Machine Learning, 48, 169–191, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Kernel Matching Pursuit

PASCAL VINCENT Vincent@iro.umontreal.ca
YOSHUA BENGIO bengioy@iro.umontreal.ca
Department of IRO, Université de Montréal, C.P. 6128, Montreal, Qc, H3C 3J7, Canada

Editors: Yoshua Bengio and Dale Schuurmans

Abstract. Matching Pursuit algorithms learn a function that is a weighted sum of basis functions, by sequentially appending functions to an initially empty basis, to approximate a target function in the least-squares sense. We show how matching pursuit can be extended to use non-squared error loss functions, and how it can be used to build kernel-based solutions to machine learning problems, while keeping control of the sparsity of the solution. We present a version of the algorithm that makes an optimal choice of both the next basis and the weights of all the previously chosen bases. Finally, links to boosting algorithms and RBF training procedures, as well as an extensive experimental comparison with SVMs for classification are given, showing comparable results with typically much sparser models.

Keywords: kernel methods, matching pursuit, sparse approximation, support vector machines, radial basis functions, boosting

1. Introduction

Recently, there has been a renewed interest for kernel-based methods, due in great part to the success of the *Support Vector Machine* approach (Boser, Guyon, & Vapnik, 1992; Vapnik, 1995). Kernel-based learning algorithms represent the function value $f(x)$ to be learned with a linear combination of terms of the form $K(x, x_i)$, where x_i is generally the input vector associated to one of the training examples, and K is a symmetric positive definite kernel function.

Support Vector Machines (SVMs) are kernel-based learning algorithms in which only a fraction of the training examples are used in the solution (these are called the Support Vectors), and where the objective of learning is to maximize a margin around the decision surface (in the case of classification).

Matching Pursuit was originally introduced in the signal-processing community as an algorithm "*that decomposes any signal into a linear expansion of waveforms that are selected from a redundant dictionary of functions.*" (Mallat & Zhang, 1993). It is a general, greedy, sparse function approximation scheme with the squared error loss, which iteratively adds new functions (i.e. basis functions) to the linear expansion. If we take as "dictionary of functions" the functions $d_i(\cdot)$ of the form $K(\cdot, x_i)$ where x_i is the input part of a training example, then the linear expansion has essentially the same form as a Support Vector Machine. Matching Pursuit and its variants were developed primarily in the signal-processing and wavelets community, but there are many interesting links with the

research on kernel-based learning algorithms developed in the machine learning community. Connections between a related algorithm (*basis pursuit* (Chen, 1995)) and SVMs had already been reported in Poggio and Girosi (1998). More recently, Smola and Schölkopf 2000 show connections between Matching Pursuit, Kernel-PCA, Sparse Kernel Feature analysis, and how this kind of greedy algorithm can be used to compress the design-matrix in SVMs to allow handling of huge data sets. Another recent work, very much related to ours, that also uses a Matching-Pursuit like algorithm is Smola and Bartlett (2001).

Sparsity of representation is an important issue, both for the computational efficiency of the resulting representation, and for its influence on generalization performance (see (Graepel, Herbrich, & Shawe-Taylor, 2000) and (Floyd & Warmuth, 1995)). However the sparsity of the solutions found by the SVM algorithm is hardly controllable, and often these solutions are not very sparse.

Our research started as a search for a flexible alternative framework that would allow us to directly control the sparsity (in terms of number of support vectors) of the solution and remove the requirements of positive definiteness of K (and the representation of K as a dot product in a high-dimensional "feature space"). It lead us to uncover connections between greedy Matching Pursuit algorithms, Radial Basis Function training procedures, and boosting algorithms (Section 4). We will discuss these together with a description of the proposed algorithm and extensions thereof to use margin loss functions.

We first (Section 2) give an overview of the Matching Pursuit family of algorithms (the basic version and two refinements thereof), as a general framework, taking a machine learning viewpoint. We also give a detailed description of our particular implementation that yields a choice of the next basis function to add to the expansion by minimizing simultaneously across the expansion weights and the choice of the basis function, in a computationally efficient manner.

We then show (Section 3) how this framework can be extended, to allow the use of other differentiable loss functions than the squared error to which the original algorithms are limited. This might be more appropriate for some classification problems (although, in our experiments, we have used the squared loss for many classification problems, always with successful results). This is followed by a discussion about margin loss functions, underlining their similarity with more traditional loss functions that are commonly used for neural networks.

In Section 4 we explain how the matching pursuit family of algorithms can be used to build kernel-based solutions to machine learning problems, and how this relates to other machine learning algorithms, namely SVMs, boosting algorithms, and Radial Basis Function training procedures.

Finally, in Section 5, we provide an experimental comparison between SVMs and different variants of Matching Pursuit, performed on artificial data, USPS digits classification, and UCI machine learning databases benchmarks. The main experimental result is that Kernel Matching Pursuit algorithms can yield generalization performance as good as Support Vector Machines, but often using significantly fewer support vectors.

2. Three flavors of matching pursuit

In this section we first describe the basic Matching Pursuit algorithm, as it was introduced by Mallat and Zhang 1993, but from a machine learning perspective rather than a signal processing one. We then present two successive refinements of the basic algorithm.

2.1. *Basic matching pursuit*

We are given l noisy observations $\{y_1, \ldots, y_l\}$ of a target function $f \in \mathcal{H}$ at points $\{x_1, \ldots, x_l\}$. We are also given a finite dictionary $\mathcal{D} = \{d_1, \ldots, d_M\}$ of M functions in a Hilbert space $\mathcal{H}$, and we are interested in sparse approximations of f that are expansions of the form

$$f_N = \sum_{n=1}^{N} \alpha_n g_n \tag{1}$$

where

- N is the number of *basis functions* in the expansion,
- $\{g_1, \ldots, g_N\} \subset \mathcal{D}$ shall be called the *basis* of the expansion,
- $\{\alpha_1, \ldots, \alpha_N\} \in \mathbb{R}^{\mathrm{N}}$ is the set of corresponding *coefficients* of the expansion,
- f_N designs an approximation of f that uses exactly N distinct basis functions taken from the dictionary.

Notice the distinction in notation, between the dictionary functions $\{d_1, \ldots, d_M\}$ ordered as they appear in the dictionary, and the particular dictionary functions $\{g_1, \ldots, g_N\}$ ordered as they appear in the expansion f_N. There is a correspondence between the two, which can be represented by a set of indices $\Gamma = \{\gamma_1, \ldots, \gamma_N\}$ such that $g_i = d_{\gamma_i} \forall i \in \{1..N\}$ with $\gamma_i \in \{1..M\}$. Choosing a basis is equivalent to choosing a set Γ of indices.

We will also make extensive use of the following vector notations:

- For any function $f \in \mathcal{H}$ we will use $\vec{f}$ to represent the l-dimensional vector that corresponds to the evaluation of f on the l training points:

$$\vec{f} = (f(x_1), \ldots, f(x_l)).$$

- $\vec{y} = (y_1, \ldots, y_l)$ is the *target vector*.
- $\vec{R}_N = \vec{y} - \vec{f}_N$ is the *residue*.
- $\langle \vec{h}_1, \vec{h}_2 \rangle$ will be used to represent the usual dot product between two vectors $\vec{h}_1$ and $\vec{h}_2$.
- $\|\vec{h}\|$ will be used to represent the usual L_2 (Euclidean) norm of a vector $\vec{h}$.

The algorithms described below use the dictionary functions as actual functions only when applying the learned approximation on new *test data*. During training, only their

values at the training points is relevant, so that they can be understood as working entirely in an l-dimensional vector space.

The basis $\{g_1, \ldots, g_N\} \subset \mathcal{D}$ and the corresponding coefficients $\{\alpha_1, \ldots, \alpha_N\} \in \mathbb{R}^N$ are to be chosen such that they minimize the squared norm of the residue:

$$\|\vec{R}_N\|^2 = \|\vec{y} - \vec{f}_N\|^2 = \sum_{i=1}^{l} (y_i - f_N(x_i))^2 .$$

This corresponds to reducing the usual squared "reconstruction" error. Later we will see how to extend these algorithms to other kinds of loss functions (Section 3), but for now, we shall consider only least-squares approximations.

In the general case, when it is not possible to use particular properties of the family of functions that constitute the dictionary, finding the optimal basis $\{g_1, \ldots, g_N\}$ for a given number N of allowed basis functions implies an exhaustive search over all possible choices of N basis functions among $M(\frac{M!}{N!(M-N)!}$ possibilities). As it would be computationally prohibitive to try all these combinations, the matching pursuit algorithm proceeds in a greedy, constructive, fashion:

It starts at stage 0 with $\vec{f}_0 = 0$, and recursively appends functions to an initially empty basis, at each stage n, trying to reduce the norm of the residue $\vec{R}_n = \vec{y} - \vec{f}_n$.

Given $\vec{f}_n$ we build

$$\vec{f}_{n+1} = \vec{f}_n + \alpha_{n+1}\vec{g}_{n+1} \tag{2}$$

by searching for $g_{n+1} \in \mathcal{D}$ and for $\alpha_{n+1} \in \mathbb{R}$ that minimize the residual error, i.e. the squared norm of the next residue:

$$\begin{aligned} \|\vec{R}_{n+1}\|^2 &= \|\vec{y} - \vec{f}_{n+1}\|^2 \\ &= \|\vec{y} - (\vec{f}_n + \alpha_{n+1}\vec{g}_{n+1})\|^2 \\ &= \|\vec{R}_n - \alpha_{n+1}\vec{g}_{n+1}\|^2. \end{aligned}$$

Formally:

$$(g_{n+1}, \alpha_{n+1}) = \underset{(g\in\mathcal{D},\alpha\in\mathbb{R})}{\arg\min} \|\vec{R}_n - \alpha\vec{g}\|^2 \tag{3}$$

For any $g \in \mathcal{D}$, the α that minimizes $\|\vec{R}_n - \alpha\vec{g}\|^2$ is given by

$$\begin{aligned} \frac{\partial\|\vec{R}_n - \alpha\vec{g}\|^2}{\partial\alpha} &= 0 \\ -2\langle\vec{g}, \vec{R}_n\rangle + 2\alpha\|\vec{g}\|^2 &= 0 \\ \alpha &= \frac{\langle\vec{g}, \vec{R}_n\rangle}{\|\vec{g}\|^2} \end{aligned} \tag{4}$$

For this optimal value of α, we have

$$\begin{aligned}\|\vec{R}_n - \alpha\vec{g}\|^2 &= \left\|\vec{R}_n - \frac{\langle\vec{g}, \vec{R}_n\rangle}{\|\vec{g}\|^2}\vec{g}\right\|^2 \\ &= \|\vec{R}_n\|^2 - 2\frac{\langle\vec{g}, \vec{R}_n\rangle}{\|\vec{g}\|^2}\langle\vec{g}, \vec{R}_n\rangle + \left(\frac{\langle\vec{g}, \vec{R}_n\rangle}{\|\vec{g}\|^2}\right)^2 \|\vec{g}\|^2 \\ &= \|\vec{R}_n\|^2 - \left(\frac{\langle\vec{g}, \vec{R}_n\rangle}{\|\vec{g}\|}\right)^2 \end{aligned} \tag{5}$$

So the $g \in \mathcal{D}$ that minimizes expression (3) is the one that minimizes (5), which corresponds to maximizing $\| \frac{\langle\vec{g}, \vec{R}_n\rangle}{\|\vec{g}\|} |$. In other words, it is the function in the dictionary whose corresponding vector is "most collinear" with the current residue.

In summary, the g_{n+1} that minimizes expression (3) is the one that maximizes $| \frac{\langle\vec{g}_{n+1}, \vec{R}_n\rangle}{\|\vec{g}_{n+1}\|} |$ and the corresponding α is:

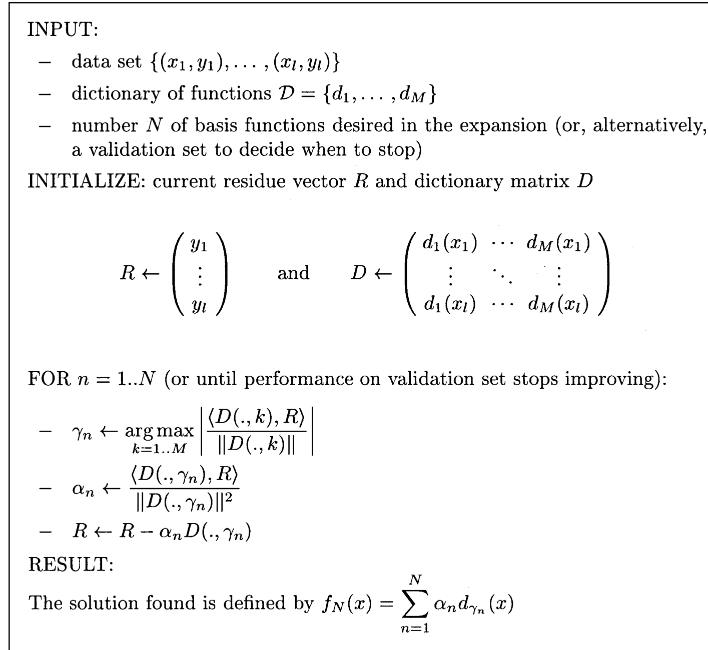
$$\alpha_{n+1} = \frac{\langle\vec{g}_{n+1}, \vec{R}_n\rangle}{\|\vec{g}_{n+1}\|^2}.$$

We have not yet specified how to choose N. Notice that, the algorithm being incremental, we don't necessarily have to fix N ahead of time and try different values to find the best one, we merely have to choose an appropriate criterion to decide when to stop adding new functions to the expansion. In the signal processing literature the algorithm is usually stopped when the *reconstruction error* $\|\vec{R}_N\|^2$ goes below a predefined given threshold. For machine learning problems, we shall rather use the error estimated on an independent validation set[1] to decide when to stop. In any case, N (even though its choice is usually indirect, determined by the early-stopping criterion) can be seen as the primary capacity-control parameter of the algorithm.

The pseudo-code for the corresponding algorithm is given in figure 1 (there are slight differences in the notation, in particular vector $\vec{g}_n$ in the above explanations corresponds to vector $D(., \gamma_n)$ in the more detailed pseudo-code, and R is used to represent a temporary vector always containing the *current* residue, as we don't need to store all intermediate residues $\vec{R}_0..\vec{R}_N$. We also dropped the arrows, as we only work with vectors and matrices, seen as one and two dimensional arrays, and there is no possible ambiguity with corresponding functions).

2.2. *Matching pursuit with back-fitting*

In the basic version of the algorithm, not only is the set of basis functions $g_{1..n}$ obtained at every step n suboptimal, but so are also their $\alpha_{1..n}$ coefficients. This can be corrected in a step often called *back-fitting* or *back-projection* and the resulting algorithm is known as *Orthogonal Matching Pursuit (OMP)* (Pati, Rezaiifar, & Krishnaprasad, 1993; Davis, Mallat, & Zhang, 1994):



INPUT:

- data set $\{(x_1, y_1), \ldots, (x_l, y_l)\}$
- dictionary of functions $\mathcal{D} = \{d_1, \ldots, d_M\}$
- number N of basis functions desired in the expansion (or, alternatively, a validation set to decide when to stop)

INITIALIZE: current residue vector R and dictionary matrix D

$$R \leftarrow \begin{pmatrix} y_1 \\ \vdots \\ y_l \end{pmatrix} \quad \text{and} \quad D \leftarrow \begin{pmatrix} d_1(x_1) & \cdots & d_M(x_1) \\ \vdots & \ddots & \vdots \\ d_1(x_l) & \cdots & d_M(x_l) \end{pmatrix}$$

FOR $n = 1..N$ (or until performance on validation set stops improving):

- $\gamma_n \leftarrow \arg\max_{k=1..M} \left| \frac{\langle D(.,k), R\rangle}{\|D(.,k)\|} \right|$
- $\alpha_n \leftarrow \frac{\langle D(.,\gamma_n), R\rangle}{\|D(.,\gamma_n)\|^2}$
- $R \leftarrow R - \alpha_n D(.,\gamma_n)$

RESULT:

The solution found is defined by $f_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 1. Basic matching pursuit algorithm.

While still choosing g_{n+1} as previously (Eq. (3)), we recompute the optimal set of coefficients $\alpha_{1..n+1}$ at each step instead of only the last α_{n+1}:

$$\alpha_{1..n+1}^{(n+1)} = \arg\min_{(\alpha_{1..n+1} \in \mathbb{R}^{n+1})} \left\| \left(\sum_{k=1}^{n+1} \alpha_k \vec{g}_k \right) - \vec{y} \right\|^2 \tag{6}$$

Note that this is just like a linear regression with parameters $\alpha_{1..n+1}$. This *back-projection* step also has a geometrical interpretation:

Let $\mathcal{B}_n$ the sub-space of $\mathbb{R}^l$ spanned by the basis $(\vec{g}_1, \ldots, \vec{g}_n)$ and let $\mathcal{B}_n^{\perp} = \mathbb{R}^l - \mathcal{B}_n$ be its orthogonal complement. Let $P_{\mathcal{B}_n}$ and $P_{\mathcal{B}_n^{\perp}}$ denote the projection operators on these subspaces.

Then, any $\vec{g} \in \mathbb{R}^l$ can be decomposed as $\vec{g} = P_{\mathcal{B}_n}\vec{g} + P_{\mathcal{B}_n^{\perp}}\vec{g}$ (see figure 2).

Ideally, we want the residue $\vec{R}_n$ to be as small as possible, so given the basis at step n, we want $\vec{f}_n = P_{\mathcal{B}_n}\vec{y}$ and $\vec{R}_n = P_{\mathcal{B}_n^{\perp}}\vec{y}$. This is what (6) insures.

But whenever we append the next $\alpha_{n+1}\vec{g}_{n+1}$ found by (3) to the expansion, we actually add its two orthogonal components:

- $P_{\mathcal{B}_n^{\perp}}\alpha_{n+1}\vec{g}_{n+1}$ contributes to reducing the norm of the residue.
- $P_{\mathcal{B}_n}\alpha_{n+1}\vec{g}_{n+1}$ which increases the norm of the residue.

However, as the latter part belongs to $P_{\mathcal{B}_n}$ it can be compensated for by adjusting the previous coefficients of the expansion: this is what the *back-projection* does.

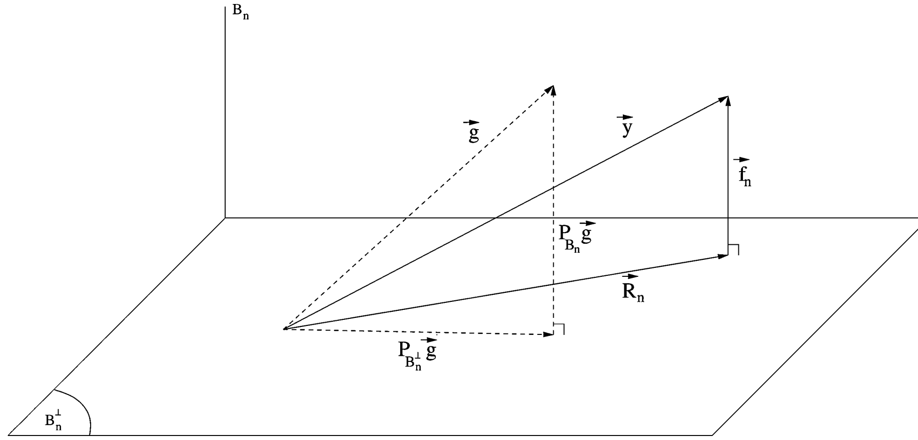


Figure 2. Geometrical interpretation of matching pursuit and back-projection.

Davis, Mallat, and Zhang (1994) suggest maintaining an additional orthogonal basis of the $\mathcal{B}_n$ space to facilitate this back-projection, which results in a computationally efficient algorithm.[2]

2.3. Matching pursuit with pre-fitting

With *back-fitting*, the choice of the function to append at each step is made regardless of the later possibility to update all weights: as we find g_{n+1} using (3) and *only then* optimize (6), we might be picking a dictionary function other than the one that would give the best fit.

Instead, it is possible to directly optimize

$$\left(g_{n+1}, \alpha_{1..n+1}^{(n+1)}\right) = \underset{(g\in\mathcal{D},\alpha_{1..n+1}\in\mathbb{R}^{n+1})}{\arg\min} \left\| \left(\sum_{k=1}^{n} \alpha_k \vec{g}_k\right) + \alpha_{n+1}\vec{g} - \vec{y} \right\|^2 \tag{7}$$

We shall call this procedure *pre-fitting* to distinguish it from the former *back-fitting* (as back-fitting is done only *after* the choice of g_{n+1}).

This can be achieved almost as efficiently as *back-fitting*. Our implementation maintains a representation of both the target and all dictionary vectors as a decomposition into their projections on $\mathcal{B}_n$ and $\mathcal{B}_n^\perp$:

As before, let $\mathcal{B}_n = span(\vec{g}_1, \ldots, \vec{g}_n)$. We maintain at each step a representation of each dictionary vector $\vec{d}$ as the sum of two orthogonal components:

– component $\vec{d}_{\mathcal{B}_n} = P_{\mathcal{B}_n}\vec{d}$ lies in the space $\mathcal{B}_n$ spanned by the current basis and is expressed as a linear combination of current basis vectors (it is an n-dimensional vector).
– component $\vec{d}_{\mathcal{B}_n^\perp} = P_{\mathcal{B}_n^\perp}\vec{d}$ lies in $\mathcal{B}_n$'s orthogonal complement and is expressed in the original l-dimensional vector space coordinates.

We also maintain the same representation for the target $\vec{y}$, namely its decomposition into the current expansion $\vec{f}_n \in \mathcal{B}_n$ plus the orthogonal residue $\vec{R}_n \in \mathcal{B}_n^\perp$.

Pre-fitting is then achieved easily by considering only the components in $\mathcal{B}_n^\perp$: we choose g_{n+1} as the $g \in \mathcal{D}$ whose $\vec{g}_{\mathcal{B}_n^\perp}$ is most collinear with $\vec{R}_n \in \mathcal{B}_n^\perp$. This procedure requires, at every step, only two passes through the dictionary (searching $\vec{g}_{n+1}$, then updating the representation) where basic matching pursuit requires one.

The detailed pseudo-code for this algorithm is given in figure 3.

INPUT:

- data set $\{(x_1, y_1), \ldots, (x_l, y_l)\}$
- dictionary of functions $\mathcal{D} = \{d_1, \ldots, d_M\}$
- number N of basis functions desired in the expansion (or, alternatively, a validation set to decide when to stop)

INITIALIZE:

Current residue vector R and dictionary components $D_{\mathcal{B}^\perp}$ and $D_{\mathcal{B}}$

$$R \leftarrow \begin{pmatrix} y_1 \\ \vdots \\ y_l \end{pmatrix} \quad \text{and} \quad D_{\mathcal{B}^\perp} \leftarrow \begin{pmatrix} d_1(x_1) & \cdots & d_M(x_1) \\ \vdots & \ddots & \vdots \\ d_1(x_l) & \cdots & d_M(x_l) \end{pmatrix}$$

$D_{\mathcal{B}}$ is initially empty, and gets appended an additional row at each step (thus, ignore the expressions that involve $D_{\mathcal{B}}$ during the first iteration when $n = 1$)

FOR $n = 1..N$ (or until performance on validation set stops improving):

- $\gamma_n \leftarrow \arg\max_{k=1..M} \left| \frac{\langle D_{\mathcal{B}^\perp}(.,k), R \rangle}{\|D_{\mathcal{B}^\perp}(.,k)\|} \right|$
- $\alpha_n \leftarrow \frac{\langle D_{\mathcal{B}^\perp}(.,\gamma_n), R \rangle}{\|D_{\mathcal{B}^\perp}(.,\gamma_n)\|^2}$
- the $\mathcal{B}^\perp$ component of $\alpha_n d_{\gamma_n}$ reduces the residue:
  $R \leftarrow R - \alpha_n D_{\mathcal{B}^\perp}(.,\gamma_n)$
- compensate for the $\mathcal{B}$ component of $\alpha_n d_{\gamma_n}$ by adjusting previous α:
  $(\alpha_1, \ldots, \alpha_{n-1}) \leftarrow (\alpha_1, \ldots, \alpha_{n-1}) - \alpha_n D_{\mathcal{B}}(.,\gamma_n)'$
- Now update the dictionary representation to take into account the new basis function d_{γ_n}:
  FOR $i = 1..M$ AND $i \neq \gamma_n$:
  $$\beta_i \leftarrow \frac{\langle D_{\mathcal{B}^\perp}(.,\gamma_n), D_{\mathcal{B}^\perp}(.,i) \rangle}{\|D_{\mathcal{B}^\perp}(.,\gamma_n)\|^2}$$
  $$D_{\mathcal{B}^\perp}(.,i) \leftarrow D_{\mathcal{B}^\perp}(.,i) - \beta_i D_{\mathcal{B}^\perp}(.,\gamma_n)$$
  $$D_{\mathcal{B}}(.,i) \leftarrow D_{\mathcal{B}}(.,i) - \beta_i D_{\mathcal{B}}(.,\gamma_n)$$
- $D_{\mathcal{B}^\perp}(.,\gamma_n) \leftarrow 0$
- $D_{\mathcal{B}}(.,\gamma_n) \leftarrow 0$
- $\beta_{\gamma_n} \leftarrow 1$
- $D_{\mathcal{B}} \leftarrow \begin{pmatrix} D_{\mathcal{B}} \\ \beta_1, \ldots, \beta_M \end{pmatrix}$

RESULT:

The solution found is defined by $f_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 3. Matching pursuit with pre-fitting.

2.4. Summary of the three variations of MP

Regardless of the computational tricks that use orthogonality properties for efficient computation, the three versions of matching pursuit differ only in the way the next function to append to the basis is chosen and the α coefficients are updated at each step n:

- *Basic version*: We find the optimal g_n to append to the basis and its optimal α_n, while keeping all other coefficients fixed (Eq. (3)).
- *back-fitting version*: We find the optimal g_n while keeping all coefficients fixed (Eq. (3)). Then we find the optimal set of coefficients $\alpha_{1..n}^{(n)}$ for the new basis (Eq. (6)).
- *pre-fitting version*: We find at the same time the optimal g_n and the optimal set of coefficients $\alpha_{1..n}^{(n)}$ (Eq. (7)).

When making use of orthogonality properties for efficient implementations of the back-fitting and pre-fitting version (as in our previously described implementation of the pre-fitting algorithm), all three algorithms have a computational complexity of the same order $\mathcal{O}(N \cdot M \cdot l)$.

3. Extension to non-squared error loss

3.1. Gradient descent in function space

It has already been noticed that boosting algorithms are performing a form of gradient descent in function space with respect to particular loss functions (Schapire et al., 1998; Mason et al., 2000). Following Fiedman (1999), the technique can be adapted to extend the Matching Pursuit family of algorithms to optimize arbitrary differentiable loss functions, instead of doing least-squares fitting.

Given a loss function $L(y_i, f_n(x_i))$ that computes the cost of predicting a value of $f_n(x_i)$ when the true target was y_i, we use an alternative residue $\tilde{R}_n$ rather than the usual $\vec{R}_n = \vec{y} - \vec{f}_n$ when searching for the next dictionary element to append to the basis at each step.

$\tilde{R}_n$ is the direction of steepest descent (the gradient) in function space (evaluated at the data points) with respect to L:

$$\tilde{R}_n = \left(-\frac{\partial L(y_1, \vec{f}_n(x_1))}{\partial \vec{f}_n(x_1)}, \ldots, -\frac{\partial L(y_l, \vec{f}_n(x_l))}{\partial \vec{f}_n(x_l)} \right) \tag{8}$$

i.e. $\vec{g}_{n+1}$ is chosen such that it is most collinear with this gradient:

$$g_{n+1} = \arg\max_{g \in \mathcal{D}} \left| \frac{\langle \vec{g}_{n+1}, \tilde{R}_n \rangle}{\|\vec{g}_{n+1}\|} \right| \tag{9}$$

A line-minimization procedure can then be used to find the corresponding coefficient

$$\alpha_{n+1} = \arg\min_{\alpha \in \mathbb{R}} \sum_{i=1}^{l} L\left(y_i, f_n(x_i) + \alpha \vec{g}_{n+1}(x_i)\right) \tag{10}$$

This would correspond to *basic matching pursuit* (notice how the original squared-error algorithm is recovered when L is the squared error: $L(a, b) = \frac{1}{2}(a-b)^2$).

It is also possible to do *back-fitting*, by re-optimizing all $\alpha_{1..n+1}$ (instead of only α_{n+1}) to minimize the target cost (with a conjugate gradient optimizer for instance):

$$\alpha_{1..n+1}^{(n+1)} = \arg\min_{(\alpha_{1..n+1} \in \mathbb{R}^{n+1})} \sum_{i=1}^{l} L\left(y_i, \sum_{k=1}^{n+1} \alpha_k g_k(x_i)\right) \tag{11}$$

As this can be quite time-consuming (we cannot use any orthogonality property in this general case), it may be desirable to do it every few steps instead of every single step. The corresponding algorithm is described in more details in the pseudo-code of figure 4 (as previously there are slight differences in the notation, in particular g_k in the above explanation corresponds to vector $D(., \gamma_k)$ in the more detailed pseudo-code).

Finally, let's mention that it should in theory also be possible to do *pre-fitting* with an arbitrary loss functions, but finding the optimal $\{g_{k+1} \in \mathcal{D}, \alpha_{1..k+1} \in \mathbb{R}^{k+1}\}$ in the general case (when we cannot use any orthogonal decomposition) would involve solving equation 11 in turn for *each* dictionary function in order to choose the next one to append to the basis, which is computationally prohibitive.

3.2. *Margin loss functions versus traditional loss functions for classification*

Now that we have seen how the matching pursuit family of algorithms can be extended to use arbitrary loss functions, let us discuss the merits of various loss functions.

In particular the relationship between loss functions and the notion of *margin* is of primary interest here, as we wanted to build an alternative to SVMs.[3]

While the original notion of margin in classification problems comes from the geometrically inspired hard-margin of linear SVMs (the smallest Euclidean distance between the decision surface and the training points), a slightly different perspective has emerged in the boosting community along with the notion of margin loss function. The margin quantity $m = y\hat{f}(x)$ of an individual data point (x, y), with $y \in \{-1, +1\}$ can be understood as a confidence measure of its classification by function $\hat{f}$, while the class decided for is given by $\text{sign}(\hat{f}(x))$. A *margin loss function* is simply a function of this margin quantity m that is being optimized.

It is possible to formulate SVM training such as to show the SVM margin loss function:

Let φ be the mapping into the "feature-space" of SVMs, such that $\langle \varphi(x_i), \varphi(x_j) \rangle = K(x_i, x_j)$.

The SVM solution can be expressed in this feature space as

$$\hat{f}(x) = \langle w, \varphi(x) \rangle + b$$

where $w = \sum_{x_i \in \mathcal{SV}} \alpha_i y_i \varphi(x_i)$, $\mathcal{SV}$ being the set of support vectors.

INPUT:

- data set $\{(x_1, y_1), \ldots, (x_l, y_l)\}$
- dictionary of functions $\mathcal{D} = \{d_1, \ldots, d_M\}$
- number N of basis functions desired in the expansion (or, alternatively, a validation set to decide when to stop)
- how often to do a full back-fitting: every p update steps
- a loss function L

INITIALIZE: current approximation $\hat{f}$ and dictionary matrix D
Notice that $\hat{f}$ here is the *current approximation vector*, which changes at every step n, so that $\hat{f}_i$ here corresponds to $f_n(x_i)$ in the accompanying text.

$$\hat{f} = \begin{pmatrix} \hat{f}_0 \\ \vdots \\ \hat{f}_l \end{pmatrix} \leftarrow 0 \quad \text{and} \quad D \leftarrow \begin{pmatrix} d_1(x_1) & \cdots & d_M(x_1) \\ \vdots & \ddots & \vdots \\ d_1(x_l) & \cdots & d_M(x_l) \end{pmatrix}$$

FOR $n = 1..N$ (or until performance on validation set stops improving):

- $\tilde{R} \leftarrow \begin{pmatrix} -\frac{\partial L(y_1, \hat{f}_1)}{\partial \hat{f}_1} \\ \vdots \\ -\frac{\partial L(y_l, \hat{f}_n)}{\partial \hat{f}_l} \end{pmatrix}$
- $\gamma_n \leftarrow \arg\max_{k=1..M} \left| \frac{\langle D(.,k), \tilde{R} \rangle}{\|D(.,k)\|} \right|$
- If n *is not* a multiple of p do a simple line minimization:

$$\alpha_n \leftarrow \arg\min_{\alpha \in \mathbb{R}} \sum_{i=1}^{l} L(y_i, \hat{f}_i + \alpha D(i, \gamma_n))$$

  and update $\hat{f}$: $\hat{f} \leftarrow \hat{f} + \alpha_n D(., \gamma_n)$
- If n *is* a multiple of p do a full back-fitting (for ex. with gradient descent):

$$\alpha_{1..n} \leftarrow \arg\min_{\alpha_{1..n} \in \mathbb{R}^n} \sum_{i=1}^{l} L(y_i, \sum_{k=1}^{n} \alpha_k D(i, \gamma_k))$$

  and recompute $\hat{f} \leftarrow \sum_{k=1}^{n} \alpha_k D(., \gamma_k)$

RESULT:

The solution found is defined by $f_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 4. Back-fitting matching pursuit algorithm with non-squared loss.

The SVM problem is usually formulated as minimizing

$$\|w\|^2 + C \sum_{i=1}^{l} \xi_i \tag{12}$$

subject to constraints $y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i$ and $\xi_i \geq 0, \forall i$. C is the "box-constraint" parameter of SVMs, trading off margin with training errors.

The two constraints for each ξ_i can be rewritten as a single one:

$$\xi_i \geq \max(0, 1 - y_i(\langle w, x_i \rangle + b))$$
or equivalently: $\xi_i \geq [1 - y_i \hat{f}(x_i)]_+$.

The notation $[x]_+$ is to be understood as the function that gives $[x]_+ = x$ when $x > 0$ and 0 otherwise.

As we are minimizing an expression containing a term $\sum_{i=1}^{l} \xi_i$, the inequality constraints over the ξ_i can be changed into equality constraints: $\xi_i = [1 - y_i \hat{f}(x_i)]_+$. Replacing ξ_i in Eq. (12), and multiplying by C we get the following alternative formulation of the SVM problem, where there are no more explicit constraints (they are implicit in the criterion optimized):

Minimize
$$\sum_{i=1}^{l} [1 - y_i \hat{f}(x_i)]_+ + \frac{1}{C}\|w\|^2. \tag{13}$$

Let $m = y\hat{f}(x)$ the *individual margin* at point x. (13) is clearly the sum of a margin loss function and a regularization term.

It is interesting to compare this *margin loss function* to those used in *boosting algorithms* and to the more traditional cost functions. The loss functions that boosting algorithms optimize are typically expressed as functions of m. Thus AdaBoost (Schapire et al., 1998) uses an exponential (e^{-m}) margin loss function, LogitBoost (Friedman, Hastie, & Tibshirani, 1998) uses the negative binomial log-likelihood, $\log_2(1 + e^{-2m})$, whose shape is similar to a smoothed version of the soft-margin SVM loss function $[1 - m]_+$, and Doom II (Mason et al., 1999) approximates a theoretically motivated margin loss with $1 - \tanh(m)$. As can be seen in figure 5 (left), all these functions encourage large positive margins, and differ mainly in how they penalize large negative ones. In particular $1 - \tanh(x)$ is expected to be more robust, as it won't penalize outliers to excess.

It is enlightening to compare these with the more traditional loss functions that have been used for neural networks in classification tasks (i.e. $y \in \{-1, +1\}$), when we express them as functions of m.

- Squared loss: $(\hat{f}(x) - y)^2 = (1 - m)^2$
- Squared loss after tanh with modified target[4]:
  $(\tanh(\hat{f}(x)) - 0.65y)^2 = (0.65 - \tanh(m))^2$

Both are illustrated on figure 5 (bottom). Notice how the squared loss after tanh appears similar to the margin loss function used in Doom II, except that it slightly increases for large positive margins, which is why it behaves well and does not saturate even with unconstrained weights (boosting algorithms impose further constraints on the weights, here denoted α's).

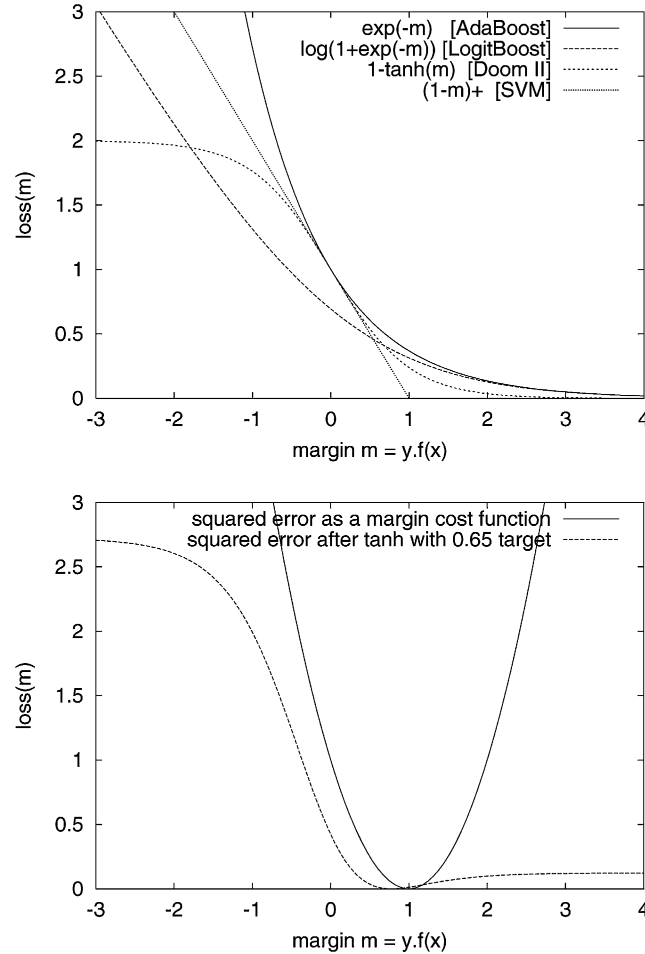


Figure 5. Boosting and SVM margin loss functions (top) vs. traditional loss fuctions (bottom) viewed as functions of the margin. Interestingly the last-born of the margin motivated loss functions (used in Doom II) is similar to the traditional squared error after tanh.

4. Kernel matching pursuit and links with other paradigms

4.1. Matching pursuit with a kernel-based dictionary

Kernel Matching Pursuit (KMP) is simply the idea of applying the Matching Pursuit family of algorithms to problems in machine learning, using a kernel-based dictionary:

Given a kernel function $K : \mathbb{R}^{d} \times \mathbb{R}^{d} \to \mathbb{R}$, we use as our dictionary the kernel centered on the training points: $\mathcal{D} = \{d_i = K(\cdot, x_i) \mid i = 1..l\}$. Optionally, the constant function can also be included in the dictionary, which accounts for a bias term b: the functional form of

approximation $\vec{f}_N$ then becomes

$$\vec{f}_N(x) = b + \sum_{n=1}^{N} \alpha_n K\left(x, x_{\gamma_n}\right) \tag{14}$$

where the $\gamma_{1..N}$ are the indices of the "support points". During training we only consider the values of the dictionary functions at the training points, so that it amounts to doing Matching in a vector-space of dimension l.

When using a squared error loss,[5] the complexity of all three variations of KMP (basic, back-fitting and pre-fitting) is $\mathcal{O}(N.M.l) = \mathcal{O}(N.l^2)$ if we use all the training data as candidate support points. But it is also possible to use a random subset of the training points as support candidates (which yields a $M < l$).

We would also like to emphasize the fact that the use of a dictionary gives a lot of *additional flexibility* to this framework, as it is possible to include any kind of function into it, in particular:

- There is no restriction on the shape of the kernel (no positive-definiteness constraint, could be asymmetrical, etc.).
- The dictionary could include more than a single fixed kernel shape: it could mix different kernel types to choose from at each point, allowing for instance the algorithm to choose among several widths of a Gaussian for each support point (a similar extension has been proposed for SVMs by Weston et al., 1999).
- Similarly, the dictionary could easily be used to constrain the algorithm to use a kernel shape specific to each *class*, based on prior-knowledge.
- The dictionary can incorporate non-kernel based functions (we already mentioned the constant function to recover the bias term b, but this could also be used to incorporate prior knowledge). In this later aspect, the work of Smola, Friess, and Schölkopf, 1999 on *semi-parametric* SVMs offers an interesting advance in that direction, within the SVM framework. This remains a very interesting avenue for further research.
- For huge data sets, a reduced subset can be used as the dictionary to speed up the training.

However in this study, we restrict ourselves to using a single fixed kernel, so that the resulting functional form is the same as the one obtained with standard SVMs.

4.2. Similarities and differences with SVMs

The functional form (14) is very similar to the one obtained with the *Support Vector Machine* (SVM) algorithm (Boser, Guyon, & Vapnik, 1992), the main difference being that SVMs impose further constraints on $\alpha_{1..N}$.

However the quantity optimized by the SVM algorithm is quite different from the KMP greedy optimization, especially when using a squared error loss. Consequently the support vectors and coefficients found by the two types of algorithms are usually different (see our experimental results in Section 5).

Another important difference, and one that was a motivation for this research, is that in KMP, capacity control is achieved by *directly* controlling the sparsity of the solution, i.e. the number N of support vectors, whereas the capacity of SVMs is controlled through the box-constraint parameter C, which has an indirect and hardly controllable influence on sparsity. See Graepel, Herbrich, and Shawe-Taylor (2000) for a discussion on the merits of sparsity and margin, and ways to combine them.

4.3. Link with radial basis functions

Squared-error KMP with a Gaussian kernel and pre-fitting appears to be identical to a particular Radial Basis Functions training algorithm called *Orthogonal Least Squares RBF* (Chen, Cowan, & Grant, 1991) (OLS-RBF).

In Schölkopf et al. (1997) SVMs were compared to "classical RBFs", where the RBF centers were chosen by unsupervised k-means clustering, and SVMs gave better results. To our knowledge, however, there has been no experimental comparison between OLS-RBF and SVMs, although their resulting functional forms are very much alike. Such an empirical comparison is one of the contributions of this paper. Basically our results (Section 5) show OLS-RBF (i.e. squared-error KMP) to perform as well as Gaussian SVMs, while allowing a tighter control of the number of support vectors used in the solution.

4.4. Boosting with kernels

KMP in its basic form generalized to using non-squared error is also very similar to boosting algorithms (Freund & Schapire, 1996; Friedman, Hastie, & Tibshirani, 1998), in which the chosen class of weak learners would be the set of kernels centered on the training points. These algorithms differ mainly in the loss function they optimize, which we have already discussed in Section 3.2.

In this respect, a very much related research is the work of Singer (2000) on *Leveraged Vector Machines*. The proposed boosting algorithm also builds support vector solutions to classification problems using kernel-based weak learners and similarly shows good performance with typically sparser models.

4.5. Matching pursuit versus basis pursuit

Basis Pursuit (Chen, 1995) is an alternative algorithm that essentially attempts to achieve the same goal as Matching Pursuit, namely to build a sparse approximation of a target function using a possibly over-complete dictionary of functions. It is sometimes believed to be a superior approach[6] because contrary to Matching Pursuit, which is a greedy algorithm, Basis Pursuit uses Linear Programming techniques to find the *exact* solution to the following problem:

$$\alpha_{1..M} = \underset{\alpha_{1..M}\in\mathbb{R}^{M}}{\arg\min} \left\| \left(\sum_{k=1}^{M} \alpha_k \vec{d}_k \right) - \vec{y} \right\|^2 + \lambda \|\alpha\|_1 \tag{15}$$

where $\|\alpha\|_1 = \sum_{k=1}^{M} |\alpha_k|$ is used to represent the L_1 norm of $\alpha_{1..M}$.

P1: Jagdish(GNM)
Machine Learning KL1429-08 December 3, 2001 8:21

The added $\lambda\|\alpha\|_1$ penalty term will drive a large number of the coefficients to 0 and thus lead to a sparse solution, whose sparsity can be controlled by appropriately tuning the hyper-parameter λ.

However we would like to point out that, as far as the primary goal is good *sparsity*, i.e. using the smallest *number* of basis functions in the expansion, both algorithms are approximate: Matching Pursuit is greedy, while Basis Pursuit finds an exact solution, but to an approximate problem (the exact problem could be formulated as solving an equation similar to (15) but where the L_0 norm would be used in the penalty term instead of the L_1 norm).

In addition Matching Pursuit had a number of advantages over Basis Pursuit in our particular setting:

- It is very simple and computationally efficient, while Basis Pursuit requires the use of sophisticated Linear Programming techniques to tackle large problems.
- It is constructive, adding the basis functions one by one to the expansion, which allows us to use a simple early-stopping procedure to control optimal sparsity. In contrast, a Basis Pursuit approach implies having to tune the hyper-parameter λ, running the optimization several times with different values to find the best possible choice.

But other than that, we might as well have used Basis Pursuit and would probably have achieved very similar experimental results. We should also mention the works of Poggio and Girosi (1998) which draws an interesting parallel between Basis Pursuit and SVMs, as well as Gunn and Kandola (2001) who use Basis Pursuit with ANOVA kernels to obtain sparse models with improved interpretability.

4.6. Kernel matching pursuit versus kernel perceptron

The perceptron algorithm (Rosenblatt, 1957) and extensions thereof (Gallant, 1986) are among the simplest algorithms for building linear classifiers. As it is a dot-product based algorithm, the *kernel trick* introduced by Aizerman, Braverman, and Rozonoer (1964) readily applies, allowing a straightforward extension to build non-linear decision surfaces in input-space, in the same way this trick is used for SVMs.

For recent research on the Kernel Perceptron, see the very interesting work of Freund and Schapire 1998, and also Graepel, Herbrich, and Shawe-Taylor (2000) who derive theoretical bounds on their generalization error. Kernel Perceptrons are shown to produce solutions that are typically more sparse than SVMs while retaining comparable recognition accuracies.

Both Kernel Matching Pursuit and Kernel Perceptron appear to be simple (they do not involve complex quadratic or linear programming) and efficient greedy algorithms for finding sparse kernel-based solutions to classification problems. However there are major differences between the two approaches:

- Kernel Matching Pursuit does not use the Kernel trick to implicitly work in a higher-dimensional mapped feature-space: it works directly in input-space. Thus it is possible to use specific Kernels that don't necessarily satisfy Mercer's conditions. This is especially interesting if you think of K as a kind of *similarity measure* between input patterns, that

could be engineered to include prior knowledge, or even learned, as it is not always easy, nor desirable, to enforce positive-definiteness in this perspective.
- The perceptron algorithm is initially a classification algorithm, while Matching Pursuit is originally more of a regression algorithm (approximation in the least-squares sense), although the proposed extension to non-squared loss and the discussion on margin-loss functions (see Section 3) further blurs this distinction. The main reason why we use this algorithm for binary classification tasks rather than regression, although the latter would seem more natural, is that our primary purpose was to compare its performance to classification-SVMs.[7]
- Similar to SVMs, the solution found by the Kernel Perceptron algorithm depends only on the retained support vectors, while the coefficients learned by Kernel Matching Pursuit depend on *all* training data, not only on the set of support vectors chosen by the algorithm. This implies that current theoretical results on generalization bounds that are derived for sparse SVM or Perceptron solutions (Vapnik, 1995; Vapnik, 1998; Littlestone & Warmuth, 1986; Floyd & Warmuth, 1995; Graepel, Herbrich, & Shawe-Taylor, 2000) cannot be readily applied to KMP. On the other hand, KMP solutions may require less support vectors than Kernel Perceptron for precisely this same reason: the information on all data points is used, without the need that they appear as support vectors in the solution.

5. Experimental results on binary classification

Throughout this section:

- any mention of KMP without further specification of the loss function means least-squares KMP (also sometimes written *KMP-mse*).
- *KMP-tanh* refers to KMP using squared error after a hyperbolic tangent with modified targets (which behaves more like a typical *margin loss function* as we discussed earlier in Section 3.2).
- Unless otherwise specified, we used the *pre-fitting matching pursuit* algorithm of figure 3 to train least-squares KMP.
- To train *KMP-tanh* we always used the *back-fitting matching pursuit with non-squared loss* algorithm of figure 4 with a conjugate gradient optimizer to optimize the $\alpha_{1..n}$.[8]

5.1. 2D experiments

Figure 6 shows a simple 2D binary classification problem with the decision surface found by the three versions of squared-error KMP (basic, back-fitting and pre-fitting) and a hard-margin SVM, when using the same Gaussian kernel.

We fixed the number N of support points for the pre-fitting and back-fitting versions to be the same as the number of support points found by the SVM algorithm. The aim of this experiment was to illustrate the following points:

- Basic KMP, after 100 iterations, during which it mostly cycled back to previously chosen support points to improve their weights, is still unable to separate the data points. This

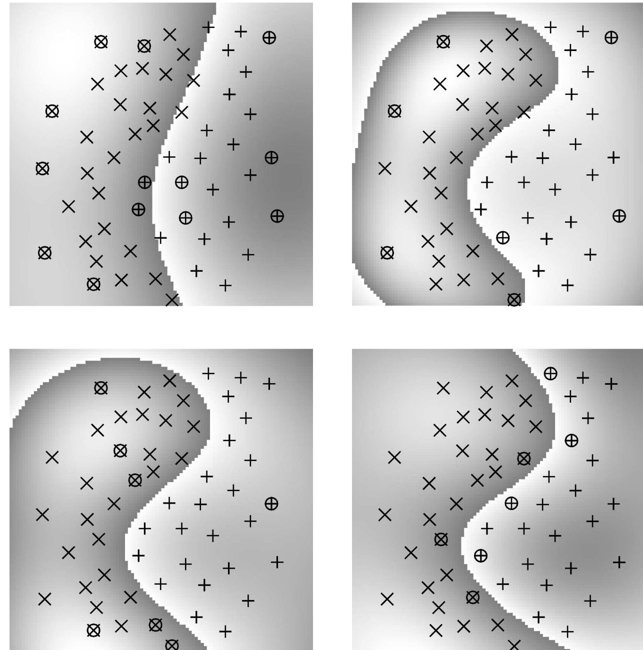


Figure 6. Top left: 100 iterations of basic KMP; Top right: 7 iterations of KMP back-fitting; Bottom left: 7 iterations of KMP pre-fitting, Bottom right: SVM. Classes are $+$ and $\times$. Support vectors are circled. Pre-fitting KMP and SVM appear to find equally reasonable solutions, though using different support vectors. Only SVM chooses its support vectors close to the decision surface. Back-fitting chooses yet another support set, and its decision surface appears to have a slightly worse margin. As for basic KMP, after 100 iterations during which it mostly cycled back to previously chosen support points to improve their weights, it appears to use more support vectors than the others while still being unable to separate the data points, and is thus a bad choice if we want sparse solutions.

shows that the back-fitting and pre-fitting versions are a useful improvement, while the basic algorithm appears to be a bad choice if we want sparse solutions.

– The back-fitting and pre-fitting KMP algorithms are able to find a reasonable solution (the solution found by pre-fitting looks slightly better in terms of margin), but choose different support vectors than SVM, that are not necessarily close to the decision surface (as they are in SVMs). It should be noted that the *Relevance Vector Machine* (Tipping, 2000) similarly produces[9] solutions in which the *relevance vectors* do not lie close to the border.

Figure 7, where we used a simple dot-product kernel (i.e. linear decision surfaces), illustrates a problem that can arise when using least-squares fit: since the squared error penalizes large positive margins, the decision surface is drawn towards the cluster on the lower right, at the expense of a few misclassified points. As expected, the use of a tanh loss function appears to correct this problem.

P1: Jagdish(GNM)
Machine Learning KL1429-08 December 3, 2001 8:21

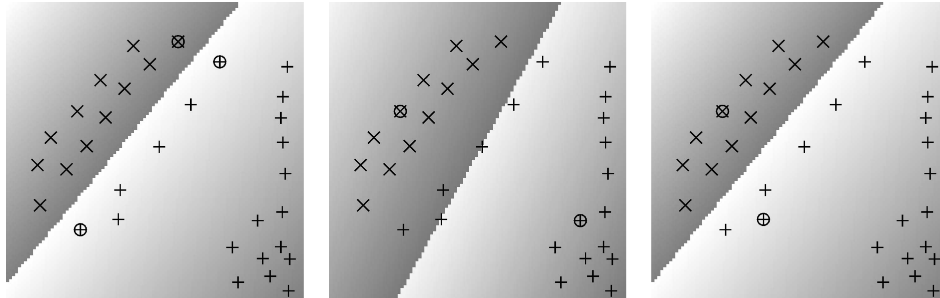


Figure 7. Problem with least squares fit that leads KMP-mse (center) to misclassify points, but does not affect SVMs (left), and is successfully treated by KMP-tanh (right).

5.2. US postal service database

The main purpose of this experiment was to complement the results of Schölkopf et al. (1997) with those obtained using KMP-mse, which, as already mentioned, is equivalent to orthogonal least squares RBF (Chen et al., 1991).

In Schölkopf et al. (1997) the RBF centers were chosen by unsupervised *k-means* clustering, in what they referred to as "Classical RBF", and a gradient descent optimization procedure was used to train the kernel weights.

We repeated the experiment using KMP-mse (equivalent to OLS-RBF) to find the support centers, with the same Gaussian Kernel and the same training set (7300 patterns) and independent test set (2007 patterns) of preprocessed handwritten digits. Table 1 gives the number of errors obtained by the various algorithms on the tasks consisting of discriminating each digit versus all the others (see Schölkopf et al., 1997 for more details). No validation data was used to choose the number of bases (support vectors) for the KMP. Instead, we trained with N equal to the number of support vectors obtained with the SVM, and also with N equal to half that number, to see whether a sparser KMP model would still yield good results. As can be seen, results obtained with KMP are comparable to those obtained for SVMs, contrarily to the results obtained with k-means RBFs, and there is only a slight loss of performance when using as few as half the number of support vectors.

Table 1. USPS Results: number of errors on the test set (2007 patterns), when using the same number of support vectors as found by SVM (except last row which uses half #sv). Squared error KMP (same as OLS-RBF) appears to perform as well as SVM.

Digit class	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
#sv	274	104	377	361	334	388	236	235	342	263
SVM	16	8	25	19	29	23	14	12	25	16
k-means RBF	20	16	43	38	46	31	15	18	37	26
KMP (same #sv)	15	15	26	17	30	23	14	14	25	13
KMP (half #sv)	16	15	29	27	29	24	17	16	28	18

5.3. Benchmark datasets

We did some further experiments, on 5 well-known datasets from the the UCI machine learning databases, using Gaussian kernels of the form

$$K(x_1, x_2) = e^{-\frac{\|x_1 - x_2\|^2}{\sigma^2}}.$$

A first series of experiments used the machinery of the Delve (Rasmussen et al., 1996) system to assess performance on the Mushrooms dataset. Hyper-parameters (the σ of the kernel, the box-constraint parameter C for soft-margin SVM and the number of support points for KMP) were chosen automatically for each run using 10-fold cross-validation.

The results for varying sizes of the training set are summarized in Table 2. The p-values reported in the table are those computed automatically by the Delve system.[10]

For Wisconsin Breast Cancer, Sonar, Pima Indians Diabetes and Ionosphere, we used a slightly different procedure.

The σ of the Kernel was first fixed to a reasonable value for the given data set.[11]

Then we used the following procedure: the dataset was randomly split into three equal-sized subsets for training, validation and testing. SVM, KMP-mse and KMP-tanh were then trained on the training set while the validation set was used to choose the optimal box-constraint parameter C for SVMs,[12] and to do early stopping (decide on the number N of s.v.) for KMP. Finally the trained models were tested on the independent test set.

This procedure was repeated 50 times over 50 different random splits of the dataset into train/validation/test to estimate confidence measures (p-values were computed using the resampled t-test studied in (Nadeau & Bengio, 2000)). Table 3 reports the average error rate measured on the test sets, and the rounded average number of support vectors found by each algorithm.

As can be seen from these experiments, the error rates obtained are comparable, but the KMP versions appear to require much fewer support vectors than SVMs. On these datasets, however (contrary to what we saw previously on 2D artificial data), KMP-tanh did not seem to give any significant improvement over KMP-mse. Even in other experiments where we added label noise, KMP-tanh didn't seem to improve generalization performance.[13]

Table 2. Results obtained on the mushrooms data set with the Delve system. KMP requires less support vectors, while none of the differences in error rates are significant.

Size of train	KMP error	SVM error	p-value (t-test)	KMP #s.v.	SVM #s.v.
64	6.28%	4.54%	0.24	17	63
128	2.51%	2.61%	0.82	28	105
256	1.09%	1.14%	0.81	41	244
512	0.20%	0.30%	0.35	70	443
1024	0.05%	0.07%	0.39	127	483

Table 3. Results on 4 UCI datasets. Again, error rates are not significantly different (values in parentheses are the p-values for the difference with SVMs), but KMPs require much fewer support vectors.

Dataset	SVM error	KMP-MSE error	KMP-TANH error	SVM #s.v.	KMP-MSE #s.v.	KMP-TANH #s.v.
Wisc. cancer	3.41%	3.40% (0.49)	3.49% (0.45)	42	7	21
Sonar	20.6%	21.0% (0.45)	26.6% (0.16)	46	39	14
Pima Indians	24.1%	23.9% (0.44)	24.0% (0.49)	146	7	27
Ionosphere	6.51%	6.87% (0.41)	6.85% (0.40)	68	50	41

6. Conclusion

We have shown how Matching Pursuit provides an interesting and flexible framework to build and study alternative kernel-based methods, how it can be extended to use arbitrary differentiable loss functions, and how it relates to SVMs, RBF training procedures, and boosting methods.

We have also provided experimental evidence that such greedy constructive algorithms can perform as well as SVMs, while allowing a better control of the sparsity of the solution, and thus often lead to solutions with far fewer support vectors.

It should also be mentioned that the use of a dictionary gives a lot of flexibility, as it can be extended in a direct and straightforward manner, allowing for instance, to mix several kernel shapes to choose from (similar to the SVM extension proposed by Weston et al., 1999), or to include other non-kernel functions based on prior knowledge (similar to the work of Smola Friess & Schölkopf, 1999 on semi-parametric SVMs). This is a promising avenue for further research.

In addition to the computational advantages brought by the sparsity of the models obtained with the kernel matching pursuit algorithms, one might suspect that generalization error also depends (monotonically) on the number of support vectors (other things being equal). This was observed empirically in our experiments, but future work should attempt to obtain generalization error bounds in terms of the number of support vectors. Note that leave-one-out SVM bounds (Vapnik, 1995; Vapnik, 1998) cannot be used here because the α_i coefficients depend on all the examples, not only a subset (the support vectors). Sparsity has been successfully exploited to obtain bounds for other SVM-like models (Littlestone & Warmuth, 1986; Floyd & Warmuth, 1995; Graepel, Herbrich, & Shaw-Taylor, 2000), in particular the kernel perceptron, again taking advantage of the dependence on a subset of the examples. A related direction to pursue might be to take advantage of the data-dependent structural risk minimization results of Shawe-Taylor et al. (1998).

Notes

1. Or a more computationally intensive cross-validation technique if the data is scarce.
2. In our implementation, we used a slightly modified version of this approach, described in the *pre-fitting* algorithm below.

3. Whose good generalization abilities are believed to be due to margin-maximization.
4. 0.65 is approximately the point of maximum second derivative of the tanh, and was advocated by LeCun et al. (1998) as a target value for neural networks, to avoid saturating the output units while taking advantage of the non-linearity for improving discrimination of neural networks.
5. The algorithms generalized to arbitrary loss functions can be much more computationally intensive, as they imply a non-quadratic optimization step.
6. It is possible to find artificial pathological cases where Matching Pursuit breaks down, but this doesn't seem to be a problem for real-world problems, especially when using the back-fitting or pre-fitting improvements of the original algorithm.
7. A comparison with regression-SVMs should also prove very interesting, but the question of how to compare two regression algorithms that do not optimize the same loss (squared loss for KMP, versus ϵ-insensitive loss for SVMs) first needs to be addressed.
8. We tried several frequencies at which to do full back-fitting, but it did not seem to have a strong impact, as long as it was done often enough.
9. However in a much more computationally intensive fashion.
10. For each size, the delve system did its estimations based on 8 disjoint training sets of the given size and 8 disjoint test sets of size 503, except for 1024, in which case it used 4 disjoint training sets of size 1024 and 4 test sets of size 1007.
11. These were chosen by trial and error using SVMs with a validation set and several values of C, and keeping what seemed the best σ, thus this choice was made at the advantage of SVMs (although they did not seem too sensitive to it) rather than KMP. The values used were: 4.0 for Wisconsin Breast Cancer, 6.0 for Pima Indians Diabetes, 2.0 for Ionosphere and Sonar.
12. Values of 0.02, 0.05, 0.07, 0.1, 0.5, 1, 2, 3, 5, 10, 20, 100 were tried for C.
13. We do not give a detailed account of these experiments here, as their primary intent was to show that the tanh error function could have an advantage over squared error in presence of label noise, but the results were inconclusive.

References

Aizerman, M., Braverman, E., & Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control, 25*, 821–837.

Boser, B., Guyon, I., & Vapnik, V. (1992). An algorithm for optimal margin classifiers. In: *Fifth Annual Workshop on Computational Learning Theory* (pp. 144–152). Pittsburgh.

Chen, S. (1995). Basis Pursuit. Ph.D. Thesis, Department of Statistics, Stanford University.

Chen, S., Cowan, F., & Grant, P. (1991). Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks, 2:2*, 302–309.

Davis, G., Mallat, S., & Zhang, Z. (1994). Adaptive time-frequency decompositions. *Optical Engineering, 33:7*, 2183–2191.

Floyd, S., & Warmuth, M. (1995). Sample compression, learnability, and the Vapnik-Chervonenkis dimension. *Machine Learning, 21:3*, 269–304.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of Thirteenth International Conference* (pp. 148–156).

Freund, Y., & Schapire, R. E. (1998). Large margin classification using the perceptron algorithm. In *Proc. 11th Annu. Conf. on Comput. Learning Theory* (pp. 209–217). New York, NY: ACM Press.

Friedman, J. (1999). Greedy function approximation: A gradient boosting machine. IMS 1999 Reitz Lecture, February 24, 1999, Department of Statistics, Stanford University.

Friedman, J., Hastie, T., & Tibshirani, R. (1998). Additive logistic regression: A statistical view of boosting. Technical Report, Aug. 1998, Department of Statistics, Stanford University.

Gallant, S. (1986). Optimal linear discriminants. In *Eighth International Conference on Pattern Recognition*. Paris, 1986 (pp. 849–852). New York: IEEE.

Au: pls. update

Graepel, T., Herbrich, R., & Shawe-Taylor, J. (in press). Generalization error bounds for sparse linear classifiers. In *Thirteenth Annual Conference on Computational Learning Theory, 2000*, Morgan Kaufmann.

Gunn, S., & Kandola, J. (2001). Structural modelling with sparse kernels. *Machine Learning* special issue on New Methods for Model Combination and Model Selection.

LeCun, Y., Bottou, L., Orr, G., & Müller, K.-R. (1998). Efficient backprop. In G. Orr & K.-R. Müller (Eds.), *Neural networks: Tricks of the trade* (pp. 9–50). Berlin: Springer.

Littlestone, N., & Warmuth, M. (1986). Relating data compression and learnability. Unpublished manuscript. University of California Santa Cruz. An extended version can be found in *Machine Learning, 21:3*, 269–304.

Mallat, S., & Zhang, Z. (1993). Matching pursuit with time-frequency dictionaries. *IEEE Trans. Signal Proc., 41:12*, 3397–3415.

Mason, L., Baxter, J., Bartlett, P., & Frean, M. (2000). Boosting algorithms as gradient descent. In S. A. Solla, T. K. Leen, and K.-R. Müller (Eds.), *Advances in neural information processing systems* (Vol. 12) (pp. 512–518). MIT Press.

Nadeau, C., & Bengio, Y. (2000). Inference for the generalization Error. In S. A. Solla, T. K. Leen, and K.-R. Müller (Eds.), *Advances in neural information processing systems* (Vol. 12) (pp. 307–313). Cambridge, MA: MIT Press.

Pati, Y., Rezaiifar, R., & Krishnaprasad, P. (1993). Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of the 27th Annual Asilomar Conference on Signals, Systems, and Computers* (pp. 40–44).

Poggio, T., & Girosi, F. (1998). A sparse representation for function approximation. *Neural Computation, 10:6*, 1445–1454.

Rasmussen, C., Neal, R., Hinton, G., van Camp, D., Ghahramani, Z., Kustra, R., & Tibshirani, R. (1996). The DELVE manual. DELVE can be found at http://www.cs.toronto.edu/∼delve

Rosenblatt, F. (1957). The perceptron—a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, N.Y.

Schapire, R. E., Freund, Y., Bartlett, P., & Lee, W. S. (1998). Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics, 26:5*, 1651–1686.

Schölkopf, B., Sung, K., Burges, C., Girosi, F., Niyogi, P., Poggio, T., & Vapnik, V. (1997). Comparing support vector machines with Gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing, 45*, 2758–2765.

Shawe-Taylor, J., Bartlett, P., Williamson, R., & Anthony, M. (1998). Structural risk minimization over data-dependent hierarchies. *IEEE Transactions on Information Theory, 44:5*, 1926–1940.

Singer, Y. (2000). Leveraged vector machines. In S. A. Solla, T. K. Leen, & K.-R. Müller (Eds.), *Advances in neural information processing systems* (Vol. 12) (pp. 610–616). MIT Press.

Smola, A. J., & Bartlett, P. (2001). Sparse greedy gaussian process regression. In *Advances in neural information processing systems* (Vol. 13).

Smola, A. J., Friess, T., & Schölkopf, B. (1999). Semiparametric support vector and linear programming machines. In M. Kearns, S. Solla, & D. Cohn (Eds.), *Advances in neural information processing systems* (Vol. 11) (pp. 585–591). MIT Press.

Smola, A., & Schölkopf, B. (2000). Sparse greedy matrix approximation for machine learning. In P. Langley (Ed.), *International Conference on Machine Learning* (pp. 911–918). San Francisco: Morgan Kaufmann.

Tipping, M. (2000). The relevance vector machine. In S. A. Solla, T. K. Leen, & K.-R. Müller (Eds.), *Advances in neural information processing systems* (Vol. 12) (pp. 652–658). Cambridge, MA: MIT Press.

Vapnik, V. (1995). *The nature of statistical learning theory*. New York: Springer.

Vapnik, V. (1998). *Statistical learning theory*. New York: Wiley. Lecture Notes in Economics and Mathematical Systems, Vol. 454.

Weston, J., Gammerman, A., Stitson, M., Vapnik, V., Vovk, V., & Watkins, C. (1999). Density estimation using support vector machines. In B. Schölkopf, C. J. C. Burges, & A. J. Smola (Eds.), *Advances in kernel methods—support vector learning* (pp. 293–306). Cambridge, MA: MIT Press.

Received August 23, 2000
Revised January 12, 2001
Accepted January 12, 2001
Final manuscript February 20, 2001