Example-dependent Basis Vector Selection for Kernel-based Classifiers

Antti Ukkonen[1⋆] and Marta Arias[2]

[1] Aalto University and
Helsinki Institute for Information Technology
Helsinki, Finland
antti.ukkonen@hiit.fi

[2] Universitat Politècnica de Catalunya, Barcelona, Spain
marias@lsi.upc.edu

Abstract. We study methods for speeding up classification time of kernel-based classifiers. Existing solutions are based on explicitly seeking sparse classifiers during training, or by using budgeted versions of the classifier where one directly limits the number of basis vectors allowed. Here, we propose a more flexible alternative: instead of using the same basis vectors over the whole feature space, our solution uses different basis vectors in different parts of the feature space. At the core of our solution lies an optimization procedure that, given a set of basis vectors, finds a good partition of the feature space and good subsets of the existing basis vectors. Using this procedure repeatedly, we build trees whose internal nodes specify feature space partitions and whose leaves implement simple kernel classifiers. Experiments suggest that our method reduces classification time significantly while maintaining performance. In addition, we propose several heuristics that also perform well.

1 Introduction

Kernel-based classifiers such as support vector machines are among the most popular classification methods. When working with big datasets or costly kernels, however, these classifiers can be slow not only during train, but also when classifying new instances. Setting the kernel computation cost aside, the classification cost of a kernel classifier is mainly governed by the number of basis vectors that it contains. Existing solutions to speed up classification time are based on explicitly seeking classifiers with few basis vectors (*sparse* classifiers) during training. This is the approach taken for example in [17] in the context of support vector machines. Another approach is to limit the number of basis vectors explicitly by means of an a-priori imposed bound, a *budget*, as is the case of the budget perceptron [10] in online learning.

⋆ This work was carried out while the author was visiting Yahoo! Research Barcelona.

In this paper we propose an alternative solution that is also based on reducing the number of basis vectors. The main difference of our approach with respect to existing solutions is that instead of using the same sparse classifier in the whole feature space, we use different sparse classifiers in different parts of the feature space. That is, *the set of basis vectors that we use to classify a new example now depends on the example*, or more accurately, depends on the region where the example lies. This added flexibility represents a challenge for learning: one not only needs to partition the feature space, but also needs to select basis vectors in each partition.

The idea of using different classifiers in different regions of the feature space is not new, and is in fact used in popular methods for regression and other learning problems, e.g. piecewise regression or functional trees [16]. However, to the best of our knowledge, this is the first time this has been applied in the context of sparse kernel methods. We use the framework of online learning and the kernel perceptron algorithm [14] as the basis of our solution. We note however, that our method can be applied as a post-processing step to any kernel-based classifier.

1.1 Related work

Kernel methods and in particular *Support Vector Machines* (SVMs) [30] have been intensely studied during the past couple of decades. Finding *sparse* SVMs has been one of the main concerns from the start. Early works on this topic include [6, 5], in which the authors devise post-processing algorithms for finding a reduced set of basis vectors from a given solution. Most of the approaches that followed differ from [6, 5] (and our paper) by formulating a version of the SVM learning problem that attempts to directly find a sparse solution. Early examples of this include [27, 13, 20, 24]. More recently, [32] discuss a method to build sparse SVMs and report results where the accuracy of the full SVM is in some cases achieved using only a very small fraction (5%) of the original vectors. Unlike other related work, [8] propose an ensemble-based solution. The most recent contribution along this line or work is [17], where authors show that they can reduce the number of basis vectors by two orders of magnitude without suffering a decrease in classification accuracy.

Most of the approaches mentioned so far determine automatically the number of basis vectors used in the final sparse SVM (or use some regularization parameter that determines the trade-off between sparsity and accuracy of the solution). In contrast, the approach in [12] admits a parameter specifying the maximum number of vectors allowed in the final solution (the *budget*). The works in [17, 32] have this ability also.

A different line of work towards finding sparse kernel-based classifiers has been done in the context of the perceptron algorithm [26]. Perceptrons use the same type of classification functions as SVMs but are much faster to train. They can also be used in conjunction with kernels [14], which has gained them much popularity [14, 3, 18, 9]. Starting with the work of [10], a new line of algorithms known as the *budget perceptron* has emerged, of which several variants exist [10, 31, 11]. All of these impose an upper bound on the number of basis vectors

Algorithm 1 The perceptron [26] and kernel perceptron [14] algorithms.

PERCEPTRON

$\mathbf{w} = 0$
for $i = 1, .., n$ **do**
  $pred = \text{sign}(\mathbf{w}^T \mathbf{x}_i)$
  if $pred \neq y_i$ **then**
    $\mathbf{w} = \mathbf{w} + y_i \mathbf{x}_i$
  end if
end for

KERNELPERCEPTRON

$S = \{\}$
for $i = 1, .., n$ **do**
  $pred = \text{sign}(\sum_{(a,\mathbf{b}) \in S} aK(\mathbf{b}, \mathbf{x}_i))$
  if $pred \neq y_i$ **then**
    $S = S \cup \{(y_i, \mathbf{x}_i)\}$
  end if
end for

allowed in the final classifier, but differ in how they behave when this budget is exceeded. For example, on exceeding the budget [31, 10, 11] use different criteria to chose which "old" vector is going to be replaced. In contrast, the *projectron* [21] attempts to project the new example back to the space spanned by the existing basis vectors if possible, or adds it otherwise. The *projectron* does not have an apriori upper bound, but it can be shown that classifiers cannot grow unboundedly.

All related work described so far produces a single, global, sparse classifier. In this paper, we depart from this by allowing different solutions in different regions of the feature space. The only works we know of that use this idea are [28] and [2]. Especially [2] discusses a method that seems similar to the one presented here as it combines decision trees with support vector machines. However, the main motivation in [2] is not to reduce the number of kernel evaluations, and the resulting algorithm is a fairly complex combination of gradient descent and tabu search. In contrast to these, the work proposed here is done in the context of online learning with kernel perceptrons. In particular, our tree-based approach combines an online partitioning of the feature space with fast classification with budgeted kernel perceptrons. We call this new model the *perceptron tree*.

2 Perceptron trees

2.1 Background on perceptrons

The *kernel perceptron* [14, 19] is an online learning algorithm which has been shown to perform very well in practice. Its simplicity, well-understood behavior, and the fact that it has the ability to incorporate kernels, make it an excellent candidate for being the basis for our solution. Given a sequence of n examples described by feature vectors $\mathbf{x}_i$ together with their $y_i \in \pm 1$ class labels

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), .., (\mathbf{x}_n, y_n)$$

the standard perceptron algorithm [26] keeps a hypothesis $\mathbf{w}$ in the form of a weight vector of the same dimensionality as the examples representing a linear classifier, which is updated as indicated in Algorithm 1 (left column).

Algorithm 2 An online algorithm for learning perceptron trees.

```
BuildTree(x, y, N)
  if N is a leaf node then
    if P_N(x) ≠ y then
      add x as a basis vector of P_N.
    end if
    if |P_N| > B then
      (f_N, N_L, N_R) ← Split(N)
    end if
  else
    BuildTree(x, y, f_N(x))
  end if
```

Notice that $\mathbf{w}$ is a linear combination of examples $\mathbf{x}_i$ on which the perceptron makes a mistake [14, 18]. This suggests representing $\mathbf{w}$ in its dual form, namely, by maintaining the set S of coefficients y_i and vectors $\mathbf{x}_i$ on which the algorithm makes mistakes during training. The advantage of this representation is that it allows the use of a kernel function $K(\cdot, \cdot)$. In the pseudocode for kernel perceptron in Algorithm 1 (right column), S is the set of pairs $(a, \mathbf{b})$ such that a is the coefficient of the term $K(\mathbf{b}, \cdot)$ in the decision function, and $\mathbf{b}$ is a basis vector.

Clearly, the cost of classifying a new example requires the computation of a weighted sum of kernel products between the example to classify and the existing basis vectors. This cost is proportional to both the number of vectors in our hypothesis $|S|$ and the time it takes to compute the kernel.

2.2 Solution overview

Our solution uses different budget perceptrons in different regions of the feature space. Therefore, we need to devise a method that is able to learn mappings from the feature space to small sets of basis vectors. On arrival of a new example $\mathbf{x}$ to be classified, we use the mapping to get the basis vectors $S(\mathbf{x})$ operating in the region that $\mathbf{x}$ belongs to, and use the rule $\text{sign}(\sum_{(a,\mathbf{b})\in S(\mathbf{x})} aK(\mathbf{b}, \mathbf{x}))$ to make the final prediction. Since we are trying to reduce classification time, we should be able to compute the region where an example falls quickly.

We propose a top-down, tree-like approach that starts with a single node training a budget perceptron. When we need to add a new basis vector and the budget is exceeded, we *split* that perceptron node and start training two new perceptrons as new left and right leaves to this node. The new perceptron leaf nodes are initialized with suitable subsets of the existing basis vectors. The split node is assigned a *branching function* f that sends examples to its left or right child. The leaves continue to split as we add new examples. When the process is finished, we end up with a tree where the branching functions f_i at the internal nodes induce a partition of the feature space, and the budget perceptrons P_j in the leaves classify examples.

Pseudocode for the tree building algorithm is shown in Algorithm 2. To construct a tree given training data, we call BUILDTREE once for each example $\mathbf{x}$ with N being the root of the tree. When the size of the perceptron at a leaf node exceeds a predefined budget B, we split the node. This is done by the SPLIT function that returns the branching function f_N for node N, as well as the left and right child nodes, N_L and N_R, rooted at N. SPLIT also initializes the perceptrons at N_L and N_R using the basis vectors at node N.

Naturally, all kinds of questions related to growing trees arise, such as when do we stop growing the tree, do we use pruning, etc. Although important in practice, these issues are not the focus of this paper. Our focus is in how to do the split: namely, how to find the branching functions, and how to select good subsets of existing basis vectors for the children of a split node.

In the following sections, we present several ways of implementing the SPLIT function. The main method that we propose is based on solving several quadratic and linear programming problems, although we also present a few heuristics that perform well in our experiments.

3 Splitting nodes

Suppose that we are at a leaf node N that has made a mistake on a new example, and N has exhausted its budget. The perceptron at this leaf node needs to be split into two new perceptrons. To carry out the split we need to define two things:

1. *Partition problem*: what examples are assigned to the left and right subtrees, respectively?
2. *Selection problem*: what basis vectors of node N form the initial perceptron in the left and right subtrees, respectively?

Ideally we would like to solve both of these tasks in a way that minimizes the number of errors on unseen examples. It is not clear to us how to do this properly. Instead, we propose a number of different approaches. Of these the most important one is based on approximating the existing classifier at node N. This strategy is the main focus in this section, while the remaining ones are covered in Section 4.

Observe that the partition and selection problems are not independent. A good partition may depend on the selection of basis vectors, and selecting the basis vectors is affected by the partition. This suggests that in order to find good solutions the two problems must be solved together in a joint optimization step. Alternatively, we can decouple the problems by first using some criteria to define the partition, and subsequently selecting the basis vectors given this partition. We discuss the joint optimization approach in this section.

3.1 Basic definitions

For now our main objective when computing the split is to maintain the current classifier as well as possible. Let the set $S_N = \{(a_1, \mathbf{b}_1), \ldots, (a_m, \mathbf{b}_m)\}$ contain

all basis vectors $\mathbf{b}_j$ together with their class labels a_j that are used by node N. Moreover, let the set $X_N = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ contain all examples that node N has seen so far. Define the matrix $\mathbf{G}$, such that $\mathbf{G}_{ij} = a_j K(\mathbf{x}_i, \mathbf{b}_j)$, and let $\mathbf{r}_i = \sum_j \mathbf{G}_{ij}$. The perceptron at node N classifies $\mathbf{x}_i$ according to $\text{sign}(\mathbf{r}_i)$. In order to maintain the behavior of this classifier, *we seek to approximate* $\mathbf{r}_i$ *on both sides of the split by using different subsets of* S_N. To make this more formal, we define the following:

- *Partition:* Denote by $\mathbf{z}$ an n-dimensional binary vector that indicates how to split X_N. We let $\mathbf{z}_i = 1$ if $\mathbf{x}_i \in X_N$ is assigned to the left subtree, and $\mathbf{z}_i = 0$ if $\mathbf{x}_i$ is assigned to the right subtree. For simplicity, we only consider perfectly balanced partitions, that is, partitions for which $\sum_i \mathbf{z}_i = n/2$.
- *Left selection:* Denote by $\mathbf{p}$ an m-dimensional binary vector that indicates which basis vectors in S_N are assigned to the left child of N. We let $\mathbf{p}_j = 1$ if basis vector $\mathbf{b}_j$ is to be placed to the left, and $\mathbf{p}_j = 0$ indicates otherwise.
- *Right selection:* We define the binary vector $\mathbf{q}$ in the same way as $\mathbf{p}$ for selecting basis vectors of the right child of N.

Using these definitions a split is represented by the triple $(\mathbf{z}, \mathbf{p}, \mathbf{q})$. Here $\mathbf{z}$ defines the assignment of examples to the subtrees, while $\mathbf{p}$ and $\mathbf{q}$ select the basis vectors to be used in the left and right child nodes. But notice that while vector $\mathbf{z}$ tells us which examples in X_N go to the left and right subtrees, it tells us nothing about new examples. In practice, we must learn the *branching function* f_N that allows us to classify new, unseen examples. Let us put this problem aside for now, however, and see how to find good splits $(\mathbf{z}, \mathbf{p}, \mathbf{q})$. Later, in Section 3.3, we show that a simple modification of this procedure allows us to learn $\mathbf{p}$ and $\mathbf{q}$ together with f_N.

3.2 Joint optimization with unconstrained branching

In this section we consider the case where vectors in X_N can be placed arbitrarily to the left and right subtree. This means that the branching function is essentially unconstrained, and has no predefined form. We define the costs of vectors $\mathbf{p}$ and $\mathbf{q}$ as follows:

$$c_{\mathbf{p}}(\mathbf{x}_i) = \big(\sum_j \mathbf{p}_j \mathbf{G}_{ij} - \mathbf{r}_i\big)^2 \text{ and } c_{\mathbf{q}}(\mathbf{x}_i) = \big(\sum_j \mathbf{q}_j \mathbf{G}_{ij} - \mathbf{r}_i\big)^2, \tag{1}$$

where $\mathbf{G}$ and $\mathbf{r}$ are defined as above. We are thus seeking to minimize the squared difference of using all the basis vectors vs. using the selection only. (Notice that $\mathbf{p}$ and $\mathbf{q}$ need not be integer for Equation 1 to make sense.) For all $\mathbf{x}_i$ that are placed to the left we pay $c_{\mathbf{p}}(\mathbf{x}_i)$, and for the remaining ones we pay $c_{\mathbf{q}}(\mathbf{x}_i)$. The total cost of the split is thus defined as

$$c(\mathbf{z}, \mathbf{p}, \mathbf{q}) = \sum_i \mathbf{z}_i c_{\mathbf{p}}(\mathbf{x}_i) + (1 - \mathbf{z}_i) c_{\mathbf{q}}(\mathbf{x}_i), \tag{2}$$

which is equivalent (up to an additive constant, see Appendix A) to

$$\hat{c}(\mathbf{z}, \mathbf{p}, \mathbf{q}) = \mathbf{p}^T \mathbf{Q}_L(\mathbf{z})\mathbf{p} - 2\mathbf{w}_L^T(\mathbf{z})\mathbf{p} + \mathbf{q}^T \mathbf{Q}_R(\mathbf{z})\mathbf{q} - 2\mathbf{w}_R^T(\mathbf{z})\mathbf{q}. \tag{3}$$

Here $\mathbf{Q}_L(\mathbf{z})$ and $\mathbf{Q}_R(\mathbf{z})$ are $m \times m$ matrices, and $\mathbf{w}_L(\mathbf{z})$ and $\mathbf{w}_R(\mathbf{z})$ are m-dimensional vectors that *depend on* $\mathbf{z}$. If we fix the vector $\mathbf{z}$, then minimizing Equation 3 is a matter of solving two independent quadratic integer programs, of which the one for $\mathbf{p}$ is shown below:

$$\begin{aligned} \mathrm{QP}_{\mathbf{p}} : \min \; & \frac{1}{2}\mathbf{p}^T \mathbf{Q}_L(\mathbf{z})\mathbf{p} - \mathbf{w}_L^T(\mathbf{z})\mathbf{p} \\ st. \; & \sum_j \mathbf{p}_j \leq C, \\ & \mathbf{p}_j \in \{0, 1\} \text{ for all } j. \end{aligned}$$

The program $\mathrm{QP}_{\mathbf{q}}$ for finding $\mathbf{q}$ is defined analogously. Notice that the matrices $\mathbf{Q}_L(\mathbf{z})$ and $\mathbf{Q}_R(\mathbf{z})$ are positive semidefinite and therefore the objective function is convex.

The constraint $\sum_j \mathbf{p}_j \leq C$ is imposed so that the left subtree uses at most C basis vectors. This constant should be set by the user depending on the needs of the application at hand or by some other empirical method beyond the scope of our present discussion. If given a budget B, a reasonable value for C could be set for example to $B/2$, so that the new perceptrons at the leaves are initialized with half of their available budget. This is, in fact, the strategy that we adopt in the experiments.

Finding $\mathbf{p}$ and $\mathbf{q}$ is therefore easy if we know $\mathbf{z}$. We continue by showing that given $\mathbf{p}$ and $\mathbf{q}$ it is easy to find $\mathbf{z}$. Clearly Equation 2 can be rewritten as

$$c(\mathbf{z}, \mathbf{p}, \mathbf{q}) = \sum_i \mathbf{z}_i \underbrace{\left(c_{\mathbf{p}}(\mathbf{x}_i) - c_{\mathbf{q}}(\mathbf{x}_i)\right)}_{\mathbf{c}_i(\mathbf{p},\mathbf{q})} + \sum_i c_{\mathbf{q}}(\mathbf{x}_i).$$

From this we see that if we know the costs $c_{\mathbf{p}}(\mathbf{x}_i)$ and $c_{\mathbf{q}}(\mathbf{x}_i)$ for all i, i.e., the vectors $\mathbf{p}$ and $\mathbf{q}$ are fixed, the optimal $\mathbf{z}$ is found by solving the linear integer program

$$\begin{aligned} \mathrm{LP}_{\mathbf{z}} : \min \; & \mathbf{z}^T \mathbf{c}(\mathbf{p}, \mathbf{q}) \\ st. \; & \sum_i \mathbf{z}_i = n/2, \\ & \mathbf{z}_i \in \{0, 1\} \text{ for all } i. \end{aligned}$$

Note that solving this is simply a matter of setting those indices of $\mathbf{z}$ to 1 that correspond to the $n/2$ smallest values of $\mathbf{c}(\mathbf{p}, \mathbf{q})$.

Ideally we would like to solve (3) for $\mathbf{z}$, $\mathbf{p}$, and $\mathbf{q}$ simultaneously. This, however, does not seem easy. But as argued above, we can find $\mathbf{p}$ and $\mathbf{q}$ given $\mathbf{z}$, and vice versa. Our solution, shown in Algorithm 3, relies on this. It iteratively optimizes each of $\mathbf{z}$, $\mathbf{p}$, and $\mathbf{q}$ while keeping the two other vectors fixed.

Algorithm 3 Joint optimization with unconstrained branching.

1: Initialize $\mathbf{p}$ and $\mathbf{q}$ at random.
2: **while** objective function value decreases **do**
3: Find $\mathbf{z}$ given $\mathbf{q}$ and $\mathbf{p}$ by solving $\mathrm{LP}_{\mathbf{z}}$.
4: Find $\mathbf{q}$ given $\mathbf{z}$ by solving a fractional relaxation of $\mathrm{QP}_{\mathbf{q}}$.
5: Find $\mathbf{p}$ given $\mathbf{z}$ by solving a fractional relaxation of $\mathrm{QP}_{\mathbf{p}}$.
6: **end while**
7: Round $\mathbf{p}$ and $\mathbf{q}$ to integers.
8: **return** $(\mathbf{z}, \mathbf{p}, \mathbf{q})$

The situation is further complicated by the integer constraints present in the quadratic programs given above. Our strategy here is to use fractional solutions (variables bounded in $[0, 1]$) during the iteration and round these to integers before returning a solution. We experimented with a number of rounding schemes, and observed the following to perform well. For each i, set $\mathbf{p}_i = 1$ with probability equal to the value of $\mathbf{p}_i$ in the fractional solution. Repeat this a number of times and keep the best.

While Algorithm 3 clearly solves the selection part of the split, it is less obvious whether we can use the vector $\mathbf{z}$ to construct a good branching function f_N. A simple approach is to find the vector $\mathbf{x}_i^* \in X_N$ that is closest to an unseen example $\mathbf{x}$, and forward $\mathbf{x}$ to the left subtree if $\mathbf{z}_i = 1$, and to the right if $\mathbf{z}_i = 0$. We refer to this approach as ZPQ in the experiments of Section 5.

Notice that finding the nearest neighbor can be slow, and ideally we want the branching functions to be (orders of magnitude) faster than classifying $\mathbf{x}$. To this end, we discuss next an alternative formulation of the split that directly includes f_N as part of the optimization problem.

3.3 Joint optimization with linear branching

In this section we propose a split where the vectors in X_N may no longer be partitioned arbitrarily. In particular, the branching function f_N is constrained to be a linear separator in the feature space. The basis vector selection remains as before. Denote the linear separator by $\mathbf{w}$. We let $\mathbf{z}$ depend on $\mathbf{w}$ as follows:

$$\mathbf{z}_i = \frac{1}{2}\operatorname{sign}(\mathbf{w}^T\mathbf{x}_i) + \frac{1}{2},$$

where $\mathbf{x}_i \in X_N$ as above. If we further assume that $\|\mathbf{x}_i\| = 1$, $\|\mathbf{w}\| = 1$, and remove the sign from $\mathbf{w}^T\mathbf{x}_i$, we can let $\mathbf{z}_i = \left(\frac{1}{2}\mathbf{w}^T\mathbf{x}_i + \frac{1}{2}\right) \in [0, 1]$. Using this the objective function of $\mathrm{LP}_{\mathbf{z}}$ from the previous section can be re-written to depend on $\mathbf{w}$ instead of $\mathbf{z}$:

$$\mathbf{z}^T\mathbf{c} = \sum_i (\frac{1}{2}\mathbf{w}^T\mathbf{x}_i + \frac{1}{2})\mathbf{c}_i = \frac{1}{2}\mathbf{w}^T \underbrace{\sum_i \mathbf{x}_i\mathbf{c}_i}_{\mathbf{d}} + \underbrace{\frac{1}{2}\sum_i \mathbf{c}_i}_{D} = \frac{1}{2}\mathbf{w}^T\mathbf{d} + D.$$

Omitting constants, our new formulation becomes

$$\begin{aligned} \mathrm{LP}_{\mathbf{w}} : \min\ & \mathbf{w}^T\mathbf{d} \\ st.\ & \|\mathbf{w}\| = 1, \end{aligned}$$

where the $\mathbf{d}$ is a column vector such that $\mathbf{d}_j = \sum_i \mathbf{x}_{ij}\mathbf{c}_i$, $\mathbf{x}_{ij}$ is the value of example $\mathbf{x}_i$ along dimension j after normalizing $\mathbf{x}_i$, and $\mathbf{c}_i$ is the cost associated with example $\mathbf{x}_i$ which can be computed for fixed $\mathbf{p}, \mathbf{q}$.

It can be shown that $\mathrm{LP}_{\mathbf{w}}$ is minimized for $\mathbf{w} = -\frac{\mathbf{d}}{\|\mathbf{d}\|}$. That is, as with $\mathrm{LP}_{\mathbf{z}}$, we do not need to solve a linear program. Notice that we have ignored the constraint present in $\mathrm{LP}_{\mathbf{z}}$ that requires a balanced solution. In this case, this can be enforced by using a threshold $\theta = \mathrm{median}\{\mathbf{w}^T\mathbf{x}_i | \mathbf{x}_i \in X_N\}$. We acknowledge that we could include θ in our optimization, but this straightforward approach leads to good results in practice.

Again we use Algorithm 3, but modify it so that line 3 reads: "Find $\mathbf{w}$ given $\mathbf{q}$ and $\mathbf{p}$ by solving $\mathrm{LP}_{\mathbf{w}}$. Let $\theta = \mathrm{median}\{\mathbf{w}^T\mathbf{x}_i | \mathbf{x}_i \in X_N\}$, and compute the associated $\mathbf{z}$ by setting $\mathbf{z}_i = \mathrm{sign}(\mathbf{w}^T\mathbf{x}_i + \theta)$.". Naturally, the same considerations regarding the rounding procedure to obtain integer solutions apply here.

With this method it is obvious that we have found both a fast to compute branching function, $\mathrm{sign}(\mathbf{w}^T\mathbf{x}_i + \theta)$, and a good set of basis vectors for the child nodes. The algorithm presented here finds these simultaneously in the sense that the solution for $\mathbf{w}$ depends on $\mathbf{p}$ and $\mathbf{q}$, and vice versa. In the next section we discuss some other approaches where the partition and selection tasks are decoupled. We refer to this approach as WPQ in the experiments of Section 5.

4 Heuristics and baselines for splitting nodes

The algorithm discussed above may be too complex for some situations as it involves solving a number of quadratic programs. Moreover, it is difficult to say how fast the objective function value converges. In practice we have observed the convergence to be rapid, but there are no theoretical guarantees for this. As a remedy we present some heuristics that are in general faster to compute.

We describe simple heuristics and baseline methods for the partition and the selection problems, respectively. These can be combined in order to develop reasonable splitting methods. As we already observed earlier, solving the selection problem becomes easy if we fix the partition.

4.1 Partition heuristics

With partition heuristics we mean heuristics for constructing the branching function f_N. We have already seen two alternatives in the previous section. The first one made use of a nearest neighbor algorithm, while the second one was based on a linear separator of the feature space.

A simple baseline, which we call RNDW, consists in obtaining a random partition by drawing a hyperplane uniformly at random.

Algorithm 4

BALANCEDBRANCHINGFUNCTION(S_N, X_N, K)

```
for (a_j, b_j) ∈ S_N do
    t_j ← median{K(x, b_j) : x ∈ X_N}
    X_L ← {x ∈ X_N : K(x, b_j) ≤ t_j}
    X_R ← {x ∈ X_N : K(x, b_j) > t_j}
    s_j ← EVALUATE(X_L, X_R)
end for
j* ← argmax_j s_j
return (b_{j*}, t_{j*})
```

In addition, we propose a class of branching functions that can be represented as axis-aligned hyperplanes in the kernel space. That is, an example $\mathbf{x}$ is directed to the left subtree if $K(\mathbf{x}, \mathbf{b}_j) \leq t$ for some $\mathbf{b}_j \in S_N$ and t, otherwise $\mathbf{x}$ goes to the right. This type of branching was used in our previous work [28]. Furthermore, we only consider *balanced* branching functions, i.e., those mapping half of the examples in X_N to the left, and the other half to the right.

Algorithm 4 shows how to split a node using such balanced branching functions. Essentially, the algorithm goes through all basis vectors in S_N and uses each in turn to construct a branching function. The partition proposed by each is evaluated using the EVALUATE function. The algorithm returns the basis vector together with the associated t that has the highest score. We can alter the optimization criteria by changing the EVALUATE function, below we describe two alternatives.

BAL This heuristic is used in combination with any of the selection methods described below. The split is evaluated in terms of the squared error between the approximation and the true $\mathbf{r}_i$ values (see Section 3.1) after solving the selection problem in the leaves.

ENT This heuristic is inspired by the impurity measure used in decision trees. Let $H(\mathbf{z}_i = 1)$ and $H(\mathbf{z}_i = 0)$ be the entropies of the class labels in left and right partitions, respectively. This heuristic chooses the split that minimizes $\min(H(\mathbf{z}_i = 1), H(\mathbf{z}_i = 0))$.

4.2 Selection heuristics

Now suppose that we have selected a branching function according to some partition heuristic from above. We have to determine how to select subsets of existing basis vectors for the left and right children. We have experimented with the following baselines and heuristics for this:

RNDS Pick two random subsets of size C from S_N to be used as the initial basis vectors in the left and right subtrees of N.

INDS Use the branching function f_N to assign existing basis vectors in S_N either to the left or right subtree. Notice that the bound C on the number of basis vectors may be violated, so some post-processing may be needed.

name	*dataset size*	*nr. attributes*	*source*
ADULT	48842	123	[1]
IJCNN	141691	22	[25]
COD-RNA	488565	8	[29]

Table 1. Data sets used in the experimental section.

OPTS Solve fractional relaxations of $\mathrm{QP}_{\mathbf{p}}$ and $\mathrm{QP}_{\mathbf{q}}$, round $\mathbf{p}$ and $\mathbf{q}$ to obtain integer solutions.

In our experiments, we use identifiers such as BAL-OPTS or ENT-INDS for our heuristics; these mean the combination of the *balanced* partition heuristic using our *optimization* as evaluation function, and the combination of the *entropy* heuristic for partition and *partition-induced* heuristic for selection, respectively.

5 Experiments

In this section we test empirically the validity of our method and the heuristics proposed. Recall that the main goal of this work is to devise a classifier where the basis vectors of the decision rule depend on the input example $\mathbf{x}$. Our approach is to partition the input space and use a different perceptron in each region. The most straightforward strategy is to divide the feature space into two disjoint regions. The first experiment is designed to show that already in this very simple setting we can gain in accuracy by using two different perceptrons in the regions.

In the second experiment we study how well the tree-based approach compares with other budgeted online learning algorithms. We do not expect them to outperform these in accuracy. Instead our main argument is that we can achieve a similar performance by using a considerably smaller number of basis vectors. While our trees can grow very large, and may hence contain a large number of leaf nodes, the total number of kernel computations for each example is bounded by the user-specified budget B. We set $B = 100$ in these experiments. A more thorough study of the effects of B is left for a longer version of this paper.

We compare our methods with the FORGETRON [11] and PROJECTRON [23] algorithms[3], and the STOPTRON, a simple budget perceptron baseline that learns a kernel perceptron but stops updating after budget is reached. The data sets we use are publicly available at [7]. Table 1 shows details of the data sets. Each data set was split into training (50%), testing (25%), and validation (25%) sets. In the experiments that follow we use different portions of the datasets, which will be appropriately described in the text.

In each case we use a Gaussian kernel. The γ parameter of the kernel was optimized for each data by running the basic perceptron algorithm on the training data using different values of γ and choosing the one with the best performance on the validation set.

[3] We use implementations of the DOGMA library [22].

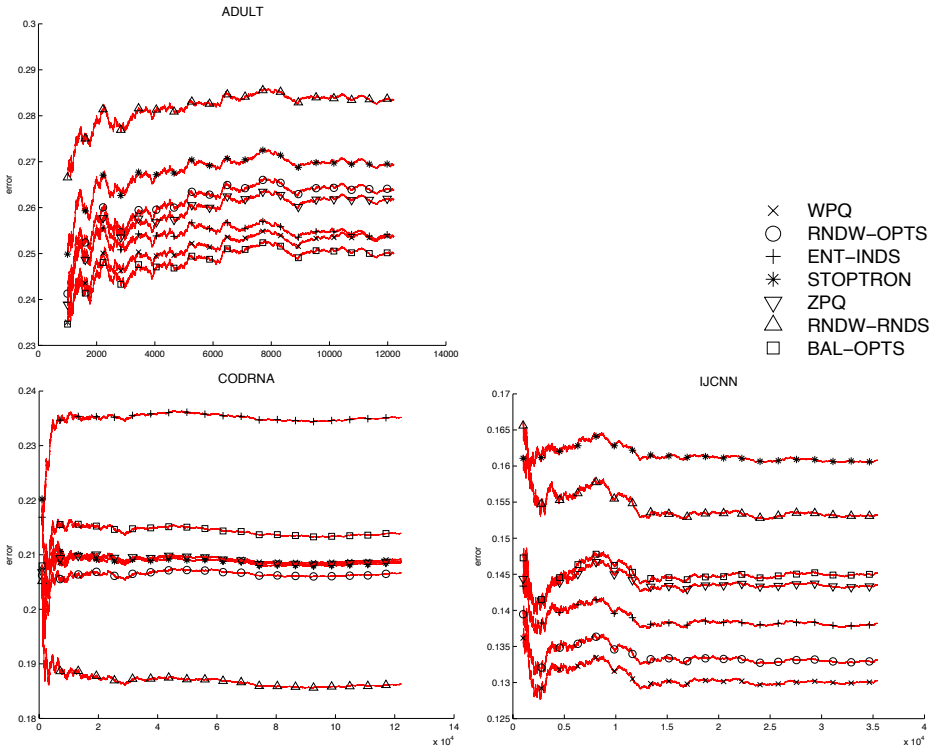

Fig. 1. Results of the splitting experiment showing medians over 30 independent runs of the average cumulative error. The x-axis represents the number of test examples processed so far, and the y-axis shows the average cumulative error obtained up until that point. The different runs are obtained by permuting the training set and so the initial *Stoptron* may differ in each separate run.

5.1 Comparing splitting functions

This experiment compares the performance of using a single STOPTRON against using two STOPTRONS at the leaves of a single split, varying and thus comparing the different splitting criteria. The setting is as follows: we start training a STOPTRON on the training data. Once the number of basis vectors hits B (set to 100), we split the node using each of the techniques described above. At this point training is stopped, and we switch to testing. For each splitting method, the resulting model is applied to the test data. We make a pass over the entire test data, and two STOPTRONS are grown on either side of the splits.

The results in Figure 1. show that splitting in combination with STOPTRON outperforms the global STOPTRON, supporting our hypothesis that splitting may be very useful. Moreover, the results also suggest that it is important to do the split in a smart way. This is indicated by the relatively poor performance of the

completely random RNDW-RDNS heuristic in case of ADULT and IJCNN. Interestingly, RNDW-OPTS performs quite well, suggesting that if the basis vectors are selected carefully, any balanced linear branching function will have a reasonable performance. Finally, our WPQ method is the only one that consistently shows good performance across all three datasets.

5.2 Algorithm comparison

This experiment measures the trade-off between classification time and classification accuracy. We use the number of basis vectors as a proxy for classification time, as this essentially determines the run-time complexity of evaluating the decision function. We compare our perceptron tree approach with the regular perceptron and recently proposed budgeted variants [11, 23]. Again we set B to 100. In case of the PROJECTRON algorithm it is not possible to set the budget explicitly. Instead, there is a parameter η that indirectly affects the size of the resulting model. The results include the values which we found to work best after manual tuning.

Figure 2 shows size vs. error for a representative set of methods. The errors are averages over 10 runs of the mean number of mistakes made by the algorithms after doing one pass over the training data. Ideally one seeks a small model with low error. These can be found in the lower left corner of the plots. We want to point out that for the perceptron trees, the size is set to B because this is the number of basis vectors that contribute to an example's classification. The actual size of the tree is much larger, but only B basis vectors are used to classify an example.

In general our methods show good performance. With the ADULT data the WPQ algorithm is not only the quickest to evaluate, but outperforms every other method in terms of the error. Unlike reported in [23] we observe a fairly poor performance of the projectron with the ADULT data. In the case of IJCNN, WPQ has a slightly higher error rate than the PROJECTRON, but is an order of magnitude smaller. With the COD-RNA data the PROJECTRON algorithm clearly outperforms all other methods both in size and error. However, our methods are competitive with the perceptron while using a much smaller number of basis vectors, which is the main objective of this work.

6 Conclusions and future work

We have proposed a solution to the problem of speeding up classification time with kernel methods that is based on partitioning the feature space into disjoint regions and using budgeted models in each region. This idea follows up preliminary work by one of the authors in [28] and is, to the best of our knowledge, the first time an example-dependent solution is applied to this problem. Clearly, data sets that present different structure in different parts of the feature space should benefit from this approach. Our experiments indicate the potential of

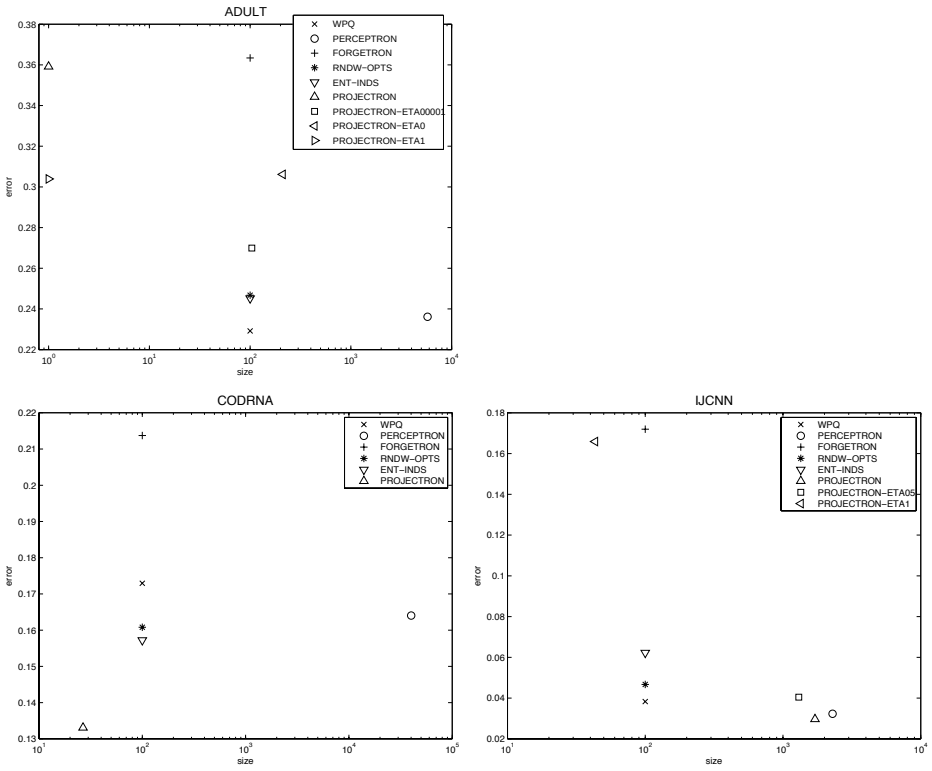

Fig. 2. Number of basis vectors versus the error. X-axis: the budget, i.e., the maximum number of basis vectors that are used to classify an unseen example. Y-axis: the average cumulative test error at the end of learning.

this method, and we want to further study the behavior of the algorithms using more datasets and explore the effect of different parameter settings.

Our optimization criterion in this paper is to approximate the behavior of the perceptron while using a smaller number of basis vectors. The motivation behind this is that in some other learning scenarios, such as ranking for example, one actually does care about the values output by the methods, and not just the labels. Naturally, other objective functions could be considered, such as directly minimizing the classification error in the context of a classification problem. In fact, the ENT heuristic from Section 4 is a first step in this direction. We plan to work on more principled approaches in the future.

The idea of example-dependent selection of basis functions carries naturally over to other domains, such as ensemble or boosting methods, e.g. random forests [4] or gradient boosted decision trees [15]. From a more theoretical perspective, it would be desirable to obtain an understanding of the generalization ability of example-dependent classifiers.

References

1. Asuncion, A., Newman, D.: UCI machine learning repository (2007), http://archive.ics.uci.edu/ml/
2. Bennett, K.P., Blue, J.A.: A support vector machine approach to decision trees. In: Proceedings of The 1998 IEEE International Joint Conference On Neural Networks. pp. 2396–2401 (1998)
3. Bordes, A., Ertekin, S., Weston, J., Bottou, L.: Fast kernel classifiers with online and active learning. The Journal of Machine Learning Research 6, 1619 (2005)
4. Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001)
5. Burges, C.J.C.: Simplified support vector decision rules. In: ICML. pp. 71–77 (1996)
6. Burges, C.J.C., Schölkopf, B.: Improving the accuracy and speed of support vector machines. In: NIPS. pp. 375–381 (1996)
7. Chang, C.C., Lin, C.J.: LIBSVM dataset site (Apr 2010), http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
8. Chen, J.H., Chen, C.S.: Reducing svm classification time using multiple mirror classifiers. IEEE Transactions on Systems, Man, and Cybernetics, Part B 34(2), 1173–1183 (2004)
9. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. pp. 263–270. Association for Computational Linguistics Morristown, NJ, USA (2001)
10. Crammer, K., Kandola, J., Singer, Y.: Online classification on a budget. Advances in neural information processing systems 16 (2004)
11. Dekel, O., Shalev-Shwartz, S., Singer, Y.: The Forgetron: A kernel-based Perceptron on a budget. SIAM Journal on Computing 37(5), 1342–1372 (2008)
12. Dekel, O., Singer, Y.: Support vector machines on a budget. In: Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference. p. 345. The MIT Press (2007)
13. Downs, T., Gates, K., Masters, A.: Exact simplification of support vector solutions. The Journal of Machine Learning Research 2, 293–297 (2002)
14. Freund, Y., Schapire, R.E.: Large margin classification using the perceptron algorithm. Machine Learning 37(3), 277–296 (1999)
15. Friedman, J.: Greedy function approximation: a gradient boosting machine. Annals of Statistics 29(5), 1189–1232 (2001)
16. Gama, J.: Functional trees. Machine Learning 55(3), 219–250 (2004)
17. Joachims, T., Yu, C.N.J.: Sparse kernel svms via cutting-plane training. Machine Learning 76(2-3), 179–193 (2009), European Conference on Machine Learning (ECML) Special Issue
18. Khardon, R., Wachman, G.: Noise tolerant variants of the perceptron algorithm. Journal of Machine Learning Research 8, 227–248 (2007)
19. Kivinen, J., Smola, A., Williamson, R.: Online learning with kernels. IEEE Transactions on Signal Processing 52(8), 2165–2176 (2004)
20. Nair, P., Choudhury, A., Keane, A.: Some greedy learning algorithms for sparse regression and classification with mercer kernels. The Journal of Machine Learning Research 3, 801 (2003)
21. Orabona, F., Keshet, J., Caputo, B.: Bounded Kernel-Based Online Learning. Journal of Machine Learning Research 10, 2643–2666 (2009)

22. Orabona, F.: DOGMA: a MATLAB toolbox for Online Learning (2009), software available at `http://dogma.sourceforge.net`
23. Orabona, F., Keshet, J., Caputo, B.: The projectron: a bounded kernel-based perceptron. In: ICML '08: Proceedings of the 25th international conference on Machine learning. pp. 720–727. ACM, New York, NY, USA (2008)
24. Osuna, E., Girosi, F.: Reducing the run-time complexity of support vector machines. Advances in Kernel Methods: Support Vector Learning, MIT press, Cambridge, MA pp. 271–284 (1999)
25. Prokhorov, D.: IJCNN 2001 neural network competition
26. Rosenblatt, F.: The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review 65(1), 386–407 (1958)
27. Tipping, M.E.: Sparse bayesian learning and the relevance vector machine. Journal of Machine Learning Research 1, 211–244 (2001)
28. Ukkonen, A.: The support vector tree. In: Algorithms and Applications. pp. 244–259 (2010)
29. Uzilov, A., Keegan, J., Mathews, D.: Detection of non-coding RNAs on the basis of predicted secondary structure formation free energy change. BMC bioinformatics 7(1), 173 (2006)
30. Vapnik, V.: The nature of statistical learning theory. Springer Verlag (2000)
31. Weston, J., Bordes, A., Bottou, L.: Online (and offline) on an even tighter budget. In: Proceedings of International Workshop on Artificial Intelligence and Statistics. Citeseer (2005)
32. Wu, M., Schölkopf, B., Bakır, G.: A direct method for building sparse kernel learning algorithms. The Journal of Machine Learning Research 7, 624 (2006)

A Derivation of Equation 3

Denote by $\mathbf{G}_{i\cdot}$ the ith row of $\mathbf{G}$. We write

$$\begin{aligned}
\sum_i \mathbf{z}_i c_{\mathbf{p}}(\mathbf{x}_i) &= \sum_i \mathbf{z}_i \Big((\sum_j \mathbf{p}_j \mathbf{G}_{ij})^2 - 2\mathbf{r}_i \sum_j \mathbf{p}_j \mathbf{G}_{ij} + \mathbf{r}_i^2 \Big) \\
&= \sum_i \mathbf{z}_i \Big(\mathbf{p}^T \mathbf{G}_{i\cdot}^T \mathbf{G}_{i\cdot} \mathbf{p} - 2\mathbf{r}_i \mathbf{G}_{i\cdot} \mathbf{p} + \mathbf{r}_i^2 \Big) \\
&= \mathbf{p}^T \underbrace{\Big(\sum_i \mathbf{z}_i \mathbf{G}_{i\cdot}^T \mathbf{G}_{i\cdot} \Big)}_{\mathbf{Q}_L(\mathbf{z})} \mathbf{p} - 2 \underbrace{\Big(\sum_i \mathbf{z}_i \mathbf{r}_i \mathbf{G}_{i\cdot} \Big)}_{\mathbf{w}_L^T(\mathbf{z})} \mathbf{p} + \sum_i \mathbf{z}_i \mathbf{r}_i^2 \\
&= \mathbf{p}^T \mathbf{Q}_L(\mathbf{z}) \mathbf{p} - 2\mathbf{w}_L^T(\mathbf{z}) \mathbf{p} + C_L,
\end{aligned}$$

Using similar manipulations we obtain

$$\sum_i (1 - \mathbf{z}_i) c_{\mathbf{q}}(\mathbf{x}_i) = \mathbf{q}^T \mathbf{Q}_R(\mathbf{z}) \mathbf{q} - 2\mathbf{w}_R^T(\mathbf{z}) \mathbf{q} + C_R.$$

The matrix $\mathbf{Q}_L(\mathbf{z})$ and the vector $\mathbf{w}_L(\mathbf{z})$ are thus based on all those rows of $\mathbf{G}$ for which $\mathbf{z}_i = 1$. Likewise, the matrix $\mathbf{Q}_R(\mathbf{z})$ and the vector $\mathbf{w}_R(\mathbf{z})$ are based on the rows for which $\mathbf{z}_i = 0$. We obtain Equation 3 by summing the above equations and omitting the constants C_L and C_R.