Algorithms for solving the SVM Optimization

1 Introduction

In previous lectures we derived the following form for the SVM optimization problem ($L_1$ soft margin):

Max $\sum \alpha_i - \frac{1}{2} \sum \alpha_i \alpha_j y_i y_j K(x_i x_j)$ subject to $0 \leq \alpha_i \leq C$, $\sum \alpha_i y_i = 0$

Basic points:

- Using KKT conditions, we can check if a proposed solution is optimal.

- The objective function is quadratic with a global maximum, so if we do an uphill search while maintaining constraints we will find the maximum. We do not necessarily need to perform gradient ascent - any step that guarantees going up is useful. The difference is whether we can get quick convergence.

- There are general solvers for quadratic programming problems, which includes SVM, but SVM problem can be simpler than the general problem. Some of the solutions methods discussed in the lecture do use such “off the shelf” solvers. However, these are only run on small subproblems.

- The main advantage in developing algorithms specific for SVM is that that SVM solutions are sparse, that is, do not have many support vectors (where $\alpha_i \neq 0$). This will be used by the fast solvers.

2 Techniques to speed up solver

- **Shrinking:** Guess examples $i$ s.t. $\alpha_i = 0$ and examples s.t. $\alpha_i = C$

  Fix these $\alpha$’s and solve subproblem on remaining examples.

  Check using the KKT conditions, if optimal then we’re done.

  If not, guess again.

- **Chunking:** pick subset $I$ to optimize over. Fix $\alpha_i$ for $i \notin I$ and solve the problem for the remaining indices.

$$L = \sum \alpha_i - \frac{1}{2} \sum \alpha_i \alpha_j y_i y_j K_{i,j}$$

$$= \sum_{i \in I} \alpha_i + \text{Const} - \frac{1}{2} \sum_{i \in I} \sum_{j \in I} \alpha_i \alpha_j y_i y_j K_{i,j} + \text{Const} - \sum_{i \in I} \alpha_i y_i \sum_{j \notin I} y_j \alpha_j K_{i,j}$$
Let $W_i = y_i \sum_{j \in I} y_j \alpha_j K_{i,j}$. then $L = Const + L_I$ where $L_I = \sum_{i \in I} \alpha_i (1 - W_i) - \sum_{i,j \in I} \alpha_i \alpha_j y_i y_j K_{i,j}$.

The reduced optimization problem is to maximize $L_I$ subject to $0 \leq \alpha_i \leq C$ for $i \in I$ and $\sum_{i \in I} \alpha_i y_i = -\sum_{i \notin I} \alpha_i y_i$.

Therefore like shrinking, chunking gives a subproblem of the same type. We can use this to develop an iterative algorithm as follows:

**Iterative Chunking:**

- Init some $I$ and $\alpha$.
- Repeat: solve $L_I$ and check if optimal if not add to $I$ any example violating KKT conditions.
- Issue 1: want to make sure $L_I$ goes up with every iteration.
- Issue 2: Hope to add to $I$ only elements that are Support Vectors in final solution. Notice that even if this happens the last problem we solve includes all the support vectors so it may still be large.

### 3 More Efficient Solvers

**Decomposition methods** avoid this by keeping $|I|$ small. One such method is to fix the size of $I$ over all iterations. The main question is how to pick $I$ s.t. $L_I$ always increases, because if we can do that then we will reach convergence.

**Fact:** It is sufficient to include one KKT violator in $I$ to guarantee that $L$ goes up.

**Why?** Solve $L_I$, we know that the current setting for $I$ is not optimal. This means that $L_I$ goes up. Therefore $L$ goes up as well (the difference between them is constant since other parameters are not changed).

So the main questions is what $I$ to choose at each step. Notice that using a small $I$ will probably give a small improvement in $L$ therefore requiring many iterations, but the QP will be solved easily so will be fast. On the other hand a large $I$ may need fewer iterations but each iteration will be slower. SVMlight picks $|I| = q$ small but $> 2$. SMO takes the tradeoff to the extreme using $I = 2$.

The advantage of SMO is that the QP can be solved analytically, there is no need for QP engine for the subproblem, and we get very fast run time per iteration. libSVM implements SMO; it differs from the original formulation of SMO in using a different formulation of stopping criterion and in the choice of $I$. It also avoids the potential for backtracking search for an $I$ that makes progress that was needed in the original formulation. In the following we provide some of the details of libSVM.

We first provide the following three ingredients:

- new criterion to check optimality
- from this, a method to choose $I$ of size 2
- analytic solution of size 2 problem

**Stopping Criterion and Choice of $I$:**

Define: $g_i = \frac{\partial L}{\partial \alpha_i} = 1 - y_i \sum_j \alpha_j y_j K_{i,j}$ and $g_i^* = g_i |_{\alpha = \alpha^*}$ where $\alpha^*$ is the optimum $\alpha$ vector.

We must always satisfy $\sum \alpha_i y_i = 0$, so if we increase one $\alpha_i y_i$ in SMO, we must decrease the other. We know $\alpha_i y_i \in [0, C]$ if $y_i = 1$ and $\alpha_i y_i \in [-C, 0]$ if $y_i = -1$. We will use the notation
\[ \alpha_i y_i \in [A_i, B_i] \] to capture both cases and simplify our notation. Define \( I_{up} = \{ i | \alpha_i y_i < B_i \} \) and \( I_{down} = \{ i | \alpha_i y_i > A_i \} \). These are the examples for which we can push the value of \( \alpha_i y_i \) up or down respectively without violating the box constraints.

**Fact:** \( \alpha \) is optimal iff \( \max_{i \in I_{up}} y_i g_i < \min_{j \in I_{down}} y_j g_j \)

We show only one direction of this condition. If \( \alpha^* \) is optimal and \( \alpha \) is not then \( L(\alpha^*) \geq L(\alpha) \) so \( L(\alpha) - L(\alpha^*) \leq 0 \). Define: \( \alpha^e = \alpha^* + \epsilon y_i 1_i - \epsilon y_j 1_j \) where \( 1_k \) is the unit vector with 1 in the \( k \)’th index. This represents a potential change to the vector \( \alpha^* \) in SMO.

By Taylor expansion we can write \( L(\alpha^\prime) = L(\alpha^*) + \epsilon y_i \frac{\partial L}{\partial \alpha_i} |_{\alpha^*} - \epsilon y_j \frac{\partial L}{\partial \alpha_j} |_{\alpha^*} \) (plus a second order term). From this, \( 0 \geq L(\alpha^*) - L(\alpha^\prime) = \epsilon(y_i g_i^* - y_j g_j^*) \). Therefore, \( y_i g_i^* \leq y_j g_j^* \). The only requirement in these equations was that \( i \in I_{up} \) and \( j \in I_{down} \) so that we can push the values in the appropriate directions. Therefore, this holds for the maximizing \( i \) and the minimizing \( j \).

So now we have a new way to define optimality, but we also have a method for picking points. Pick the max violating pair, defined as \( (\max_{i \in I_{up}} y_i g_i^*, \min_{j \in I_{down}} y_j g_j^*) \). This pair is guaranteed to make some progress in \( L_I \).

**Analytic solution of 2 point problem:**

Define \( \alpha^\lambda = \alpha^* + \lambda y_i 1_i - \lambda y_j 1_j \). We want to optimize \( L(\alpha^\lambda) = \sum \alpha_k^\lambda - \frac{1}{2} \sum \alpha_k^\lambda \alpha_l y_k y_l K_{k,l} \) where \( \lambda \) is constrained such that \( \alpha_k^\lambda \in [A_i, B_i] \) and \( \alpha_l^\lambda \in [A_j, B_j] \).

We start by solving the unconstrained problem. Recall Newton’s method to optimize \( f(\lambda) \): iteratively update \( \lambda^{new} = \lambda^{old} - \frac{f'(\lambda) \lambda^{old}}{f''(\lambda) \lambda^{old}} \). This gives the optimal solution in one step if \( f() \) is quadratic. For example, let us optimize \( f(\lambda) = \lambda^2 - 5\lambda + 6 \) and pick initial value of \( \lambda = 0 \). We have \( f'(\lambda) = 2(\lambda - 5) \)\( \lambda = 0 \) = -5 and \( f''(\lambda) |_{\lambda = 0} = 2 \). Therefore we get \( \lambda^{new} = -5/2 = 2.5 \) which is indeed the minimum.

Applying the same methodology to \( L(\lambda) = \sum \alpha_k^\lambda - \frac{1}{2} \sum \alpha_k^\lambda \alpha_l y_k y_l K_{k,l} \) we get:

\[ L'(\lambda) = y_i - y_j - y_i^2 \sum \alpha_k^\lambda y_k K_{i,k} + y_j^2 \sum \alpha_k^\lambda y_k K_{j,k} \]

and

\[ L'(\lambda) |_{\lambda = 0} = y_i g_i - y_j g_j \]

where we have substituted the values of \( g_i, g_j \) from above. Next we get

\[ L''(\lambda) = -y_i^2 K_{i,i} + y_j^2 K_{j,j} - y_i^2 K_{i,j} - y_j^2 K_{j,i} = -(K_{i,i} + K_{j,j} - 2K_{i,j}) \]

Therefore:

\[ \lambda^{new} = \frac{y_i g_i - y_j g_j}{K_{i,i} + K_{j,j} - 2K_{i,j}} \]

Finally, we reconsider the constraints. If \( \lambda \) is too big then we restrict the update to the boundary. Thus we choose:

\[ \lambda = \min\{ B_i - y_j \alpha_i, y_j \alpha_j - A_j, \frac{y_i g_i - y_j g_j}{K_{i,i} + K_{j,j} - 2K_{i,j}} \} \]

**The algorithm:** we initialize with \( \alpha = 0 \) and iteratively, (1) calculate \( g_i \)’s, (2) check stopping criterion: \( \max_{i \in I_{up}} y_i g_i - \min_{j \in I_{down}} y_j g_j \leq \epsilon = 0.001 \), if not done then (3) pick \( I \), and (4) calculate the next \( \alpha \). Various points are important for efficient implementation, for example, updating the \( g_i \)’s incrementally and caching values.