The Support Vector Tree

Antti Ukkonen

HIIT, Helsinki University of Technology**
antti.ukkonen@hiit.fi

Abstract. Kernel based methods, such as nonlinear support vector machines, have a high classification accuracy in many applications. But classification using these methods can be slow if the kernel function is complex and if it has to be evaluated many times. Existing solutions to this problem try to find a representation of the decision surface in terms of only a few basis vectors, so that only a small number of kernel evaluations is needed. However, in all of these methods the set of basis vectors used is independent of the example to be classified. In this paper we propose to adaptively select a small number of basis vectors given an unseen example. The set of basis vectors is thus not fixed, but it depends on the input to the classifier. Our approach is to first learn a non-sparse kernel machine using some existing techique, and then using training data to find a function that maps unseen examples to subsets of the basis vectors used by theis kernel machine. We propose to represent this function as a binary tree, called a support vector tree, and devise a greedy algorithm for finding good trees. In the experiments we observe that the proposed approach outperforms existing techniques in a number of cases.

1 Introduction

Classification is a fundamental problem in machine learning. Typical research on classification methods concentrates on improving either the scalability of the learning algorithm, or accuracy of the resulting classifier, or both. These properties are important problems in most, if not all applications. However, in some cases the running time of the classifier itself can be of importance. Speech recognition and packet classification in IP networks are classical examples of applications where it is crucial that classification can be carried out in "real time". Another example are information retrieval systems that use machine learning algorithms to classify documents to relevant and non-relevant ones. If the classifier is applied to every document that match a given query, a single evaluation of the classifier has to be very fast, as there can be tens of thousands of such documents. In such cases it is thus necessary to resort to relatively simple classifiers.

However, we might obtain a better classification accuracy with kernel or ensemble based methods that are computationally slower. In this paper we study the problem of speeding up classification using kernel machines. We use the

** The work was conducted while the author was visiting Universitat Pompeu Fabra / Yahoo Research Barcelona.

nonlinear support vector machine (SVM) [23] as an example, but our approach is applicable to any classifier that uses a similar decision rule. In particular, we can consider any algorithm where the decision is based on the following sum:

$$\text{class}(\mathbf{x}) = \text{sign}\Big(\sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x}) + \theta \Big). \quad (1)$$

The set S together with the function g represent the classifier. Evaluating this sum can be slow if the set S is large, and if the function g is computationally complex. One solution is to explicitly restrict the size of S when learning the classifier. This is a common approach, as it can also result in faster learning algorithms. The idea we propose in this paper is rather different. *Instead of computing the sum in Equation 1 over the entire set S, we compute it over the set $f(\mathbf{x}) \subset S$ that depends on the example $\mathbf{x}$ we are classifying.*

We consider the following approach: First we find the set S representing a kernel based classifier, such as an SVM. This can be done using any standard algorithm. Given S we learn a function f that maps any given example $\mathbf{x}$ to a subset $f(\mathbf{x}) \subset S$. This function can be learned either with the same training data that were used to find the set S, or using a different set of examples. When classifying $\mathbf{x}$, we compute the sum in Equation 1 only over the basis vectors in $f(\mathbf{x})$. To represent the function f, we propose a binary tree that induces a disjoint partition of the feature space. Examples belonging to the same region are mapped to the same subset of S. We call this tree the *support vector tree*.

The rest of this paper is structured as follows: This section concludes with a discussion of related work and the detailed contributions of this article. In Section 2 we give a general definition of the problem, while the support vector trees are described in Section 3. Finding an optimal tree is likely to be hard, and in Section 4 we propose a greedy heuristic with polynomial running time that finds good trees in practice. In the experiments (Section 5) we compare our method with recent related work, and Section 6 is a short conclusion.

1.1 Related Work

Kernel machines and in particular SVMs [23] have been studied considerably for the past fifteen years. A complete review of this work is obviously beyond the scope of this paper. However, we try to mention most articles that are relevant considering the main objective of this paper: *speeding up classification with SVMs*. Also, for the basics of SVM learning we recommend the excellent tutorial by Burges [4], as we do not discuss these here.

Finding sparse SVMs is an old and well studied problem. For interesting initial work on the topic we refer the reader to [5, 3]. Burges and Schölkopf [5] devise a post-processing algorithm for finding a reduced set of basis vector given the S of support vectors. Most of the approaches that followed differ from [5] and our paper by formulating a version of the SVM learning problem that attempts to *directly* find a sparse solution. Examples of older work on this kind of techniques include [20], [11], [22], and [19]. More recently, in [24] Wu et. al. discuss another

direct method for building sparse kernel machines. They report experimental results where the accuracy of the full SVM is in some cases achieved using only a very small fraction (5%) of the original support vectors. Unlike other related work [8] proposes an ensemble-like method.

A recent paper that studies the problem of sparse SVM learning is [16]. While the main motivation for [16] seems to be making SVM training more scalable, the proposed algorithm also has the property of giving solutions that can have a considerably smaller number of support vectors without a significant decrease in accuracy. In the experiments of [16] it is shown that the number of basis vectors can be reduced by two orders of magnitude without affecting the accuracy of the resulting classifier. This is similar to the results in [24]. Another interesting property of [16] is that the set S may contain vectors outside the training data. We compare our method against the algorithm proposed in [16].

Most of the methods for sparse SVM learning let the learning algorithm automatically determine the number of support vectors. However, in some applications it is useful to be able to set the desired number of support vectors in advance. An algorithm that admits this is proposed in [10]. Also the method we describe here allows the "budget" to be specified in advance, as do the methods of [24] and [16].

1.2 Contributions of this paper

- We propose a method to speed up classification using kernel machines by using only a subset of support vectors. This subset is a function of the example to be classified.
- We propose a method called the support vector tree to efficiently select the support vectors given an unseen example. The method resembles a decision tree but differs on a number of important aspects.
- We propose a greedy algorithm for learning a support vector tree given training data. An analysis based on the Master theorem [9] shows that the running time of this algorithm is at least of order $O(n^3)$ where n is the size of the training data.
- We describe experiments where the support vector tree is compared with a classical nonlinear SVM and a state-of-the art algorithm for learning sparse SVMs on a number of benchmark data sets.

2 Problem definition

We continue with some formal definitions. Let Ω be a universe of objects. Usually we let $\Omega = \mathbb{R}^n$, but the proposed method is to a large extent oblivious to the type of input examples. Let $S \subset \Omega \times \mathbb{R}$ be a set of objects from Ω together with a *weight* associated with each object. That is, we have $S = \{(\mathbf{s}_j, a_j)\}_{i=1}^m$. Moreover, denote by $g : \Omega \times \Omega \to \mathbb{R}$ a function mapping pairs of objects from Ω to the set of reals. We can think that the set S represents for example a nonlinear SVM. The function g is the kernel function, and the $(\mathbf{s}_j, a_j)$ pairs are

the support vectors and their weights. We consider classifiers where an example $\mathbf{x} \in \Omega$ is assigned to the class $+1$ or -1 depending on the sign of the sum in Equation 1.

Using this sum to classify a new example $\mathbf{x}$ requires m evaluations of the function g. This may be a problem in some applications if the set S is large and computing g is slow. This can happen if g is e.g. a string kernel [18, 21]. As a remedy, most previous approaches to speed up classification with kernel machines look for sparse solutions to the learning problem. Roughly put, the idea is to find a set S', so that $|S'| \ll |S|$, and

$$\sum_{(\mathbf{s}'_j, b_j) \in S'} b_j g(\mathbf{s}'_j, \mathbf{x}) \approx \sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x}).$$

A common property of the previous approaches is thus that all input instances are classified using the same set S'. However, it is easy to imagine that if S' is selected separately for each unseen example $\mathbf{x}$, we may obtain a better approximation, and possibly need a smaller number of evaluations of the function g. More precisely, instead of computing the sum over a fixed set S' when classifying $\mathbf{x}$, we compute it over a set S' that depends on the input $\mathbf{x}$. We express this idea more formally in the rest of this section.

Let S and g be as defined above, and let $D \subset \Omega$ be a set of input examples. The examples in D can be labeled or unlabeled. Furthermore, let Φ be a family of functions that map objects from the set Ω to subsets of the set S. The general formulation of our problem is as follows:

Problem 1. Given the data $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the set $S = \{(\mathbf{s}_1, a_1), \ldots, (\mathbf{s}_m, a_m)\}$, the function g, and the family of functions Φ, find the function $f \in \Phi$ that minimizes the cost

$$\sum_{\mathbf{x}_i \in D} \underbrace{\left(\sum_{(\mathbf{s}_j, a_j) \in f(\mathbf{x}_i)} a_j g(\mathbf{s}_j, \mathbf{x}_i) \; - \sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x}_i) \right)^2}_{\mathrm{c}(\mathbf{x}_i)}. \tag{2}$$

Note that we are approximating the sum instead of only its' sign. There are two reasons for this. First, we expect this to better retain the generalization ability of the resulting classifier. Second, in some applications we are not only interested in the sign, but also the exact value of the sum. This is the case for instance if the kernel machine is to be used for ranking [13].

Clearly Problem 1 is under-constrained in the sense that if the family Φ is not chosen carefully, we may end up with a trivial solution that simply maps every $\mathbf{x} \in \Omega$ to the set S. This is not meaningful considering that we want to reduce the number of evaluations of the function g. Therefore, the problem is interesting only if we restrict the kinds of functions Φ may contain.

From a practical standpoint we are interested in functions f such that $|f(\mathbf{x})| \leq k$ for all $\mathbf{x}$ for some fixed k. To solve Problem 1, we can consider each $\mathbf{x}_i \in D$ separately, and for each find a subset S' of S, $|S'| \leq k$, that minimizes $\mathrm{c}(\mathbf{x}_i)$.

This can be seen as a variant of the NP-complete SUBSET-SUM problem, where the question is to find a subset of a given set A of numbers that sum up to a given number B [12]. In our case we have $A = \{a_j g(\mathbf{s}_j) \mid (a_j, x_j) \in S\}$ and $B = \sum_{a \in A} a$. (SUBSET-SUM is usually defined for integers. We can scale and subsequently round the values in $\mathbf{X}^{D,S}$ so that the input is integer valued.) Finding an optimal $f(\mathbf{x}_i)$ is thus unlikely to be easy. And even if we could find the optimal subset for each instance in the training data, we still need to be able to use the function f with unseen examples. One solution would be to store all of D together with the $f(\mathbf{x}_i)$s, and for an example $\mathbf{x} \notin D$ let $f(\mathbf{x}) = f(\mathbf{x}^*)$ where $\mathbf{x}^* = \arg\min_{\mathbf{x}_i \in D} \mathrm{dist}(\mathbf{x}_i, \mathbf{x})$. This means, however, that we have to solve a nearest neighbor query when evaluating $f(\mathbf{x})$.

The proposed method can be of practical interest only if computing $f(\mathbf{x})$ is considerably faster than evaluating the function g a number of times. Therefore, we must restrict ourselves to functions that can be computed very efficiently. The family of functions we consider in this paper is discussed next.

3 Tree based partitioning of the input space

In this paper we consider functions f that can be represented as binary trees. These trees partition the feature space to disjoint subsets, and provide an efficient means to locate an unseen example $\mathbf{x}$ in the subset it belongs to. Each subset of the feature space uses a different set of support vectors. The concept is somewhat similar to the decision tree classifier, but it's implementation and use are quite different.

3.1 Basic definitions

Let $\mathcal{T}$ be a binary tree, and denote by N a node of $\mathcal{T}$. With every node N is associated a pair $(a, \mathbf{s}) \in S$. The *node score* of N given by $a_N g(\mathbf{x}, \mathbf{s}_N)$, where $a_N \in \mathbb{R}$ and $\mathbf{s}_N \in \Omega$ are the values associated with N, and g is e.g. a kernel function. The set $f(\mathbf{x})$ is found by following a path from the root of $\mathcal{T}$ to a leaf node. Based on the value of the node score at a node N we enter either the left or right subtree of N until a leaf node is reached. When this happens, we sum the node scores on the path from the root to the leaf, and use this as an approximation of the sum in Equation 1. There are some aspects to this that should be emphasized:

1. Unlike with decision and regression trees, branching is not based on the value of a feature, but on the node score $a_N g(\mathbf{x}, \mathbf{s}_N)$. This means the partition of the feature space induced by the tree is *not* in general the disjoint union of axis-aligned (hyper)rectangles.
2. The value computed by the tree is the sum of the node scores on the path from the root to a leaf node. This is in contrast to regression trees where the output is simply a value stored at each leaf. A consequence of this is that two examples, $\mathbf{x}_1, \mathbf{x}_2 \in \Omega$, that both follow the same path and hence end up at the same leaf, may still produce considerably different output values.

We continue with the definition of the support vector tree $\mathcal{T}$.

Definition 1. *A support vector tree $\mathcal{T}$ is the tuple $(\mathcal{N}, R, l, r)$, where $\mathcal{N}$ is a set of nodes, $R \in \mathcal{N}$ is the root node of the tree, and l and r are functions mapping the set $\mathcal{N}$ onto $\{\mathcal{N} \cup \emptyset\}$. Given a node $N \in \mathcal{N}$, $l(N)$ and $r(N)$ are the root nodes of the left and right subtrees of N, respectively. To each node $N \in \mathcal{N}$ is associated three values: $t_N \in \mathbb{R}$, $a_N \in \mathbb{R}$, and $\mathbf{s}_N \in \Omega$.*

Using $\mathcal{T}$ we find the set $f(\mathbf{x})$ by collecting all $(\mathbf{s}, a)$ pairs that are associated with nodes on a path from the root of $\mathcal{T}$ to a leaf node. At every node N the path goes either in the left or right subtree depending on the value $a_N g(\mathbf{x}, \mathbf{s}_N)$. If this value is less or equal to the node-specific threshold t_N the path continues to the left subtree, otherwise it continues to the right subtree. Note that instead of computing the set $f(\mathbf{x})$, it is more convenient to evaluate the sum in Equation 2 directly over the tree $\mathcal{T}$. We define the following:

Definition 2. *Let $g : \Omega \times \Omega \to \mathbb{R}$ be a function, denote by $\mathcal{T} = (\mathcal{N}, R, l, r)$ a support vector tree as defined above, and let $\mathbf{x} \in \Omega$. Denote by $\text{score}_N(\mathbf{x})$ the node score of $\mathbf{x}$ at node $N \in \mathcal{N}$. We let $\text{score}_N(\mathbf{x}) = a_N g(\mathbf{x}, \mathbf{s}_N)$. Denote by $\text{value}_N(\mathbf{x})$ the value of $\mathbf{x}$ in the subtree of $\mathcal{T}$ rooted at node $N \in \mathcal{N}$. We let*

$$\text{value}_N(\mathbf{x}) = \begin{cases} 0 & \text{if } N = \emptyset, \\ \text{score}_N(\mathbf{x}) + \text{value}_{l(N)}(\mathbf{x}) & \text{if } \text{score}_N(\mathbf{x}) \leq t_N, \\ \text{score}_N(\mathbf{x}) + \text{value}_{r(N)}(\mathbf{x}) & \text{if } \text{score}_N(\mathbf{x}) > t_N. \end{cases}$$

Finally, denote by $\mathcal{T}(\mathbf{x})$ the value of $\mathbf{x}$ in the entire tree $\mathcal{T}$. We let $\mathcal{T}(\mathbf{x}) = \text{value}_R(\mathbf{x})$, where R is the root node of $\mathcal{T}$.

Definition 1 does not restrict the size of $\mathcal{T}$ in any way. To reduce the number of evaluations of the function g, we must constrain the height of the tree. Denote by $\Phi_{\mathcal{T}}^h$ the set of trees where the length of the longest path from the root to a leaf is at most h. The general problem we discuss in the remaining of this paper is the following.

Problem 2. Given the training data $D \subset \Omega$, the set $S \subset \Omega \times \mathbb{R}$, and the function g, find the tree $\mathcal{T} \in \Phi_{\mathcal{T}}^h$ s.t.,

$$\sum_{\mathbf{x} \in D} \big(\mathcal{T}(\mathbf{x}) - \sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x})\big)^2 \tag{3}$$

is minimized.

Note that if $D = \{\mathbf{x}\}$, i.e., D contains only one example $\mathbf{x}$, the solution to Problem 2 is a path. Clearly no branching is needed for a single input instance. As discussed above, this problem is related to SUBSET-SUM, and hence it is unlikely that efficient solutions exist for Problem 2. In this paper we consider a greedy heuristic that leads to well performing trees in practice.

3.2 A simple exact algorithm for balanced trees

Before presenting the main algorithm of this paper, we briefly describe and analyze the trivial algorithm for solving Problem 2 exactly in the special case where the set $\Phi_{\mathcal{T}}^h$ is further restricted to contain only balanced trees, meaning that the number of training examples belonging to the left subtree of a node is the same as the number belonging to the right subtree. For the remaining discussion it is convenient to consider the following matrix:

Definition 3. *Given the data $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the set $S = \{(\mathbf{s}_1, a_1), \ldots, (\mathbf{s}_m, a_m)\}$, and the function g, denote by $\mathbf{X}^{D,S}$ the $n \times m$ matrix with*

$$\mathbf{X}_{ij}^{D,S} = a_j g(\mathbf{x}_i, \mathbf{s}_j).$$

The ith row of $\mathbf{X}^{D,S}$ is denoted by $\mathbf{X}_{i\cdot}^{D,S}$, and the jth column of $\mathbf{X}^{D,S}$ is denoted by $\mathbf{X}_{\cdot j}^{D,S}$. Moreover, given the sets I and J of integers, denote by $\mathbf{X}_{I,J}^{D,S}$ the sub-matrix of $\mathbf{X}^{D,S}$ containing the rows specified by I and columns specified by J.

Let $\mathbf{r}^{D,S}$ be the vector of row sums of $\mathbf{X}^{D,S}$, and denote by $\mathbf{r}_i^{D,S}$ the ith component of $\mathbf{r}^{D,S}$. That is, we have $\mathbf{r}_i^{D,S} = \sum_j \mathbf{X}_{ij}^{D,S}$. In the following we write simply $\mathbf{X}$ and $\mathbf{r}$ if D and S are clear from the context or otherwise irrelevant. Clearly for the ith item in D the second sum in Equation 3 is precisely $\mathbf{r}_i$.

Expressed in terms of the matrix $\mathbf{X}$, the learning task of Problem 2 is to approximate the vector $\mathbf{r}$ with appropriate subsets of the columns. It is useful to think that to every node N of $\mathcal{T}$ is associated the matrix $\mathbf{X}_{I,\cdot}$, where I is a subset of the row indices of $\mathbf{X}$. To learn $\mathcal{T}$ we must find an optimal split for the rows of $\mathbf{X}_{I,\cdot}$ at each node N. Since the jth column of $\mathbf{X}^{D,S}$ corresponds to the pair $(a_j, \mathbf{s}_j) \in S$, we can parameterize this optimization problem with two parameters per node N: the threshold t_N and column j_N. These define a split of $\mathbf{X}_{I,\cdot}$ at node N.

Define the sets $L_I(t_N, j_N)$ and $R_I(t_N, j_N)$ so that $L_I(t_N, j_N) = \{i \in I \mid \mathbf{X}_{ij_N} \leq t_N\}$ and $R_I(t_N, j_N) = \{i \in I \mid \mathbf{X}_{ij_N} > t_N\}$. Moreover, let $P(N)$ denote the set of nodes on the path from the root of a tree to the parent of node N. Using this, we let

$$\sigma(N)_i = \sum_{N' \in P(N)} X_{ij_{N'}}.$$

That is, $\sigma(N)_i$ is the value we use to approximate the ith row sum at the parent node of N. The cost of an optimal tree rooted at node N that is associated with the matrix $\mathbf{X}_{I,\cdot}$ is given by the following equation:

$$\mathrm{c}(N, I) = \begin{cases} \min_{t,j} \Big\{ \mathrm{c}\big(l(N), L_I(t,j)\big) + \mathrm{c}\big(r(N), R_I(t,j)\big) \Big\} & \text{if } \mathbf{X}_{I\cdot} \text{ should be spilt,} \\ \min_j \Big\{ \sum_{i \in I} \big(\sigma(N)_i + \mathbf{X}_{ij} - \mathbf{r}_i\big)^2 \Big\} & \text{otherwise.} \end{cases} \tag{4}$$

The cost of the tree $\mathcal{T}$ is $c(R, \{1, \ldots, n\})$, where R is the root of $\mathcal{T}$ and n the number of rows in $\mathbf{X}$. A node N should not be split if $|P(N)| + 1 = h$, but also

in the case where the cost of splitting is larger than not splitting. The latter condition implies, that even after we have split the node N we should check if a solution where N is unsplit has a lower cost.

The optimal tree for a given matrix $\mathbf{X}$ is thus found by considering all possible splits (defined by t and j), and finding the optimal trees for the sub-matrices $\mathbf{X}_{L_I(t,j)}$ and $\mathbf{X}_{R_I(t,j)}$. If we require that the tree is balanced at node N, we only have to optimize over j, because t is implicitly given by the requirement that $|L_I(t,j)| = |R_I(t,j)|$. A rough outline for an exact algorithm for solving this restricted variant of the problem is shown in Algorithm 1.

Algorithm 1 exact-balanced-tree

Input: set of integers I

if I should not be split **then**
  return "cost of I"
end if
for $j = 1$ to number of columns in $\mathbf{X}$ **do**
  $t \leftarrow$ median of column $\mathbf{X}_{I,j}$
  $c_j \leftarrow$ exact-balanced-tree($L_I(t,j)$) + exact-balanced-tree($R_I(t,j)$)
end for
return $\min\{c\}$

Assuming that $|I| = n$, and that $\mathbf{X}$ has m columns, we can express the running time of Algorithm 1 with the recurrence

$$T(n) \leq 2mT(\frac{n}{2}) + cmn. \tag{5}$$

This holds since we make $2m$ recursive calls to exact-balanced-tree with inputs of size $n/2$, and we must find the median (an $O(n)$ operation) m times. Using the Master method [9] it is easy to show that the running time of Algorithm 1 (in terms of n) is of order $O(n^{\log n})$. This makes exact-tree a quasi-polynomial time algorithm. It is slower than polynomial, but not exponential. Also note that the exact solution with unbalanced trees is even harder, because we also have to optimize over t. Of course this simple analysis does not rule out the existence of efficient solutions for Equation 4, but it suggests that they are not trivial to devise. Therefore, our aim in this paper is not to find trees that are optimal in terms of Equation 4, but instead we propose a heuristic for finding good trees using a greedy algorithm.

4 An inexact greedy algorithm

In this section we present a greedy algorithm for learning a tree $\mathcal{T}$ given the matrix $\mathbf{X}^{D,S}$. The algorithm contains parts of which it is not straightforward to analyze the running time, but we will argue in Section 4.2 that if the splits made by the algorithm are not very imbalanced (that is, the sizes of $L_I(t,j)$ and $R_I(t,j)$ are not very different), it's running time is of order $O(n^3)$.

Algorithm 2 build-sv-tree

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column index j
Output: the triple $(N, \mathcal{T}_l, \mathcal{T}_r)$

if stopping-condition-met($\mathbf{X}$, $\mathbf{r}$) **then**
  return $((j, -1), \emptyset, \emptyset)$
end if
$(t, j_l, j_r) \leftarrow$ find-optimal-split($\mathbf{X}$, $\mathbf{r}$, j)
$L \leftarrow \{i \ : \ \mathbf{X}_{ij} \leq t\}$
$R \leftarrow \{i \ : \ \mathbf{X}_{ij} > t\}$
$\mathcal{T}_l \leftarrow$ build-sv-tree($\mathbf{X}_{L\cdot}$, $(\mathbf{r}_L - \mathbf{X}_{Lj_l})$, j_l)
$\mathcal{T}_r \leftarrow$ build-sv-tree($\mathbf{X}_{R\cdot}$, $(\mathbf{r}_R - \mathbf{X}_{Lj_r})$, j_r)
return $((j, t), \mathcal{T}_l, \mathcal{T}_r)$

4.1 Algorithm description

On a high level the greedy algorithm is similar to the exact one discussed above. Each call to the algorithm finds a split of $\mathbf{X}$ that is in some sense optimal, and then recursively processes the two resulting sub-matrices. However, now we consider a somewhat different notion of optimality of a split induced by t and j. With the exact algorithm an optimal split is defined in terms of the optimal costs of the resulting sub-matrices, as can be seen in Equation 4. In particular, Alg. 1 computes the optimal subtrees rooted at a node N to evaluate the cost of splitting $\mathbf{X}$ along the jth column. The greedy algorithm takes a "myopic" approach. It only considers subtrees of size 1, meaning that they consist of *only one leaf node* each. In other words, we compute the cost of a split at node N under the restriction that $l(N)$ and $r(N)$ will not be split further, even if these actually are split later.

For any vector $\mathbf{x}$, we have $||\mathbf{x}|| = \mathbf{x}^{\mathrm{T}}\mathbf{x}$. Let $L_I(t, j)$ and $R_I(t, j)$ be defined as above, and suppose for a moment that we are given the column j. We define the cost of a "greedy split" as

$$\mathrm{c}(t, j_l, j_r) = ||\mathbf{X}_{L_I(t,j),j_l} - \mathbf{r}_{L_I(t,j)}|| + ||\mathbf{X}_{R_I(t,j),j_r} - \mathbf{r}_{R_I(t,j)}||, \tag{6}$$

where j_l and j_r are the columns that are associated with the left and right leafs, respectively. This is almost the same as Equation 5 if we assume that the child nodes of the current node are not split further. Another difference is that we are only optimizing over t, this time the parameter j was assumed to be given a priori. This may seem strange at first, but it is in fact quite natural: When splitting a node N using the cost in Equation 6, we must find the parameters j_l and j_r. These are used as the splitting-column when processing $l(N)$ and $r(N)$. The parameter j_N is thus already found when splitting the parent node of N. Pseudo-code of the greedy heuristic incorporating this principle is shown in Algorithm 2.

To find the optimal split, we must thus find the threshold t, and the column indices j_l and j_r. In theory there are $O(nm^2)$ different combinations for an input matrix $\mathbf{X}$ of size $n \times m$. Iterating over all of these is obviously not scalable.

Algorithm 3 find-optimal-split

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column index j
Output: the triple (t, j_l, j_r)

1: $t \leftarrow$ random element of $\mathbf{X}_{\cdot j}$
2: $c \leftarrow \infty$
3: **while** cost c is decreasing **do**
4: $\quad (j_l, j_r, c) \leftarrow$ optimize-columns($\mathbf{X}$, $\mathbf{r}$, j, t)
5: $\quad (t, c) \leftarrow$ optimize-threshold($\mathbf{X}$, $\mathbf{r}$, j, j_l, j_r)
6: **end while**
7: **return** (t, j_l, j_r)

Algorithm 4 optimize-columns

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column index j, threshold t
Output: triple (j_l, j_r, c)

1: $L \leftarrow \{i : \mathbf{X}_{ij} \leq t\}$
2: $R \leftarrow \{i : \mathbf{X}_{ij} > t\}$
3: $j_l \leftarrow \text{argmin}_{j'} ||\mathbf{X}_{Lj'} - \mathbf{r}_L||_2$
4: $j_r \leftarrow \text{argmin}_{j'} ||\mathbf{X}_{Rj'} - \mathbf{r}_R||_2$
5: $c \leftarrow ||\mathbf{X}_{Lj_l} - \mathbf{r}_L|| + ||\mathbf{X}_{Rj_r} - \mathbf{r}_R||$
6: **return** (j_l, j_r, c)

However, we can resort to a local optimization technique, that resembles the EM-algorithm and is of considerably lower, albeit unknown, complexity. Note that finding j_l and j_r is easy if we are given the value of t. In that case we simply have to try out the $O(m)$ different alternatives. Likewise, given j_l and j_r it is easy to find an optimal value for t by checking all n possible choices. We can thus alternatively solve for j_l and j_r given t, then solve for t given j_l and j_r, and continue this until convergence. The method is outlined in Algorithm 3. Algorithms 4 and 5 show the pseudo-code for the two subroutines in find-optimal-split.

4.2 Analysis of the greedy algorithm

To analyze the complexity of build-sv-tree, we resort again to the Master theorem [9]. This time the recurrence is

$$T(n) \leq 2T(\frac{n}{b}) + \underbrace{ch(n,m)(m+1)n}_{q(n)},$$

where $h(n, m)$ is a function that bounds the number of iterations of the loop on lines 3–5 in Algorithm 3. The optimize-columns algorithm (Alg. 4) runs in time $O(nm)$, while optimize-threshold (Alg. 5) can be implemented to run in time $O(n)$. Since we are not enforcing the split to be balanced, we use the parameter b in the analysis. Also, we assume that $m = n$, which corresponds to the case where all training examples end up as support vectors and the same data is

Algorithm 5 optimize-threshold

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column indices j, j_l, j_r
Output: pair (t^*, c^*)

$t^* \leftarrow -1$, $c^* \leftarrow \infty$
for $h = 1$ to number of rows in $\mathbf{X}$ **do**
  $t \leftarrow \mathbf{X}_{hj}$
  $L \leftarrow \{i : \mathbf{X}_{ij} \leq t\}$
  $R \leftarrow \{i : \mathbf{X}_{ij} > t\}$
  $c \leftarrow ||\mathbf{X}_{Lj_l} - \mathbf{r}_L|| + ||\mathbf{X}_{Rj_r} - \mathbf{r}_R||$
  if $c < c^*$ **then**
    $c^* \leftarrow c$, $t^* \leftarrow t$
  end if
end for
return (t^*, c^*)

used to find $\mathcal{T}$. This way $q(n)$ is of order $n^{2+\gamma}$, where γ is dependant on the complexity of the function h. That is, if h was of order $O(n)$ we would have $\gamma = 1$, for example.

To use the Master theorem we must compare $q(n)$ with the function $n^{\log_b 2+\epsilon}$. There are three cases based on the value of ϵ that result in different running times for the algorithm. More precisely, we study for what values of b, γ, and ϵ it holds that $n^{2+\gamma} = n^{\log_b 2+\epsilon}$. Solving this for ϵ gives

$$\epsilon = \frac{(2+\gamma)\log_2 b - 1}{\log_2 b}. \tag{7}$$

Now we consider three possible cases for ϵ, that is, $\epsilon < 0$, $\epsilon = 0$, and $\epsilon > 0$. By setting Equation 7 equal to zero and simplifying we obtain $\log_2 b = (2+\gamma)^{-1}$, or $b = 2^{\frac{1}{(2+\gamma)}}$. This gives us a relationship between b and γ that we can use to distinguish the different cases of the Master theorem:

- Case 1: $\epsilon < 0 \Leftrightarrow b < 2^{\frac{1}{(2+\gamma)}}$, which implies $T(n) = \Theta(n^{\log_b 2})$.
- Case 2: $\epsilon = 0 \Leftrightarrow b = 2^{\frac{1}{(2+\gamma)}}$, which implies $T(n) = \Theta(n^{\log_b 2} \log_2 n)$.
- Case 3: $\epsilon > 0 \Leftrightarrow b > 2^{\frac{1}{(2+\gamma)}}$, which implies $T(n) = \Theta(q(n))$ if the regularity condition holds as well.

There is thus a threshold for b, the value of which will depend on γ. Of course the actual value of b will vary, as different inputs lead to different splits. Some of these will be more balanced (b close to 2) than others (b close to 1). The value of γ depends on the convergence of the optimization in Algorithm 3. We do not know the exact rate of convergence in terms of n and m. However, based on empirical observations it is realistic to assume that for most inputs we have $0 < \gamma \leq 1$. I.e., $h(n, m)$ is at most linear in n, but possibly sublinear.

Letting $\gamma = 1$ gives us the threshold $2^{1/3} \approx 1.26$. This corresponds to an unbalanced split where roughly 80 percent of the input end up in the same subtree. Case 1 of the Master theorem concerns the case where the split is

even more unbalanced, while Case 3 covers more balanced splits. To analyze the deviation of b from the threshold $2^{1/3}$, we introduce a parameter β and let $b = 2^{1/3+\beta}$, so that $\beta < 0$, $\beta = 0$, and $\beta > 0$ correspond to the cases 1, 2, and 3, respectively. Note that for the degree of the polynomial in cases 1 and 2 we obtain $\log_b 2 = (\frac{1}{3} + \beta)^{-1}$.

In Case 2 ($\beta = 0$) the running time of Alg. 2 is therefore simply $\Theta(n^3 \log_2 n)$. For Case 1 ($\beta < 0$) we obtain a running time of $\Theta(n^{\frac{1}{(1/3+\beta)}})$, which is already $\Theta(n^6)$ if we let $\beta = -\frac{1}{6}$. This makes sense as very unbalanced splits are bound to slow down the algorithm considerably. And even with balanced splits represented by Case 3 ($\beta > 0$), the running time is bounded by $\Theta(q(n))$[1], where $q(n)$ is a polynomial of degree 3. That is, given that we assumed $\gamma = 1$, this type of analysis results in a cubic running time of Algorithm 2 even in the "best" case. However, it is still a considerable improvement over the quasi-polynomial exact algorithm discussed in Section 3.2.

4.3 Scaling the columns of $\mathbf{X}^{D,S}$

We can make a small modification to the algorithm that should improve it's performance. Instead of approximating $\mathbf{r}^{D,S}$ directly with the contents of $\mathbf{X}^{D,S}$, we can scale the values on the columns to minimize the difference to $\mathbf{r}^{D,S}$. More precisely, consider the lines 3 and 4 in Algorithm 4. The optimal columns j_l and j_r are found by minimizing the error $||\mathbf{X}_{\cdot j} - \mathbf{r}||$ subject to j. We add an additional parameter c, so that the new optimization problem becomes $\min_{j,c} ||c\mathbf{X}_{\cdot j} - \mathbf{r}||$.

Moreover, for a given j it is easy to find the optimal c. To see this, recall that we have $||c\mathbf{X}_{\cdot j} - \mathbf{r}|| = (c\mathbf{X}_{\cdot j} - \mathbf{r})^{\mathrm{T}}(c\mathbf{X}_{\cdot j} - \mathbf{r})$. Setting the 1st derivative of this with respect to c to zero gives

$$c = \frac{\mathbf{X}_{\cdot j}^{\mathrm{T}}\mathbf{r}}{\mathbf{X}_{\cdot j}^{\mathrm{T}}\mathbf{X}_{\cdot j}}.$$

This obviously also introduces a new parameter, c_N, to each node N of $\mathcal{T}$, and the score of node N for an unseen example $\mathbf{x} \in \Omega$ is now computed as $\mathrm{score}_N(\mathbf{x}) = c_N a_N g(\mathbf{x}, \mathbf{s}_N)$. The algorithm we use in the experiments makes use of this additional heuristic.

5 Empirical evaluation

We continue with empirical results. Recall that our main motivation is to speed up classification. Hence we are mostly interested in cases where the support vector tree outperforms either a linear SVM or a sparse SVM.

More precisely, we use five methods in our experiments: a linear SVM, a non-linear SVM using the RBF kernel, a support vector tree based on the nonlinear SVM, and two variants of the Cutting Plane Subspace Pursuit (cpsp) algorithm

[1] The regularity condition required by the theorem is also trivially satisfied for $b > 2^{1/3}$.

Table 1. Accuracies and number of support vectors of the various methods on different benchmark data sets. The best performing algorithm in the set {l-svm, sv-tree, cpsp(tr)} is indicated in bold.

dataset	dummy	l-svm	svm	sv-tree	cpsp	cpsp(tr)
heart	0.56	0.82	0.82	0.75	0.84	**0.84**
			60	8.2	9	9
thyroid	0.71	0.89	0.96	**0.92**	0.92	0.90
			13	9.3	10	10
breastcancer	0.70	0.70	0.71	0.70	0.73	**0.73**
			120	9.1	10	10
waveform	0.67	0.87	0.90	0.86	0.90	**0.88**
			229	9.5	10	10
german	0.70	**0.76**	0.75	0.70	0.76	**0.76**
			373	10.9	11	11
image	0.57	0.85	0.97	**0.89**	0.89	0.87
			213	11.9	12	12
diabetis	0.65	**0.77**	0.77	0.73	0.77	**0.77**
			249	10.0	11	11
ringnorm	0.50	0.75	0.98	**0.95**	0.84	0.74
			152	10.5	11	11
splice	0.51	**0.83**	0.89	0.68	0.86	0.77
			588	11.4	12	12
twonorm	0.50	**0.97**	0.98	0.91	0.98	**0.97**
			263	9.5	10	10
banana	0.54	0.55	0.89	**0.88**	0.86	**0.88**
			151	9.2	10	10

[16]. The first cpsp variant may use arbitrary basis vectors in the set S, while the second one is restricted to choose these from the training data in the style of a classical SVM. As in [16], we denote the 2nd variant by cpsp(tr). To learn the linear and nonlinear SVM we use LibSVM [7], while the cpsp algorithm is implemented in the svm-perf package [14–16]. To compare the methods we use standard benchmark data sets that are publicly available[2]. All cases are binary classification problems.

We consider the cpsp(tr) a more interesting comparison as cpsp, since our trees are also restricted to use only examples from the training data. It is obvious from the experiments in [16] and those shown here that the use of arbitrary basis vectors is in many cases beneficial. However, efficient implementations of this approach seem to be rather nontrivial to implement for arbitrary kernel functions as cpsp requires solving the pre-image problem [17]. While it is true that pre-images can be found even for structured objects such as strings [1] and graphs [2], the sv-tree can be seen as a more powerful approach as it is oblivious to the representation of the input examples given any suitable kernel function. For instance, svm-perf can only use RBF kernels with the cpsp algorithm. Hence

[2] `http://ida.first.fraunhofer.de/projects/bench/`

it is more interesting to see if the sv-tree can beat the linear SVM, and achieve at least the performance of the cpsp(tr) algorithm.

Instead of explicitly giving a maximum height for the support vector tree, we use a stopping criteria that is based on the size of the input. The call to stopping-condition-met on line 1 in Algorithm 2 returns true if the number of rows in $\mathbf{X}$ is less or equal to five. We could just as well use a stopping condition based on the height of the tree. However, considering the size of a node has the advantage that we do not split matrices with only a couple of rows, and conversely allow a node to split when there is still enough data to work with. With both SVMs and cpsp we use a grid-search to find good values for the regularization parameter C and the parameter γ used by the RBF kernel. Furthermore, to have an interesting comparison with the cpsp method, we set it's budget equal to the average number of kernel evaluations needed by the tree to classify a given test set.

Results for the accuracies and sizes of the model are given in Table 1. The reported numbers are averages over 20 disjoint training-test splits. The 2nd column shows the accuracy of a dummy classifier that simply assigns everything in the test data to the class that was more common in the training data. Of the studied algorithms the linear SVM is the simplest, and is the preferred choice if classification speed is an issue. Indeed, in a few cases the accuracy of l-svm is comparable with that of svm. When compared with the nonlinear SVM, the sv-tree has a lower accuracy with all data sets, which is to be expected. With the 'heart' and 'splice' data sets the sv-tree seems to especially have problems. But the sv-tree in fact outperforms l-svm and cpsp(tr) a number of times. This is the case with the 'thyroid', 'image', 'ringnorm', and 'banana' data sets. Also note that in many cases where cpsp(tr) outperforms sv-tree, the linear SVM outperforms cpsp(tr), and would hence be the method of choice to speed up classification. With the chosen stopping condition for the sv-tree, the number of kernel evaluations is with these data sets reduced by one order of magnitude.

6 Conclusion

We have presented an approach to speed up classification with kernel machines based on adaptive selection of basis vectors given the example to be classified. Despite the large body of existing literature on kernel machines this idea seems to be, to the best of our knowledge, novel. To quickly find the subset of basis vectors to be used in the decision rule, we propose the use of a binary tree, the support vector tree, that induces a disjoint partition of the feature space. To learn this tree we propose a greedy heuristic that can result in suboptimal trees but runs in polynomial time. Our experiments suggest that the support vector tree can in some cases outperform existing state-of-the-art algorithms for learning sparse SVMs.

It must be noted that the idea proposed here is not specific to kernel machines. The same approach can be employed also in case of ensemble classifiers. Even though the weak learners (or base classifiers) in general are fast to compute, there can be several hundreds of them. This can become a bottleneck for

e.g. ensemble method based document ranking functions. For instance in [6] the evaluation of the ensemble is interrupted when it becomes unlikely that the outputs of the remaining weak learners would significantly change the already computed score of the document being ranked. Our method could be applied as such in this setting by replacing kernel computations with evaluations of the weak lerners.

Obvious future research concerns improved algorithms for finding trees with better accuracy. Alternatively we can consider other representations of the function f. One problem with the current approach is that we must first learn a kernel machine using some legacy algorithm. Instead of a post-processing algorithm, we can also devise algorithms that find a tree directly based on training data. Finally, a more detailed experimental evaluation of the current algorithm using large real world data sets is also of interest.

References

1. G. Bakir, J. Weston, and B. Schölkopf. Learning to find pre-images. In *Advances in Neural Information Processing Systems 16 (NIPS 2003)*, 2003.
2. G. Bakir, A. Zien, and K. Tsuda. Learning to find graph pre-images. In *Pattern Recognition, 26th DAGM Symposium*, pages 253–261, 2004.
3. C. Burges. Simplified support vector decision rules. In *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96)*, pages 71–77, 1996.
4. C. Burges. A tutorial on support vector machines for pattern recognition. *Knowledge Discovery and Data Minnig*, 2(2):121–167, 1998.
5. C. Burges and B. Schölkopf. Improving the accuracy and speed of support vector machines. In *Advances in Neural Information Processing Systems 9, NIPS*, pages 375–381, 1996.
6. B. Cambazoglu, H. Zaragoza, and O. Chapelle. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM 2010*, 2010. (to appear).
7. C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/ cjlin/libsvm`.
8. J.-H. Chen and C.-S. Chen. Reducing svm classification time using multiple mirror classifiers. *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, 34(2):1173–1183, 2004.
9. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT PRess, 2nd edition, 2001.
10. O. Dekel and Y. Singer. Support vector machines on a budget. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems*, pages 345–352, 2006.
11. T. Downs, K. E. Gates, and A. Masters. Exact simplification of support vector solutions. *Journal of Machine Learning Research*, 2:293–297, 2001.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. Freeman, 1979.
13. T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142, 2002.

14. T. Joachims. A support vector method for multivariate performance measures. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 377–384, 2005.
15. T. Joachims. Training linear svms in linear time. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 217–226, 2006.
16. T. Joachims and C.-N. J. Yu. Sparse kernel svms via cutting-plane training. *Machine Learning*, 76:179–193, 2009.
17. J. T. Kwok and I. W. Tsang. The pre-image problem in kernel methods. *IEEE Transactions on Neural Networks*, 15(6):1517–1525, 2004.
18. H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444, 2002.
19. P. B. Nair, A. Choudhury, and A. J. Keane. Some greedy learning algorithms for sparse regression and classification with mercer kernels. *Journal of Machine Learning Research*, 3:781–801, 2002.
20. E. Osuna and F. Girosi. *Advances in Kernel Methods: Support Sector Learning*, chapter Reducing the run-time complexity in support vector machines. MIT Press, 1999.
21. C. H. Teo and S. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *Proceedings of 23rd International Conference on Machine Learning*, pages 929–936, 2006.
22. M. E. Tipping. Sparse bayesian learning and the relevance vector machine. *Journal of Machine Learning Research*, 1:211–244, 2001.
23. V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.
24. M. Wu, B. Schölkopf, and G. Bakir. A direct method for building sparse kernel learning algorithms. *Journal of Machine Learning Research*, 7:603–624, 2006.