# Alpha Seeding for Support Vector Machines

Dennis DeCoste
Machine Learning Systems Group
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive; Pasadena, CA 91109
http://www-aig.jpl.nasa.gov/home/decoste/
decoste@aig.jpl.nasa.gov

Kiri Wagstaff
Department of Computer Science
Cornell University
4156 Upson Hall; Ithaca, NY 14853
http://www.cs.cornell.edu/home/wkiri/
wkiri@cs.cornell.edu

## ABSTRACT

A key practical obstacle in applying support vector machines to many large-scale data mining tasks is that SVM training time generally scales quadratically (or worse) in the number of examples or support vectors. This complexity is further compounded when a specific SVM training is but one of many, such as in Leave-One-Out-Cross-Validation (LOOCV) for determining optimal SVM parameters or as in wrapper-based feature selection. In this paper we explore new techniques for reducing the amortized cost of each such SVM training, by seeding successive SVM trainings with the results of previous similar trainings.

## Categories and Subject Descriptors

I.5.5 [**Computing Methodologies**]: Pattern Recognition-Implementation

## General Terms

support vector machines, classification, training speed-ups

## 1. INTRODUCTION

Recent progress on speeding up the training time for support vector machines (e.g. [7],[5]) has made SVM's practical now for training sets that are fairly large. However, the time complexities of those approaches are still typically quadratic in the number of examples ($N$) in the training data set. This is especially problematic in a data mining context, due both to commonality of enormous data set sizes and to the frequent need for high-quality model selection over many candidate SVM's.

We therefore seek methods which can reuse previous results from similar SVM's in order to amortize training costs. In the best case, this could lead to amortized SVM training costs which are linear in $N$. For example, Leave-One-Out-Cross-Validation (LOOCV) estimates of generalization error for a data set of $N$ examples involve $N$ trainings, each involving $N-1$ training examples , thus leading to cubic overall time complexity. If each SVM for each of the size $N-1$ data sets could be intelligently initialized from the result of the SVM trained on all $N$ examples, only a small amount of additional work might be required for each. The overall cost might well remain quadratic in $N$ (i.e. dominated by the cost of the SVM trained on the full data set) — and thus effectively have cost linear in $N$ for each of the $N$ SVM's trained for the different size $N-1$ data sets.

An underlying motivation of our work is to try to bring SVM's substantially closer to the fast near-linear complexity of LOOCV using k nearest-neighbors, (a factor in k-NN's popularity in practice), while retaining the advantages of SVM's (e.g. maximum margins).[1]

After reviewing the basic aspects of SVM classification, we present a variety of "alpha seeding" methods for reducing SVM training time. We then present some empirical results which illustrate the potential promise of such alpha seeding and help us begin to understand the tradeoffs involved. Although we have not yet achieved linear amortized costs, our results appear promising towards that effort, as well as of practical use in their own right.

## 2. SUPPORT VECTOR MACHINES

Support vector machines [9, 10] represent a relatively new and promising approach to machine learning. Recent work has established SVM's as providing state-of-the-art performance on classification and regression tasks across a variety of real-world applications (e.g. see [9] and [4]).

In this paper, we focus on SVM's for binary classification [1]. Each label is valued either "+1" or "-1", indicating either a positive or negative example, respectively.

Let $X_A$ be an $n_A$ by $D$ matrix representing the training set and $X_B$ be an $n_B$ by $D$ matrix representing the test set, where $D$ is the dimensionality of the input space (i.e. $D$ features) and $n_A$ and $n_B$ are the number of training and

[1] For Euclidian distance, complexity logarithmic in $N$ is often achieved for k-NN, using indexing schemes such as k-d trees. However, for the general distance metrics employed within SVM kernel methods [8, 3] sub-linear performance for k-NN's is not as obviously achieved.

test examples, respectively. Let $L_A$ be a vector of the $n_A$ known labels for the training set and $L_B$ be a vector of the $n_B$ actual (often unknown) labels for the test set. Let $y_B$ be a vector of the $n_B$ label predictions of the automated classifier for the test set $X_B$.

The following constrained quadratic optimization (QP) problem is commonly used to train a SVM classifier:

$$\begin{array}{ll} \textit{maximize:} & \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{N} \alpha_i\alpha_j L_i L_j K(x_i, x_j) \\ \textit{subject to:} & 0 \le \alpha_i \le C,\ \sum_{i=1}^{N} \alpha_i L_i = 0, \end{array}$$

using notational simplifications: $N = n_A$, $L = L_A$, and $x_i$ is i-th example (row) in $X_A$.

The SVM prediction, for example $x$ (vector of size D), is:

$$f(x) = sign(\sum_{i=1}^{n} \alpha_i L_i K(x, x_i) + b),$$

where scalar $b$ (bias) and vector $\alpha$ (of size $N$) contains the variables determined by the above QP optimization. Thus, the test predictions $y_B$ are $f(x)$, for each $x$ in $X_B$.

$K(x_i, x_j)$ represents a *kernel* which implicitly projects two given examples from $D$ dimensional input space into some (possibly infinite) feature space. The simplest is the *linear* kernel, implemented as a simple dot product:

$$K(u, v) \equiv u \otimes v \equiv \sum_{i=1}^{d} u_i \cdot v_i.$$

The *polynomial* kernel is defined by a non-linearly squashed dot-product of the following form:

$$K(u, v) \equiv (u \otimes v + r)^d,$$

with polynomial degree parameter $d$. Varying the continuous offset parameter $r$ changes the relative weighting of the (implicit) terms in the nonlinear polynomial feature space.

*Support vectors* are those training example vectors for which $\alpha_i > 0$. As can be seen from the above summation used to generate predictions, a zero $\alpha_i$ means that the i-th training example does not contribute to the prediction. In SVM applications often only 10% or less of the training examples become supports. Such sparsity is a key property of SVM's that helps them avoid overfitting noise. A general rule of thumb is that the expected test error of the SVM is proportional to the ratio of the number of support vectors to the number of all training examples.

Parameter $C > 0$ defines a *soft margin* — a trade-off between regularization (sparsity of non-zero alphas) and training errors. All misclassified training examples, for example, end up with alphas at C. An appropriate value for C is typically determined via cross validation.

# 3. TYPES OF ALPHA SEEDING

Existing SVM methods initialize all alpha ($\alpha$) values for the QP optimization to 0. We use the term *alpha seeding* to refer to any method which provides initial estimates of the alpha values. We restrict ourselves to methods which start each SVM training with *feasible* alphas (i.e. which satisfy the bounds and the single equality constraint), although it is conceivable that infeasible seeds may sometimes be useful for specific SVM training algorithms. Specifically, we investigate seeding methods which make use of final output alphas from one training to initialize a similar one.

To motivate our work and establish a framework, below we discuss a variety of ways in which alpha seeding can be used to improve various aspects of SVM training. In Section 4, we empirically explore some of these in more detail.

All the tasks for which we introduce alpha seeding methods can be solved without seeding (i.e. just start each with zero alphas). Thus, alpha seeding offers no new theoretical advance, as, say, a new type of SVM kernel might. Instead, the goal of alpha seeding is drastically faster convergence to the final alpha values for the SVM problem(s) of interest. However, it is important to keep in mind that resource allocation is almost always a concern in practice. For example, if one can speed the SVM training for one kernel or C value by a factor of 10, one may be able to search for the optimal of ten different types of kernels (or C values) in the same fixed available overall training time.

It is also useful to keep in mind that all of these approaches to alpha seeding can amortize the cost of kernel computations across the entire set of of SVM trainings. Dot-product caches are common even for single SVM trainings, as in most practical SVM trainers (e.g. [5]). Our alpha seeding techniques exploit dot-product caches even further, with later trainings often requiring no additional kernel computations. When input dimensionality $D$ is large, these savings can be substantial (typically more than 200% versus no cache).

A fundamental issue is how alphas from a previous training should be adapted into appropriate seeds for the next training. As we shall explore, there are typically much more effective approaches than simply passing the alphas unchanged between trainings.

The key issue determining whether a given alpha seeding method is effective for a given task is, of course, whether the sum of the training costs over the sequence of successively seeded SVM's is lower than the cost of directly training the non-seeded SVM of interest. We will explore that issue in Section 4, after first discussing the various methods.

## 3.1 Computing Actual LOOCV Error

One of the simplest and yet effective alpha seeding methods is for efficient LOOCV estimation of generalization error. LOOCV requires $N$ SVM trainings, where the i-th SVM is tested on only the i-th example and is trained on the $N-1$ other examples. Unlike other methods below, each such case is for fixed parameters (e.g. for given $C$, type of kernel, etc.). Doing multiple LOOCV's, for various parameter values, provides a popular empirical-based means of model selection.

SVM theory provides estimates of the worst case bounds on the LOOCV error, such as the fraction of training examples which become support vectors. However, since such bounds are necessarily loose, it can be useful for accurate model selection to compute the actual LOOCV error, especially if it can be obtained efficiently.

Our alpha seeding approach to LOOCV is as follows. First, train the SVM for all $N$ examples. Denote the resulting alphas as $\beta$. For each of the examples ($i$) out of the full $N$, pretend in turn that that i-th one is not in the data set.[2] If $\beta_i$ is already 0, then simply classify this i-th example as the full SVM does (and record if it disagrees with $L_i$). Otherwise, initialize the $N$ alphas ($\alpha$) to be those of $\beta$ and set $\alpha_i$ to 0 (i.e. forget it). In that case, the equality constraint $\sum_{i=1}^{N} \alpha_i L_i = 0$ is violated, by a residual of magnitude $\beta_i$. To re-establish the equality, we must distribute that residual to some of the other alphas. Finally, after training the i-th SVM from the so-adjusted alphas $\alpha$, we classify the i-th example (and record if it disagrees with $L_i$).

We have found that a simple and yet rather effective method is to redistribute the residual among all the *in-bound* alphas (i.e. those greater than 0 and less than C). A key motivation is that modern SVM trainers tend to work on in-bound alphas before re-examining at-bound ones. This is because generally once an alpha reaches 0 or C it will tend to stay there during the remainder of an SVM training.

We have explored various schemes for redistributing the residual among the in-bound alphas. One which routinely performs well, although not the best in every case, is to uniformly add an equal portion of $\beta_i$ to each in-bound alpha $\alpha_j$ for which its corresponding example $j$ is in same class as the hold-out (i.e. same label $L_i$). That is, add $\frac{\beta_i}{z}$ to each, where $z$ is the number of other examples of that class with in-bound alphas. The exception is that if this causes some alpha to reach (i.e. want to exceed) the limit $C$, then any remaining residual is (uniformly) redistributed among the remaining in-bound alphas of that class, until all residual is gone. We call this scheme *uniform same-class residual redistribution*, and report results with it in Section 4.1.

### 3.2 GrowC: Quick Training for Large C

A more complex alpha seeding method involves training SVM's using successively larger C values. It is commonly observed in SVM literature that larger C values tend to require substantially more training time than smaller values. However, we theorized that initial training with a smaller C could quickly identify approximate alpha weights which later trainings with larger C's would be able to refine.

More precisely, let $S = [C_1, \ldots, C_n]$ where $C_i < C_{i+1}$ be a training schedule that produces correct alpha weights for $C_n$, the target value of C. We will refer to the training phase that uses $C_i$ as $S_i$. The GrowC approach takes the alphas produced at the end of $S_i$ and uses them as seeds for $S_{i+1}$.

The heart of any such strategy relies on determining an effective schedule for growing C. Our goal in this work is to establish that good schedules do exist. We defer in-depth study into automatically producing them to future work.

Another key issue involves adjustments to the alphas between training phases. When moving from $S_i$ to $S_{i+1}$, the range of allowable alpha values expands from $[0 \ldots C_i]$ to $[0 \ldots C_{i+1}]$. There are several options available. The alphas from $S_i$ can be passed unchanged to $S_{i+1}$. Alternatively, the $S_i$ alphas that are at $C_i$ can be moved to $C_{i+1}$. A third alternative is to scale all of the alphas into the new range. Lastly, a more complex (possibly adaptive) method could adjust only those alphas that are likely to move from their $S_i$ values. In Section 4.2, we compare the results of the first three options empirically and demonstrate the importance of good choices for alpha adjustment between training phases.

### 3.3 Kernel Parameter Via Cross Validation

Another natural use of alpha seeding is for successive cross validations over some range of settings for a kernel parameter. Previous work with Kernel Adatron SVM trainers [2] showed this can be effective, often not costing much more to train for a large number of parameter values than for the first one.

### 3.4 Heuristically Guessing Initial Alphas

Alpha seeds need not be based on previous trainings of very similar SVM's. For example, they could be based on geometrical arguments for why a given example is likely to be support vector or not, or likely to be at C (i.e. a noisy example). Guessing which examples will be at 0 or C can be particularly useful for many SVM training methods, since such at-bounds cases can often be ignored in many iterations of those algorithms.

A particular method in this area which we have explored is training a SVM using a linear kernel and then using those alphas to seed training a SVM for some target nonlinear kernel. The intuition is that for problems which are only slightly nonlinear, such seeds can be very close to optimal for the nonlinear case as well. This idea is especially appealing given the substantial time savings possible for linear kernels, due to the feasibility of folding all $N$ alphas into only $D$ weights necessary to evalute the SVM output for any example in the linear special case.

## 4. EXAMPLES

To empirically explore alpha seeding, we modified two common SVM algorithms, our implementation of $SMO$ [7] (with improvements of [6]) and the freely available $SVM^{light}$ [5]. Our modifications include taking seed alphas as arguments, instead of beginning training from (default) zero alphas.

For our initial experiments to report in this paper we selected the UCI **Adult** data set, since a fair amount of related work with this data set has already been published using the $SMO$ and $SVM^{light}$ algorithms. In particular, for direct comparison we used Platt's discretized versions, consisting of 123 binary input attributes and various subsets of the full set of 32562 [7]. All tests were performed on a 450Mhz Sun Ultra 60 workstation with 2 Gb of RAM.

### 4.1 LOOCV Results

For LOOCV tests, we used the smallest subset Platt reported on in his work [7], which consists of 1604 examples.

[2]For efficiency, we do not actually destroy the original data set, but instead have refined our SVM algorithms to allow ignoring one selected example during the QP optimization.

Figure 1 (left) shows the cumulative run times for standard SVM (zero alphas for each of the $N$ LOOCV retrainings) and our uniform same-class residual redistribution LOOCV alpha seeding method (as described in Section 3.1). Our method was nearly 5 times faster (1733 vs 380 secs).

The training time for full data set was 2.86 secs and resulted in 714 out of 1604 examples being support vectors. The LOOCV training for each of the 714 hold-outs which were support vectors each took roughly the amount of time as that for full training: mean 2.943 secs, standard deviation .2923 secs, maximum 4.51 secs, and minimum 2.24 secs. Using our alpha seeding, training times for the support vector hold-outs were faster: mean 0.6452 secs, standard deviation .2245, maximum 1.54 secs, and minimum 0.22secs.

Both methods, of course, computed the same LOOCV error rate (16.55%), since their only difference is in speed of convergence. It is interesting to confirm that this rate is far below the (well-known to be loose) LOOCV error estimate bounds (44.51%) that the standard ratio of support vectors divided by the number of examples would suggest.

Figure 1 (right) illustrates why our method performed much better than a standard non-seeded method. It plots all $N$ training times, sorted from smallest to largest for both methods. LOOCV for the 890 non-support vector examples requires no retraining, indicated by a majority of zero training times. For the 714 support vector examples, there is substantially more area under the curve using zero seeds versus redistribution-based seeds. Most zero-seed trainings required roughly equal time — about the same as the initial training (2.86 secs). Each of the 714 redistribution-seeded trainings were faster (slowest was 1.54 secs).

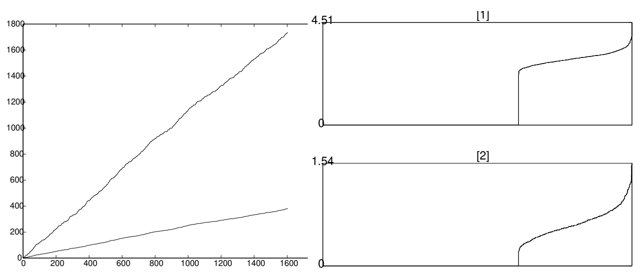


**Figure 1: SMO LOOCV train times (Adult1).** **Left**: Cumulative time (y-axis) over $N = 1604$ LOOCV trainings (x-axis). Higher curve is standard SVM (zero alpha seeds). Lower curve is our alpha seeding method (Section 3.1). For linear kernel, with C=1, for UCI Adult1 data set. **Right**: sorted times for alpha seeds of zeros (top) vs redistribution-based (bottom). Note different y ranges.

## 4.2 GrowC Results

For both our modified $SMO$ and $SVM^{light}$ algorithms, we experimented with several schedules for gradually growing C. In general, we observed that alpha seeding obtained dramatic reductions in total runtime for both algorithms. The Adult data set we used for these experiments is referred to as "Adult small" in [7], consisting of 11221 training examples.

We have verified that the number of bound and in-bound alphas we obtain correspond to those reported by Platt. All runs used a linear kernel and runtimes are averaged over five trials. We also made use of the cache that stores kernel computations, so that they need not be recomputed. This cache persists over each training phase $S_i$ (after the first in a sequence of trainings), to make it comparable to training from scratch (where the cache is available throughout the course of training).

Section 3.2 outlined four options for how to seed $S_{i+1}$ using the results of $S_i$. Below reports how the first three perform.

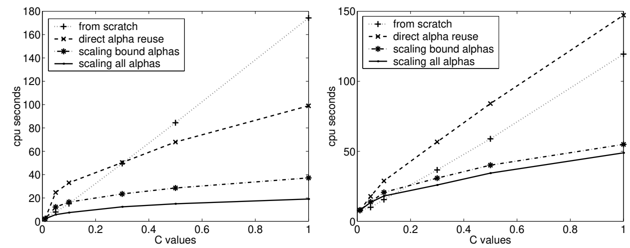


**Figure 2: GrowC times ($SMO$,$SVM^{light}$): seedings.**

### 4.2.1 *Direct Alpha Reuse*

Using successively larger values of C and seeding each iteration with the alphas found at the end of the previous one does not always yield runtime benefits, as shown in Figure 2. For C values less than 0.3 for $SMO$ and for *all* tested C values for $SVM^{light}$, it is actually more expensive to use this form of alpha seeding than to proceed from scratch. A smaller $C_i$ restricts which possible alpha values are explored, thus limiting the initial runtime. But when these alphas are used as seeds for $S_{i+1}$ with a larger $C_{i+1}$, a lot of time can be spent adjusting them gradually into the larger range. This is especially true for alphas that are at $C_i$ at the end of $S_i$ — it is likely that they will end up being at $C_{i+1}$ at the end of $S_{i+1}$, but it may take a long time to push them that far.

### 4.2.2 *Scaling Bound Alphas*

This observation leads naturally to the second option: at the end of $S_i$, change all alphas that have a value of $C_i$ (the "bound" alphas) to the new $C_{i+1}$ directly. The fact that an alpha is bound in $S_i$ often indicates that it will be bound in $S_{i+1}$. If so, a lot of time can be saved by immediately jumping to the new boundary value, $C_{i+1}$. Figure 2 shows that this improves runtime for $SMO$ over Direct Alpha Reuse, but can still (for C less than 0.1) be more expensive than training from scratch. Similar trends appear for $SVM^{light}$.

### 4.2.3 *Scaling All Alphas*

Our next option is to scale each alpha value produced by $S_i$ into the new range allowed in $S_{i+1}$. This is accomplished by multiplying each alpha value by $\frac{C_{i+1}}{C_i}$. This has the effect of sending all alphas at $C_i$ to the new value $C_{i+1}$ and spreading the rest of the in-bound alphas into the new range. In addition, it keeps zero-valued alphas at 0. As shown in Figure 2, this strategy achieves the greatest improvements in runtime. Training $SMO$ from scratch for $C = 1.0$ requires about 175 seconds. Scaling All Alphas requires just 19 seconds, a savings of 89% of the total runtime. For $SVM^{light}$, training from scratch requires 120 seconds, but Scaling All Alphas requires only 49 seconds (59% savings).

As noted above, the choice of schedule $S$ impacts the effectiveness of alpha seeding. The seeding results in Figure 2 were all produced using schedule $S_1$ = [0.01, 0.05, 0.1, 0.3, 0.5, 1.0], which was experimentally determined to work well with the Adult data. Experiments with other schedules indicate that more graduations tend to yield greater overall benefits for $SMO$, but the reverse trend appears for $SVM^{light}$. Further investigation is required to fully understand what strategies for constructing training sequences are of most use to each algorithm.

Clearly, intelligent adjustments to alphas between training phases are essential. It is possible that better alpha adjustment strategies could result in even larger runtime improvements for alpha seeding. In addition, these results were all gained while using a linear kernel; other kernel types may require different alpha seeding (or C scheduling) strategies.

#### *4.2.4 Larger C Values*

Our results demonstrate significant improvements in performance for $SMO$ for C values less than or equal to 1.0. Most of those C values are accompanied by a similar improvement for $SVM^{light}$. However, it is not usually possible to predict ahead of time what a good C value will be for a problem. Therefore, good performance over a variety of C values is important. In particular, large C values have been a challenge for SVM algorithms. In separate experiments, we were able to train on the Adult data with a C of 500 in under 85 seconds.[3] It took $SVM^{light}$ and $SMO$ over 10 minutes to train with such a large C.[4] Clearly, alpha seeding reduces these previously computationally-expensive trainings to reasonable durations.

Another benefit of using a seeding approach is that it can significantly reduce the time required to find a good value for C on a new data set. Instead of performing a series of trainings, all from scratch, with various values of C, we instead obtain multiple results together, by using a training sequence that contains many C values of interest. The SVM produced for each intermediate $C_i$ can be used to compute a test set error, selecting the value giving lowest test error.

## 5. CONCLUSIONS

Our results suggest that alpha seeding is a promising way for speeding up SVM training. Although our speedups are often essentially constant ones, these factors are often much larger than the impact of other recently published methods for speeding up SVM's (e.g. bias intervals in [6] and "shrinking" in [5]). Thus, they are of significant practical importance.

There are many directions for future work. One is to understand the nature of the best alpha seedings better, toward speedups that are typically more than nearly-constant ones (ideally, with amortized linear time cost for each SVM training). Another is to understand sensitivity issues, such as how close to the final values the alpha seeds have to be, for significant speedup gains to be realized. Yet another is to develop means for automatically finding good growth schedules for any given task, for our GrowC method.

We also plan to contrast our efficient LOOCV alpha seeding approach with Leave-One-Out SVM's (LOOSVM's, [11]). Empirical studies of the computational costs of LOOSVM's are not yet available, so it is unclear when each is most effective — explicit selection from a set of C values as in our case versus folding the search for C within the optimization problem (as in LOOSVM's).

[3] The training sequence used was [0.01, 0.05, 0.1, 0.3, 0.5, 1.0, 3.0, 5.0, 10, 15, 20, 30, 50, 100, 500].

[4] We terminated the training for each one at that point.

## 6. ACKNOWLEDGEMENTS

This research was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

## 7. REFERENCES

[1] C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2), 1998.

[2] N. Cristianini, C. Campbell, and J. Shawe-Taylor. Dynamically adapting kernels in support vector machines. Technical Report NeuroCOLT Technical Report NC-TR-98-017, Royal Holloway College, University of London, May 1998.

[3] Dennis DeCoste and Michael Burl. Distortion-invariant recognition via jittered queries. In *Computer Vision and Pattern Recognition (CVPR-2000)*, June 2000.

[4] Isabelle Guyon. Online SVM application list. (See http://www.clopinet.com/isabelle/Projects/SVM/applist.html.).

[5] T. Joachims. Making large-scale support vector machine learning practical, 1999. In *Advances in Kernel Methods: Support Vector Machines* [9].

[6] 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 CD-99-14, Dept. of Mechanical and Production Engineering, National University of Singapore, 1999.

[7] John Platt. Fast training of support vector machines using sequential minimal optimization, 1999. In *Advances in Kernel Methods: Support Vector Machines* [9].

[8] B. Schölkopf, A. Smola, and K.R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. Technical report no. 44, Max-Planck-Institut for Biologische Kybernetik, Tübingen, Dec 1996.

[9] B. Schoelkopf, C. Burges, and A. Smola. *Advances in Kernel Methods: Support Vector Machines.* MIT Press, Cambridge, MA, 1999.

[10] V. Vapnik. *The Nature of Statistical Learning Theory.* Springer Verlag, New York, 1995.

[11] Jason Weston. Leave-on-out support vector machines. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.