A Comparison of Iterative Methods for Least Squares Support Vector Machine Classifiers

B. Hamers, J.A.K. Suykens, B. De Moor *

November 14, 2001

Abstract

In this paper we make a comparison with respect to the convergence speed of different training algorithms for Least Squares Support Vector Machine. We compare training algorithms based on Cholesky factorisation, successive overrelaxation and Krylov methods (conjugate gradient, block-conjugate gradient). The latter show the best performance for large datasets.

keywords: support vector machines, large scale data sets, Krylov Methods, SOR

*Departement of Electrical Engineering - ESAT/SISTA, K.U. Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium, email: bart.hamers@esat.kuleuven.ac.be, johan.suykens@esat.kuleuven.ac.be and bart.demoor@esat.kuleuven.ac.be.

1 Introduction

Support vector machines (SVM) have become an important method for solving classification problems. Several articles have shown that the performance of these methods are among the best in a variety of application areas such as optical character recognition, text classification [1], genome analysis [2], [3],... Still an important topic of research is the use of SVM's on large data sets. For the standard Vapnik SVM [4], one needs to solve a quadratic optimization problem. The dimension of this optimization problem grows with the number of training data points. When we want to train an SVM with N training data we need to solve a quadratic constrained optimization problem in N unknowns. This is computationally intensive because of the fact that the current methods used to solve such problems demand the storage of the full $N \times N$ Hessian matrix. To avoid this, the state of the art SVM algorithms use the sparseness of the solution to make a good decomposition of the original large optimization problem into smaller ones. This leads to methods of *chunking* and *decomposition* [5],[6],[7].

In this paper, we investigate several algorithms for Least Squares Support Vector Machine classification (LS-SVM), introduced in [8], that involve solving linear systems. This LS-SVM approach is a modification to the Vapnik formulation, where one avoids the inequality constraints in the QP-problem. The resulting constrained optimization problem for the LS-SVM reduces, via the Karush-Kuhn-Tucker conditions, to the solution of a set of linear equations. This gives us the opportunity to solve large problems without applying iterative chunking. The disadvantage is that the sparseness of the solution is lost. The main focus of this paper is the comparison of different iterative solvers for a LS-SVM. This comparison will be done based on the number of iterations that are needed for convergence.

We will illustrate that for small training sets the *Cholesky factorization* is still the most recommended algorithm. For large training sets, however, the iterative methods have more advantages. In this article we will mainly focus on two iterative algorithms: the *successive overrelaxation method* and the *conjugate gradient method.* For the first, the use of acceleration methods (GM–acceleration) and adaptations for memory efficiency (block-SSOR) will be shown. Although these methods give good results in reducing the number of iteration steps, they are outperformed by the Krylov methods (conjugate gradient). Because, as will be shown later, we need to solve two almost identical linear systems, we can use the *block conjugate gradient method.* This method proofs to give a faster convergence on both systems simultaneously in comparison with solving the two linear systems separately.

This paper is organized as follows. In Section 2 we will describe the basic features of the LS-SVM classifier. This will lead us to the linear system for training of the LS-SVM. An overview of the different methods to solve this linear system will be covered in Section 3. Section 4 will describe the numerical results we have obtained by training the LS-SVM based on these algorithms. These tests were done by making use of the UCI repository data sets [9].

The following notations will be used. A vector will be written in bold case $\mathbf{x} \in \mathbb{R}^d$ with x_i as its elements. The vector $\mathbf{1}_d = [1; ...; 1] \in \mathbb{R}^d$. The inner product between two vectors is defined as $\mathbf{x} \cdot \mathbf{z} = \sum_{k=1}^{d} x_k z_k$. The vector norm used in this article is the Euclidean norm, given by $\|\mathbf{x}\| = \left(\sum_{k=1}^{d} x_k^2 \right)^{1/2}$. Sometimes we will also use the ∞-norm which is defined as $\|\mathbf{x}\|_\infty = \max_i |x_i|$. Matrices are denoted by capital letters $A \in \mathbb{R}^{m \times n}$. $\|A\| = \max \left\{ \sqrt{\lambda} : \lambda \text{ is an eigenvalue of } A^T A \right\}$ (maximal singular value of A). The transpose of A is written as A^T. The identity matrix of dimension $m \times m$ is writen as I_m.

2 Least Squares Support Vector Machine Classifiers

Least Squares Support Vector Machine classifiers (LS-SVM) [8] is a modification of the standard Vapnik SVM classifier which considers equality constraints and squared loss functions. Recently also multiclass versions have been proposed for the LS-SVM [8],[10]. Because they are all based on the same idea, we will only discuss the binary version of this method.

Assume given a training set$\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ with inputs $\mathbf{x}_i \in \mathbb{R}^d$ and given class label $y_i \in \{-1, +1\}$. Training the SVM consists of finding the linear separating hypersurface $\mathbf{w} \cdot \phi(x) + b = 0$ in the so called *feature space* F (corresponding to a nonlinear separation in the original input space). This comes down to finding the appropriate vector $\mathbf{w}$ and scalar b so that the following relations hold:

$$\begin{cases} \mathbf{w} \cdot \phi(\mathbf{x}_i) + b \geq +1 \text{ if } y_i = +1 \\ \mathbf{w} \cdot \phi(\mathbf{x}_i) + b \leq -1 \text{ if } y_i = -1, \forall i \in \{1, ..., N\}. \end{cases} \tag{1}$$

The function $\phi : \mathbb{R}^d \to F$ is a non-linear mapping of the *input space* $\mathbb{R}^d$ into a higher dimensional *feature space* F (possibly infinite dimensional).

¿From a statistical learning theory viewpoint there is an additional element which is desirable to take into account. For a good generalization the distance between those two hyperplanes $\mathbf{w} \cdot \phi(x_i) + b = +1$ and $\mathbf{w} \cdot \phi(x_i) + b = -1$ in the feature space has to be maximized [4]. The margin, between the two, is given by $\frac{2}{\|\mathbf{w}\|}$. This then leads us to the constrained optimization problem

$$\begin{cases} \min\limits_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t. } y_i (\mathbf{w} \cdot \phi(\mathbf{x}_i) + b) \geq 1, \ \forall i \in \{1, ..., N\}. \end{cases} \tag{2}$$

To avoid the inequality constraints, which lead to a quadratic programming problem [11],[12], the training problem can be reformulated in another form. Instead of using the inequality constraints to find the decision hyperplane, the LS-SVM uses equality constraints. The difference is compensated by adding an extra term to the cost function. The training problem now has the form

$$\begin{cases} \min\limits_{\mathbf{w}} \frac{1}{2}\left\|\mathbf{w}\right\|^2 + \frac{\gamma}{2}\left\|\mathbf{e}\right\|^2 \\ \text{s.t. } y_i\left(\mathbf{w}\cdot\phi\left(\mathbf{x}_i\right)+b\right) = 1 - e_i,\ \forall i \in \{1,...,N\} \end{cases} \tag{3}$$

where γ plays the role of a regularization constant and $\mathbf{e} = [e_1; e_2; ...; e_N]$. The optimality conditions of this problem are given by the Karush-Kuhn-Tucker conditions [13]. With the Lagrangian given by

$$\pounds\left(\mathbf{w}, b, \mathbf{e}, \alpha\right) = \frac{1}{2}\left\|\mathbf{w}\right\|^2 + \frac{\gamma}{2}\left\|\mathbf{e}\right\|^2 - \sum_{i=1}^{N}\alpha_i\left(y_i\left(\mathbf{w}\cdot\phi\left(\mathbf{x}_i\right)+b\right) - 1 + e_i\right) \tag{4}$$

we get the following set of linear equations

$$\begin{cases} \frac{\partial \pounds}{\partial w} = 0 \rightarrow \mathbf{w} = \sum\limits_{i=1}^{N}\alpha_i y_i \phi\left(\mathbf{x}_i\right) \\ \frac{\partial \pounds}{\partial b} = 0 \rightarrow \sum\limits_{i=1}^{N}\alpha_i y_i = 0 \\ \frac{\partial \pounds}{\partial e_i} = 0 \rightarrow \alpha_i = \gamma e_i,\ \forall i \in \{1,...,N\} \\ \frac{\partial \pounds}{\partial \alpha_i} = 0 \rightarrow y_i\left(\mathbf{w}\cdot\phi\left(\mathbf{x}_i\right)+b\right) = 1 - e_i,\ \forall i \in \{1,...,N\} \end{cases} \tag{5}$$

where α is a vector of scalar Lagrangian multipliers.

The resulting classifier takes the form

$$\begin{aligned} f\left(\mathbf{x}\right) &= \operatorname{sign}\left(\mathbf{w}\cdot\phi\left(\mathbf{x}\right)+b\right) \\ &= \operatorname{sign}\left(\sum_{i=1}^{N}\alpha_i y_i K\left(\mathbf{x}, \mathbf{x}_i\right) + b\right). \end{aligned} \tag{6}$$

The *kernel function* K in this formulation is defined as $K : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R} : \left(\mathbf{x}_i, \mathbf{x}_j\right) \longmapsto K\left(\mathbf{x}_i, \mathbf{x}_j\right) = \phi\left(\mathbf{x}_i\right)\cdot\phi\left(\mathbf{x}_j\right)$ and gives the result of the inner product between two vectors in the *feature space*. There are many choices of functions to be used as kernel functions as long as they satisfy the Mercer condition. This Mercer condition guarantees that there exists an expansion for the kernel function K into a mapping ϕ. A second feature is that the matrix of all kernel evaluations of the trainingpoints is a symmetric positive definite matrix. Some commonly used kernels are:

- linear kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$
- polynomial kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i \cdot \mathbf{x}_j)^d$; $d = 1, 2, ...$
- radial basis function: $K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(\frac{-\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\sigma^2}\right)$; $\sigma \in \mathbb{R}$
- MLP-kernels: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\kappa \mathbf{x}_i \cdot \mathbf{x}_j - \delta)$; $\kappa, \delta \in \mathbb{R}$. [1]

We see that the final classification of new points only relies on the Langrange multipliers α and the bias term b. These will be found by solving the set of linear equations (5). Therefore we will first rewrite them into easier form by eliminating $\mathbf{w}$ and $\mathbf{e}$:

$$\begin{bmatrix} 0 & \mathbf{y}^T \\ \mathbf{y} & H \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1}_N \end{bmatrix} \tag{7}$$

where $\mathbf{y} = [y_1; y_2; ...; y_N]$ denotes the column vector formed by the labels of the training points. The positive definite matrix H is defined by its elements $h_{ij} = y_i y K(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij} \frac{1}{\gamma}$ where δ_{ij} denotes the Kronecker delta. However, the linear system (7) is not positive definite, which makes it more difficult to solve. This can be overcome by rewriting it into the following formulation [14]

$$\begin{bmatrix} s & 0 \\ 0 & H \end{bmatrix} \begin{bmatrix} b \\ \alpha + bH^{-1}\mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{y}^T\left(H^{-1}\mathbf{1}_N\right) \\ \mathbf{1}_N \end{bmatrix} \tag{8}$$

[1] Notice that this kernel does not satisfy the Mercer condition for all values of $\kappa, \delta \in \mathbb{R}$. (see also [11, p.21])

$$s = \mathbf{y}^T \left(H^{-1} \mathbf{y} \right) > 0. \tag{9}$$

This new linear system is positive definite. This opens many opportunities for using fast and efficient numerical optimization methods. The solution of the problem can be found in the following three step method [14]:

1. $\begin{cases} H\eta = \mathbf{y} \\ H\nu = \mathbf{1}_N \end{cases}$
2. $s = \mathbf{y} \cdot \eta$
3. $\begin{cases} b = \frac{\eta \cdot \mathbf{1}_N}{s} \\ \alpha = \nu - b\eta \end{cases}$

So, we see that the solution of the training procedure can be found by solving two sets of linear equations with the same positive definite coefficient matrix $H \in \mathbb{R}^{N \times N}$. This means that the linear systems involved in training the LS-SVM scale up equally with N, the amount of training points. Methods for solving this training procedure for large training sets will be discussed in the next section.

3 Comparison of different training algorithms

In the previous section, we have shown that training an LS-SVM involves solving two linear systems. The next important step is to find a good method for solving the linear equations as stated in step 1 of the previously mentioned training procedure. The difficulty lies in the fact that the size grows with the amount N of data points in our training set, e.g. for 10000 data points, we will have 10000 linear equations in 10000 unknown variables. Because the matrices are in general not sparse, we will have to store a lot of data in computer memory. Obviously, one will soon face practical limits if one wants to increase the number of data points. We will focus on this training step and discuss several methods that can be found in literature with respect to this problem. In the sequel the two linear systems $H\eta = \mathbf{y}$ and

$H\nu = \mathbf{1}_N$ will be denoted by $H\mathbf{x} = \mathbf{b}$ where $\mathbf{x}$ will stand for η or ν and $\mathbf{b}$ for $\mathbf{y}$ or $\mathbf{1}_N$, respectively.

3.1 Direct methods

One of the standard numerical methods for solving linear systems $H\mathbf{x} = \mathbf{b}$ is the *LU-factorization* [15, p. 92]. In this method the matrix H is written as a product of a lower triangular L and upper triangular U matrix. So $H = LU$, which then gives

$$H\mathbf{x} = LU\mathbf{x} = \mathbf{b} \Leftrightarrow \begin{cases} L\mathbf{z} = \mathbf{b} \\ U\mathbf{x} = \mathbf{z}. \end{cases} \tag{10}$$

The latter equations are easy to solve via forward and back substitution.

Since H is a symmetric positive definite matrix one can prove that there exists a unique matrix G, lower triangular with positive diagonal entries, so that $H = GG^T$. With this special feature we can rewrite the previous method:

$$H\mathbf{x} = \mathbf{b} \Leftrightarrow \begin{cases} G\mathbf{z} = \mathbf{b} \\ G^T\mathbf{x} = \mathbf{z}. \end{cases} \tag{11}$$

This method is known as the *Cholesky factorization* [15]. An advantage of these direct methods is that they always give the exact solution of the linear system, to within machine precision. A second advantage of this method is that its computation time doesn't depend on the condition number of the matrix H. The number of floating point operations needed is in the order $O\left(\frac{N^3}{3}\right)$ (N is the dimension of the system). An important disadvantage is that the matrix H has to be completely stored in memory. This is of course impossible when we go to very large matrices. Therefore, it is interesting to also consider less computationally intensive methods with smaller memory requirements. For smaller problems, up to say $N = 1000$, our advise is to invoke Cholesky factorization [2].

3.2 Iterative methods: Jacobi, Gauss-Seidel and SOR

Iterative methods are characterized by the fact that they start from an initial guess $\mathbf{x}^{(0)}$ and at each iteration step they create a new candidate solution. This sequence of iteration vectors $\{\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n)}, ...\}$ will converge to the optimal solution $\mathbf{x}^{*}$. The iterative nature, as such, is an important advantage. We can terminate the iteration process once an intermediate solution is satisfactory. We will give a small overview of the basic iterative methods [16],[17].

The first and simplest method is the *Jacobi method.* Here the components of the vectors are iteratively calculated as :

For $\quad i = 1:1:N$

$$x_i^{(k+1)} = \tfrac{1}{h_{ii}} \left(b_i - \sum_{j=1}^{i-1} h_{ij} x_j^{(k)} - \sum_{j=i+1}^{N} h_{ij} x_j^{(k)} \right)$$

end

where the i-th component of $\mathbf{x}$ after k iterations is written as $x_i^{(k)}$.

The second method is the *Gauss-Seidel* method. In this method the new information about the first $i-1$ of $\mathbf{x}^{(k+1)}$ components is used:

For $\quad i = 1:1:N$

$$x_i^{(k+1)} = \tfrac{1}{h_{ii}} \left(b_i - \sum_{j=1}^{i-1} h_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{N} h_{ij} x_j^{(k)} \right)$$

end

The third method is the Successive Overrelaxation (SOR) method:

For $\quad i = 1:1:N$

$$x_i^{(k+1)} = \tfrac{\omega}{h_{ii}} \left(b_i - \sum_{j=1}^{i-1} h_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{N} h_{ij} x_j^{(k)} \right) + (1-\omega)\, x_i^{(k)}$$

end

This method was already successfully applied for SVM training by Mangasarian and Musicant in [18]. The convergence of this method depends on the value of

[2] This turnover point is of course very speculative since it is very dependent on the computer memory and the implementation of the Cholesky factorisation.

the *overrelaxation parameter* ω. In general this parameter can be found from the minimization problem

$$\omega_{optimal} = \underset{\omega}{\operatorname{argmin}}\ \rho\left((D-\omega L)^{-1}(\omega U + (1-\omega)D)\right) \tag{12}$$

where $\rho(X)$ denotes the spectral radius of the matrix. The matrices L, D and U are the lower diagonal, diagonal and upper diagonal submatrices of H such that $H = L + D + U$.

Finding the optimal overrelaxation parameter based on this formula is rarely done because it is computationally too expensive. In our specific case, where the matrix H is symmetric and positive definite, there exist mathematical proofs [16] that the algorithm will converge for all overrelaxation parameters between $0 < \omega < 2$. But nevertheless, there may exist large differences in the convergence rate for different overrelaxation parameters.

3.2.1 Acceleration methods for SOR

A first improvement that we investigate to the SOR-algorithm is the use of the Graves-Morris (GM) Acceleration [19], a generalization of the Aitken's Δ^2 process. This method constructs a new sequence of solution vectors $\mathbf{t}^{(k)}$ based on previous iterations

$$\mathbf{t}^{(k)} = \mathbf{x}^{(k)} - \left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right)\frac{\left(\mathbf{x}^{(k-1)} - \mathbf{x}^{(k-2)}\right)\cdot\left(\mathbf{x}^{(k-1)} - \mathbf{x}^{(k-2)}\right)}{\left(\mathbf{x}^{(k-1)} - \mathbf{x}^{(k-2)}\right)\cdot\left(\mathbf{x}^{(k)} - 2\mathbf{x}^{(k-1)} + \mathbf{x}^{(k-2)}\right)}. \tag{13}$$

After the first three iterations this new sequence $\mathbf{t}^{(k)}$ will replace the original sequence. The acceleration method not only reduces the required amount of iterations for convergence, the SOR method is also less sensitive to the value of the overrelaxation parameter. The corresponding speed up is experimentally verified as will be shown in the sequel.

3.2.2 Symmetric Successive Overrelaxation (SSOR)

A second improvement is based on exploiting the structure of the matrix. Because we know that the matrix H is always symmetric, we can exploit this fact. Typical for these matrices is that one can develop a backward SOR [15, 513]. This means that we can update our solution vector in reverse direction. This will be combined with the traditional forward SOR in the following two-step procedure:

%Forward Loop

For $i = 1:1:N$

$$x_i^{\left(\frac{k+1}{2}\right)} = \frac{\omega}{h_{ii}}\left(b_i - \sum_{j=1}^{i-1} h_{ij} x_j^{\left(\frac{k+1}{2}\right)} - \sum_{j=i+1}^{N} h_{ij} x_j^{(k)}\right) + (1-\omega)\, x_i^{(k)}$$

end

%Backward Loop

For $i = N:-1:1$

$$x_i^{(k+1)} = \frac{\omega}{h_{ii}}\left(b_i - \sum_{j=1}^{i-1} h_{ij} x_j^{\left(\frac{k+1}{2}\right)} - \sum_{j=i+1}^{N} h_{ij} x_j^{(k+1)}\right) + (1-\omega)\, x_i^{\left(\frac{k+1}{2}\right)}$$

end

where $\mathbf{x}^{\left(\frac{k+1}{2}\right)}$ is an intermediate step.

3.2.3 Block Successive Overrelaxation and Block Symmetric Overrelaxation

The next step is that we will divide the matrices $H, \mathbf{x}$ and $\mathbf{b}$ in submatrices. In this way we can formulate the block-SOR [17, p78-82]. Then the linear equation $H\mathbf{x} = \mathbf{b}$ is formulated as

$$\begin{bmatrix} H_{11} & H_{12} & ... & H_{1p} \\ H_{21} & H_{22} & ... & H_{2p} \\ ... & ... & ... & ... \\ H_{p1} & H_{p2} & ... & H_{pp} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ ... \\ X_p \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ ... \\ B_p \end{bmatrix}.$$

The block-SOR is given by

For $\quad i = 1:1:N$

$$X_i^{(k+1)} = \omega H_{ii}^{-1}\left(B_i - \sum_{j=1}^{i-1} H_{ij} X_j^{(k+1)} - \sum_{j=i+1}^{N} H_{ij} X_j^{(k)}\right) + (1-\omega)\, X_i^{(k)}$$

end.

As stated in [17], the use of these block-methods can give some improvements on convergence rate, although it is not guaranteed. However, as we will discuss, we experimentally checked this property and found that it is almost always fulfilled. One has to keep in mind that when increasing the size of the blocks at each step the inverse of the block-matrix on the diagonal must be computed. For numerical stability this is done by solving a linear system via Cholesky factorization. In this sense we divide the large linear system into smaller linear systems.

Also the SSOR can be redefined using the block-formulation. The Block Symmetric Overrelaxation is given by:

%Forward Loop

For $\quad i = 1:1:N$

$$X_i^{\left(\frac{k+1}{2}\right)} = \omega H_{ii}^{-1}\left(B_i - \sum_{j=1}^{i-1} H_{ij} X_j^{\left(\frac{k+1}{2}\right)} - \sum_{j=i+1}^{N} H_{ij} X_j^{(k)}\right) + (1-\omega)\, X_i^{(k)}$$

end

%Backward Loop

For $\quad i = N:-1:1$

$$X_i^{(k+1)} = \omega H_{ii}^{-1}\left(B_i - \sum_{j=1}^{i-1} H_{ij} X_j^{\left(\frac{k+1}{2}\right)} - \sum_{j=i+1}^{N} H_{ij} X_j^{(k+1)}\right) + (1-\omega)\, X_i^{\left(\frac{k+1}{2}\right)}$$

end.

Since for large data sets we cannot store the complete matrix in memory[3], this method has the advantage of using less I/O operations in the case that we need to store the H matrix in an external storage device. Each loop of the SSOR through the matrix H can reuse the last blocks of the previous loop. This finds its cause in the intelligent switching between forward and backward propagation intrinsically

[3]Note that for data sets with 4000 data points, the matrix H needs $4000^2 * 8bit = 128Mbyte$ for storage, where we use a 8bit representation for any real number. This easily exceeds the standard Ram-memory capacity of a desktop PC.

done by the SSOR. Depending on the size of the blocks, this eliminates a big swap between hard disk and main memory of the computer.

For all these methods the convergence is proven as long as the overrelaxation parameter lies in the interval $0 < \omega < 2$.

3.2.4 Stopping criteria and convergence properties

When one solves a linear system, the exact solution is never reached because of the limited accuracy in the floating point arithmetic. Especially in iterative methods it is important to know what constitutes an acceptable approximation to the solution. Based on this knowledge a stopping criterium can be formulated. Often it is desired to have an approximate solution for which some standard norm of the error, say the two norm or ∞-norm, is less than a certain tolerance.

It would be nice if we could monitor the evolution of the *real error* $H^{-1}\mathbf{b} - \mathbf{x}^{(k)}$ during the iterations. Then it would be easy to know when the desired accuracy is reached. Unfortunately this is impossible, the only information we have is the residual $\mathbf{r}^{(k)} = \mathbf{b} - H\mathbf{x}^{(k)}$. But we can estimate the real desired error for iterative methods using the residual by the following rule which gives the relation between the relative error norm and the relative residual norm [20, p. 107]

$$\frac{1}{\kappa(H)} \frac{\left\|\mathbf{b} - H\mathbf{x}^{(k)}\right\|}{\|\mathbf{b}\|} \leq \frac{\left\|H^{-1}\mathbf{b} - \mathbf{x}^{(k)}\right\|}{\left\|H^{-1}\mathbf{b}\right\|} \leq \kappa(H) \frac{\left\|\mathbf{b} - H\mathbf{x}^{(k)}\right\|}{\|\mathbf{b}\|}$$

where $\kappa(H) = \left\|H^{-1}\right\| \|H\|$ the condition number of matrix H. Given a tolerance ε one can always iterate until the desired accuracy on $\frac{\left\|\mathbf{r}^{(k)}\right\|}{\left\|\mathbf{r}^{(0)}\right\|} = \frac{\left\|\mathbf{b}-H\mathbf{x}^{(k)}\right\|}{\|\mathbf{b}\|}$ is reached[4]. Controlling this residual then gives a lower and upper limit on the real error. A stopping criterium based on this idea will be used in the Krylov-methods we will discuss in the next Section 3.3.3.

[4]In all the case we use $\mathbf{x}^{(0)} = \mathbf{0}$ as the starting point of the iterative procedure.

Another stopping criterium, as defined in most publications and books on numerical techniques [16],[21],[17] is based on the differences $\Delta x_i^{(k)} = x_i^{(k)} - x_i^{(k-1)}$. In the beginning of the process a parameter ε is defined to control the accuracy of the algorithm. The algorithm will stop iterating at step k when $\max_i \left|\Delta x_i^{(k)}\right| < \varepsilon$. This correspond to the ∞-norm $\left\|\Delta \mathbf{x}^{(k)}\right\|_\infty = \max_i \left|\Delta x_i^{(k)}\right|$. Computationally this rule is very easy because it doesn't need any matrix-vector evaluation as would be the case if we would use the residual $\mathbf{r}^{(k)} = \mathbf{b} - H\mathbf{x}^{(k)}$. Although for some iterative methods it is known that this rule can lead to termination before the desired level of accuracy is achieved. Our experiments showed that this rule is sufficient. The experiments showed that demanding a high accuracy takes much computation time on large matrices. For our applications however, very high accuracy for the solution vector is not always needed.

3.3 Iterative methods: Krylov methods

An important group of iterative methods to solve linear systems are Krylov methods. The *conjugate gradient* is most suitable for matrices with a positive definite linear system [20],[13],[15]. Its use was already experimentally verified in [14] and [22]. The two linear systems $H\eta = \mathbf{y}$ and $H\nu = \mathbf{1}_N$ are solved separately. The idea is that the solution of the linear system is also the minimum of a quadratic convex cost function :

$$\mathbf{x}^* = \min_{\mathbf{x}} J(\mathbf{x}) = \min \ \frac{1}{2}\mathbf{x} \cdot (H\mathbf{x}) - \mathbf{b} \cdot \mathbf{x} \Rightarrow H\mathbf{x}^* = \mathbf{b}.$$

The search procedure for this minimum is done via the conjugate directions of the matrix H. This leads to the following algorithm to solve $H\mathbf{x} = \mathbf{b}$:

Input $\mathbf{x}^{(0)}$, $\mathbf{r}^{(0)} = H\mathbf{x} - \mathbf{b}$, $\mathbf{p}^{(0)} = -\mathbf{r}^{(0)}$

While $\mathbf{r}^{(k)} \neq 0$

$\alpha^{(k)} = \frac{\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}}{\mathbf{p}^{(k)} \cdot H\mathbf{p}^{(k)}}$

$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$

$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \alpha^{(k)}H\mathbf{p}^{(k)}$

$\beta^{(k)} = \frac{\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}}$

$\mathbf{p}^{(k+1)} = -\mathbf{r}^{(k)} + \beta^{(k+1)}\mathbf{p}^{(k)}$

$k = k + 1$

end

In this formulation, respectively $\mathbf{r}^{(k)}$ and $\mathbf{p}^{(k)}$ are the residuals and conjugate directions updated at each iteration step. The most demanding part in this algorithm is the matrix-vector product $H\mathbf{p}^{(k)}$. This deserves some extra attention. Remember also here that again for large applications this matrix H cannot be stored in main memory. Therefore the matrix-vector computation is done row by row where each time the individual rows of H are recomputed. This demands the smallest memory requirements and therefor can be computed on low-end personal computers. There are of course more intelligent procedures to do this matrix-vector multiplication by exploiting the symmetric structure of the matrix H. A basic procedure would be storing those elements of each row that come back in the next rows because of the symmetric property of the matrix. This avoids the recomputation of those elements or the I/O operation of reloading then into the main memory from an external storage device. But these procedures need, obviously, more memory. A second possibility to improve this computationally intensive step is computing the matrix-vector product on a parallel or distributed machine.

3.3.1 Convergence Properties of CG

Since every loop through the algorithm is very time consuming, we want to keep control over the number of iterations that are needed for convergence. This convergence of the CG algorithm is theoretically guaranteed in at most N steps. This is however a conservative upper limit. The convergence of the CG algorithm is very dependent on the eigenvalue spectrum and condition number of H. In [13, p. 114] a more intuitive explanation is given. If H has only r distinct eigenvalues, then the CG iteration will terminate at the solution in at most r iterations.

In our application we often notice that only a few eigenvalues are distinct, which is related to the geometrical distribution of the training data.

Also the condition number $\kappa(H)$ can give us some more insight in the convergence. As is stated in [21, p. 300], if the condition number $\kappa(H)$ is large but not too large, the convergence to a specified convergence can be expected in $O(\sqrt{\kappa})$ iterations. But remember that this also is only an upperbound. Convergence will be faster if the eigenvalue spectrum is clustered.

A second important aspect is the dependency of the condition number on the hyperparameters γ and σ. This can be easily seen since $H(\gamma, \sigma) = K(\sigma) + \frac{1}{\gamma} I_N$ (in case of the RBF kernel). These hyperparameters do not only influence the generalization capability of the SVM but also the convergence rate of the training procedure. We can see in Figure 1 that there is a large influence of the γ parameter on the condition number.

3.3.2 Block Conjugate Gradient

Since the conjugate gradient directions for both linear systems only depend on H, both linear systems $H\eta = \mathbf{y}, H\nu = \mathbf{1}_N$ can be solved at ones: $H\begin{bmatrix}\eta & \nu\end{bmatrix} = \begin{bmatrix}\mathbf{y} & \mathbf{1}_N\end{bmatrix}$. This method is called the *block conjugate gradient method* [20, p. 113],[23].

The algorithm to solve $HX = B$ where $H \in \mathbb{R}^{n \times n}$; $B = \begin{bmatrix}\mathbf{y} & \mathbf{1}_N\end{bmatrix}$, $X = \begin{bmatrix}\eta & \nu\end{bmatrix} \in \mathbb{R}^{n \times 2}$ is given by:

Input $X^{(0)}, R^{(0)} = HX^{(k)} - B, P^{(k)} = -R^{(k)}$

While $R^{(k)} \neq 0$

$\alpha^{(k)} = \left(P^{(k)T} A P^{(k)}\right)^{-1} R^{(k)T} R^{(k)}$

$$X^{(k+1)} = X^{(k)} + P^{(k)}\alpha^{(k)}$$

$$R^{(k+1)} = R^{(k)} + HP^{(k)}\alpha^{(k)}$$

$$\beta^{(k)} = \left(R^{(k)T}R^{(k)}\right)^{-1} R^{(k+1)T}R^{(k+1)}$$

$$P^{(k+1)} = -R^{(k+1)} + P^{(k)}\beta^{(k+1)}$$

$$k = k + 1$$

end

Notice that the algorithm is only well-defined if the matrices $P^{(k)}$ and $R^{(k)}$ posses full rank. Because $span\left\{P^{(0)}, ..., P^{(k)}\right\} = span\left\{R^{(0)}, ..., R^{(k)}\right\}$ the ranks of both matrices P_k and R_k are the same. Thus we can monitor the stability of the algorithm by calculating the rank of the matrix $P^{(k)T}\ P^{(k)}$. As long as this matrix maintains its full rank, the stability is guaranteed. If not, which means that columns become linearly dependent, the equations corresponding to dependent columns can be treated separately. The algorithm will be continued with the remaining equations.

The block conjugate gradient is in some occasions to be preferred over CG. The following arguments were given in [23]:

1. To solve the two systems the algorithm will need at most $n/2$ iterations and may involve less work then solving the two systems separately.

2. If the matrix has several eigenvalues widely separated from the others, the block CG will converge significantly faster then the CG.

3. If the matrix is stored on a secondary device or need to be regenerated at every use, the block CG can be implemented much more efficient.

These features were experimentally verified and will be discussed in the next Section.

3.3.3 The stopping criterium for CG and Block-CG algorithm

We previously explained for the SSOR (3.2.4) that a good stopping criterium is essential in every iterative method because of the limited machine accuracy. In [22] is described that the optimal stopping criterium for our application is a combined set of rules that keep track of the change in the residual vector $\mathbf{r}^{(k)}$ and the evolution of the corresponding cost function $J(\mathbf{x})$. The CG-algorithm will stop the iteration when at least one of the following three criteria has been reached:

- $k \geq k_{\max}$
- $\left\|\mathbf{r}^{(k)}\right\| \leq \varepsilon_1 \left\|\mathbf{r}^{(0)}\right\|$
- $J\left(\mathbf{x}^{(k-1)}\right) - J\left(\mathbf{x}^{(k)}\right) \leq \varepsilon_2.$

The last criterium is not often mentioned in literature. In our experiments we often noticed that the evolution of the norm of residual $\mathbf{r}^{(k)}$ is usually not monotonically decreasing. The evolution of $J(\mathbf{x})$ does monotonically decrease and can be very easily monitored by the following reformulation: $J\left(\mathbf{x}^{(k)}\right) = \frac{1}{2}\mathbf{x}^{(k)} \cdot \left(H\mathbf{x}^{(k)} - \mathbf{b}\right) - \frac{1}{2}\mathbf{x}^{(k)} \cdot \mathbf{b} = -\frac{1}{2}\mathbf{x}^{(k)}\left(\mathbf{r}^{(k)} - \mathbf{b}\right)$. The constants $k_{\max}, \varepsilon_1$ and ε_2 are determined by the user according to required numerical accuracy. In most application $\varepsilon_1 = \varepsilon_2 = 10^{-9}$ is sufficient. For the block-CG this criteria are applied to both systems separately.

4 Numerical Results

4.1 Implementation details

All the algorithms were implemented and tested in matlab6.0. The computation of the matrix H, which is a bottleneck in all the algorithms, was done in C and the communication via the CMEX interface of matlab. The tests were done on an AMD Athlon 500Mhz processor with 256Mb RAM running MS Windows NT 4.0. The C-code compilation was done with MS Visual C++ compiler. The computationally intensive tests on the adult data set were done on a dual Pentium III 850 MHz processor and 1Gb RAM running Linux. The C-code there was compiled with gcc. In the implementation we reduced the needed memory as much as possible. For the SSOR method this means that we stored only those rows of H in memory that were needed by the block-SSOR. The CG-algorithm did the matrix-vector multiplication in a row by row manner. At each step the matrix H is recomputed.

The hyperparameter selection for the different kernels is done on the basis of the tests performed and reported in [22]. The classification performance of all the tested methods is therefore equal. The data sets that we used are obtained from the UCI benchmark repository [9]. The starting point $\mathbf{x}^{(0)}$ of the iterative procedures (SOR, SSOR, CG, Block-CG,...) was always set equal to $\mathbf{0}$. The following data sets are used: the Statlog Australian Credit (acr), the Bupa Liver disorder (bld), the Statlog German credit (gcr), the Statlog heart disease (hea), the John Hopkins university ionosphere (ion), the Pima Indians diabetes (pid), the sonar (snr), the tic-tac-toe endgame (ttt), the Wisconsin breast cancer (wbc) and the adult dataset (adu). All their characteristics can be found in Table 1. N stands for the total number of data points in the data set. This data set is divided in N_{train} training points and N_{test} test points. The dimensions of the input are given by d, which can be divided in d_{num} numerical attributes and d_{cat} categorical ones according to the data sets.

Although all the UCI data sets, except the adult data set, are rather small data sets, they reflect very well the ideas we want to point out in this paper. Normally we would advice to train the LS-SVM with a Cholesky factorization on these data sets. Here, however, also the other training procedures are used on these data sets to verify the methods.

4.2 Experimental results of the SOR methods

Here we describe the experimental results we performed on the statlog heart disease data set from the UCI benchmark repository [9]. It is a data set with 270 data points and 13 attributes. We used 180 points to train the LS-SVM and the other 90 points are used as test set. We trained the LS-SVM with both the linear kernel and the RBF kernel. In both occasions we used the optimal hyperparameters γ and σ as stated in [22]. We trained the LS-SVM with a training algorithm based on the block-SSOR and the block-SSOR with GM-acceleration and this for different block-sizes. The overrelaxation parameter was set $\omega = 1.0$. Improvements in convergence migth still be made by adapting this parameter. Remember that for each training two sets of linear equations have to be solved. To avoid the fact that the matrix H has to be consulted in two separate algorithms, we implemented the SSOR such that the two linear systems $H\eta = \mathbf{y}$ and $H\nu = \mathbf{1}_N$ are computed simultaneously. The results are given in Figure 2.

As criterium to compare the algorithms we used the number of iteration steps before convergence upon a certain tolerance. This parameter best reflects the true computational cost that is needed by the algorithms. This parameter is also not dependent on implementation techniques nor hardware. The number of iteration steps is equal to the number of times that the matrix H has to be consulted. For large applications, where the matrix H cannot be stored in main memory, every consultation of H corresponds to an I/O operation to external storage devices or a recomputation of the matrix. Because of the forward and backward loop made by the SSOR, each iteration taken by the algorithm corresponds to two consultations of H.

In Figure 2 we see that the block-size reduces the number of iterations needed to converge. This influence is largest on the classic SSOR method. The disadvantage is that the algorithm will need more memory. Notice that for large data sets the block-size compared to the dimensions of H decreases because of memory limitations. This unfortunately increases the number of iterations that are needed. The SSOR with GM-acceleration is less dependent on the block-size. The speedup in number of iterations accomplished by the GM-acceleration is largest for small block-sizes.

The minimum number of iterations needed to converge is 95 in the case of the RBF kernel with a block size of 60. This corresponds to 190 consultations of the H matrix with a block size which is 1/3 of the actual dimensions of the linear system. This is a large number compared to the Krylov-methods as we will see in the next section.

4.3 Comparison between block-SSOR and CG

To compare the performance of the training routines based on the SSOR with GM-acceleration and the one based on the CG, we trained the LS-SVM with both methods on the UCI-data sets. We trained the LS-SVM with the linear kernel and the optimal hyperparameters. The overrelaxation parameter was set $\omega = 1.0$. The block-size was set equal to 1 so that both algorithm needed the same amount of memory. The training algorithms iterated upon a tolerance of 10^{-10}. Results of the number of iterations that are needed, is given in Table 2.

We see that the training method based on the CG-algorithm perform better in all the tested data sets. As stated in the previous chapter, better performance for the SSOR can be achieved if we use other overrelaxation parameters. But the choice of this optimal overrelaxation parameter in function of the data set is still an open problem.

4.4 Comparison between CG and Block-CG

In this section we make a comparison of both Krylov methods, the conjugate gradient (3.3.1) and the block conjugate gradient (3.3.2). Therefore we trained the

LS-SVM on the UCI data set with training algorithm based on both methods. We trained with three different kernels: linear, polynomial and RBF. The needed number of iterations upon a tolerance of 10^{-10} are given in Table 3.

In Table 3 we show that the block-CG method (BCG) has most of the time a higher convergence rate compared to the CG-method for the non-linear kernels. In case of a linear kernel it still is preferable to use a training algorithm based on the CG. The performances on the polynomial kernel show an average decrease of 20% in needed number of iterations. In the best occasion, the Bupa Liver disorder data set, we can even notice a decrease of 55% . The tests performed on the adult dataset show an advantage in number of iterations of the BCG compared to the CG for every kernel.

Figure 3a and Figure 3b show the convergence properies of both algorithm CG and BCG. The tests were performed on the adult dataset with 10000 training points and the RBF kernel. We plotted in Figure 3a the evolution of the log $J(\mathbf{x})$ for both linear systems individualy $H\eta = \mathbf{y}$ and $H\nu = \mathbf{1}_N$ as CG1 and CG2 respectively for the conjugate gradient, BCG1 and BCG2 for the block conjugate gradient. The same is done for the resudue of those linear systems in Figure 3b. These figures clearly show that for the same nummer of iterations the BCG has a smaller residue and a steeper descrease of the $J(\mathbf{x})$, which results in a faster convergence.

5 Conclusions

In this paper we have presented different iterative methods for LS-SVM classifier. The training of the LS-SVM consists of solving a set of linear systems. Because of the symmetric positive definite structure of these linear systems, many efficient numerical algorithms exist. We showed that towards large data sets, the iterative methods (SOR,CG,...) have many advantages. A first advantage is that they can be implemented in a very memory efficient way. Second, they are in most cases much faster than the direct methods (Cholesky Factorization) for large data sets.

The first iterative method we tested was the successive overrelaxation method. Exploiting the structure in combination with a block-implementation of the SSOR gave a more memory efficient algorithm. We also showed that the GM-acceleration reduces the needed amount of iterations before convergence. The extra advantage is that its dependency on the block size and the overrelaxation parameter is reduced. This GM-acceleration combined with a block-SSOR gives many advantages toward large data sets compared with the classical SOR method.

If one compares the block-SSOR (with GM acceleration) with the conjugate gradient (CG) method, this last one outperforms the block-SSOR in all cases. A second Krylov method that we tested besides CG was the block-CG algorithm. This method solves the two linear systems, that are needed to train the LS-SVM, at ones. In most tests we performed on the UCI-data sets, the block-CG method converges faster for the non-linear classification tasks.

Acknowledgements: B. Hamers is a Research Assistant with the I.W.T. (Flemish Institute for Scientific and Technological Research in Industry). J.A.K. Suykens is a Postdoctoral Researcher with the F.W.O. (Fund for Scientific Research-Flanders). This work was supported by grants and funding from the Research Council of the K.U.Leuven (PhD/postdoc grants, IDO , GOA Mefisto-666), the Foundation for Scientific Research Flanders (FWO)(PhD and postdoc grants and projects (G.0115.01, G.0197.02, G.0407.02), research communities (ICCoS, ANMMM)), the Flemish regional government (Bilateral Coll., IWT-project Soft4s, Eureka projects Synopsis, Impact, FLiTE, STWW-project Genprom, GBOU-project McKnow, IWT PhD grants), the Belgian Federal Government (DWTC, IUAP IV-02, Sustainable Development MD/01/024), the European Commission (TMR Alapedes, Ernsi) and industry supported direct contract research (Electrabel, ELIA, ISMC, Data4s).

References

[1] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *Proceedings of the European Conference on Machine Learning* (C. Nédellec and C. Rouveirol, eds.), (Berlin), pp. 137–142, Springer, 1998.

[2] M. Brown, W. Grundy, D. Lin, N. Cristianini, C. Sugnet, M. Ares, and D. Haussler, "Support vector machine classification of microarray gene expression data," *Proceedings of the National Academy of Sciences*, vol. 97, no. 1, pp. 262–267, 2000.

[3] S. Mukherjee, P. Tamayo, J. Mesirov, D. Slonim, A. Verri, and T. Poggio, "Support vector machine classification of microarray data," Tech. Rep. 182, AI Memo 1676, CBCL, 1999.

[4] V. Vapnik, *The nature of statistical learning theory.* New York: Springer-Verlag, 1995.

[5] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," in *Neural Networks for Signal Processing VII — Proceedings of the 1997 IEEE Workshop* (J. Principe, L. Giles, N. Morgan, and E. Wilson, eds.), (New York), pp. 276–285, IEEE, 1997.

[6] T. Joachims, "Making large–scale SVM learning practical," in *Advances in Kernel Methods — Support Vector Learning* (B. Schölkopf, C. J. C. Burges, and A. J. Smola, eds.), (Cambridge, MA), pp. 169–184, MIT Press, 1999.

[7] J. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods — Support Vector Learning* (B. Schölkopf, C. J. C. Burges, and A. J. Smola, eds.), (Cambridge, MA), pp. 185–208, MIT Press, 1999.

[8] J. Suykens and J. Vandewalle, “Least squares support vector machine classifier,” *Neural Processing Letters*, vol. 9, pp. 293–300, Jun 1999.

[9] C. Blake and C. Merz, “UCI repository of machine learning databases.” http://www.ics.uci.edu/mlearn/MLRepository.html.

[10] T. Van Gestel, J. Suykens, G. Lanckriet, A. Lambrechts, and B. De Moor, “Multiclass LS-SVMs : Moderated outputs and coding-decoding schemes,” *Neural Processing Letters*, 2001. in press.

[11] C. Burges, “A tutorial on support vector machines for pattern recognition,” *Knowledge Discovery and Data Mining*, vol. 2, no. 2, 1998.

[12] N. Cristianini and J. Shawe-Taylor, *An introduction to Support Vector Machines and other kernel-based learning methods.* Cambridge: Cambridge University Press, 2000.

[13] J. Nocedal and S. Wright, *Numerical Optimization.* New York: Springer-Verlag, 1999.

[14] J. Suykens, P. Van Dooren, B. De Moor, and J. Vandewalle, “Least squares support vector machine classifiers: a large scale algorithm,” *European Conference on Circuit Theory and Design, ECCTD'99*, pp. 839–842, 1999.

[15] G. Golub and C. V. Loan, *Matrix Computations.* Baltimore: The John Hopkins University Press, 2nd ed., 1990.

[16] D. Young, *Iterative solution of large linear systems.* New York: Academic Press, 1971.

[17] R. Varga, *Matrix Iterative Analysis.* Englewood Cliffs: Prentice-Hall, 1962.

[18] O. Mangasarian and D. Musicant, “Successive overrelaxation for support vector machines,” *IEEE Transactions on Neural Networks*, vol. 10, pp. 1032–1037, 1999.

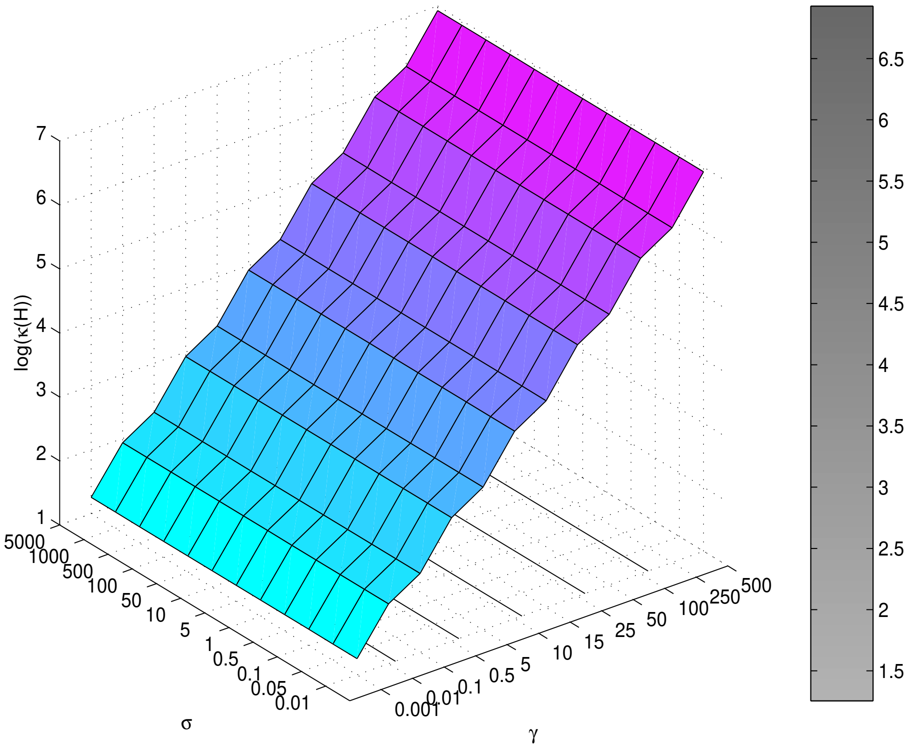
[19] R. da Cunha and T. Hopkins, "A comparison of acceleration techniques applied to the SOR method," Tech. Rep. 4-94*, University of Kent, Computing Laboratory, March 1994.

[20] A. Greenbaum, *Iterative Methods for Solving Linear Systems.* Philadelphia: SIAM, 1997.

[21] L. Trefethen and D. Bau, *Numerical Linear Algebra.* Philadelphia: SIAM, 1997.

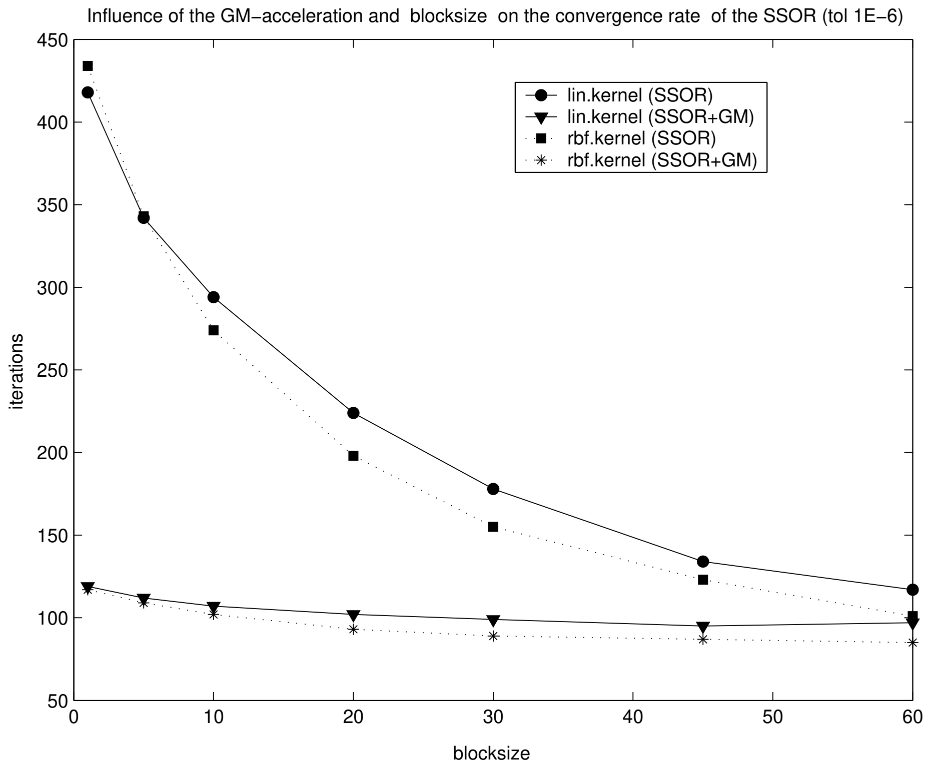
[22] T. Van Gestel, J. Suykens, B. Baesens, S. Viaene, J. Vanthienen, G. Dedene, B. De Moor, and J. Vandewalle, "Benchmarking least squares support vector machine classifiers," Tech. Rep. 00-37, University of Leuven, dept. ESAT/SISTA, Belgium, 1999. submitted for publication.

[23] D. O'Leary, "The block conjugate gradient algorithm and related methods," *Linear Algebra and its Applications*, vol. 29, pp. 293–322, 1980.



log(κ(H))
7
6
5
4
3
2
1
5000
1000
500
100
50
10
5
1
0.5
0.1
0.05
0.01
σ
0.001
0.01
0.1
0.5
5
10
15
25
50
100
250
500
γ
6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5

Figure 1



Influence of the GM−acceleration and blocksize on the convergence rate of the SSOR (tol 1E−6)
lin.kernel (SSOR)
lin.kernel (SSOR+GM)
rbf.kernel (SSOR)
rbf.kernel (SSOR+GM)
iterations
blocksize
450
400
350
300
250
200
150
100
50
0
10
20
30
40
50
60

Figure 2

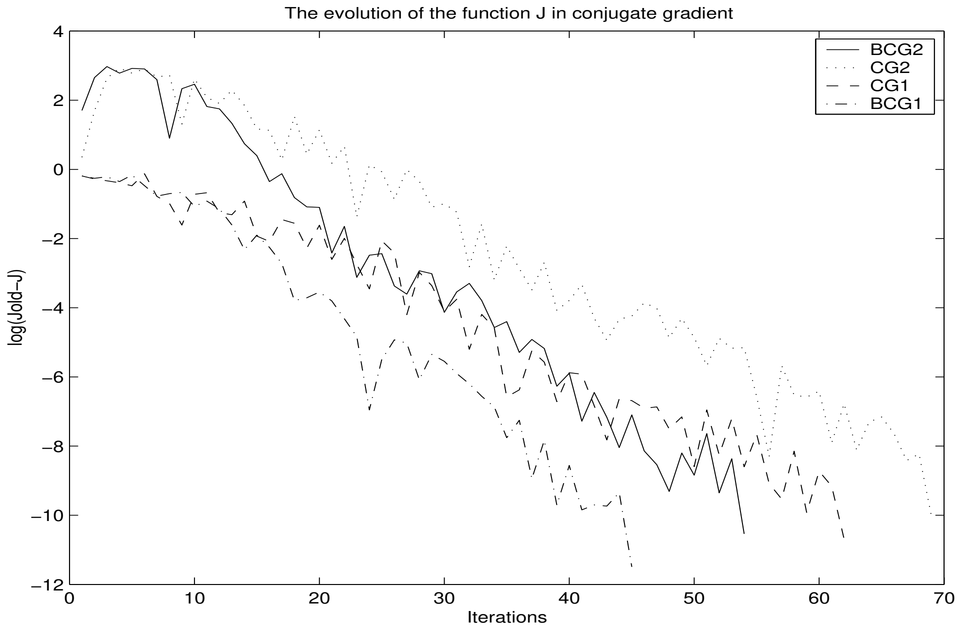


Figure 3a

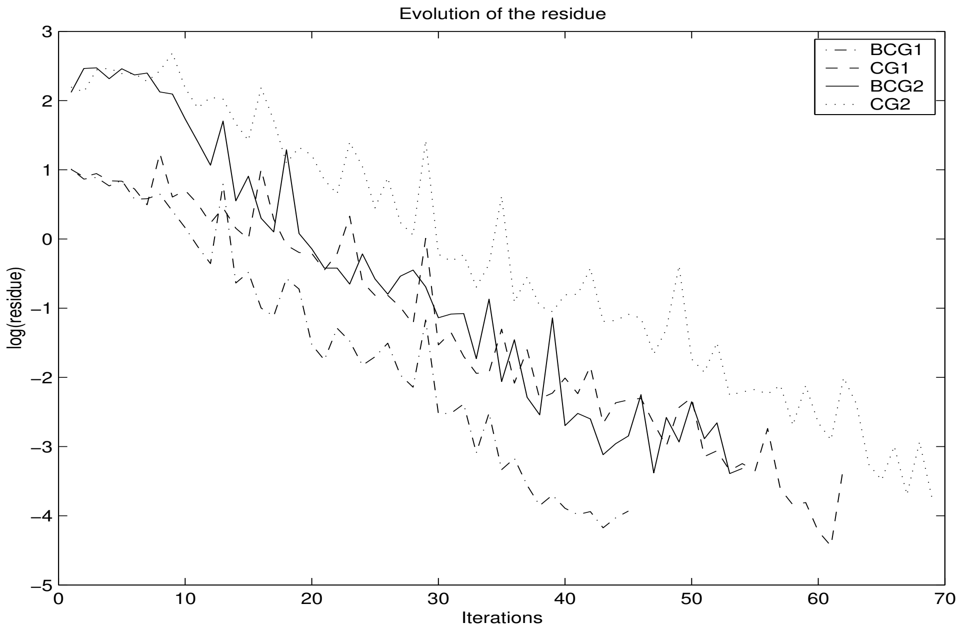


Figure 3b

Captions of figures

Figure 1: This figure shows the dependency of the condition number of the H matrix on the hyperparameters γ and σ. This plot is made by using the German Credit data sets (grc) with 666 training data points and the RBF kernel. The $\log(\kappa(H))$ is plotted with respect to γ and σ.

Figure 2: In this figure we show the number of iterations that are needed for the training algorithms on the statlog Heart disease data set (hea) with linear and RBF kernel. The training algorithms are based on the Block-SSOR and block-SSOR algorithm with GM-acceleration. The training is done with variable block sizes.

Figure 3: This figure compares the convergence properties of the CG and Block-CG tested on the the adult-dataset with the RBF-kernel. (We took a subsample of 10000 datapoints of the total dataset.) *Part a* shows the logarithm of the evolution of the J function used in the CG and Block-CG algorithm with respect to the number of the iterations. We make a distinction between to linear systems that have to be solved for training the LS-SVM. In *part b* the logarithm of the residue is plotted with respect to the number of iterations for all four linear systems.

	N_{train}	N_{test}	N	d_{num}	d_{cat}	d
acr	460	230	690	6	8	14
bld	230	115	345	6	0	6
gcr	666	334	1000	7	13	20
hea	180	90	270	7	6	13
ion	234	117	351	33	0	33
pid	512	256	768	8	0	8
snr	138	70	208	60	0	60
ttt	638	320	958	0	9	9
wbc	455	228	683	9	0	9
adu	33000	12222	45222	6	8	14

Table 1:

	SSOR+GM	CG
acr	55	13
bld	21157	8
grc	298	17
hea	169	14
ion	197	20
pid	44339	9
snr	60	19
ttt	16	9
wbc	3183	11

Table 2:

	Lin CG	BCG	Pol CG	BCG	RBF CG	BCG
arc	13	21	29	22	14	12
bld	8	5	42	19	43	164
gcr	17	47	9	9	66	75
hea	14	26	19	16	11	10
ion	20	39	12	11	33	26
pid	9	74	18	15	16	9
snr	19	21	6	6	20	16
ttt	9	11	35	23	25	36
wbc	11	43	21	15	12	9
adu	11	8	29	22	69	54

Table 3:

Caption of Tables

Table 1: Characteristics of the UCI data sets.

Table 2: Comparison of needed number of iteration steps for linear kernel between the conjugate gradient algorithm and the SSOR with GM-acceleration.

Table 3: Comparison of needed number of iteration steps for the linear, polynomial and RBF kernel between the conjugate gradient and block conjugate gradient method. The results on the adult data set are based on training with a subset of 10000 datapoints.