Journal of Machine Learning Research 3 (2003) 1333-1356 Submitted 5/02; Published 3/03

Grafting: Fast, Incremental Feature Selection by Gradient Descent in Function Space

Simon Perkins S.PERKINS@LANL.GOV
Space and Remote Sensing Sciences
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Kevin Lacker LACKER@EECS.BERKELEY.EDU
Department of Computer Science
University of California, Berkeley
CA 94720, USA

James Theiler JT@LANL.GOV
Space and Remote Sensing Sciences
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Editors: Isabelle Guyon and André Elisseeff

Abstract

We present a novel and flexible approach to the problem of feature selection, called *grafting*. Rather than considering feature selection as separate from learning, grafting treats the selection of suitable features as an integral part of learning a predictor in a regularized learning framework. To make this regularized learning process sufficiently fast for large scale problems, grafting operates in an incremental iterative fashion, gradually building up a feature set while training a predictor model using gradient descent. At each iteration, a fast gradient-based heuristic is used to quickly assess which feature is most likely to improve the existing model, that feature is then added to the model, and the model is incrementally optimized using gradient descent. The algorithm scales linearly with the number of data points and at most quadratically with the number of features. Grafting can be used with a variety of predictor model classes, both linear and non-linear, and can be used for both classification and regression. Experiments are reported here on a variant of grafting for classification, using both linear and non-linear models, and using a logistic regression-inspired loss function. Results on a variety of synthetic and real world data sets are presented. Finally the relationship between grafting, stagewise additive modelling, and boosting is explored.

Keywords: Feature selection, functional gradient descent, loss functions, margin space, boosting.

1. Introduction

Systems for performing automated feature selection have long occupied a strange position, acting as a bridge between the harsh reality of the real world, and the cozy idealistic environments inhabited by most machine learning algorithms. No wonder then, that feature selection is often seen as something rather separate from learning, and altogether much more *ad hoc* and mysterious. As a result, most feature selection methods are rather independent of the learning systems they work with. Filter methods, for example the RELIEF algorithm (Kira and Rendell, 1992), use a quickly computed heuristic to estimate the value of each feature, individually or in combination, and use this to select a set of features before the underlying learning engine ever sees the data. Wrapper methods (Kohavi and John, 1997), do at least interact with an underlying learning engine, but they typically only communicate with it through brief summaries of performance, *e.g.* cross-validation scores. All the other information that the learning system might have gleaned from the data is usually ignored when choosing features.

More recently however, great efforts have been made to expand the applicability and robustness of general learning engines, and as a result, the distinction between feature selection and learning is beginning to look a little artificial. After all, are they not just two sides of the same overall task of learning a good model, given a set of training data described using a large number of features, and without using any special domain knowledge?

This observation motivates the approach presented in this paper, where we view feature selection, whether for performance or pragmatic purposes, as part of an integrated criterion optimization scheme.

Section 2 of this paper presents the regularized risk minimization framework that forms the core of this scheme, and Section 3 introduces a fast, incremental method for finding optimal (or sometimes approximately optimal) solutions, which we call grafting.[1] Section 4 provides an empirical comparison of grafting with a number of other learning and feature selection techniques, and finally, Section 5 draws conclusions and contrasts grafting with related work in stagewise additive modelling and boosting.

2. Learning to Select Features

In this paper, we view feature selection as just one aspect of the single problem of learning a mapping based on training data described by a large number of features. At the core of this view is a common modern approach to machine learning, which can be described as *regularized risk minimization*. In the rest of this section, we review this approach, consider how it can be adapted to include feature selection, and explain why this might be a good idea.

2.1 Learning as Regularized Risk Minimization

First, a few definitions. We assume that we are trying to find a predictor function $f(\cdot)$ that maps feature[2] vectors $\mathbf{x}$ of fixed length n, onto a scalar output value. If we have a binary classification problem then we derive an output label $y \in \{-1, +1\}$ with $y = \mathrm{sign}(f(\mathbf{x}))$. If we have a regression problem then we produce an output predicted value $y = f(\mathbf{x})$. Since this is a machine learning paper, $f(.)$ is derived, via a learning procedure, from a training set consisting of m randomly sampled $(\mathbf{x}, y)$ pairs drawn from the distribution we're attempting to model.

We assume that $f(\cdot)$ is a member of a family of predictor functions that are parameterized by a set of parameters θ. We can specify a particular member of this family explicitly as $f_\theta(\cdot)$, but we will often omit the subscript for brevity. We want to come up with a θ that minimizes the expected risk:

$$R(\theta) = \int L(f_\theta(\mathbf{x}), y)\, p(\mathbf{x}, y)\, d\mathbf{x}\, dy \tag{1}$$

1. The name "grafting" is derived from "gradient feature testing", for reasons that will become clear.
2. The term "feature" rather than "variable" is used throughout this paper to cover the general case where the arguments of $f(\cdot)$ are themselves functions of the raw variables describing the problem.

where $L(f(\mathbf{x}),y)$ is a loss function that specifies how much we are penalized for returning $f(\mathbf{x})$ when the true target value is y; and $p(\mathbf{x},y)$ is the joint probability density function for $\mathbf{x}$ and y. For classification problems the most common loss function is the misclassification rate: $L \equiv \frac{1}{2}\,|y - \text{sign}(f(\mathbf{x}))|$. In general we do not know $p(\mathbf{x},y)$ in (1), so it is usually not possible to directly optimize that criterion with respect to θ. Instead, we usually work with the empirical risk R_{emp} calculated from the training data, with the integral in (1) replaced by a sum over all data points. As is well known, directly optimizing the empirical risk can lead to overfitting, so it is common in modern machine learning to attempt to minimize a combination of the empirical risk plus a regularization term to penalize over-complex solutions. That is, we attempt to minimize a criterion of the form:

$$C(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(f_\theta(\mathbf{x}_i), y_i) + \Omega(\theta) \tag{2}$$

where $\Omega(\theta)$ is a regularization function that has a high value for "complex" predictors f.

2.1.1 LOSS FUNCTIONS FOR CLASSIFICATION

In theory, we could simply use the error rate as the loss function in (2), but there are two main problems with this. First, this loss function almost inevitably leads to an optimization problem that is hard to solve exactly. Second, experience and theory have shown that we obtain robust and better generalizing classifiers if we prefer classifiers that separate the data by as wide a *margin* as possible. Discussion of this phenomenon, and many examples, can be found in Smola et al. (2000). We can usually improve generalization performance and make the criterion easier to optimize by choosing a loss function that encourages such large-margin solutions.

We define the margin for a classifier f on a single data point $\mathbf{x}$ with true label $y \in \{-1,+1\}$ to be $\rho = yf(\mathbf{x})$. The margin is positive if the point is correctly classified (the sign of $f(\mathbf{x})$ agrees with the sign of y), and negative otherwise.

Many commonly used loss functions can be conveniently defined in terms of this margin. Figure 1 illustrates a few of these.

In our classification work, we use the Binomial Negative Log Likelihood loss function (Hastie et al., 2001, p. 308):

$$L_{bnll} = \ln(1 + e^{-\rho})$$

This loss function is derived from a model that treats $f(\mathbf{x})$ as the log of the ratio of the probability that $y = +1$ to the probability that $y = -1$.

The main value of this assumption is that it allows us to calculate $p(\mathbf{x}) \equiv p(y = +1|\mathbf{x})$ from $f(\mathbf{x})$ using the following relation:

$$p(\mathbf{x}) = \frac{e^{f(\mathbf{x})}}{1 + e^{f(\mathbf{x})}}$$

This loss function can also be readily generalized to a multi-class classification problem using the ideas of multi-class logistic regression (Hastie et al., 2001, pp. 95–100). That reference also contains a derivation of the loss function.

L_{bnll} defines the BNLL loss for a single data point. It also useful to define the total loss over the training set, also known as the *empirical risk*, which is the first term in (2):

$$L_{BNLL} = \frac{1}{m}\sum_{i=1}^{m} L_{bnll}$$

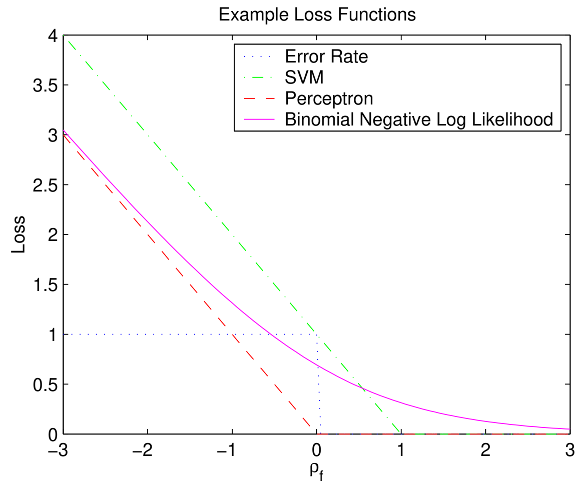

Figure 1: Commonly used loss functions, plotted as a function of the margin $\rho = yf(\mathbf{x})$. Shown are the "SVM" loss function: $L_{svm} = \max(0, 1-\rho)$; the perceptron criterion: $L_{per} = \max(0, \rho)$; and the binomial negative log-likelihood: $L_{bnll} = \ln(1+e^{-\rho})$.

2.1.2 REGULARIZATION

Possibly the most straightforward approach to machine learning involves defining a family of models and then selecting the model that minimizes the empirical risk. While this is certainly an often-used technique, it has two serious problems.

The first problem is that for certain combinations of model family, empirical risk function, and training data, the optimization problem can be unbounded with respect to the model parameters θ. For example, consider the linear model defined by:

$$f(\mathbf{x}) \equiv \sum_{i=1}^{n} w_i x_i + b \tag{3}$$

where $\mathbf{w} = (w_1, \ldots, w_n)^T$ is a vector of weights, x_i is the i'th feature of the feature vector $\mathbf{x}$, and b is a constant offset. If the training data is linearly separable, then we can increase the magnitude of $\mathbf{w}$ indefinitely to reduce L_{BNLL}.

The second problem is the well-known overfitting problem. Given a sufficiently flexible family of classifiers, it is often possible to find one that has a very low empirical risk, but that generalizes very badly to previously unseen data.

Both these problems can be tackled by adding a regularization term to the empirical risk. The regularization term (or regularizer) is a function of the model parameters that returns a high value for "unlikely" or complex models that are liable to generalize badly. By optimizing a sum of the regularizer and the empirical risk, we achieve a trade-off between model simplicity and empirical risk. If the balance is chosen appropriately then we can often improve generalization performance significantly compared with simple empirical risk minimization.

The form of the regularizer depends to some extent on the form of the model, but here we restrict ourselves to a class of models where the model parameters θ take the form of a vector of real-valued numbers of length p, which we will refer to as a weight vector $\mathbf{w}$. This class of models includes linear models and many types of multi-layer perceptron (MLP).

Given this parameter vector, we can define a commonly employed family of regularizers parameterized by a non-negative integer q, and a vector of positive real numbers α:

$$\Omega_q(\mathbf{w}) = \lambda \sum_{i=1}^{p} \alpha_i |w_i|^q \tag{4}$$

Members of this regularizer family correspond to different kinds of weighted Minkowski norm of the parameter vector, and so Ω_q is often referred to as an l_q regularizer. Usually, we choose $\alpha_i \in \{0,1\}$ so as to simply include or exclude certain elements of $\mathbf{w}$ from the regularization.[3] The essence of these regularizers is that they penalize large values of w_i when $\alpha_i > 0$.

It is easy to show that the solutions found by unconstrained minimization of (2) using an Ω_q regularizer are equivalent to those found by the following constrained optimization problem:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}_i), y_i) \\ \text{s.t.} \quad & \sum_{i=1}^{p} \alpha_i |w_i|^q \leq \gamma \end{aligned} \tag{5}$$

There is a one-to-one (but not necessarily simple) correspondence between the parameters λ and γ. This alternative formulation is useful when we consider the Ω_0 regularizer.

The most interesting members of the family involve $q \in \{0,1,2\}$. The following is a summary of the properties and peculiarities if these three regularizers.

Ω_2 This regularizer is seen in ridge regression (Hoerl and Kennard, 1970), the support vector machine (Boser et al., 1992, Schölkopf and Smola, 2002) and regularization networks (Girosi et al., 1995). Those references give various justifications for this form of regularization. One reason for preferring it over other Ω_q regularizers is that it is the only one which produces the same solution under an arbitrary rotation of the feature space axes. The l_2 norm also makes the solution to (2) bounded. As λ is increased, the magnitudes of the elements of $\mathbf{w}$ will tend to decrease, but in general none will go to zero. The l_2 norm is a convex function of weights, and so if the loss function being optimized is also a convex function of the weights, then the regularized loss has a single local (and global) optimum.

Ω_1 The l_1-based regularizer is also known as the "lasso". Tibshirani (1994) describes it in great detail and notes that one of its main advantages is that it often leads to solutions where some elements of $\mathbf{w}$ are exactly zero. As λ is increased, the number of zero weights steadily increases. Unlike the Ω_2 regularizer, using the l_1 norm means that an arbitrary rotation of the feature axes in general produces a different solution, so the feature axes have special status with this choice of regularizer. Like the Ω_2 regularizer, this regularizer leads to bounded solutions. Similarly, the l_1 norm is a convex function of weights, and so if the loss function being optimized is also a convex function of the weights, then the regularized loss has a single local (and global) optimum.

Ω_0 If we define $0^0 \equiv 0$, then this contributes a fixed penalty α_i for each weight $w_i \neq 0$. If all α_i are identical, then this is equivalent to setting a limit on the maximum number of non-zero weights. The l_0 norm is, however, a non-convex function of weights, and this tends to make exact optimization of (2) computationally expensive.

3. For instance in a linear model we usually want to exclude the constant offset term.

2.2 Feature Selection as Regularization

If we have a mathematical expression for a model in which feature vector elements only ever appear with an associated multiplicative weight, then the process of feature selection amounts to producing a model in which only a subset of weights associated with features are non-zero. Of the regularizers described above, Ω_0 and Ω_1 lead to solutions with some weights set to exactly zero. But can they be justified in terms of the standard reasons for feature selection?

There are many motivations for feature selection, but we will consider two broad classes which generally encompass the reasons most commonly given.

2.2.1 Pragmatic Motivations for Feature Selection

Often, the motivations for feature selection are pragmatic. We wish to reduce training time; reduce the time it takes to apply a learned model to new data; reduce the storage requirements for the model; or improve the intelligibility of the model. We can interpret all of these as either constituting a fixed penalty for including a feature in the model, or as a constraint on the maximum number of features in the model. For the simplest linear model case, where each feature appears in the model associated with just a single weight, then it is easy to see that both of these interpretations correspond to a Ω_0 regularizer with $\alpha_i > 0$ only where w_i is the multiplicative weight on a feature. For more complex models, where each feature might be associated with several weights, then we can handle this with a slightly modified version of the Ω_0 regularizer:

$$\Omega_0(\mathbf{w}) = \sum_{i=1}^{n} \alpha_i \delta_i \tag{6}$$

where $\delta_i = \max_{j \in s_i}(w_j^0)$, in which s_i is the set of weight indices associated with the i'th feature.

If different features carry different costs, for instance if some features are very expensive to compute, then we can adjust the α_i associated with those features accordingly.

2.2.2 Performance Motivations for Feature Selection

The other common motivation for feature selection is to improve the generalization performance of our learned models. In general, the more feature dimensions a model includes, the greater its "capacity", and hence the greater the tendency for it to overfit the training data — the so-called "curse of dimensionality". But regularization techniques are intended to prevent overfitting, and the question arises: if we use regularization, do we need to do any additional feature selection? Or alternatively, can we achieve improved generalization performance by using a regularizer that encourages zero-weighted features in our model, such as the Ω_1 and Ω_0 regularizers?

In order to explore this issue, we performed a simple experiment to compare the generalization performance of the same simple linear classifier using Ω_0, Ω_1 and Ω_2 regularizers, in the presence of varying numbers of irrelevant features. We created a sequence of simple n-feature two-class problem as follows. For the first class, the n features for each training example are drawn independently from a normal distribution with mean equal to -1 and standard deviation σ. The n features of the second class are generated in the same way except a normal distribution with a mean of +1 is used. We then randomly permute the feature values between all the training examples for all elements of the feature vector except the first two features. This produces a training set where each feature has exactly the same distribution, but only the first two are correlated with the class label and the other $n-2$

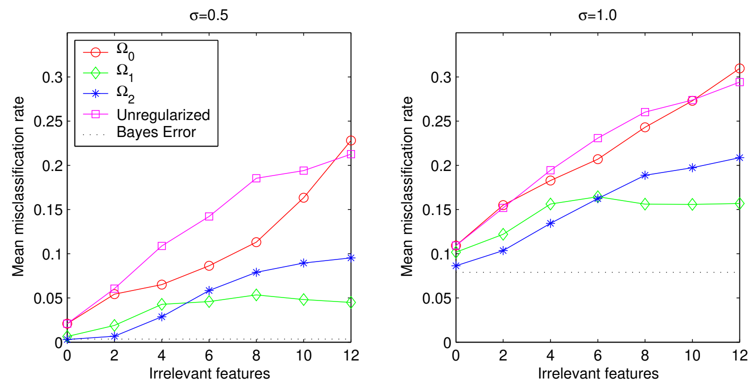

Figure 2: Comparison of different regularization schemes on a problem with varying numbers of irrelevant features and different optimal Bayes error.

features are irrelevant. Training sets with between 0 and 12 irrelevant features were generated, each containing 10 samples drawn from each class. We compared problems with $\sigma = 0.5$ and $\sigma = 1.0$. The former has a Bayes misclassification rate of 0.0035, the latter has a Bayes misclassification rate of 0.079. Test sets were generated in the same way, but with 1000 samples in each class. We used a linear model as in (3) which was trained by optimizing a regularized risk criterion (2), using the logistic regression loss function L_{bnll} and one of the three Ω_q regularizers, or using no regularization. For Ω_1 and Ω_2 we used a gradient descent algorithm (Nelder and Mead, 1965), and for Ω_0 we used backward elimination (Kohavi and John, 1997), choosing to eliminate the feature that increased the loss function by the least at each step. This is a greedy procedure that may not produce the optimal answer, but exhaustive subset comparison was impractically slow. The regularization parameters (λ for the Ω_1 and Ω_2 regularizers, γ for the Ω_0 regularizer) were found by generating multiple instances of each problem and searching for the values that minimized the average misclassification rate.

Figure 2 compares the performance of the various regularizers as the number of irrelevant features is altered. For each problem type, 200 training and test sets were randomly generated. The plots show the mean test score for each type of regularizer, and for the two different values of σ. For reasons of clarity, error bars indicating the standard error of the mean misclassification rate are not shown here, but they are small compared to the separation between the curves.

Both values of σ produce qualitatively similar results. The unregularized and subset selection (Ω_0) experiments perform worst, although subset selection does relatively better when the informative features are well-separated in the low σ case. Of the other two experiments, when most features are relevant, then Ω_2 regularization slightly outperforms Ω_1 regularization. But as the number of irrelevant features increases, Ω_1 regularization takes the lead. Interestingly, when more than two-thirds of the features are irrelevant in this case, the test performance using Ω_1 regularization seems to level off, while the performance using Ω_2 regularization continues to degrade.

In conclusion, if performance alone is the key concern, then either Ω_1 or Ω_2, rather than Ω_0, seem to be the preferred regularizers. If we expect a large fraction of irrelevant features, then we might prefer the Ω_1 regularizer. These conclusions are somewhat similar to those reached by Tibshirani (1994).

Criterion	Ω_0	Ω_1	Ω_2
Models pragmatic motivations for feature selection?	Yes	No	No
Models performance motivations for feature selection?	No	Yes	Yes
Leads to sparse solutions?	Yes	Yes	No
Performance when most features are relevant?	OK	Good	Excellent
Performance when most features are irrelevant?	Poor	Good	OK
Convex regularizer? (doesn't add extra local optima)	No	Yes	Yes
Numerical friendliness	Poor	Good	Excellent

Table 1: Comparison of three different Ω_q regularizers.

2.3 A Unified Optimization Criterion

We have argued that for a significant class of models, described by real-valued parameter vectors, many motivations for feature selection can be incorporated into a regularized risk minimization framework, using a suitable combination of Ω_q regularizers.

Table 1 summarizes the different qualities of the three Ω_q regularizers we have considered. In general, we might want to use all three, which leads to the following optimization criterion:

$$C(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}), y_i) + \lambda_2 \sum_{i=1}^{p} \alpha_{2,i}|w_i|^2 + \lambda_1 \sum_{i=1}^{p} \alpha_{1,i}|w_i| + \lambda_0 \sum_{i=1}^{n} \alpha_{0,i}\delta_i \tag{7}$$

3. Grafting

We wish to find a minimum of (7), with respect to our model parameters. We now consider how this might be done, and introduce the *grafting* algorithm, as a fast way of getting to an optimal or approximately optimal solution, if λ_0 or λ_1 is non-zero.

3.1 Direct Gradient Descent Optimization

Probably the most direct method of solution of (7) is to perform gradient descent with respect to the model parameters $\mathbf{w}$, until a minimum is found. If we can compute the gradient of the loss function and the regularization term(s) with respect to these parameters, then we can use conjugate gradient or quasi-Newton methods. If not, then we can use a minimization method that doesn't require gradient information, such as Powell's direction set method. See Press et al. (1992, chap. 10) for overviews of these methods.

Unfortunately there are a number of problems with this approach. Firstly, the gradient descent can be quite slow. Algorithms such as conjugate gradient descent typically scale quadratically with the number of dimensions,[4] and we are particularly interested in the domain where we have many features and so p is large. This quadratic dependence on number of model weights seems particularly wasteful if we are using the Ω_0 and/or Ω_1 regularizer, and we know that only some subset of those weights will be non-zero.

The second problem is that the Ω_0 and Ω_1 regularizers do not have continuous first derivatives with respect to the model weights. This can cause numerical problems for general purpose gradient descent algorithms that expect these derivatives to be continuous.

4. Conjugate gradient descent requires only $O(p)$ line minimizations, but the gradient calculation required to determine the direction for each minimization is also typically $O(p)$, giving total complexity that is closer to $O(p^2)$ for large n.

Finally, we have the problem of local minima. The Ω_2 and Ω_1 regularizers are convex functions of the weights and so if the loss function being used is also a convex function of weights, then we have a single optimum. The Ω_0 regularizer on the other hand is not convex and introduces many local minima into the optimization problem.

3.2 Stagewise Gradient Descent Optimization

If we are using the Ω_1 or Ω_0 regularizer, and we suspect that the number N of non-zero weights in the final model is going to be much less than the total number of weights n, then a more efficient stagewise optimization procedure suggests itself. We call this algorithm *grafting*. The basic plan is to begin with a model in which almost all weights are at zero. At each iteration of the grafting procedure, we use a fast gradient-based heuristic to decide which zero weight should be adjusted away from zero in order to decrease the optimization criterion by the maximum amount. We then perform gradient descent using that weight and any other non-zero weights in the model, and continue until no further progress can be made.

3.2.1 THE BASIC GRAFTING ALGORITHM

For ease of presentation, we first consider the case where $\lambda_0 = 0$ in (7), but λ_1 and λ_2 are non-zero. We will return to the case where $\lambda_0 \neq 0$ later. At this stage, our discussion applies to a broad class of models and loss functions.

As described above, we assume that the model we are using is parameterized by a weight vector $\mathbf{w}$. At any stage in the grafting process, the model weights are divided into two disjoint sets. Those weights $w_i \in \mathcal{F}$ are "free" to be altered as desired. The remaining weights $w_i \in \mathcal{Z} (\equiv \neg\mathcal{F})$ are fixed at zero.

We also assume that the output of the model for a given training example is differentiable with respect to the model weights, *i.e.* we can calculate $\partial f(\mathbf{x}_i)/\partial w_j$ for an arbitrary feature vector $\mathbf{x}_i$ and and an arbitrary weight w_j.

As explained below, after each grafting step (and before the first step) we minimize (7) with respect to the free weights, so before the k'th grafting step, we have:

$$\forall i \in F \qquad \frac{\partial C}{\partial w_i} = 0$$

During the k'th grafting step, we wish to move one weight from $\mathcal{Z}$ to $\mathcal{F}$. It seems sensible to select the weight which is going to have the greatest effect on reducing the optimization criterion C. The gradient of the criterion with respect to an arbitrary model weight w_i is:

$$\begin{aligned} \frac{\partial C}{\partial w_i} &= \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\partial L}{\partial f(\mathbf{x}_i)}\frac{\partial f(\mathbf{x}_i)}{\partial w_i}\right) + 2\lambda_2\alpha_{2,i}w_i + \lambda_1\alpha_{1,i}\mathrm{sign}(w_i) \\ &= \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\partial L}{\partial f(\mathbf{x}_i)}\frac{\partial f(\mathbf{x}_i)}{\partial w_i}\right) \pm \lambda_1\alpha_{1,i} \end{aligned} \tag{8}$$

The contribution from the Ω_2 term disappears because $w_i = 0$ for all $w_i \in \mathcal{Z}$. Slightly more subtle is the replacement of $\mathrm{sign}(w_i)$ with ± 1, which invites the question of what sign should be used, and whether in fact $\mathrm{sign}(0)$ has a well-defined value at all. Recall, however, that we are interested in determining which weight, when adjusted in the appropriate direction, will decrease C at the fastest rate. Consider $\partial L_{TOT}/\partial w_i$, the derivative of the total loss with respect to w_i (this is just the first term in the above expression). Suppose that $\partial L_{TOT}/\partial w_i > \lambda_1 \alpha_{1,i}$. This means that $\partial C/\partial w_i > 0$, regardless of the sign of w_i. In this case, in order to decrease C, we will want to decrease w_i. Since w_i starts at zero, the very first infinitesimal adjustment to w_i will take it negative. Therefore for our purposes we can let $\text{sign}(w_i) = -1$. Similarly, if $\partial L_{TOT}/\partial w_i < -\lambda_1 \alpha_{1,i}$, then we can effectively let $\text{sign}(w_i) = +1$. Essentially, the effect of the Ω_1 derivative is simply to reduce the *magnitude* of $\partial C/\partial w_i$ by an amount $\lambda_1 \alpha_{1,i}$. The same argument shows that if $|\partial L_{TOT}/\partial w_i| < \lambda_1 \alpha_{1,i}$ then it is not possible to produce any local decrease in C by adjusting w_i away from zero. This is the essence of why the Ω_1 regularizer leads to solutions with zero-valued weights, and also provides the basis for one of the two stopping conditions discussed below.

At each grafting step, we calculate the magnitude of $\partial C/\partial w_i$ for each $w_i \in \mathcal{Z}$, and determine the maximum magnitude. We then add the weight to the set of free weights $\mathcal{F}$, and call a general purpose gradient descent routine to optimize (7) with respect to all the free weights. Since we know how to calculate the gradient of $\partial C/\partial w_i$, we can use an efficient quasi-Newton method. We use the Broyden-Fletcher-Goldfarb-Shanno (BFGS) variant (see Press et al. 1992, chap. 10 for a description). We start the optimization at the weight values found in the $k-1$'th step, so in general only the most recently added weight changes significantly.

Note that choosing the next weight based on the magnitude of $\partial C/\partial w_i$ does not guarantee that it is the best weight to add at that stage. However, it is *much* faster than the alternative of trying an optimization with each member of $\mathcal{Z}$ in turn and picking the best. We shall see below that this procedure will take us to a solution that is at least locally optimal.

3.2.2 INCORPORATING THE Ω_0 REGULARIZER

Use of the Ω_0 regularizer means that transferring a weight w_i from $\mathcal{Z}$ to $\mathcal{F}$ incurs a penalty of $\lambda_0 \alpha_{0,i} \delta_i$. This fixed penalty makes it substantially harder to determine which weight is the most promising one to transfer to $\mathcal{F}$.[5] The heuristic we use in this case is based upon the empirical observation that in a sequence of grafting steps, the magnitude of the most recently added weight in $\mathcal{F}$ typically decreases monotonically. This allows us to estimate an upper limit on the magnitude of the weight we are about to add, which in turn means we can roughly estimate a bound on the change in C which will result from adding a weight w_i in the grafting step after a weight w_j was added:

$$\Delta C(w_i) \geq \lambda_0 \alpha_{0,i} \delta_i - |w_j| \left(\left| \frac{\partial L_{TOT}}{\partial w_i} \right| - \lambda_1 \alpha_{1,i} - \lambda_2 \alpha_{2,i} |w_j| \right) \tag{9}$$

Picking the best weight to add then amounts to choosing $w_i \in \mathcal{Z}$ that minimizes (9). Note that if $\lambda_0 = 0$ and $\forall \{w_i, w_j\} \subset \mathcal{Z} : \alpha_{2,i} = \alpha_{2,j}$, then this heuristic is equivalent to the simple heuristic previously discussed.

3.2.3 STOPPING CONDITIONS

If only the Ω_2 regularizer is being used, then in general the model will contain no zero-valued weights, and the grafting procedure will not terminate until $\mathcal{Z}$ is empty. In this case there is no advantage in using grafting over full gradient descent optimization.

5. One exception is when all weights in $\mathcal{Z}$ incur the same Ω_0 penalty, as is the case with a simple linear model where we simply penalize the number of included features. In this case it is reasonable to use the standard heuristic.

If we are using the Ω_1 regularizer, then we can reach a point where:

$$\forall w_i \in \mathcal{Z} \qquad \left|\frac{\partial L_{TOT}}{\partial w_i}\right| \leq \lambda_1 \alpha_{1,i}$$

At this point it is not possible to make any further decrease in C by either moving a weight from $\mathcal{Z}$ to $\mathcal{F}$, or by adjusting any weights in $\mathcal{F}$ and so we are at a local (and perhaps global) minimum, and can terminate the grafting procedure.

If we are using the Ω_0 regularizer, then we may reach a point where C increases after adding w_i to $\mathcal{F}$. We must then set w_i to zero, remove it from $\mathcal{F}$, and undo the last optimization step (it is convenient to keep a copy of the previous model around in order to avoid an extra optimization step). We then have a choice. It is possible that a different choice of w_i might lead to a decrease in C, so we could try the optimization step again with the w_i associated with the next lowest value of $\Delta C(w_i)$. This cycle could be repeated until all remaining weights in $\mathcal{Z}$ have been eliminated, and the algorithm then terminates. Alternatively we can just terminate the algorithm the first time this happens, recognizing that with the Ω_0 regularizer, our solution will be a greedy approximation to the optimal solution at best. The latter approach is the one we recommend in most cases.

3.2.4 OPTIMALITY

If we have a convex loss function (as a function of weights) and are using just the Ω_2 and/or Ω_1 regularizers (which are themselves convex functions of the weights), then there is only one minimum of (7). Examination of the stopping conditions above reveal that the grafting algorithm is guaranteed to stop at a local optimum, and so grafting is guaranteed to find the global optimum in these cases.

As we have seen by now, use of the Ω_0 regularizer makes it much harder to find an optimal solution. The grafting procedure with non-zero λ_0 amounts to a greedy heuristic forward subset selection method, which sacrifices optimality in return for fast learning. Whether this is good enough for the problem at hand depends on the situation.

One should note however, that as λ_0 is made smaller and smaller relative to λ_1, then the chances of ending up in a sub-optimal situation decrease. Hence we are inclined to make λ_0 fairly small in most cases.

3.2.5 COMPUTATIONAL COMPLEXITY

We have claimed that grafting is substantially faster than full gradient descent. We will now examine this claim more carefully. If there are p weights in our weight vector, then full gradient descent requires some multiple of p line minimizations to optimize our criterion, let's say cp minimizations.[6] Determining the direction requires p derivatives $\partial C/\partial w_i$ to be computed. The computation of each derivative is dominated by the computation of $\partial L_{TOT}/\partial w_i$, which is simply a weighted sum of m simple derivatives $\partial f(\mathbf{x}_j)/\partial w_i$. The line minimizations themselves require a few $O(m)$ function evaluations, but if p is large, then this is a minor contribution. If we denote the time taken to calculate one simple derivative as τ, then the total time taken for full gradient descent is $\approx cmp^2\tau$.

Under grafting we will select some number s weights before the algorithm terminates. Since we select one weight at each grafting step we take s steps. The k'th step consists of two phases. First we evaluate $\partial C/\partial w_i$ for each of the $(p-k)$ weights in $\mathcal{Z}$. As noted above, the derivative calculation takes $m\tau$, and so the time devoted to gradient testing over s steps is $\approx msp\tau$. The

6. Here, $c = 1$ if our criterion is a perfect quadratic form, and $c > 1$ otherwise.

second phase involves optimizing with respect to the k free weights. At most this might take ck line minimizations, but it should take less than this since most of the free weights will be close to their optimal values. To compensate for using a constant c that is probably too high here, we again ignore the time taken for the line minimizations themselves (some small multiple of $m\tau$). Therefore the time taken for optimization at the k'th step is $\approx cmk^2\tau$. Over s steps, this is $\approx \frac{1}{3}cms^3\tau$. Putting this together the total grafting run time is $\approx (msp + \frac{1}{3}cms^3)\tau$. If we assume that $s \ll p$ then it is clear that the grafting algorithm should be substantially quicker than the $\approx cmp^2\tau$ required for full gradient descent. Also note that the full gradient descent algorithm has to deal with discontinuities in the gradient which can slow it down significantly. By keeping zero-valued weights out of the optimization steps, grafting avoids this difficulty.

3.2.6 NORMALIZATION

In order to make the gradient magnitude heuristic a fair comparison, it is usually important to normalize all features so that they have approximately the same scale. Before we begin, we linearly scale all feature vectors so that each feature has a mean value of zero and has a standard deviation of one. It is of course important to scale testing data using the same scaling parameters derived from the training data.

3.3 Grafting Examples

It is helpful to illustrate the grafting algorithm in more detail for some particular models and loss functions. Here we will concern ourselves only with binary classification problems and so a suitable loss function to use is the binomial negative log likelihood L_{bnll}. For simplicity, we will also assume only Ω_1 and Ω_0 regularization are used, and that all $\alpha_{1,i} \in \{0,1\}$.

3.3.1 LINEAR MODEL

We first consider linear models with $n+1$ weights, of the form:

$$f(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i + w_0$$

If we define the margin for a given training pair $(\mathbf{x}_i, y_i)$ as $\rho_i = y_i f(\mathbf{x}_i)$, then the following is the regularized optimization criterion:

$$C(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}(1+e^{-\rho_i}) + \lambda_1 \sum_{i=1}^{n}|w_i| + \lambda_0 s \tag{10}$$

where s is the number of selected features. Note that the constant offset term w_0 does not appear in the regularizer since we do not want to penalize a mere translation of the linear discriminant surface.

The derivatives we need (ignoring the Ω_0 term for now) are:

$$\begin{aligned}
\frac{\partial C}{\partial w_j} &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L_{bnll}}{\partial \rho_i}\frac{\partial \rho_i}{\partial w_j} \pm \lambda w_j \\
&= -\frac{1}{m}\sum_{i=1}^{m}\frac{1}{1+e^{\rho_i}} y_i x_{i,j} \pm \lambda w_j
\end{aligned}$$

where $x_{i,j}$ is the j'th component of $\mathbf{x}_i$.

It is instructive to interpret this derivative in a geometric way. First, we imagine an m-dimensional *margin space*, which has one dimension for each training point. If we think of each of the coordinate axes as representing the margins ρ_i for the current model on the point $\mathbf{x}_i$, then the total loss function can be thought of as a function over this space, and we can calculate the full gradient of that function:

$$\nabla_\rho L = \left(\frac{\partial L}{\partial \rho_1}, \frac{\partial L}{\partial \rho_2}, \ldots, \frac{\partial L}{\partial \rho_m} \right)^T$$

In the same margin space we can also imagine the *feature margin vector* $\mathbf{r}_j$:

$$\mathbf{r}_j = (y_1 x_{1,j}, y_2 x_{2,j}, \ldots, y_m x_{m,j})^T$$

Given this we can write:

$$\frac{\partial C}{\partial w_j} = \nabla_\rho L \cdot \mathbf{r}_j \pm \lambda w_j$$

Since the grafting heuristic for selecting the next weight to add to the model is only interested in the magnitude of this derivative, and since the regularizer component always acts to reduce this magnitude, we can see that picking the next weight amounts to choosing the feature margin vector that is most well-aligned with the direction of steepest descent of the loss function in margin space.

For the linear model, we initialize $\mathcal{F}$ to contain just w_0 and perform an initial optimization. We then proceed in the usual grafting fashion, picking one new weight to add to $\mathcal{F}$ at each grafting step until an Ω_0 or Ω_1 stopping condition is reached. In this case, each weight corresponds to one feature.

The simplicity of the linear model means that it sometimes cannot fit the training data very well. But this problem is somewhat reduced when we have many features since in general the extra features will tend to make the problem more linearly separable.

3.3.2 MULTI LAYER PERCEPTRON MODEL

For a more powerful model, we might use an MLP with h hidden units having sigmoid transfer functions and a linear output unit with unit output weights. We can write this MLP function as:

$$f(\mathbf{x}) = \sum_{j=1}^{h} g\left(\sum_{i=1}^{n} w_i^{(j)} x_i + w_0^{(j)} \right) + w_0^{(0)}$$

In this model $w_i^{(j)}$ is the weight from the i'th feature to the j'th hidden unit. The sigmoid transfer function $g(\cdot)$ is defined as:

$$g(x) = \frac{2}{1+e^{-x}} - 1$$

which is the usual neural net sigmoid function, scaled vertically so that $g(0) = 0$, $g(+\infty) = +1$ and $g(-\infty) = -1$, which makes the net slightly better behaved under grafting.

The optimization criterion is almost identical to (10) with the different $f(\cdot)$ substituted. All the weights in the MLP model are included equally in the regularization term, except for the "bias" weights on each node which are excluded, and the constant weights on the output node. Rather than use a fixed number of hidden units, the grafting procedure we describe here allows hidden units to be added as the grafting process continues. In the MLP model, each weight added corresponds to a new connection in the network. We begin with $\mathcal{F}$ containing just the output bias weight $w_0^{(0)}$ and a network with no hidden units, and perform an initial optimization with respect to just that bias weight. At each grafting step we decide how to grow the network in one of two possible ways:

Adding a new hidden unit: If there are $k-1$ hidden units already, this involves adding a hidden node, along with a new connection to the output node, and a new connection from the hidden node to a feature input, with weight $w_i^{(k)}$. The question becomes: which feature should be connected to the new hidden unit? The derivative we need is:

$$\frac{\partial C}{\partial w_i^{(k)}} = -\frac{1}{m}\sum_{j=1}^{m}\frac{1}{1+e^{\rho_i}}y_j x_{j,i} \pm \lambda w_i^{(k)}$$

Adding an input connection to an existing hidden unit: Each of the h hidden units may be connected to any of the n input features and any of these connections are candidates for adding to the model (if they are not present already). If we are considering adding a connection from the i'th input feature to the k'th hidden unit, then the relevant derivative is:

$$\frac{\partial C}{\partial w_i^{(k)}} = -\frac{1}{m}\sum_{j=1}^{m}\frac{1}{1+e^{\rho_i}}y_j x_{j,i}\, g'\left(\sum_{l=1}^{n} w_l^{(k)} x_{i,l} + w_0^{(k)}\right) \pm \lambda w_i^{(k)}$$

with:

$$g'(x) = \frac{2e^{-x}}{(1+e^{-x})^2}$$

After t grafting steps, there are n possibilities to consider for adding a new hidden unit, and $(hn-t)$ possibilities to consider for adding a connection to a new hidden unit. Following the usual grafting procedure, we calculate the derivatives for all these candidates and pick the one with the largest magnitude. The corresponding weight is added to $\mathcal{F}$ and we reoptimize. The cycle is repeated until one of the stopping conditions is met. If a new hidden unit is added, then we also need to include the associated bias weight, initialized at zero, in $\mathcal{F}$.

3.4 Variants

A number of variants to the basic grafting procedure are possible. One interesting alternative is not to attempt a full optimization of the full set $\mathcal{F}$ after each grafting step. Instead only the most recently added weight and perhaps the bias weights are adjusted. This makes each grafting step faster, at the expense of a loss in accuracy and the strong possibility of ending up at a solution that is not even a local optimum. In practice, if only a small fraction of the possible weights are non-zero when grafting finishes, then the time spent checking gradients to determine the next weight to add often dominates the run time, and so saving a little effort on the optimization makes little difference.

Grafting can also be readily extended to regression problems through the use of a suitable loss function, such as the squared error loss function. This has not yet been implemented.

4. Grafting Experiments

In this section, we compare the performance of grafting and a number of other different approaches to feature selection, on a set of synthetic and real world test problems. For simplicity, we concentrate entirely on binary classification problems in this paper.

4.1 The Datasets

Five datasets were used in these experiments, labeled **A** through **E**. Each dataset consists of a training set and a test set. Datasets **A**, **B** and **C** are synthetic problems, and are all instances of the same basic task described below. Datasets **D** and **E** are real world problems, taken from the online UCI Machine Learning Repository (Blake and Merz, 1998).

The three synthetic problems are variations of a task we call the *threshold max* (TM) problem. In the most basic version of this problem, the feature space contains n_r informative features, each of which is uniformly distributed between -1 and +1. The output label for a given point is defined as:

$$y = \begin{cases} +1 & \text{if } \max(x_i) > 2^{(1-1/n_r)} - 1 \qquad ; i = 1 \ldots n_r \\ \text{-1} & \text{Otherwise} \end{cases}$$

The $y = -1$ points occupy a large hypercube wedged into one corner of the larger hypercube containing all the points. The $y = +1$ points fill the remaining space. The constant in the above expression is chosen so that half the feature space belongs to each class. Variations of this basic problem are derived by adding n_i irrelevant features uniformly distributed between -1 and +1, and n_c redundant features which are just copies of the informative features. The TM problem is designed so that each of the informative features only provides a little information, but by using all of them together, the problem is completely separable. In addition, the optimal discriminating surface for the problem is very non-linear, but the problem is asymmetric so a linear discriminant should be able to do at least better than random.

Ten instantiations of the training and testing sets of each of the three synthetic problems were generated to obtain some statistics on relative algorithm performance.

In more detail, the datasets are:

Dataset A The TM problem, with $n_r = 10$, $n_c = 0$ and $n_i = 90$. Both the training set and the test set contain 1000 points each. This dataset explores the effect of irrelevant features in the TM problem.

Dataset B The TM problem, with $n_r = 10$, $n_c = 90$ and $n_i = 0$. Both the training set and the test set contain 1000 points each. This dataset explores the effect of redundant features in the TM problem.

Dataset C The TM problem, with $n_r = 10$, $n_c = 490$ and $n_i = 500$. The training set contains only 100 training points, despite the problem dimensionality of 1000. The test set contains 1000 points. This dataset explores an extreme situation in which there are many more features than training points.

Dataset D The "Multiple Features" database from the UCI repository. This is actually a handwritten digit recognition task, where digitized images of digits have been represented using 649 features of various types. The task tackled here is to distinguish the digit "0" from all other digits. The training and test sets both consist of 1000 points. The features were all scaled to have zero mean and unit variance before being used here.

Dataset E The "Arrhythmia" database from the UCI repository. The task here is to distinguish normal from abnormal heartbeat behavior from ECG data described by 279 numeric and binary attributes. The data was slightly modified from the original to make it easier to use. Feature number 14 ("J") was missing in most of the records, so it was removed from all the records. Of the 452 instances in the database, 32 had other missing attribute values and so those instances were also removed, leaving 420 instances described by 278 attributes. These were divided into a training set of 280 points, and a test set of 140 points.

All the datasets used in these experiments can be found online at:

```
http://nis-www.lanl.gov/~simes/data/jmlr03/
```

4.2 The Algorithms

Eight different algorithms, which we denote by the letters **(a)** through **(h)**, were compared on the five datasets described above. Except where described below, all implementations relied on Matlab (including the Matlab Optimization Toolbox).

(a) The linear grafting algorithm described in Section 3.3, and using both Ω_0 and Ω_1 regularization.

(b) The MLP grafting algorithm described in Section 3.3, and using both Ω_0 and Ω_1 regularization.

(c) Simple gradient descent to fit an Ω_1 regularized linear combination of all the input features. After gradient descent is complete, any weights with a magnitude less than 10^{-4} of the maximum magnitude in the weight vector, are pruned in order to obtain the subset selection driven by the Ω_1 regularization.

(d) Simple gradient descent to fit an Ω_1 regularized, fully connected MLP of a similar form to that learned by the MLP grafting algorithm, with the exception that the number of hidden nodes is fixed at 10 nodes (and of course the connectivity is much higher). After gradient descent is complete, any weights with a magnitude less than 10^{-4} of the maximum magnitude in the weight vector, are pruned in order to obtain the subset selection driven by the Ω_1 regularization.

(e) Linear SVM, which effectively uses Ω_2 regularization and a slightly different loss function from that used by the grafting implementations. The SVM implementation we used is `libsvm` (Chang and Lin, 2001).

(f) Gaussian RBF kernel SVM, using the default `libsvm` kernel parameters.

(g) Gaussian RBF kernel SVM as above, but in conjunction with "wrapper" feature subset selection. At each feature selection step, all possible features are considered for addition to the current feature set, and 3-fold cross-validation is used to select the best one. This process is repeated until we have selected 10 features, and a final RBF SVM is trained using just those features. This is essentially greedy forward subset selection.

(h) Gaussian RBF kernel SVM, but in conjunction with a "filter" feature subset selection. For each dataset, we simply take the 10 features that are most highly correlated with the label and train our SVM using those features.

Note that an efficient implementation of grafting must directly exploit the sparsity of the models being trained. Our Matlab implementations take care to do this.

	Linear Graft/GD (λ_1/λ_0)	MLP Graft/GD (λ_1/λ_0)	Linear SVM (C)	RBF SVM (C)
A	3×10^{-4}/0.005	10^{-6}/0.005	0.01	10
B	10^{-4}/0.005	10^{-6}/0.005	0.01	0.1
C	0.2/0.001	0.1/0.001	0.001	100
D	0.005/0.001	3×10^{-4}/0.001	1	1
E	0.05/0.001	3×10^{-5}/0.001	0.01	1

Table 2: Regularization parameters used in experiments for each dataset, chosen using five-fold cross-validation. "GD" stands for Gradient Descent.

4.3 Experimental Details

Each algorithm was applied to each of the datasets. The exception to this was the fully connected MLP (algorithm **(d)**), which failed to converge for several of the larger datasets. We suspect that this was caused by the large number of weights close to zero in the model, and the discontinuous derivative of the regularizer at this point. For the three synthetic problems the training runs were repeated for ten different random instantiations of the training and test sets, to assess sensitivity to small changes in the dataset.

The regularization parameters used in these experiments were chosen using five-fold cross validation on each of the training sets. Algorithms **(a)** and **(c)** share the same parameters; as do algorithms **(b)** and **(d)**; and algorithms **(f)**, **(g)** and **(h)**. Note that the grafting algorithms, **(a)** and **(b)**, use both Ω_1 and Ω_0 regularization, requiring parameters λ_1 and λ_0, while the corresponding simple gradient descent algorithms, **(c)** and **(d)**, use only Ω_1 regularization. This is due to the difficulty of incorporating Ω_0 regularization into a standard gradient descent algorithm. The parameters are listed in Table 2.

For the Matlab implementations (algorithms **(a)**, **(b)**, **(c)** and **(d)**) the training time on each dataset was recorded to see if grafting gave a speedup over simple gradient descent. A direct speed comparison between the SVM algorithms (written in C) and the Matlab implementations was not attempted at this time, due to the inherent efficiency differences between Matlab and C code.

We also recorded the number of features selected (not the same as the number of weights in general), for those algorithms that select a subset of the features (all algorithm except **(e)** and **(f)**). In addition, for the three synthetic problems, we can directly calculate a measure of how useful the selected feature subsets are. Recall that in each of these problems there are n_r informative features, not including redundant duplicates. We can define a feature set *saliency* measure:

$$s = \frac{n_g}{n_r} + \frac{n_g}{n_f} - 1$$

where n_g is the number of "good" features selected (*i.e.* informative features, but not including any duplicate redundant features), and n_f is the total number of features selected. The saliency evaluates to 1 if all n_r good features are selected and no others are selected. It evaluates to -1 if no good features are selected, and it evaluates to approximately zero if all the good features are selected, but only along with many irrelevant or redundant features.

Note that the number of selected features for the linear and MLP models trained by simple gradient descent (algorithms **(c)** and **(d)**) relies on a somewhat subjective pruning of low valued weights, as described above. Therefore the measurements of numbers of selected features, and

	Linear Graft	MLP Graft	Linear GD	MLP GD
A	0.5	12.1	1.1	14.6
B	0.5	13.1	0.9	13.1
C	1.1	0.6	216.9	-
D	1.1	9.4	350.4	-
E	0.4	3.3	58.9	-

Table 3: Timing comparisons for Matlab-implemented algorithms on the five datasets. The entries in the table are the mean number of CPU seconds used by each training algorithm on each dataset. Note that the MLP gradient descent algorithm could not be applied to the larger datasets due to convergence problems.

feature subset saliency should be treated with some suspicion for these two algorithms. Grafting avoids this issue by only adjusting weights that are likely to end up as non-zero.

4.4 Results and Analysis

Figure 3 summarizes the test performance and subset selection results of the experiments. Table 3 details the mean run time for the various Matlab algorithms on each dataset.

We will consider the results for each dataset in turn.

Dataset A 90% of the features in this dataset are irrelevant, while each of the others is only weakly relevant by itself. The problem is not linearly separable so the error rate of the linear algorithms **(a)**, **(c)** and **(e)** probably couldn't be much better than the observed 0.36–0.38 level, even with an optimal choice of hyperplane. Curiously enough though, the non-linear SVM algorithm **(f)** also performed badly, presumably due to the large number of irrelevant features. Adding subset selection to the non-linear SVM made it perform somewhat better — the filter selection method **(g)** slightly outperforming the wrapper method **(h)**. The winner by a very long way though is the MLP graft algorithm **(b)**. The reasons behind these performance differences can largely be explained by the saliency chart. The gradient test heuristic was able to pick out just the salient features with almost 100% accuracy, and the MLP was flexible enough to fit the non-linear discriminant boundary to a high degree of accuracy. The linear grafting algorithm also picked out good sets of features, but was unable to exploit these due to its inflexible model. The correlation filter method picked the third most salient set of features and came third in terms of performance. It is very likely that at least some of the performance difference between the MLP grafting algorithm and the non-linear SVM, is due to the fact that the MLP network matches the structure of the threshold max problem better than an RBF network.

Dataset B This time there are no irrelevant features, but 90% of them are merely copies of other features. The task is to pick out which ones are necessary. The linear models do as poorly as before, although again, linear grafting was able to pick out a very salient and minimal subset of features. In the absence of the noise features, the non-linear SVM did much better. Interestingly, the wrapper subset selection did not affect the performance of the SVM significantly. Again the MLP grafting algorithm is the runaway winner, picking out an extremely salient feature subset and achieving much lower error rates than the other algorithms. The worst algorithm by a significant margin is the

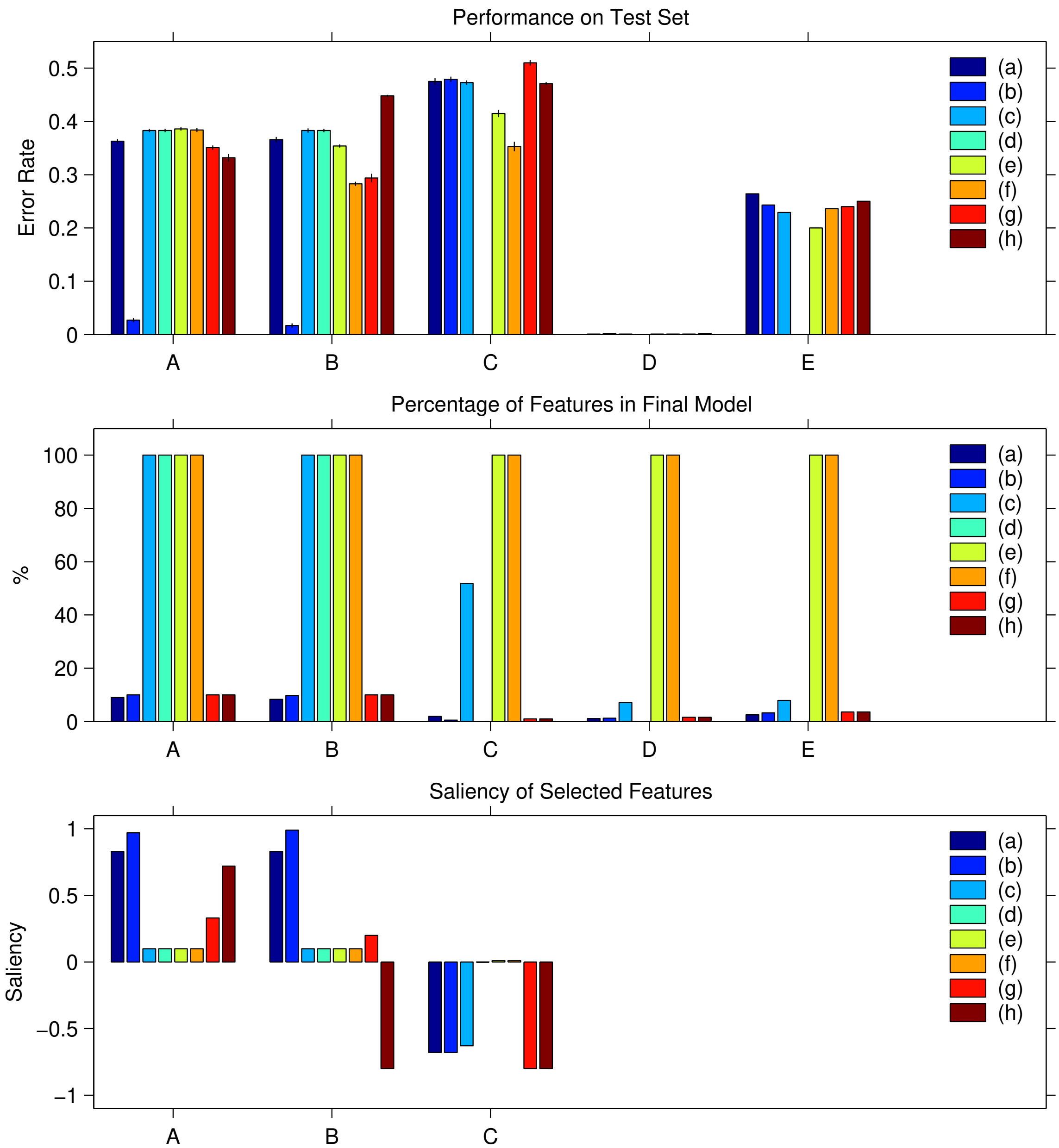

Figure 3: Experimental comparison of linear and non-linear grafting, with other learning methods on five datasets. See the main text for a description of the data sets and the algorithms. The short lines seen at the top of the first three groups of bars in the test performance chart, represent the standard error of the mean test scores, obtained over ten different random instantiations of the training and test sets. For datasets **C**, **D** and **E**, the bar corresponding to algorithm **(d)** is missing. This algorithm, which is a gradient descent trained, fully connected, Ω_1 regularized MLP, failed to converge to a solution for these larger datasets.

filter-based subset selection technique. This is to be expected since this simple filter has no reason not to pick multiple redundant features.

Dataset C With 1000 dimensions and only 100 training points this is a very hard problem. Looking at the saliency chart, it is clear that the grafting algorithms, along with both the wrapper and filter selection methods, pick out very poor subsets of features. The only algorithms to perform much better than a random classifier are the two SVM algorithms without subset selection, **(e)** and **(f)**. The non-linear SVM performs best by a significant margin. The problem seems to be that, given enough features and few enough data points, the grafting and other subset selection methods are very likely to find irrelevant features that explain the labels better than the true informative features, merely by chance. They then seize on those features as the most useful. In contrast, the basic SVM, with l_2 regularization and no subset selection, has less tendency to focus on just a few features, and so does better.

Dataset D The digit recognition problem turns out to be very easy, with most algorithms misclassifying only one member of the test set, out of 1000. The challenge here is to solve the problem quickly and using as few features as possible — many of the features are relatively expensive to compute so this is a real-world challenge. The linear and non-linear grafting algorithms **(a)** and **(b)** do an excellent job in this respect, achieving excellent performance using only about 1% of the features. This parsimony also pays off in terms of training time, as the linear grafting algorithm is over 300 times faster than training a full linear model using gradient descent.

Dataset E The arrhythmia problem is quite a hard one. The best algorithm is the linear SVM **(e)**, with an average error rate of around 0.2. All the other algorithms achieve performance figures in the vicinity of 0.25, although the grafting algorithms achieve this error rate using a very small fraction of the available features.

5. Discussion

We finish with some conclusions, and compare grafting to some better-known algorithms.

5.1 Connections with Stagewise Additive Modelling

Both the linear and MLP models considered in this paper are examples of additive models:

$$E(y|\mathbf{x}) = \sum_{i=1}^{k} f_i(\mathbf{x})$$

Hastie et al. (2001, chap. 9) provide an in-depth discussion of this class of models which have a long history in statistics. They also describe a method called additive regression that allows us to fit the above model using *backfitting* (Friedman and Stuetzle, 1981). Backfitting is a simple iterative procedure whereby the individual functions f_i are adjusted one at a time to optimize a chosen loss function, and by cycling through this process enough times we can optimize the whole model. Additive *logistic* regression specializes this problem for classification by using the binomial negative log likelihood loss function as an optimization criterion. For the classes of additive model considered in this paper, we can simply use sophisticated general purpose gradient descent algorithms to optimize the model, which are likely to be more efficient than the simple backfitting procedure.

As an alternative to using a model with a fixed number of components k, we can incrementally develop such a model using a greedy stagewise algorithm. In this approach, at each time step, we add a new component to the additive model, which is designed to reduce the current loss as much as possible. The previous components are fixed, and only the new function is adjusted. An example of this is *matching pursuit*, described by Mallat and Zhang (1993). This approach is also discussed at length by Friedman et al. (2000) where they develop a greedy stagewise form of additive logistic regression.

All this looks very similar to the grafting procedure described in this paper, so what is new about grafting? Here are some of the more important differences:

- Grafting is used to build up models out of components that may be combined together in much more complicated ways than simply being added together. In MLP graft, for instance, at each grafting step we consider both adding a new hidden node, and adding an input to an existing node. These options are considered in the same framework, but involve rather different alterations to the model. In stagewise additive modelling we are usually only concerned with adding another f_i to the model at each step.

- At each step, grafting considers many possibilities for altering the current model (typically one for each weight in $\mathcal{F}$) and decides between them using the gradient test heuristic. In stagewise additive modelling, we usually only consider one possibility, which is to add (literally) a new function f_i, which will be fit to a weighted version of the training set.

- After updating the model, stagewise additive modelling typically only adjusts parameters relating to the most recently added f_i. Grafting usually adjusts all the free parameters in the model using a gradient descent engine. This is feasible because our model output is always differentiable with respect to all the model parameters. Additive model practitioners do sometimes use backfitting to fit their models, but this is likely to be less efficient than using a quasi-Newton gradient descent algorithm.

5.2 Connections with Boosting

Boosting (Freund and Schapire, 1996) is one of the most active fields of research in modern machine learning. Boosting, and in particular the AdaBoost algorithm, provides a way of training a sequence of classifiers on some data and then combining them together to form an *ensemble classifier* that is substantially better than any of its member classifiers.

Boosting can be viewed as a form of stagewise additive modelling (Friedman et al., 2000) that optimizes a logistic regression-like loss function, combined with an l_1 regularizer. It works by reweighting the training data at each stage, training a new classifier using a "weak learner", and forming a composite classifier as a linear combination of the classifiers learned at each stage. The reweighting of the data can be shown to "encourage" a new classifier to be learned that maximally decreases the logistic regression loss (Friedman et al., 2000).

If we consider the variant of grafting mentioned in Section 3.4, where we only update the most recently added weights, and use a linear model with Ω_1 regularization, then we end up with something that is very much like an extreme version of boosting where the weak learner simply fits a model of the form $f(\mathbf{x}) = w_0 + w_1 x_k$, choosing a single feature x_k to add to the model. The technique of "boosting with stumps" is similar to this, except that $f(\mathbf{x})$ is usually thresholded at zero to produce an output that is -1 or +1, as required by the original AdaBoost algorithm. More recent versions of boosting relax this requirement (Schapire and Singer, 1999) however, so in fact this variant of linear grafting is a simple form of boosting.

The original boosting algorithm mandated an l_1 regularizer and the logistic regression loss function. However, Mason et al. (2000) have devised a general purpose algorithm called AnyBoost, which, like grafting (and stagewise additive modelling) can work with a variety of loss functions and regularization methods.

5.3 Implications for Regularization

We achieved best performance with the grafting algorithms using both a Ω_1 and a Ω_0 regularizer. It has been argued, (e.g., see Tibshirani, 1994), that Ω_1 regularization represents a good compromise between the needs of coefficient shrinkage for generalization performance, and the needs of subset selection for pragmatic reasons. However, as discussed in Section 2.1.2, these are really two separate goals and perhaps should be accounted for separately. Our experience is that if we attempted to use only Ω_1 regularization then it was often impossible to find a single value of λ_1 that produced good generalization performance without also incorporating many largely irrelevant features. It should be noted, though, that the Ω_q regularizer with $q = 1$ is the smallest q-value where the regularizer is convex.

5.4 Other Related Work

Das (2001) describes a technique for using boosting for feature selection. However he does not combine feature selection with model building in the integrated way we have described.

Jebara and Jaakkola (2000) also examine feature selection in a regularized risk minimization context for linear models. They define a prior describing likely values of weights, and choose one that tends to drive a number of weights to zero, in a similar fashion to l_1 regularization. In fact l_1 regularization can be viewed as being equivalent to a sharply peaked prior for weight values (Tibshirani, 1994).

Weston et al. (2000) describe a gradient descent method that can be used to select a subset of features non-linear for support vector machines. The input features are first weighted using a vector of weights σ. This vector is then optimized using gradient descent, to minimize a bound on the expected leave-one-out error, subject to a l_1 constraint on the magnitude of σ. This leads to some of the elements of σ being driven to zero.

SVMs are normally associated with l_2 regularization, but it is possible to formulate them using an l_1 norm as well. Fung and Mangasarian (2002) describe a gradient descent technique for rapidly optimizing such l_1 SVMs, which have the property that, as usual, many weights are driven to zero.

References

C.L. Blake and C.J. Merz. UCI repository of machine learning databases. `http://www.ics.uci.edu/~mlearn/MLRepository.html`, 1998. University of California, Irvine, Dept. of Information and Computer Science.

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

C. Chang and C. Lin. LIBSVM: A library for support vector machines. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`, 2001.

S. Das. Filters, wrappers and a boosting-based hybrid for feature selection. In *Proc. ICML*. Morgan Kauffmann, 2001.

Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. In *Machine Learning: Proc. 13th Int. Conf.*, pages 148–156. Morgan Kaufmann, 1996.

J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: A statistical view of boosting. *Annals of Statistics*, 28:337–307, 2000.

J. Friedman and W. Stuetzle. Projection pursuit regression. *Journal of the American Statistical Association*, 76:817–823, 1981.

G. Fung and O.L. Mangasarian. A feature selection Newton method for support vector machine classification. Technical Report 02-03, Data Mining Institute, Dept. of Computer Sciences, University of Wisconsin: Madison, Sept 2002.

F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7:219–269, 1995.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

A.E. Hoerl and R. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12:55–67, 1970.

T. Jebara and T. Jaakkola. Feature selection and dualities in maximum entropy discrimination. In *Proc. Int. Conf. on Uncertainity in Artificial Intelligence*, 2000.

K. Kira and L. Rendell. A practical approach to feature selection. In D. Sleeman and P. Edwards, editors, *Proc. Int. Conf. on Machine Learning*, pages 249–256. Morgan Kaufmann, 1992.

R. Kohavi and G.H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97:273–324, 1997.

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

L. Mason, J. Baxter, P.L. Bartlett, and M. Frean. Functional gradient techniques for combining hypotheses. In A.J. Smola, P.L Bartlett, B. Scölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 221–246. MIT Press, 2000.

J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7: 308–313, 1965.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.

R.E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

A.J. Smola, P.J. Bartlett, B. Schölkopf, and D. Schuurmans, editors. *Advances in Large Margin Classifiers*. MIT Press, 2000.

R. Tibshirani. Regression shrinkage and selection via the lasso. Technical report, Dept. of Statistics, University of Toronto, 1994.

J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggio, and V. Vapnik. Feature selection for SVMs. *Advances in Neural Information Processing Systems*, 13, 2000.