Kernel Matching Pursuit

Pascal Vincent and Yoshua Bengio
Dept. IRO, Université de Montréal
C.P. 6128, Montreal, Qc, H3C 3J7, Canada
{vincentp,bengioy}@iro.umontreal.ca

Technical Report #1179
Département d'Informatique et Recherche Opérationnelle
Université de Montréal

August 28th, 2000

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 also derive MDL motivated generalization bounds for this type of algorithm, and compare them to related SVM (Support Vector Machine) bounds. 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 sparser models.

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 and Vapnik, 1992; Vapnik, 1995). Kernel-based learning algorithms represent the function $f(x)$ to be learnt 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 and 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 $d_i(x)$ of the form $K(x, x_i)$ 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) shows 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.

Sparsity of representation is an important issue, both for the computational efficiency of the resulting representation, and for its theoretical and practical influence on generalization performance (see (Graepel, Herbrich and Shawe-Taylor, 2000) and (Floyd and 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.

In section 5, we use previous theoretical work on the minimum description length principle to construct generalization error bounds for the proposed algorithm. Basically, the generalization error is bounded by the training error plus terms that grow with the fraction of support vectors. These bounds are compared with bounds obtained for Support Vector Machines.

Finally, in section 6, 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 functions in a Hilbert space $\mathcal{H}$, and we are interested in sparse approximations of f that are expansions of the form

$$\hat{f}_N = \sum_{n=1}^{N} \alpha_n g_n \tag{1}$$

where $(\alpha_1, \ldots, \alpha_N) \in \mathbb{R}^{\mathsf{N}}$ and $\{g_1, \ldots, g_N\} \subset \mathcal{D}$ are chosen to minimize the squared norm of the residue $\|R_N\|^2 = \|f - \hat{f}_N\|^2$.

We shall call the set $\{g_1, \ldots, g_N\}$ our *basis*, and N the number of *basis functions* in the expansion.

Notice that, in a typical machine-learning framework, all we have are noisy observations of the target function f at the data points $x_{1..l}$. So we sometimes abuse the notation, using f to actually mean $(y_1, \ldots, y_l)$. Also, throughout this article, for all practical purposes, during training, any function in $\mathcal{H}$ can be associated to an l dimensional vector that represents the function evaluated at the $x_{1..l}$ data points. We will make extensive use of this abuse of notation for convenience; in particular the notation $\langle g, h \rangle$ will be used to represent the dot product between the two l dimensional vectors associated with functions g and h, and $\|h\|$ is used to represent the L_2 norm of the vector associated to a function h. Only when using the learnt approximation on new *test data* do we use the dictionary functions as actual *functions*.

Now, finding the optimal basis $\{g_1, \ldots, g_N\}$ for a given number N of allowed basis functions is in general an NP-complete problem. So the matching pursuit algorithm proceeds in a greedy constructive, fashion:

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

Given $\hat{f}_n$ we build

$$\hat{f}_{n+1} = \hat{f}_n + \alpha_{n+1} g_{n+1}$$

by searching for $g_{n+1} \in \mathcal{D}$ and for $\alpha_{n+1} \in \mathbb{R}$ that minimize the squared norm of the residue, $\|R_{n+1}\|^2 = \|R_n - \alpha_{n+1} g_{n+1}\|^2$, i.e.

$$(g_{n+1}, \alpha_{n+1}) = \underset{(g \in \mathcal{D}, \alpha \in \mathbb{R})}{\arg\min} \| \underbrace{\left(\sum_{k=1}^{n} \alpha_k g_k \right)}_{\hat{f}_n} + \alpha g - f \|^2 \tag{2}$$

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: residue vector R and dictionary matrix D

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

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 $\hat{f}_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 1: *Basic Matching Pursuit Algorithm*

The g_{n+1} that minimizes this expression is the one that maximizes $\left| \frac{\langle g_{n+1}, R_n \rangle}{\|g_{n+1}\|} \right|$ and the corresponding α_{n+1} is $\alpha_{n+1} = \frac{\langle g_{n+1}, R_n \rangle}{\|g_{n+1}\|^2}$

We have not yet specified how to choose N (i.e. when to stop). In the signal processing literature the algorithm is usually stopped when the *reconstruction error* ($\|R\|^2$) 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 can be seen as the primary capacity-control parameter of the algorithm. In section 5, we show that the generalization error of matching pursuit algorithms can be directly linked to the ratio $\frac{N}{l}$ (l is the number of training examples).

The pseudo-code for the corresponding algorithm is given in figure 1 (there are slight differences in the notation, in particular g_n in the above explanations corresponds to vector $D(.,\gamma_n)$ in the more detailed pseudo-code).

2.2 Matching Pursuit with backfitting

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

[1] or a more computationally intensive cross-validation technique if the data is scarce.

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 and Krishnaprasad, 1993; Davis, Mallat and Zhang, 1994):

While still choosing g_{n+1} as previously (equation 2), we recompute the optimal set of coefficients $\alpha_{1..n+1}$ at each step instead of only the last α_{n+1}:

$$\alpha_{1..n+1}^{(n+1)} = \underset{(\alpha_{1..n+1} \in \mathbb{R}^{n+1})}{\arg\min} \left\| \left(\sum_{k=1}^{n+1} \alpha_k g_k \right) - f \right\|^2 \tag{3}$$

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 $\mathcal{H}$ spanned by the basis $(g_1, \ldots, g_n)$ and let $\mathcal{B}_n^\perp = \mathcal{H} - \mathcal{B}_n$ be its orthogonal complement. Let $P_{\mathcal{B}_n}$ and $P_{\mathcal{B}_n^\perp}$ denote the projection operators on these subspaces.

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

Ideally, we want the residue R_n to be as small as possible, so given the basis at step n, we want $\hat{f}_n = P_{\mathcal{B}_n} f$ and $R_n = P_{\mathcal{B}_n^\perp} f$. This is what (3) insures.

But whenever we append the next $\alpha_{n+1} g_{n+1}$ found by (2) to the expansion, we actually add its two orthogonal components:

- $P_{\mathcal{B}_n^\perp} \alpha_{n+1} g_{n+1}$ contributes to reducing the norm of the residue.
- $P_{\mathcal{B}_n} \alpha_{n+1} g_{n+1}$ 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.

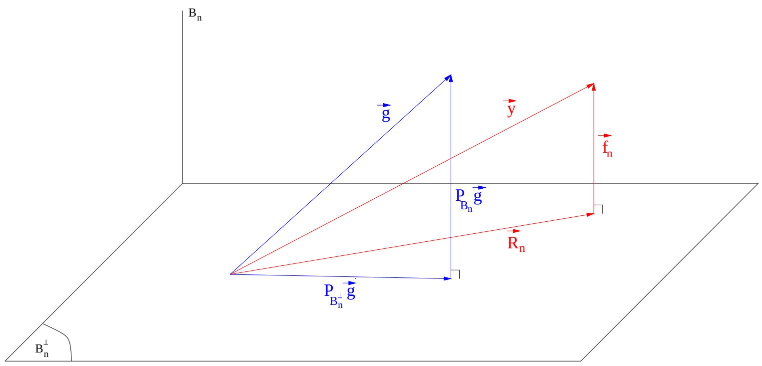

Figure 2: *Geometrical interpretation of Matching Pursuit and backprojection*

(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] In our implementation, we used a slightly modified version of this approach, described in the *prefitting* algorithm below.

2.3 Matching Pursuit with prefitting

With *backfitting*, 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 (2) and *only then* optimize (3), we might be picking a dictionary function other than the one that would give the best fit.

Instead, it is possible to directly optimize

$$\left(g_{n+1}, \alpha_{1..n+1}^{(n+1)}\right) = \underset{(g \in \mathcal{D}, \alpha_{1..n+1} \in \mathbb{R}^{n+1})}{\arg\min} \left\| \left(\sum_{k=1}^{n} \alpha_k g_k \right) + \alpha_{n+1} g - f \right\|^2 \quad (4)$$

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

This can be achieved almost as efficiently as *backfitting*. 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 $\mathcal{B}_n = span(g_1, \ldots, g_n)$. We maintain at each step a representation of each dictionary vector d as the sum of two orthogonal components:

- component $d_{\mathcal{B}_n} = P_{\mathcal{B}_n} d$ lies in the space $\mathcal{B}_n$ spanned by the current basis and is expressed as a linear combination of current basis vectors (it's a n dimensional vector).
- component $d_{\mathcal{B}_n^{\perp}} = P_{\mathcal{B}_n^{\perp}} d$ 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 y, namely its decomposition into the current expansion $\hat{f}_n \in \mathcal{B}_n$ plus the orthogonal residue $R_n \in \mathcal{B}_n^{\perp}$.

Prefitting 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 $g_{\mathcal{B}_n^{\perp}}$ is most collinear with $R_n \in \mathcal{B}_n^{\perp}$. This procedure requires, at every step, only two passes through the dictionary (searching 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.

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 (equation 2).
- **backfitting version:** We find the optimal g_n while keeping all coefficients fixed (equation 2). Then we find the optimal set of coefficients $\alpha_{1..n}^{(n)}$ for the new basis (equation 3).
- **prefitting version:** We find at the same time the optimal g_n and the optimal set of coefficients $\alpha_{1..n}^{(n)}$ (equation 4).

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: residue vector R and dictionary matrix component $D_{\mathcal{B}^\perp}$ and $D_{\mathcal{B}}$

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

$D_{\mathcal{B}}$ 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):

- $\gamma_n \leftarrow \arg\max_{k=1..m} \left| \dfrac{\langle D_{\mathcal{B}^\perp}(.,k), R\rangle}{\|D_{\mathcal{B}^\perp}(.,k)\|} \right|$
- $\alpha_n \leftarrow \dfrac{\langle D_{\mathcal{B}^\perp}(.,\gamma_n), R\rangle}{\|D_{\mathcal{B}^\perp}(.,\gamma_n)\|^2}$
- 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 $\hat{f}_N(x) = \sum_{n=1}^{N} \alpha_n d_{\gamma_n}(x)$

Figure 3: *Matching Pursuit with prefitting*

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

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 (Friedman, 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 $L(y_i, \hat{f}_n(x_i))$ that computes the cost of predicting a value of $\hat{f}_n(x_i)$ when the true target was y_i, we use an alternative residue $\tilde{R}_n$ rather than the usual $R_n = y - \hat{f}_n$ 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:

$$\tilde{R}_n = \left(-\frac{\partial L\left(y_1, \hat{f}_n(x_1)\right)}{\partial \hat{f}_n(x_1)}, \ldots, -\frac{\partial L\left(y_l, \hat{f}_n(x_l)\right)}{\partial \hat{f}_n(x_l)} \right) \tag{5}$$

i.e. g_{n+1} is chosen such that it is most collinear with this gradient:

$$g_{n+1} = \arg\max_{g \in \mathcal{D}} \left| \frac{\left\langle g_{n+1}, \tilde{R}_n \right\rangle}{\|g_{n+1}\|} \right| \tag{6}$$

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

$$\alpha_{n+1} = \arg\min_{\alpha \in \mathbf{R}} \sum_{i=1}^{l} L\left(f(x_i), \hat{f}_n(x_i) + \alpha g_{n+1}(x_i)\right) \tag{7}$$

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

It is also possible to do *backfitting*, 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):

$$\alpha_{1..n+1}^{(n+1)} = \arg\min_{\left(\alpha_{1..n+1} \in \mathbf{R}^{n+1}\right)} \sum_{i=1}^{l} L\left(f(x_i), \sum_{k=1}^{n+1} \alpha_k g_k\right) \tag{8}$$

But as this can be quite time-consuming (as 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 *prefitting* 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 8 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 $\mathrm{sign}(\hat{f}(x))$. 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

$$< \varphi(x_i), \varphi(x_j) >= K(x_i, x_j)$$

The SVM solution can be expressed in this feature space as

$$\hat{f}(x) =< w, \varphi(x) > +b \quad \text{where} \quad w = \sum_{x_i \in \mathcal{SV}} \alpha_i y_i \varphi(x_i)$$

Where $\mathcal{SV}$ is the set of support vectors and the solution is the one that minimizes

$$\sum_{i=1}^{l} [1 - y_i \hat{f}(x_i)]_+ + \frac{1}{C} \|w\|^2 \qquad (9)$$

Where C is the "box-constraint" parameter of SVMs, and the notation $[x]_+$ is to be understood as the function that gives $[x]_+ = x$ when $x > 0$ and 0 otherwise.

Let $m = y\hat{f}(x)$ the *individual margin* at point x. (9) 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 and Tibshirani, 1998) uses the negative binomial log-likelihood, $\log_2(1 + e^{-2m})$, whose shape is similar to a smoothed version of the

[3] whose good generalization abilities are believed to be due to margin-maximization.

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 backfitting: every p update steps
- a loss function L

INITIALIZE: current approximation $\hat{f}$ and dictionary matrix D

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

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

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

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

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

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

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

RESULT:

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

Figure 4: *Backfitting Matching Pursuit Algorithm with non-squared loss*

soft-margin SVM loss function $[1-m]_+$, and Doom II (Mason et al., 2000) 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:
  $(\tanh(\hat{f}(x)) - 0.65y)^2 = (0.65 - \tanh(m))^2$

Both are illustrated on figure 5 (right). 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 and SVM algorithms impose constraints on the weights, here denoted α's).

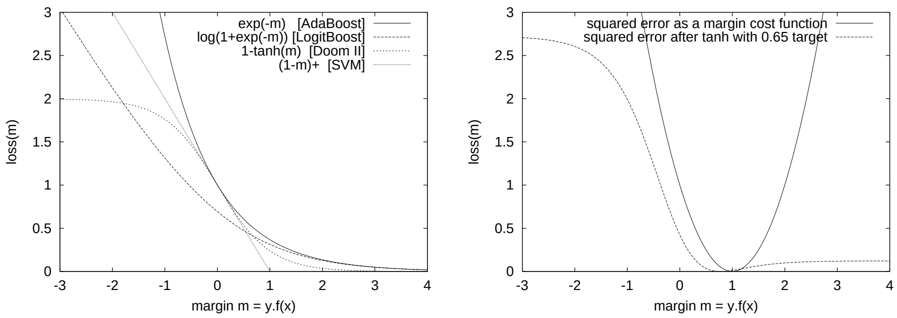

Figure 5: *Boosting and SVM margin loss functions (left) vs. traditional loss functions (right) viewed as functions of the margin. Interestingly the last-born of the margin motivated loss functions (used in Doom II) is similar to the traditional squared error after* tanh.

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

$$\hat{f}_N(x) = b + \sum_{n=1}^{N} \alpha_n K(x, x_{\gamma_n}) \tag{10}$$

where the $\gamma_{1..N}$ are the indexes 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[4], the complexity of all three variations of KMP (basic, backfitting and prefitting) 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 assymetrical, 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.
- 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).
- 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 SVMs.

4.2 Similarities and differences with SVMs

The functional form (10) is very similar to the one obtained with the *Support Vector Machine* (SVM) algorithm (Boser, Guyon and 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 6).

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]The algorithms generalized to arbitrary loss functions can be much more computationally intensive, as they imply a non-linear optimization step.

4.3 Link with Radial Basis Functions

Squared-error KMP with a Gaussian kernel and prefitting appears to be identical to a particular Radial Basis Functions training algorithm called *Orthogonal Least Squares RBF* (Chen, Cowan and 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 6) 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 and Schapire, 1996; Friedman, Hastie and 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.

5 Bounds on generalization error

The results of Vapnik on the Minimum Description Length (Vapnik, 1995; Vapnik, 1998) provide a possible framework for establishing bounds on expected generalization error for KMP algorithms. One can also simply use the results on the generalization error obtained when the number of possible functions is a finite number M, (and the capacity is therefore bounded by $\log M$). We will show that, essentially, the bound depends *linearly on the number of support vectors* and *logarithmically on the total number of training examples*.

Vapnik's result (theorem 4.3, (Vapnik, 1995)) states that the expected generalization error rate, E_{gen}, for binary classification, when training with l examples, is less than $2\mathcal{C}\log(2) - 2\log(\eta)/l$ with probability greater than $1 - \eta$, where $\mathcal{C}$ is the *compression rate*: the number of bits to transfer the compressed conditional value of the training target classes (given the training input points) divided by the number of bits required to transmit them without compression, i.e., l. When there are training errors, we can incorporate them into the "compressed message" by sending the identity (and the labels, in the multiclass case) of the wrongly labeled examples.

The compression is due to the representation learned by the training algorithm. A "good" representation is one that requires few bits to represent the learned function, while keeping the training error low. This assumes that the number of possible functions is finite (which we will obtain by quantizing the α coefficients). To obtain compression, we take advantage of the sparse representation of the learned function in terms of only N "support points".

To obtain a rough bound we will encode the target outputs using three sets of bits, corresponding to three terms for $\mathcal{C}$:

1. The first one is due to the classification errors: we have to send the identity and the correct class of the training errors. If the number of errors is $e = lE_{emp}$, that will cost $\log_2 \binom{l}{e}$ bits. In the case where the number

of classes is $N_c > 2$, there is an increase in the number of bits by a factor $\log_2(N_c - 1)$, but there is a similar increase in the numerator of $\mathcal{C}$ (to encode the correct classes of all the training examples).

2. The second term is required to encode the identity of the support points: to choose N among l examples requires $\log_2 \binom{l}{N}$ bits.
3. The third term is to encode the quantized weights α_k associated with each support point, which will cost Np bits, where p is the number of bits of precision to quantize the weights, and it can be chosen as the smallest number that allows to obtain with the discretized α's the same classes on the training set as the undiscretized α's.

To summarize, for KMP, we have, for e training errors and N support vectors out of l examples, with probability greater than $1 - \eta$ (over the choice of training set),

$$E_{gen} < 2\frac{\log \binom{l}{e} + \log \binom{l}{N} + (Np \log 2)}{l} - 2\log(\eta)/l \tag{11}$$

Note that $\binom{l}{n}$ is poorly bounded by $n \log_2 l$, in which the e/l and N/l ratios become apparent, but where a too large $\log l$ factor appears.

Slightly tighter bounds can be obtained using the result (Vapnik, 1995; Vapnik, 1998) for learning by choosing one function among $M < \infty$ functions: with probability at least $1 - \eta$,

$$E_{gen} \leq E_{emp} + \frac{\log M - \log \eta}{l} \left(1 + \sqrt{1 + \frac{2E_{emp} l}{\log M - \log \eta}}\right). \tag{12}$$

Using the same quantization of the α's (with precision p), one obtains with $\log M = \log(\binom{l}{N} 2^{Np})$,

$$E_{gen} < E_{emp} + \frac{\log \binom{l}{N} + Np \log 2 - \log \eta}{l} \left(1 + \sqrt{1 + \frac{2E_{emp} l}{\log \binom{l}{N} + Np \log 2 - \log \eta}}\right). \tag{13}$$

In contrast, one can obtain an expectation bound (Vapnik, 1995) for SVMs that is $E[E_{gen}] < E[E_{emp}] + E[\frac{N}{l}]$, where E is the expectation over training sets (note that for SVMs, N is random because it depends on the training set). Note that the probability bounds can be readily converted into expectation bounds. For example, in the case of the MDL bound (eq. 11), one obtains that in expectation,

$$E_{gen} < 2\frac{E[\log \binom{l}{e}] + \log \binom{l}{N} + (Np \log 2) + 1}{l}.$$

To see the role of the ratio $\frac{N}{l}$ in the above, one can note that $\frac{\log \binom{l}{N}}{l} < \frac{N \log l}{l}$ (but keep in mind that this is a rather poor bound).

Note that several related compression bounds have been studied, e.g. (Littlestone and Warmuth, 1986; Floyd and Warmuth, 1995; Graepel, Herbrich and Shawe-Taylor, 2000). The results of (Graepel, Herbrich and Shawe-Taylor, 2000) are meant for maximum margin classifiers and draw interesting connections between sparsity and maximum margin. The results in (Littlestone and Warmuth, 1986; Floyd and Warmuth, 1995) are very general (and very much linked to the above discussion), but they apply to classifiers which can be specified using only a subset of the training examples. However, note that the case of Matching Pursuit, the classifier requires not only the support vectors but also the weights α_i, which in general depend on the whole training set.

6 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 unless*margin loss function* as we discussed earlier in section 3.2).
- Unless otherwise specified, we used the *prefitting matching pursuit* algorithm of figure 3 to train least-squares KMP.
- To train *KMP-tanh* we always used the *backfitting matching pursuit with non-squared loss* algorithm of figure 4 with a conjugate gradient optimizer to optimize the $\alpha_{1..n}$[5].

6.1 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, backfitting and prefitting) and a hard-margin SVM, when using the same Gaussian kernel.

We fixed the number N of support points for the prefitting and backfitting 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 separate the data points. This shows that the backfitting and prefitting versions are a useful improvement, while the basic algorithm appears to be a bad choice if we want sparse solutions.
- The backfitting and prefitting KMP algorithms are able to find a reasonable solution (the solution found by prefitting 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[6] 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.

[5]We tried several frequencies at which to do full-backfitting, but it did not seem to have a real impact, as long as it was done often enough.

[6]however in a much more computationally intensive fashion.

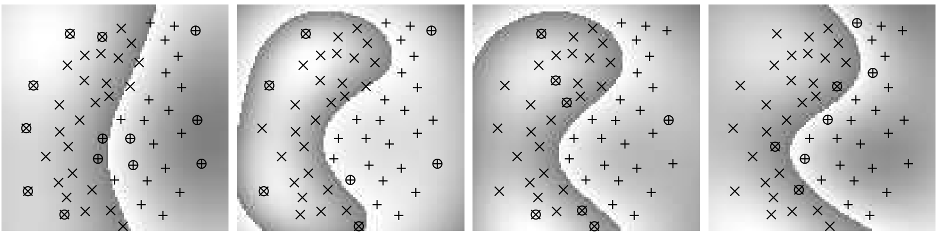

Figure 6: *From left to right: 100 iterations of basic KMP, 7 iterations of KMP backfitting, 7 iterations of KMP prefitting, and SVM. Classes are + and ×. Support vectors are circled. Prefitting KMP and SVM appear to find equally reasonable solutions, though using different support vectors. Only SVM chooses its support vectors close to the decision surface. backfitting 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.*

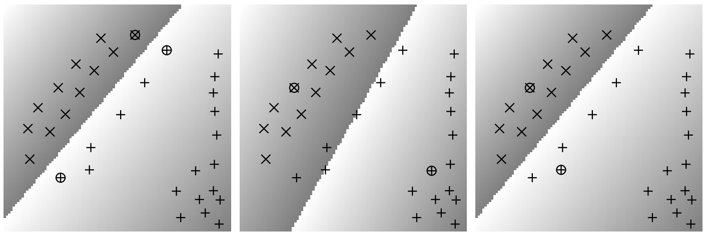

Figure 7: *Problem with least squares fit that leads KMP-mse (center) to misclassify points, but does not affect SVMs (left), and is successfully treated by KMP-tanh (right).*

6.2 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, Cowan and Grant, 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

6.3 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[7].

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

[7]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.

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[8].

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[9], and to do early stopping (decide on the number N of s.v.) for KMP. And 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 (Nadeau and 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 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[10].

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

[8] 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.

[9] Values of 0.02, 0.05, 0.07, 0.1, 0.5, 1, 2, 3, 5, 10, 20, 100 were tried for C.

[10] 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.

7 Conclusion

We have shown how Matching Pursuit provides a 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 additional flexibility, as it can be used, for instance, to mix several kernel shapes to choose from, similar to what has been done in (Weston et al., 1999), or to include other non-kernel functions based on prior knowledge, which opens the way to further research.

References

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

Chen, S. (1995). *Basis Pursuit*. PhD thesis, Department of Statistics, Stanford University.

Chen, S., Cowan, F., and 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., and Zhang, Z. (1994). Adaptive time-frequency decompositions. *Optical Engineering*, 33(7):2183–2191.

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

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

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

Friedman, J., Hastie, T., and Tibshirani, R. (1998). Additive logistic regression: a statistical view of boosting. Technical report, August 1998, Department of Statistics, Stanford University.

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

Littlestone, N. and Warmuth, M. (1986). Relating data compression and learnability. Unpublished manuscript. University of California Santa Cruz. An extended version can be found in (Floyd and Warmuth 95).

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

Mason, L., Baxter, J., Bartlett, P., and Frean, M. (2000). Boosting algorithms as gradient descent. In Solla, S. A., Leen, T. K., and Mller, K.-R., editors, *Advances in Neural Information Processing Systems*, volume 12, pages 512–518. MIT Press.

Nadeau, C. and Bengio, Y. (2000). Inference for the generalization error. In Solla, S. A., Leen, T. K., and Mller, K.-R., editors, *Advances in Neural Information Processing Systems*, volume 12, pages 307–313. MIT Press.

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

Poggio, T. and 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., and Tibshirani, R. (1996). The DELVE manual. DELVE can be found at http://www.cs.toronto.edu/~delve.

Schapire, R. E., Freund, Y., Bartlett, P., and 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., and Vapnik, V. (1997). Comparing support vector machines with gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing*, 45:2758–2765.

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

Tipping, M. (2000). The relevance vector machine. In Solla, S. A., Leen, T. K., and Mller, K.-R., editors, *Advances in Neural Information Processing Systems*, volume 12, pages 652–658. MIT Press.

Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. Springer, New York.

Vapnik, V. (1998). *Statistical Learning Theory*. Wiley, Lecture Notes in Economics and Mathematical Systems, volume 454.

Weston, J., Gammerman, A., Stitson, M., Vapnik, V., Vovk, V., and Watkins, C. (1999). Density estimation using support vector machines. In Schölkopf, B., Burges, C. J. C., and Smola, A. J., editors, *Advances in Kernel Methods — Support Vector Learning*, pages 293–306, Cambridge, MA. MIT Press.