Introduction to Support Vector Machines

Dustin Boswell

August 6, 2002

1 Description

Support Vector Machines (SVM's) are a relatively new learning method used for binary classification. The basic idea is to find a hyperplane which separates the d-dimensional data perfectly into its two classes. However, since example data is often not linearly separable, SVM's introduce the notion of a "kernel induced feature space" which casts the data into a higher dimensional space where the data *is* separable. Typically, casting into such a space would cause problems computationally, and with overfitting. The key insight used in SVM's is that the higher-dimensional space doesn't need to be dealt with directly (as it turns out, only the formula for the dot-product in that space is needed), which eliminates the above concerns. Furthermore, the VC-dimension (a measure of a system's likelihood to perform well on unseen data) of SVM's can be explicitly calculated, unlike other learning methods like neural networks, for which there is no measure. Overall, SVM's are intuitive, theoretically well- founded, and have shown to be practically successful. SVM's have also been extended to solve *regression* tasks (where the system is trained to output a numerical value, rather than "yes/no" classification).

2 History

Support Vector Machines were introduced by Vladimir Vapnik and colleagues. The earliest mention was in (Vapnik, 1979), but the first main paper seems to be (Vapnik, 1995).

3 Mathematics

We are given l training examples $\{\mathbf{x}_i, y_i\},\ i = 1, \cdots, l\ ,$ where each example has d inputs ($\mathbf{x}_i \in \mathbf{R}^d$), and a class label with one of two values ($y_i \in \{-1, 1\}$). Now, all hyperplanes in $\mathbf{R}^d$ are parameterized by a vector ($\mathbf{w}$), and a constant (b), expressed in the equation

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{1}$$

(Recall that $\mathbf{w}$ is in fact the vector orthogonal to the hyperplane.) Given such a hyperplane ($\mathbf{w}$,b) that separates the data, this gives the function

$$f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \tag{2}$$

which correctly classifies the training data (and hopefully other "testing" data it hasn't seen yet). However, a given hyperplane represented by ($\mathbf{w}$,b) is equally expressed by all pairs $\{\lambda\mathbf{w}, \lambda b\}$ for $\lambda \in \mathbf{R}^+$. So we define the *canonical hyperplane* to be that which separates the data from the hyperplane by a "distance" of at least[1] 1. That is, we consider those that satisfy:

$$\mathbf{x_i} \cdot \mathbf{w} + b \geq +1 \quad \text{when} \quad y_i = +1 \tag{3}$$
$$\mathbf{x_i} \cdot \mathbf{w} + b \leq -1 \quad \text{when} \quad y_i = -1 \tag{4}$$

or more compactly:

$$y_i(\mathbf{x_i} \cdot \mathbf{w} + b) \geq 1 \qquad \forall i \tag{5}$$

All such hyperplanes have a "functional distance" ≥ 1 (quite literally, the function's value is ≥ 1). This shouldn't be confused with the "geometric" or "Euclidean distance" (also known as the *margin*). For a given hyperplane ($\mathbf{w}$,b), all pairs $\{\lambda\mathbf{w}, \lambda b\}$ define the exact same hyperplane, but each has a different functional distance to a given data point. To obtain the geometric distance from the hyperplane to a data point, we must normalize by the magnitude of $\mathbf{w}$. This distance is simply:

$$d\Big((\mathbf{w}, b)\ ,\ \mathbf{x}_i\Big) = \frac{y_i(\mathbf{x_i} \cdot \mathbf{w} + b)}{\parallel \mathbf{w} \parallel} \geq \frac{1}{\parallel \mathbf{w} \parallel} \tag{6}$$

Intuitively, we want the hyperplane that maximizes the geometric distance to the closest data points. (See Figure 1.)

[1] In fact, we require that at least one example on both sides has a distance of *exactly* 1. Thus, for a given hyperplane, the scaling (the λ) is implicitly set.

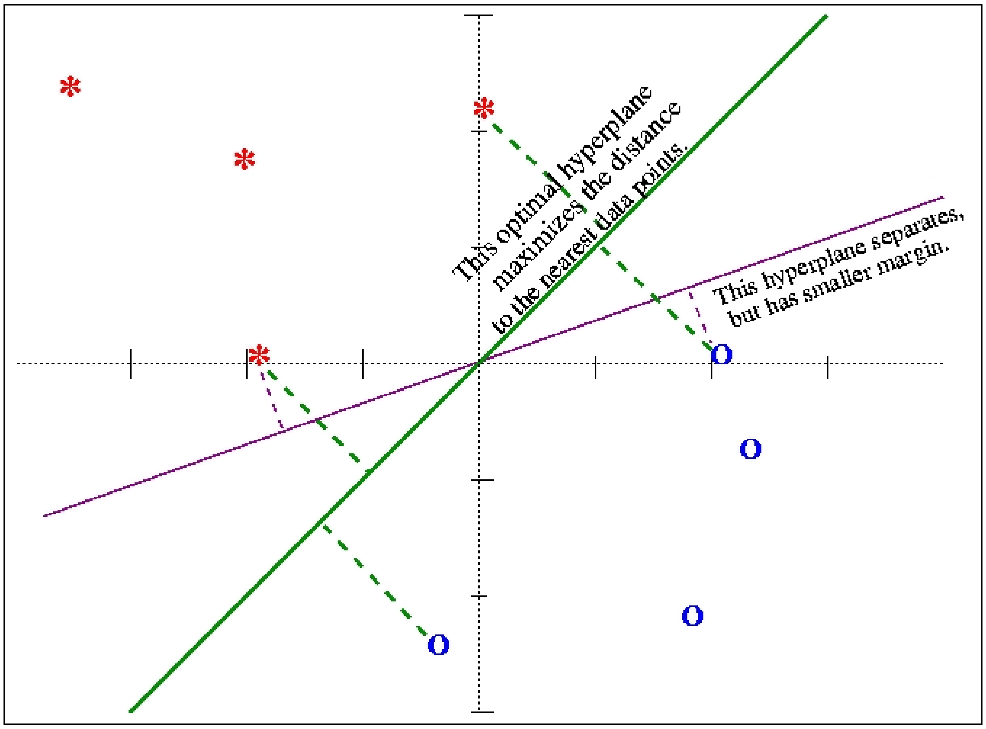

Figure 1: Choosing the hyperplane that maximizes the margin.

From the equation we see this is accomplished by minimizing $\parallel \mathbf{w} \parallel$ (subject to the distance constraints). The main method of doing this is with Lagrange multipliers. (See (Vapnik, 1995), or (Burges, 1998) for derivation details.) The problem is eventually transformed into:

$$\begin{array}{rl} \text{minimize:} & W(\alpha) = -\sum_{i=1}^{l} \alpha_i + \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j (\mathbf{x_i} \cdot \mathbf{x_j}) \\ \text{subject to:} & \sum_{i=1}^{l} y_i \alpha_i = 0 \\ & 0 \leq \alpha_i \leq C \quad (\forall i) \end{array}$$

where α is the vector of l non-negative Lagrange multipliers to be determined, and C is a constant (to be explained later). We can define the matrix $(H)_{ij} = y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$, and introduce more compact notation:

$$\text{minimize:} \quad W(\alpha) = -\alpha^T \mathbf{1} + \frac{1}{2} \alpha^T H \alpha \tag{7}$$

$$\text{subject to:} \quad \alpha^T y = 0 \tag{8}$$

$$\mathbf{0} \leq \alpha \leq C\mathbf{1} \tag{9}$$

(This minimization problem is what is known as a *Quadratic Programming Problem (QP)*. Fortunately, many techniques have been developed to solve them. They will be discussed in a later section.)

In addition, from the derivation of these equations, it was seen that the optimal hyperplane can be written as:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \tag{10}$$

That is, the vector $\mathbf{w}$ is just a linear combination of the training examples.

Interestingly, it can also be shown that

$$\alpha_i(y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1) = 0 \quad (\forall i)$$

which is just a fancy way of saying that when the functional distance of an example is strictly greater than 1 (when $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) > 1$), then $\alpha_i = 0$. So only the closest data points contribute to $\mathbf{w}$. These training examples for which $\alpha_i > 0$ are termed *support vectors*. They are the only ones needed in defining (and finding) the optimal hyperplane. [2] Intuitively, the support-vectors are the “borderline cases” in the decision function we are trying to learn. [3] Even more interesting is that α_i can be thought of as a “difficulty rating” for the example $\mathbf{x}_i$ - how important that example was in determining the hyperplane.

Assuming we have the opitimal α (from which we construct $\mathbf{w}$), we must still determine b to fully specify the hyperplane. To do this, take any “positive” and “negative” support vector, $\mathbf{x}^+$ and $\mathbf{x}^-$, for which we know

$$(\mathbf{w} \cdot \mathbf{x}^+ + b) = +1$$
$$(\mathbf{w} \cdot \mathbf{x}^- + b) = -1$$

Solving these equations gives us

$$b = -\frac{1}{2}\Big(\mathbf{w} \cdot \mathbf{x}^+ + \mathbf{w} \cdot \mathbf{x}^-\Big) \tag{11}$$

Now, you may have wondered the need for the constraint (eq. 9)

$$\alpha_i \leq C \quad (\forall i)$$

[2] and had we thrown away the non-support vectors before training, the exact same hyperplane would have been found.

[3] For example, in (Freund, et al. 1997a) they show the support vectors found by a SVM classifying pictures of human faces vs. non-faces. It is interesting to see that the support vectors on the non-face side are pictures that look somewhat like faces.

When $C = \infty$, the optimal hyperplane will be the one that completely separates the data (assuming one exists). For finite C, this changes the problem to finding a "soft-margin" classifier [4], which allows for some of the data to be misclassified. One can think of C as a tunable parameter: higher C corresponds to more importance on classifying all the training data correctly, lower C results in a "more flexible" hyperplane that tries to minimize the margin error (how badly $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) < 1$) for each example. Finite values of C are useful in situations where the data is not easily separable (perhaps because the input data $\{\mathbf{X}_i\}$ are noisy).

4 The Generalization Ability of Perfectly Trained SVM's

Suppose we find the optimal hyperplane separating the data. And of the l training examples, N_s of them are support vectors. It can then be shown that the expected out-of-sample error (the portion of unseen data that will be misclassified), Π is bound by

$$\Pi \leq \frac{N_s}{l-1} \tag{12}$$

This is a very useful result. It ties together the notions that simpler systems are better (Ockham's Razor principle) and that for SVM's, fewer support vectors are in fact a more "compact" and "simpler" representation of the hyperplane and hence should perform better. If the data cannot be separated however, no such theorem applies, which at this point seems to be a potential setback for SVM's.

5 Mapping the Inputs to other dimensions - the use of Kernels

Now just because a data set is not linearly separable, doesn't mean there isn't some other concise way to separate the data. For example, it might be easier to separate the data using polynomial curves, or circles. However, finding the optimal curve to fit the data is difficult, and it would be a shame not to use the method of finding the optimal hyperplane that we investigated in the previous section. Indeed there is a way to "pre-process" the data in such a way that the problem is transformed into one of finding a simple

[4] See any of the main SVM references for more information on "soft-margin" classifiers.

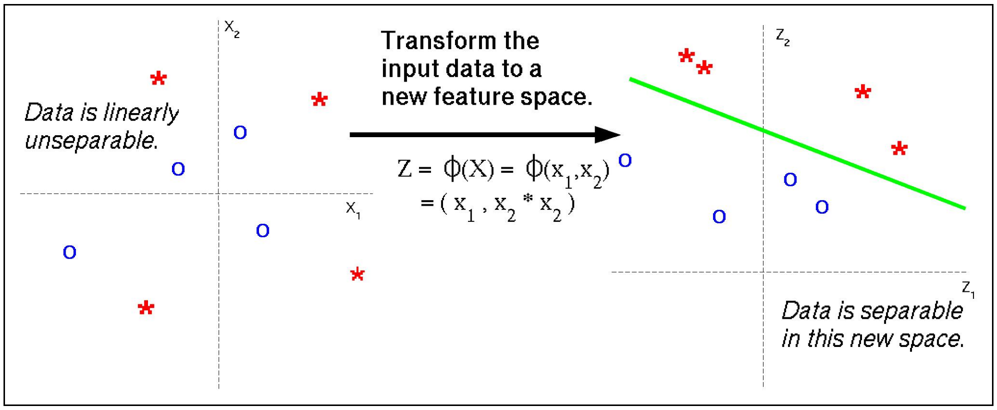
x2
Data is linearly unseparable.
x1
Transform the input data to a new feature space.
Z = ϕ(X) = ϕ(x1,x2)
= (x1 , x2 * x2)
z2
z1
Data is separable in this new space.

Figure 2: Separating the Data in a Feature Space.

hyperplane. To do this, we define a mapping $\mathbf{z} = \phi(\mathbf{x})$ that transforms the d dimensional input vector $\mathbf{x}$ into a (usually higher) d' dimensional vector $\mathbf{z}$. We hope to choose a $\phi()$ so that the new training data $\{\phi(\mathbf{x}_i), y_i\}$ *is* separable by a hyperplane. (See Figure 2.)

This method looks like it might work, but there are some concerns. Firstly, how do we go about chosing $\phi()$? It would be a lot of work to have to construct one explicitly for any data set we are given. Not to fear, if $\phi(\mathbf{x})$ casts the input vector into a high enough space ($d' \gg d$), the data should eventually become separable. So maybe there is a standard $\phi()$ that does this for most data... But casting into a very high dimensional space is also worrysome. Computationally, this creates much more of a burden. Recall that the construction of the matrix H requires the dot products $(\mathbf{x}_i \cdot \mathbf{x}_j)$. If d' is exponentially larger than d (and it very well could be), the computation of H becomes prohibitive (not to mention the extra space requirements). Also, by increasing the complexity of our system in such a way, *overfitting* becomes a concern. By casting into a high enough dimensional space, it is a fact that we can separate *any* data set. How can we be sure that the system isn't just fitting the idiosyncrasies of the training data, but is actually learning a legitimate pattern that will generalize to other data it hasn't been trained on?

As we'll see, SVM's avoid these problems. Given a mapping $\mathbf{z} = \phi(\mathbf{x})$, to set up our new optimization problem, we simply replace all occurences of $\mathbf{x}$ with $\phi(\mathbf{x})$. Our QP problem (recall eq. 7) would still be

$$\text{minimize:} \quad W(\alpha) = -\alpha^T \mathbf{1} + \tfrac{1}{2}\alpha^T H \alpha$$

but instead of $(H)_{ij} = y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$, it is $(H)_{ij} = y_i y_j (\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j))$. (Eq.

10) would be

$$\mathbf{w} = \sum_i \alpha_i y_i \phi(\mathbf{x}_i)$$

And (Eq. 2) would be

$$\begin{aligned} f(\mathbf{x}) &= \quad \operatorname{sign}(\mathbf{w} \cdot \phi(\mathbf{x}) + b) \\ &= \operatorname{sign}\Big([\textstyle\sum_i \alpha_i y_i \phi(\mathbf{x}_i)] \cdot \phi(\mathbf{x}) + b \Big) \\ &= \operatorname{sign}\Big(\textstyle\sum_i \alpha_i y_i (\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x})) + b \Big) \end{aligned}$$

The important observation in all this, is that any time a $\phi(\mathbf{x}_a)$ appears, it is always in a dot product with some other $\phi(\mathbf{x_b})$. That is, if we knew the formula (called a *kernel*) for the dot product in the higher dimensional feature space,

$$K(\mathbf{x_a}, \mathbf{x_b}) = \phi(\mathbf{x_a}) \cdot \phi(\mathbf{x_b}) \tag{13}$$

we would never need to deal with the mapping $\mathbf{z} = \phi(\mathbf{x})$ directly. The matrix in our optimization would simply be $(H)_{ij} = y_i y_j (K(\mathbf{x}_i, \mathbf{x}_j))$. And our classifier $f(\mathbf{x}) = \operatorname{sign}\Big(\sum_i \alpha_i y_i (K(\mathbf{x}_i, \mathbf{x})) + b \Big)$ Once the problem is set up in this way, finding the optimal hyperplane proceeds as usual, only the hyperplane will be in some unknown feature space. In the original input space, the data will be separated by some curved, possibly non-continuous contour.

It may not seem obvious why the use of a kernel aleviates our concerns, but it does. Earlier, we mentioned that it would be tedious to have to design a different feature map $\phi()$ for any training set we are given, in order to linearly separate the data. Fortunately, useful kernels have already been discovered. Consider the "polynomial kernel"

$$K(\mathbf{x_a}, \mathbf{x_b}) = (\mathbf{x_a} \cdot \mathbf{x_b} + 1)^p \tag{14}$$

where p is a tunable parameter, which in practice varies from 1 to $\sim$ 10. Notice that evaluating K involves only an extra addition and exponentiation more than computing the original dot product. You might wonder what the implicit mapping $\phi()$ was in the creation of this kernel. Well, if you were to expand the dot product inside K ...

$$K(\mathbf{x_a}, \mathbf{x_b}) = (\mathbf{x}_{a1}\mathbf{x}_{b1} + \mathbf{x}_{a2}\mathbf{x}_{b2} + \ldots + \mathbf{x}_{ad}\mathbf{x}_{bd} + 1)^p \tag{15}$$

and multiply these $(d{+}1)$ terms by each other p times, it would result in $\binom{d+p-1}{p}$ terms each of which are polynomials of varying degrees of the input vectors. Thus, one can think of this polynomial kernel as the dot product of two exponentially large $\mathbf{z}$ vectors. By using a larger value of p the dimension of the feature space is implicitly larger, where the data will likely be easier to separate. (However, in a larger dimensional space, there might be more support vectors, which we saw leads to worse generalization.)

5.1 Other Kernels

Another popular one is the *Gaussian RBF* Kernel

$$K(\mathbf{x_a}, \mathbf{x_b}) = exp\left(- \frac{\parallel \mathbf{x_a} - \mathbf{x_b} \parallel^2}{2\sigma^2} \right) \tag{16}$$

where σ is a tunable parameter. Using this kernel results in the classifier

$$f(\mathbf{x}) = \text{sign}\left[\sum_i \alpha_i y_i exp\left(- \frac{\parallel \mathbf{x} - \mathbf{x_i} \parallel^2}{2\sigma^2} \right) + b \right]$$

which is really just a Radial Basis Function, with the support vectors as the centers. So here, a SVM was implicitly used to find the number (and location) of centers needed to form the RBF network with the highest expected generalization performance.

At this point one might wonder what other kernels exist, and if making your own kernel is as simple as just dreaming up some function $K(\mathbf{x_a}, \mathbf{x_b})$. As it turns out, K must in fact be the dot product in a feature space for *some* $\phi()$, if all the theory behind SVM's is to go through. Now there are two ways to ensure this. The first, is to create some mapping $\mathbf{z} = \phi(\mathbf{x})$ and then derive the analytic expression for $K(\mathbf{x_a}, \mathbf{x_b}) = \phi(\mathbf{x_a}) \cdot \phi(\mathbf{x_b})$. This kernel is most definitely the dot product in a feature space, since it was created as such. The second way is to dream up some function K and then *check* if it is valid by applying *Mercer's condition.* Without giving too many details, the condition states: Suppose K can be written as $K(\mathbf{x_a}, \mathbf{x_b}) = \sum_{i=1}^{\infty} f_i(\mathbf{x_a}) f_i(\mathbf{x_b})$ for some choice of the $f_i's$. If K is indeed a dot product in a feature space then:

$$\begin{aligned} \int\int K(\mathbf{x_a}, \mathbf{x_b}) g(\mathbf{x_a}) g(\mathbf{x_b}) d\mathbf{x_a} d\mathbf{x_b} > 0 \\ \forall\ g \ \ \text{such that} \\ 0 < \int g^2(\mathbf{x}) d\mathbf{x} < \infty \end{aligned}$$

The mathematially inclined reader interested in the derivation details is encouraged to see (Cristianini, Shawe-Taylor, 2000). It is indeed a strange mathematical requirement. Fortunately for us, the polynomial and RBF kernels have already been proven to be valid. And most of the literature presenting results using SVM's all use these two simple kernels. So most SVM users need not be concerned with creating new kernels, and checking that they meet Mercer's condition. (Interestingly though, kernels satisfy many closure properties. That is, addition, multiplication, and composition of valid kernels all result in valid kernels. Again, see (Cristianini, Shawe-Taylor, 2000).)

5.2 How to Choose the right Kernel for your Data Set

When designing a SVM, the user is faced with the choice of which kernel to use, and for a given kernel, how to set the parameter(s). (Recall, the polynomial kernel has a degree p to set; RBF kernels have the parameter σ.) The novice might be tempted to try all types of kernels/parameters, and choose the kernel with the best performance. This however, is likely to cause overfitting. The theoretical framework predicting a SVM's generalization performance is based on the assumption that the kernel is chosen *a priori*. How then, is one supposed to choose one kernel over another, ahead of time? Ideally, the user should incorporate his or her knowledge about the problem type, without looking at the specific training data given. For example, if one were training a SVM to do face recognition, they should look into previous work done with SVM's and face recognition. If past work has shown polynomial kernels with degree $p = 2$ to perform well in general, then that would be the best guess to use.

6 Implementing SVM's

The most straightforward way to train a SVM (determine the optimal hyperplane $(\mathbf{w}, b)$) is to feed (eqs.7 - 9) to a Quadratic Programming Solver. In Matlab, this would be the function *quadprog()*.[5] For training sets of less than 1000 examples, solvers like these may be fine. However, as the number of examples gets larger, there are two concerns. First, since *quadprog* (and many other QP packages) explicitly requires the matrix H, the memory consumption is $O(l^2)$. With 8-byte precision, and 500MB of RAM, this imposes a limit of $\sim$ 8,000 training examples. The second concern is that numerical

[5] Some older versions of Matlab have *qp()* instead, which is older, and less robust.

instability (resulting in sub-optimal solutions, or sometimes no solution at all) becomes a potential hazard. And as far as quadratic programming goes in general, numerical instability is a tricky problem to get rid of. Nevertheless, "off the shelf" methods like *quadprog* are unsuitable for problem instances of $l = 100{,}000$ examples or more.

6.1 Decomposition Methods

The general trick to avoid these problems is to decompose the problem into subproblems, each of which are small enough to solve with a generic QP solver.

One simple method of "chunking" (Boser, Guyon, Vapnik, 1992) starts with an arbitrary subset of the data, and solves the problem for those q examples ($q \ll l$). The support vectors from that subset are added to a second chunk of data, and the process is repeated until all vectors are found. Notice that toward the end of the algorithm, as all of the support vectors are gathered, the subproblems contain $O(N_s := \# \; of \; support \; vectors)$ examples, which results in a $O(N_s^2)$ memory requirement for the underlying QP-solver. Clearly, this method works best when $N_s \ll l$. Otherwise, if $N_s \sim l$, the "subproblem" is just as big as the original one. The run-time of this algorithm is supposedly $O(l^3)$.

Another decomposition method given by (Freund, Girosi, Osuna, 1997b) avoids this potential difficulty by creating a strict upper limit (q) on the number of examples dealt with at a time. The algorithm has a so-called "working set" of examples on which the objective function is currently being minimized. Depending on how/if they violate the optimality conditions, training examples are added to/removed from the working set in such a way that an increase in the objective function occurs after each QP subproblem. Since the overall QP problem is convex, any local maximum achieved this way is in fact a global maximum. [6] The memory requirement of this procedure is $O(ql)$. As for runtime, we should expect it takes longer than the simpler "chunking" version. It does work well in practice, but there is no proof of a convergence rate. (Freund, Girosi, Osuna, 1997b) successfully apply this method to a task where $l = 110{,}000$ and $N_s = 100{,}000$. (Joachims, 1999) extends the above method by giving run-time improvement heuristics, such as how to choose a good working set.

[6] There seems to be some conflicting views in the literature about whether the convergence of this method is guaranteed.

6.2 Optimization techniques for QP problems

At this point, we should mention some of the types of optimization techniques and packages (particularly those that solve QP problems). The main techniques are: stochastic gradient ascent, Newton's method, conjugate gradient descent, and primal-dual interior point method.

The naive method to find the optimal α for a given set of training examples is *stochastic gradient ascent*. The algorithm described in (Cristianini, Shawe-Taylor, 2000) simply updates one α_i at a time, and loops until the termination criteria is met. Since it only deals with one example at a time, it has virtually no memory requirements. Unfortunately, this method has no convergence guarantees either.

The Newton method (used by (Kaufman, 1999)), is a one-step solution to finding the optimal α vector. It requires $O(q^2)$ memory and $O(q^3)$ computation, where q is the number of examples being solved for.

The conjugate gradient technique (also used by (Kaufman, 1999)), is a q-step solution. Overall, it also has a run-time of $O(q^3)$. I've read literature describing both conjugate and Newton methods to have the same inherent memory complexity of $O(q^2)$, although it is ambiguous as to why this is the case for conjugate gradient. (Kaufman, 1999) judges the Newton method to be slightly more efficient than conjugate gradient in general.

Lastly, "Primal-Dual Interior Point" methods (Vanderbei, 1994) are often cited. This is the method used in the LOQO software package, and was also incorporated in SVM^{light} package. I don't know any details of this algorithm, except that supposedly it performs well in training sets where the number of support vectors is a large percentage of the whole training set (contrast this to the decomposition methods that do best when support vectors are sparse in the data set).

MINOS is a commercial QP package from the Systems Optimization Laboratory at Stanford, invented by Bruce Murtagh and Michael Saunders. I am unaware of the implementation details outside of the fact that it uses a hybrid strategy of different techniques. It is mentioned here because it seems to be a popular choice, and was used by (Freund, Girosi, Osuna, 1997ab) in their experiments.

6.3 Sequential Minimal Optimization - Decomposition to the Extreme

So far, all of the SVM training methods have the same flavor: since solving the underlying QP problem outright is inefficient, they break down the problem in one way or another, and then solve the QP tasks of these carefully chosen subproblems. In doing so there is a balance. The more sophisticated our decomposition algorithm (resulting in smaller subproblems), the more time is spent iterating over subproblems, and the less assurance we have of good convergence rates. The benefit, of course, is that less memory is required in solving the QP subproblems, and that we can worry less about the numerical instability of QP packages. On the other hand, the less "fiddling" we do in the way of decomposition, the more "intact" our problem remains, and the more assurance we have in the program's outputting a global optimum. Indeed, the reliance on QP packages is bothersome. As mentioned before, numerical instability is a concern for QP solvers. And it is a little unsettling knowing that most QP routines are too complex and difficult to be implemented "on the fly" by the user. For this reason, the method of Sequential Minimal Optimization (SMO), given in (Platt, 1999), is very appealing.

Platt takes decomposition to the extreme by only allowing a working set of size 2. Solving a QP problem of size 2 can be done analytically, and so this method avoids the use of a numerical QP solver altogether! The tradeoff, naturally, is that pairs of examples optimized in this way must be iterated over many many times. The claim, however, is that since the heart of the algorithm is just a simple analytic formula, the overall runtime is reduced. The biggest advantage of this method is that the derivation and implementation are astoundingly simple. Psuedocode can be found in (Platt, 1999) or (Cristianini, Shawe-Taylor, 2000).

It should be noted that SMO, while introduced as a method for training SVM classifiers, has also been extended to SVM regression (Flake, Lawrence, 2000). Also, (Keerthi, et al. 1999b) argue that Platt's method of computing the threshold (b) in SMO is inefficient, and give modifications to his psuedocode that make the algorithm much faster.

6.4 Other Techniques and Methods

6.4.1 Geometric Approaches

The intuition of finding the maximal margin hyperplane is very much a geometric one. To solve it, Lagrangian multipliers are used to set up the QP problem, and from there on, it's pure mathematics. However, finding the optimal hyperplane can be done in a geometric setting, too.

Consider the case when the data is separable in the hyperspace. It is a fact then, that the *convex hulls* [7] of the two classes are disjoint. The optimal separating hyperplane would be parallel to *some* "face" of both convex hulls. (And the points on that face would be the support vectors.) See (Keerthi, et al. 1999a) for a method of training SMV's in such a geometric way.

But even if the SVM isn't trained geometrically, the notion of the convex hull and that support vectors can only be points on the face of the convex hull leads to an interesting data-filtering scheme. Recall that only the training examples that become support-vectors are actually needed in determining the optimal hyperplane. If there were some way to know ahead of time which data points are not support vectors, we could throw them out, and this would reduce the problem size (assuming N_s is somewhat smaller than l). (Ahuja, Yang, 2000) call such points on the convex hull *guard vectors* (basically, "potential" support vectors). They show that determining which data points are guard vectors amounts to solving a series of Linear Programming problems (which are much easier to solve than QP). Non-guard vectors can then be thrown out before the regular SVM training procedure begins.

7 Web-sites, Software Packages, and Further Reading

The book *An Introduction to Support Vector Networks and other kernel-based learning methods* is a good place to start (along with their web-site http://www.support-vector.net), as is the book *Advances in Kernel Methods - Support Vector Learning.* (Vapnik, 1995) and (Burges, 1998) are good papers introducing SVM's. http://www.kernel-machines.org has a lot of useful links to papers and SVM software.

One particularly easy to use software package is "SVM light" (which was written by T. Joachims): http://www.cs.cornell.edu/People/tj/svm_light

[7] The convex hull is simply the subset of points on the "exterior" of the set. For example, if you wrapped cyran-wrap around the "ball" of points, the convex shape formed would involve exactly those points from the convex hull. The rest of the points would be on the interior, untouched by the cyran-wrap.

References

- Narendra Ahuja, Ming-Hsuan Yang. “A Geometric Approach to Train Support Vector Machines” Proceedings of the 2000 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2000), pp. 430-437, vol. 1, Hilton Head Island, June, 2000.
(http://vision.ai.uiuc.edu/mhyang/papers/cvpr2000-paper1.pdf)

- Bernhard E. Boser, Isabelle M. Guyon, Vladimir Vapnik. “A Training Algorithm for Optimal Margin Classifiers.” Fifth Annual Workshop on Computational Learning Theory. ACM Press, Pittsburgh. 1992
(http://citeseer.nj.nec.com/boser92training.html)

- Christopher J.C. Burges, Alexander J. Smola, and Bernhard Scholkopf (editors). *Advances in Kernel Methods - Support Vector Learning* MIT Press, Cambridge, USA, 1999

- Christopher J.C. Burges. ”A Tutorial on Support Vector Machines for Pattern Recognition”, Data Mining and Knowledge Discovery 2, 121-167, 1998
(http://www.kernel-machines.org/papers/Burges98.ps.gz)

- Nello Cristianini, John Shawe-Taylor. *An Introduction to Support Vector Networks and other kernel-based learning methods.* Cambridge University Press, 2000
(http://www.support-vector.net)

- Flake, G. W., Lawrence, S. “Efficient SVM Regression Training with SMO.” NEC Research Institute, (submitted to Machine Learning, special issue on Support Vector Machines). 2000
(http://www.neci.nj.nec.com/homepages/flake/smorch.ps)

- Robert Freund, Federico Girosi, Edgar Osuna. “Training Support Vector Machines: an Application to Face Detection.” IEEE Conference on Computer Vision and Pattern Recognition, pages 130-136, 1997a
(http://citeseer.nj.nec.com/osuna97training.html)

- Robert Freund, Federico Girosi, Edgar Osuna. “An Improved Training Algorithm for Support Vector Machines.” In J. Principe, L. Gile, N. Morgan, and E. Wilson, editors, *Neural Networks for Signal Processing VII – Proceeding of the 1997 IEEE Workshop*, pages 276-285, New York, 1997b
(http://citeseer.nj.nec.com/osuna97improved.html)

- Thorsten Joachims. ”Text Categorization with Support Vector Machines: Learning with Many Relevant Features”, 1998 (http://www.joachims.org/publications/joachims_98a.ps.gz)

- Thorsten Joachims. ”Making Large-Scale SVM Learning Practical”, 1999 (Chapter 11 of (Burges, 1999)) (http://www.joachims.org/publications/joachims_99a.ps.gz)

- Linda Kaufman. “Solving the Quadratic Programming Problem Arising in Support Vector Classification”, 1999 (Chapter 10 of (Burges, 1999))

- S.S. Keerthi, S.K. Shevade, C. Bhattacharyya and K.R.K. Murthy. “A fast iterative nearest point algorithm for support vector machine classifier design,” Technical Report TR-ISL-99-03, Intelligent Systems Lab, Dept. of Computer Science & Automation, Indian Institute of Science, 1999a. (http://citeseer.nj.nec.com/keerthi99fast.html)

- S.S. Keerthi, S.K. Shevade, C. Bhattacharyya, and K.R.K. Murthy. “Improvements to Platt’s SMO algorithm for SVM classifier design.” Technical report, Dept of CSA, IISc, Bangalore, India, 1999b. (http://guppy.mpe.nus.edu.sg/~mpessk/smo_mod.ps.gz)

- John C. Platt. “Fast Training of Support Vector Machines using Sequential Minimal Optimization” (Chapter 12 of (Burges, 1999)) (http://www.research.microsoft.com/~jplatt/smo-book.ps.gz)

- Robert Vanderbei. “Loqo: An Interior Point Code for Quadratic Programming.” Technical Report SOR 94-15, Princeton University, 1994 (http://www.sor.princeton.edu/~rvdb/ps/loqo6.pdf)

- Vladimir Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, New York, 1995

- Vladimir Vapnik, Corinna Cortes. ”Support vector networks,” Machine Learning, vol. 20, pp. 273-297, 1995. (http://citeseer.nj.nec.com/cortes95supportvector.html)