RECENT ADVANCES in PREDICTIVE (MACHINE) LEARNING

Jerome H. Friedman
Deptartment of Statisitcs and
Stanford Linear Accelerator Center,
Stanford University, Stanford, CA 94305
(jhf@stanford.edu)

Prediction involves estimating the unknown value of an attribute of a system under study given the values of other measured attributes. In prediction (machine) learning the prediction rule is derived from data consisting of previously solved cases. Most methods for predictive learning were originated many years ago at the dawn of the computer age. Recently two new techniques have emerged that have revitalized the field. These are support vector machines and boosted decision trees. This paper provides an introduction to these two new methods tracing their respective ancestral roots to standard kernel methods and ordinary decision trees.

I. INTRODUCTION

The predictive or machine learning problem is easy to state if difficult to solve in general. Given a set of measured values of attributes/characteristics/properties on a object (observation) $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ (often called "variables") the goal is to predict (estimate) the unknown value of another attribute y. The quantity y is called the "output" or "response" variable, and $\mathbf{x} = \{x_1, \cdots, x_n\}$ are referred to as the "input" or "predictor" variables. The prediction takes the form of function

$$\hat{y} = F(x_1, x_2, \cdots, x_n) = F(\mathbf{x})$$

that maps a point $\mathbf{x}$ in the space of all joint values of the predictor variables, to a point $\hat{y}$ in the space of response values. The goal is to produce a "good" predictive $F(\mathbf{x})$. This requires a definition for the quality, or lack of quality, of any particular $F(\mathbf{x})$. The most commonly used measure of lack of quality is prediction "risk". One defines a "loss" criterion that reflects the cost of mistakes: $L(y, \hat{y})$ is the loss or cost of predicting a value $\hat{y}$ for the response when its true value is y. The prediction risk is defined as the average loss over all predictions

$$R(F) = E_{y\mathbf{x}} L(y, F(\mathbf{x})) \tag{1}$$

where the average (expected value) is over the joint (population) distribution of all of the variables $(y, \mathbf{x})$ which is represented by a probability density function $p(y, \mathbf{x})$. Thus, the goal is to find a mapping function $F(\mathbf{x})$ with low predictive risk.

Given a function f of elements w in some set, the choice of w that gives the smallest value of $f(w)$ is called $\arg\min_w f(w)$. This definition applies to all types of sets including numbers, vectors, colors, or functions. In terms of this notation the optimal predictor with lowest predictive risk (called the "target function") is given by

$$F^* = \arg\min_F R(F). \tag{2}$$

Given joint values for the input variables $\mathbf{x}$, the optimal prediction for the output variable is $\hat{y} = F^*(\mathbf{x})$.

When the response takes on numeric values $y \in R^1$, the learning problem is called "regression" and commonly used loss functions include absolute error $L(y, F) = |y - F|$, and even more commonly squared–error $L(y, F) = (y - F)^2$ because algorithms for minimization of the corresponding risk tend to be much simpler. In the "classification" problem the response takes on a discrete set of K unorderable categorical values (names or class labels) $y, F \in \{c_1, \cdots, c_K\}$ and the loss criterion $L_{y,F}$ becomes a discrete $K \times K$ matrix.

There are a variety of ways one can go about trying to find a good predicting function $F(\mathbf{x})$. One might seek the opinions of domain experts, formally codified in the "expert systems" approach of artificial intelligence. In predictive or machine learning one uses data. A "training" data base

$$D = \{y_i, x_{i1}, x_{i2}, \cdots, x_{in}\}_1^N = \{y_i, \mathbf{x}_i\}_1^N \tag{3}$$

of N previously solved cases is presumed to exist for which the values of all variables (response and predictors) have been jointly measured. A "learning" procedure is applied to these data in order to extract (estimate) a good predicting function $F(\mathbf{x})$. There are a great many commonly used learning procedures. These include linear/logistic regression, neural networks, kernel methods, decision trees, multivariate splines (MARS), etc. For descriptions of a large number of such learning procedures see Hastie, Tibshirani and Friedman 2001.

Most machine learning procedures have been around for a long time and most research in the field has concentrated on producing refinements to these long standing methods. However, in the past several years there has been a revolution in the field inspired by the introduction of two new approaches: the extension of kernel methods to support vector machines (Vapnik 1995), and the extension of decision trees by

boosting (Freund and Schapire 1996, Friedman 2001). It is the purpose of this paper to provide an introduction to these new developments. First the classic kernel and decision tree methods are introduced. Then the extension of kernels to support vector machines is described, followed by a description of applying boosting to extend decision tree methods. Finally, similarities and differences between these two approaches will be discussed.

Although arguably the most influential recent developments, support vector machines and boosting are not the only important advances in machine learning in the past several years. Owing to space limitations these are the ones discussed here. There have been other important developments that have considerably advanced the field as well. These include (but are not limited to) the bagging and random forest techniques of Breiman 1996 and 2001 that are somewhat related to boosting, and the reproducing kernel Hilbert space methods of Wahba 1990 that share similarities with support vector machines. It is hoped that this article will inspire the reader to investigate these as well as other machine learning procedures.

II. KERNEL METHODS

Kernel methods for predictive learning were introduced by Nadaraya (1964) and Watson (1964). Given the training data (3), the response estimate $\hat{y}$ for a set of joint values $\mathbf{x}$ is taken to be a weighted average of the training responses $\{y_i\}_1^N$:

$$\hat{y} = F_N(\mathbf{x}) = \sum_{i=1}^{N} y_i \, K(\mathbf{x}, \mathbf{x}_i) \Big/ \sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i). \quad (4)$$

The weight $K(\mathbf{x}, \mathbf{x}_i)$ assigned to each response value y_i depends on its location $\mathbf{x}_i$ in the predictor variable space and the location $\mathbf{x}$ where the prediction is to be made. The function $K(\mathbf{x}, \mathbf{x}')$ defining the respective weights is called the "kernel function", and it defines the kernel method. Often the form of the kernel function is taken to be

$$K(\mathbf{x}, \mathbf{x}') = g(d(\mathbf{x}, \mathbf{x}')/\sigma) \quad (5)$$

where $d(\mathbf{x}, \mathbf{x}')$ is a defined "distance" between $\mathbf{x}$ and $\mathbf{x}'$, σ is a scale ("smoothing") parameter, and $g(z)$ is a (usually monotone) decreasing function with increasing z; often $g(z) = \exp(-z^2/2)$. Using this kernel (5), the estimate $\hat{y}$ (4) is a weighted average of $\{y_i\}_1^N$, with more weight given to observations i for which $d(\mathbf{x}, \mathbf{x}_i)$ is small. The value of σ defines "small". The distance function $d(\mathbf{x}, \mathbf{x}')$ must be specified for each particular application.

Kernel methods have several advantages that make them potentially attractive. They represent a universal approximator; as the training sample size N becomes arbitrarily large, $N \to \infty$, the kernel estimate (4) (5) approaches the optimal predicting target function (2), $F_N(\mathbf{x}) \to F^*(\mathbf{x})$, provided the value chosen for the scale parameter σ as a function of N approaches zero, $\sigma(N) \to 0$, at a slower rate than $1/N$. This result holds for almost any distance function $d(\mathbf{x}, \mathbf{x}')$; only very mild restrictions (such as convexity) are required. Another advantage of kernel methods is that no training is required to build a model; the training data set *is* the model. Also, the procedure is conceptually quite simple and easily explained.

Kernel methods suffer from some disadvantages that have kept them from becoming highly used in practice, especially in data mining applications. Since there is no model, they provide no easily understood model summary. Thus, they cannot be easily interpreted. There is no way to discern how the function $F_N(\mathbf{x})$ (4) depends on the respective predictor variables $\mathbf{x}$. Kernel methods produce a "black–box" prediction machine. In order to make each prediction, the kernel method needs to examine the entire data base. This requires enough random access memory to store the entire data set, and the computation required to make each prediction is proportional to the training sample size N. For large data sets this is much slower than that for competing methods.

Perhaps the most serious limitation of kernel methods is statistical. For any *finite* N, performance (prediction accuracy) depends *critically* on the chosen distance function $d(\mathbf{x}, \mathbf{x}')$, especially for regression $y \in R^1$. When there are more than a few predictor variables, even the largest data sets produce a very sparse sampling in the corresponding n–dimensional predictor variable space. This is a consequence of the so called "curse–of–dimensionality" (Bellman 1962). In order for kernel methods to perform well, the distance function must be carefully matched to the (unknown) target function (2), and the procedure is not very robust to mismatches.

As an example, consider the often used Euclidean distance function

$$d(\mathbf{x}, \mathbf{x}') = \left[\sum_{j=1}^{n} (x_j - x'_j)^2 \right]^{1/2}. \quad (6)$$

If the target function $F^*(\mathbf{x})$ dominately depends on only a small subset of the predictor variables, then performance will be poor because the kernel function (5) (6) depends on all of the predictors with equal strength. If one happened to know *which* variables were the important ones, an appropriate kernel could be constructed. However, this knowledge is often not available. Such "kernel customizing" is a requirement with kernel methods, but it is difficult to do without considerable a priori knowledge concerning the problem at hand.

The performance of kernel methods tends to be fairly insensitive to the detailed choice of the function

$g(z)$ (5), but somewhat more sensitive to the value chosen for the smoothing parameter σ. A good value depends on the (usually unknown) smoothness properties of the target function $F^*(\mathbf{x})$, as well as the sample size N and the signal/noise ratio.

III. DECISION TREES

Decision trees were developed largely in response to the limitations of kernel methods. Detailed descriptions are contained in monographs by Brieman, Friedman, Olshen and Stone 1983, and by Quinlan 1992. The minimal description provided here is intended as an introduction sufficient for understanding what follows.

A decision tree partitions the space of all joint predictor variable values $\mathbf{x}$ into J–disjoint regions $\{R_j\}_1^J$. A response value $\hat{y}_j$ is assigned to each corresponding region R_j. For a given set of joint predictor values $\mathbf{x}$, the tree prediction $\hat{y} = T_J(\mathbf{x})$ assigns as the response estimate, the value assigned to the region containing $\mathbf{x}$

$$\mathbf{x} \in R_j \Rightarrow T_J(\mathbf{x}) = \hat{y}_j \,. \tag{7}$$

Given a set of regions, the optimal response values associated with each one are easily obtained, namely the value that minimizes prediction risk in that region

$$\hat{y}_j = \arg\min_{y'} E_y[L(y, y') \,|\, \mathbf{x} \in R_j]. \tag{8}$$

The difficult problem is to find a good set of regions $\{R_j\}_1^J$. There are a huge number of ways to partition the predictor variable space, the vast majority of which would provide poor predictive performance. In the context of decision trees, choice of a particular partition directly corresponds to choice of a distance function $d(\mathbf{x}, \mathbf{x}')$ and scale parameter σ in kernel methods. Unlike with kernel methods where this choice is the responsibility of the user, decision trees attempt to use the data to estimate a good partition.

Unfortunately, finding the optimal partition requires computation that grows exponentially with the number of regions J, so that this is only possible for very small values of J. All tree based methods use a greedy top–down recursive partitioning strategy to induce a good set of regions given the training data set (3). One starts with a single region covering the entire space of all joint predictor variable values. This is partitioned into two regions by choosing an optimal splitting predictor variable x_j and a corresponding optimal split point s. Points $\mathbf{x}$ for which $x_j \leq s$ are defined to be in the left daughter region, and those for which $x_j > s$ comprise the right daughter region. Each of these two daughter regions is then itself optimally partitioned into two daughters of its own in the same manner, and so on. This recursive partitioning continues until the observations within each region all have the same response value y. At this point a recursive recombination strategy ("tree pruning") is employed in which sibling regions are in turn merged in a bottom–up manner until the number of regions J^* that minimizes an estimate of future prediction risk is reached (see Breiman *et al* 1983, Ch. 3).

A. Decision tree properties

Decision trees are the most popular predictive learning method used in data mining. There are a number of reasons for this. As with kernel methods, decision trees represent a universal method. As the training data set becomes arbitrarily large, $N \to \infty$, tree based predictions (7) (8) approach those of the target function (2), $T_J(\mathbf{x}) \to F^*(\mathbf{x})$, provided the number of regions grows arbitrarily large, $J(N) \to \infty$, but at rate slower than N.

In contrast to kernel methods, decision trees do produce a model summary. It takes the form of a binary tree graph. The root node of the tree represents the entire predictor variable space, and the (first) split into its daughter regions. Edges connect the root to two descendent nodes below it, representing these two daughter regions and their respective splits, and so on. Each internal node of the tree represents an intermediate region and its optimal split, defined by a one of the predictor variables x_j and a split point s. The terminal nodes represent the final region set $\{R_j\}_1^J$ used for prediction (7). It is this binary tree graphic that is most responsible for the popularity of decision trees. No matter how high the dimensionality of the predictor variable space, or how many variables are actually used for prediction (splits), the entire model can be represented by this two–dimensional graphic, which can be plotted and then examined for interpretation. For examples of interpreting binary tree representations see Breiman *et al* 1983 and Hastie, Tibshirani and Friedman 2001.

Tree based models have other advantages as well that account for their popularity. Training (tree building) is relatively fast, scaling as $nN \log N$ with the number of variables n and training observations N. Subsequent prediction is extremely fast, scaling as $\log J$ with the number of regions J. The predictor variables need not all be numeric valued. Trees can seamlessly accommodate binary and categorical variables. They also have a very elegant way of dealing with missing variable values in both the training data and future observations to be predicted (see Breiman *et al* 1983, Ch. 5.3).

One property that sets tree based models apart from all other techniques is their invariance to monotone transformations of the predictor variables. Replacing any subset of the predictor variables $\{x_j\}$ by (possibly different) arbitrary strictly monotone func-

tions of them $\{ x_j \leftarrow m_j(x_j)\}$, gives rise to the same tree model. Thus, there is no issue of having to experiment with different possible transformations $m_j(x_j)$ for each individual predictor x_j, to try to find the best ones. This invariance provides immunity to the presence of extreme values "outliers" in the predictor variable space. It also provides invariance to changing the measurement scales of the predictor variables, something to which kernel methods can be very sensitive.

Another advantage of trees over kernel methods is fairly high resistance to irrelevant predictor variables. As discussed in Section II, the presence of many such irrelevant variables can highly degrade the performance of kernel methods based on generic kernels that involve all of the predictor variables such as (6). Since the recursive tree building algorithm estimates the optimal variable on which to split at each step, predictors unrelated to the response tend not to be chosen for splitting. This is a consequence of attempting to find a good partition based on the data. Also, trees have few tunable parameters so they can be used as an "off–the–shelf" procedure.

The principal limitation of tree based methods is that in situations not especially advantageous to them, their performance tends not to be competitive with other methods that might be used in those situations. One problem limiting accuracy is the piecewise–constant nature of the predicting model. The predictions $\hat{y}_j$ (8) are constant within each region R_j and sharply discontinuous across region boundaries. This is purely an artifact of the model, and target functions $F^*(\mathbf{x})$ (2) occurring in practice are not likely to share this property. Another problem with trees is instability. Changing the values of just a few observations can dramatically change the structure of the tree, and substantially change its predictions. This leads to high variance in potential predictions $T_J(\mathbf{x})$ at any particular prediction point $\mathbf{x}$ over different training samples (3) that might be drawn from the system under study. This is especially the case for large trees.

Finally, trees fragment the data. As the recursive splitting proceeds each daughter region contains fewer observations than its parent. At some point regions will contain too few observations and cannot be further split. Paths from the root to the terminal nodes tend to contain on average a relatively small fraction of all of the predictor variables that thereby define the region boundaries. Thus, each prediction involves only a relatively small number of predictor variables. If the target function is influenced by only a small number of (potentially different) variables in different local regions of the predictor variable space, then trees can produce accurate results. But, if the target function depends on a substantial fraction of the predictors everywhere in the space, trees will have problems.

IV. RECENT ADVANCES

Both kernel methods and decision trees have been around for a long time. Trees have seen active use, especially in data mining applications. The classic kernel approach has seen somewhat less use. As discussed above, both methodologies have (different) advantages and disadvantages. Recently, these two technologies have been completely revitalized in different ways by addressing different aspects of their corresponding weaknesses; support vector machines (Vapnik 1995) address the computational problems of kernel methods, and boosting (Freund and Schapire 1996, Friedman 2001) improves the accuracy of decision trees.

A. Support vector machines (SVM)

A principal goal of the SVM approach is to fix the computational problem of predicting with kernels (4). As discussed in Section II, in order to make a kernel prediction a pass over the entire training data base is required. For large data sets this can be too time consuming and it requires that the entire data base be stored in random access memory.

Support vector machines were introduced for the two–class classification problem. Here the response variable realizes only two values (class labels) which can be respectively encoded as

$$y = \begin{cases} +1 & \text{label = class 1} \\ -1 & \text{label = class 2} \end{cases} . \tag{9}$$

The average or expected value of y given a set of joint predictor variable values $\mathbf{x}$ is

$$E\left[y \,|\, \mathbf{x}\right] = 2 \cdot \Pr(y = +1 \,|\, \mathbf{x}) - 1. \tag{10}$$

Prediction error rate is minimized by predicting at $\mathbf{x}$ the class with the highest probability, so that the optimal prediction is given by

$$y^*(\mathbf{x}) = sign(E\left[y \,|\, \mathbf{x}\right]).$$

From (4) the kernel estimate of (10) based on the training data (3) is given by

$$\hat{E}\left[y \,|\, \mathbf{x}\right] = F_N(\mathbf{x}) = \sum_{i=1}^{N} y_i \, K(\mathbf{x}, \mathbf{x}_i) \Big/ \sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i) \tag{11}$$

and, assuming a strictly non negative kernel $K(\mathbf{x}, \mathbf{x}_i)$, the prediction estimate is

$$\hat{y}(\mathbf{x}) = sign(\hat{E}\left[y \,|\, \mathbf{x}\right]) = sign\left(\sum_{i=1}^{N} y_i \, K(\mathbf{x}, \mathbf{x}_i)\right). \tag{12}$$

Note that ignoring the denominator in (11) to obtain (12) removes information concerning the absolute value of $\Pr(y = +1 \,|\, \mathbf{x})$; only the estimated sign of (10) is retained for classification.

A support vector machine is a weighted kernel classifier

$$\hat{y}(\mathbf{x}) = sign\left(a_0 + \sum_{i=1}^{N} \alpha_i \, y_i \, K(\mathbf{x}, \mathbf{x}_i)\right). \qquad (13)$$

Each training observation $(y_i, \mathbf{x}_i)$ has an associated coefficient α_i additionally used with the kernel $K(\mathbf{x}, \mathbf{x}_i)$ to evaluate the weighted sum (13) comprising the kernel estimate $\hat{y}(\mathbf{x})$. The goal is to choose a set of coefficient values $\{\alpha_i\}_1^N$ so that many $\alpha_i = 0$ while still maintaining prediction accuracy. The observations associated with non zero valued coefficients $\{\mathbf{x}_i \,|\, \alpha_i \neq 0\}$ are called "support vectors". Clearly from (13) only the support vectors are required to do prediction. If the number of support vectors is a small fraction of the total number of observations computation required for prediction is thereby much reduced.

1. Kernel trick

In order to see how to accomplish this goal consider a different formulation. Suppose that instead of using the original measured variables $\mathbf{x} = (x_1, \cdots, x_n)$ as the basis for prediction, one constructs a very large number of (nonlinear) functions of them

$$\{z_k = h_k(\mathbf{x})\}_1^M \qquad (14)$$

for use in prediction. Here each $h_k(\mathbf{x})$ is a different function (transformation) of $\mathbf{x}$. For any given $\mathbf{x}$, $\mathbf{z} = \{z_k\}_1^M$ represents a point in a M–dimensional space where $M >> \dim(\mathbf{x}) = n$. Thus, the number of "variables" used for classification is dramatically expanded. The procedure constructs simple *linear* classifier in $\mathbf{z}$–space

$$\begin{aligned}\hat{y}(\mathbf{z}) &= sign\left(\beta_0 + \sum_{k=1}^{M} \beta_k \, z_k\right) \\ &= sign\left(\beta_0 + \sum_{k=1}^{M} \beta_k \, h_k\left(\mathbf{x}\right)\right).\end{aligned}$$

This is a highly *non*–linear classifier in $\mathbf{x}$–space owing to the nonlinearity of the derived transformations $\{h_k(\mathbf{x})\}_1^M$.

An important ingredient for calculating such a linear classifier is the inner product between the points representing two observations i and j

$$\begin{aligned}\mathbf{z}_i^T \mathbf{z}_j &= \sum_{k=1}^{M} z_{ik} z_{jk} \qquad (15)\\ &= \sum_{k=1}^{M} h_k\left(\mathbf{x}_i\right) h_k\left(\mathbf{x}_j\right) \\ &= H(\mathbf{x}_i, \mathbf{x}_j).\end{aligned}$$

This (highly nonlinear) function of the $\mathbf{x}$–variables, $H(\mathbf{x}_i, \mathbf{x}_j)$, defines the simple bilinear inner product $\mathbf{z}_i^T \mathbf{z}_j$ in $\mathbf{z}$–space.

Suppose for example, the derived variables (14) were taken to be all d–degree polynomials in the original predictor variables $\{x_j\}_1^n$. That is $z_k = x_{i_1(k)} x_{i_2(k)} \cdots x_{i_d(k)}$, with k labeling each of the $M = (n+1)^d$ possible sets of d integers, $0 \leq i_j(k) \leq n$, and with the added convention that $x_0 = 1$ even though x_0 is not really a component of the vector $\mathbf{x}$. In this case the number of derived variables is $M = (n+1)^d$, which is the order of computation for obtaining $\mathbf{z}_i^T \mathbf{z}_j$ directly from the $\mathbf{z}$ variables. However, using

$$\mathbf{z}_i^T \mathbf{z}_j = H(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j + 1)^d \qquad (16)$$

reduces the computation to order n, the much smaller number of originally measured variables. Thus, if for any particular set of derived variables (14), the function $H(\mathbf{x}_i, \mathbf{x}_j)$ that defines the corresponding inner products $\mathbf{z}_i^T \mathbf{z}_j$ in terms of the original $\mathbf{x}$–variables can be found, computation can be considerably reduced.

As an example of a very simple linear classifier in $\mathbf{z}$–space, consider one based on nearest–means.

$$\hat{y}(\mathbf{z}) = sign(\,||\,\mathbf{z} - \bar{\mathbf{z}}_-\,||^2 - ||\,\mathbf{z} - \bar{\mathbf{z}}_+\,||^2). \qquad (17)$$

Here $\bar{\mathbf{z}}_\pm$ are the respective means of the $y = +1$ and $y = -1$ observations

$$\bar{\mathbf{z}}_\pm = \frac{1}{N_\pm} \sum_{y_i = \pm 1} \mathbf{z}_i.$$

For simplicity, let $N_+ = N_- = N/2$. Choosing the midpoint between $\bar{\mathbf{z}}_+$ and $\bar{\mathbf{z}}_-$ as the coordinate system origin, the decision rule (17) can be expressed as

$$\begin{aligned}\hat{y}(\mathbf{z}) &= sign(\mathbf{z}^T(\bar{\mathbf{z}}_+ - \bar{\mathbf{z}}_-)) \qquad (18)\\ &= sign\left(\sum_{y_i=1} \mathbf{z}^T \mathbf{z}_i - \sum_{y_i=-1} \mathbf{z}^T \mathbf{z}_i\right) \\ &= sign\left(\sum_{i=1}^{N} y_i \mathbf{z}^T \mathbf{z}_i\right) \\ &= sign\left(\sum_{i=1}^{N} y_i H(\mathbf{x}, \mathbf{x}_i)\right).\end{aligned}$$

Comparing this (18) with (12) (13), one sees that ordinary kernel rule ($\{\alpha_i = 1\}_1^N$) in $\mathbf{x}$–space is the nearest–means classifier in the $\mathbf{z}$–space of derived variables (14) whose inner product is given by the kernel function $\mathbf{z}_i^T\mathbf{z}_j = K(\mathbf{x}_i, \mathbf{x}_j)$. Therefore to construct an (implicit) nearest means classifier in $\mathbf{z}$–space, all computations can be done in $\mathbf{x}$–space because they only depend on evaluating inner products. The explicit transformations (14) need never be defined or even known.

2. Optimal separating hyperplane

Nearest–means is an especially simple linear classifier in $\mathbf{z}$–space and it leads to no compression: $\{\alpha_i = 1\}_1^N$ in (13). A support vector machine uses a more "realistic" linear classifier in $\mathbf{z}$–space, that can also be computed using only inner products, for which often many of the coefficients have the value zero ($\alpha_i = 0$). This classifier is the "optimal" separating hyperplane (OSH).

We consider first the case in which the observations representing the respective two classes are linearly separable in $\mathbf{z}$–space. This is often the case since the dimension M (14) of that (implicitly defined) space is very large. In this case the OSH is the unique hyperplane that separates two classes while maximizing the distance to the closest points in each class. Only this set of closest points equidistant to the OSH are required to define it. These closest points are called the support points (vectors). Their number can range from a minimum of two to a maximum of the training sample size N. The "margin" is defined to be the distance of support points from OSH. The $\mathbf{z}$–space linear classifier is given by

$$\hat{y}(\mathbf{z}) = sign\left(\beta_0^* + \sum_{k=1}^{M} \beta_k^* z_k\right) \tag{19}$$

where $(\beta_0^*,\ \beta^* = \{\beta_k^*\}_1^M\ )$ define the OSH. Their values can be determined using standard quadratic programming techniques.

An OSH can also be defined for the case when the two classes are not separable in $\mathbf{z}$–space by allowing some points to be on wrong side of their class margin. The amount by which they are allowed to do so is a regularization (smoothing) parameter of the procedure. In both the separable and non separable cases the solution parameter values (β_0^*, β^*) (19) are defined only by points close to boundary between the classes. The solution for β^* can be expressed as

$$\beta^* = \sum_{i=1}^{N} \alpha_i^*\, y_i\, \mathbf{z}_i$$

with $\alpha_i^* \neq 0$ only for points on, or on the wrong side of, their class margin. These are the support vectors. The SVM classifier is thereby

$$\begin{aligned}\hat{y}(\mathbf{z}) &= sign\left(\beta_0^* + \sum_{i=1}^{N} \alpha_i^*\, y_i\, \mathbf{z}^T\mathbf{z}_i\right) \\ &= sign\left(\beta_0^* + \sum_{\alpha_i^* \neq 0} \alpha_i^*\, y_i\, K(\mathbf{x}, \mathbf{x}_i)\right).\end{aligned}$$

This is a weighted kernel classifier involving only support vectors. Also (not shown here), the quadratic program used to solve for the OSH involves the data only through the inner products $\mathbf{z}_i^T\mathbf{z}_j = K(\mathbf{x}_i, \mathbf{x}_j)$. Thus, one only needs to specify the kernel function to implicitly define $\mathbf{z}$–variables (kernel trick).

Besides the polynomial kernel (16), other popular kernels used with support vector machines are the "radial basis function" kernel

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\,||\,\mathbf{x} - \mathbf{x}'\,||^2/2\sigma^2), \tag{20}$$

and the "neural network" kernel

$$K(\mathbf{x}, \mathbf{x}') = \tanh(a\,\mathbf{x}^T\mathbf{x}' + b). \tag{21}$$

Note that both of these kernels (20) (21) involve additional tuning parameters, and produce infinite dimensional derived variable (14) spaces ($M = \infty$).

3. Penalized learning formulation

The support vector machine was motivated above by the optimal separating hyperplane in the high dimensional space of the derived variables (14). There is another equivalent formulation in that space that shows that the SVM procedure is related to other well known statistically based methods. The parameters of the OSH (19) are the solution to

$$(\beta_0^*, \beta^*) = \arg\min_{\beta_0,\beta} \sum_{i=1}^{N} [1 - y_i(\beta_0 + \beta^T\mathbf{z}_i)]_+ + \lambda \cdot\, ||\,\beta\,||^2. \tag{22}$$

Here the expression $[\eta]_+$ represents the "positive part" of its argument; that is, $[\eta]_+ = \max(0, \eta)$. The "regularization" parameter λ is related to the SVM smoothing parameter mentioned above. This expression (22) represents a penalized learning problem where the goal is to minimize the empirical risk on the training data using as a loss criterion

$$L(y, F(\mathbf{z})) = [1 - yF(\mathbf{z})]_+, \tag{23}$$

where

$$F(\mathbf{z}) = \beta_0 + \beta^T\mathbf{z},$$

subject to an increasing penalty for larger values of

$$||\,\beta\,||^2 = \sum_{j=1}^{n} \beta_j^2. \tag{24}$$

This penalty (24) is well known and often used to regularize statistical procedures, for example linear least squares regression leading to ridge–regression (Hoerl and Kannard 1970)

$$(\beta_0^*, \beta^*) = \arg\min_{\beta_0,\beta} \sum_{i=1}^{N} [y_i - (\beta_0 + \beta^T \mathbf{z}_i)]^2 + \lambda \cdot ||\, \beta \,||^2. \tag{25}$$

The "hinge" loss criterion (23) is not familiar in statistics. However, it is closely related to one that is well known in that field, namely conditional negative log–likelihood associated with logistic regression

$$L(y, F(\mathbf{z})) = -\log[1 + e^{-yF(\mathbf{z})}]. \tag{26}$$

In fact, one can view the SVM hinge loss as a piecewise–linear approximation to (26). Unregularized logistic regression is one of the most popular methods in statistics for treating binary response outcomes (9). Thus, a support vector machine can be viewed as an approximation to *regularized* logistic regression (in $\mathbf{z}$–space) using the ridge–regression penalty (24).

This penalized learning formulation forms the basis for extending SVMs to the regression setting where the response variable y assumes numeric values $y \in R^1$, rather than binary values (9). One simply replaces the loss criterion (23) in (22) with

$$L(y, F(\mathbf{z})) = (|y - F(\mathbf{z})| - \varepsilon)_+. \tag{27}$$

This is called the "ε–insensitive" loss and can be viewed as a piecewise–linear approximation to the Huber 1964 loss

$$L(y, F(\mathbf{z})) = \begin{cases} |y - F(\mathbf{z})|^2/2 & |y - F(\mathbf{z})| \le \varepsilon \\ \varepsilon(|y - F(\mathbf{z})| - \varepsilon/2) & |y - F(\mathbf{z})| > \varepsilon \end{cases} \tag{28}$$

often used for robust regression in statistics. This loss (28) is a compromise between squared–error loss (25) and absolute–deviation loss $L(y, F(\mathbf{z})) = |y - F(\mathbf{z})|$. The value of the "transition" point ε differentiates the errors that are treated as "outliers" being subject to absolute–deviation loss, from the other (smaller) errors that are subject to squared–error loss.

4. SVM properties

Support vector machines inherit most of the advantages of ordinary kernel methods discussed in Section II. In addition, they can overcome the computation problems associated with prediction, since only the support vectors ($\alpha_i \neq 0$ in (13)) are required for making predictions. If the number of support vectors is much smaller that than the total sample size N, computation is correspondingly reduced. This will tend to be the case when there is small overlap between the respective distributions of the two classes in the space of the original predictor variables $\mathbf{x}$ (small Bayes error rate).

The computational savings in prediction are bought by dramatic increase in computation required for training. Ordinary kernel methods (4) require no training; the data set is the model. The quadratic program for obtaining the optimal separating hyperplane (solving (22)) requires computation proportional to the *square* of the sample size (N^2), multiplied by the number of resulting support vectors. There has been much research on fast algorithms for training SVMs, extending computational feasibility to data sets of size $N \lesssim 30,000$ or so. However, they are still not feasible for really large data sets $N \gtrsim 100,000$.

SVMs share some of the disadvantages of ordinary kernel methods. They are a black–box procedure with little interpretive value. Also, as with all kernel methods, performance can be very sensitive to kernel (distance function) choice (5). For good performance the kernel needs to be matched to the properties of the target function $F^*(\mathbf{x})$ (2), which are often unknown. However, when there is a known "natural" distance for the problem, SVMs represent very powerful learning machines.

B. Boosted trees

Boosting decision trees was first proposed by Freund and Schapire 1996. The basic idea is rather than using just a single tree for prediction, a linear combination of (many) trees

$$F(\mathbf{x}) = \sum_{m=1}^{M} a_m T_m(\mathbf{x}) \tag{29}$$

is used instead. Here each $T_m(\mathbf{x})$ is a decision tree of the type discussed in Section III and a_m is its coefficient in the linear combination. This approach maintains the (statistical) advantages of trees, while often dramatically increasing accuracy over that of a single tree.

1. Training

The recursive partitioning technique for constructing a single tree on the training data was discussed in Section III. Algorithm 1 describes a forward stagewise method for constructing a prediction machine based on a linear combination of M trees.

Algorithm 1
Forward stagewise boosting

1 $F_0(\mathbf{x}) = 0$
2 For $m = 1$ to M do:
3 $\quad (a_m, T_m(\mathbf{x})) = \arg\min_{a, T(\mathbf{x})}$
4 $\quad \sum_{i=1}^{N} L(y_i, F_{m-1}(\mathbf{x}_i) + aT(\mathbf{x}_i))$
5 $\quad F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}_i) + a_m T_m(\mathbf{x})$
6 EndFor
7 $F(\mathbf{x}) = F_M(\mathbf{x}) = \sum_{m=1}^{M} a_m T_m(\mathbf{x})$

The first line initializes the predicting function to everywhere have the value zero. Lines 2 and 6 control the M iterations of the operations associated with lines 3–5. At each iteration m there is a current predicting function $F_{m-1}(\mathbf{x})$. At the first iteration this is the initial function $F_0(\mathbf{x}) = 0$, whereas for $m > 1$ it is the linear combination of the $m-1$ trees induced at the previous iterations. Lines 3 and 4 construct that tree $T_m(\mathbf{x})$, and find the corresponding coefficient a_m, that minimize the estimated prediction risk (1) on the training data when $a_m T_m(\mathbf{x})$ is added to the current linear combination $F_{m-1}(\mathbf{x})$. This is then added to the current approximation $F_{m-1}(\mathbf{x})$ on line 5, producing a current predicting function $F_m(\mathbf{x})$ for the next $(m+1)$st iteration.

At the first step, $a_1 T_1(\mathbf{x})$ is just the standard tree build on the data as described in Section III, since the current function is $F_0(\mathbf{x}) = 0$. At the next step, the estimated optimal tree $T_2(\mathbf{x})$ is found to add to it with coefficient a_2, producing the function $F_2(\mathbf{x}) = a_1 T_1(\mathbf{x}) + a_2 T_2(\mathbf{x})$. This process is continued for M steps, producing a predicting function consisting of a linear combination of M trees (line 7).

The potentially difficult part of the algorithm is constructing the optimal tree to add at each step. This will depend on the chosen loss function $L(y, F)$. For squared–error loss

$$L(y, F) = (y - F)^2$$

the procedure is especially straight forward, since

$$\begin{aligned} L(y, F_{m-1} + aT) &= (y - F_{m-1} - aT)^2 \\ &= (r_m - aT)^2. \end{aligned}$$

Here $r_m = y - F_{m-1}$ is just the error ("residual") from the current model F_{m-1} at the mth iteration. Thus each successive tree is built in the standard way to best predict the *errors* produced by the linear combination of the previous trees. This basic idea can be extended to produce boosting algorithms for any differentiable loss criterion $L(y, F)$ (Friedman 2001).

As originally proposed the standard tree construction algorithm was treated as a primitive in the boosting algorithm, inserted in lines 3 and 4 to produced a tree that best predicts the current errors $\{r_{im} = y_i - F_{m-1}(\mathbf{x}_i)\}_1^N$. In particular, an optimal tree size was estimated at each step in the standard tree building manner. This basically assumes that each tree will be the last one in the sequence. Since boosting often involves hundreds of trees, this assumption is far from true and as a result accuracy suffers. A better strategy turns out to be (Friedman 2001) to use a constant tree size (J regions) at each iteration, where the value of J is taken to be small, but not too small. Typically $4 \leq J \leq 10$ works well in the context of boosting, with performance being fairly insensitive to particular choices.

2. Regularization

Even if one restricts the size of the trees entering into a boosted tree model it is still possible to fit the training data arbitrarily well, reducing training error to zero, with a linear combination of enough trees. However, as is well known in statistics, this is seldom the best thing to do. Fitting the training data too well can increase prediction risk on future predictions. This is a phenomenon called "over–fitting". Since each tree tries to best fit the errors associated with the linear combination of previous trees, the training error monotonically decreases as more trees are included. This is, however, not the case for *future* prediction error on data not used for training.

Typically at the beginning, future prediction error decreases with increasing number of trees M until at some point M^* a minimum is reached. For $M > M^*$, future error tends to (more or less) monotonically increase as more trees are added. Thus there is an optimal number M^* of trees to include in the linear combination. This number will depend on the problem (target function (2), training sample size N, and signal to noise ratio). Thus, in any given situation, the value of M^* is unknown and must be estimated from the training data itself. This is most easily accomplished by the "early stopping" strategy used in neural network training. The training data is randomly partitioned into learning and test samples. The boosting is performed using only the data in the learning sample. As iterations proceed and trees are added, prediction risk as estimated on the test sample is monitored. At that point where a definite upward trend is detected iterations stop and M^* is estimated as the value of M producing the smallest prediction risk on the test sample.

Inhibiting the ability of a learning machine to fit the training data so as to increase future performance is called a "method–of–regularization". It can be motivated from a frequentist perspective in terms of the "bias–variance trade–off" (Geman, Bienenstock and Doursat 1992) or by the Bayesian introduction of a prior distribution over the space of solution functions. In either case, controlling the number of trees is not the only way to regularize. Another method commonly used in statistics is "shrinkage". In the context of boosting, shrinkage can be accomplished by replac-

ing line 5 in Algorithm 1 by

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + (\nu \cdot a_m)\, T_m(\mathbf{x})\,. \qquad (30)$$

Here the contribution to the linear combination of the estimated best tree to add at each step is reduced by a factor $0 < \nu \leq 1$. This "shrinkage" factor or "learning rate" parameter controls the rate at which adding trees reduces prediction risk on the learning sample; smaller values produce a slower rate so that more trees are required to fit the learning data to the same degree.

Shrinkage (30) was introduced in Friedman 2001 and shown empirically to dramatically improve the performance of all boosting methods. Smaller learning rates were seen to produce more improvement, with a diminishing return for $\nu \lesssim 0.1$, provided that the estimated optimal number of trees $M^*(\nu)$ for that value of ν is used. This number increases with decreasing learning rate, so that the price paid for better performance is increased computation.

3. Penalized learning formulation

The introduction of shrinkage (30) in boosting was originally justified purely on empirical evidence and the reason for its success was a mystery. Recently, this mystery has been solved (Hastie, Tibshirani and Friedman 2001 and Efron, Hastie, Johnstone and Tibshirani 2002). Consider a learning machine consisting of a linear combination of *all* possible (J–region) trees:

$$\hat{F}(\mathbf{x}) = \sum \hat{a}_m T_m(\mathbf{x}) \qquad (31)$$

where

$$\{\hat{a}_m\} = \arg\min_{\{a_m\}} \sum_{i=1}^{N} L\left(y_i, \sum a_m T_m(\mathbf{x}_i)\right) + \lambda \cdot P(\{a_m\}). \qquad (32)$$

This is a penalized (regularized) linear regression, based on a chosen loss criterion L, of the response values $\{y_i\}_1^N$ on the predictors (trees) $\{T_m(\mathbf{x}_i)\}_{i=1}^N$. The first term in (32) is the prediction risk on the training data and the second is a penalty on the values of the coefficients $\{a_m\}$. This penalty is required to regularize the solution because the number of all possible J–region trees is infinite. The value of the "regularization" parameter λ controls the strength of the penalty. Its value is chosen to minimize an estimate of future prediction risk, for example based on a left out test sample.

A commonly used penalty for regularization in statistics is the "ridge" penalty

$$P(\{a_m\}) = \sum a_m^2 \qquad (33)$$

used in ridge–regression (25) and support vector machines (22). This encourages small coefficient absolute values by penalizing the l_2–norm of the coefficient vector. Another penalty becoming increasingly popular is the "lasso" (Tibshirani 1996)

$$P(\{a_m\}) = \sum |\, a_m \,|. \qquad (34)$$

This also encourages small coefficient absolute values, but by penalizing the l_1–norm. Both (33) and (34) increasingly penalize larger average absolute coefficient values. They differ in how they react to dispersion or variation of the absolute coefficient values. The ridge penalty discourages dispersion by penalizing variation in absolute values. It thus tends to produce solutions in which coefficients tend to have equal absolute values and none with the value zero. The lasso (34) is indifferent to dispersion and tends to produce solutions with a much larger variation in the absolute values of the coefficients, with many of them set to zero. The best penalty will depend on the (unknown population) optimal coefficient values. If these have more or less equal absolute values the ridge penalty (33) will produce better performance. On the other hand, if their absolute values are highly diverse, especially with a few large values and many small values, the lasso will provide higher accuracy.

As discussed in Hastie, Tibshirani and Friedman 2001 and rigorously derived in Efron *et al* 2002, there is a connection between boosting (Algorithm 1) with shrinkage (30) and penalized linear regression on all possible trees (31) (32) using the lasso penalty (34). They produce very similar solutions as the shrinkage parameter becomes arbitrarily small $\nu \to 0$. The number of trees M is inversely related to the penalty strength parameter λ; more boosted trees corresponds to smaller values of λ (less regularization). Using early stopping to estimate the optimal number M^* is equivalent to estimating the optimal value of the penalty strength parameter λ. Therefore, one can view the introduction of shrinkage (30) with a small learning rate $\nu \lesssim 0.1$ as approximating a learning machine based on all possible (J–region) trees with a lasso penalty for regularization. The lasso is especially appropriate in this context because among all possible trees only a small number will likely represent very good predictors with population optimal absolute coefficient values substantially different from zero. As noted above, this is an especially bad situation for the ridge penalty (33), but ideal for the lasso (34).

4. Boosted tree properties

Boosted trees maintain almost all of the advantages of single tree modeling described in Section III A while often dramatically increasing their accuracy. One of the properties of single tree models leading to inaccuracy is the coarse piecewise constant nature of the resulting approximation. Since boosted tree machines

are linear combinations of individual trees, they produce a superposition of piecewise constant approximations. These are of course also piecewise constant, but with many more pieces. The corresponding discontinuous jumps are very much smaller and they are able to more accurately approximate smooth target functions.

Boosting also dramatically reduces the instability associated with single tree models. First only small trees (Section IV B 1) are used which are inherently more stable than the generally larger trees associated with single tree approximations. However, the big increase in stability results from the averaging process associated with using the linear combination of a large number of trees. Averaging reduces variance; that is why it plays such a fundamental role in statistical estimation.

Finally, boosting mitigates the fragmentation problem plaguing single tree models. Again only small trees are used which fragment the data to a much lesser extent than large trees. Each boosting iteration uses the entire data set to build a small tree. Each respective tree can (if dictated by the data) involve different sets of predictor variables. Thus, each prediction can be influenced by a large number of predictor variables associated with all of the trees involved in the prediction if that is estimated to produce more accurate results.

The computation associated with boosting trees roughly scales as $nN \log N$ with the number of predictor variables n and training sample size N. Thus, it can be applied to fairly large problems. For example, problems with $n \backsim 10^2$–10^3 and $N \backsim 10^5$–10^6 are routinely feasible.

The one advantage of single decision trees not inherited by boosting is interpretability. It is not possible to inspect the very large number of individual tree components in order to discern the relationships between the response y and the predictors $\mathbf{x}$. Thus, like support vector machines, boosted tree machines produce black–box models. Techniques for interpreting boosted trees as well as other black–box models are described in Friedman 2001.

C. Connections

The preceding section has reviewed two of the most important advances in machine learning in the recent past: support vector machines and boosted decision trees. Although motivated from very different perspectives, these two approaches share fundamental properties that may account for their respective success. These similarities are most readily apparent from their respective penalized learning formulations (Section IV A 3 and Section IV B 3). Both build linear models in a very high dimensional space of derived variables, each of which is a highly nonlinear function of the original predictor variables $\mathbf{x}$. For support vector machines these derived variables (14) are implicitly defined through the chosen kernel $K(\mathbf{x}, \mathbf{x}')$ defining their inner product (15). With boosted trees these derived variables are all possible (J–region) decision trees (31) (32).

The coefficients defining the respective linear models in the derived space for both methods are solutions to a penalized learning problem (22) (32) involving a loss criterion $L(y, F)$ and a penalty on the coefficients $P(\{a_m\})$. Support vector machines use $L(y, F) = (1 - y\,F)_+$ for classification $y \in \{-1, 1\}$, and (27) for regression $y \in R^1$. Boosting can be used with any (differentiable) loss criterion $L(y, F)$ (Friedman 2001). The respective penalties $P(\{a_m\})$ are (24) for SVMs and (34) with boosting. Additionally, both methods have a computational trick that allows all (implicit) calculations required to solve the learning problem in the very high (usually infinite) dimensional space of the derived variables $\mathbf{z}$ to be performed in the space of the original variables $\mathbf{x}$. For support vector machines this is the kernel trick (Section IV A 1), whereas with boosting it is forward stagewise tree building (Algorithm 1) with shrinkage (30).

The two approaches do have some basic differences. These involve the particular derived variables defining the linear model in the high dimensional space, and the penalty $P(\{a_m\})$ on the corresponding coefficients. The performance of any linear learning machine based on derived variables (14) will depend on the detailed nature of those variables. That is, different transformations $\{h_k(\mathbf{x})\}$ will produce different learners as functions of the original variables $\mathbf{x}$, and for any given problem some will be better than others. The prediction accuracy achieved by a particular set of transformations will depend on the (unknown) target function $F^*(\mathbf{x})$ (2). With support vector machines the transformations are implicitly defined through the chosen kernel function. Thus the problem of choosing transformations becomes, as with any kernel method, one of choosing a particular kernel function $K(\mathbf{x}, \mathbf{x}')$ ("kernel customizing").

Although motivated here for use with decision trees, boosting can in fact be implemented using any specified "base learner" $h(\mathbf{x}; \mathbf{p})$. This is a function of the predictor variables $\mathbf{x}$ characterized by a set of parameters $\mathbf{p} = \{p_1, p_2, \cdots\}$. A particular set of joint parameter values $\mathbf{p}$ indexes a particular function (transformation) of $\mathbf{x}$, and the set of all functions induced over all possible joint parameter values define the derived variables of the linear prediction machine in the transformed space. If all of the parameters assume values on a finite discrete set this derived space will be finite dimensional, otherwise it will have infinite dimension. When the base learner is a decision tree the parameters represent the identities of the predictor variables used for splitting, the split points, and the response values assigned to the induced regions.

The forward stagewise approach can be used with any base learner by simply substituting it for the decision tree $T(\mathbf{x}) \rightarrow h(\mathbf{x};\mathbf{p})$ in lines 3–5 of Algorithm 1. Thus boosting provides explicit control on the choice of transformations to the high dimensional space. So far boosting has seen greatest success with decision tree base learners, especially in data mining applications, owing to their advantages outlined in Section III A. However, boosting other base learners can provide potentially attractive alternatives in some situations.

Another difference between SVMs and boosting is the nature of the regularizing penalty $P(\{a_m\})$ that they implicitly employ. Support vector machines use the "ridge" penalty (24). The effect of this penalty is to shrink the absolute values of the coefficients $\{\beta_m\}$ from that of the unpenalized solution $\lambda = 0$ (22), while discouraging dispersion among those absolute values. That is, it prefers solutions in which the derived variables (14) all have similar influence on the resulting linear model. Boosting implicitly uses the "lasso" penalty (34). This also shrinks the coefficient absolute values, but it is indifferent to their dispersion. It tends to produce solutions with relatively few large absolute valued coefficients and many with zero value.

If a very large number of the derived variables in the high dimensional space are all highly relevant for prediction then the ridge penalty used by SVMs will provide good results. This will be the case if the chosen kernel $K(\mathbf{x},\mathbf{x}')$ is well matched to the unknown target function $F^*(\mathbf{x})$ (2). Kernels not well matched to the target function will (implicitly) produce transformations (14) many of which have little or no relevance to prediction. The homogenizing effect of the ridge penalty is to inflate estimates of their relevance while deflating that of the truly relevant ones, thereby reducing prediction accuracy. Thus, the sharp sensitivity of SVMs on choice of a particular kernel can be traced to the implicit use of the ridge penalty (24).

By implicitly employing the lasso penalty (34), boosting anticipates that only a small number of its derived variables are likely to be highly relevant to prediction. The regularization effect of this penalty tends to produce large coefficient absolute values for those derived variables that appear to be relevant and small (mostly zero) values for the others. This can sacrifice accuracy if the chosen base learner happens to provide an especially appropriate space of derived variables in which a large number turn out to be highly relevant. However, this approach provides considerable robustness against less than optimal choices for the base learner and thus the space of derived variables.

V. CONCLUSION

A choice between support vector machines and boosting depends on one's a priori knowledge concerning the problem at hand. If that knowledge is sufficient to lead to the construction of an especially effective kernel function $K(\mathbf{x},\mathbf{x}')$ then an SVM (or perhaps other kernel method) would be most appropriate. If that knowledge can suggest an especially effective base learner $h(\mathbf{x};\mathbf{p})$ then boosting would likely produce superior results. As noted above, boosting tends to be more robust to misspecification. These two techniques represent additional tools to be considered along with other machine learning methods. The best tool for any particular application depends on the detailed nature of that problem. As with any endeavor one must match the tool to the problem. If little is known about which technique might be best in any given application, several can be tried and effectiveness judged on independent data not used to construct the respective learning machines under consideration.

VI. ACKNOWLEDGMENTS

Helpful discussions with Trevor Hastie are gratefully acknowledged. This work was partially supported by the Department of Energy under contract DE-AC03-76SF00515, and by grant DMS–97–64431 of the National Science Foundation.

[1] Bellman, R. E. (1961). Adaptive Control Processes. Princeton University Press.

[2] Breiman, L. (1996). Bagging predictors. *Machine Learning* **26**, 123-140.

[3] Breiman, L. (2001). Random forests, random features. Technical Report, University of California, Berkeley.

[4] Breiman, L., Friedman, J. H., Olshen, R. and Stone, C. (1983). *Classification and Regression Trees.* Wadsworth.

[5] Efron, B., Hastie, T., Johnstone, I., and Tibshirani, R. (2002). Least angle regression. Annals of Statistics. To appear.

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

[7] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics* **29**, 1189-1232.

[8] Geman, S., Bienenstock, E. and Doursat, R. (1992). Neural networks and the bias/variance dilemma. Neural Computation **4**, 1-58.

[9] Hastie, T., Tibshirani, R. and Friedman, J.H. (2001). *The Elements of Statistical Learning.* Springer–

Verlag.
[10] Hoerl, A. E. and Kennard, R. (1970). Ridge regression: biased estimation for nonorthogonal problems. *Technometrics* **12**, 55-67
[11] Nadaraya, E. A. (1964). On estimating regression. *Theory Prob. Appl.* 10, 186-190.
[12] Quinlan, R. (1992). *C4.5: Programs for machine learning*. Morgan Kaufmann, San Mateo.
[13] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *J. Royal Statist. Soc.* **58**, 267-288.
[14] Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory.* Springer.
[15] Wahba, G. (1990). *Spline Models for Observational Data.* SIAM, Philadelphia.
[16] Watson, G. S. (1964). Smooth regression analysis. *Sankhya Ser. A.* **26**, 359-372.