B.Tech. Project Report

Issues in kernel-based machine learning: optimizing kernels, online learning and support vector reduction

Sumeet Agarwal
Y2387
sagarwal@cse.iitk.ac.in
Guide: Dr. Harish Karnick
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

April 14, 2006

Abstract

We look at two major issues in kernel-based learning. First, we look to come up with a framework for determining the optimal kernel to be used in a machine learning task, for example, in the use of a Support Vector Machine (SVM) classifier. The motivation for doing this is that kernel-based classifiers like SVMs are being very widely used today for all kinds of machine learning applications, and any concrete result leading to an improvement in their performance would have immediate and widespread utility. Second, we look at applying kernel-based learning methods to online learning situations, and the related requirement of reducing the complexity of the learnt classifier. Online methods are particularly useful in situations which involve streaming data, such as medical or financial applications. We show that the concept of *span* of support vectors can be used to build a classifier that performs reasonably well whilst satisfying given space and time constraints, thus making it potentially suitable for such online situations.

1 Introduction

Kernel-based learning methods [17] have been successfully applied to a wide range of problems. The key idea behind these methods is to implicitly map the input data to a new feature space, and then find a suitable hypothesis in this space. The mapping to the new space is defined by a function called the kernel function. A key problem is the choice of the kernel function itself; which function should one use to obtain the best generalization results? Should the choice of kernel be made based on the training data, or can we define a universally 'good' kernel? These are some of the key questions we sought to address in the first part of this project. Based on a review of recently done work in this field, we can say that these questions have been partially answered, but there are still several issues to be ironed out.

The second part of this project looks at trying to use kernels to build a classifier that can be used in situations with streaming data. Such a classifier should consume limited space, and should be able to update itself each time a new point (which it fails to classify correctly) comes in. Since the number of support vectors, i.e. the classifier complexity, goes on increasing with increase in the amount of training data [18], the problem is essentially to reduce this complexity

in order to meet the space constraints, whilst minimizing the loss in generalization error. We look at a couple of ways of doing this, based on a concept known as support vector span, and another idea known as data squashing. Somewhat surprisingly, we find that it is often possible to reduce classifier complexity by an order of magnitude or more, without any significant loss in performance.

The rest of this report is organized as follows: Section 2 presents the basic ideas and concepts employed in kernel-based learning. It also looks at a couple of key results: the Representer theorem, which makes it computationally feasible to find the optimal function; and the concept of support vector span, which allows us to bound the generalization error. Section 3 looks at the idea of learning the optimal kernel, exploring two frameworks for this: the possibility of allowing the kernel to vary over a compact and convex space; and the use of a hyperkernel formulation for learning the 'optimal' kernel function over a Hilbert space. Section 4 looks at the basic problem of support vector reduction, and gives explanations of and results from the two methods tried by us for this purpose, span-based reduction and data squashing. We also look at some promising recently published work in this area. In Section 5, we describe the online learning framework, and present our proposed span-based algorithm, comparing its performance with previous results obtained in this setting. Finally, Section 6 summarizes the project and gives the overall conclusions of the work done by us, whilst also stating possible directions for future work. Section 7 has acknowledgments.

2 Basic Ideas

Supposing we wish to learn a function $f : \mathcal{X} \to \mathbb{R}$, given m input-output pairs $\{(x_i, y_i)\colon i \in \mathbb{N}_m, x_i \in \mathcal{X}, y_i \in \mathbb{R}\}$ (here $\mathbb{N}_m$ is the set of the first m natural numbers). One way to do this is by choosing a symmetric, positive definite function, $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, known as the kernel function. The intuitive idea behind using such a function is that it defines a 'kernel mapping', which implicitly maps the training data to a new (generally a higher-dimensional) space. In this space, it is typically easier to come up with a hypothesis that accurately explains the data. Some examples of the kinds of functions commonly used as kernels include Polynomials, Gaussians and other Radial Basis Functions. For a detailed overview of kernel-based learning methods, see [17].

The kernel method is based on Reproducing Kernel Hilbert Space (RKHS) theory [2]. A Hilbert space is an inner product space which is complete with respect to the norm induced on it by the inner product (if the inner product is denoted by $< ., . >$, then the norm of any element x is defined as $\|x\| = < x, x >$). An RKHS is a Hilbert space which is completely generated by a single kernel function. It can be shown that any symmetric, positive definite function has an associated RKHS (see, for example, [15]). Thus, once K has been decided upon, the function f is chosen from the RKHS generated by K, $\mathcal{H}_K$. Naturally, we would like to minimize the error of the chosen function. So, we may look to carry out Empirical Risk Minimization (ERM), by choosing $f \in \mathcal{H}_K$ such that the quantity

$$\frac{1}{m}\Sigma_{i\in\mathbb{N}_m}(f(x_i) - y_i)^2 \tag{1}$$

is minimized. However, this problem is in general ill-posed; in any case, it corresponds to minimizing the error on the training set only, whereas we would actually like to minimize the error on our entire input space. In order to take care of these issues, we regularize (Tikhonov; see [19]) the objective function to be minimized as follows:

$$Q(f(\mathbf{x}), \mathbf{y}) + \mu\|f\|_K^2 \tag{2}$$

Here, Q is a generic loss function, which could be of the form (1), $\|f\|_K$ is the norm of f in $\mathcal{H}_K$, and μ is a parameter which determines the trade-off between fitting the training set and the smoothness, or generalization capability, of the chosen function. The last term, known as the regularizer, ensures the uniqueness of the solution and also allows us to introduce the generalization

error as a criterion for the choice of function.

The choice of the loss function Q depends on the problem domain; the popular SVM classifier [6] uses the function

$$Q(f(\mathbf{x}), \mathbf{y}) = \Sigma_{i \in \mathbb{N}_m} (1 - y_i f(x_i)) \tag{3}$$

for binary classification, where the y_i's are all either 1 or -1. So, in this case, the loss incurred for each data point has just two possible values: 0 (if the point is correctly classified) or 2 (if the point is wrongly classified). A common choice of Q for regression problems is the square loss function (1). The parameter μ is empirically determined, using techniques like cross-validation. A central issue here is the choice of the kernel K itself, and we would like to know how to go about making this choice in order to get the best results. This is the key focus of this project. A considerable amount of work has been done in this direction recently, and some useful results are now known, which are summarized in the following sections. Before going into those, however, we will look at a very important mathematical result which makes the task of finding the minimizer of (2) much easier than it would seem to be at first sight.

2.1 The Representer Theorem

The regularized functional (2) gives us a means of defining the optimal function, but how do we actually carry out the required minimization in order to obtain this function? A well known result in Machine Learning, the Representer theorem (see, for example, [15]), tells us that irrespective of the nature of the loss function Q, the minimizer will be of the form:

$$f = \Sigma_{i \in \mathbb{N}_m} c_i K(x_i, .) \tag{4}$$

For a given Q, μ and K, it is possible to find the optimal $f \in \mathcal{H}_K$ by substituting its value from (4) in (2), and optimizing with respect to the vector $\mathbf{c} = \{c_1, ...c_m\}$ [17]. In particular, if Q is chosen to be the square-loss functional (1), then we get:

$$c_i = \frac{y_i - f(x_i)}{m\mu} \tag{5}$$

This means that we can find the coefficients by solving a system of linear equations, making it perfectly feasible to do so by computational means. Thus, the Representer theorem makes kernel-based learning methods practical to employ, and this has led to their wide use today.

2.2 Span of Support Vectors

The concept of support vector span was defined by Vapnik and Chapelle [20]. They showed that this could be used to obtain a tight bound on the generalization error of a given support vector machine. Suppose that we have a set of n support vectors denoted by

$$(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n) \tag{6}$$

where $\mathbf{x}_i$ denotes a feature vector and y_i the corresponding label (± 1 for binary classification). Also, let

$$\alpha^0 = (\alpha_1^0, ..., \alpha_n^0) \tag{7}$$

be the set of coefficients of the support vectors in feature space, for the optimal classification function (hyperplane). For any support vector $\mathbf{x}_p$, we define

$$\Lambda_p = \left\{ \sum_{i=1, i \neq p}^{n} \lambda_i x_i : \sum_{i=1, i \neq p}^{n} \lambda_i = 1;\ \forall i \neq p,\ \alpha_i^0 + y_i y_p \alpha_p^0 \lambda_i \geq 0 \right\} \tag{8}$$

as a constrained linear combination of the other support vectors. Note that λ_i can be negative.

We then define S_p, which we call the span of the support vector $\mathbf{x}_p$, as the distance between $\mathbf{x}_p$ and λ_p:

$$S_p = d(\mathbf{x}_p, \Lambda_p) = min_{\mathbf{x} \in \Lambda_p} \|\mathbf{x} - \mathbf{x}_p\| \tag{9}$$

The maximal value of S_p over all the support vectors is called the S-span:

$$S = max_{p \in \mathbb{N}_n} S_p \tag{10}$$

This definition of span holds for the case when the training data is linearly separable (in feature space); a slightly modified expression can be used to account for non-separable data as well. Vapnik and Chapelle show that $S \leq D_{SV}$, where D_{SV} is the diameter of the smallest sphere enclosing all the support vectors. Using this definition of span, they derive the following bound:

$$Ep_{error}^{l-1} \leq E\left(\frac{SD}{l\rho^2}\right) \tag{11}$$

This means that the expected value of generalization error for an SVM trained on a training set of size $l-1$ is bounded by the expression on the right hand side, where D is the diameter of the smallest sphere enclosing the training data, S is the S-span of the support vector set, and ρ is the margin. These quantities are all taken for training sets of size l; the difference in size of one element comes in due to the fact that the bound is derived via a leave-one-out procedure on the training data.

Vapnik and Chapelle also demonstrate via experiment that in practice, this is a tight bound. We will employ their results to justify our span-based approach to support vector reduction. For the proof of the stated bound, the reader is referred to the original paper [20].

3 Learning Optimal Kernels

3.1 Optimizing the Kernel Function over Compact and Convex Spaces

Given a certain hypothesis space for the choice of the kernel function K, we would like to be able to find the one that gives us the best classification results for the given training set. Recent work [12] has shown that if the set from which K is to be chosen is compact and convex, then there exists an 'optimal' kernel (that is, the optimal f for this choice of kernel is better than the optimal f for any other choice). Formally, for a given loss function Q and kernel function K, let $Q_\mu(K)$ be the infimum of (2) over all $f \in \mathcal{H}_K$. Also, let $\mathcal{K}$ be the hypothesis space from which K is taken. Then, the optimal kernel is defined as the minimizer of $Q_\mu(K)$ over all $K \in \mathcal{K}$.

Micchelli and Pontil [12] have obtained some meaningful results in this regard, in particular the following:

Theorem 1 *If $\mathcal{G}$ is a compact set of basic kernels, $\mathcal{K}$ is the closure of the convex hull of $\mathcal{G}$, the loss function Q is continuous and μ is a positive number then there exists $T \subseteq \mathcal{G}$ containing at most $m+2$ basic kernels such that Q_μ admits a minimizer over the convex hull of T, and this is the same as its minimizer over $\mathcal{K}$.*

This result is significant, as it puts a bound on the number of basic kernels that contribute to the optimal kernel. An algorithm for actually computing the optimal kernel, as defined here, has also been described [1]. The kernel so obtained can then be used to learn the best classification or regression function for the data, employing the usual approach and making use of the Representer theorem. However, the optimality of the kernel obtained by this method is restricted

to the given training set; for a new training set, the process will have to be repeated. Also, the method works only for compact and convex sets. To address these limitations, we decided to explore a new idea; defining the kernel hypothesis space itself as an RKHS, and optimizing over kernels in a way analogous to that used for optimizing over functions.

3.2 The Hyperkernel Formulation

Just as a kernel function K is defined in order to optimize f over the associated RKHS, we can think of defining a kernel function K^*, such that we look to optimize K itself over the RKHS generated by K^*. We will refer to the function K^* as the hyperkernel. Formally, K^* : $\mathcal{X} \times \mathcal{X} \times \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Given such a hyperkernel, we could then seek to optimize the kernel function K over this 'Hyper RKHS', in the same way as we normally optimize a function over an RKHS. The Representer theorem could, naturally, be applied here as well; it would tell us that the optimal kernel function is of the form

$$K(x, x\prime) = \Sigma_{i \in \mathbb{N}_m, j \in \mathbb{N}_m} c_{ij} K^*(x_i, x_j, x, x\prime) \tag{12}$$

The key issues here are those of appropriately defining the hyperkernel function, and computing the coefficients (all the c_{ij} values). In order to do the latter, we require actual label values on the training data set. Earlier, our training data consisted of the vector $\mathbf{x}$, and we had the corresponding vector of true labels $\mathbf{y}$. Now, however, our training data consists of pairs of the form (x_i, x_j); we need to attach a real-number label to each such pair. This label ought to be the desired value of $K(x_i, x_j)$, where K is the kernel function we are looking to learn. We do not know these values explicitly. However, Lanckriet et al. [10] have presented a Semi-Definite Programming (SDP) formulation for learning the kernel matrix (i.e., the values of the optimal kernel on the $m \times m$ data point pairs). So, we could first use this approach to learn these values, and then use them as the labels in order to learn the kernel function itself. To put it formally, suppose $\mathbf{K_{m \times m}}$ is the matrix of kernel values learnt by the SDP method. Then, we define a loss functional for the kernel as follows:

$$Q^*(K, \mathbf{x}, \mathbf{K_{m \times m}}) = \Sigma_{i \in \mathbb{N}_m} \Sigma_{j \in \mathbb{N}_m} (\mathbf{K_{ij}} - K(x_i, x_j))^2 \tag{13}$$

Now, following the standard regularization approach, we define the optimal kernel function to be the minimizer of

$$Q^*(K, \mathbf{x}, \mathbf{K_{m \times m}}) + \mu^* \|K\|^2_{K^*} \tag{14}$$

where $K \in \mathcal{H}_{K^*}$, the Hilbert space generated by the hyperkernel, and $\|K\|_{K^*}$ is the norm of K over this space.

This is the approach we decided to take, and we were looking at completing the formulation by defining the hyperkernel appropriately and seeing how exactly the SDP approach [10] could be used to get $\mathbf{K_{m \times m}}$ (The method requires both the training and test sets in order to learn the kernel matrix, but we have only the training set, so some kind of cross-validation procedure would be required). At this point, we discovered that the hyperkernel approach to learning the kernel function has very recently been formulated and described by Ong et al. [13]. The basic idea used by them is exactly that which is described above. They have also given certain recipes for constructing hyperkernels, and used these to test the method on several real-world datasets. The results obtained have been fairly good, showing significant improvement over fixed-kernel methods in some cases and slight improvement in others.

The precise formulation of the hyperkernel approach [13], for the standard case of squared-loss error, amounts to minimizing the following regularized functional:

$$\frac{1}{m} \Sigma_{i \in \mathbb{N}_m} (y_i - f(x_i))^2 + \mu \|f\|^2_K + \mu^* \|K\|^2_{K^*} \tag{15}$$

Here, a double minimization has to be done; as f varies over $\mathcal{H}_K$, and as K varies over $\mathcal{H}_{K^*}$. This problem is then converted into an SDP formulation, following the approach referred to earlier [10] . By this means, it is possible to effectively 'learn' the best kernel function, using the given training set; and the use of regularization means that the utility of this kernel can be expected to generalize to other datasets as well.

3.3 Comparing the Two Approaches

We have seen a method for computing an optimal kernel over a compact and convex set, and another one for doing so over a Hilbert space. How do these two relate to each other? Looking at the definition of optimality used in the first case, we find, from (2) that it corresponds, for the square-loss error case, to minimizing

$$\frac{1}{m}\Sigma_{i\in\mathbb{N}_m}(y_i - f(x_i))^2 + \mu\|f\|_K^2 \tag{16}$$

as f varies over $\mathcal{H}_K$, and as K varies over $\mathcal{K}$. For the hyperkernel approach, $\mathcal{K}$ is $\mathcal{H}_{K^*}$; and comparing the functionals (15) and (16), we find that the only difference between the two is the kernel regularization term, $\mu^*\|K\|_{K^*}^2$, which is there for the hyperkernel case but not for the other one. So, the key conceptual difference is that while Micchelli and Pontil's work [12] involves finding an optimal kernel for a specific training set, Lanckriet et al. [10] look to learn a kernel that will perform well in general, not just in a single instance. As stated earlier, the problem of minimizing an unregularized functional of the form (1) is in general ill-posed; however, the results cited in Section 4 show that it can be done for the specific case where the hypothesis space is compact and convex.

So, we conclude that the first approach finds a kernel that works well only on the given dataset, and is restricted to a compact and convex kernel hypothesis space. The second method optimizes the kernel over an RKHS, and finds a more general solution, which should be useful for learning from any training set taken from the specified domain.

4 Support Vector Reduction

Although Support Vector Machines offer state-of-the-art classification performance, they have not been used much for applications where time efficiency is crucial, such as in online settings. This is due to the fact that for large data sets, SVMs return a large number of support vectors, and both the space and time requirements of an SVM classifier scale linearly with the number of support vectors. In fact, a recent result by Steinwart [18] actually theoretically establishes the fact that the number of support vectors returned by an SVM increases linearly with the size of the training set.

However, it would intuitively seem that the support vector set should saturate at some level; once we have enough points to uniquely determine the separating hyperplane, further additions should be redundant. Thus, it seems sensible to try and prune some support vectors from those returned by an SVM; and it seems as if it should often be possible to do so without much loss in terms of generalization ability. Indeed, this has already been demonstrated by the work of Burges [3]; the only reason why his method did not prove to be of much utility was that it was too time-consuming.

We looked at two different ways of reducing the size of the support vector set: by using the concept of span [20] to try and prune vectors in such a way as to minimize the increase in the generalization error bound; and by using the technique of data squashing [7] to reduce the set of support vectors to a smaller set of new, representative points, each with a certain weight attached to it.

4.1 Span-based Support Vector Reduction

Let us say that our objective is to remove points from the support vector set, one by one, so as to try and keep the bound on the predicted generalization error minimal at each stage. From (11), it is clear that the expected value of the generalization error of a given SVM depends on the value of S, the span of the set of support vectors. Clearly, it also depends on some other quantities, namely D, l and ρ, but these will generally change quite predictably as support vectors are removed: l, the size of the training set, will incrementally go down, while D, the diameter of the training data, must also decrease monotonically. The margin, ρ, will increase monotonically, as we are always looking to maximize it, and this can be done to a greater and greater extent as the constraints are lessened (i.e. support vectors are removed). So, we cannot really control the variation in these 3 quantities, as they will follow the same trend irrespective of which vectors we throw out. However, the variation in span is not bound to follow any such trend, and this therefore becomes the key determining factor. So, our idea was to try and purge points in such a way as to minimize the S-span of the resulting set.

Suppose we have n support vectors, $(\mathbf{x}_1, ..., \mathbf{x}_n)$, with optimal multipliers $(\alpha_1^0, ..., \alpha_n^0)$. Let the span of this set, as defined by (10), be denoted by $S(\{\mathbf{x}_1, ..., \mathbf{x}_n\})$. We would like to leave out one of these vectors, say $\mathbf{x}_p$, such that the condition

$$p = argmin_{i \in \mathbb{N}_n} S(\{\mathbf{x}_1, ..., \mathbf{x}_n\} - \{\mathbf{x}_i\}) \tag{17}$$

is satisfied. However, in order to do so, we would need to try out each of the n possible choices, and do the span computation n times on sets of size $n-1$. Since this would be computationally prohibitive, we adopted instead the heuristic of leaving out the point with the highest individual span from amongst the current support vector set, i.e. p was chosen such that

$$p = argmax_{i \in \mathbb{N}_n} S_i \tag{18}$$

was satisfied. Clearly this is not guaranteed to minimize the span of the resulting set, but there's a good chance that it'll do better than most other choices. This is because the point we are leaving out is the determiner of the current span; if we leave out any other point, chances are the span won't decrease, because for that the determiner's span would have to decrease. By leaving out the determiner we maximize the chances of the span decreasing, by assuming that the individual spans of other points will not change too drastically in a single iteration. While we were unable to come up with a strict theoretical justification of this procedure, it works quite well in practice, as can be seen from our experimental results.

We summarize below the steps in our algorithm for span-based support vector reduction:

1. Train the SVM on the current set of points; evaluate it on an independent test set. Let n support vectors be obtained.
2. Compute the span for each support vector, using the Opper-Winther bound:

$$S_p = \frac{1}{(K_{SV}^{-1})_{pp}} \tag{19}$$

   Here K_{SV} denotes the kernel matrix on the support vector set, i.e. $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, where k is the kernel function. This bound gives a good approximation to the span (see [5]), and is easier to compute than using the actual definition.
3. Remove the point with maximum span from the current set. If the set size has reduced to the desired level, stop; else go to step 1.

As a representative example of the working of this algorithm, see Figure 1. This shows the variation of generalization error with reduction in the number of support vectors, for a realization of the UCI 'heart' dataset. The graph shows that although the initial number of support vectors

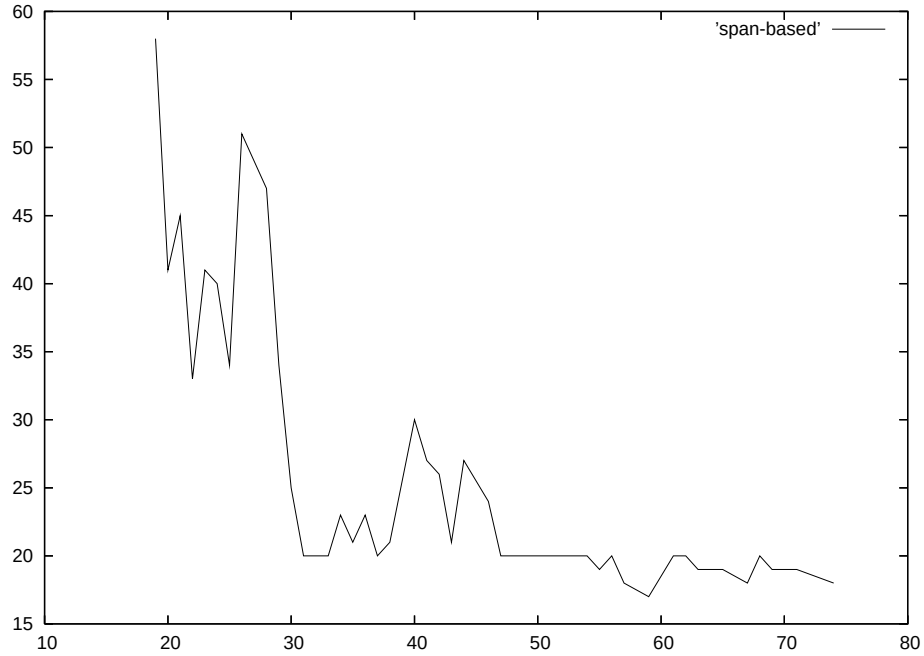

Figure 1: Variation of generalization error (in %) as support vectors are reduced based on span, on realization 1 of the benchmark 'heart' dataset [16].

returned is around 75, we are able to reduce it to 30 without any significant change in the error; only below 30 does it start to shoot up. This demonstrates the efficacy of the algorithm in purging support vectors; similar results were obtained on other data sets.

4.2 Squashing the Support Vectors

The idea of squashing [7] was initially proposed as a way of generating a more concise representation of a large dataset, whilst preserving its statistical properties. This idea was then employed to construct scalable SVMs [14], by squashing the data set before using it for training. Picking up on the same idea, we decided to try using squashing to reduce the size of a given set of support vectors. For this, we employed a slightly modified version of the original squashing idea, known as likelihood-based data squashing [11]. The idea here is to take some estimate of the actual classification function, say f, and then generate a number of 'neighbours' of f, say $(f_1, ..., f_m)$. Each point in the data set is then mapped to a vector of the values of all these functions at that point, i.e. $\mathbf{x}_i \rightarrow (f_1(\mathbf{x}_i), ..., f_n(\mathbf{x}_i))$. These vectors are then clustered into k groups (where k is the number of support vectors desired after squashing). For all the vectors in each cluster, the mean of the corresponding original data points is then taken as the representative for that cluster. The weight of this representative is taken as the number of points in the given cluster. Thus, we obtain a squashed set of k weighted support vectors. The weight can be seen as the frequency, or number of instances, of a given point, and this interpretation can then be used for further learning and squashing operations on the new data set.

In our case, it is natural to take the estimated classification function as the one obtained by

training an SVM on the current data set. This is defined as:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}) \tag{20}$$

In order to generate neighbours of this function, all the α_i values were perturbed by a given small quantity, δ, in both positive and negative directions. Thus, if the original function had the set of coefficients $(\alpha_1, ..., \alpha_n)$, then the two functions obtained by perturbing α_1 would have coefficients $(\alpha_1 + \delta, ..., \alpha_n)$ and $(\alpha_1 - \delta, ..., \alpha_n)$. In this way, a total of $2n$ neighbours of f would be obtained, where n is the size of the current support vector set. These would then be used to generate a set of $2n + 1$ function values for each support vector, and these values would constitute the vectors to be clustered. For clustering, we used the standard k-means algorithm.

To summarize the algorithm:

1. Train the SVM on the current set of points; evaluate it on an independent test set. Let n support vectors be obtained.
2. Generate $2n$ neighbours of the classification function predicted by training the SVM, by perturbing each of the α_i values positively and negatively.
3. Use these neighbours plus the original function to map each support vector to a vector of $2n + 1$ function values.
4. Use k-means clustering (k is some specified number) to cluster the function value vectors obtained in step 3.
5. For each of the k clusters, compute the mean of the support vectors whose corresponding function value vectors lie in that cluster. Take this mean as a new support vector, with weight equal to the number of points in this cluster. This gives us a new set of k weighted support vectors, which we can then interpret as a set of n vectors where each of the k unique points occurs as many times as its weight.
6. Train the SVM on this new set of support vectors, and evaluate it again on the independent test set.

Experiments with this procedure were done on several data sets. The results were quite good for a variety of synthetic data generated by us; see Figure 2 for an example. However, the method did not work as well for some of the real-world UCI datasets. This is probably because the amount of noise in these data sets is significantly greater. The squashing algorithm's performance depends critically on how good the estimated classification function is, and it is not possible to obtain a very good estimate when the original data set has high noise.

4.3 Comparison and Comments

The span-based method seems to work better in general, and is also more efficient time wise. Figure 3 gives a comparison between the performance of the two methods on two different data sets, a synthetic one consisting of 2 3-D 'bells', and a realization of the benchmark 'heart' data set. It can be seen that the squashing method does only a bit worse on the synthetic data, but significantly underperforms compared to the span-based method on the real-world data. As discussed earlier, this is probably due to higher noise in the latter.

The results from both methods do confirm the fact that there is great scope for reducing the number of support vectors returned by an SVM. Till now, the main hurdle to doing so has been time efficiency. The span-based method seems to be promising in this regard; although retraining

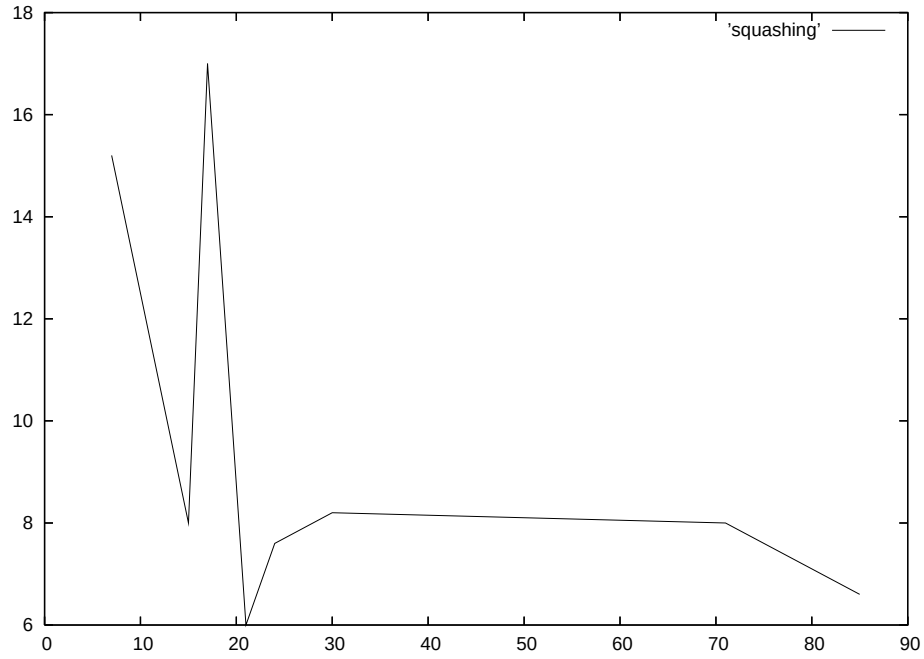

Figure 2: Variation of generalization error (in %) as support vectors are reduced via squashing, on a synthetic two-class data set consisting of two overlapping 3-D Gaussian 'bells', with 10% Gaussian noise added in.

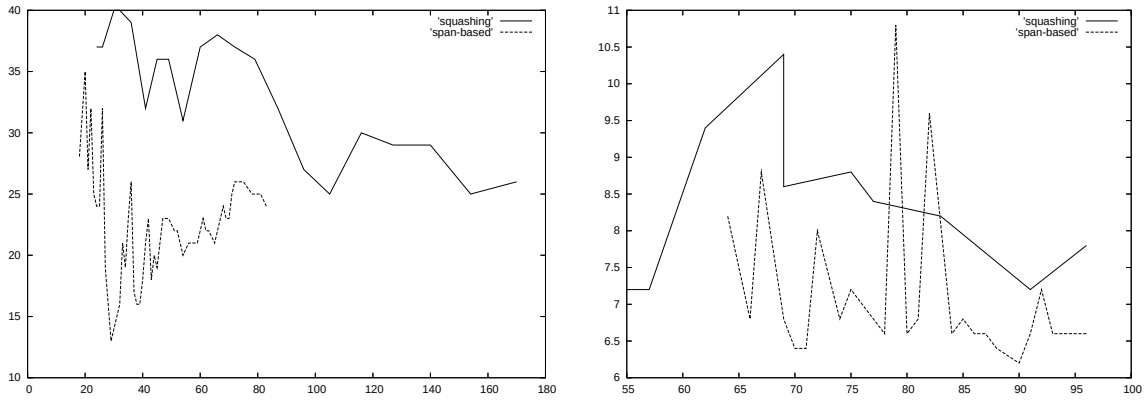

Figure 3: Comparison of variation of generalization error (in %) as support vectors are reduced via span-based and squashing methods. The first graph is for realization 2 of the benchmark 'heart' dataset [16], while the second is for a synthetic two-class data set consisting of two overlapping 3-D Gaussian 'bells', with 5% Gaussian noise added in.

the SVM from scratch each time is quite costly, if we use the decremental learning procedure proposed by Poggio and Cauwenberghs [4], we can adjust the coefficient (α_i) values quite fast, whilst still maintaining the optimality conditions for the SVM. Since the span computation itself does not take too much time, on the whole it is possible to carry out the procedure quite efficiently. A very recent paper by Keerthi et al. [8] proposes a new method of reducing SVM complexity, which works by choosing certain 'basis functions' to represent the training data, rather than computing all the support vectors. Their results are very promising, and they claim to be able to carry out the complexity reduction much faster than any previously proposed methods.

5 Online Learning Applications

In many real-world applications, there arises the necessity of learning from streaming data, which involves new data points coming in continuously at some constant rate. In these scenarios, the amount of information available for training is constantly increasing, and one would like to be able to update the learnt functional in real-time, in order to have the best hypothesis possible at each stage. Also, there is bound to be a memory limitation, so, if an SVM is being used, one has to decide which support vectors to retain and which to discard. In other words, it essentially boils down to the problem we have just been looking at, i.e. support vector purging. So it is natural to try adapting the methods used there for the online context as well. The squashing method is in general too slow to be useful for online learning, but the span-based method was found to work quite well in this setting.

To formalize the online setting, let us say we have a memory limit of N support vectors. The memory starts out empty, and data points keep continuously streaming in. We cannot retrain the SVM from scratch each time a data point is received; that would be prohibitively expensive. The solution is to use some kind of gradient-descent method to adjust the old coefficient (α_i) values, based on the extent to which the new point was misclassified (note that we need not bother about points which are correctly classified by the current function; such points are not support vectors and can be ignored). At the same time, we also need to compute the multiplier for the new point. This goes on as long as the memory isn't full. Once we have N support vectors, we cannot add any more without first discarding an earlier one. This is where the span-based method comes in; we can choose which vector to throw out based on the principle of trying to minimize span. The simplest way to do it would be to leave out the oldest point in the memory; this is essentially what was done by Smola et al. [9] in their proposed algorithm. Although this has the advantage of implicitly accounting for time-varying distributions, it also runs the risk of discarding a more informative point in favour of a less useful one. This is why we decided to try out the span-based idea and see how the two methods compare.

The algorithm we implemented for span-based online learning is summarized below:

1. Let the current set of support vectors in the memory be $((\mathbf{x}_1, y_1), ..., (\mathbf{x}_N, y_N))$, and let their multipliers be $(\alpha_1, ..., \alpha_N)$ (we consider the general case when the memory is full; otherwise, one can just go on adding the incoming support vectors until this state is reached). Let $(\mathbf{x}, y)$ be an incoming data point. We first compute:

$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}) \tag{21}$$

   Here k is the usual kernel function.

2. If $f(\mathbf{x}) = y$, the new point is not a support vector; we ignore it, return to step 1 and wait for the next point. Otherwise, we combine this point with the N old support vectors, and compute the span for each of these $N + 1$ points, using the Opper-Winther bound (19).

3. As per our earlier heuristic, we leave out the point with the highest individual span amongst the current $N+1$ points. If this happens to be the new point, there is nothing to be done; we return to step 1. Otherwise, let's say the point to be left out is $\mathbf{x}_p, p \in \mathbf{N}_N$. We need to make $\alpha_p = 0$, whilst simultaneously adjusting the other coefficients in order to maintain the optimality conditions for the SVM. This is done via the Poggio-Cauwenberghs decremental unlearning procedure [4].
4. Now, we add the new point to the support vector set, i.e. we compute the appropriate multiplier for it. Once again the other α_i values must be adjusted so as to maintain optimality. We now use the Poggio-Cauwenberghs incremental learning procedure [4] to carry out this adjustment.
5. We now have a new set of N support vectors, with one new point added at the cost of one of the earlier points. At this stage we may run the current classifier on an independent test set, to gauge its generalization performance. We then return to step 1 and await the next point.

To try out our algorithm, we ran it on some synthetic datasets, using them as streaming data. We did experiments with both static distribution and time-varying distribution data. For the static distribution case, a memory limit would be fixed (smaller than the training set size), and all the training data streamed in. The final classifier obtained would then be tested on an independent test set drawn from the same distribution. This procedure was repeated for varying values of the memory limit. For the time-varying distribution case, the procedure used was similar, except that the training data's distribution varied gradually as it was streamed in. The test set here would be drawn from the current distribution, i.e. the distribution prevailing at the end of the training data. In this way, the ability of the algorithm to adapt to the variation could be gauged.

We compared our algorithm's performance with two other approaches. One involved simply leaving out the oldest point in the memory whenever a new support vector came in, and using the same incremental and decremental learning/unlearning procedures [4] for coefficient adjustment. We denote this approach by LRU (Least Recently Used). The second comparison was against the algorithm proposed by Smola et al. [9], which is similar to LRU, but uses a different gradient descent method for adjusting the coefficient values, trying to do it in such a way as to minimize the influence of older points. This involves multiplying each α_i by a decay term at each iteration.

The results are shown in Figure 4. It is evident that all 3 methods are subject to random ups and downs to some extent; however, our span-based method seems to be most consistent in its performance as the memory limit is reduced. For the static distribution case, the span-based method clearly seems to outperforms the other two. For the time-varying distribution, the Smola et al. method seems to work better for somewhat larger memory limits, though its error shoots up as the size becomes very small. This relatively better performance is due to the fact that their method is specifically designed to handle changing distributions, by incorporating a decay factor. Our method has no such provision, though it should be possible to modify it so that it gives preference to retaining more recent points. On the whole, the span-based method seems to give better results in an online setting than any other kernel-based online learning method proposed yet. Although not as fast as the simple LRU approach, this algorithm is also reasonably time efficient; the only extra step involved is span computation, which is essentially the inversion of a matrix whose size is bounded by the memory limit. So for small memory sizes in particular, the algorithm competes very well with the other two in terms of time complexity.

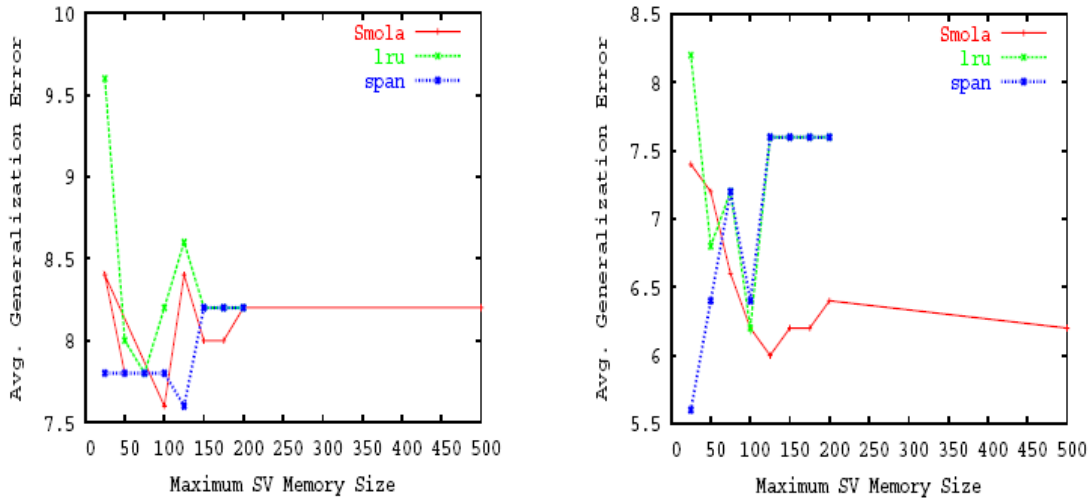

Figure 4: Comparison of variation of generalization error (in %) as memory limit is varied, for the three different online learning methods examined. The first graph is for a dataset with a fixed distribution, while the second is for a time-varying distribution. In both cases, we have generated synthetic two-class data consisting of two overlapping 3-D Gaussian 'bells', with 5% Gaussian noise added in.

6 Conclusions and Future Work

Our chief goals at the start of this project were to look for a method for learning optimal kernels, and to study how advantageous it actually is to search for them. We now know that multiple methods for the learning task have already been proposed; however, the second issue has not yet been fully addressed. Error bounds for these methods have not been given thus far, so it is not clear how much of a performance improvement they can actually guarantee. One key direction for future work is to see if one can quantify the effect of using kernel optimization on the overall learning task. It would also be useful to examine in detail how well the kernels learnt via the hyperkernel approach actually generalize to datasets other than the given training set. So far, there appears to be no concrete evidence in this regard. Another issue is that of efficiency; will the extent of the improvement in learning accuracy be justified by the extra time needed to incorporate kernel learning? In particular, it might be fruitful to explore the idea of improving efficiency by devising incremental approaches to learning the kernel, so that when new elements are added to the training set, the kernel learnt earlier can be used to find a new one, which is optimal given the expanded dataset.

The idea of trying to study incremental approaches to learning the kernel got us onto the general area of kernel-based online learning, and this was the key theme for the second half of this project. We found that there was not much published work in this area, and we had the idea of trying to use the concept of support vector span to come up with a new online learning procedure. The experimental results obtained using this method are quite encouraging; however, the key issue of establishing some sort of performance guarantee in the form of error bounds remains unresolved.

While doing our online learning experiments, one thing that we repeatedly noticed was that there appeared to be a large redundancy in the number of support vectors returned by the SVM method; it seemed that we could reduce the number of vectors significantly without altering clas-

sification accuracy. This got us looking at the general problem of reducing SVM complexity by pruning support vectors; clearly this is essentially the same problem as that encountered in online settings, and our span-based method is equally applicable here. We also looked at an alternative approach to reducing support vectors: via data squashing. This did not work quite as well as the first method, and also turned out to be more time-consuming. In this case too, the span-based method is only a heuristic, and there are no performance guarantees as of now, though the experimental results are fairly useful. Recent work [8] seems to suggest that there is huge scope for SVM complexity reduction, and that it can be done quite efficiently as well. Trying to come up with a rigorous, theoretically justified way of doing so appears to be one of the more exciting research programs in machine learning today.

7 Acknowledgments

I would like to thank Vijaya V. Saradhi, with whom this work was jointly done, and Dr. Harish Karnick for his constant guidance.

References

[1] A. Argyriou, C. A. Micchelli, and M. Pontil. Learning convex combinations of continuously parametrized basic kernels. In *Proceedings of the 18th Annual Conference on Learning Theory (COLT'05)*, June, 2005.

[2] N. Aronszajn. Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68:337–404, 1950.

[3] C. J. C. Burges and B. Schölkopf. Improving the accuracy and speed of support vector machines. In *Neural Information Processing Systems*, 1997.

[4] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. In *Neural Information Processing Systems*, 2000.

[5] O. Chapelle. *Support Vector Machines: Induction Principles, Adaptive Tuning and Prior Knowledge*. PhD thesis, LIP6, Paris, 2002.

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

[7] W. DuMouchel, C. Volinsky, T. Johnson, C. Cortes, and D. Pregibon. Squashing flat files flatter. In *KDD*, 1999.

[8] Sathiya Keerthi, Olivier Chapelle, and Dennis DeCoste. Building support vector machines with reduced classifier complexity. *Journal of Machine Learning Research*, to appear.

[9] J. Kivinen, A. J. Smola, and R. C. Williamson. Online learning with kernels. *IEEE Transactions on Signal Processing*, 52(8):2165–2176, August, 2004.

[10] G. Lanckriet, N. Cristianini, P. Bartlett, L. El Ghaoui, and M. I. Jordan. Learning the kernel matrix with semi-definite programming. *Journal of Machine Learning Research*, 5:27–72, 2004.

[11] D. Madigan, N. Raghavan, W. DuMouchel, M. Nason, C. Posse, and G. Ridgeway. Likelihood-based data squashing: A modeling approach to instance construction. *Data Mining and Knowledge Discovery*, 6:173–190, 2002.

[12] C. A. Micchelli and M. Pontil. Learning the kernel function via regularization. *Journal of Machine Learning Research*, 6:1099–1125, July 2005.

[13] C. S. Ong, A. J. Smola, and R. C. Williamson. Learning the kernel with hyperkernels. *Journal of Machine Learning Research*, 6:1043–1071, 2005.

[14] D. Pavlov, D. Chudova, and P. Smyth. Towards scalable support vector machines using squashing. In *KDD*, 2000.

[15] T. Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the American Mathematical Society*, 50(5):537–544, 2003.

[16] G. Rätsch. Benchmark repository. Technical report, Intelligent Data Analysis Group, Fraunhofer-FIRST, 2005.

[17] B. Schölkopf and A. J. Smola. *Learning with Kernels*. The MIT Press, Cambridge, MA, USA, 2002.

[18] I. Steinwart. Sparseness of support vector machines. *Journal of Machine Learning Research*, 4:1071–1105, 2003.

[19] A. N. Tikhonov. Solution of incorrectly formulated problems and the regularization method. *Soviet Math. Dokl.*, 4:1035–1038, 1963.

[20] V. Vapnik and O. Chapelle. Bounds on error expectation for SVM. *Neural Computation*, 12:2013–2036, 2000.