Part III

Implementation
Chapter 9

Implementation Techniques

In the previous chapters it was showed that the training of Support Vector Machines can be reduced to maximizing a convex quadratic function with subject to linear constraints (see chapter 5.5.1). Such convex quadratic functions have only one local maxima (the global one) and their solution can always be found efficiently. Furthermore the dual representation of the problem showed how the training could be successfully performed even in very high dimensional feature spaces.

The problem of minimizing differentiable functions of many variables has been widely studied, especially in the convex case, and most of the standard techniques can be directly applied to SVM training. However there exist specific techniques to exploit particular features of this problem. For example the large size of the training set is a formidable obstacle to a direct use of standard techniques, since just storing the kernel matrix requires a memory space that grows quadratically with the sample size.

9.1 General Techniques

A number of optimization techniques have been devised over the years, and many of them can be directly applied to quadratic programs. As examples think of the Newton method, conjugate gradient or the primal dual interior-point methods. They can be applied to the case of Support Vector Machines straightforward. Not only this, they can also be considerably simplified because of the fact, that the specific structure of the objective function is given.

Conceptually they are not very different from the simple gradient ascent\(^\text{16}\) strategy, known from the Neural Networks. But many of this techniques require that the kernel matrix is stored completely in memory. The quadratic form in (5.18) involves a matrix that has a number of elements equal to the square of the number of training examples. This matrix then e.g. cannot fit into a memory of size 128 Megabytes if there are more than 4000 training examples (assuming each element is stored as an 8-byte double precision number). So for large size problems the approaches de-

\(^{16}\) For an adaption to SVMs, see [Nel00]
scribed above can be inefficient or even impossible. So they are used in conjunction with the so called decomposition techniques (“Chunking and Decomposition”, for explanation see [Nel00]). The main idea behind this methods is to subsequently optimize only a small subset of the problem in each iteration.

The main advantages of such techniques is that they are well understood and widely available in a number of commercial and freeware packages. These were mainly used for Support Vector Machines before special algorithms were developed. The most common algorithms were, for example, the MINOS package from the Stanford Optimization Laboratory (hybrid strategy) and the LOQO package (primal dual interior-point method). In contrast to these, the quadratic program subroutine $qp$ provided in the MATLAB optimization toolbox is very general but the routine $quadprog$ is significantly better than $qp$.

9.2 Sequential Minimal Optimization (SMO)

The algorithm used in nearly any implementation of SVMs in a slightly changed manner and in the one of this diploma thesis, too, is the SMO algorithm. It was developed by John C. Platt [Pla00] and its main advantage besides being one of the most competitive is the fact that it is simple to implement.

The idea behind this algorithm is derived by taking the idea of the decomposition method to its extreme and optimizing a minimal subset of just two points at each iteration. The power of this approach resides in the fact that the optimization problem for two data points admits an analytical solution, eliminating the need to use an iterative quadratic program optimizer as part of the algorithm. So SMO breaks the large QP problem into a series of smallest possible QP problems and solves them analytically, which avoids using a time-consuming numerical QP optimization as an inner loop. Therefore the amount of memory required for SMO is linear in the training set size, no more quadratically, which allows SMO to handle very large training sets. The computation time of SMO is mainly dominated by SVM evaluation, which will be seen below.

The smallest possible subset for optimization involves two Lagrange multipliers, because the multipliers must obey the linear equality constraint (of 5.20) $\sum_{i=1}^{l} \alpha_i y_i = 0$ and therefore updating one multiplier $\alpha_k$, at least one other multiplier $\alpha_p$ ($k \neq p$, and $0 < k, p \leq l$) has to be adjusted in order to keep the condition true.
At every step, SMO chooses two Lagrange multipliers to jointly optimize, finds the optimal values for them, and updates the SVM to reflect the new optimal values.

So the advantage of SMO, to repeat it again, lies in the fact that solving for two Lagrange multipliers can be done analytically. Thus, an entire inner iteration due to numerical QP optimization is avoided. Even though more optimization sub-problems are solved now, each sub-problem is so fast solvable, such that the overall QP problem can be solved quickly (comparison between the most commonly used methods can be found in [Pla00]).

In addition, SMO does not require extra matrix storage (ignoring the minor amounts of memory required to store any 2x2 matrices required by SMO). Thus, very large SVM training problems can fit even inside of the memory of an ordinary personal computer.

The SMO algorithm mainly consists of three components:

- An analytic method to solve for the two Lagrange multipliers
- A heuristic for choosing which multipliers to optimize
- A method for computing the bias $b$

As even mentioned in chapter 5.2.1 the computation of the bias $b$ can be problematic, when simply taking the average value for $b$ after summing up all calculated $b$'s for each i. This was shown by [Ker01]. The main problem arising when using an averaged value of the bias for recalculation in the SMO algorithm is, that the convergence speed of it is not guaranteed. Sometimes it is slower and sometimes it is faster. So Keerthi suggested an improvement for the SMO algorithm where two threshold values $b_{up}$ and $b_{low}$ are used instead of one. It has been shown in this paper that the modified SMO algorithm is more efficient on any tested dataset in contrast to the original one. The speed-up is significant!

But as a first introduction the original SMO algorithm will be used here and can be extended later. Before continuing, one disadvantage of the SMO algorithm should be stated here. In the original form implemented in nearly any toolbox, it cannot handle the 2-norm case. Because the KT-conditions are others, as can be seen in chapter 5.5.2. Therefore nearly any toolbox, which wants to implement the 2-norm case, uses optimization techniques mentioned above. Only one implements the 1- and 2-norm case at the same time with an extended form of the SMO algorithm (LibSVM by Chih-Jen Lin). The 2-norm case will also be added to the developed SMO algorithm in this diploma thesis.

As it will be seen, SMO will spend most of the time evaluating the decision function, rather than performing QP, it can exploit data sets which contain substantial number of zero elements. Such sets will be called *sparse.*
9.2.1 Solving for two Lagrange Multipliers

First recall the general mathematical formulated problem:

\[
\begin{align*}
\text{Maximize} & \quad L_0(w, b, \xi, \alpha, \beta) = W(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j \langle x_i \cdot x_j \rangle \\
\text{subject to} & \quad \sum_{i=1}^{l} \alpha_i y_i = 0 \\
& \quad 0 \leq \alpha_i \leq C \quad i = 1 \ldots l
\end{align*}
\]

With the following KT conditions fulfilled, if the QP problem is solved for all \(i\) (for maximal-margin and 1-norm):

\[
\begin{align*}
0 & \leq \alpha_i \leq C \\
\alpha_i &= 0 \implies y_i (\langle w^\top \cdot x_i \rangle + b^\top) \geq 1 \\
0 & < \alpha_i < C \implies y_i (\langle w^\top \cdot x_i \rangle + b^\top) = 1 \\
\alpha_i &= C \implies y_i (\langle w^\top \cdot x_i \rangle + b^\top) \leq 1
\end{align*}
\]

For convenience, all quantities referring to the first multiplier will have a subscript 1 and those referring to the second a subscript 2. Without the other subscript “old”, they are meant to be the just optimized values “new”.

For initializing \(\alpha^{\text{old}}\) is set to zero.

In order to take the step to the overall solution two \(\alpha_i\)’s are picked and SMO calculates the constraints on these two multipliers and then solves for the constrained maximum. Because there are only two constraints now, they can be easily displayed in two dimension (see figure 9.1). The constraints \(0 \leq \alpha_i \leq C\) cause the Lagrange multipliers to lie inside a box, while the linear equality constraint \(\sum_{i=1}^{l} \alpha_i y_i = 0\) causes them to lie on a diagonal line. Thus, the constrained maximum of the objective function \(W(\alpha)\) must lie on a diagonal line segment (explanation in figure 9.1 and following pages).

In other words, to not violate the linear constraint on the two multipliers they must fulfil: \(\alpha_1 y_1 + \alpha_2 y_2 = \text{const.} = \alpha_1^{\text{old}} y_1 + \alpha_2^{\text{old}} y_2\) (lie on a line) in the box constrained by \(0 \leq \alpha_1, \alpha_2 \leq C\).

So this one-dimensional problem resulting from the restriction of the objective function to such a line can be solved analytically.
Figure 9.1: Two cases of optimization: \( y_1 \neq y_2 \) and \( y_1 = y_2 \). The two Lagrange multipliers chosen for subset optimization must fulfill all of the constraints of the full problem. The inequality constraints cause them to lie inside a box and the linear equality constraint causes them to lie on a diagonal line. Therefore, one step of SMO must find an optimum of the objective function on a diagonal line segment. In this figure, \( \gamma = \alpha_1^{\text{old}} + m\alpha_2^{\text{old}} \), which is a constant that depends on the previous values of \( \alpha_1, \alpha_2 \) and \( m = y_1 y_2 = \{1, -1\} \). (\( \gamma = \alpha_1 + m\alpha_2 = \alpha_1^{\text{old}} + m\alpha_2^{\text{old}} \))

Without loss of generality, the algorithm first computes the second multiplier \( \alpha_2 \) and computes the ends of the diagonal line segments in terms of this one. So it is successively used to obtain \( \alpha_1 \). The bounds on the new multiplier \( \alpha_2 \) can be formulated more restrictive with use of the linear constraint and the equality constraint (also see figure 9.2). But first recall for each \( \alpha_i \): \( 0 \leq \alpha_i \leq C \) and also the linear constraint has to hold:

\[
\sum_{i=1}^{l} \alpha_i y_i = 0.
\]

Using the two actual multipliers to be optimized we write

\[
\alpha_1 y_1 + \alpha_2 y_2 + \sum_{i=3}^{l} \alpha_i y_i \quad \text{and therefore} \quad \alpha_1 y_1 + \alpha_2 y_2 = \gamma \quad \text{where} \quad \gamma = -\sum_{i=3}^{l} \alpha_i y_i.
\]

There are two cases to consider (remember \( y_i \in \{-1, +1\} \)):
Case 1: \( y_1 \neq y_2 \) then \( \alpha_1 - \alpha_2 = \gamma \)  \hspace{1cm} (9.1)

Case 2: \( y_1 = y_2 \) then \( \alpha_1 + \alpha_2 = \gamma \)  \hspace{1cm} (9.2)

Then let \( m = y_i y_2 \), then the two above equations can be written as

\[
\alpha_1 + m\alpha_2 = \gamma
\]  \hspace{1cm} (9.3)

and before optimization \( \gamma = \alpha_1^{old} + m\alpha_2^{old} \).

Then the end points of the searched diagonal line (figure 9.2 and 9.3) can be expressed with help of the old, possibly not optimized values:

Case 1: \( y_1 \neq y_2 \)

\[
\alpha_1^{old} - \alpha_2^{old} = \gamma
\]

L (\( \alpha_2 \) at the lower end point) is:

\[
\max (0, -\gamma) = \max (0, \alpha_2^{old} - \alpha_1^{old})
\]

H (\( \alpha_2 \) at the higher end point) is:

\[
\min (C, C - \gamma) = \min (C, C + \alpha_2^{old} - \alpha_1^{old})
\]
where, if $y_1 \neq y_2$:

$$L = \max (0, \alpha_2^{\text{old}} - \alpha_1^{\text{old}})$$

$$H = \min (C, C - \alpha_1^{\text{old}} + \alpha_2^{\text{old}})$$

and if $y_1 = y_2$:

$$L = \max (0, \alpha_1^{\text{old}} + \alpha_2^{\text{old}} - C)$$

$$H = \min (C, \alpha_1^{\text{old}} + \alpha_2^{\text{old}})$$

(9.4)
\[ L \leq \alpha_2 \leq H \]

At a first glance, this only appears to be applicable to the 1-norm case, but treating C as infinite for the hard-margin case reduces the constraints on the interval \([L, H]\):

\[ L \leq \alpha_2 \leq H \]

where, if \( y_1 \neq y_2 \):

\[ L = \max (0, \alpha_2^{\text{old}} - \alpha_1^{\text{old}}) \]

only lower bounded and if \( y_1 = y_2 \):

\[ L = 0 \quad H = \alpha_1^{\text{old}} + \alpha_2^{\text{old}} \]

Now that the other \( \alpha_i \)'s are assumed fixed, the objective function \( W(\alpha_1, \alpha_2) = L_D \) can be rewritten (as abbreviation \( \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle = \mathbf{x}_i^T \mathbf{x}_j \) is written here as \( \mathbf{x}_i \mathbf{x}_j \)):

\[
L_D = \alpha_1 + \alpha_2 + \text{const.} - \frac{1}{2} (\alpha_1^2 y_1 y_1 \mathbf{x}_1 \mathbf{x}_1 + \alpha_2^2 y_2 y_2 \mathbf{x}_2 \mathbf{x}_2 + 2\alpha_1 \alpha_2 y_1 y_2 \mathbf{x}_1 \mathbf{x}_2) + 2(\sum_{i=3}^{I} \alpha_i y_i \mathbf{x}_i \mathbf{x}_i) (\alpha_1 y_1 \mathbf{x}_1 + \alpha_2 y_2 \mathbf{x}_2) + \text{const.})
\]

“const.” are the parts dependable on the multipliers not optimized in this step, so they are regarded as constant values simply added. Now for simplification assume the following substitutions:

\[ K_{11} = \mathbf{x}_1 \mathbf{x}_1, \quad K_{22} = \mathbf{x}_2 \mathbf{x}_2, \quad K_{12} = \mathbf{x}_1 \mathbf{x}_2 \] and

\[ v_j = \sum_{i=3}^{I} \alpha_i y_i \mathbf{x}_i \mathbf{x}_j \]

As in figure 9.1, assume \( m = y_1 y_2 \) and with the equality constraint we get

\[ \alpha_1 y_1 + \alpha_2 y_2 = \text{const.}, \] multiplied with \( y_1 \), leading to \( \alpha_1 = \gamma - ma_2 (y_1 y_1 = 1) \)

where \( \gamma \equiv \alpha_1 + ma_2 = \alpha_1^{\text{old}} + ma_2^{\text{old}} \).

And resubstituting all these relations back into \( L_D \) the formula becomes:
\[ L_D = \alpha_1 + \alpha_2 - \frac{1}{2} (\alpha_1^2 K_{11} + \alpha_2^2 K_{22} + 2m\alpha_1\alpha_2 K_{12} + 2\alpha_1\nu_1 y_1 + 2\alpha_2\nu_2 y_2) + \text{const.} \]

Where \( \text{const.} \) is \( \sum_{i=3}^{i} \alpha_i - \frac{1}{2} \sum_{i,j=3}^{i,j} \alpha_i \alpha_j y_j K_{ij} \)

And by using the help of \( \gamma \) to only have a function dependable on \( \alpha_2 \):

\[
\begin{align*}
L_D &= \gamma - m\alpha_2 + \alpha_2 - \frac{1}{2} (K_{11}(\gamma - m\alpha_2)^2 + K_{22}\alpha_2^2 + 2mK_{12}(\gamma - m\alpha_2)\alpha_2 \\
&\quad + 2\nu_1 y_1(\gamma - m\alpha_2) + 2\nu_2 y_2\alpha_2) + \text{const.} \\
&= \gamma - m\alpha_2 + \alpha_2 - \frac{1}{2} K_{11}\gamma^2 + K_{11}m\gamma\alpha_2 - \frac{1}{2} K_{11}\alpha_2^2 - \frac{1}{2} K_{22}\alpha_2^2 - mK_{12}\gamma\alpha_2 \\
&\quad + K_{12}\alpha_2^2 - \nu_1 y_1\gamma + \nu_1 y_1 m\alpha_2 - \nu_2 y_2\alpha_2 + \text{const.} \\
&= W(\alpha_2)
\end{align*}
\]

To find the maximum of this function there is need for the first and second derivate of \( W \) with respect to \( \alpha_2 \):

\[
\frac{\partial W}{\partial \alpha_2} = -m + 1 + m\gamma K_{11} - K_{11}\alpha_2 - K_{22}\alpha_2 - m\gamma K_{12} + 2K_{12}\alpha_2 + m\nu_1 y_1 - \nu_2 y_2
\]

\[
= -m + 1 + mK_{11}(\gamma - m\alpha_2) - K_{22}\alpha_2 + K_{12}\alpha_2 - mK_{12}(\gamma - m\alpha_2) + y_2 (\nu_1 - \nu_2)
\]

where \( m\gamma_1 = y_1 y_2 y_1 = y_1^2 y_2 = y_2 \)

\[
\frac{\partial^2 W}{\partial^2 \alpha_2} = 2K_{12} - K_{11} - K_{22} = \eta
\]

The following new notation will simplify the statement. \( f(x) \) is the current hypothesis function \( y_i((w \cdot x_i) + b) \) determined by the values of the actual vector \( \alpha \) and the bias \( b \) at a particular stage of learning. So the following new introduced element \( E \) is the difference between the function output (classification by the up to now trained machine) and the target classification (given by the supervisor in the training set) on the training points \( x_i \) or \( x_j \). Meaning this is the training error on the \( i \)th example.

\[
E_i = f(x_i) - y_i = u_i - y_i = (\sum_{j=1}^{j} \alpha_j y_j K(x_j, x_i) + b) - y_i ; i = 1, 2
\]
This value may be large even if a point is correctly classified. As an example if \( y_i = 1 \) and the function output is \( f(x_i) = 5 \), the classification is correct but \( E_1 = 4 \).

Recall the substitution \( v_j = \sum_{i=3}^{l} \alpha_i y_i x_i x_j \) so from 9.5 \( u_i \) is written as:

\[
\begin{align*}
\alpha_v &= \sum_{j=3}^{l} \alpha_j y_j K_{1j} - b \\
&= \sum_{j=3}^{l} \alpha_j y_j K_{1j} + \alpha_{1\text{old}} y_i K_{11} + \alpha_{2\text{old}} y_2 K_{12} - b \\
&= v_1 - b + \alpha_{1\text{old}} y_i K_{11} + \alpha_{2\text{old}} y_2 K_{12}
\end{align*}
\]

and so

\[
v_1 = u_1 + b - \alpha_{2\text{old}} y_2 K_{12} - \alpha_{1\text{old}} y_i K_{11}
\]

\[
= u_1 + b - \alpha_{2\text{old}} y_2 K_{12} - m y_2 \alpha_{1\text{old}} K_{11}
\]

(9.6)

\[
u_2 = u_2 + b - \alpha_{1\text{old}} y_i K_{12} - \alpha_{2\text{old}} y_2 K_{22}
\]

(9.7)

At the maximal point the first derivate \( \frac{\partial W}{\partial \alpha_2} \) is zero and the second one has to be negative. Hence

\[
\alpha_2 (K_{11} + K_{22} - K_{12}) = m (K_{11} - K_{12}) y + y_2 (v_1 - v_2) + 1 - m
\]

And with equations 9.6 and 9.7 this becomes (remember \( m^2 = 1 \) and \( y_i^2 = 1 \)):

\[
\alpha_2 K_{11} + \alpha_2 K_{22} - 2\alpha_2 K_{12} = 1 - m + y_2 (v_1 - v_2) + m y (K_{11} - K_{12})
\]
\((-\eta)\alpha_2 = m\gamma(K_{11} - K_{12}) + y_2(u_1 + b - u_2 - b) - \alpha_2^{old} K_{12} - m\alpha_1^{old} K_{11} + m\alpha_1^{old} K_{12} + \alpha_2^{old} K_{22} + y_2^2 - y_1 y_2 \)
\(= m(\alpha_1^{old} + \alpha_2^{old})(K_{11} - K_{12}) - \alpha_2^{old} K_{12} + m\alpha_1^{old} K_{12} - m\alpha_1^{old} K_{11} + \alpha_2^{old} K_{22} + y_2(u_1 - u_2 + y_2)\)
\(= m\alpha_1^{old} K_{11} - m\alpha_1^{old} K_{12} + \alpha_2^{old} K_{11} - \alpha_2^{old} K_{12} - \alpha_2^{old} K_{12} + m\alpha_1^{old} K_{12} - m\alpha_1^{old} K_{11} + \alpha_2^{old} K_{22} + y_2(u_1 - u_2 + y_2)\)
\(= \alpha_2^{old} (-2K_{12} + K_{11} + K_{22}) + y_2(E_1 - E_2)\)
\(= (-\eta)\alpha_2^{old} + y_2(E_1 - E_2)\)

So the new multiplier \(\alpha_2\) can be expressed as:

\[
\alpha_2^{new} = \alpha_2^{old} - \frac{y_2(E_1 - E_2)}{\eta} \tag{9.8}
\]

This is the unconstrained maximum, so this has to be constrained to lie within the ends of the diagonal line, meaning \(L \leq \alpha_2^{new} \leq H\) (see figure 9.1):

\[
\alpha_2^{new,clipped} = \begin{cases} 
H & \text{if } \alpha_2^{new} \geq H \\
\alpha_2^{new} & \text{if } L < \alpha_2^{new} < H \\
L & \text{if } \alpha_2^{new} \leq L 
\end{cases} \tag{9.9}
\]

The value of \(\alpha_1^{new}\) is obtained from equation \(\alpha_1^{new} + m\alpha_2^{new,clipped} = \alpha_1^{old} + m\alpha_2^{old}\) and therefore

\[
\alpha_1^{new} = \alpha_1^{old} + m(\alpha_2^{old} - \alpha_2^{new,clipped}) \tag{9.10}
\]

As stated above, the second derivate has to be negative to ensure a maximum. But under unusual circumstances it will not be negative. A zero second derivate can occur if more than one training example has the same input vector \(x\). In any event, SMO will work even if the second deri-
vate is not negative, in which case the objective function $W$ should be evaluated at each end of the line segment. Then SMO uses the Lagrange multipliers at the end point, which yields the highest value of the objective function. These circumstances are regarded and “solved” in the next subchapter about choosing the Lagrange multipliers to be optimized.

9.2.2 Heuristics for choosing which Lagrange Multipliers to optimize

The SMO algorithm is based on the evaluation of the KT conditions. Because when every multiplier fulfills these conditions of the problem, the solution is found. These KT conditions normally are verified to a certain tolerance level $\epsilon$. As Platt mentioned in his paper, the value of $\epsilon$ is typically in the range of $10^{-2}$ to $10^{-3}$ implying that e.g. outputs on the positive (+1) margin are between 0.999 and 1.001. Normally this tolerance is enough when using an SVM for recognition. Applying higher accuracy the algorithm will not converge very fast.

There are two heuristics used for choosing the two multipliers to optimize. The choice of the first heuristic for $\alpha^{\text{old}}_1$ provides the outer loop of the SMO algorithm. This loop first iterates over the entire training set, determining whether an example violates the KT conditions. If so, then this example is immediately chosen for optimization. The second example, and therefore the candidate for $\alpha^{\text{old}}_2$ is found by the second choice heuristic and then these two multipliers are jointly optimized. At the end of this optimization, the SVM is updated and the algorithm resumes iterating over the training examples looking for KT violators. To speed up the training, the outer loop does not always iterate over the entire training set. After one pass through the training set, the outer loop only iterates those examples whose Lagrange multipliers are neither 0 nor $C$ (the non-bound examples). Then again, each example is checked against the KT conditions, and violating ones are chosen for immediate optimization and update. So the outer loop makes repeated passes over the non-bound examples until all of them obtain the KT conditions within the tolerance level $\epsilon$. Then the outer loop iterates over the whole training set again to find violators. So all in all the outer loop keeps alternating between single passes over the whole training set and multiple passes over the non-bound subset until the entire set obeys the KT condition within the tolerance level $\epsilon$. At this point the algorithm terminates.

Once the first Lagrange multiplier to be optimized is chosen, the second one has to be found. The heuristic for this one is based on maximizing the step that can be taken during joint optimization. Evaluating the Kernel function for doing so will be time-consuming, so SMO uses an approximation on the step size by using equation (9.8). So the maximum possible step size is the one having the biggest value $\|E_1 - E_2\|$. To speed up, a
cached error value E is kept for every non-bound example from which SMO chooses the one to approximately maximize the step size. If $E_1$ is positive, then the example with minimum error $E_2$ is chosen. If $E_1$ is negative, then the example with largest error $E_2$ is chosen.

Under unusual circumstances, as the ones remarked at the end of the last sub-chapter (two identical training vectors), SMO cannot make positive progress using this second choice heuristic. To avoid this, SMO uses a hierarchy of second choice heuristics until it finds a pair of multipliers, making positive progress. If there is no positive progress using above approximation, the algorithm starts iterating through the non-bound examples at a random position. If none of them makes positive progress the algorithm starts iterating through the entire training set at a random position to find a suitable multiplier $\alpha_2^{old}$ that will make positive progress in the joint optimization. The randomness in choosing the starting position is used to avoid bias towards examples stored at the beginning of the training set. In very extreme degenerative cases, a second multiplier making positive progress cannot be found. In such cases the first multiplier is skipped and a new one is chosen.

### 9.2.3 Updating the threshold b and the Error Cache

Since the solving for the Lagrange multipliers does not determine the threshold $b$ of the SVM, and there is need for updating the value of the error cache $E$ at the end of each optimization step, the value of $b$ has to be re-evaluated after each optimization. So $b$ is re-computed after each step, so that the KT conditions are fulfilled for both optimized examples.

Now let $u_1$ be the output of the SVM with the old $\alpha_i$ and $\alpha_2$:

$$u_1 = \alpha_1^{old} y,K_1 + \alpha_2^{old} y_2 K_{12} + \sum_{i=3}^{l} \alpha_i y K_{yi} - b^{old} \tag{9.11}$$

$$u_1 - E_1 = y_1 \tag{9.12}$$

As in figure 9.4, if the new $\alpha_1$ is not at the bounds, then the output of the SVM after optimization on example 1 will be $y_1$, its label value. And therefore:

$$y_1 = \alpha_1^{new} y_1 K_{11} + \alpha_2^{new,clipped} y_2 K_{12} + \sum_{i=3}^{l} \alpha_i y K_{yi} - b_1 \tag{9.13}$$

And substituting (9.13) and (9.11) into (9.12):
\[
\begin{align*}
    b_1 &= E_1 + b_1^{\text{old}} + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})K_{11} + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})K_{12} \\
    b_2 &= E_2 + b_2^{\text{old}} + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})K_{12} + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})K_{22}
\end{align*}
\]

(9.14)

Similarly obtaining an equation for \( b_2 \), such that the output of the SVM after optimization is \( y_2 \) when \( \alpha_2 \) is not at the bounds:

\[
\begin{align*}
    b_2 &= E_2 + b_2^{\text{old}} + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})K_{12} + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})K_{22}
\end{align*}
\]

(9.15)

When both \( b_1 \) and \( b_2 \) are valid, they are equal (see figure 9.4 again).

When both new calculated Lagrange multipliers are at the bound and if \( L \) is not equal to \( H \), then the interval \([b_1, b_2]\) describes all threshold consistent with the KT conditions. Then SMO chooses \( b \) to be \( b^{\text{new}} = \frac{b_1 + b_2}{2} \).

This formula is only valid, if \( b \) is subtracted from the weighted sum of the kernels, not added. If one multiplier is at the bound and the other one not, then the value of \( b \) calculated using the non-bound multiplier is used as the new updated threshold. As mentioned above, this step is regarded as problematic by [Ker01]. But to avoid this, the original SMO algorithm discussed here has to be modified in its' whole and therefore only a reference to the improved algorithm is given here. The modified pseudo code will be stated together with the original one in the appendix.

As seen in the former chapter, a cached error value \( E \) is kept for every example whose Lagrange multiplier is neither zero nor \( C \) (non-bound). So if a Lagrange multiplier is non-bound after being optimized, its' cached error is zero (it is classified correctly). Whenever a joint optimization occurs, the stored error of the other not involved multipliers have to be updated using the following equation:

\[
E_i^{\text{new}} - E_i^{\text{old}} = u_i^{\text{new}} - u_i^{\text{old}}
\]

\[
E_i^{\text{new}} = E_i^{\text{old}} + u_i^{\text{new}} - u_i^{\text{old}}
\]

And re-substituted this becomes:

\[
E_i^{\text{new}} = E_i^{\text{old}} + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})K_{11} + y_2(\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}})K_{22} + b^{\text{old}} - b^{\text{new}}
\]

(9.16)
Figure 9.4: Threshold \( b \) when both \( \alpha \)'s are bound (== \( C \)). The support vectors A and B give the same threshold \( b \), that is the distance of the optimal separating hyperplane from the origin. Point D and E give \( b_1 \) and \( b_2 \) respectively. They are error points within the margin. The searched \( b \) is somewhere between \( b_1 \) and \( b_2 \).

Overall, when an error value \( E \) is required by the SMO algorithm, it will look it up in the error cache if the corresponding Lagrange multiplier is not at bound. Otherwise, it will evaluate the current SVM decision function (classify the given point and compare it to the given label) based on the current \( \alpha \)'s.

### 9.2.4 Speeding up SMO

There are certain points in the SMO algorithm, where some useful techniques can be considered to speed up the calculation. As said in the summary on linear SVM, it is possible there to store the weight vector directly, rather than all of the training examples that correspond to non-zero Lagrange multipliers. But this optimization is only possible for the linear kernel. After the joint optimization succeeded, the stored weight vector must be updated to reflect the new Lagrange multipliers found. This update is easy, due to the linearity of the SVM:

\[
\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + y_1 (\alpha_1^{\text{new}} - \alpha_1^{\text{old}}) \mathbf{x}_1 + y_2 (\alpha_2^{\text{new,clipped}} - \alpha_2^{\text{old}}) \mathbf{x}_2
\]
This is a speed-up because much of the computation time in SMO is spent to evaluate the decision function, and therefore speeding up the decision function speeds up SMO. Another optimization that can be made is using the sparseness of the input vectors. Normally, an input vector is stored as a vector of floating point numbers. A sparse input vector (with zeros in it) is stored by the meaning of two arrays: \texttt{id} and \texttt{val}. The \texttt{id} array is an integer array storing the location of the non-zero inputs, while the \texttt{val} array is a floating point array storing the corresponding non-zero values. Then the very often used computation of the dot product between such stored vectors (\texttt{id1}, \texttt{val1}, length=\texttt{num1}) and (\texttt{id2}, \texttt{val2}, length=\texttt{num2}) can be done quickly, as shown in the pseudo code below:

\begin{verbatim}
    p1 = 0, p2 = 0, dot = 0

    while (p1 < num1 && p2 < num2)
    {
        a1 = id1[p1], a2 = id2[p2]

        if (a1 == a2)
        {
            dot += val1[p1]*val2[p2]
            p1++, p2++
        }

        else if (a1 > a2)
        p2++

        else
        p1++
    }
\end{verbatim}

This can be used to calculate linear and polynomial kernels directly. Gaussian kernels can also use this optimization through the usage of the following identity:

\[
\|x - y\|^2 = x \cdot x - 2x \cdot y + y \cdot y
\]

To speed up more in the Gaussian case, for every input the dot product with itself can be pre-computed.

Another optimization technique for linear SVMs regards the weight vector again. Because it is not stored as a sparse array, the dot product of the weight vector with a sparse input vector (\texttt{id}, \texttt{val}) can be expressed as:
\[
\sum_{i=0}^{num} w[i][j] \cdot \text{val}[i]
\]

And for binary inputs storing the array \text{val} is not even necessary, since it is always 1. Therefore the dot product calculation in the pseudo code above becomes a simple increment and for a linear SVM the dot product of the weight vector with a sparse input vector becomes:

\[
\sum_{i=0}^{num} w[i][j]
\]

As mentioned in Platt's paper there are more speed-up techniques that can be used but they will not be discussed in detail here.

### 9.2.5 The improved SMO algorithm by Keerthi

In his paper [Ker01] Keerthi points out some difficulties encountered in the original SMO algorithm by explicitly using the threshold \(b\) for checking the KT conditions. His modified algorithm will be stated here as Pseudo-Code with a little explanation, but for further details please refer to Keerthi's paper.

Keerthi uses some new notations:

Define \(F_i = \langle w \cdot x_i \rangle - y_i\). Now the KT conditions can be expressed as:

\[
\begin{align*}
\alpha_i &= 0 \quad \Rightarrow y_i (F_i - b) \geq 0 \\
0 < \alpha_i < C &\quad \Rightarrow y_i (F_i - b) \approx 0 \\
\alpha_i &= C \quad \Rightarrow y_i (F_i - b) \leq 0
\end{align*}
\]

and these can be written as:

\[
\begin{align*}
i &\in l_0 \cup l_1 \cup l_2 \quad \Rightarrow F_i \geq b \\
i &\in l_0 \cup l_3 \cup l_4 \quad \Rightarrow F_i \leq b
\end{align*}
\]

where

\[
\begin{align*}
l_0 &\equiv \{i : 0 < \alpha_i < C\} \\
l_1 &\equiv \{i : y_i = +1, \alpha_i = 0\} \\
l_2 &\equiv \{i : y_i = -1, \alpha_i = C\} \\
l_3 &\equiv \{i : y_i = +1, \alpha_i = C\} \\
l_4 &\equiv \{i : y_i = -1, \alpha_i = 0\}
\end{align*}
\]
And now to check if the KT conditions hold, Keerthi also defines:

\[
\begin{align*}
    b_{up} &= \min\{F_i : i \in I_0 \cup I_1 \cup I_2\} = F_{i_{-up}} \quad (A)^{17} \\
    b_{low} &= \max\{F_i : i \in I_0 \cup I_3 \cup I_4\} = F_{i_{-low}} \quad (B)
\end{align*}
\]

The KT conditions then imply \( b_{up} \geq b_{low} \) and similarly

\[
\forall i \in I_0 \cup I_1 \cup I_2, F_i \geq b_{low} \quad \text{and} \quad \forall i \in I_0 \cup I_3 \cup I_4, F_i \leq b_{up}.
\]

These comparisons do not use the threshold \( b \).

As an added benefit, given the first \( \alpha_i^{old} \), these comparisons automatically find the second multiplier for joint optimization.

The pseudo code, as it can be found in Keerthi’s paper, can be found in appendix D.

As seen in the pseudo code and in Keerthi’s paper, there are two modifications on the SMO algorithm. Both were tested in the paper on different datasets and showed a significant speed-up in contrast to the original SMO algorithm by Platt. Also they overcome the problem arising when only using a single threshold (an example, why there are arising problems can also be found in Keerthi’s paper). As a conclusion on all tests Keerthi showed that the second modifications fares better overall.

### 9.2.6 SMO and the 2-norm case

As stated before, the SMO algorithm is not able to handle the 2-norm case without altering the code. Recall that there are two differences to the maximal margin and the 1-norm case: First the addition of \( 1/C \) to the diagonal of the kernel matrix and second the altered KT conditions, which are used in SMO as the stopping criterion:

\[
\begin{align*}
    \alpha_i &\geq 0 \\
    \alpha_i = 0 &\Rightarrow y_i (\langle w^* \cdot x_i \rangle + b^*) \geq 1 \\
    0 < \alpha_i &\Rightarrow y_i (\langle w^* \cdot x_i \rangle + b^*) = 1 - \frac{\alpha_i}{C}
\end{align*}
\]

And as the original SMO algorithm tests the KT conditions only in the outer loop when selecting the first multiplier to optimize, this is the point to alter. Also the kernel evaluation has to be extended to add the diagonal values.

In the pseudo-code above, the checking of the KT conditions is processed by:

\[
(A) \text{ and } (B) \text{ are links to the pseudocode in the appendix}
\]

---

\(^{17}\) (A) and (B) are links to the pseudocode in the appendix
\[ E_2 = \text{SVM output on point}[i] \cdot y_2 \text{ (check in error cache)} \]
\[ r_2 = E_2 \cdot y_2 \]
\[ \text{if } ((r_2 < -\text{tol}) \lor (r_2 > \text{tol} \land \alpha_2 > 0)) \]

where \( r_2 \) is the same as \( y_i f(x_i) - 1 \). So the KT conditions are tested against \( > 0 \) and \( < 0 \), where 0 is replaced by the tolerance “tol”. So for the 2-norm case the test is rewritten as:

\[ E_2 = \text{SVM output on point}[i] \cdot y_2 \text{ (check in error cache)} \]
\[ r_2 = E_2 \cdot y_2 + \alpha_2/C \]
\[ \text{if } ((r_2 < -\text{tol} \land \alpha_2 < C) \lor (r_2 > \text{tol} \land \alpha_2 > 0)) \]

Second, as in the maximal margin case, the box constraint on the multipliers has to be removed, because they are no longer upper bounded by C.

And last but not least, the bias has to be calculated only using alphas fulfilling the equation \( 0 < \alpha_i \Rightarrow y_i(\langle w \cdot x_i \rangle + b^*) = 1 - \frac{\alpha_i}{C} \).

9.3 Data Pre-processing

As one can read in [Lin03] they have some propositions on the handling of the used data.

9.3.1 Categorical Features

SVMs require that each data instance is represented as a vector of real numbers. Hence, if there are categorical attributes, they have to first be converted into numeric data. Cheng recommends to use \( m \) numbers for representing an \( m \)-category attribute. Then only one of the \( m \) numbers is one, and the others are zero. Consider the three category attribute {red, green, blue} which then can be represented as (0,0,1), (0,1,0) and (1,0,0). Cheng’s experience indicates that if the number of values in an attribute is not too many, this coding might be more stable than using a single number to represent a categorical attribute.
9.3.2 Scaling

Scaling the data before applying it to an SVM is very important. [Lin03] explains why the scaling is so important, and most of these considerations also apply to SVMs.

The main advantage is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. Because kernel values usually depend on the inner products of feature vectors, large attributes may cause numerical problems. So Cheng recommends linearly scaling each attribute to the range of [-1, +1] or [0, 1]. In the same way, the testing data then has to be scaled before testing it on the trained machine.

In this diploma thesis the most used scaling to [-1, +1] is used and the according formula for scaling an input \( x \) in this interval with length two is:

\[
\begin{align*}
x_{i, \text{scal}} &= \frac{x_i - x_{i, \text{min}}}{x_{i, \text{max}} - x_{i, \text{min}}} - \frac{1}{2} = 2 \cdot \frac{x_i - x_{i, \text{min}}}{x_{i, \text{max}} - x_{i, \text{min}}} - 1
\end{align*}
\]

with \( i \in \{1, 2, \ldots, n\} \). The scaling has to be done for each feature separately. So the min- and max-values are taken, regarding the current feature in each vector. To go in detail, the reason for doing this is follows:

Imagine a vector of 2 features (2-dimensional), the first has a value of 5, the second of 5000. Assume the other vectors behave the same way. So the first feature would not have a very great impact on distinguishing between the classes, because the change in feature one is numerically very small in contrast to that of feature two, whose numbers are in a much higher range.

Other long studied methods for scaling the data and showing very good results use the co-variance matrix from the Gaussian theory.

9.4 Matlab Implementation and Examples

This chapter is intended to show some examples and to get an impression how the different tuneable values, such as the penalty \( C \), the kernel parameters and the choice of maximal margin, 1-norm or 2-norm, affect the resulting classifier.
The implementation in Matlab with the original SMO algorithm can be found here together with the training sets (these files were used for making the following pictures possible):

Matlab Files\SVM

It should be mentioned, that the SMO implementation in Matlab is somewhat very slow. Therefore nearly any Toolbox for SVMs available written in Matlab implements the SMO algorithm as C-Code and calls it in Matlab through the so-called “Mex-functions” (Interface to C/Matlab). But for examining the small examples used here, the use of pure Matlab is acceptable. Later the whole code for Support Vector Machines will be implemented in C++ anyway to be integrated in the “Neural Network Tool” already existent at Siemens VDO.

For any upcoming visualisation the dimension of the training and test vectors is restricted to the two-dimensional case, because only such examples are visualizeable two- and three-dimensional, to be discussable. The three-dimensional pictures will show the values calculated by the learned decision function without applying the classification by the signum function “sgn” to it on the z-axis. The boundary will be shaded too, respectively to the functional margin of that point. Or in other words: The darker the shading, the more the point belongs to that specific class. The pictures will give clarification on this.

9.4.1 Linear Kernel

For examples using the linear “kernel”, the linear separable cases of the binary functions OR and AND are considered (figure 9.5 and 9.6).

The dashed lines represent the margin. The size of the functional margin is indicated by the level of shading.

A test of the same machine on the XOR case results in a classification with one error because of the nature of the XOR function to be non separable in input space (figure 9.6).

\[\text{18 A complete list with the usage and a short description of each file will be given in chapter 10.}\]
**Figure 9.5:** A linear kernel with maximal margin \((C = \infty)\) applied to the linear separable case of the binary OR function.

**Figure 9.6:** A linear kernel with maximal margin \((C = \infty)\) applied to the linear separable case of the binary AND function.
9.4.2 Polynomial Kernel

As seen before, the XOR case is non-separable in input space. Therefore the usage of a kernel mapping the data to a higher space and separating it there linearly could produce a classifier in input space, separating the data correctly. To test this, a polynomial kernel with maximal margin (C = \(\infty\)) of degree two is used. The result can be seen in figure 9.8.

To get an impression on how this data becomes separable by mapping it to a higher dimensional space, the three-dimensional picture in figure 9.9 visualizes the output of the classification step before applying the signum (sgn) function to it on the z axis.

Figure 9.7: A linear kernel with soft margin (C = 1000) applied to the linearly non-separable case of the XOR function. The error is 0.25 %, as one point is misclassified.
Figure 9.8: A polynomial kernel of degree 2 with maximal margin (C = inf) applied to the XOR dataset.
Figure 9.9: The classifier of figure 9.8 visualized by showing the calculated value of the classification on the z axis before the application of the signum (sgn) function.

Here one can see that the yellow regions applying to one of the classes have greater positive values and the green region applying to the other class has values lower than zero. The change of separation from one class to the other is at the zero level of the classifier output (z axis), as the signum function changes sign there.

The main conclusion drawing from the pictures up to now and from further ones is, that the application of a kernel measures the similarity between the data in some way. Because regarding the last two figures again, one can see that the points belonging to the same class are mapped to the same “direction” (output values >= 0 or < 0). The upcoming pictures on the Gaussian kernel will stress this fact.

9.4.3 Gaussian Kernel (RBF)

As stated in the chapter on the kernels, if one has no idea on how the data is dependable, as a first start the Gaussian kernel(s) or in other words, the radial basis function(s) is/are a good choice.

Sure in the XOR case applying this kernel will be the same as shooting with canons on sparrows, but the pictures resulting from doing so anyway, stress the fact that a kernel measures the similarity of data in some way (the resulting value before applying the signum function). Another fact is
that here the result of changing the sigma value (variance, “window width”, see 6.2.3) can be seen quite clear.

Figure 9.10: The RBF kernel applied to the XOR data set with $\sigma = 0.1$ and maximal margin ($C = \infty$).

To see how the change of the sigma value (variance) affects the resulting classifier, compare figures 9.10 and 9.11 to figures 9.12 and 9.13. Notice the smoother and wider course of the curves at the given training points.
**Figure 9.11:** The classifier of figure 8.9 ($\sigma = 0.1$), visualized by showing the calculated value of the classification on the $z$ axis before the application of the signum (sgn) function. Remarkable are the “Gauss curves” at the position of the four given training points (Here the classifier is more confident that a point in that region belongs to the specific class).

**Figure 9.12:** The RBF kernel applied to the XOR data set with $\sigma = 0.5$ and maximal margin ($C = \infty$).
Figure 9.13: The classifier of figure 9.9 ($\sigma = 0.5$), visualized by showing the calculated value of the classification on the z axis before the application of the signum (sgn) function. Remarkable are the “Gauss curves” at the position of the four given training points. But in contrast to figure 9.11 with a value of sigma $\sigma = 0.1$ they are much smoother and “wider”, as sigma changes the “width” (Consider the affect of the variance in Gaussian distribution).

To get an impression on how different values of the penalty parameter C (soft margin case for $0 < C < \infty$) affect the resulting classifier the next pictures illustrate this application of C.

As a starting point assume the classification problem of figure 9.14, classified by a SVM with a Gaussian kernel using $\sigma = 0.2$ and the maximal margin concept, allowing no training errors. The resulting classification regions are not very smooth, due to the two training points lying in the midst of the other class. Therefore applying the same machine on the dataset but with the soft margin approach by applying the upper bound by setting C to five results in the classifier of picture 9.15.

Here the whole decision boundary is much smoother than in the maximal margin case. The main “advantage” is the broader margin, implying a better generalization. This fact is also stressed in the figures of 9.16 and the next sub chapter.
Figure 9.14: A Gaussian kernel with $\sigma = 0.2$ and maximal margin ($C = \infty$). The dashed margins are not really "wide", because of the two points lying in the midst of the other class and the application of the maximal margin classifier (no errors allowed).
Figure 9.15: A Gaussian kernel with $\sigma = 0.2$ and soft margin (C = 5). This approach gives smoother decision boundaries in contrast to the classifier in figure 8.14 but at the expense of misclassifying two points now.

9.4.4 The Impact of the Penalty Parameter C on the Resulting Classifier and the Margin

Now the change of the resulting classifier (boundary, margins) when applying the maximal margin and the soft margin approach will be analyzed in detail.

Assume the training set used in figure 9.16. The SVM used there is based on a Gaussian kernel applying the concept of the maximal margin approach, allowing no training error (C = inf). As one can see, the resulting classifier does not have a very broad margin. And therefore, as stated in the Theory on Generalization in part one of this diploma thesis, this classifier is assumed not to generalize very well.

In contrast to this the approaches in figures 9.17 to 9.19 use the soft margin optimization and result in a broader margin but this on the expense of allowing training errors. But such “errors” can also be interpreted as the classifier does not overestimate the influence of some “outliers” in the training set (because of such ones the “hill” in figure 9.16 is in the midst of where one can imagine the other class should be).
Figure 9.16: A SVM with a Gaussian kernel with $\sigma = 0.8$ and maximal margin ($C = \inf$).
The resulting classifier is compatible with the training set without error, but has no broad margin.

So these classifiers are assumed to generalize better in this case, what is the goal of a classifier:

*He must generalize very well but minimize the error of classification.*

As stated in chapter two another very general estimation of the generalization error of SVMs are the number of support vectors gained after training:

\[
\frac{\text{# SV}}{l}
\]

So small numbers of support vectors are expected to give better generalisation. Another advantage in practice is, that the fewer support vectors there are the less expensive is the computation of the classification of a point.

So to summarize, as the theory on generalization stated, a broad margin and few support vectors are indications for good generalization. So the application of the soft margin approach can be seen as a compromise between minor empirical risk and minor optimism.
Figure 9.17: A SVM with a Gaussian kernel with $\sigma = 0.8$ and soft margin ($C = 100$). Notice the broader margin in contrast to figure 9.16. The boundary has become smoother and the three (four, one is a margin error, the others are “real” errors) misclassified points do not have as much impact on the boundary as in figure 9.16.

Figure 9.18: A SVM with a Gaussian kernel with $\sigma = 0.8$ and soft margin ($C = 10$). Notice the broader margin in contrast to figure 9.16 and 9.17. The boundary is much more smoother.
Figure 9.19: A SVM with a Gaussian kernel with $\sigma = 0.8$ and soft margin ($C = 1$). Notice the broader margin in contrast to figure 9.16, 9.17 and 9.18, and the much smoother boundary.