\mathbf{d_i} := \mathbf{r_i} + \gamma_i \mathbf{d_{i-1}}$

  $\hat{\beta}_{\mathbf{i+1}} :=$ ls$(\hat{\beta}_{\mathbf{i}}, \mathbf{d_i})$

  $\mathbf{r_{i+1}} := -g'(\hat{\beta}_{\mathbf{i+1}})$

  /*`cgwindow` */

  $\text{DEV}_{i+1} := \text{DEV}(\mathbf{x_{i+1}})$

  if $\text{DEV}_{i+1} \leq$ devmin **then**

    window := `cgwindow`

    devmin := DEV_{i+1}

  else window := window - 1

/*`Find best` $\hat{\beta}$. */

8.3 $i^* := \text{argmin}_{j=0\ldots i}\ \text{DEV}_j$

$\hat{\beta} := \hat{\beta}_{i^*}$

Algorithm 8: Our CG-MLE implementation.

Chapter 6

Characterizing Logistic Regression

In previous chapters we discussed various ways to implement and control LR. In particular, Chapter 5 chooses a set of LR controls and enhancements by analyzing performance on a collection of real-world datasets. The result was three versions of LR: IRLS with `cgeps`, IRLS with `cgdeveps`, and MLE with CG. These will be called LR-CGEPS, LR-CGDEVEPS, and CG-MLE in this chapter. In this chapter we will examine how these LR implementations compare to each other and to other classifiers.

Section 6.1 discusses several classifiers we will use in our comparisons. These are support vector machines (SVM), k-nearest-neighbor (KNN) and Bayes' Classifier (BC). We are not testing decision trees on these datasets because we expect very poor performance on the linearly-separable synthetic datasets, and we have observed poor performance on our real-world datasets. It is possible that other varieties of tree-based classifiers, such as bagged or boosted decision trees, would perform better. However, we do not have an implementation of decision trees other than C4.5 [38].

Section 6.2 contains experimental results on the synthetic datasets described in Section 5.1.3.3. These datasets help us measure classifier score and speed as a function of the dataset's number of rows, number of attributes, sparsity and attribute coupling. In Section 6.3 we have a showdown between all the classifiers on the six real-world datasets described in Sections 5.1.3.1 and 5.1.3.2.

To compare the classifiers we will use AUC scores and times. The AUC score is defined in Section 5.1.4. In particular we will include a 95% confidence interval for all AUC scores. Our method of timing is described in Section 5.1.5. Efforts have been made to find reasonable parameter choices for these classifiers. Where possible we have used the implementor's default settings unless performance was unreasonable and a better value was found. Non-default parameter choices will be described in Section 6.1. Please note that the timings in this chapter are from computers with the new BIOS, as explained in Section 5.1.5. For this reason, these timings are generally faster than the timings in Chapter 5. The one exception is KNN, which appears to run between five and fifteen percent slower on the new BIOS. The larger the dataset, the smaller the KNN slowdown. However, this slowdown is immaterial in the comparisons to other classifiers.

6.1 Classifiers

We assume that LR has seen enough discussion in the preceding chapters, and use this space to briefly describe the SVM, KNN and BC classifiers. However, we will take a moment to refresh our memory of the time complexity of our LR implementations. We made the claim that LR-CGEPS and LR-CGDEVEPS

have time complexity $\mathrm{O}(m)$ where m is the number of nonzero entries in the matrix X in Section 5.1.2. CG-MLE has the same time complexity as CG, which is again $\mathrm{O}(m)$. We will use LR to refer to the three LR implementations collectively.

6.1.1 SVM

To describe the SVM classifier, we must first make several definitions. Let S_- be the set of negative rows from $\mathbf{X}$, and S_+ be the set of positive rows. Suppose that S_- and S_+ are linearly separable. A *separating hyperplane* is a hyperplane with unit normal β and distance to the origin β_0 such that

$$\forall \mathbf{x} \in S_+ \quad \beta^T \mathbf{x} + \beta_0 > 0 \tag{6.1}$$

$$\forall \mathbf{x} \in S_- \quad \beta^T \mathbf{x} + \beta_0 < 0 \tag{6.2}$$

That is to say all of S_+ is on one side of the separating hyperplane while all of S_- is on the other side. Define the *positive hyperplane* as a hyperplane parallel to the separating hyperplane which touches the convex hull of S_+, that is

$$\forall \mathbf{x} \in S_+ \qquad \beta^T \mathbf{x} + \beta_+ \geq 0 \tag{6.3}$$

$$\exists \mathbf{x} \in S_+ \qquad \text{such that } \beta^T \mathbf{x} + \beta_+ = 0 \tag{6.4}$$

Define the *negative hyperplane* in the same manner. The *margin* is the space bounded by the positive and negative hyperplanes.

For datasets with binary outcomes, we can code each outcome y_i as 1 and -1 instead of 1 and 0. With this coding we can unify Equations 6.1 and 6.2 as

$$y_i(\beta^T \mathbf{x} + \beta_+) > 0 \tag{6.5}$$

If the sets S_+ and S_- are not linearly separable then our definition of a separating hyperplane can be adjusted to allow exceptions. For instance we can allow violations by replacing Equation 6.5 with

$$y_i(\beta^T \mathbf{x} + \beta_+) \geq C \tag{6.6}$$

where, for instance, a small value C would allow only minor infractions. Observe that when the data is linearly separable, C can be as large as the half the width of the margin without any violations occurring. Though the margin as defined above does not exist for data which is not linearly separable, it is common to refer to C as the size of the margin.

SVMs are often described as a *maximum margin* classifier because they choose a separating hyperplane which maximizes C under some constraint on the number and size of separation violations. A precise definition of an SVM can be written as the mathematical program

$$\begin{array}{lrcl} \text{Maximize} & C & & \\ \text{subject to} & y_i(\beta^T \mathbf{x_i} + \beta_+) & \geq & C(1-\xi_i) \\ & \sum_{i=1}^{R} \xi_i & \leq & D \\ & \xi_i & \geq & 0 \end{array} \tag{6.7}$$

where D is a restriction on the cumulative size of the violations ξ_i and $\mathbf{x_i}, y_i$ come from the dataset. This formulation of SVMs roughly follows that of Hastie et al. [10], where it is shown that the above mathematical

program is equivalent to the quadratic program

$$
\begin{array}{llll}
QP: & \text{Maximize} & \sum_{i=1}^{R} \alpha_i - \frac{1}{2}\sum_{i=1}^{R}\sum_{j=1}^{R} \alpha_i \alpha_j y_i y_j \mathbf{x}^T \mathbf{x} & \\
 & \text{subject to} & \alpha_i = \gamma - \mu_i & \\
 & \sum_{i=0}^{R} \alpha_i y_i & = 0 & \\
 & \alpha_i & \leq \gamma & \\
 & \alpha_i, \mu_i, \xi_i & \geq 0 &
\end{array}
\tag{6.8}
$$

The SVM described above is a *linear SVM*. It may be desirable to apply a nonlinear transform to data which is not linearly separable before applying SVM. The separating hyperplane found for the transformed space becomes a nonlinear boundary in the original space. A common transformation uses *Radial Basis Functions*, which are simply a basis created from a properly scaled probability density function (pdf). Note that this pdf is not interpreted probabilistically. [10; 34]

We did not implement SVM, and instead chose to use the popular SVM^{light} package, version 5 [18; 16]. This software uses a variety of tricks to find solutions to QP, defined in Equation 6.8, more quickly than traditional quadratic programming software. SVM^{light} can function as a linear or RBF SVM, where the RBF SVM uses the Normal pdf and scale parameter `gamma`. The default parameters for the linear SVM work well in our experiments. The default value of `gamma` made RBF SVM experiments too slow, and we adjusted it to `gamma`=0.001. Smaller values of `gamma` did not improve speed further. Joachims [16] describes the time complexity of SVM^{light} in a manner that we cannot easily describe here. While this complexity is bounded above by $\mathrm{O}\left(R^2 M\right)$, much better performance is generally observed.

We have written a wrapper around SVM^{light} so that it may be timed in the same manner as the other algorithms, as well as report scores in the same manner as the other algorithms. Before calling SVM^{light}, we must first write data to a file in a special format. When the training phase is done, we read data from a file created by SVM^{light}. A similar process happens during classification. This is repeated for every fold. We assume that SVM^{light} spends the same amount of time reading the files as our SVM wrapper spends writing them, and vice-versa. Let the *filetime* be the amount of time we spend reading and writing files. We do not want to include the filetime in the published algorithm time. Therefore we report the algorithm time as we normally measure it, less twice the filetime.

6.1.2 KNN

KNN is a well-known non-parametric classifier. The intuition for KNN is simply to use the k nearest data points to determine whether or not a test point $\mathbf{x}$ is positive. If $k > 1$, some method of combining k binary values into a single prediction is needed. For instance, one might predict the class assigned to the majority of the k nearest neighbors.

This geometric idea requires a distance metric, and the Euclidean distance is commonly used. A naive implementation of KNN would keep the training data in memory, and compute the distance between all R data points and the test point. Since each distance computation requires M time, or MF time on average if the data is sparse, classifying a single test point is $\mathrm{O}\left(MRF\right)$. If another test point were used, another R distance evaluations are made. Thus the weakness of KNN is that it ignores structure in the data, and making more than a few classifications can be very expensive.

One non-naive approach to KNN uses one or more space-partitioning trees to organize the training data according to its geometric structure. In our comparisons, we use the ball tree-based KNS2 algorithm described in Liu et al. [24]. The KNS2 algorithms uses two ball trees to separately store the positive and

negative training examples. If the data has a very uneven class distribution, then one of the ball trees will be small.

Suppose for concreteness that the positive class is small, as is the case with our real-world datasets. To decide whether a test point is positive or negative, KNS2 first finds the k nearest positive points to the test point. This is fast because the positive tree is small. Then KNS2 analyzes the negative points in the larger tree, stopping as soon as it can determine the exact number of negative points among the k nearest neighbors. The ratio of positive to negative nearest neighbors is used to rank the test point.

Building a ball tree requires $\mathrm{O}(FMR\log R)$ time if the tree is balanced. Once the positive and negative trees are built, the time required for the necessary queries will depend on the geometric structure of the data. If the data is clustered and the class distribution is skewed, queries will be very fast and KNS2 should vastly outperform naive KNN. In such cases KNS2 can scale to much larger datasets than have been successfully attempted with naive KNN. If the data is uniformly distributed and there are as many positive as negative points, the ball trees will be useless and queries will be relatively slow. This latter scenario is roughly the case for our synthetic datasets. A further problem occurs for non-clustered high-dimensional data. In such data all data points are equidistant from one another, invalidating the intuition behind KNN [9].

For our experiments, we tried $k = 1$, $k = 9$, and $k = 129$ due to our experience with KNS2 on real-world datasets. We can expect poor predictions with high variance from values of k which are too small, while large values of k increase the time required to compute the majority class.

We had some difficulty during ball tree construction in KNS2. On many synthetic datasets, and on the larger real-world datasets, ball trees with the default settings required more than 8GB of memory. We were able to decrease the size of the ball tree to under 4GB by increasing the `rmin` parameter, which controls the number of training rows stored in the leaf nodes. However, the resulting trees did not successfully accelerate KNS2. Such experiments were abandoned and are absent from our graphs and tables.

6.1.3 BC

Bayes' classifier, also known as Naive Bayes, is a simple application of Bayes' rule to allow computation of the probability that row i is a positive experiment. If we assume that the attributes are conditionally independent given the outcome, that is $P(x_{ij}|x_{ik},y_i) = P(x_{ij}|y_i), j \neq k$, then we may apply Bayes' rule to compute

$$P(Y=1|\mathbf{X},\mathbf{y},\tilde{\mathbf{x}}) = \frac{P(Y=1)\prod P(\tilde{x_j}|Y)}{\prod P(\tilde{x_j}|Y=0)P(Y=0)+\prod P(\tilde{x_j}|Y=1)P(Y=1)} \tag{6.9}$$

where $\tilde{\mathbf{x}}$ is the experiment for which we wish to predict the outcome, and $P(Y=1)$ and $P(\tilde{x_j}|Y)$ are estimated using the dataset $\mathbf{X},\mathbf{y}$. Thus BC has no parameters. Another approach to BC allows the class prior probabilities $P(Y=1)$ and $P(Y=0)$ to be chosen by the experimenter. However, changing the class prior distribution does not change the rank-ordering of the predictions, and hence does not affect the AUC score.

There are two principal problems with BC. The first and less important problem is that the attributes are rarely conditionally independent, making the predictions suspect. The second problem is that BC is a *generative* classifier. A generative classifier is one which attempts to learn the joint probability $P(Y,\mathbf{x})$ and makes predictions by forming the conditional distribution $P(Y|\mathbf{x})$. This indirect approach contrasts with that of a *discriminative* classifier, such as LR, which estimates $P(Y,\mathbf{x})$ directly from the data. The result is that generative classifiers are not directly optimizing the quantity of interest [33]. Russel and Norvig [39] contains a nice description of BC.

Computing $P(Y=1)$ and $P(\tilde{x_j}|Y)$ only needs to be done once, thus BC is always fast. By exploiting

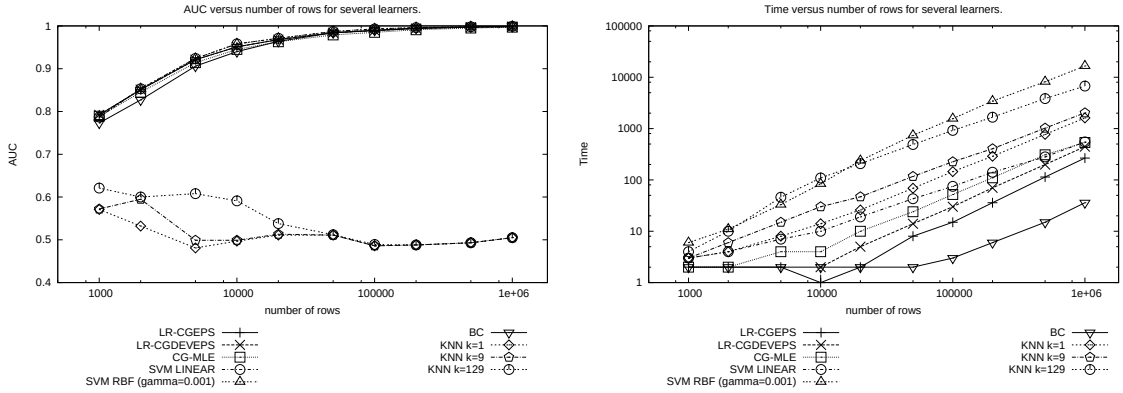


Figure 6.1: AUC and time versus number of rows. Note that the vertical time axis in the right plot is logarithmic.

sparsity and using a few computation tricks, we have made our implementation of BC very fast. Our implementation is linear in the number of nonzero entries of $\mathbf{X}$.

6.2 Synthetic Datasets

We would like to compare LR-CGEPS, LR-CGDEVEPS, CG-MLE with the classifiers described in Section 6.1 in a controlled environment. To that end we have run experiments on the synthetic datasets of Section 5.1.3.3 to examine how performance is affected by the number of rows, number of attributes, sparsity and attribute coupling of the data. We allowed twenty-four hours per classification task. We do not report any results longer than this period.

6.2.1 Number of Rows

Figure 6.1 shows how the classifiers performed as the number of rows in the dataset increased. Note that there is no coupling between the attributes of this dataset, as shown in Table 5.2. Furthermore, half of all dataset rows are positive. This is quite unlike the real-world datasets we will see later in this chapter.

The most obvious feature is the poor KNN AUC scores, shown on the left side of the figure. We will see that KNN does poorly on all of our synthetic datasets, and we attribute this to the independence of the attributes, the linear boundaries, and the equal number of positive and negative rows. We will see in Section 6.2.4 that KNN performs somewhat better as the coupling between attributes increases, and in Section 6.3 that KNN surpasses all of the other classifiers on our PCA-compressed datasets `ds1.100pca` and `ds1.10pca`. There is little interesting happening between the other classifiers in this AUC plot.

On the other hand, it is easy to differentiate the learners in the time plot, which is on the right side of Figure 6.1. Note that the vertical time axis is logarithmic, and hence a unit change vertically represents an order of magnitude change in the number of seconds required to make classifications. The learners capable of representing nonlinear classification boundaries, namely KNN and SVM RBF, are the slowest. In Section 6.3

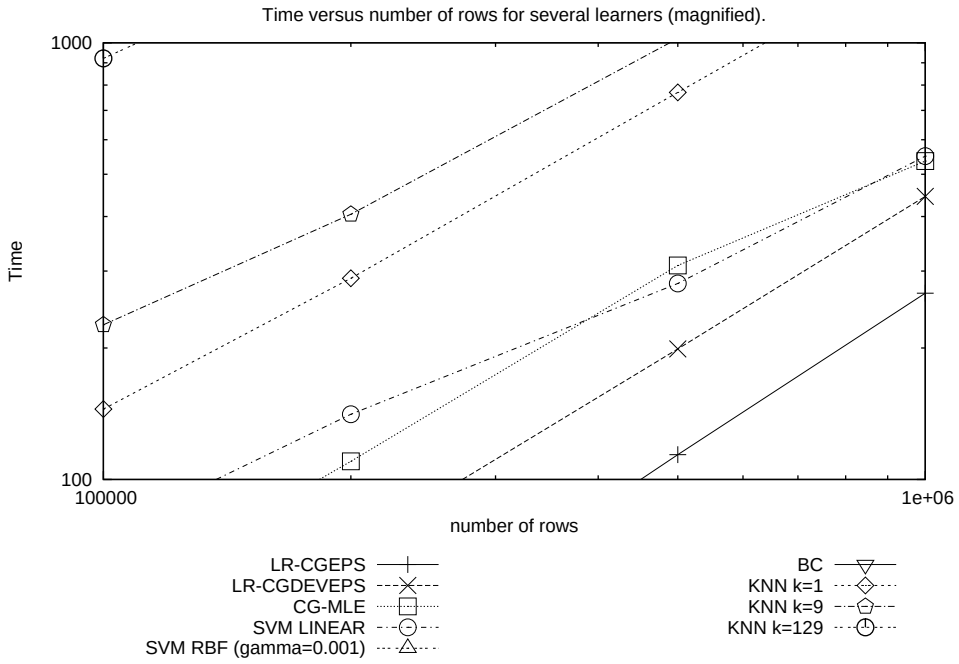


Figure 6.2: AUC versus sparsity, magnified.

we will be able to evaluate whether it is worthwhile to spend extra time on nonlinear boundaries for high-dimensional real-world data.

It appears that the LR classifiers are faster than SVM LINEAR. Though the CG-MLE and SVM LINEAR graphs cross, as shown in Figure 6.2, they also cross a second time. Nevertheless, the effectiveness of the SVM^{light} optimizations for solving the SVM quadratic program is remarkable. Between the LR classifiers, it appears that LR-CGEPS has a small advantage. Careful inspection of this log-log plot shows that the slope is one, and hence we are observing the predicted linear performance of the LR algorithms.

At the bottom of Figure 6.1 we see that BC is indeed a very fast algorithm. Though the attributes in this synthetic dataset do not satisfy BC's conditional independence assumptions, as described in Section 6.1.3, the noiseless linear boundary helps BC make good predictions.

6.2.2 Number of Attributes

The AUC and time plots in Figure 6.3 compare classifier performance as the number of attributes increases from one thousand to one-hundred thousand. Again we see KNN scoring poorly and running slowly. This is mostly due to poor ball tree performance on this dataset. SVM RBF is similarly slow but achieves scores on par with the remaining classifiers. It is now easier to distinguish the LR classifiers from SVM LINEAR.

Perhaps the only important feature of these graphs is that the relative scores and speeds are consistent with the previous section. The nonlinearity of these graphs is likely due to the slight amount of coupling between the dataset attributes. This suggests that the effects of attribute correlation on performance increases with the

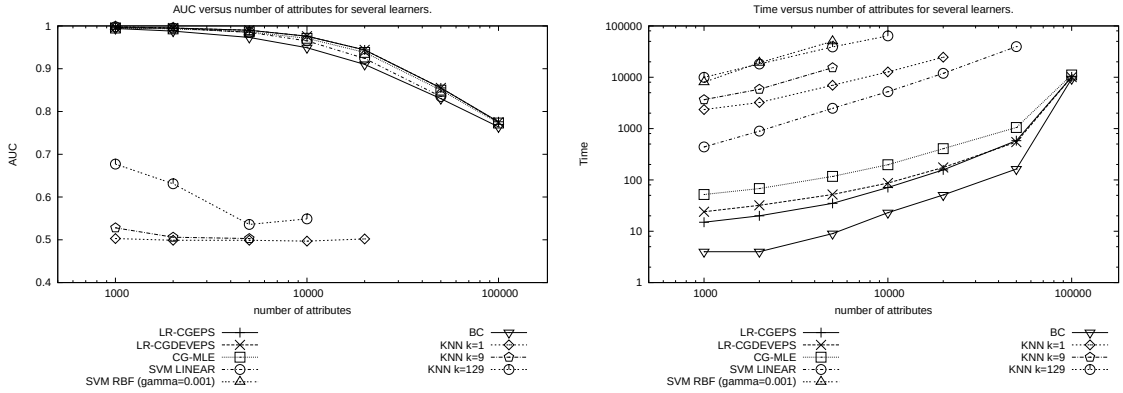


Figure 6.3: AUC and time versus number of attributes. Note that the vertical time axis in the right plot is logarithmic.

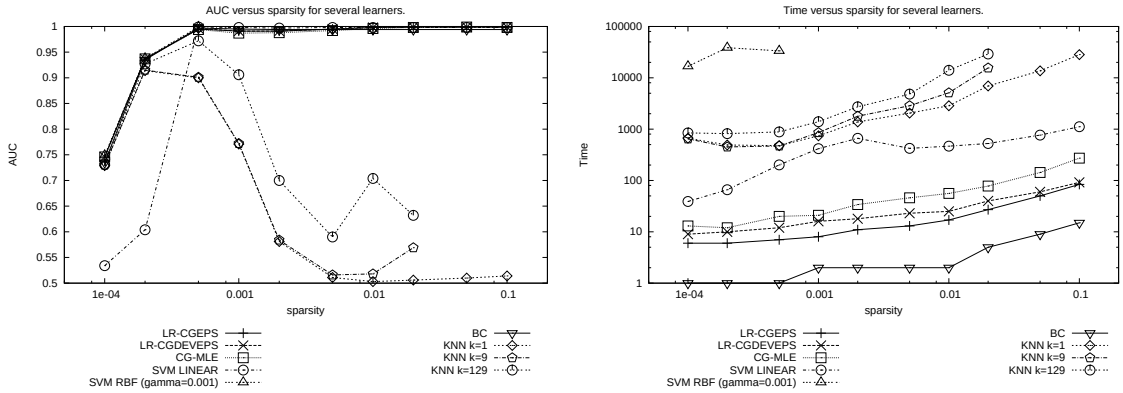


Figure 6.4: AUC and time versus sparsity. Note that the vertical time axis in the right plot is logarithmic.

number of attributes. This follows naturally from the hypothesis that extra iterations are spent overcoming cancellation between attributes. In Section 6.2.4, we'll see that changing the level of correlation has less impact on speed than changing the number of correlated attributes.

6.2.3 Sparsity

The sparsity plots in Figure 6.4 are quite jagged and irregular. The datasets have no coupling and half of the rows are positive, just as in the number of rows tests of Section 6.2.1. We are unable to explain why LR was able to find a reasonable linear classification boundary for the sparsest datasets, while SVM LINEAR was not. The erratic behavior of KNN also suggests that there is some interesting geometric structure in the data. If all of the algorithms behaved this way, we would blame the data generation procedure. However, SVM RBF, BC, and the LR classifiers all performed reasonably and similarly for the amount of data available in

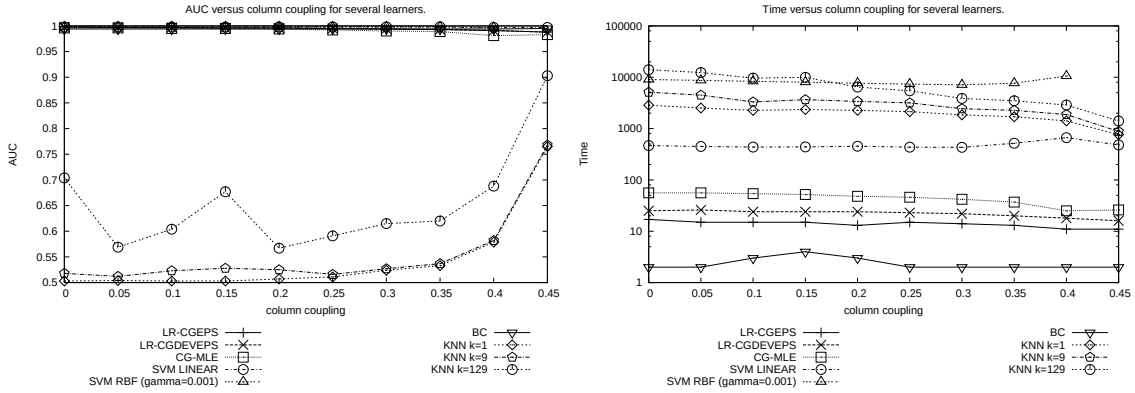


Figure 6.5: AUC and time versus coupling, as defined in Section 5.1.3.3. Note that the vertical time axis in the right plot is logarithmic.

these datasets.

The timings are also somewhat erratic, but the rank order of the classifiers is virtually identical to what we've seen already. The times for the faster algorithms, such as LR and SVM LINEAR, appear to increase linearly. The time required is not growing at the rate predicted by their worst case complexity. This suggests that the effects of sparsity, and hence the number of nonzero entries in the data, are less relevant to speed than the dimensions of the data.

6.2.4 Attribute Coupling

Figure 6.5 examines the impact of attribute coupling on classification performance. We finally see KNN performing somewhat better as the coupling parameter approaches 0.45. We also see the LR classifiers doing slightly worse as we approach this extreme level of coupling. One of the three datasets in Minka [27] deliberately introduced correlations among the attributes. The author found that Newton's method performed better than CG on this dataset.

Figure 6.6 is a magnification of the upper right corner of the AUC plot from Figure 6.5. We can see that LR performance is degrading with increased coupling, and CG-MLE degrades more than the two IRLS methods. Our results seem less extreme than Minka's, perhaps because we are measuring AUC instead of likelihood. Minka concludes that one should decorrelate data before running LR. Decorrelation may be unreasonable for many high-dimensional datasets, and it does not seem worthwhile if AUC is the focus.

The time graph in Figure 6.5, shown on the right in the figure, suggests that KNN and LR speed may increase as the level of coupling increases. The increase appears to be at most two-fold for LR and five- to ten-fold for KNN. The other algorithms seem virtually unaffected by the increase in coupling.

6.3 Real-world Datasets

We would like to compare performance of LR, SVM, KNN and BC on real-world datasets. Unfortunately, we do not have real-world datasets available other than those used for choosing the LR parameters in Chapter 5.

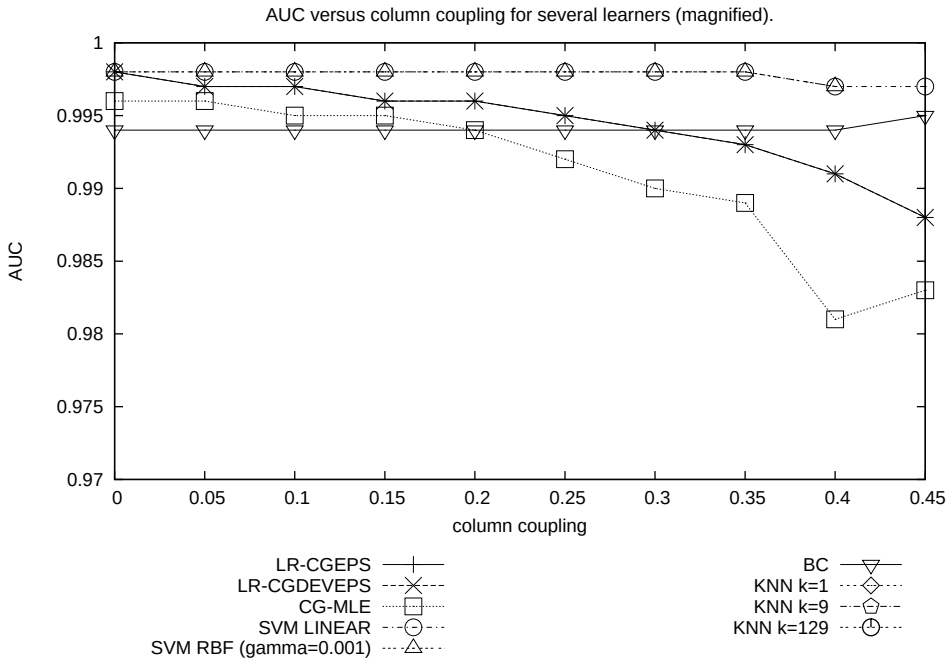


Figure 6.6: AUC versus coupling, magnified.

While we have done our best to tune SVM LINEAR, SVM RBF and KNN for these experiments, we cannot claim to have been as thorough as we were in Chapter 5. There is no tuning to be done for BC, since it has no parameters. We must be careful when comparing timings on these datasets.

In the previous section we saw that BC was the fastest algorithm, the LR classifiers were next, followed by SVM and NEWKNN. These results were for synthetic datasets, on which NEWKNN did very poorly. The time spent on nonlinear boundaries by NEWKNN and SVM RBF was certainly wasted, since the synthetic datasets are linearly separable. The datasets used in this section, which are described in Section 5.1.3, may or may not be linearly separable. These datasets are noisy, and the class distribution is heavily skewed with few positive rows.

Tables 6.1 and 6.2 show the AUC and timing results for our experiments. Looking first at the AUC scores, we see that the LR algorithms are performing comparably. SVM LINEAR usually scores a little lower than SVM RBF, except for `ds1.10pca` where it scores well below every classifier. Our assumption for the `ds1.10pca` experiment is that SVM^{light} is not adjusting properly for such a small dataset, and we are not sure how to compensate. We are tempted to conclude that the nonlinear classification boundaries used in SVM RBF worked better than the linear boundaries of SVM LINEAR. However, SVM RBF did not out-score the linear boundaries used by LR.

We were unable to build a useful ball tree for KNN within 4GB of memory for `imdb`, `citeseer` and `ds2`. Using the computers with more memory, creating a ball tree for `ds2` required over 4.6GB, and for `imdb` and `citeseer` required over 8GB. We allowed a `ds2` experiment to complete to determine whether we should adjust `rmin` just for this dataset so that the memory requirements were met. With k=9, KNN finished the

Table 6.1: Classifier performance on real-world datasets, 1 of 2.

	ds1		imdb		citeseer	
Classifier	Time	AUC	Time	AUC	Time	AUC
LR-CGEPS	86	0.949±0.009	303	0.983±0.007	43	0.946±0.021
LR-CGDEVEPS	59	0.948±0.009	320	0.983±0.007	67	0.945±0.021
CG-MLE	151	0.946±0.008	369	0.983±0.009	97	0.946±0.021
SVM LINEAR	188	0.918±0.012	565	0.938±0.016	87	0.810±0.049
SVM RBF	1850	0.924±0.012	6553	0.954±0.011	1456	0.854±0.045
KNN K=1	424	0.790±0.029	NA	NA±NA	NA	NA±NA
KNN K=9	782	0.909±0.016	NA	NA±NA	NA	NA±NA
KNN K=129	2381	0.938±0.010	NA	NA±NA	NA	NA±NA
BC	4	0.884±0.011	33	0.507±0.023	10	0.501±0.038

Table 6.2: Classifier performance on real-world datasets, 2 of 2.

	ds2		ds1.100pca		ds1.10pca	
Classifier	Time	AUC	Time	AUC	Time	AUC
LR-CGEPS	2983	0.720±0.030	44	0.918±0.011	8	0.846±0.013
LR-CGDEVEPS	1647	0.722±0.032	35	0.913±0.011	9	0.842±0.015
CG-MLE	3198	0.724±0.030	364	0.916±0.012	48	0.844±0.014
SVM LINEAR	2536	0.693±0.034	130	0.874±0.012	68	0.582±0.048
SVM RBF	67117	0.700±0.034	1036	0.897±0.010	490	0.856±0.017
KNN K=1	NA	NA±NA	74	0.785±0.024	9	0.753±0.028
KNN K=9	NA	NA±NA	166	0.894±0.016	14	0.859±0.019
KNN K=129	NA	NA±NA	819	0.938±0.010	89	0.909±0.013
BC	127	0.533±0.020	8	0.890±0.012	2	0.863±0.015

ten-fold cross-validation in approximately 5 hours, and scored around 0.654. This score might improve with k=129, but the other KNN results suggest it won't improve enough to make it competitive with the LR scores. Furthermore, the time would increase with k=129 or a smaller ball tree, further reducing the appeal of KNN relative to LR for this dataset. That we can run KNN on this dataset is very impressive. However, the focus of this thesis is LR and, in this context, it does not make sense to spend time on KNN for these datasets.

BC put in a miserable performance on these real datasets. We cannot blame this on tuning since BC has no parameters, and the best explanation is that these datasets do not satisfy the conditional independence assumption made by BC. Consider `imdb`, in which each row represents an entertainment production and the output represents the participation of voice actor Mel Blanc. Suppose a link contains animation director Chuck Jones and composer/conductor Carl Stalling. Chuck and Carl are conditionally independent if

$$P(\text{Carl}|\text{Mel},\text{Chuck}) = P(\text{Carl}|\text{Mel}) \tag{6.10}$$

This statement would not be true if Carl and Chuck worked together regardless of whether Mel did the voices. While we cannot comment on the veracity of this particular scenario, it seems likely that the link datasets have many attributes which are not conditionally independent given the output.

LR generally out-scored the other learners on these datasets, except on the PCA-compressed datasets. In several cases, SVM was not far below LR, and more comprehensive tuning could even the scores. KNN was much more competitive here than it was on the synthetic datasets, even beating LR on `ds1.100pca` and `ds1.10pca`. BC did poorly almost everywhere.

To draw final conclusions from these tables, we should examine the timings. Though BC was very fast, it's poor scores suggest it is not well-suited to this data. SVM RBF was very slow relative to SVM LINEAR and LR, and only scored as well as LR on `ds1.10pca`. Though SVM LINEAR was closer to the LR classifiers' speeds, it consistently scored well below LR. KNN was usually slow, but for the PCA datasets the time was not unreasonable and the scores were worthwhile. Our conclusion, on these datasets, is that the LR algorithms are the best choice. Furthermore, it seems that our LR-CGEPS and LR-CGDEVEPS perform better than our implementation of the traditional CG-MLE.

We conclude this section with some brief observations of the ROC curves for these experiments. We have organized these curves by dataset. Figures 6.7, 6.8 and 6.9 correspond to `ds1`, `imdb` and `citeseer`, while Figures 6.10, 6.11 and 6.12 correspond to `ds2`, `ds1.100pca` and `ds1.10pca`. Each figure has two graphs. The top graph has a linear "False positives" axis, while the bottom graph has a logarithmic "False positives" axis. Each plot is built from as many data points as there are rows in the corresponding dataset. For technical reasons these points are interpolated with cubic splines, and the splines are resampled where the bullets appear in the plots. One should not associate the bullets with the individual steps of the ROC curve.

The most interesting part of these figures is the bottom left corner of the logarithmic plot. It is here that we see how accurate the classifiers are for the first few predictions. In the `ds1` and `imdb` graphs, we see LR and SVM close together for their top five predictions, but SVM appears to have an edge over LR for initial accuracy. In the `citeseer` graph, SVM's initial performance is even stronger. In the long run, we know that LR outperforms SVM as indicated by the AUC scores, and careful observation of the graphs confirms this. The `ds1.100pca` and `ds1.10pca` linear ROC curves show KNN outperforming the other algorithms, though the initial segment does not show any clear leaders. The most important conclusion from the ROC curves is that the classifier with the best initial ranking performance might not be the most consistent classifier.

6.4 Conclusions

This chapter has demonstrated that LR is a capable classifier for high-dimensional datasets. The LR algorithms were more consistent than any other classifier tested. Though the SVM classifiers have excellent initial predictions on some datasets, their performance fell behind LR in every case. Our results suggest that the modified IRLS techniques we have introduced in this thesis are better performers than the traditional CG-MLE approach. It is difficult to choose between LR-CGEPS and LR-CGDEVEPS, though the latter performed exceptionally well on `ds2`.

It is reasonable to wonder whether the strong LR performance on the real-world datasets is due to having selected default LR values based on these same datasets. Unfortunately, we do not have additional real-world data available for further binary classification experiments. However, we have conducted some preliminary tests on the Reuters-21578 corpus, a dataset used in text classification experiments. These are multiclass experiments, unlike the two class experiments we have presented. The nature of the classification task requires significant and subjective preprocessing. Because of these issues, and because we do not have much experience in the text classification discipline, we have not included our early results. Our macro- and micro-averaged F1 scores were on par with the best we were able to produce using SVM LINEAR and SVM RBF, and were similar to scores reported by the SVM^{light} author on his version of the Reuters-21578 corpus [17]. Our LR implementation computed these results in less than one-third the time of linear SVM with

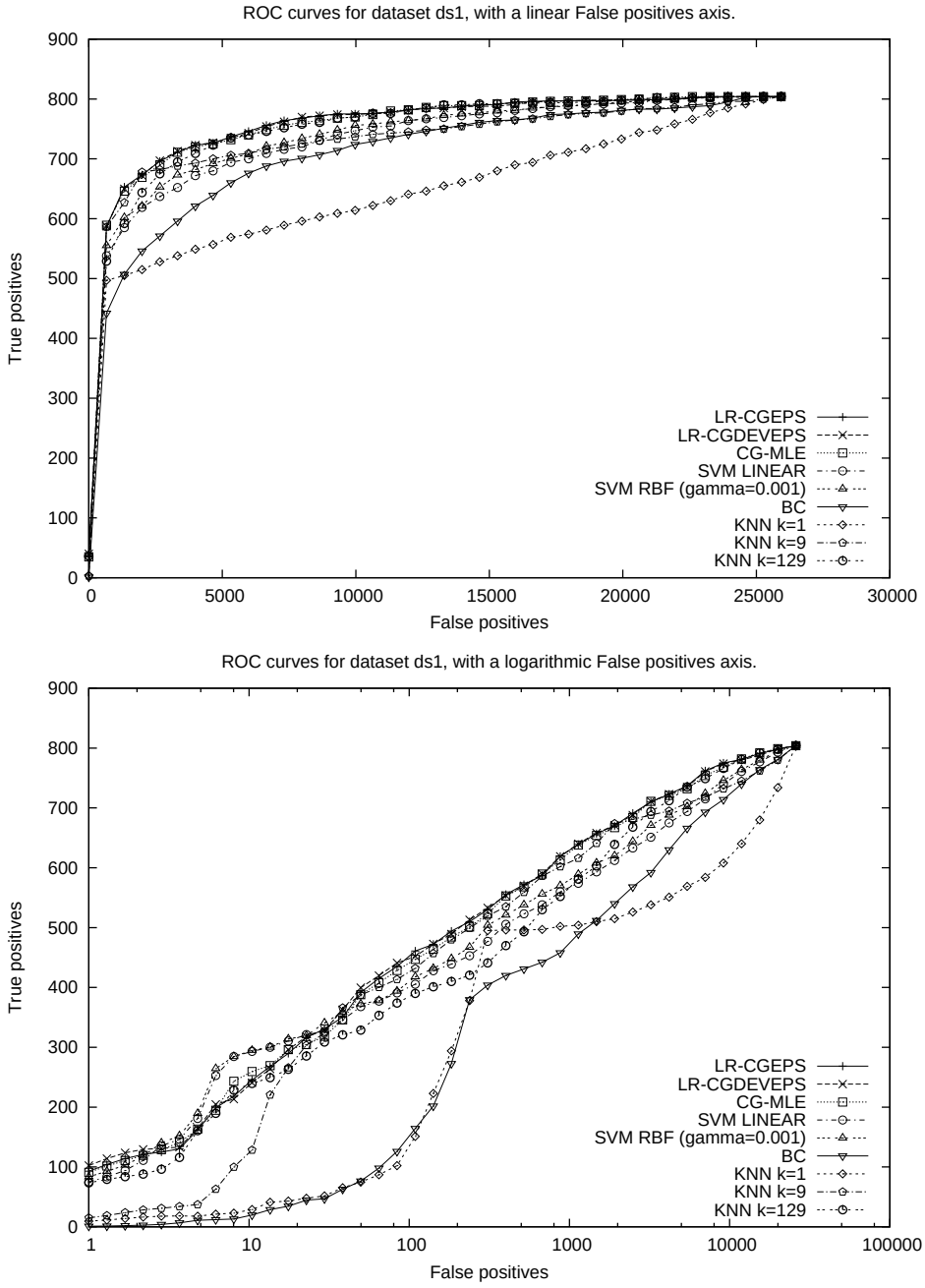


Figure 6.7: ROC curves for `ds1`. Note that the bottom graph has a logarithmic "False positives" axis.

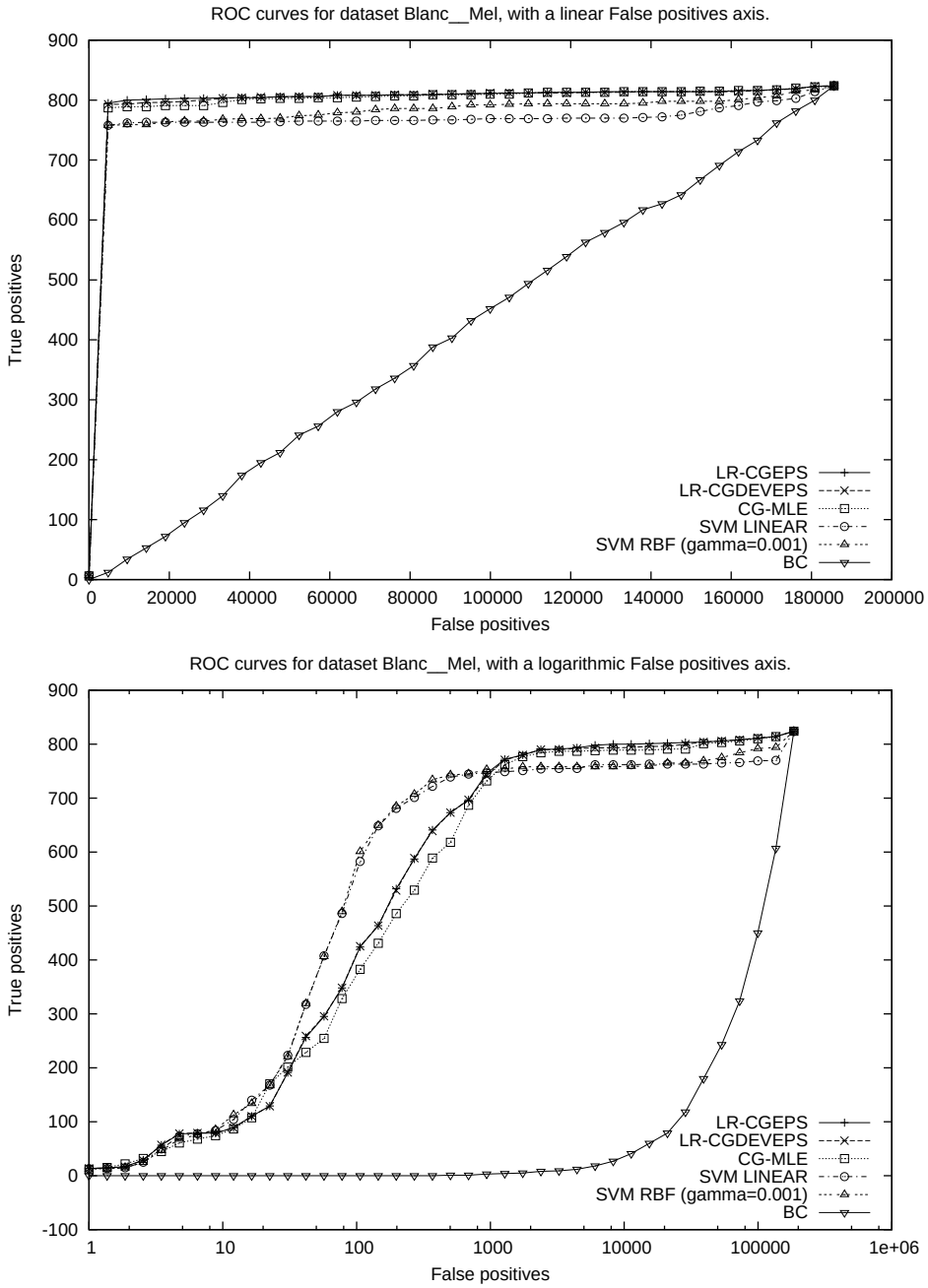


Figure 6.8: ROC curve for `imdb`. Note that the bottom graph has a logarithmic "False positives" axis. The ball trees used for the KNS2 algorithm exceeded 4GB of memory and were terminated.

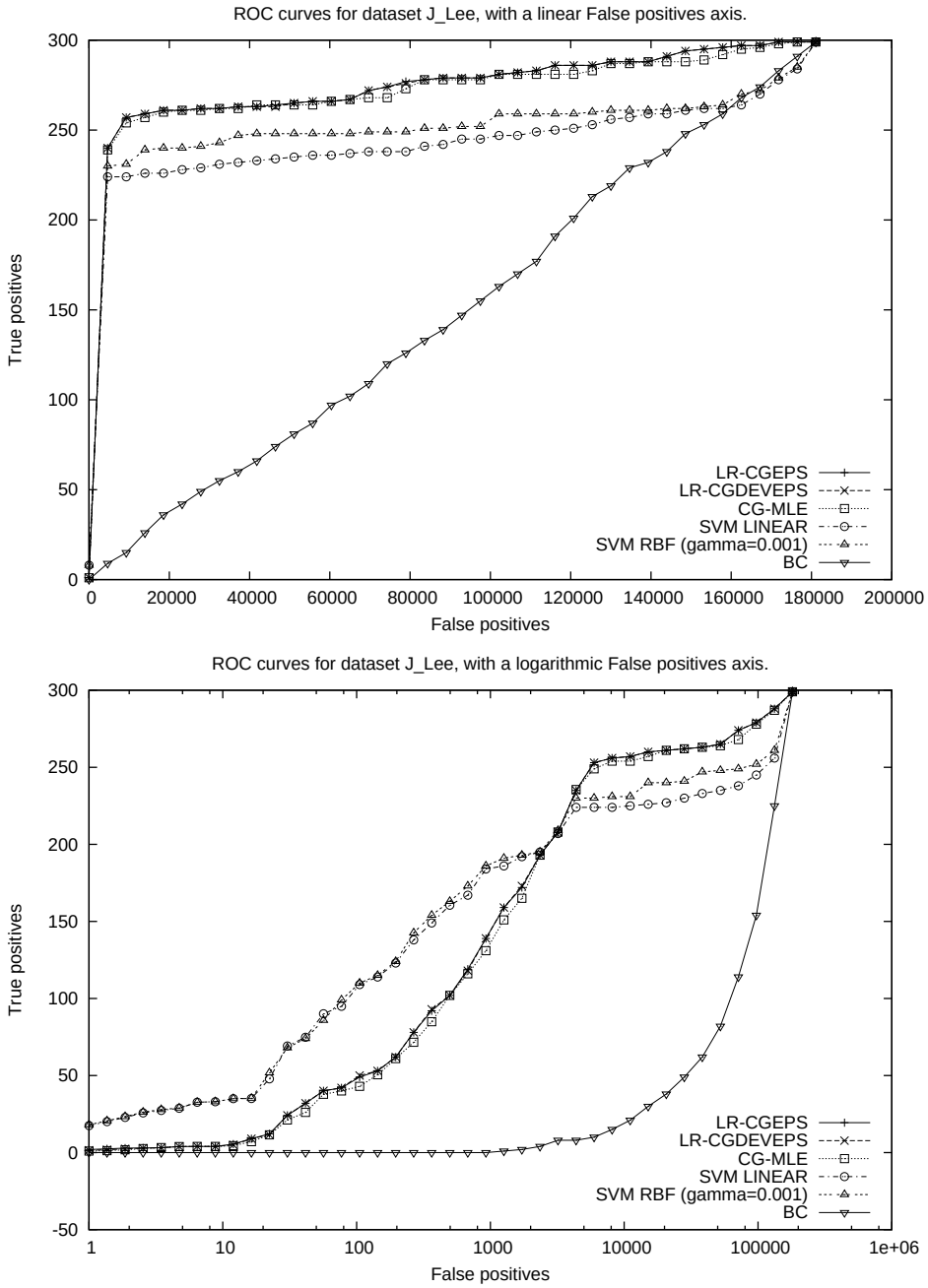


Figure 6.9: ROC curve for `citeseer`. Note that the bottom graph has a logarithmic "False positives" axis. The ball trees used for the KNS2 algorithm exceeded 4GB of memory and were terminated.

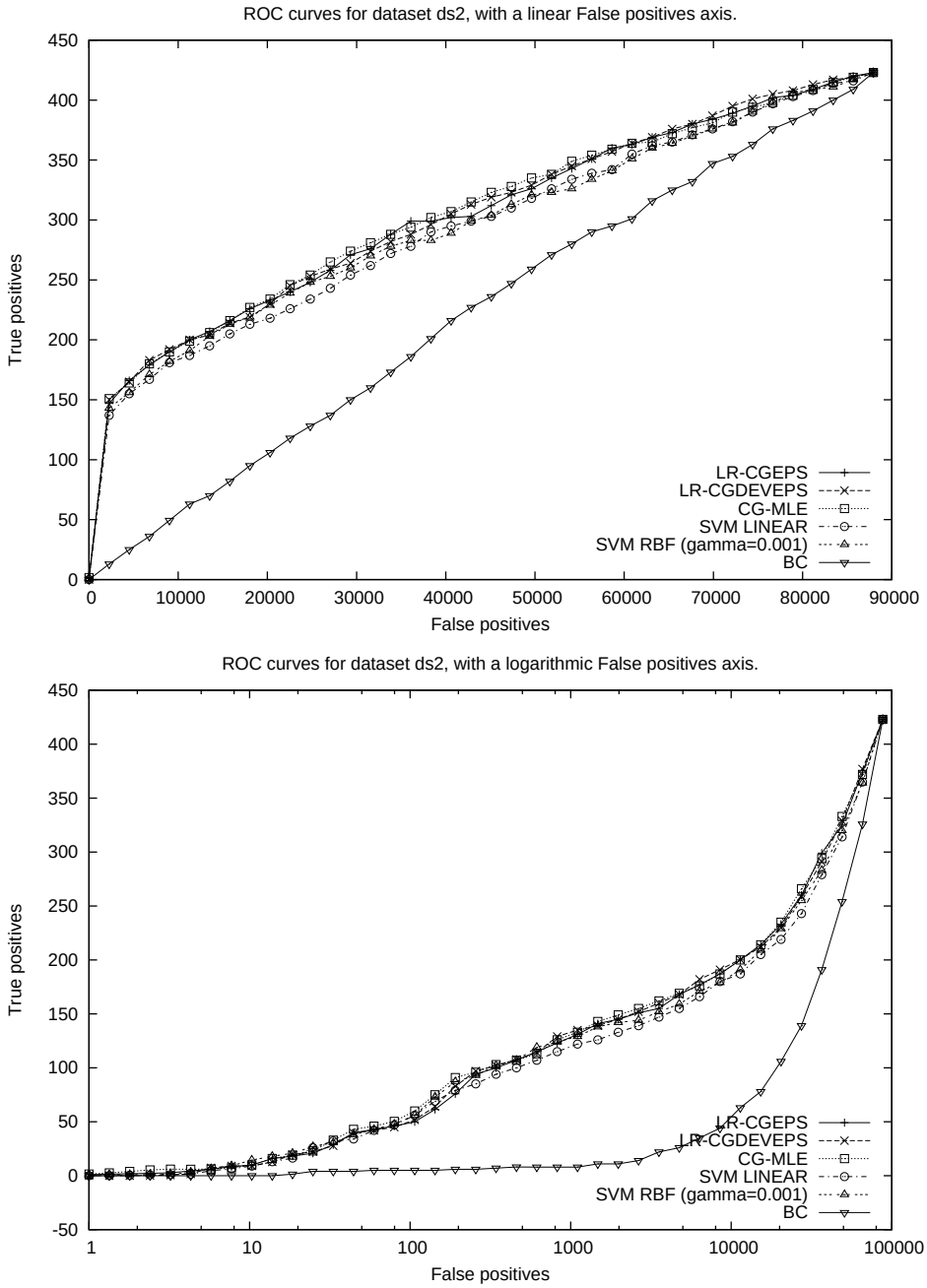
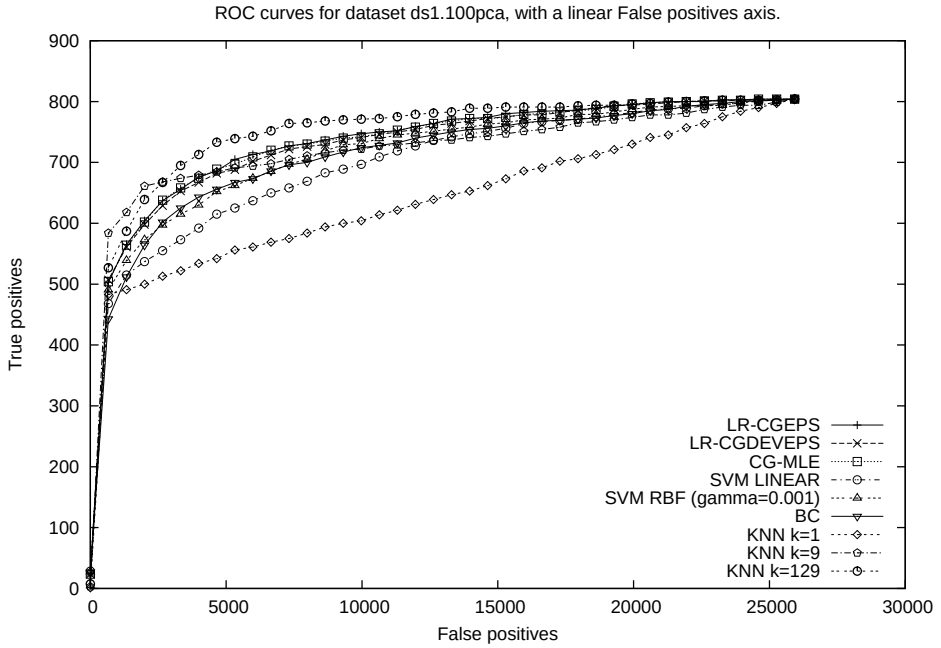


Figure 6.10: ROC curve for ds2. Note that the bottom graph has a logarithmic "False positives" axis. The ball trees used for the KNS2 algorithm exceeded 4GB of memory and were terminated.



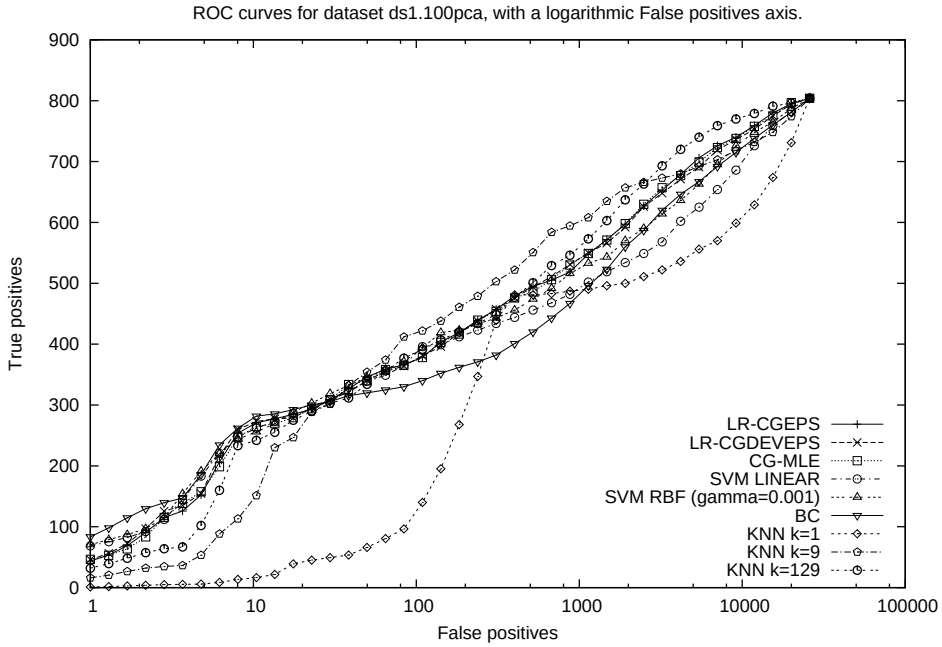


Figure 6.11: ROC curve for `ds1.100pca`. Note that the bottom graph has a logarithmic "False positives" axis.

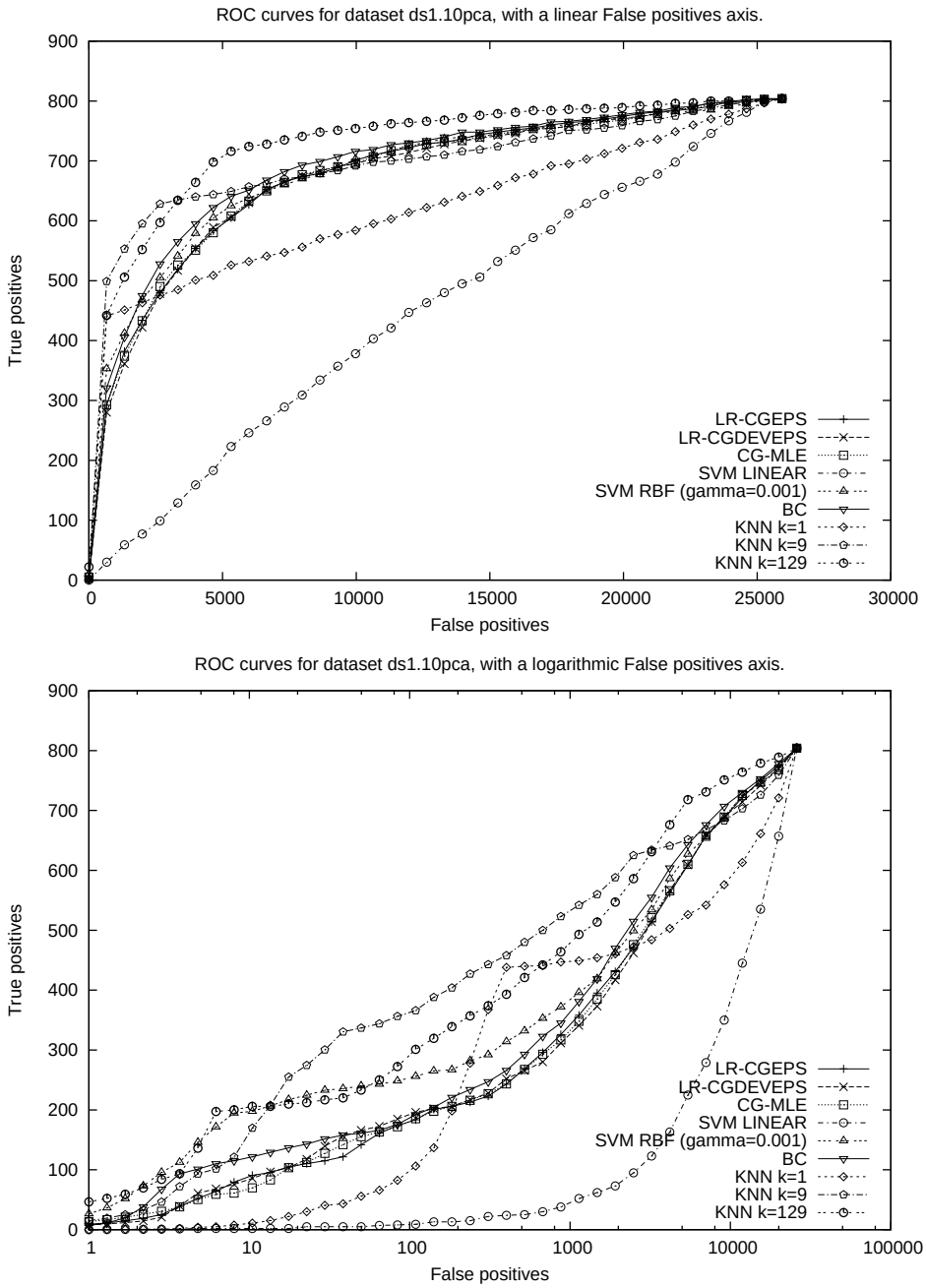


Figure 6.12: ROC curve for `ds1.10pca`. Note that the bottom graph has a logarithmic "False positives" axis.

SVM^{light}. This encourages us to believe that the results presented in this chapter reasonably represent the relative performance of SVMs and LR.

It is reasonable to ask why we are concerned with fast, autonomous classifiers for use in science and business. Since the underlying research that produces large datasets is often time-consuming, what does it matter if the classifier finishes in one minute or one day? There are several answers to this question. The first is that the size of datasets is likely to grow faster than we imagine. Existing datasets often have attributes which are in some way chosen by humans, and are limited by the number of sensors on the assembly line, or the manner in which a molecule is simplified using binary features. Adding new sensors, or binary molecule descriptors, is often cheap and easy. It increases the power of the classifying model, but due care must be taken to avoid overfitting. Adding more data records is expensive when humans are running the experiments, but roboticization is decreasing those costs. We expect that both the number of attributes and the number of records will increase rapidly as adequate analysis tools become available.

When datasets are large enough that the tools available take a nontrivial amount of time to run, it might be expensive to search for a good set of algorithm parameters. For this reason we endeavor to build self-tuning algorithms, for instance with proper scaling for `cgeps` and `cgdeveps`. The use of validation sets, mentioned in Chapter 10 below, may prove even more adaptive when sufficient data is available. Of course, when the algorithm is sufficiently fast we can run cross-validations with many choices of parameters in a reasonable amount of time. Though not addressed here, this should allow brute-force tuning systems.

For those not interested in cross-validations, tuning, and large datasets, speed may still be of interest. Finding a good subset of M attributes, as required for traditional interpretation of LR, can require a search over many possible models. Various exact and approximate approaches to pruning this model space have been created to reduce the amount of time required. With a fast version of LR, approximations can be improved and exact methods become more palatable.

Many large datasets can be represented with significantly fewer attributes than they possess. For instance, KNN performs nearly as well on the PCA-compressed datasets `ds1.100pca` as LR performs on original `ds1`. Why should we work on high-dimensional problems when many can be transformed to low-dimensional problems? Transformations such as PCA are not free. For instance, PCA requires at least a partial singular value decomposition, which is itself tricky to implement correctly and efficiently [44]. Once the problem is transformed into only a few dimensions, it is likely that nonlinear classification boundaries will be required for optimal performance. Algorithms capable of reliably finding nonlinear boundaries, such as KNN and SVM RBF, often require more computation than linear classifiers. Recall from Tables 6.1 and 6.2 that LR with `cgdeveps` ran faster on `ds1` than KNN or SVM RBF ran on `ds1.100pca`. Finally, the extra step required to project the raw dataset onto a small basis requires additional time and attention from the user. We conclude that using fast algorithms on the raw dataset is preferable to using dimensionality reduction techniques such as PCA.

Part IV

Conclusion

Chapter 7

Related Work

The earliest work we are aware of exploring the application of CG to LR is McIntosh [26]. In his thesis, McIntosh observes

> The increasing power and decreasing price of small computers, especially "personal computers", has made them increasingly popular in statistical analysis. The day may not be too far off when every statistician has on his or her desktop computing power on a par with the large mainframe computers of 15 or 20 years ago.

McIntosh goes on to extol the limited memory required by CG compared to QR algorithms. The author modified GLIM [6] statistics software to use CG, and explored its application to several generalized linear models including LR. It appears that McIntosh replaced IRLS entirely, using nonlinear CG to create what we describe as CG-MLE:

> ...using a version of GLIM in which the standard fitting algorithm had been replaced by the Hestenes-Stiefel algorithm. ... Since fitting generalized linear models involves the minimization of a function that is not necessarily quadratic, the algorithms used were not exactly algorithms A.1 and A.3. The only difference is that the gradient $g(\beta)$ is computed directly rather than recursively.

McIntosh's applications were traditional statistics problems with carefully chosen attributes and a few tens of data points. He comments on several numerical issues. This is the most detailed and comprehensive examination of CG-MLE and linear models to date.

The use of CG to directly maximize the LR likelihood has been reported more recently in several papers. Minka [27] compares the effectiveness of several numerical methods for maximizing the LR likelihood in a short technical report. He reports FLOP counts on three small, dense synthetic datasets. The author finds that CG is often very effective, but also states that Newton-Raphson may be more suitable for correlated data. Our approach to ridge regression for CG-MLE is similar to Minka's. This technical report describes traditional IRLS, coordinate-wise Newton-Raphson, nonlinear CG, annealed coordinate-wise MAP on the dual of the LR likelihood, the quasi-Newton Böhning's method with back-substitution, and two versions of iterative scaling. Implementation details are not discussed, except to say that Matlab was used.

Zhang et al. [48] describes a modified LR algorithm for use in text classification. Their approach yields an iterative approximation to an SVM. The authors use CG to find the MLE, and state that in their experiments the Hestenes-Stiefel direction updates for nonlinear CG were more efficient than Polak-Ribiére or

Fletcher-Reeves. The authors claim that their algorithm is more efficient than SVM^{light} on "large-scale text categorization collections".

In the text categorization, information retrieval, document filtering and document routing literatures, many papers apply LR in some form. Besides the previously mentioned work by Zhang et al. [48], there is Zhang and Oles [49]. The authors discuss the reputation of LR as slow and numerically unstable in text categorization, referring to several papers. They suggest that the failures reported in Schutze et al. [40] could be the result of that paper's lack of regularization in general, and

> [a]nother reason could be their choice of the Newton-Raphson method of numerical optimization, which in our experience could become unstable, especially without regularization.

The authors use the same regularization techniques that we have already discussed, related to ridge regression. They develop a common framework for describing several linear classifiers including linear regression, LR, and SVMs. They create specialized solvers for each of these using their framework and a modified Gauss-Seidel method. Intriguingly, they make an aside comment that

> instead of Algorithm 1 [modified Gauss-Seidel], one may also apply Newton's method directly to (12) [the general framework for linear classifiers], as suggested in Hastie and Tibshirani (1990) and used in Schütze et al. (1995). The resulting linear system from Newton's method can be solved by an iterative solver such as the Gauss-Seidel iteration or CG.

which makes one wonder whether they are referring to something like our IRLS and CG combination. However, they continue

> However, a properly implemented line search method, which can be complicated, is required to guarantee convergence. Such a line search method can also result in small step sizes, which slows down the convergence.

Thus they do not appear to be suggesting solving the weighted least square subproblem of IRLS with CG, since the application of CG to any linear system does *not* require a line search or small step sizes. Among their conclusions is that a properly regularized implementation of LR is competitive with SVMs for classification of documents, though they remark it is somewhat slower. This statement runs contrary to the results of this thesis, suggesting that they did not anticipate or properly evaluate the combination of IRLS and CG.

We are not aware of any other literature closely related to our work. Hastie et al. [10] comments that "Logistic regression models are used mostly as a data analysis and inference tool, where the goal is to understand the role of the input variables in *explaining* the outcome." Our work in applying LR to data mining and high-dimensional classification problems was first discussed in Komarek and Moore [20]. We have made some initial text classification experiments, similar to those of Yang and Liu [47]. These were promising in that our scores were competitive with those of SVM^{light}, which is often considered state-of-the-art for text classification; and further our IRLS implementation ran nearly four times faster than SVM^{light}. Our IRLS implementation participated in a short survey of approaches for solving the link classification problem described in Section 5.1.3.1. This paper, Kubica et al. [22], demonstrated that LR could be reasonably applied to large multiclass problems requiring as many models as there were independent variables.

Chapter 8

Conclusions

In this thesis we analyzed two approaches to fitting LR models for high-dimensional data for classification and data mining. The data was restricted to sparse binary or dense real-valued attributes, and binary outputs. The first approach used an iteratively re-weighted least squares algorithm, accelerated by CG, and regularized through an application of ridge regression. We believe the result is a novel LR fitting procedure, and we proposed two variants with different termination criteria.

The second approach was the traditional likelihood maximization with CG, regularized through a typical application of ridge regression. This combination is not novel, but we believe we are the first to make a large, careful empirical study of it.

For both approaches we discussed many possible modifications. These ideas were explored in a thorough, empirical setting. We eliminated all but a small set of parameterized modifications which enhanced stability, optimality and speed. For each of the chosen modifications a default value was chosen via analysis of a large collection of experiments. The values chosen offered robust performance under a wide variety of conditions and removed any need for tuning.

Our three LR variants were tested on synthetic datasets to establish the relation between performance and the number of dataset rows, the number of dataset attributes, the sparsity of the dataset, and the level of interdependence among the attributes. The same experiments were conducted on linear and radial basis function Support Vector Machines, an accelerated version of k-nearest neighbor with two different values of k, and a fast implementation of Bayes' classifier. The LR algorithms had predictive behavior similar to the other classifiers, with the exception of k-nearest neighbor which performed poorly on these synthetic datasets. The LR algorithms were faster than the Support Vector Machines and the accelerated k-nearest neighbor algorithm.

The set of eight classifiers just described were tested on six real-world datasets from life sciences data mining and link analysis applications. These datasets had up to one million attributes and up to one-hundred thousand rows. The sparsity varied from one-hundred percent down to two-thousandths of one percent, and the total number of nonzero dataset entries varied from thirty million to one-quarter million. The output class distribution was highly skewed. The tests were ten-fold cross-validations, scored using the "area under curve" (AUC) metric to assess ranking ability. For all but the smallest datasets, the LR algorithms had AUC scores as high or higher than the competing algorithms. The LR algorithms also ran faster than all of the other algorithms, with the exception of Bayes' classifier which was rendered unusable by poor AUC scores.

We have demonstrated that LR is a capable and fast tool for data mining and high-dimensional classification problems. Our novel IRLS fitting procedures outperformed the traditional combination of CG and

LR maximum likelihood estimation, as well as state-of-the-art techniques such as Support Vector Machines. This superior performance was demonstrated on many synthetic and real-world datasets.

Chapter 9

Contributions

Throughout this thesis we have tried to note our contributions. This chapter is a short summary of all the contributions we mentioned, or should have mentioned. Our major contributions include

- Novel LR fitting procedure which uses CG to find an approximate solution the IRLS weighted linear least squares subproblem. This improves speed and handles data with linear dependencies.
- Demonstration of LR's suitability for data mining and classification applications on high-dimensional datasets.
- Publically available high-performance implementation of our IRLS algorithm, as well as the traditional CG-MLE. This software includes the CG implementation we wrote to support our LR software. Our CG implementation appears to be faster than the GNU Scientific Library's implementation.
- In-depth survey of IRLS and CG-MLE modifications, along with comprehensive empirical results.

In the course of our work, we also introduced a modification to the Cholesky decomposition to overcome linear dependency problems, described in Komarek and Moore [20]. However, LR methods using this decomposition were inferior to our final approach.

Chapter 10

Future Work

In this thesis, we have shown promising results for LR as a data mining tool and high-dimensional classifier. There is, of course, more research to be done. We list below several potential research topics which we believe may improve performance or broaden LR's appeal. This list includes all potential research topics mentioned previously in this thesis.

- We stated in Chapter 2 that CG performance depends on the condition number of the Hessian [41]. A general technique called *preconditioning* can be used to improve (reduce) the condition number of a matrix. This technique is popular in the CG literature [26; 41; 7], possibly because the preconditioning can be done using only matrix-vector operations. Preconditioning should be explored for our implementations.

- We discussed several CG, IRLS and CG-MLE termination techniques. One we did not discuss, but which is appropriate when data is plentiful, is the use of *validation sets* [10]. In this approach to termination, part of the training data is held out to be used for approximation of prediction performance. When this estimate of the prediction performance is deemed adequate, training is terminated.

- Though CG is designed for positive definite matrices, it has performed well on a variety of ill-conditioned data matrices arising from datasets such as `ds1` with linearly dependent or duplicate columns. A version of CG known as *biconjugate gradient* is designed for positive semidefinite matrices, and this might further accelerate IRLS [7]. Other iterative methods for linear equations, such as *MINRES*, might also be explored.

- Truncated-Newton methods should be investigated for finding the zeros of the LR score equations, and the result compared to our combination of IRLS and CG.

- We concluded in the summary for CG-MLE computations, Section 5.3.5, that the Fletcher-Reeves direction update may be competitive with the modified Polak-Ribiére direction update when our other regularization and failsafe checks are used. This should be an easy area to explore. Because the modified Polak-Ribiére direction update includes a Powell restart, one might wish to implement separate Powell restarts or an alternative when using Fletcher-Reeves.

- The CG residual is a cheap way to terminate CG iterations in IRLS. Using the deviance requires significantly more computation but keeps our focus on the LR likelihood. An easily-computed approximation

to the deviance, or other related statistic, could improve performance for IRLS with `cgdeveps` as well as CG-MLE. One possible replacement for the deviance is the Pearson χ^2 statistic $\sum_{i=1}^{N} \frac{(y_i-\mu_i)^2}{\mu_i(1-\mu_i)}$ [30].

- Regularization is an important part of LR performance. We have suggested in this thesis that a single Ridge Regression parameter λ can be used across a wide variety of datasets. Furthermore we apply the same value of λ to all slope coefficients in the LR model. In localized Ridge Regression the penalization parameter becomes a penalization vector such that the penalty function is $\beta^T \mathrm{diag}(\lambda)\beta$. Other penalty functions may also prove useful.

- We terminate CG iterations when more than `cgwindow` iterations fail to improve the residual (`cgeps`) or deviance (`cgdeveps`). One alternative is to halve the step size when iterations fail to make an improvement [10]. This technique is sometimes called *fractional increments* [30].

- We have ignored feature selection because of our focus on autonomous high-dimensional classification. However, model selection is very important when one intends to make a careful interpretation of the LR model coefficients. Iterative model selection is the process of searching through the space of possible models to find the "best" one. Because there are 2^M possible models for a dataset with M attributes, it is not generally possible to estimate $\hat{\beta}$ for all of them. Instead the search is directed by local improvements in model quality. One common iterative model selection method is *stepwise logistic regression* [13]. Such techniques need to fit many LR models, and may benefit from appropriate application of our fast implementations.

- We have described IRLS in the context of LR, with the result that we have equated IRLS and Newton-Raphson. For generalized linear models this is not always true [25]. Our modifications to IRLS apply in the general case, and can be used with other generalized linear models. This has not been explored yet.

- Text classification is an interesting and active research area. Many classifiers are used, and SVMs are often considered the state-of-the-art. LR has a troubled history in text classification, as well as a promising future [40; 49; 48]. Even when classifiers capable of handling high-dimensional inputs are used, many authors apply feature selection to reduce the number of attributes. Joachims [15] has argued that feature selection may not be necessary, and may hurt performance. We believe our implementation of LR is suitable for text classification, and could be competitive with the state-of-the-art techniques.

Appendix A

Acknowledgements

To finish a thesis, and a doctoral program, requires the cooperation of many people. I am indebted to the staff who have taken care of my academic details while I have been a student, both at Carnegie Mellon University and Western Washington University. I have only reached my goals through the assistance of many dedicated faculty, including my undergraduate advisor Thomas Read and graduate advisor Andrew Moore.

My education was made possible by the generosity of several public and private institutions. The list includes the State of Washington, the National Science Foundation, the Haskell Corporation, and many others.

Nobody has sacrificed more during my degree, or been more patient with my progress, than my wife Jeanie. She proofread two pounds of thesis-laden paper, hoping I'd finish sooner that way. I would also like to thank my parents, who always stressed that my education came before all other endeavors. Though I deliberately misinterpreted that statement when I had chores to do, things worked out well-enough in the end.

Appendix B

Software Documentation

The Auton Lab [1] has makes much of its research software available from its website `http://www.autonlab.org`. The implementation of LR described in this thesis, as well as the BC and KNN implementations and a wrapper around SVM^{light}, are available as part of the *Auton Fast Classifiers* project. Documentation [19] for a research version of our code is included in this appendix as an reference for our software's capabilities. The distributed version of this software includes a graphical interface and a modified version of the documentation included in this appendix.

B.1 Introduction

We assume that the AUTON learning algorithms program has been built, and that the executable is named `./afc`. All commands in this document are indented as shown below.

```
./afc action args
```

Names of arguments are written as *argname*, while actual argument values are written as `argval`. The example command above shows the canonical form for running the `./afc` executable. The first argument must be an *action*, as described in Section B.2, followed by a list of arguments. The argument list is keyword-based, meaning that the order of the arguments is unimportant. However, some arguments consist of a keyword and a value, with the value occurring immediately after the keyword. A "dataset" is a file containing data one wishes to analyze, or the filename of such a file. The set of keywords which affect the action depends not only upon the action, but also on other keywords. The following sections will describe the actions and arguments in more detail.

B.2 Actions

An action represents several things. At the highest level it represents a task. However, it also represents a keyword in the command line and usually a single source file for this program. More information on the available actions may be found in the next two subsections.

B.2.1 Action Summary

Name	Description
convert	Converts between dataset formats.
loglip	Reduces dimension of dataset, similar to PCA.
mroc	Similar to **roc**, but operates on multiple output columns.
outputs	Prints output values from dataset.
predict	Loads a trained learner from a file and computes predictions on a dataset – see **train**.
roc	Train/test or k-fold cross-validation with output suitable for ROC curves and AUC calculation.
spardat	Loads a sparse file (SVM^{light} format) and prints some summary information. This action can also create a new sparse dataset with random subset of attributes from the input dataset.
train	Trains a learner on a dataset and stores the trained learner in a file – see **predict**.

B.2.2 Action Details

In some actions one must specify one or more datasets, or one or more learning algorithms. The syntax for specifying datasets and learners is detailed in Sections B.3 and B.4, respectively. No matter what action is chosen, one may always add the `arghelp` or `verbosity` keywords to the command-line. The `arghelp` keyword causes the command-line parsing functions to print information about the keywords they are checking

for, including the default value for each keyword's argument. The `verbosity` keyword takes an integer value to control the printed output. The default value for `verbosity` is zero, and increasing this value increases the volume of output. Note that options which take no value may not appear in the output from `arghelp`.

B.2.2.1 convert

This action converts one type of dataset into another. See Section B.3 for more information about dataset formats and naming conventions. From the command-line help:

> Usage: `./afc` convert in <input_name> out <output_name> [subsize <int>]
>
> Type of input and output file is guessed from the filenames.
>
> | | | |
> |---|---|---|
> | .csv, .fds | => | dense dataset with names |
> | .ssv | => | 'June' format |
> | .hb | => | Harwell-Boeing sparse format |
> | otherwise | => | spardat format |
>
> If the output filename is foo.ssv, then two files are created, named foo.factors.ssv and foo.activations.ssv.
>
> If the output filename ends in .ssv or .fds, the output will be stored in the last column. To load data from a dense file without translation to binary values, use the csv keyword.
>
> Harwell-Boeing files are of HB type RRA with one right-hand-side of type F. Most of the HB conversion code is copyright of the US National Institute of Standards and Technology. See this program's supporting documentation or source code for details about the copyright and license for the NIST HB code.
>
> If subsize <int> is specified, the output dataset will contain a random subset of input attributes. The output attribute will be preserved.

Most of the source code for reading and writing files in the Harwell-Boeing sparse format comes from the NIST Harwell-Boeing File I/O in C code. Below is the copyright notice and license for this body of code.

```
Fri Aug 15 16:29:47 EDT 1997

                        Harwell-Boeing File I/O in C
                                 V. 1.0

            National Institute of Standards and Technology, MD.
                                K.A. Remington

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 NOTICE

 Permission to use, copy, modify, and distribute this software and
 its documentation for any purpose and without fee is hereby granted
 provided that the above copyright notice appear in all copies and
 that both the copyright notice and this permission notice appear in
```

```
 supporting documentation.

 Neither the Author nor the Institution (National Institute of Standards
 and Technology) make any representations about the suitability of this
 software for any purpose. This software is provided ''as is'' without
 expressed or implied warranty.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Mandatory keywords and arguments:

- `in` *dataset*: Specify the input dataset. Section B.3 describes data file formats and naming conventions.
- `out` *dataset*: Specify the output (converted) dataset. Section B.3 describes data file formats and naming conventions.

Common keywords and arguments:

- `subsize` *int*: Specify the number of attributes, chosen randomly, which will appear in the output file. The original output attribute will always appear in the output file, and is not counted as one of the random attributes.

B.2.2.2 loglip

This action projects the data from a dataset into a lower-dimensional space. This is useful if the original data is too high-dimensional. The projection method performs the same function as PCA, namely to transform high dimensional data into low dimensional data, but it does so much more quickly.

You will find a new file is created, which contains dense data with *num_comps* input attributes and the original untransformed output attribute.

Mandatory keywords and arguments:

- `in` *dataset*: Specify the input dataset. Section B.3 describes data file formats and naming conventions.

Common keywords and arguments:

- `out` *dataset*: Specify the destination file to which the projections will be saved. Default is out.csv. Output file will be a dense dataset, as defined in Section B.3.
- `num_comps` *int*: How many components do you wish to project down onto? Default is 10.

B.2.2.3 mroc

This action name is an abbreviation for multiple ROC. See the **roc** action description for details about the **roc** action. "Multiple ROC" may mean one of several things depending on an additional mandatory keyword:

`multipredict`: Automation of multiple ROC experiments.

`multiclass`: Greater-than-binary classification.

`multilc` "Link Completion" – each column from the data is used as the output, one at a time, with the remaining columns used as the inputs.

Each of these multiple ROC types are described in turn below.

B.2.2.3.1 `multipredict` For a `multipredict` experiment, the data must be in the *June* format. The *June* name should include an output attribute name, as usual for that format. However, the attribute name specified is ignored since all output attributes will be used.

For each output a ROC experiment is performed and the results printed to the screen. The normal ROC keywords `pout`, `fout` and `rocname` are supported. Because there are multiple ROC experiments, filenames specified with these options will have the output attribute name appended, separated by "–". The `frocname` is not current supported, but can be added if requested. It is more efficient to run a `multipredict` experiment than to run several regular ROC experiments on the same training data.

The output from the `pout` files can be used to compute the micro- and macro-averaged precision, recall and F1 scores used in text classification experiments. Scripts are available on request which simplify this process.

B.2.2.3.2 `multiclass` A regular ROC experiment is a binary classification problem, with the goal of ranking test data points in order of the probability they belong to the positive class. A `multiclass` experiment extends the number of available classes beyond the two classes of a regular ROC experiment. The goal is not typically to order data points by likelihood of belonging to a particular class, but to find the class with highest likelihood for the data point. This can be inferred from the output of a `multipredict` experiment, but `multiclass` is more efficient.

For a `multiclass` experiment, the data must be in the *June* format. The *June* name should include an output attribute name, as usual for that format. However, the attribute name specified is ignored since all output attributes will be used.

The normal ROC keywords `pout` and `fout` are supported, but `rocname` and `frocname` is not. Each prediction value in the `pout` file is the highest prediction value observed across all output attributes for the corresponding data point. Also available is the `cout` option, which allows specification of a filename into which the output attribute with highest probability is written as an integer. The integer represents the predominant attribute's index in the output dataset, starting from zero. Without the `cout` file, the other output files are difficult to interpret.

B.2.2.3.3 `multilc` A `multilc` experiment automates the task of predicting every input attribute using every other input attribute. Suppose we are given a good dataset in which every data point has all attributes correctly recorded, and a bad dataset in which someone has removed one attribute from each data point. For each row in the bad dataset, a `multilc` experiment ranks all attributes according to their likelihood of having been erased from that row. All attributes still in the bad dataset row are excluded from this ranking.

The input dataset may be specified in either the *June* or *spardat* format, and the output value or values are ignored. Only train/test experiments are allowed, and hence `testset` specification is mandatory. All `multilc` experiments must specify an output file with the `lcout` keyword. The format of this file is described in the keyword lists, below. The output file keywords `cout`, `fout`, `pout` are not allowed, and `rocname` and `frocname` have unspecified meaning.

A precise definition of `multilc` is as follows. Let superscripts indicate attribute indices and subscripts indicate row indices in a binary dataset, and treat the dataset rows as sets of nonzero attributes. Let $A = \{a^1, \ldots, a^M\}$ be the set of M attributes of the dataset $X = \{x_1, \ldots, x_R\}$ and a testing set $\widehat{X} = \{\hat{x}_1, \ldots, \hat{x}_{R_{\hat{\text{test}}}}\}$. For each attribute $a^k \in A$ execute the following steps:

1. Create a new training dataset X^k by removing attribute a^k. That is, $X^k = \left\{x_i^k \mid x_i^k = x_i \setminus \{a^k\}\right\}$.

2. Create an output vector $Y^k = [y_i^k]$ using the presence or absence of a^k in the original dataset row x_i. That is, $y_i^k = 1$ if $a^k \in x_i$ and $y_i^k = 0$ otherwise.

3. Train the learner on $\{X^k, Y^k\}$.

4. Predict $E(\hat{y}_i^k | \hat{x}_i^k)$ for each row i in the testing dataset. Note if a^k is in the test data point, then it does not make sense to compute this expectation for `multilc`, and therefore the expectation is set to zero.

If storing the rank of all attributes for every test data point is undesirable, the `lcheap` parameter may be used to choose how many of the top attributes to remember for each row. If `lcheap` is not set or is non-positive, ranks are stored for every attribute for every row, and are written to the `lcout` file in the same order as the attributes are indexed. If `lcheap` is positive, only the top `lcout` attributes are stored for each row and the `lcout` file will contain (index prediction) pairs for each of the top `lcout` attributes for each row.

Mandatory keywords and arguments:

- `cout` *filename*: This option is required for `multiclass` experiments, and is not appropriate for `multipredict` or `multilc` experiments. It specifies a file into which the index of the output attribute with highest probability is written. For a train/test experiment these are the classes o test dataset records. For a k-fold cross-validation experiment these are the classes of the records of the training dataset, as computed for each record when the record is held-out for prediction.

- `in` *dataset*: Specify the training dataset. Section B.3 describes data file formats and naming conventions.

- `multipredict` or `multiclass` or `multilc`: This keyword further specifies the experiment type. For details, please read the above documentation about these keywords.

- `lcoutname` *filename*: This keyword is only mandatory for `multilc` experiments, and is ignored otherwise. The format of the file depends on the rarely-used `lcheap` parameter. See the description for `lcheap`, below, for further information about the contents of this file.

- `learner` *learner_specification*: Specify the learner. Section B.4 describes the available learners.

- `testset` *dataset* or `num_folds` *int*: Use `testset` to specify the testing dataset in a train/test experiment. Section B.3 describes data file formats and naming conventions. Use `num_folds` to specify the number of folds in a k-fold cross-validation. `testset` and `num_folds` are mutually exclusive. For `multilc` experiments only `testset` may be specified.

Common keywords and arguments:

- `fout` *filename*: Specify a file into which information about the folds of a k-fold cross-validation experiment are written. One line is written for each row in the training dataset. Each line has one integer which indicates in which fold the corresponding dataset row was "held out". This keyword is only valid for k-fold cross-validation experiments.

- `pout` *filename*: Specify a file into which the learner's predictions are written. For a train/test experiment these are the predictions on the test dataset. For a k-fold cross-validation experiment these are the combined predictions on the training dataset. For a `multipredict` experiment, multiple files are created with names *filename–OutputArgIndex*. For a `multiclass` experiment only one `pout` file is created – see the description above for more information.

- `rocname` *filename*: Specify a file into which the coordinate pairs for the ROC curve are written. The file may have additional comments before or after the ROC data, denoted by an initial "#" symbol. Comments before the ROC data usually list some details about the experiment which produced the ROC data. Comments after the ROC data are usually AUC and related values for the experiment.

Rare keywords and arguments:

- `lcheap` *int*: This keyword only applies to `multilc` experiments. This allows control over how many outputs are stored in memory and written into the `lcout` file. If `lcout` is not specified or is non-positive, all attributes' predictions are written into the `lcout` file in the same order as the attributes appear in the input file. If `lcout` is positive, only the predictions for the top `lcout` attributes are written into the `lcout` file. The attribute indices are written into the `lcout` file as well, appearing before the prediction value and separated from it by a single space. In both cases the predictions are separated from one another by commas.

B.2.2.4 outputs

This action prints the outputs of a dataset.

Mandatory keywords and arguments:

- `in` *dataset*: Specify the training dataset. Section B.3 describes data file formats and naming conventions.

B.2.2.5 predict

This action loads a trained learner from a file and computes predictions on a dataset. The saved learner file may be generated by the **train** action. The AUC score, as described in the **roc** action description, is reported. If the `pout` keyword is used, a file of the prediction values will be written. Together the **train** and **predict** actions are functionally equivalent to a train/test experiment with the **roc** action. The **train** and **predict** pair becomes useful when one wishes to compute predictions on multiple datasets, or with the **super** learner. See the **train** description below, and the **super** learner description in Section B.4 for more details.

Mandatory keywords and arguments:

- `in` *dataset*: Specify the testing dataset. Section B.3 describes data file formats and naming conventions.
- `load` *filename*: Specify the filename used for loading the trained learner.

Common keywords and arguments:

- `pout` *filename*: Specify a file into which the learner's predictions are written.

B.2.2.6 roc

ROC is an abbreviation for Receiver Operating Characteristic. An ROC curve measures how well a learner ranks factors (data points) in order of decreasing probability of being active (belonging to the positive class). See [5] for a more detailed description of ROC curves. To summarize an ROC curve, the area under the curve (AUC) is compared to the area under the ROC curve of a learner that optimally ranks the factors. An AUC near 1.0 is very good, and an AUC of 0.5 is expected when randomly guessing output values. The **roc** action can run and score either a train/test experiment or a k-fold cross-validation experiment. For a train/test

experiment the AUC for the testing dataset are reported. For a k-fold cross-validation experiment the average AUC per fold is reported, as well as the composite AUC for all fold predictions combined.

Mandatory keywords and arguments:

- `in` *dataset*: Specify the training dataset. Section B.3 describes data file formats and naming conventions.
- `learner` *learner_specification*: Specify the learner. Section B.4 describes the available learners.
- `testset` *dataset* or `num_folds` *int*: Use `testset` to specify the testing dataset in a train/test experiment. Section B.3 describes data file formats and naming conventions. Use `num_folds` to specify the number of folds in a k-fold cross-validation. `testset` and `num_folds` are mutually exclusive.

Common keywords and arguments:

- `frocname` *filename*: Specify a filename into which ROC data averaged across the folds of a k-fold cross-validation experiment is written. This file may contain the same comments that a a file specified by `rocname` is specified. This keyword is only valid for k-fold cross-validation experiments.
- `fout` *filename*: Specify a file into which information about the folds of a k-fold cross-validation experiment are written. One line is written for each row in the training dataset. Each line has one integer which indicates in which fold the corresponding dataset row was "held out". This keyword is only valid for k-fold cross-validation experiments.
- `pout` *filename*: Specify a file into which the learner's predictions are written. For a train/test experiment these are the predictions on the test dataset. For a k-fold cross-validation experiment these are the combined predictions on the training dataset.
- `rocname` *filename*: Specify a file into which the coordinate pairs for the ROC curve are written. The file may have additional comments before or after the ROC data, denoted by an initial "#" symbol. Comments before the ROC data usually list some details about the experiment which produced the ROC data. Comments after the ROC data are usually AUC and related values for the experiment.

B.2.2.7 spardat

This action loads a sparse dataset, as defined in the Section B.3, and prints summary information.

Mandatory keywords and arguments:

- `in` *dataset*: Specify the training dataset. Section B.3 describes data file formats and naming conventions.

B.2.2.8 train

This action trains a learner as would be done for a train/test experiment, except no predictions are made. Instead, the trained learner is written to a file. This file may later be used with **predict** action to generate predictions on a dataset. This file may also be used with the **super** learner; please see the **super** learner description below in Section B.4.

Mandatory keywords and arguments:

- `in` *dataset*: Specify the training dataset. Section B.3 describes data file formats and naming conventions.
- `learner` *learner_specification*: Specify the learner. Section B.4 describes the available learners.
- `save` *filename*: Specify the filename used for saving the trained learner.

B.3 Data

The `./afc` program is able to read several dataset formats and store them in dense or sparse data structures. As a result, there is some complication in understanding and specifying datasets on the command-line.

B.3.1 Dataset files

The dataset file may be dense or sparse. A dense dataset file specifies the value of every attribute for every record using any real number. A sparse dataset contains only binary attribute values, and only the indices of the nonzero attributes are listed for each record.

B.3.1.1 Dense Datasets

All rows in a dense dataset file are comma-separated lists of real values. The first line of a dense dataset file should be a list of the input and output attribute names, and the the second line should be blank. All remaining lines are treated as records. Such dataset files are referred to as *csv* files. Any attribute of a *csv* file can serve as the output, and the output attribute is selected at dataset load time. Some flexibility is gained if the output is the last attribute. There is a second dense format, called *fds*, which is simply a specially compressed form of a *csv* file. This format is not documented, but can be created using the **convert** action described in Section B.2. Files in *fds* format are smaller and load much faster than *csv* files.

B.3.2 Sparse Datasets

There are two sparse dataset formats, *spardat* and *June*. Both are whitespace delimited, and both require the input attributes to be binary. In the *spardat* format, the output value is included in the dataset as the first token of each line, and can be any real value. However, the output attribute will be thresholded to a binary value at dataset load time. All remaining tokens on the line are interpreted as indices of nonzero attributes. Attribute numbering begins at zero, and lines beginning with "#" are ignored.

In the *June* format the input attributes and the output are in separate files. The rows of the input file consist entirely of attribute indices separated by whitespace. As with the *spardat* format, these indices start at zero and are interpreted as the indices of nonzero attributes. Attribute numbering begins at zero, and lines beginning with "#" are ignored. The output file is comma-delimited and can contain multiple columns. The first line should be a list of output names, and the second line should be blank. All remaining lines can contain almost anything, but only columns containing exclusively ACT and NOT_ACT tokens can be used as an output column. Note that attribute indices in the output file begin with zero and include columns not suitable for use as outputs; this is important when interpreting certain output files associated with the **mroc** action. As with *csv* files, selection of the output attribute is deferred until the dataset is loaded.

One additional dataset format modification is understood. If the attribute indices in a *spardat* or *June* formatted file are appended with ":1", the ":1" will be ignored and the data will be loaded as expected.

B.3.3 Dataset naming

All three dataset file formats, *csv*, *spardat* and *June*, require information beyond the file name when they are specified on the command-line. This information is specified though additional keywords and possibly extra characters appended to the filename. The dataset filename may also affect how the dataset is loaded. If the file ends in `.csv` or `.fds`, it is assumed to be in *csv* or *fds* format and an error will occur otherwise. Files not ending in `.csv` or `.fds` are assumed to be in either the *spardat* or *June* format.

When loading *csv* or *fds* files one must specify the filename and the output attribute. There are two methods for choosing the output attribute. If the keyword *csv* is written after the dataset name, separated by the usual whitespace between command-line tokens, then the last column of the dataset will be used as the output. The inputs are stored internally as real numbers in a dense matrix. The *csv* keyword is the only way to force our algorithms to use dense computations. Using the *csv* keyword with files not ending in `.csv` or `.fds` is an error. Not all learners can perform dense computations, and an error message will be displayed if such a learner is used with the *csv* keyword. The error message is likely to claim that the specified learner does not exist. Below is an example of performing dense computations. The *csv* dataset would be loaded, the last attribute used as the output, and the learner would use dense computations:

```
./afc roc in a-or-d.csv csv ...
```

The second method of choosing an output for *csv* and *fds* files uses the *output* keyword after the dataset name. This keyword must be followed by a valid attribute name from the *csv* or *fds* dataset. If this method is used, the dataset is assumed to consist entirely of binary input values, written in integer or floating point notation. Datasets in this form are stored internally using a sparse format, and the learning algorithms will generally employ efficient sparse techniques to speed learning and reduce memory load. If the dataset filename does not end in `.csv` or `.fds`, the *output* keyword will be ignored. The example below causes the same file used in the *csv* keyword example above to use attribute "y" as the output and allows learners to use sparse computations.

```
./afc roc in a-or-d.csv output y ...
```

It is important to realize that the `csv` or `output` keywords apply to all datasets on the command-line. For instance, in a train/test experiment, specifying the `csv` keyword causes the last attribute of both the training and the testing datasets to be used as the output. In this example, both datasets must be dense as well.

To use a *June* format file, one must specify the filename of the input dataset, the filename of the output dataset, and an attribute name from the output dataset. These are specified in the order just listed, separated by colons, and the output attribute must not end in `.csv` or `.fds`. The output is necessarily binary because of the constraints of the *June* format, as are the inputs. Datasets in the *June* format are stored internally in sparse form, and most learners will employ sparse computations for these datasets. If the input dataset filename is `factors.ssv`, the output dataset filename is `activations.ssv`, and attribute `Act` of `activations.ssv` is to be used as the output, one might type

```
./afc roc in factors.ssv:activations.ssv:Act ...
```

If a file fits neither the *csv*, *fds*or *June* formats, it is assumed to be in *spardat* format. The *spardat* format includes the output as the first column of each record, stored as a real number. Because the output will be stored internally in a sparse binary structure, the real-valued outputs in the file must be thresholded. The threshold specifier consists of a colon, a threshold value, and a plus or minus sign. A plus sign indicates that output values greater than or equal to the threshold value are considered positive. A minus sign indicates

that values less than or equal to the threshold are considered positive. As an example, suppose `sp.txt` is a *spardat* format dataset, and we wish to consider all outputs with value greater than 42.42 as positive outputs. The command line below specifies this using the notation just described.

```
./afc roc in sp.txt:42.42+ ...
```

B.4 Learners

Descriptions of all current learners in the software are described below. Each learner description includes all available keywords and a table summarizing the keywords and their argument types and bounds, where appropriate. From this information it is easy to write a learner specification for the actions which require a learner. In every case the learner name follows the `learner` keyword, separated by the customary whitespace. Keywords for the learner can be specified anywhere else on the command-line. It is possible for a learner to have keywords whose argument specifies one or more learners, for example with the **super** learner. The syntax used for such keywords is detailed in the keyword's description.

B.4.1 bc

This is an implementation of a Naive Bayesian Classifier [28; 5] with binary-valued inputs attributes (all input values must be zero or one). The implementation has been optimized for speed and accuracy on very high dimensional sparse data.

$$P(y = \text{ACT}|x_1, x_2 \ldots x_m) = \tag{B.1}$$

$$\frac{P(x_1, x_2 \ldots x_m|y = \text{ACT})P(y = \text{ACT})}{P(x_1, x_2 \ldots x_m|y = \text{ACT})P(y = \text{ACT}) + P(x_1, x_2 \ldots x_m|y = \text{NOT_ACT})(1 - P(y = \text{ACT}))} = \tag{B.2}$$

$$\frac{\theta_{\text{ACT}} p_{\text{ACT}}}{\theta_{\text{ACT}} p_{\text{ACT}} + \theta_{\text{NOT_ACT}}(1 - p_{\text{ACT}})} \tag{B.3}$$

where

$$\theta_c = \prod_{k=1}^{m} P(x_k|y = c) \tag{B.4}$$

under the (dubious) assumption that the input attributes $x_1 \ldots x_m$ are conditionally independent of each other given a known activity level, and where

$$p_{\text{ACT}} = P(Y = \text{ACT}) \tag{B.5}$$

Learning is trivial. From the dataset, p_{ACT} is simply estimated as the fraction of records in which $y_i = \text{ACT}$. And $P(x_k = v|y = c)$ is estimated as

$$\frac{\text{Number of records with } x_i = v \text{ and } y_i = \text{ACT}}{\text{Number of records with } y_i = \text{ACT}} \tag{B.6}$$

Because of the vast numbers of attributes and because of the need to exploit sparseness, a number of computational tricks are used, including a method to permit all probabilities to be evaluated in log space, and a subtraction trick to avoid needing to ever explicitly iterate over elements of a record in who have a value of zero.

BC needs no keywords or arguments.

B.4.2 dtree

This is an implementation of a Decision Tree Classifier with binary-valued inputs attributes (all input values must be zero or one).

Keyword	Arg Type	Arg Vals	Default
Common			
`chi_threshold`	float	[0,∞)	0.05

Common keywords and arguments:

- `chi_threshold` *float*: χ^2-threshold, written as a decimal (not a percentage).

B.4.3 lr

This is an implementation of logistic regression (LR). LR computes β for which the model values μ_i best approximate the dataset outputs y_i under the model

$$\mu_i = \frac{\exp(\beta_0 + \beta_1 x_{i1} + \cdots \beta_M x_{iM})}{1 + \exp(\beta_0 + \beta_1 x_{i1} + \cdots \beta_M x_{iM})}$$

where x_i is one dataset row. Please see [13; 25; 10; 20] for details about logistic regression. This implementation uses iterative re-weighted least squares (IRLS) [10; 20] to maximize the LR log-likelihood

$$\sum_n i = 1^R y_i \ln(u_i) + (1 - y_i)\ln(1 - u_i)$$

where R is the number of rows in the dataset. For logistic regression, IRLS is equivalent to Newton-Raphson [25]. To improve the speed of IRLS, this implementation uses conjugate gradient [31; 7; 26; 27] as an approximate linear solver [20]. This solver is applied to the linear regression

$$\left(X^T W X\right)\beta_{\text{new}} = X^T W z \tag{B.7}$$

where $W = \text{diag}(\mu_i(1-\mu_i))$ and $z = X\beta_{\text{old}} + W^{-1}(y - \mu)$. The current estimate of β is scored using the likelihood ratio $-2log(L_{\text{sat}}/L_{\text{current}})$, where $L_{\text{sat})}$ is the likelihood of of a saturated model with R parameters and L_{current} is the likelihood of the current model. This ratio is called the "deviance", and the IRLS iterations are terminated when the relative difference of the deviance between iterations is sufficiently small. Other termination measures can be added, such as a maximum number of iterations.

Keyword	Arg Type	Arg Vals	Default
Common			
`cgdeveps`	float	[1e-10, ∞)	0.0
`cgeps`	float	[1e-10, ∞)	0.001
`lrmax`	int	0,. . . , ∞	30
`rrlambda`	float	[0, ∞)	10.0
Rare			
`binitmean`	none		
`cgbinit`	none		
`cgwindow`	int	0, . . . , ∞	3
`cgdecay`	float	[1.0, ∞)	1000
`cgmax`	int	0, . . . , ∞	200
`holdout_size`	float	[0.0, 1.0]	0.0
`lrdevdone`	float	[0.0, ∞]	0.0
`lreps`	float	[1e-10, ∞)	0.05
`margin`	float	[0.0, ∞)	0
`modelmax`	float	(`modelmin`, 1.0]	1.0
`modelmin`	float	[0.0, `modelmax`)	0.0
`wmargin`	float	[0.0, 0.5)	0.0

Common keywords and arguments:

- `cgdeveps` *float*: This parameter applies to conjugate gradient iterations while approximating the solution β_{new} in Equation B.7. If `cgdeveps` is positive, conjugate gradient is terminated if relative difference of the deviance falls below `cgdeveps`. The recommended value of `cgdeveps` is 0.005. Decrease `cgdeveps` for a tighter fit. Increase `cgdeveps` to save time or if numerical issues appear. Exactly one of `cgdeveps` and `cgeps` can be positive.

- `cgeps` *float*: This parameter applies to conjugate gradient iterations while approximating the solution β_{new} in Equation B.7. If `cgeps is positive`, conjugate gradient is terminated when the conjugate gradient residual falls below `cgeps` times the number of attributes in the data. Decrease `cgeps` for a tighter fit. Increase `cgeps` to save time or if numerical issues appear. Exactly one of `cgeps` and `cgdeveps` can be positive.

- `lrmax` *float*: This parameter sets an upper bound on the number of IRLS iterations. If performing a train/test experiment, you may want to decrease this value to 5, 2, or 1 to prevent over-fitting of the training data.

- `rrlambda` *float*: This is the ridge regression parameter λ. When using ridge regression, the large-coefficient penalty $\lambda\beta^T\beta$ is added to the sum-of-squares error for the linear regression represent by Equation B.7.

Rare keywords and arguments:

- `binitmean`: If this keyword is used, the model offset parameter β_0 is initialized to the mean of the output values y_i. [26] reports that this eliminated some numerical problems in his implementation. We have not observed significant changes in our implementation when using this technique.

- `cgbinit`: If `cgbinit` is specified, each IRLS iteration will start the conjugate gradient solver at the current value of β. By default the conjugate gradient solver is started at the zero vector.
- `cgwindow` *int*: If the previous `cgwindow` conjugate gradient iterations have not produced a β with smaller deviance than the best deviance previously found, then conjugate gradient iterations are terminated. As usual, the β corresponding to the best observed deviance is returned to the IRLS iteration.
- `cgdecay` *float*: If the deviance ever exceeds `cgdecay` times the best deviance previously found, conjugate gradient iterations are terminated. As usual, the β corresponding to the best observed deviance is returned to the IRLS iteration.
- `cgmax` *int*: This is the upper bound on the number of conjugate gradient iterations allowed. The final value of β is returned.
- `holdout_size` *float*: If this parameter is positive IRLS iterations are terminated based on predictions made on a holdout set, rather then according to the relative difference of the deviance. This parameter specified the percentage, between 0.0 and 1.0, of the training data to be held out. In a k-fold cross-validation experiment, the training data is not the whole dataset but the data available for training during each fold. The deviance on the holdout set replaces the deviance on the prediction data.
- `lrdevdone` *float*: IRLS iterations will be terminated if the deviance becomes smaller than `lrdevdone`.
- `lreps` *float*: IRLS iterations are terminated If the relative difference of the deviance between IRLS iterations is less than `lreps`.
- `margin` *float*: The dataset outputs, which must be zero or one (c.f. Section B.3), are "shrunk" by this amount. That is, 1 is changed to 1 - `margin` and 0 is changed to `margin`. It is recommended that this parameter be left at its default value.
- `modelmax` *float*: This sets an upper bound on μ_i computed by the logistic regression. Although the computation of μ_i should assure it is positive, round-off error can create problems. It is recommended that this parameter be left at its default value.
- `modelmin` *float*: This sets a lower bound on μ_i computed by the logistic regression. Although the computation of μ_i should assure it is strictly less than one, round-off error can create problems. It is recommended that this parameter be left at its default value.
- `wmargin` *float*: wmargin is short for "weight margin". If nonzero, weights less than `wmargin` are are changed to `wmargin` and weights greater than 1 - `wmargin` are changed to 1 - `margin`. This is option is very helpful for controlling numerical problems, especially when `rrlambda` is zero. Values between 0.001 and 1e-15 are reasonable.

B.4.4 lr_cgmle

This is an implementation of logistic regression (LR). LR computes β for which the model values μ_i best approximate the dataset outputs y_i under the model

$$\mu_i = \frac{\exp(\beta_0 + \beta_1 x_{i1} + \cdots \beta_M x_{iM})}{1 + \exp(\beta_0 + \beta_1 x_{i1} + \cdots \beta_M x_{iM})}$$

where x_i is one dataset row. Please see [13; 25; 10; 20] for details about logistic regression. This implementation uses a conjugate gradient algorithm [31; 7; 26; 27] to maximize the LR log-likelihood

$$\sum_{i=1}^{R} y_i \ln(u_i) + (1 - y_i) \ln(1 - u_i)$$

where R is the number of rows in the dataset. The current estimate of β is scored using the likelihood ratio $-2log(L_{\text{sat}}/L_{\text{current}})$, where $L_{\text{sat)}}$ is the likelihood of of a saturated model with R parameters and L_{current} is the likelihood of the current model. This ratio is called the "deviance", and the conjugate gradient iterations are terminated when the relative difference of the deviance between iterations is sufficiently small. Other termination measures can be added, such as a maximum number of iterations.

Keyword	Arg Type	Arg Vals	Default
Common			
`cgeps`	float	[1e-10, ∞)	0.0005
`rrlambda`	float	[0.0, ∞)	10.0
Rare			
`binitmean`	none		
`cgdecay`	float	[1.0, ∞)	1000.0
`cgmax`	int	0,...,∞	100
`cgwindow`	int	0,...,∞	1
`dufunc`	int	1,..., 4	1
`initalpha`	float	(0, ∞)	1e-8
`margin`	float	[0, ∞)	0.0
`modelmax`	float	(`modelmin`, 1.0]	1.0
`modelmin`	float	[0.0, `modelmax`)	0.0

Common keywords and arguments:

- `cgeps` *float*: If the relative difference of the deviance is below `cgeps`, iterations are terminated. Decrease `cgeps` for a tighter fit. Increase `cgeps` to save time or if numerical issues appear.
- `rrlambda` *float*: This is the ridge regression parameter λ. When using ridge regression, the large-coefficient penalty $\lambda\beta^T\beta$ is subtracted from likelihood. This is equivalent to adding $2\lambda\beta^T\beta$ to the logistic regression deviance. This is not really ridge regression, because this isn't a linear regression problem with a sum-of-squared error.

Rare keywords and arguments:

- `binitmean`: If this keyword is used, the model offset parameter β_0 is initialized to the mean of the output values y_i. [26] reports that this eliminated some numerical problems in his implementation. We have not observed significant changes in our implementation when using this technique.
- `cgdecay` *float*: If the deviance ever exceeds `cgdecay` times the best deviance previously found, conjugate gradient iterations are terminated. The β corresponding to the best observed deviance among the iterations is used as the solution for the fold.

- `cgmax` *int*: This is the upper bound on the number of conjugate gradient iterations allowed. The β corresponding to the best observed deviance among the iterations is used as the solution for the fold.

- `cgwindow` *int*: If the previous `cgwindow` conjugate gradient iterations have not produced a β with smaller deviance than the best deviance previously found, then conjugate gradient iterations are terminated. The β corresponding to the best observed deviance among the iterations is used as the solution for the fold.

- `dufunc` *int*: The "direction update" function chooses the next linesearch direction in conjugate gradient iterations. There are four options represented by the integers one through four.

1	Polak-Ribiere (non-neg)	$\max\left(0.0, \frac{r_i^T z_i - r_i^T z_{i-1}}{r_{i-1} z_{i-1}}\right)$
2	Polak-Ribiere	$\frac{r_i^T z_i - r_i^T z_{i-1}}{r_{i-1} z_{i-1}}$
3	Hestenes-Stiefel	$\frac{r_i^T z_i - r_i^T z_{i-1}}{r_i^T p_{i-1} - r_{i-1} p_{i-1}}$
4	Fletcher-Reeves	$\frac{r_i^T z_i}{r_{i-1} z_{i-1}}$

  The symbol r_i represents the opposite of the gradient during the ith iteration, z_i is related to the preconditioning matrix and is the same as r_i at this time, and p_{i-1} is the previous search direction. The default direction update function is Polak-Ribiere (non-neg). All four direction updates produce the same result for quadratic objective functions. However, the logistic regression likelihood function is not quadratic in it's parameters.

- `initalpha` *float*: This parameter is the initial line search step length for each conjugate gradient iteration. It is unlikely to affect the quality of results so long as it is sufficiently small. However, it may efficiency.

- `margin` *float*: The dataset outputs, which must be zero or one (c.f. Section B.3), are "shrunk" by this amount. That is, 1 is changed to 1 - `margin` and 0 is changed to `margin`. It is recommended that this parameter be left at its default value.

- `modelmax` *float*: This sets an upper bound on μ_i computed by the logistic regression. Although the computation of μ_i should assure it is positive, round-off error can create problems. It is recommended that this parameter be left at its default value.

- `modelmin` *float*: This sets a lower bound on μ_i computed by the logistic regression. Although the computation of μ_i should assure it is strictly less than one, round-off error can create problems. It is recommended that this parameter be left at its default value.

B.4.5 newknn

This is an implementation of a fast Ball-tree based Classifier with binary-valued input attributes. We rely on the fact that these two questions are not the same: (a)"Find me the *k* nearest neighbors." and (b) "How many of the *k*-nearest neighbors are in the positive class?" Answering (b) exactly does not necessarily require us to answer (a).

newknn attacks the problem by building two ball trees: one ball tree is built from all the positive points in the dataset, another ball tree is built from all negative points. Then it is time to classify a new target point **t**, we compute *q*, the number of k nearest neighbors of **t** that are in the positive class, in the following fashion:

- Step 1. Find the k nearest positive class neighbors of $\mathbf{t}$ (and their distances to $\mathbf{t}$) using conventional ball tree search.
- Step 2. Do sufficient search of the negative tree to prove that the number of positive data points among k nearest neighbor is q for some value of q.

We expect **newknn** can achieve a good speed up for the dataset with much more negative output than positive ones.

Keyword	Arg Type	Arg Vals	Default
Common			
`k`	int	[0,∞)	9
`rmin`	int	[1,∞)	5
`mbw`	float	[1e-7,∞)	0.01

Common keywords and arguments:

- `k` *int*: number of nearest neighbor you are interested.
- `rmin` *int*: the maximum number of points that will be allowed in a tree node. Sometimes it saves memory to use rmin > 1. It even seems to speed things up, possibly because of L1/L2 cache effects.
- `mbw` *float*:"min_ball_width" = a node will not be split if it finds that all its points are within distance "mbw" of the node centroid. It will not be split (and will hence be a leaf node).

B.4.6 oldknn

This is an implementation of the conventional Ball-tree based k nearest neighbor search.

A *ball tree* is a binary tree in which each node represents a set of points, called Points(Node). Given a dataset, the *root node* of a ball tree represents the full set of points in the dataset. A node can be either a *leaf node* or a *non-leaf node*. A leaf node explicitly contains a list of the points represented by the node. A non-leaf node does not explicitly contain a set of points. It has two children nodes: *Node.child1* and *Node.child2*, where

$$Points(child1) \cap Points(child2) = \phi \text{ and } Points(child1) \cup Points(child2) = \textit{Points(Node)}$$

Points are organized spatially so that balls lower down the tree cover smaller and smaller volumes. This is achieved by insisting, at tree construction time, that

$$\begin{aligned} \mathbf{x} \in Points(Node.child1) \quad &\Rightarrow \quad |\,\mathbf{x} - Node.child1.Pivot\,| \;\leq\; |\,\mathbf{x} - Node.child2.Pivot\,| \\ \mathbf{x} \in Points(Node.child2) \quad &\Rightarrow \quad |\,\mathbf{x} - Node.child2.Pivot\,| \;\leq\; |\,\mathbf{x} - Node.child1.Pivot\,| \end{aligned}$$

This gives the ability to bound distance from a target point $\mathbf{t}$ to any point in any ball tree node. If $\mathbf{x} \in$ *Points(Node)* then we can be sure that:

$$|\mathbf{x} - \mathbf{t}| \;\geq\; |\mathbf{t} - Node.\mathbf{Pivot}| - Node.Radius \tag{B.8}$$

$$|\mathbf{x} - \mathbf{t}| \;\leq\; |\mathbf{t} - Node.\mathbf{Pivot}| + Node.Radius \tag{B.9}$$

The process of **oldknn** is described as below:

- Step 1. Build a ball tree on the training dataset.
- Step 2. Search the ball tree (Depth first recursive search) to find the k nearest neighbors of **t**. Some nodes can be pruned according to the above inequalities.

Keyword	Arg Type	Arg Vals	Default
Common			
`k`	int	[0,∞)	9
`rmin`	int	[1,∞)	5
`mbw`	float	[1e-7,∞)	0.01

Common keywords and arguments:

- `k` *int*: number of nearest neighbor you are interested.
- `rmin` *int*: the maximum number of points that will be allowed in a tree node. Sometimes it saves memory to use rmin > 1. It even seems to speed things up, possibly because of L1/L2 cache effects.
- `mbw` *float*: "min_ball_width" = a node will not be split if it finds that all its points are within distance "mbw" of the node centroid. It will not be split (and will hence be a leaf node).

B.4.7 super

This learner combines the results of other learners into a single prediction. This two-step technique starts by applying a set of learners, called the sublearners, to part of the training dataset. The sublearners then make predictions on remainder of the training dataset, and these predictions are stored in a new, temporary dataset. The outputs for the temporary dataset are the same as the outputs for corresponding rows of the original training dataset. In the second step, a learner called the superlearner is trained on the temporary dataset. The superlearner is then applied to the testing dataset or to the held-out data points in a k-fold cross-validation experiment. This idea has been thoroughly investigated under the name *stacking* [45].

Keyword	Arg Type	Arg Vals	Default
Common			
`learners`	string		
`superlearner`	string		lr

Common keywords and arguments:

- `learners` *string*: This keyword allows specification of the sublearners. The *string* argument allows passing of sublearner names and arguments. The sublearner name is separated from its arguments by

spaces, while the learners are separated from one-another by colons. This information must be parsed as a a single string, and your command shell may require quoting of the *string* argument. For example to specify Bayes Classifier and K-NN with $k = 9$ as sublearners, one might use "bc:newknn k 9" as the *string* argument for the `learners` keyword. It is also possible to use learners saved with the `train` action. If the trained learner is saved in a file `learnerfile`, then it may be specified with the format "FILE learnerfile" as a sublearner.

- `superlearner` *string*: This keyword allows specification of the superlearner. The *string* argument allows passing of the superlearner name and arguments. The superlearner name is separate from its arguments by spaces. This information must be parsed as a a single string, and your command shell may require quoting of the *string* argument. For example to specify K-NN with $k = 9$ as the superlearner, one might use "newknn k 9" as the *string* argument. Note that it is not possible to load the superlearner from a file saved by the `train` action using the `superlearner` keyword; please use the `predict` action instead.

B.4.8 svm

This is an implementation of a support vector machine (SVM) classifier. We didn't actually implement our own SVM, but embedded SVM^{light} [18] in the software. SVM is a novel type of learning machine which tries to find the largest margin linear classifier. Vapnik [42] shows how training a support vector machine leads to the following quadratic optimization problem:

$$\begin{array}{rcl} \mathbf{W}(\alpha) & = & -\sum_{i=1}^{l}\alpha_i + \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j k(x_i, x_j) \\ \sum_{i=1}^{l} y_i \alpha_i & = & 0 \\ \alpha_i & \geq & 0, \quad i{=}1\ldots\ell \\ \alpha_i & \leq & C, \quad i{=}1\ldots\ell \end{array} \tag{B.10}$$

The number of training examples is denoted by ℓ. α is a vector of ℓ variables, where each component α_i corresponds to a training example (x_i, y_i). The solution of Problem B.10 is the vector α^* for which $\mathbf{W}(\alpha)$ is minimized and the constraints are satisfied. Defining the matrix Q as $Q_{ij} = y_i y_j k(x_i, x_j)$, this can equivalently be written as

$$\begin{array}{lrcl} \text{minimize} & \mathbf{W}(\alpha) & = & -\alpha^T 1 + \frac{1}{2}\alpha^T Q \alpha \\ \text{such that} & \alpha^T y & = & 0 \\ & \alpha_i & \geq & 0, \quad i{=}1\ldots\ell \\ & \alpha_i & \leq & C1, \quad i{=}1\ldots\ell \end{array} \tag{B.11}$$

Two good tutorials for SVM are [29] and [3].

Keyword	Arg Type	Arg Vals	Default
Common			
`capacity`	float	[1e-10, ∞)	10.0
`cost`	float	[1e-10, 1e+10]	1.0
`gamma`	float	(0, ∞)	0.002
`kernel`	int	0, 2	0
`svmdir`	path		./
`usedefcapacity`	none		
`usedefgamma`	none		
Rare			
`learn`	filename		svm_learn
`classify`	filename		svm_classify
`example`	filename		svm_example
`model`	filename		svm_model
`result`	filename		svm_result

Common keywords and arguments:

- `capacity` *float*: This corresponds to the -c option for svm_learn. From the SVM^{light} command-line help:

  > C: trade-off between training error and margin (default avg. $(x^T x)^{-1}$)

  If you would like to use the SVM^{light} default value, please see the `usedefcapacity` argument, below.

- `cost` *float*: This corresponds to the -j option for svm_learn. From the SVM^{light} command-line help:

  > Cost: cost-factor, by which training errors on positive examples outweigh errors on negative examples (default 1) (see K. Morik, P. Brockhausen, and T. Joachims, Combining statistical learning with a knowledge-based approach - A case study in intensive care monitoring. International Conference on Machine Learning (ICML), 1999.

- `gamma` *float*: This corresponds to the -g option for svm_learn. From the SVM^{light} command-line help:

  > parameter gamma in rbf kernel

  If you would like to use the SVM^{light} default value, please see the `usedefgamma` argument, below.

- `kernel` *int*: This corresponds to the -k option for svm_learn. From the SVM^{light} command-line help:

  > type of kernel function:
  > 1: linear (default)
  > 2: polynomial $(sa * b + c)^d$
  > 3: radial basis function $\exp(-gamma||a-b||^2)$
  > 4: sigmoid $tanh(sa * b + c)$
  > 5: user defined kernel from kernel.h

Note that at present we only support linear and radial basis function kernels; additional kernels from the list above will be supported upon request.

- `svmdir` *path*: This option specifies the directory in which the SVM^{light} executables can be found. If the `learn` or `classify` options are set, their values are appended to the value of `svmdir`
- `usedefcapacity` *none*: If you specify this option, the default capacity of SVM^{light} will be used in place of this program's default.
- `usedefgamma` *none*: If you specify this option, the default gamma of SVM^{light} will be used in place of this program's default.

Rare keywords and arguments:

- `classify` *filename*: This allows specification of the svm_classify executable name. The value of *classify* will be appended to the value of *svmdir* (path separators are inserted as necessary).
- `example` *filename*: This allows specification of the svm_example executable name. The value of *example* will be appended to the value of *svmdir* (path separators are inserted as necessary).
- `learn` *filename*: This allows specification of the svm_learn executable name. The value of *learn* will be appended to the value of *svmdir* (path separators are inserted as necessary).
- `model` *filename*: This allows specification of the svm_classify executable name. The value of *model* will be appended to the value of *svmdir* (path separators are inserted as necessary).
- `result` *filename*: This allows specification of the svm_classify executable name. The value of *result* will be appended to the value of *svmdir* (path separators are inserted as necessary).

B.5 Examples

Below are examples of how to invoke the `./afc` software. For consistency among the examples, the same dataset names are used throughout. `dense-train.csv` and `dense-test.csv` are dense binary datasets, with the test datasets having an attribute named `rslt`. `spardat-train.txt` and `spardat-test.txt` are *spardat* format datasets. Recalling that *June* format datasets are split into an input and output file, there are four files comprising two *June* examples: `june-train.inputs.ssv`, `june-train.outputs.ssv`, `june-test.inputs.ssv` and `june-test.outputs.ssv`. The *June* output files also have an attribute named `rslt`.

Note that some examples are too long to fit on one line. However, all examples should be entered on a single line.

B.5.1 Simple roc examples on several datasets

Dense input file, dense computations, k-fold cross-validation experiment with 10 folds:

```
./afc roc in dense-train.csv csv num_folds 10 learner lr arghelp
```

Dense input files, dense computations, train/test experiment:

```
./afc roc in dense-train.csv csv testset dense-test.csv learner lr
```

Dense input files, sparse computations, train/test experiment:

```
./afc roc in dense-train.csv output rslt testset dense-test.csv learner lr
```

Sparse input file, sparse computations, k-fold cross-validation experiment with 10 folds:

```
./afc roc in spardat-train.txt:0.5+ num_folds 10 learner bc
```

It is possible to train on a *spardat* format dataset and test on a *June* format file:

```
./afc roc in spardat-train.txt:0.5+ testset june-test.inputs.ssv:june-test.outputs.ssv:rslt
learner bc
```

B.5.2 Examples with various actions

B.5.2.1 convert

In this example, a dense dataset is converted to a sparse dataset. The output dataset, `new-spardat.txt`, is in *spardat* format because the filename didn't end in `.csv`, `.fds` or `.ssv`. The *spardat* outputs are written as 0.0 and 1.0, with 1.0 representing positive records from the dense dataset. When using `new-spardat.txt`, a threshold of 0.5 is appropriate.

```
./afc convert in dense-train.csv csv out new-spardat.txt
```

Convert from *spardat* to *csv*:

```
./afc convert in spardat-train.txt:0.5+ out new-dense.csv
```

When convert to *June* format, multiple files are created. In this example, a *spardat* format dataset is converted into the pair of *June*-formatted files `new-june.factors.ssv` and `new-june.activations.ssv`. The "factors" file contains the inputs, and the "activations" file contains the outputs.

```
./afc convert in spardat-train.txt:0.5+ out new-june.ssv
```

Create a dense subset from a dense dataset:

```
./afc convert in dense-train.csv csv out new-dense.csv subsize 5
```

B.5.2.2 mroc

This `multipredict` example performs one **roc** experiment for each suitable attribute in `june-train.outputs.txt`. A suitable attribute is one for which records have only the values ACT and NOT_ACT. For each of these **roc** experiments, a `fout`, `pout` and `rocname` file will be written, each with `--out_attname` appended where `out_attname` is the output attribute's name for the current **roc** experiment. Note that the specification of attribute `rslt` in the *June* dataset specification has no meaning, but is necessary.

```
./afc mroc multipredict in june-train.inputs.txt:june-train.outputs.txt:rslt
num_folds 10 learner lr fout folds.txt pout predicts.txt rocname roc.txt
```

This example shows a `multiclass` train/test experiment. The predicted classes, which are indices of output attributes from the output files, are stored in the file specified by the `cout` option.

```
./afc mroc multiclass in june-train.inputs.txt:june-train.outputs.txt:rslt testset
june-test.inputs.txt:june-test.outputs.txt:rslt learner lr fout folds.txt pout
predicts.txt rocname roc.txt cout class.txt
```

B.5.2.3 train and predict

Training a learner:

```
./afc train in spardat-train.txt:0.5+ learner newknn k 5 save saved.bin
```

Predicting using a trained learner:

```
./afc predict in spardat-test.txt:0.5+ load saved.bin pout predicts.txt
```

B.5.3 Examples with various learners

An example of **super** learner use, where one of the sublearners is loaded from the file `saved.bin` created by the **train** example above:

```
./afc roc in spardat-train.txt:0.5+ testset spardat-test.txt:0.5+ learner super
superlearner ‘‘newknn k 101’’ learners ‘‘lr:bc:dtree chi 0.01:FILE saved.bin’’
pout predicts.txt
```

Appendix C

Miscellanea

We encountered several challenges while researching and writing this thesis, many of which had little to do with the research-at-hand. The problem of radically different BIOS images was explained in Section 5.1.5. We also discovered that the Intel Pentium 4 was unable to handle floating point exceptions efficiently unless the multimedia SSE2 registers were used for computation. Otherwise, a 1.7GHz Pentium 4 Xeon (circa 2001) was outperformed by a 200MHz Pentium MMX (circa 1997) on a *robust Cholesky decomposition* of a random symmetric matrix. [35; 20].

The most expensive problem was the repeated air conditioning outages in our computer lab. The root causes ranged from cracked chilled water coils to incorrect software configurations. The final four weeks of computations for this thesis were made possible by renting a portable air conditioner; see Figure C.1. Unfortunately we were too late – within hours of the air conditioner arrival, our file server refused to POST consistently, and shortly thereafter it was never heard from again.

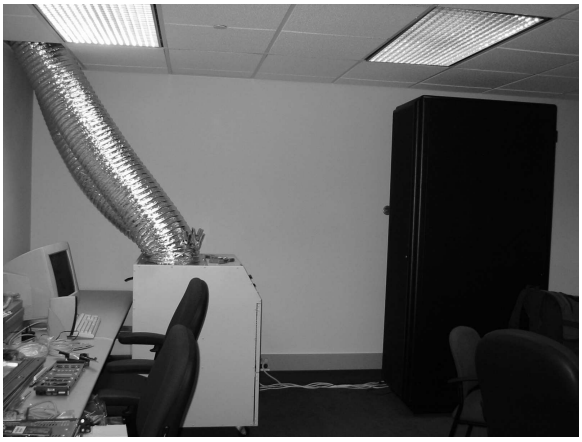


Figure C.1: On the right is a rack of computers, including our nine dual Opteron 242s used for this thesis. On the left is the portable air conditioning unit needed to keep our computers from combusting.

Bibliography

[1] AUTON (2004). The Auton Lab. http://www.autonlab.org.

[2] Breese, J. S., Heckerman, D., and Kadie, C. (1998). Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proceedings of UAI-1998: The Fourteenth Conference on Uncertainty in Artificial Intelligence*.

[3] Burges, C. (1998). A tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):955–974.

[4] CiteSeer (2002). CiteSeer Scientific Digital Library. http://www.citeseer.com.

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

[6] GLIM (2004). Generalised Linear Interactive Modeling package. http://www.nag.co.uk/stats/GDGE_soft.asp, http://lib.stat.cmu.edu/glim/.

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

[8] GSL (2004). GNU Scientific Library (GSL). http://www.gnu.org/software/gsl.

[9] Hammersley, J. M. (1950). The Distribution of Distance in a Hypersphere. *Annals of Mathematical Statistics*, 21:447–452.

[10] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Verlag.

[11] Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture: A Quantitative Approach*. Morgan Kaufman.

[12] Hogg, R. V. and Craig, A. T. (1995). *Introduction to Mathematical Statistics*. Prentice-Hall.

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

[14] IMDB (2002). Internet Movie Database. http://www.imdb.com.

[15] Joachims, T. (1998). Text categorization with support vector machines: learning with many relevant features. In Nédellec, C. and Rouveirol, C., editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, pages 137–142, Chemnitz, DE. Springer Verlag, Heidelberg, DE.

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

[17] Joachims, T. (2002a). *Learning to Classify Text Using Support Vector Machines*. PhD thesis, Cornell University.

[18] Joachims, T. (2002b). SVM^{light}. `http://svmlight.joachims.org`.

[19] Komarek, P., Liu, T., and Moore, A. (2004). Auton Fast Classifiers. `http://www.autonlab.org`.

[20] Komarek, P. and Moore, A. (2003). Fast Robust Logistic Regression for Large Sparse Datasets with Binary Outputs. In *Artificial Intelligence and Statistics*.

[21] Kozen, D. C. (1992). *The Design and Analysis of Algorithms*. Springer-Verlag.

[22] Kubica, J., Goldenberg, A., Komarek, P., Moore, A., and Schneider, J. (2003). A comparison of statistical and machine learning algorithms on the task of link completion. In *KDD Workshop on Link Analysis for Detecting Complex Behavior*, page 8.

[23] Lay, D. C. (1994). *Linear Algebra And Its Applications*. Addison-Wesley.

[24] Liu, T., Moore, A., and Gray, A. (2003). Efficient Exact k-NN and Nonparametric Classification in High Dimensions. In *Proceedings of Neural Information Processing Systems*, volume 15.

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

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

[27] Minka, T. P. (2001). Algorithms for maximum-likelihood logistic regression. Technical Report Stats 758, Carnegie Mellon University.

[28] Moore, A. W. (2001a). A Powerpoint tutorial on Probabilistic Machine Learning. Available from http://www.cs.cmu.edu/∼awm/tutorials/prob.html.

[29] Moore, A. W. (2001b). A Powerpoint tutorial on Support Vector Machines. Available from http://www.cs.cmu.edu/∼awm/tutorials/svm.html.

[30] Myers, R. H., Montgomery, D. C., and Vining, G. G. (2002). *Generalized Linear Models, with Applications in Engineering and the Sciences*. John Wiley & Sons.

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

[32] NCI Open Compound Database (2000). National Cancer Institute Open Compound Database. `http://cactus.nci.nih.gov/ncidb2`.

[33] Ng, A. Y. and Jordan, M. I. (2001). On Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes. In *Proceedings of Neural Information Processing Systems*, volume 13.

[34] Orr, M. (1996). Introduction to Radial Basis Function Networks. `http://www.anc.ed.ac.uk/~mjo/intro/intro.html`.

[35] Paul Komarek (2003). Robust Cholesky Performance on Intel's P4. http://www.andrew.cmu.edu/~komarek/notwork/ramblings/RobustCholeskyPerf/RobustCholeskyPerf.htm

[36] Pelleg, D. and Moore, A. (2002). Using Tarjan's Red Rule for Fast Dependency Tree Construction. In *Proceedings of Neural Information Processing Systems*, volume 14.

[37] Press, W. H., Teukolsky, S. A., Vetterlling, . W. T., and Flannery, B. P. (2002). *Numerical Recipes in C*. Cambridge University Press.

[38] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. San Mateo: Morgan Kaufmann.

[39] Russel, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[40] Schutze, H., Hull, D. A., and Pedersen, J. O. (1995). A Comparison of Classifiers and Document Representations for the Routing Problem. In *Research and Development in Information Retrieval*, pages 229–237.

[41] Shewchuk, J. R. (1994). An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical Report CS-94-125, Carnegie Mellon University, Pittsburgh.

[42] Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag.

[43] Wasserman, L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer.

[44] Watkins, D. S. (1991). *Fundamentals of Matrix Computations*. John Wiley & Sons.

[45] Wolpert, D. H. (1990). Stacked Generalization. Technical Report LA-UR-90-3460, Los Alamos National Labs, Los Alamos, NM.

[46] Yan, L., Dodier, R., Mozer, M. C., and Wolniewicz, R. (2003). Optimizing classifier performance via an approximation to the wilcoxon-mann-whitney statistic. In *Proceedings of the 20th International Conference on Machine Learning*.

[47] Yang, Y. and Liu, X. (1999). A re-examination of text categorization methods. In *22nd Annual International SIGIR*, pages 42–49, Berkeley.

[48] Zhang, J., Jin, R., Yang, Y., and Hauptmann, A. G. (2003). Modified logistic regression: An approximation to svm and its applications in large-scale text categorization. In *Proceedings of the 20th International Conference on Machine Learning*.

[49] Zhang, T. and Oles, F. J. (2001). *Text Categorization Based on Regularized Linear Classification Methods*. Kluwer Academic Publishers.