ABSTRACT

Title of thesis: THE USE OF PRECONDITIONING FOR TRAINING SUPPORT VECTOR MACHINES

Jhacova Ashira Williams, Master of Science, 2008

Thesis directed by: Dr. Dianne P. O'Leary
Department of Computer Science and Institute for Advanced Computer Studies

Since the introduction of support vector machines (SVMs), much work has been done to make these machines more efficient in classification. In our work, we incorporated the preconditioned conjugate gradient method (PCG) with an adaptive constraint reduction method developed in 2007 [8, 9] to improve the efficiency of training the SVM when using an Interior-Point Method. As in [8, 9], we reduced the computational effort in assembling the matrix of normal equations by excluding unnecessary constraints. By using PCG and refactoring the preconditioner only when necessary, we also reduced the time to solve the system of normal equations. We also compared two methods to update the preconditioner. Both methods consider the two most recent diagonal matrices in the normal equations. The first method [13] chooses the indices to be updated based on the difference between the diagonal elements while the second method chooses based on the ratio of these elements. Promising numerical results for dense matrix problems are reported.

THE USE OF PRECONDITIONING FOR TRAINING SUPPORT VECTOR MACHINES

by

Jhacova Ashira Williams

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2008

Advisory Committee:
Dr. Dianne P. O'Leary, Chair/Advisor
Dr. Brian Hunt
Dr. Howard Elman

Table of Contents

List of Abbreviations

SVM	Support Vector Machine
CG	Conjugate Gradient Method
PCG	Preconditioned Conjugate Gradient Method
IPM	Interior Point Method
MPC	Mehrotra's Predictor-Corrector
CQP	Convex Quadratic Program

Chapter 1

Introduction

A *support vector machine* (SVM) is a tool used to classify patterns. Several patterns with associated predetermined classification labels (positive or negative) are given as input. A SVM is *trained* by finding a hyperplane that separates the two classes of patterns. Once the training procedure is complete, the machine can be used to classify future patterns. A pattern is labeled as positive or negative depending on which side of the hyperplane it lies on. If there is a hyperplane that separates the positive patterns from the negative patterns, then the separating hyperplane can be found by solving a convex quadratic program (CQP).

Our work uses the *Interior-Point Method* (IPM) with adaptive constraint reduction developed in [8, 9] to find the hyperplane. During the reduction process, patterns that are farthest from the separating hyperplane are eliminated since they have no effect on training the SVM. We assume our problem can be modeled as a CQP and use primal-dual interior-point methods to find the best separating hyperplane. Our contribution is to use the *preconditioned conjugate gradient method* (PCG) to solve the *normal equations* problem at each step of the IPM. PCG solves a system of linear equations by using a preconditioner matrix that is chosen such that convergence to the "true" solution is rapid. The matrix of the normal equations is symmetric and positive definite. These equations are very useful because they deter-

mine our IPM step but involve fewer variables, which reduces the complexity of the computation. For PCG, we tested several preconditioners and found that using the Cholesky factorization of a previous normal equations matrix as a preconditioner significantly reduced computational time. We also applied a low-rank update to the preconditioner to try to improve efficiency.

In the next chapter, we discuss SVMs in detail and give an example of a data set used in the training procedure. A literature review can be found in Chapter 3. We define the CQP used for training the SVM in Chapter 4. Also in this chapter, the separation margin is defined and we derive the CQP's primal and dual problem. Chapter 5 defines a Primal-Dual Interior-Point Method for solving the CQP and defines the normal equations. We also describe Mehrotra's Predictor-Corrector Algorithm and discuss adaptive constraint reduction. The Conjugate Gradient Method and PCG are discussed in Chapter 6. Algorithms for both methods can be found in this chapter. Our results are presented in Chapter 7 and the conclusion can be found in Chapter 8.

Chapter 2

Support Vector Machine

2.1 Overview

A SVM is used to classify patterns (data vectors) as "positive" or "negative". Patterns are classified by answering "yes" the pattern lies in a particular halfspace or "no" it does not, based on the data. Several patterns, $\overline{a}_i$, with predetermined classification labels, $d_i \in \{+1, -1\}$, are given as input and are used to train the machine by constructing a hyperplane that separates the classes of patterns. The equation for the separating hyperplane is

$$h(x) = w^T x - \gamma, \tag{2.1}$$

such that

$$\text{sign}(h(\overline{a}_i)) = d_i. \tag{2.2}$$

The training procedure determines the vector w and the scalar γ in (2.1) that can be used to classify new patterns. If $d_i \geq 0$, $\overline{a}_i$ is positive; otherwise, $\overline{a}_i$ is negative. Many datasets used to train the SVM consist of a large number of patterns which are denoted $\overline{a}_i, i = 1, ..., m$. For example, the mushroom dataset includes over 8,000 samples corresponding to 23 species of mushrooms. Each sample is identified as edible (positive label) or poisonous (negative label) by the attributes in its data vector. The attributes describe physical characteristics of mushrooms such as scaly,

smooth, grooved and are used to categorize a mushroom as edible or poisonous. SVM finds a hyperplane to separate the known edible mushroom samples from the known poisonous mushroom samples. Then using (2.1) a new mushroom sample can be identified as edible or poisonous. For further discussion see [8, 9].

Chapter 3

Literature Review

The SVM was first developed by Vladimir Vapnik and his co-workers at AT&T Bell Labs in the mid 1990's. Since their introduction, much has been done to make these machines more efficient in classification. An in-depth tutorial on these machines is given by Burges [4]. The tutorial defines a SVM and illustrates its training process.

Inspired by the fact that many of the training patterns have no effect on training the machine, Jung [8, 9] used an adaptive constraint reduction interior-point method to assemble the normal equations. His algorithm adaptively eliminates constraints and uses the remaining constraints to construct an approximation to the normal equations. Since forming the matrix of normal equations is very costly, the reduction of constraints greatly reduces computation. The constraints that are eliminated at each iteration correspond to patterns that are far from the separating hyperplane. The reasoning for this is that patterns farthest from the hyperplane have little or no contribution to training the machine. Jung used Cholesky factorization and forward and backward substitution to solve the normal equations in the primal-dual interior-point method in order to obtain the search directions.

Wang [13] used an efficient method to reduce the computational work in finding the search direction for linear programming problems. He used a preconditioned

conjugate gradient method to reduce time complexity and space. In order to reduce computational work, his method applies a small rank change to update the preconditioner rather than recomputing it at every step. Updates correspond to constraints whose contributions to the matrix have changed the most between iterations. Wang also used an adaptive procedure to determine the appropriate time to use a direct method instead of an iterative method. It was found that a direct method should be used in the first step and an iterative method should be used during the middle stages. If the iterative method is found to be too costly, the algorithm switches to a direct method. This usually occurs during the final stages.

Wang's algorithm for solving the normal equations is compared to an algorithm developed by Baryamureeba, Steihaug, and Zhang in [2]. The algorithm is similar to Wang's algorithm in that it applies a low-rank correction to the preconditioner. However, the constraints are chosen differently, as we discuss in Chapter 6.

Chapter 4

The Convex Quadratic Program for training the SVM

The discussion in this chapter follows [8, 9]. If the positive and negative patterns are strictly separable by a hyperplane, the training procedure can be implemented by solving a convex quadratic program (CQP) as we show here. Assume that we have determined a vector w so that the equations for the positive and negative halfspace respectively are

$$w^T \overline{a}_i - \gamma \geq +1, \tag{4.1}$$

$$w^T \overline{a}_i - \gamma \leq -1, \tag{4.2}$$

for $i = 1, 2, ..., m$, with equality holding in (4.1) and (4.2) for some values of i. Then each halfspace contains a pattern that is closest to the separating hyperplane. The patterns that are closest to the separating hyperplane are called *support vectors* and are very useful in the training procedure. If we define two hyperplanes through the support vectors that are parallel to the separating hyperplane, the equations for the positive and negative hyperplanes respectively are

$$w^T x - \gamma = +1, \tag{4.3}$$

$$w^T x - \gamma = -1. \tag{4.4}$$

If we define x^- to be any point on the negative plane and x^+ to be the closest point to x^- on the positive plane, then equations (4.3) and (4.4) imply

$$w^T x^+ - \gamma = +1, \tag{4.5}$$

$$w^T x^- - \gamma = -1. \tag{4.6}$$

The distance between the two planes is

$$\|x^+ - x^-\|. \tag{4.7}$$

Solving equations (4.5) and (4.6) for x^+ and x^- respectively and substituting the result in (4.7), we find that the distance is $\frac{2}{\|w\|}$. This distance is called the *separation margin*. The best separating hyperplane will maximize the separation margin. Therefore, we find w by solving the CQP

$$\text{Min}_{w,\gamma} : \frac{1}{2} \|w\|_2^2 \tag{4.8}$$

$$\text{s.t. } D(Aw - \gamma e) \geq e, \tag{4.9}$$

where $D = diag(d_i)$, d_i is the classification label for pattern i, $A = (\overline{a}_1, ..., \overline{a}_m)^T$, and $e = (1, ..., 1)^T$. The matrix A has dimension $m \times n$. The constraints of the CQP require the patterns to be strictly separable. If our patterns are not strictly separable, this method cannot be applied to train the machine. In this case a penalty is added to the objective function to account for misclassification of patterns. In this case, the CQP becomes

$$\text{Min}_{w,\gamma,\tau} : \frac{1}{2} \|w\|_2^2 + \tau e^T y \tag{4.10}$$

$$\text{s.t. } D(Aw - \gamma e) + y \geq e, \tag{4.11}$$

where τ is the penalty parameter. Every primal problem such as (4.10) - (4.11) has an associated *dual problem* that consists of the same data arranged in a different way [14]. A solution to either the primal or dual problem determines a solution to both. The associated dual for the CQP in (4.10)-(4.11) is

$$\text{Max} : -\frac{1}{2}v^T Hv + e^T v \tag{4.12}$$

$$\text{s.t. } e^T Dv = 0, \tag{4.13}$$

$$0 \leq v \leq \tau e, \tag{4.14}$$

where $H = DAA^T D$. See [8, 9] for further discussion.

Chapter 5

A Primal-Dual Interior-Point Method for solving the CQP

5.1 The Idea Behind IPMs

In this chapter we describe the algorithm we used to solve the CQP. For further details, see [8, 9, 14]. The *Karush-Kuhn-Tucker (KKT) conditions* are the conditions that are necessary for a solution to the CQP to be optimal. The training is performed by finding the solution to these conditions. The KKT conditions for our primal and dual problems are

$$w - A^T Dv = 0, \tag{5.1}$$

$$d^T v = 0, \tag{5.2}$$

$$\tau e - v - u = 0, \tag{5.3}$$

$$DAw - \gamma d + y - e - s = 0, \tag{5.4}$$

$$Sv = 0, \tag{5.5}$$

$$Yu = 0, \tag{5.6}$$

$$s, u, v, y \geq 0, \tag{5.7}$$

where s and u are slack variables, $S = \text{diag}\,(s)$ and $Y = \text{diag}\,(y)$. Noice that (5.5)-(5.6) are nonlinear. We used an interior-point method (IPM) to solve the CQP.

In this method the primal-dual solution, $(w^*, \gamma^*, v^*, y^*, s^*, u^*)$, is found by applying Newton's method to a perturbation of the KKT conditions and computing a sequence of points that converges to the solution. All iterates satisfy the bound equation (5.7) so the sequence remains in the interior of the set of all feasible solutions. This set is defined as the *feasible region*. The main work in an IPM consists of determining search directions and step lengths. The search direction is calculated at each step by applying a variant of Newton's method, which forms a linear model around the current point. The search directions $(\Delta w, \Delta\gamma, \Delta v, \Delta y, \Delta s, \Delta u)$ satisfy

$$\begin{bmatrix} I & 0 & -A^TD & 0 & 0 & 0 \\ 0 & 0 & d^T & 0 & 0 & 0 \\ 0 & 0 & -I & 0 & 0 & -I \\ DA & -d & 0 & I & -I & 0 \\ 0 & 0 & S & 0 & V & 0 \\ 0 & 0 & 0 & U & 0 & Y \end{bmatrix} \begin{bmatrix} \Delta w \\ \Delta\gamma \\ \Delta v \\ \Delta y \\ \Delta s \\ \Delta u \end{bmatrix} = \begin{bmatrix} -w + A^TDv \\ -d^Tv \\ -\tau e + v + u \\ -DAw + \gamma d - y + e + s \\ -Sv \\ -Yu \end{bmatrix}, \tag{5.8}$$

where the left matrix is the Jacobian matrix of the KKT conditions, the right hand side of (5.8) is the residual of the KKT conditions, and $e = (1, ..., 1)^T$. If the iterate is in the interior of the feasible region, the step equation satisfies

$$\begin{bmatrix} I & 0 & -A^T D & 0 & 0 & 0 \\ 0 & 0 & d^T & 0 & 0 & 0 \\ 0 & 0 & -I & 0 & 0 & -I \\ DA & -d & 0 & I & -I & 0 \\ 0 & 0 & S & 0 & V & 0 \\ 0 & 0 & 0 & U & 0 & Y \end{bmatrix} \begin{bmatrix} \Delta w \\ \Delta \gamma \\ \Delta v \\ \Delta y \\ \Delta s \\ \Delta u \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -Sve \\ -Yue \end{bmatrix}. \tag{5.9}$$

The appropriate step length, α, is found by performing a line search along the Newton direction and is used to calculate the new iterate

$$(w, \gamma, v, y, s, u) = (w, \gamma, v, y, s, u) + \alpha(\Delta w, \Delta \gamma, \Delta v, \Delta y, \Delta s, \Delta u), \tag{5.10}$$

where $\alpha \in [0, 1]$.

A *central path*, C, along with the search directions and step lengths is used in the IPM to find the set of primal-dual solutions. C is a path of points

$$(w_\tau, \gamma_\tau, v_\tau, y_\tau, s_\tau, u_\tau) \tag{5.11}$$

parameterized by $\tau > 0$, that satisfy

$$w - A^T D v = 0, \tag{5.12}$$

$$d^T v = 0, \tag{5.13}$$

$$\tau e - v - u = 0, \tag{5.14}$$

$$DAw - \gamma d + y - e - s = 0, \tag{5.15}$$

$$Sv = \tau e, \tag{5.16}$$

$$Yu = \tau e, \tag{5.17}$$

$$s, u, v, y \geq 0. \tag{5.18}$$

These conditions are obtained from the KKT conditions by relaxing the complementary conditions, (5.5) - (5.6) to (5.16) - (5.17) equal to a positive scalar τ. The IPM converges to the solution by following C in the direction of decreasing τ. Each search direction is a step toward a point on C with a smaller value of τ. However since (5.7) must hold, the IPM remains in the interior of the feasible region. As a result, we hope that the IPM can take large steps before violating the bounds. In [14], a centering parameter $\sigma \in [0, 1]$ and a duality measure $\mu = \frac{s^T v + y^T u}{2m}$, where m is the number of patterns, are used to define a biased search direction. In this case, the step equation (5.9) becomes

$$\begin{bmatrix} I & 0 & -A^T D & 0 & 0 & 0 \\ 0 & 0 & d^T & 0 & 0 & 0 \\ 0 & 0 & -I & 0 & 0 & -I \\ DA & -d & 0 & I & -I & 0 \\ 0 & 0 & S & 0 & V & 0 \\ 0 & 0 & 0 & U & 0 & Y \end{bmatrix} \begin{bmatrix} \Delta w \\ \Delta \gamma \\ \Delta v \\ \Delta y \\ \Delta s \\ \Delta u \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -Sve + \sigma\mu e \\ -Yue + \sigma\mu e \end{bmatrix}. \tag{5.19}$$

There are many variants on this basic IPM idea, and in the next section we describe the one we used.

5.2 Mehrotra's Predictor-Corrector Algorithm

In our work, we used Mehrotra's predictor-corrector (MPC) algorithm. MPC generates feasible iterates that satisfy (5.7), and the search direction consists of two components:

- An affine-scaling predictor direction − the solution to (5.8),
- A centering and corrector direction that attempts to compensate for nonlinearity in the central path by using a centering parameter σ that is adaptively chosen.

Starting with an initial point in a defined neighborhood of C, the affine-scaling direction computes the search direction to reduce τ. If the affine-scaling direction reduces the duality measure μ significantly, the new iterate has not strayed far from C and little centering is needed; thus the centering term σ is chosen to be close to zero. Otherwise, much centering is needed and σ is chosen to be closer to one. The corrector step moves closer to C, giving the algorithm more room to maneuver during the next iteration in the affine-scaling direction. This process is repeated until τ is driven to zero. For more discussion, see [14]. We will discuss each of these two steps for our algorithm in turn.

From (5.8) we obtain the affine-scaling search direction by solving the Newton system of equations

$$\Delta w - A^T D \Delta v = -(w - A^T D v) \equiv -r_w, \tag{5.20}$$

$$d^T \Delta v = -d^T v \equiv -r_v, \tag{5.21}$$

$$-\Delta v - \Delta u = -(\tau e - v - u) \equiv -r_u, \tag{5.22}$$

$$DA\Delta w - d\Delta\gamma + \Delta y - \Delta s = -(DAw - \gamma d + y - e - s) \equiv -r_s, \tag{5.23}$$

$$S\Delta v + V\Delta s \equiv -r_{sv}, \tag{5.24}$$

$$Y\Delta u + U\Delta y \equiv -r_{yu}. \tag{5.25}$$

In the affine-scaling step, we set

$$r_{sv} = Sv, \tag{5.26}$$

$$r_{yu} = Yu. \tag{5.27}$$

In the centering and corrector step, we set

$$r_{sv} = Sv - \sigma\mu e + \Delta S^{aff}\Delta v^{aff}, \tag{5.28}$$

$$r_{yu} = Yu - \sigma\mu e + \Delta Y^{aff}\Delta u^{aff}. \tag{5.29}$$

We solved for the search directions using a smaller system of equations. The normal equations are found by solving the Newton system of equations (5.20) - (5.25) for the search direction. First, we solved (5.24) for Δs and solved (5.25) for Δu and obtain $\Delta s = -V^{-1}(r_{sv} + S\Delta v)$ and $\Delta u = -Y^{-1}(r_{yu} + U\Delta y)$. Next, we substitute the results for Δu and Δs in (5.22) and (5.23) respectively. Both equations can be rewritten as

$$-\Delta v + Y^{-1}U\Delta y = -r_u - Y^{-1}r_{yu} \equiv -\overline{r}_u, \tag{5.30}$$

$$DA\Delta w - d\Delta\gamma + \Delta y + V^{-1}S\Delta v = -r_s - V^{-1}r_{sv} \equiv -\overline{r}_s. \tag{5.31}$$

From (5.30) we obtain $\Delta y = U^{-1}Y(-\overline{r}_u + \Delta v)$. We then substitute Δy in (5.31) and rewrite the equation as

$$DA\Delta w - d\Delta\gamma + \Omega\Delta v = -\overline{r}_s + U^{-1}Y\overline{r}_u \equiv -r_\Omega \tag{5.32}$$

where $\Omega = V^{-1}S + U^{-1}Y$ and $r_\Omega = r_s + V^{-1}r_{sv} - U^{-1}Y\overline{r}_u$. The remaining equations (5.20), (5.21), and (5.32) are

$$\begin{bmatrix} I & 0 & -A^TD \\ 0 & 0 & d^T \\ DA & -d & \Omega \end{bmatrix} \begin{bmatrix} \Delta w \\ \Delta\gamma \\ \Delta v \end{bmatrix} = \begin{bmatrix} -r_w \\ -r_v \\ -r_\Omega \end{bmatrix}, \tag{5.33}$$

The normal equations are obtained by eliminating Δv and $\Delta\gamma$ using row reduction in (5.33):

$$(I + A^TD\Omega^{-1}DA - \frac{\overline{d}\ \overline{d}^{\ T}}{d^T\Omega^{-1}d})\Delta w = -\overline{r}_w - \frac{\overline{d}}{d^T\Omega^{-1}d}\overline{r}_v \tag{5.34}$$

where $\overline{r}_w = r_w + A^TD\Omega^{-1}r_\Omega, \overline{r}_v = r_v - d^T\Omega^{-1}r_\Omega$ and $\overline{d} = A^TD\Omega^{-1}d$. Once we have computed Δw from (5.34) we obtain the other variables from these relations:

$$\Delta\gamma = \frac{-\overline{r}_v + \overline{d}^T\Delta w}{d^T\Omega^{-1}d} \tag{5.35}$$

$$\Delta v = -\Omega^{-1}(r_\Omega + DA\Delta w - d\Delta\gamma) \tag{5.36}$$

$$\Delta y = -U^{-1}Y(\overline{r}_u - \Delta v) \tag{5.37}$$

$$\Delta u = -Y^{-1}(r_{yu} + U\Delta y) \tag{5.38}$$

$$\Delta s = -V^{-1}(r_{sv} + S\Delta v) \tag{5.39}$$

5.3 Adaptive Constraint Reduction

Jung proposed an adaptive constraint reduction primal-dual interior-point method for training the SVM [8, 9]. The matrix of the normal equations in (5.34)

is

$$M \equiv (I + A^T D\Omega^{-1} DA - \frac{\overline{d}\ \overline{d}^{\ T}}{d^T\Omega^{-1}d}) = I + \sum_{i=1}^{m} \omega_i^{-1} a_i a_i^T - \frac{(\sum_{i=1}^{m} \omega_i^{-1} a_i)(\sum_{i=1}^{m} \omega_i^{-1} a_i)^T}{\sum_{i=1}^{m} \omega_i^{-1}} \tag{5.40}$$

where $\omega_i^{-1} = \frac{v_i u_i}{s_i u_i + y_i v_i}$. Computation is greatly reduced by ignoring terms that have little contribution to the matrix M. In (5.40) patterns with large ω_i^{-1} make the largest contribution to the matrix. During the constraint reduction process, q is defined as the number of terms used to construct the matrix M_Q that approximates M and is chosen adaptively during each iteration. Once q is determined, the algorithm chooses a set of patterns Q used to assemble M_Q by one of the following criteria:

- Q is the set of q patterns with the smallest signed distance to the class boundary hyperplanes.
- Q is the set of q patterns with the smallest absolute distance to the class boundary hyperplanes.
- Q is the set of q patterns with the smallest ω_i^{-1}.

The set of patterns may contain an unbalanced number of positive and negative patterns. If a balanced number of patterns is needed, Jung partitions $q = q^+ + q^-$ and uses q^+ positive patterns and q^- negative patterns. Therefore, instead of M, we use

$$M_Q \equiv (I + A_Q^T D_Q \Omega_Q^{-1} D_Q A_Q - \frac{\overline{d}_Q\ \overline{d}_Q^T}{d_Q^T \Omega_Q^{-1} d_Q}) \tag{5.41}$$

and solve

$$M_Q \Delta w = -\overline{r}_w - \frac{\overline{d}}{d^T\Omega^{-1}d}\overline{r}_v \tag{5.42}$$

The adaptive procedure greatly reduces the amount of time to construct the matrix of normal equations, so we also used it in our algorithm. The Adaptive Constraint Reduction Method is summarized in Algorithm 1. We set $\sigma = (\frac{\mu_{aff}}{\mu})$ since Mehrotra has proved it to be effective in computational testing. We also chose Q to contain a balanced number of positive and negative patterns and assembled M_Q with the set of patterns with smallest distance to the class boundary hyperplanes.

Algorithm 1 Adaptive Constraint Reduction Method for SVM Training

Given a starting point (w, γ, y, s, v, u) with $(y, s, v, u) > 0$.

for $k = 0,1,2,...$ until convergence or infeasibility, **do**

1. Determine q
2. Determine Q.
3. Solve (5.42) and (5.35)-(5.39) using the quantities in (5.26) - (5.27), for $\left(\Delta w^{aff}, \Delta\gamma^{aff}, \Delta y^{aff}, \Delta s^{aff}, \Delta v^{aff}, \Delta u^{aff}\right)$.
4. Determine predictor step length:

$$\alpha_{aff} = \arg\max_{\alpha\in[0,1]}(y, s, v, u) + \alpha(\Delta y^{aff}, \Delta s^{aff}, \Delta v^{aff}, \Delta u^{aff}) \geq 0.$$

5. Set $\mu_{aff} = \frac{(s+\alpha_{aff}\Delta s^{aff})^T(v+\alpha_{aff}\Delta v^{aff})+(y+\alpha_{aff}\Delta y^{aff})^T(u+\alpha_{aff}\Delta u^{aff})}{2m}$.
6. Set $\sigma = (\frac{\mu_{aff}}{\mu})^3$.
7. Solve (5.42) and (5.35)-(5.39) using (5.28) - (5.29), for $(\Delta w, \Delta\gamma, \Delta y, \Delta s, \Delta v, \Delta u)$.
8. Determine step length for the combined step:

$$\alpha_{max} = \arg\max_{\alpha\in[0,1]}(y, s, v, u) + \alpha(\Delta y, \Delta s, \Delta v, \Delta u) \geq 0.$$

9. Select $\alpha_k = \min(0.99\alpha_{max}, 1)$.
10. Set $(w, \gamma, y, s, v, u) = (w, \gamma, y, s, v, u) + \alpha(\Delta w, \Delta\gamma, \Delta y, \Delta s, \Delta v, \Delta u)$.

end for

Chapter 6

Conjugate Gradient Method and Preconditioning

We need an algorithm for solving the normal equations (5.34) or (5.42) and we chose the conjugate gradient method (CG). For simplicity, we consider (5.34). CG is an iterative method used to find the solution to a particular kind of linear system of equations, namely systems whose matrix is symmetric and positive definite. The solution is found by computing approximate values at each iteration that converge to the true solution. It is useful for sparse matrices because it never modifies the matrix and only uses it for matrix-vector products. The search direction is found by solving the Newton system of equations (5.20)- (5.25) which requires a matrix-vector product at every step

$$Mp \equiv (I + A^T D\Omega^{-1} DA - \frac{\overline{d}\ \overline{d}^{\ T}}{d^T\Omega^{-1}d})p \tag{6.1}$$

where p is a vector [13]. If M is dense then (6.1) requires $2mn$ floating point multiplications if we compute this product as

$$Mp = p + A^T[(D\Omega^{-1}D)(Ap)] - \frac{\overline{d}\ (\overline{d}^{\ T}p)}{d^T\Omega^{-1}d}. \tag{6.2}$$

CG was used during the affine-scaling and centering-corrector step of the interior-point method to solve for Δw in Steps 3 and 7 of Algorithm 1. Assume our normal equations are of the form

$$Mu = b \tag{6.3}$$

where M is a matrix and u and b are vectors. CG from [10] is defined as follows:

Algorithm 2 Conjugate Gradient Method for solving $Mu = b$

Given an initial guess u_0.

1. Compute the residual $r_0 = b - Mu_0$ and set $d_0 = r_0$.

for $k = 0,1,...$ until convergence, **do**

2. Compute the appropriate step length, the new iterate, and the new residual.

$$\alpha_k = \frac{r_k^T r_k}{d_k^T M d_k},$$

$$u_{k+1} = u_k + \alpha_k d_k,$$

$$r_{k+1} = r_k - \alpha_k M d_k.$$

3. Compute the new search direction.

$$\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k},$$

$$d_{k+1} = r_{k+1} + \beta_{k+1} d_k.$$

end for

6.1 Preconditioned Conjugate Gradient Method

Many matrices used in CG are very troublesome for numerical computations. As a result, preconditioners can be used as input to CG to reduce the computation. Preconditioners reduce the condition number of the matrix which generally reduces the number of CG iterations. The number of CG iterations is related to the eigenvalues; if the condition number is low, the eigenvalues are clustered together and CG converges quickly. For example, if all eigenvalues are equal, PCG converges in

1 iteration.

Assume there exists a matrix C that is symmetric and positive definite and C is a close approximation to M but is easier to invert. Also assume that $C^{-1}M$ has a lower condition number. We can solve (6.3) by solving

$$C^{-1}Mu = C^{-1}b. \tag{6.4}$$

The purpose of preconditioners is to reduce the number of CG iterations. The preconditioners tested were the Identity, Diagonal, Cholesky, and Incomplete Cholesky matrices.

- The Diagonal preconditioner is a diagonal matrix with entries identical to the diagonal elements of the matrix of normal equations.

- The Cholesky preconditioner is the Cholesky factorization of the matrix of normal equations. Since M is symmetric and positive definite, we can decompose M into factors

  $$C = LL^T \tag{6.5}$$

  where L is a lower triangular matrix and L^T is the transpose of L. The factorization is done by using a modified version of Gaussian elimination. With this preconditioner, PCG converges in a single iteration for (6.4), but we use the same C for several IPM iterations.

- The Incomplete Cholesky preconditioner also uses Cholesky factorization to compute

  $$C \approx \overline{L}\ \overline{L}^T, \tag{6.6}$$

where $\overline{L}$ and $\overline{L}^T$ are the Incomplete Cholesky factors of C. If C is sparse, Cholesky factorization destroys the entries in C that are zero, which usually makes its factors less sparse. However, the Incomplete Cholesky factors of C are computed by Cholesky-like formulas that discard part of the less sparse entries of L in (6.5). Entries can be discarded based on position or value. For entries that are discarded based on position, a set $S \subseteq \{(i,j), i,j = 1, ..., n\}$. S defines positions in $\overline{L}$ in (6.6) that are allowed to be nonzero. Incomplete Cholesky factorization can be described as

$$C_{ij} = \begin{cases} C_{ij} - \frac{C_{ik}}{C_{kk}} C_{kj} & \text{if } (i,j) \in S \\ C_{ij} & \text{otherwise} \end{cases} \tag{6.7}$$

for each k and for i,$j > k$. This Incomplete Cholesky factorization discards entries based on position. In an alternative algorithm, a drop tolerance is given to the procedure. A *drop tolerance* is a positive scalar, and all entries in $\overline{L}$ with absolute value smaller than the drop tolerance are replaced by zero [3, 7, 11].

The preconditioned conjugate gradient method (PCG) from [10] is as follows.

Algorithm 3 Preconditioned Conjugate Gradient Method for solving $Mu = b$ with preconditioner C

Given an initial guess u_0.

1. Compute the residual

$$r_0 = b - Mu_0,$$

$$z_0 = d_0 = C^{-1}r_0.$$

for $k = 0,1,...$ until convergence, **do**

2. Compute the step length and the new iterate.

$$\alpha_k = \frac{r_k^T z_k}{d_k^T M d_k},$$

$$u_{k+1} = u_k + \alpha_k d_k.$$

3. Compute the new residual $r_{k+1} = r_k - \alpha_k M d_k$.

4. Let $z_{k+1} = C^{-1}r_{k+1}$

5. Compute the new search direction

$$\beta_{k+1} = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k},$$

$$d_{k+1} = z_{k+1} + \beta_{k+1} d_k.$$

end for

6.2 Updating/Downdating Cholesky Preconditioner

Computing a new preconditioner at each iteration requires much computation. In order to save time, a rank-α change can be applied to update the preconditioner. During IPM iterations, the diagonal matrix $\widehat{D} = D^2\Omega^{-1}$ is the only component of the matrix of normal equations that changes; thus the change in the diagonal matrix is used to update the preconditioner. We applied a sequence of rank-one updates to compute the preconditioner, as seen in Algorithm 4. See [13] for more details.

In [2], a rank-α change is also applied to update the preconditioner. However, the algorithm differs from [13] in that the choice of indices that are updated is based on the ratios of elements in $\widehat{D}$ to elements in $\widehat{D}_{old}$. During each iteration, the ratio of the diagonal elements are sorted and the indices corresponding to the largest and smallest elements are updated. We also scale the ratio by its mean so that its elements are closer to 1; thus the eigenvalues are more clustered together making convergence more rapid. Results in [2] showed that applying a low-rank update in this manner gives a tighter bound on the condition number for the preconditioner for an IPM for linear programming; thus making PCG converge faster. The algorithm can be seen in Algorithm 5.

Algorithm 4 Updating/Downdating the Preconditioner [13]

1. Let $C = LL^T$ be the Cholesky factorization of matrix M_{old}.

2. Let $\widehat{D}_{old}$ be the diagonal matrix for which we have a Cholesky factorization of M_{old}.

3. Let $\widehat{D}$ be the current diagonal matrix.

4. Define $\Delta D = \left|\widehat{D} - \widehat{D}_{old}\right|$.

5. Sort ΔD in descending order.

6. Let $\mathcal{I}$ be the set of α indices corresponding to large values of ΔD.

7. Update/Downdate the preconditioner and $\widehat{D}_{old}$:

   Set $\widehat{L} = L$, which is the Cholesky factor of M_{old}.

   for $i \in \mathcal{I}$, **do**

      Update the Cholesky factor to include $\pm\Delta d_i a_i^T a_i$.

      Set $\widehat{D}_{old}(i) = \widehat{D}_{old}(i) \pm \Delta\ d_i$.

   end for

Note: $\widehat{C} = \widehat{L}\widehat{L}^T$ is the preconditioner obtained by applying a rank-α update to C.

Algorithm 5 Updating/Downdating the Preconditioner [2]

1. Let $\widehat{D}_{old}$ be the diagonal matrix for which we have a Cholesky factorization of M_{old}.
2. Let $\widehat{D}$ be the current diagonal matrix.
3. Define $ratio_j = \max(\frac{(\widehat{D}_{old})_{jj}}{\widehat{D}_{jj}}, \frac{\widehat{D}_{jj}}{(\widehat{D}_{old})_{jj}})$.
4. Let $scale =$ mean of $ratio$.
5. Set $ratio = ratio/scale$.
6. Sort $ratio$ in descending order.
7. Let $\mathcal{I}$ be the set of indices corresponding to the largest values of $ratio$.
8. Define $\Delta D = \widehat{D}/scale - \widehat{D}_{old}$.
9. Update/Downdate the preconditioner and $\widehat{D}_{old}$:

   Set $\widehat{L} = L$, which is the Cholesky factor of M_{old}.

   for $i \in \mathcal{I}$, **do**

      Update the Cholesky factor to include $\pm\Delta d_i a_i^T a_i$.

      Set $\widehat{D}_{old}(i) = \widehat{D}_{old}(i) \pm \Delta\ d_i$.

   end for

Note: $\widehat{C} = \widehat{L}\widehat{L}^T$ is the preconditioner obtained by applying a rank-α update to C.

Chapter 7

Numerical Results

We implemented Algorithm 4, and Algorithm 5 in MATLAB. The matrix of normal equations can be either sparse or dense. If the matrix is sparse, we used functions ldlchol, ldlupdate, and ldlsolve created by Timothy A. Davis [5] to update the matrix. These functions are accessed through a Mex interface. Functions *ldlchol*, *ldlupdate*, and *ldlsolve* compute the Cholesky factorization, apply a rank one update, and solve $\widehat{C}u = b$ for u. If the matrix is dense, we used the MATLAB function *cholupdate* to apply a rank one update. We tested each algorithm using MATLAB version R14 on a machine running Windows XP with Intel Centrino Duo Processor 1.83GHz. We set the iteration limit to 75 and used balance reduction so that the subset of normal equations contained an equal number of positive and negative patterns that were the closest in distance to the boundary hyperplanes. As in [8, 9], we set $\beta = 4$, tol_r and tol_μ were both set to 10^{-5} and the initial point was set to $w = 0, \gamma = 0, y, s, v, u = 2e$. The tolerance for PCG was set to 10^{-3}.

We compared three algorithms on several problems. Jung's algorithm, seen in Algorithm 1, obtains a direct solution of the normal equations. The second algorithm uses PCG with Wang preconditioner by applying Algorithm 1 using PCG to obtain the normal equations and Algorithm 4 to obtain the preconditioner. The last algorithm uses PCG with B&S preconditioner by obtaining the normal equations

by using PCG in Algorithm 1 and obtaining the preconditioner using Algorithm 5. The problems tested were either sparse or dense. When a penalty parameter was needed, we used the same value as in [8, 9]. The sparse problems are a1a, a2a, a3a, a4a, a5a, a6a, a7a, a8a, a9a. These problems are adult data sets created by [12] and predict whether the income of a person is greater than $ 50,000 based on several census parameters such as age, race, education and marital status. Each problem differs in its size and sparsity which can be seen in the tables below. Tables 7.1 - 7.5 show our results for the sparse problems. Jung's algorithm outperforms our method in all problems. As one can see, applying no updates to the preconditioner is faster than applying 5 updates in each problem. We also tested preconditioning by refactoring every 3 and every 5 updates, but in each case, refactoring every 2 iterations works best. We also tried applying a larger number of updates, namely 100, 500, 1000. In each case, the time to solve the system of equations significantly increased.

We also tested our algorithm on several dense problems: letter, mushroom, wave, and wavenoise [6]. The dense problems are used for recognizing hand written letters, determining edible mushrooms and poisonous mushrooms, and categorizing waves. Our method along with Jung's method was tested on each dense problem. The results show that PCG with B & S preconditioner outperforms Jung's in each problem while PCG with Wang preconditioner outperforms Jung's method most of the time. As with the sparse problems, we also tested the problems on refactoring every 3 and every 5 iterations and tested applying 100, 500, and 1000 updates. Each of these problems outperform Jung's also; however, B & S Preconditioner

outperforms Wang's in these cases.

Table 7.1: Jin Jung Results for Sparse Matrices

Matrix	Size	Sparsity (Percent)	IP Iterations	Time to Run (sec.)
a1a	1605 x 119	16.21	14	0.328
a2a	2265 x 119	11.24	13	0.344
a3a	3185 x 122	7.56	14	0.500
a4a	4781 x 122	4.76	16	0.844
a5a	6414 x 122	3.34	17	1.125
a6a	11220 x 122	1.55	26	2.828
a7a	16100 x 122	0.84	19	2.984
a8a	22696 x 123	0.35	23	5.078

Table 7.2: Results for PCG with Wang Preconditioner.

Refactor every 2 iterations. (No Updates Applied)

Matrix	Size	IPM Iterations	PCG Iterations	Refactor	Time (sec.)
a1a	1605 x 119	12	51	6	0.750
a2a	2265 x 119	11	49	5	0.875
a3a	3185 x 122	12	58	6	1.188
a4a	4781 x 122	13	62	6	1.781
a5a	6414 x 122	14	58	7	2.453
a6a	11220 x 122	22	77	11	6.484
a7a	16100 x 122	16	67	8	7.734
a8a	22696 x 123	19	67	9	13.656

Table 7.3: Results for PCG with Wang Preconditioner.

Refactor every 2 iterations. (Apply 5 Updates)

Matrix	Size	IPM Iterations	PCG Iterations	Refactor	Time (sec.)
a1a	1605 x 119	12	57	6	0.875
a2a	2265 x 119	11	63	5	0.938
a3a	3185 x 122	12	71	6	1.328
a4a	4781 x 122	13	76	6	1.922
a5a	6414 x 122	14	70	7	2.703
a6a	11220 x 122	22	95	11	6.969
a7a	16100 x 122	16	77	8	8.391
a8a	22696 x 123	19	81	9	14.797

Table 7.4: Results for PCG with B&S Preconditioner scaled.

Refactor every 2 iterations. (No Updates Applied)

Matrix	Size	IPM Iterations	PCG Iterations	Refactor	Time (sec.)
a1a	1605 x 119	12	65	5	0.766
a2a	2265 x 119	11	51	5	0.781
a3a	3185 x 122	12	60	5	1.172
a4a	4781 x 122	13	58	6	1.625
a5a	6414 x 122	14	62	6	2.203
a6a	11220 x 122	21	77	10	4.984
a7a	16100 x 122	16	65	7	5.563
a8a	22696 x 123	19	71	9	9

Table 7.5: Results for PCG with B&S Preconditioner scaled.

Refactor every 2 iterations. (Apply 5 Updates)

Matrix	Size	IPM Iterations	PCG Iterations	Refactor	Time (sec.)
a1a	1605 x 119	12	127	5	0.906
a2a	2265 x 119	11	107	5	1.094
a3a	3185 x 122	12	108	5	1.406
a4a	4781 x 122	13	87	6	1.953
a5a	6414 x 122	14	116	6	2.734
a6a	11220 x 122	21	157	10	6.406
a7a	16100 x 122	16	96	7	6.375
a8a	22696 x 123	19	104	9	10.500

Table 7.6: Jin Jung Results for Dense Matrices

Matrix	Size	IPM Iterations	Time (sec.)
letter	20000 x 16	41	16.734
mushroom	8124 x 22	17	7.281
wave	5000 x 21	13	3.828
wavenoise	5000 x 40	12	35.016

Table 7.7: Results for PCG with Wang Preconditioner for Dense Matrices.

Refactor every 2 iterations. (No Updates Applied)

Matrix	Size	IPM Iterations	PCG Iterations	Refactor	Time (sec.)
letter	20000 x 16	39	117	19	27.422
mushroom	8124 x 22	17	45	7	6.766
wave	5000 x 21	11	46	5	3.703
wavenoise	5000 x 40	11	62	5	25.609

Table 7.8: Results for PCG with Wang Preconditioner for Dense Matrices.

Refactor every 2 iterations. (Apply 5 Updates)

Matrix	Size	IPM Iterations	PCG Iterations	Refactor	Time (sec.)
letter	20000 x 16	39	117	19	28.469
mushroom	8124 x 22	14	45	7	6.828
wave	5000 x 21	11	46	5	3.781
wavenoise	5000 x 40	11	61	5	26.109

Table 7.9: Results for PCG with B&S Preconditioner scaled for Dense Matrices.

Refactor every 2 iterations. (No Updates Applied)

Matrix	Size	IPM Iterations	PCG Iterations	Refactor	Time (sec.)
letter	20000 x 16	38	125	18	16.438
mushroom	8124 x 22	14	46	6	6.125
wave	5000 x 21	11	37	5	3.344
wavenoise	5000 x 40	11	47	5	22.859

Table 7.10: Results for PCG with B&S Preconditioner scaled for Dense Matrices.

Refactor every 2 iterations. (Apply 5 Updates)

Matrix	IPM Iterations	PCG Iterations	Refactor	Time (sec.)	
letter	20000 x 16	38	131	18	16.813
mushroom	8124 x 22	14	48	6	6.156
wave	5000 x 21	11	39	5	3.375
wavenoise	5000 x 40	11	51	5	23.359

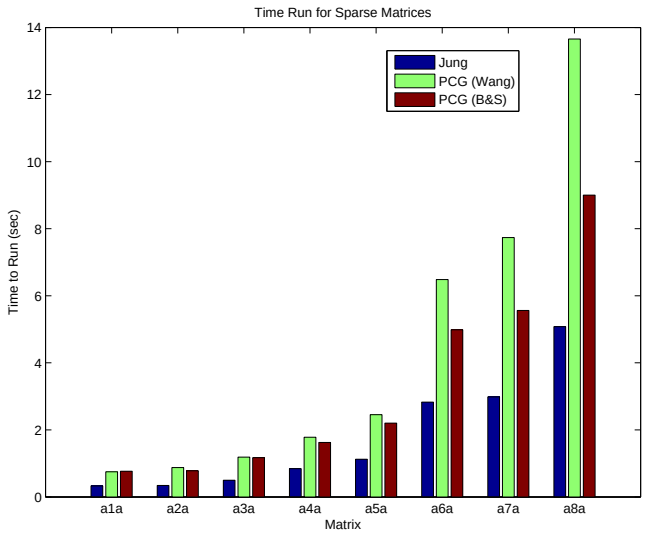

Figure 7.1: Time for the algorithms to run on sparse problems. We used the fastest times for our method which occurred when no updates were applied and we refactored the matrix every 2 iterations.

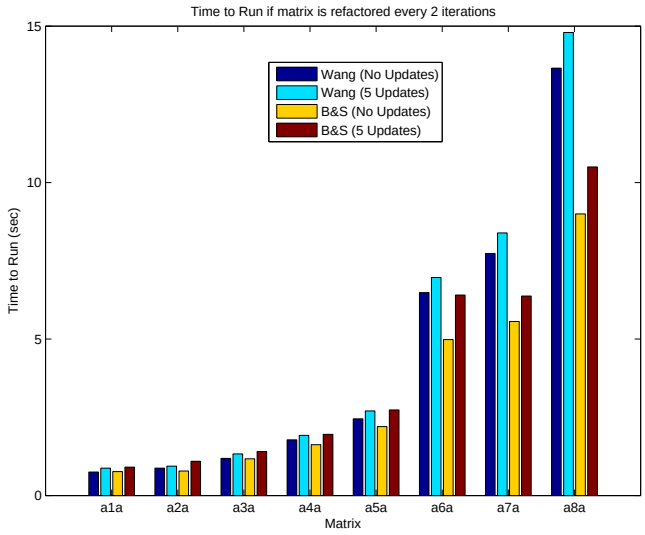

Figure 7.2: Time for the algorithms to run on sparse problems if the matrix is refactored every 2 iterations. The algorithms are tested on applying 5 updates and no updates. We can see that applying no updates to B&S preconditioner works best.

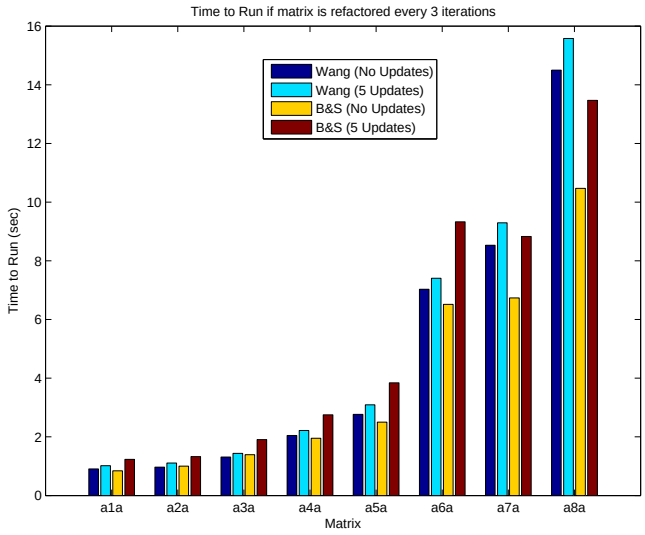

Figure 7.3: Time for the algorithms to run on sparse problems if the matrix is refactored every 3 iterations. The algorithms are tested on applying 5 updates and no updates. We can see that applying no updates to B&S preconditioner works best.

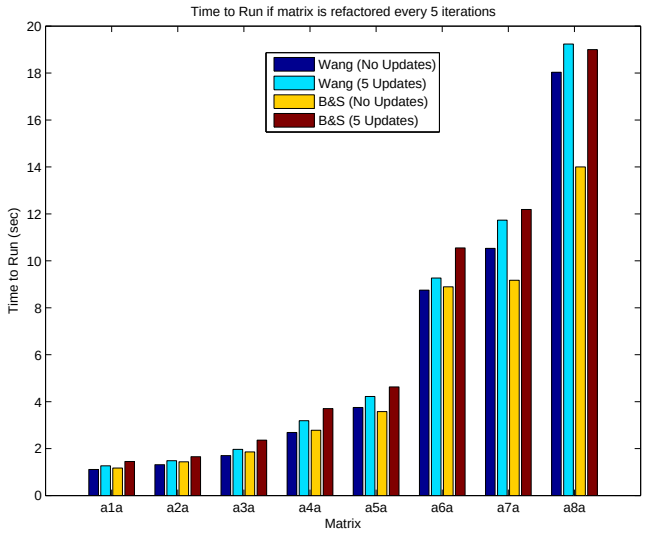

Figure 7.4: Time for the algorithms to run on sparse problems if the matrix is refactored every 5 iterations. The algorithms are tested on applying 5 updates and no updates. We can see that applying no updates to B&S preconditioner works best.

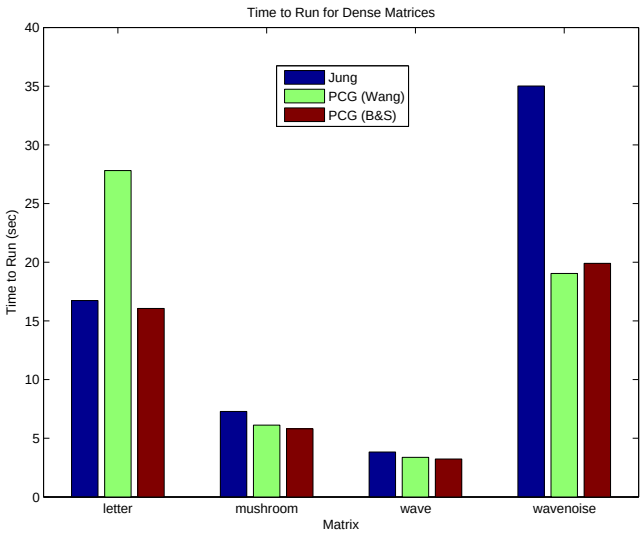

Figure 7.5: Time for the algorithms to run on dense problems. Our algorithm outperforms Jung's algorithm in every case. Also B&S preconditioner outperforms Wang's preconditioner.

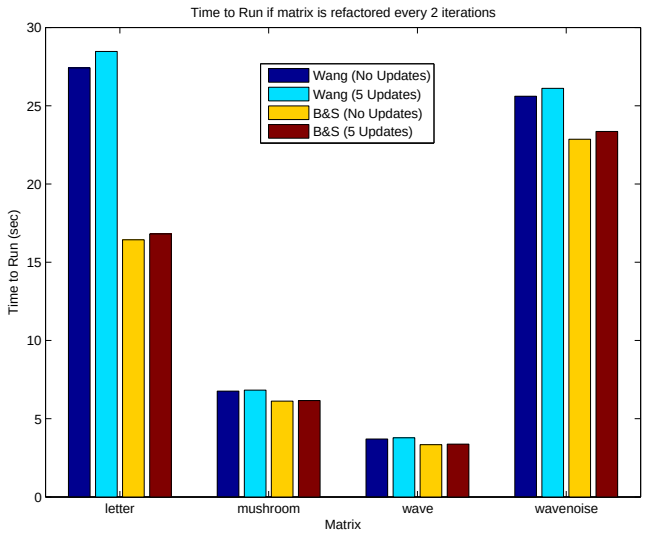

Figure 7.6: Time for the algorithms to run on dense problems if the matrix is refactored every 2 iterations. The algorithms are tested on applying 5 updates and no updates. Applying no updates to B&S' preconditioner works best.

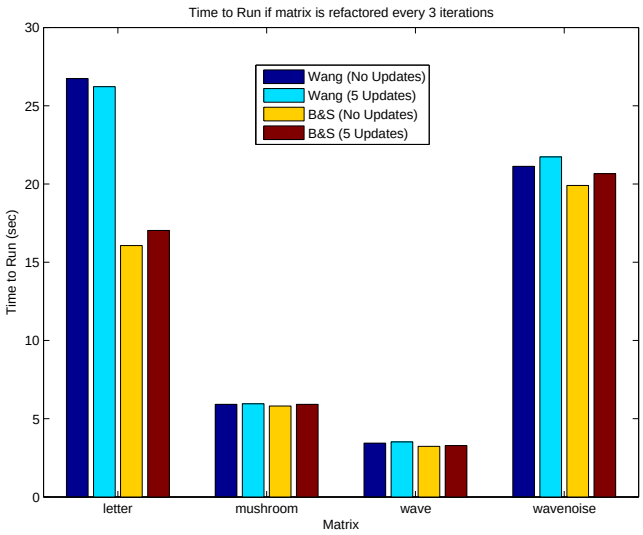

Figure 7.7: Time for the algorithms to run on dense problems if the matrix is refactored every 3 iterations. The algorithms are tested on applying 5 updates and no updates. Applying no updates to B&S' preconditioner works best.

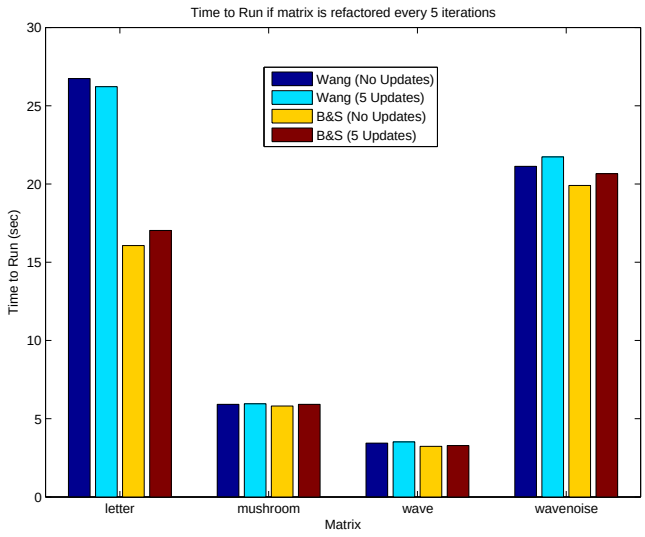

Figure 7.8: Time for the algorithms to run on dense problems if the matrix is refactored every 2 iterations. The algorithms are tested on applying 5 updates and no updates. Applying no updates to B&S' preconditioner works best.

Chapter 8

Conclusion and Future Work

Incorporating PCG and the adaptive constraint reduction method into the IPM algorithm for SVM training reduces the time to solve the system of equations for dense problems. However, the reduction in time can only be seen if only a small number of updates are applied and the matrix of normal equations is refactored every so many iterations. Also the indices to be updated should be chosen based on the ratio of the diagonal matrix between iterations. Improved performance is obtained by scaling the ratio by its mean since this gives better normalization and more elements close to 1.

Future work should focus on improving the choice of indices for update. Also, block updates should be applied to decrease the time complexity. Finally, future work should focus on all types of convex quadratic programming problems.

Although our method reduces time for dense matrices, it does not for sparse problems. This could be a consequence of overhead in the MEX interface or loss of sparsity in the updated factors. Future work should investigate the overhead and if it is large, Algorithm 4 and Algorithm 5 should be implemented in C rather than MATLAB to reduce the overhead.

Bibliography

[1] Kendall E. Atkinson, *An Introduction to Numerical Analysis, 2nd Edition*, John Wiley and Sons, 1989.

[2] Venansius Baryamureeba and Trond Steihaug, *On a Class of Preconditioners for Interior Point Methods* University of Bergen Department of Informatics. http://www.ii.uib.no/ trond/publications/proceedings/Nordic_MPS_99.ps. Date accessed: April 3, 2008.

[3] Michele Benzi, *Preconditioning Techniques for Large Sparse Systems: A Survey* Journal of Computational Physics, 182(2002), pp.418 - 477.

[4] Christopher Burges, *A Tutorial on Support Vector Machines for Pattern Recognition* Data Mining and Knowledge Discovery, 2(2):121-167, 1998.

[5] Timothy A. Davis, *User Guide for CHOLMOD: A Sparse Cholesky Factorization and Modification Package* University of Florida Department of Computer and Information Science and Engineering. http:/www.cise.ufl.edu/research/sparse/cholmod/CHOLMOD/DOC/UserGuide.pdf. Date accessed: April 7, 2008.

[6] E. Michael Gertz and Joshua D. Griffin, *Support vector machine classifiers for large data sets* Preprint, October 2005.

[7] Mark S. Gockenbach, *Understanding and Implementing the Finite Element Method* SIAM, 2006.

[8] Jin Jung, *Adaptive Constraint Reduction for Convex Quadratic Programming and Training Support Vector Machines*, Ph.D. thesis, Computer Science Department, University of Maryland, College Park, MD, 2008.

[9] Jin Hyuk Jung, Dianne P. O'Leary, and Andre' L. Tits, *Adaptive Constraint Reduction for Training Support Vector Machines* Preprint, May 2007. http://www.optimization-online.org/DB_HTML/2007/10/1820.html. Date accessed: May 1, 2008.

[10] Jonathan Richard Shewchuk, *An Introduction to Conjugate Gradient Methods without the Agonizing Pain* http://www.cs.cmu.edu/∼quake-papers/painless-conjugate-gradient.pdf, August 1994. Date accessed: April 3, 2008.

[11] Lloyd Nicholas Trefethen and David Bau III, *Numerical Linear Algebra*, SIAM, 1997.

[12] Ronny Kohavi and Barry Becker, *Adult Data Set* Data Mining and Visualization Silicon Graphics. http://archive.ics.uci.edu/ml/datasets/Adult. Date accessed: May 1, 2008.

[13] Weichung Wang and Dianne P. O'Leary, *Adaptive use of iterative methods in predictor-corrector interior point methods for linear programming* Numerical Algorithms, 25(1-4):387-406, 2000.

[14] Stephen Wright, *Primal-Dual Interior-Point Methods* SIAM, 1997.