Learning Nonlinear Functions Using Regularized Greedy Forest

Rie Johnson and Tong Zhang

Abstract—We consider the problem of learning a forest of nonlinear decision rules with general loss functions. The standard methods employ boosted decision trees such as Adaboost for exponential loss and Friedman's gradient boosting for general loss. In contrast to these traditional boosting algorithms that treat a tree learner as a black box, the method we propose directly learns decision forests via fully-corrective regularized greedy search using the underlying forest structure. Our method achieves higher accuracy and smaller models than gradient boosting on many of the datasets we have tested on.

Index Terms—Boosting, decision tree, decision forest, ensemble, greedy algorithm

1 Introduction

Many application problems in machine learning require learning nonlinear functions from data. A popular method to solve this problem is through decision tree learning (such as CART [4] and C4.5 [23]), which has an important advantage for handling heterogeneous data with ease when different features come from different sources. This makes decision trees a popular "off-the-shelf" machine learning method that can be readily applied to any data without much tuning; in comparison, alternative algorithms such as neural networks require significantly more tuning. However, a disadvantage of decision tree learning is that it does not generally achieve the most accurate prediction performance, when compared to other methods. A remedy for this problem is through *boosting* [12], [15], [25], where one builds an additive model of decision trees by sequentially building trees one by one. In general "*boosted decision trees*" is regarded as the most effective off-the-shelf nonlinear learning method for a wide range of application problems.

In the boosted tree approach, one considers an additive model over multiple decision trees, and thus, we will refer to the resulting function as a *decision forest*. Other approach to learning decision forests include *bagging* and *random forests* [5], [6]. In this context, we may view boosted decision tree algorithms as methods to learn decision forests by applying a greedy algorithm (boosting) on top of a decision tree base learner. This indirect approach is sometimes referred to as a *wrapper* approach (in this case, wrapping boosting procedure over decision tree base learner); the boosting wrapper simply treats the decision tree base learner as a black box and it does not take advantage of the tree structure itself. The advantage of such a wrapper approach is that the underlying base learner can be changed to other procedures with the same wrapper; the disadvantage is that for any specific base learner which may have additional structure to explore, a generic wrapper might not be the optimal aggregator.

Due to the practical importance of boosted decision trees in applications, it is natural to ask whether one can design a more direct procedure that specifically learns decision forests without using a black-box decision tree learner under the wrapper. The purpose of doing so is that by directly taking advantage of the underlying tree structure, we shall be able to design a more effective algorithm for learning the final nonlinear decision forest. This paper attempts to address this issue, where we propose a direct decision forest learning algorithm called *Regularized Greedy Forest* or RGF. We are specifically interested in an approach that can handle general loss functions (while, for example, Adaboost is specific to a certain loss function), which leads to a wider range of applicability. An existing method with this property is *gradient boosting decision tree* (GBDT) [15]. We show that RGF can deliver better results than GBDT on a number of datasets we have tested on.

2 Problem Setup

We consider the problem of learning a single nonlinear function $h(\mathbf{x})$ on some input vector $\mathbf{x} = [\mathbf{x}[1], \ldots, \mathbf{x}[d]] \in \mathbb{R}^d$ from a set of training examples. In supervised learning, we are given a set of input vectors $X = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$ with labels $Y = [y_1, \ldots, y_m]$ (here m may not equal to n). Our training goal is to find a nonlinear prediction function $\hat{h}(\mathbf{x})$ from a function class $\mathcal{H}$ that minimizes a risk function

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \mathcal{L}(h(X), Y). \quad (1)$$

Here $\mathcal{H}$ is a pre-defined nonlinear function class, $h(X) = [h(\mathbf{x}_1), \ldots, h(\mathbf{x}_n)]$ is a vector of size n, and $\mathcal{L}(h, \cdot)$ is a general loss function of vector $h \in \mathbb{R}^n$.

- *R. Johnson is with RJ Research Consulting, Tarrytown, NY 10591 USA. E-mail: riejohnson@gmail.com.*
- *T. Zhang is with the Statistics Department, Rutgers University, Piscataway, NJ 08854 USA. E-mail: tongz@rci.rutgers.edu.*

Manuscript received 26 Sep. 2012; revised 10 May 2013; accepted 21 July 2013. Date of publication 20 Aug. 2013. Date of current version 29 Apr. 2014.
Recommended for acceptance by G. Lanckriet.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier 10.1109/TPAMI.2013.159

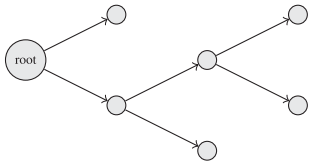

Fig. 1. Decision tree.

The loss function $\mathcal{L}(\cdot,\cdot)$ is given by the underlying problem. For example, for regression problems, we have $y_i \in \mathbb{R}$ and $m = n$. If we are interested in the conditional mean of y given $\mathbf{x}$, then the underlying loss function corresponds to least squares regression as follows:

$$\mathcal{L}(h(X), Y) = \sum_{i=1}^{n} (h(\mathbf{x}_i) - y_i)^2.$$

In binary classification, we assume that $y_i \in \{\pm 1\}$ and $m = n$. We may consider the logistic regression loss function as follows:

$$\mathcal{L}(h(X), Y) = \sum_{i=1}^{n} \ln(1 + e^{-h(\mathbf{x}_i) y_i}).$$

Another important problem that has drawn much attention in recent years is the pair-wise preference learning (for example, see [13], [19]), where the goal is to learn a nonlinear function $h(\mathbf{x})$ so that $h(\mathbf{x}) > h(\mathbf{x}')$ when $\mathbf{x}$ is preferred over $\mathbf{x}'$. In this case, $m = n(n-1)$, and the labels encode pair-wise preference as $y_{(i,i')} = 1$ when $\mathbf{x}_i$ is preferred over $\mathbf{x}_{i'}$, and $y_{(i,i')} = 0$ otherwise. For this problem, we may consider the following loss function that suffers a loss when $h(\mathbf{x}) \le h(\mathbf{x}') + 1$. That is, the formulation encourages the separation of $h(\mathbf{x})$ and $h(\mathbf{x}')$ by a margin when $\mathbf{x}$ is preferred over $\mathbf{x}'$:

$$\mathcal{L}(h(X), Y) = \sum_{(i,i'):y_{(i,i')}=1} \max(0, 1 - (h(\mathbf{x}_i) - h(\mathbf{x}_{i'})))^2.$$

Given data (X, Y) and a general loss function $\mathcal{L}(\cdot,\cdot)$ in (1), there are two basic questions to address for nonlinear learning. The first is the form of nonlinear function class $\mathcal{H}$, and the second is the learning/optimization algorithm. This paper achieves nonlinearity by using additive models of the form:

$$\mathcal{H} = \left\{ h(\cdot) : h(\mathbf{x}) = \sum_{j=1}^{K} \alpha_j b_j(\mathbf{x});\ \forall j, b_j \in \mathcal{C} \right\}, \tag{2}$$

where each $\alpha_j \in \mathbb{R}$ is a coefficient that can be optimized, and each $b_j(\mathbf{x})$ is by itself a nonlinear function (which we may refer to as a nonlinear basis function or an atom) taken from a base function class $\mathcal{C}$. The base function class typically has a simple form that can be used in the underlying algorithm. This work considers decision rules as the underlying base function class that is of the form

$$\mathcal{C} = \left\{ b(\cdot) : b(\mathbf{x}) = \prod_j \mathcal{I}(\mathbf{x}[i_j] \le t_j) \prod_k \mathcal{I}(\mathbf{x}[i_k] > t_k) \right\}, \tag{3}$$

where $\{(i_j, t_j), (i_k, t_k)\}$ are a set of (feature-index, threshold) pair, and $\mathcal{I}(x)$ denotes the indicator function: $\mathcal{I}(p) = 1$ if p is true; 0 otherwise. Decision rules can be graphically represented with a tree structure. In Fig. 1, each tree edge e is associated with a variable k_e and threshold t_e, and denotes a decision of the form $\mathcal{I}(\mathbf{x}[k_e] \le t_e)$ or $\mathcal{I}(\mathbf{x}[k_e] > t_e)$. Each node denotes a nonlinear decision rule of the form (3), which is the product of decisions along the edges leading from the root to this node.

Since the space of decision rules is rather large, for computational purposes, we have to employ a structured search over the set of decision rules. The optimization procedure we propose is a structured greedy search algorithm which we call regularized greedy forest (RGF). To introduce RGF, we first discuss pros and cons of the existing method for general loss, gradient boosting [15], in the next section.

Algorithm 1: Gradient Boosted Decision Tree (GBDT) [15]

$h_0(\mathbf{x}) \leftarrow \arg\min_\rho \mathcal{L}(\rho, Y)$
for $k = 1$ **to** K **do**
  $\tilde{Y}_k \leftarrow -\partial \mathcal{L}(h, Y)/\partial h|_{h=h_{k-1}(X)}$
  Build a J-leaf decision tree $T_k \leftarrow \mathcal{A}(X, \tilde{Y}_k)$ with leaf-nodes $\{b_{k,j}\}_{j=1}^{J}$
  for $j = 1$ **to** J **do**
  $\beta_{k,j} \leftarrow \arg\min_{\beta \in \mathbb{R}} \mathcal{L}(h_{k-1}(X) + \beta \cdot b_{k,j}(X), Y)$
  $h_k(\mathbf{x}) \leftarrow h_{k-1}(\mathbf{x}) + s \sum_{j=1}^{J} \beta_{k,j} \cdot b_{k,j}(\mathbf{x})$
  // s is a shrinkage parameter
end
return $h(\mathbf{x}) = h_K(\mathbf{x})$

3 Gradient Boosted Decision Tree

Gradient boosting is a method to minimize (1) with additive model (2) by assuming that there exists a nonlinear base learner (or oracle) $\mathcal{A}$ that satisfies Assumption 1.

Assumption 1. *A base learner for a nonlinear function class $\mathcal{A}$ is a regression optimization method that takes as input any pair $\tilde{X} = [\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_n]$ and $\tilde{Y} = [\tilde{y}_1, \ldots, \tilde{y}_n]$ and outputs a nonlinear function $\hat{g} = \mathcal{A}(\tilde{X}, \tilde{Y})$ that approximately solves the regression problem:*

$$\hat{b} \approx \arg\min_{b \in \mathcal{C}} \min_{\beta \in \mathbb{R}} \sum_{j=1}^{n} (\beta \cdot b(\tilde{\mathbf{x}}_j) - \tilde{y}_j)^2.$$

The gradient boosting method is a wrapper (boosting) algorithm that solves (1) with a base learner $\mathcal{A}$ defined above and additive model defined in (2). Of special interest for this paper and for general applications is the decision tree base learner, for which $\mathcal{C}$ is the class of J-leaf decision trees, with each node associated with a decision rule of the form (3). In order to take advantage of the fact that each element in $\mathcal{C}$ contains J (rather than one) decision rules, the gradient boosting method can be modified by adding a partially corrective update step that optimizes all J coefficients associated with the J decision rules returned by $\mathcal{A}$. This adaption was suggested by Friedman. We shall refer to this modification as gradient boosted decision tree (GBDT), and the details are listed in Algorithm 1.

Gradient boosting may be regarded as a functional generalization of gradient descent method $h_k \leftarrow h_{k-1} -$

$s_k \frac{\partial \mathcal{L}(h)}{\partial h}|_{h=h_{k-1}}$, where the shrinkage parameter s corresponds to the step size s_k in gradient descent, and $-\frac{\partial \mathcal{L}(h)}{\partial h}|_{h=h_{k-1}}$ is approximated using the regression tree output. The shrinkage parameter $s > 0$ is a tuning parameter that can affect performance, as noticed by Friedman. In fact, the convergence of the algorithm generally requires choosing $s\beta_k \to 0$ as indicated in the theoretical analysis of [29], which is also natural when we consider that it is analogous to step size in gradient descent. This is consistent with Friedman's own observation, who argued that in order to achieve good prediction performance (rather than computational efficiency), one should take as small a step size as possible (preferably infinitesimal step size each time), and the resulting procedure is often referred to as ϵ-boosting.

GBDT constructs a decision forest which is an additive model of K decision trees. The method has been very successful for many application problems, and its main advantage is that the method can automatically find nonlinear interactions via decision tree learning (which can easily deal with heterogeneous data), and it has relatively few tuning parameters for a nonlinear learning scheme (the main tuning parameters are the shrinkage parameter s, number of terminals per tree J, and the number of trees K). However, it has a number of disadvantages as well. First, there is no explicit regularization in the algorithm, and in fact, it is argued in [29] that the shrinkage parameter s plus early stopping (that is K) interact together as a form of regularization. In addition, the number of nodes J can also be regarded as a form of regularization. The interaction of these parameters in terms of regularization is unclear, and the resulting implicit regularization may not be effective. The second issue is also a consequence of using small step size s as implicit regularization. Use of small s could lead to a huge model, which is very undesirable as it leads to high computational cost of applications (i.e., making predictions). Third, the regression tree learner is treated as a black box, and its only purpose is to return J nonlinear terminal decision rule basis functions. This again may not be effective because the procedure separates tree learning and forest learning, and hence the algorithm itself is not necessarily the most effective method to construct the decision forest.

4 Fully-Corrective Greedy Update and Structured Sparsity Regularization

As mentioned above, one disadvantage of gradient boosting is that according to Friedman, in order to achieve good performance in practice, the shrinkage parameter s may need to be small, and he also argued for infinitesimal step size. This practical observation is supported by the theoretical analysis in [29] which showed that if we vary the shrinkage s for each iteration k as s_k, then for general loss functions with appropriate regularity conditions, the procedure converges as $k \to \infty$ if we choose the sequence s_k such that $\sum_k s_k|\beta_k| = \infty$ and $\sum_k s_k^2\beta_k^2 < \infty$. This condition is analogous to a related condition for the step size of gradient descent method which also requires the step-size to approach zero. Fully Corrective Greedy Algorithm is a modification of Gradient Boosting that can avoid the potential small step size problem. The procedure is described in Algorithm 2.

Algorithm 2: Fully-Corrective Gradient Boosting [27]

$h_0(\mathbf{x}) \leftarrow \arg\min_\rho \mathcal{L}(\rho, Y)$
for $k = 1$ **to** K **do**
  $\tilde{Y}_k \leftarrow -\partial \mathcal{L}(h, Y)/\partial h|_{h=h_{k-1}(X)}$
  $b_k \leftarrow \mathcal{A}(X, \tilde{Y}_k)$
  let $\mathcal{H}_k = \{\sum_{j=1}^k \beta_j b_j(\mathbf{x}) : \beta_j \in \mathbb{R}\}$
  $h_k(\mathbf{x}) \leftarrow \arg\min_{h \in \mathcal{H}_k} \mathcal{L}(h(X), Y)$
  // fully-corrective step
end
return $h(\mathbf{x}) = h_K(\mathbf{x})$

In gradient boosting or its variation with tree base learner of Algorithm 1, the algorithm only does a partial corrective step that optimizes either the coefficient of the last basis function b_k (or the last J coefficients). The main difference of the fully-corrective gradient boosting is the fully-corrective-step that optimizes all coefficients $\{\beta_j\}_{j=1}^k$ for basis functions $\{b_j\}_{j=1}^k$ obtained so far at each iteration k. It was noticed empirically that such fully-corrective step can significantly accelerate the convergence of boosting procedures [28]. This observation was theoretically justified in [27] where the following rate of convergence was obtained under suitable conditions: there exists a constant C_0 such that

$$\mathcal{L}(h_k(X), Y) \le \inf_{h \in \mathcal{H}} \left[\mathcal{L}(h(X), Y) + \frac{C_0 \|h\|_{\mathcal{C}}^2}{k} \right],$$

where C_0 is a constant that depends on properties of $\mathcal{L}(\cdot, \cdot)$ and the function class $\mathcal{H}$, and

$$\|h\|_{\mathcal{C}} = \inf \left\{ \sum_j |\alpha_j| : h(X) = \sum_j \alpha_j b_j(X);\ b_j \in \mathcal{C} \right\}.$$

In comparison, with only partial corrective optimization as in the original gradient boosting, no such convergence rate is possible. Therefore the fully-corrective step is not only intuitively sensible, but also important theoretically. The use of fully-corrective update (combined with regularization) automatically removes the need for using the undesirable small step s needed in the traditional gradient boosting approach.

However, such an aggressive greedy procedure will lead to quick overfitting of the data if not appropriately regularized (in gradient boosting, an implicit regularization effect is achieved by small step size s, as argued in [29]). Therefore we are forced to impose an explicit regularization to prevent overfitting.

This leads to the second idea in our approach, which is to impose explicit regularization via the concept of *structured sparsity* that has drawn much attention in recent years [1]–[3], [20]–[22]. The general idea of structured sparsity is that in a situation where a sparse solution is assumed, one can take advantage of the sparsity structure underlying the task. In our setting, we seek a *sparse* combination of decision rules (i.e., a compact model), and we have the forest

structure to explore, which can be viewed as graph sparsity structures. Moreover, the problem can be considered as a variable selection problem. Search over all nonlinear interactions (atoms) over $\mathcal{C}$ is computationally difficult or infeasible; one has to impose structured search over atoms. The idea of structured sparsity is that by exploring the fact that not all sparsity patterns are equally likely, one can select appropriate variables (corresponding to decision rules in our setting) more effectively by preferring certain sparsity patterns more than others. For our purpose, one may impose structured regularization and search to prefer one sparsity pattern over another, exploring the underlying forest structure.

This work considers the special but important case of learning a forest of nonlinear decision rules; although this may be considered as a special case of the general structured sparsity learning with an underlying graph, the problem itself is rich and important enough and hence requires a dedicated investigation. Specifically, we integrate this framework with specific tree-structured regularization and structured greedy search to obtain an effective algorithm that can outperform the popular and important gradient boosting method. In the context of nonlinear learning with graph structured sparsity, we note that a variant of boosting was proposed in [14], where the idea is to split trees not only at the leaf nodes, but also at the internal nodes at every step. However, the method is prone to overfitting due to the lack of regularization, and is computationally expensive due to the multiple splitting of internal nodes. We shall avoid such a strategy in this work.

5 Regularized Greedy Forest

The method we propose addresses the issues of the standard method GBDT described above by directly learning a decision forest via fully-corrective regularized greedy search. The key ideas discussed in Section 4 can be summarized as follows.

First, we introduce an explicit regularization functional on the nonlinear function h and optimize

$$\hat{h} = \arg\min_{h \in \mathcal{H}} [\mathcal{L}(h(X), Y) + \mathcal{R}(h)] \tag{4}$$

instead of (1). In particular, we define regularizers that explicitly take advantage of individual tree structures.

Second, we employ *fully-corrective greedy* algorithm which repeatedly re-optimizes the coefficients of *all* the decision rules obtained so far while rules are added into the forest by greedy search. Although such an aggressive greedy procedure could lead to quick overfitting if not appropriately regularized, our formulation includes explicit regularization to avoid overfitting and the problem of huge models caused by small s.

Third, we perform structured greedy search *directly* over forest nodes based on the forest structure (graph sparsity structure) employing the concept of structured sparsity. At the conceptual level, our nonlinear function $h(\mathbf{x})$ is explicitly defined as an additive model on forest nodes (rather than trees) consistent with the underlying forest structure. In this framework, it is also possible to build a forest by growing multiple trees simultaneously.

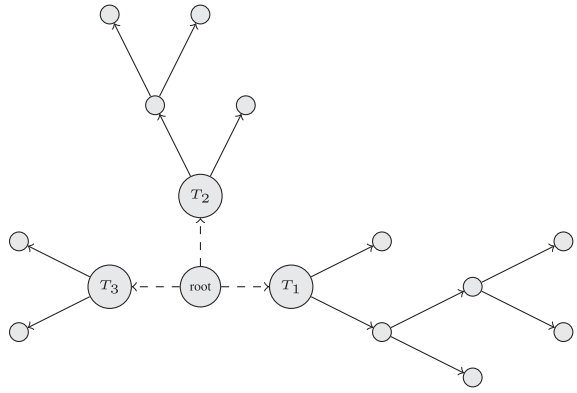

Fig. 2. Decision forest.

Before going into more detail, we shall introduce some definitions and notation that allow us to formally define the underlying formulations and procedures.

5.1 Definitions and Notation

A forest is an ensemble of multiple decision trees $T_1, \ldots, T_K$. The forest shown in Fig. 2 contains three trees T_1, T_2, and T_3. Each tree edge e is associated with a variable k_e and threshold t_e, and denotes a decision of the form $\mathcal{I}(\mathbf{x}[k_e] \leq t_e)$ or $\mathcal{I}(\mathbf{x}[k_e] > t_e)$. Each node denotes a nonlinear decision rule of the form (3), which is the product of decisions along the edges leading from the root to this node.

Mathematically, each node v of the forest is associated with a decision rule of the form

$$b_v(\mathbf{x}) = \prod_j \mathcal{I}(\mathbf{x}[i_j] \leq t_{i_j}) \prod_k \mathcal{I}(\mathbf{x}[i_k] > t_{i_k}),$$

which serves as a basis function or atom for the additive model considered in this paper. Note that if v_1 and v_2 are the two children of v, then $b_v(\mathbf{x}) = b_{v_1}(\mathbf{x}) + b_{v_2}(\mathbf{x})$. This means that any internal node is redundant in the sense that an additive model with basis functions $b_v(\mathbf{x}), b_{v_1}(\mathbf{x}), b_{v_2}(\mathbf{x})$ can be represented as an additive model over basis functions $b_{v_1}(\mathbf{x})$ and $b_{v_2}(\mathbf{x})$. Therefore it can be shown that an additive model over all tree nodes always has an *equivalent model* (equivalent in terms of output) over leaf nodes only. This property is important for computational efficiency because it implies that we only have to consider additive models over leaf nodes.

Let $\mathcal{F}$ represent a forest, and each node v of $\mathcal{F}$ is associated with (b_v, α_v). Here b_v is the basis function that this node represents; α_v is the *weight* or coefficient assigned to this node. The additive model of this forest $\mathcal{F}$ considered in this paper is: $h_{\mathcal{F}}(\mathbf{x}) = \sum_{v \in \mathcal{F}} \alpha_v b_v(\mathbf{x})$ with $\alpha_v = 0$ for any internal node v. In this setting, the regularized loss in (4) is a function of decision forest:

$$\mathcal{Q}(\mathcal{F}) = \mathcal{L}(h_{\mathcal{F}}(X), Y) + \mathcal{R}(h_{\mathcal{F}}). \tag{5}$$

5.2 Algorithmic Framework

The training objective of RGF is to build a forest that minimizes $\mathcal{Q}(\mathcal{F})$ defined in (5). Since the exact optimum solution

Algorithm 3: Regularized greedy forest framework

1 $\mathcal{F} \leftarrow \{\}$.
repeat
2 $\quad \mathcal{F} \leftarrow$ the optimum forest that minimizes $\mathcal{Q}(\mathcal{F})$ among all the forests that can be obtained by applying one step of structure-changing operation to the current forest $\mathcal{F}$.
3 $\quad$ **if** *some criterion is met* **then** optimize the leaf weights in $\mathcal{F}$ to minimize loss $\mathcal{Q}(\mathcal{F})$.
until *some exit criterion is met*;
Optimize the leaf weights in $\mathcal{F}$ to minimize loss $\mathcal{Q}(\mathcal{F})$.
return $h_{\mathcal{F}}(\mathbf{x})$

is difficult to find, we *greedily* select the *basis functions* and optimize the *weights*. At a high level, we may summarize RGF in a generic algorithm in Algorithm 3. It essentially has two main components as follows.

- Fix the *weights*, and change the *structure* of the forest (which changes basis functions) so that the loss $\mathcal{Q}(\mathcal{F})$ is reduced the most (Line 2).
- Fix the *structure* of the forest, and change the *weights* so that loss $\mathcal{Q}(\mathcal{F})$ is minimized (Line 3).

5.3 Specific Implementation

There may be more than one way to instantiate useful algorithms based on Algorithm 3. Below, we describe what we found effective and efficient.

5.3.1 Search for the Optimum Structure Change (Line 2)

For computational efficiency, we only allow the following two types of operations in the search strategy:

- to split an existing leaf node,
- to start a new tree (i.e., add a new stump to the forest).

The operations include assigning weights to new leaf nodes and setting zero to the node that was split. Search is done with the weights of all the existing leaf nodes fixed, by repeatedly evaluating the maximum loss reduction of all the possible structure changes. When it is prohibitively expensive to search the entire forest (and that is often the case with practical applications), we limit the search to the most recently-created t trees with the default choice of $t = 1$. This is the strategy in our current implementation. For example, Fig. 3 shows that at the same stage as Fig. 2, we may either consider splitting one of the leaf nodes marked with symbol X or grow a new tree T_4 (split T_4's root).

Note that RGF does not require the tree size parameter needed in GBDT. With RGF, the size of each tree is automatically determined as a result of minimizing the regularized loss.

a) Computation Consider the evaluation of loss reduction by splitting a node associated with (b, α) into the nodes associated with $(b_{u_1}, \alpha+\delta_1)$ and $(b_{u_2}, \alpha+\delta_2)$, and let us write $\tilde{\mathcal{F}}(\delta_1, \delta_2)$ for the new tree. Then the model associated with

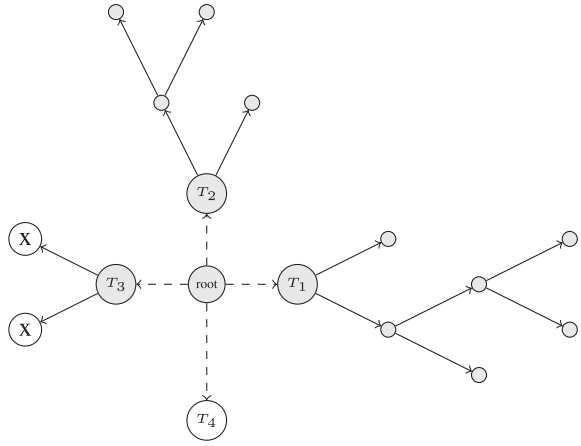

Fig. 3. Decision forest splitting strategy (we may either split a leaf in T_3 or start a new tree T_4).

the new forest $\tilde{\mathcal{F}}(\delta_1, \delta_2)$ can be written as:

$$h_{\tilde{\mathcal{F}}(\delta_1,\delta_2)}(\mathbf{x}) = h_{\mathcal{F}}(\mathbf{x}) - \alpha \cdot b(\mathbf{x}) + \sum_{k=1}^{2}(\alpha + \delta_k) b_{u_k}(\mathbf{x})$$
$$= h_{\mathcal{F}}(\mathbf{x}) + \sum_{k=1}^{2} \delta_k \cdot b_{u_k}(\mathbf{x}). \quad (6)$$

Recall that our additive models are over leaf nodes only. The node that was split is no longer leaf and therefore $\alpha \cdot b(\mathbf{x})$ is removed from the model. The second equality is from $b(\mathbf{x}) = b_{u_1}(\mathbf{x}) + b_{u_2}(\mathbf{x})$ due to the parent-child relationship. Note that, for the purpose of finding the optimum forest, we let $\tilde{\mathcal{F}}(\delta_1, \delta_2)$ go through all the possible forests that can be obtained by splitting one leaf node of the current forest $\mathcal{F}$. However, our immediate goal here is to find $\arg\min_{\delta_1,\delta_2} \mathcal{Q}(\tilde{\mathcal{F}}(\delta_1, \delta_2))$.

Actual computation depends on the loss function and the regularization term. In general, there may not be an analytic solution for this optimization problem, whereas we need to find the solution in an inexpensive manner as this computation is repeated frequently. For fast computation, one may employ gradient-descent approximation as used in gradient boosting. However, the sub-problem we are looking at is simpler, and thus instead of the simpler gradient descent approximation, we perform one Newton step which is more accurate; namely, we obtain the approximately optimum $\hat{\delta}_k$ $(k = 1, 2)$ as:

$$\hat{\delta}_k = -\frac{\mathcal{Q}'_{\delta_k}(\tilde{\mathcal{F}}(\delta_1, \delta_2))}{\mathcal{Q}''_{\delta_k}(\tilde{\mathcal{F}}(\delta_1, \delta_2))}|_{\delta_1=0,\delta_2=0} \ ,$$

where $\mathcal{Q}'_{\delta_k}(\cdot)$ and $\mathcal{Q}''_{\delta_k}(\cdot)$ are the first and second partial derivatives of $\mathcal{Q}(\cdot)$ with respect to δ_k $(k = 1, 2)$. For example, with square loss and L_2 regularization penalty, i.e., $\mathcal{Q}(\mathcal{F}) = \sum_{i=1}^{n}(h_{\mathcal{F}}(\mathbf{x}_i) - y_i)^2 + \lambda \sum_{v\in\mathcal{F}} \alpha_v^2$ with a constant λ, we have

$$\hat{\delta}_k = \frac{\sum_{b_{u_k}(\mathbf{x}_i)=1}(y_i - h_{\mathcal{F}}(\mathbf{x}_i)) - n\lambda\alpha}{\sum_{b_{u_k}(\mathbf{x}_i)=1} 1 + n\lambda} \ ,$$

which is the exact optimum for the given split.

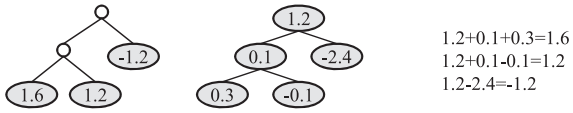

Fig. 4. Example of equivalent models.

5.3.2 Weight Optimization/Correction (Line 3)

With the basis functions fixed, the weights can be optimized using a standard procedure if the regularization penalty is standard (e.g., L_1- or L_2-penalty). In our implementation we perform coordinate descent, which iteratively goes through the basis functions and in each iteration updates the weights by a Newton step with a small step size:

$$\alpha_v \leftarrow \alpha_v - \eta \cdot \frac{\mathcal{Q}'_{\delta_v}(\mathcal{F}(\delta_v))}{\mathcal{Q}''_{\delta_v}(\mathcal{F}(\delta_v))}\Big|_{\delta_v=0}, \tag{7}$$

where δ_v is the additive change to α_v.

Since the initial weights of new leaf nodes set in Line 2 are approximately optimal at the moment, it is not necessary to perform weight correction in every iteration, which is relatively expensive. Based on the preliminary experiments using synthesized data, we found that correcting the weights every time k new leaf nodes are added works well. Obviously, k's setting (the interval between fully-corrective updates) should not be extreme – if k is extremely large, it would be equivalent to doing fully-corrective update just once in the end and would lose the benefit of the interleaving approach; if k is extremely small (e.g., $k = 1$), it would slow down training. Empirically, as long as k is not an extreme value, the choice of k is not crucial. Therefore, we simply fixed k to 100 in all of our experiments including the competitions we won, described later.

5.4 Tree-structured Regularization

Explicit regularization is a crucial component of this framework. To simplify notation, we define regularizers over a single tree. The regularizer over a forest can be obtained by adding the regularizers described here over all the trees. Therefore, suppose that we are given a tree T with an additive model over leaf nodes:

$$h_T(\mathbf{x}) = \sum_{v \in T} \alpha_v b_v(\mathbf{x}) \quad , \quad \alpha_v = 0 \text{ for } v \notin L_T$$

where L_T denotes the set of leaf nodes in T.

To consider useful regularizers, first recall that for any additive model over leaf nodes only, there always exist *equivalent models* over all the nodes of the same tree that produce the same output. More precisely, let $A(v)$ denote the set of ancestor nodes of v and v itself, and let $T(\beta)$ be a tree that has the same topological structure as T but whose node weights $\{\alpha_v\}$ are replaced by $\{\beta_v\}$. Then we have

$$\forall u \in L_T\colon \sum_{v \in A(u)} \beta_v = \alpha_u \quad \Leftrightarrow \quad h_{T(\beta)}(\mathbf{x}) \equiv h_T(\mathbf{x})$$

as illustrated in Fig. 4. Our basic idea is that it is natural to give the same regularization penalty to all equivalent models defined on the same tree topology. One way to define a regularizer that satisfies this condition is to choose a model of some desirable properties as the unique representation for all the equivalent models and define the regularization penalty based on this unique representation. This is the high-level strategy we take. That is, we consider the following form of regularization:

$$\mathcal{R}(h_T) = \sum_{v \in T(\beta)} r(v) : \; h_{T(\beta)}(\mathbf{x}) \equiv h_T(\mathbf{x}) \; .$$

Here node v includes both internal and leaf nodes; the additive model $h_{T(\beta)}(\mathbf{x})$ serves as the unique representation of the set of equivalent models; and $r(v)$ is a penalty function of v's weight β_v and v's attributes such as the node depth. Each β_v is a function of given leaf weights $\{\alpha_u\}_{u \in L_T}$, though the function may not be a closed form. Since regularizers in this form utilize the entire tree including its topological structure, we call them *tree-structured regularizers*. Below, we describe three tree-structured regularizers using three distinct unique representations.

5.4.1 L_2 Regularization on Leaf-only Models

The first regularizer we introduce simply chooses the given leaf-only model as the unique representation and uses the standard L_2 regularization. This leads to a regularization term:

$$\mathcal{R}(h_T) = \lambda \sum_{v \in T} \alpha_v^2/2 = \lambda \sum_{v \in L_T} \alpha_v^2/2$$

where λ is a constant for controlling the strength of regularization. A desirable property of this unique representation is that among the equivalent models, the leaf-only model is often (but not always[1]) the one with the smallest number of basis functions, i.e., the most sparse.

5.4.2 Minimum-penalty Regularization

Another approach we consider is to choose the model that minimizes some penalty as the unique representative of all the equivalent models, as it is the most preferable model according to the defined penalty. We call this type of regularizer a *min-penalty regularizer*. In the following min-penalty regularizer, the complexity of a basis function is explicitly regularized via the node depth.

$$\mathcal{R}(h_T) = \lambda \cdot \min_{\{\beta_v\}} \left\{ \sum_{v \in T} \frac{1}{2} \gamma^{d_v} \beta_v^2 \colon h_{T(\beta)}(\mathbf{x}) \equiv h_T(\mathbf{x}) \right\} \; . \tag{8}$$

Here d_v is the depth of node v, which is the distance from the root, and γ is a constant. A larger $\gamma > 1$ penalizes deeper nodes more severely, which are associated with more complex decision rules, and we assume that $\gamma \geq 1$.

a) Computation To derive an algorithm for computing this regularizer, first we introduce auxiliary variables $\{\bar{\beta}_v\}_{v \in T}$, recursively defined as:

$$\bar{\beta}_{o_T} = \beta_{o_T} \; , \quad \bar{\beta}_v = \beta_v + \bar{\beta}_{p(v)} \; ,$$

where o_T is T's root, and $p(v)$ is v's parent node, so that we have

$$h_{T(\beta)} \equiv h_T \Leftrightarrow \forall v \in L_T . \left[\bar{\beta}_v = \alpha_v\right] \; , \tag{9}$$

1. For example, consider a leaf-only model on a stump whose two sibling leaf nodes have the same weight $\alpha \neq 0$. Its equivalent model with the fewest basis functions (with nonzero coefficients) is the one whose weight is α on the root and zero on the two leaf nodes.

Algorithm 4: Min-penalty regularization with (8)

$$\textbf{for } v \in T \textbf{ do } \bar{\beta}_{v,0} \leftarrow \begin{cases} \alpha_v & v \in L_T \\ 0 & v \notin L_T \end{cases}$$
$$\textbf{for } i = 1 \textbf{ to } m \textbf{ do}$$
$$\quad \textbf{for } v \in L_T \textbf{ do } \bar{\beta}_{v,i} \leftarrow \alpha_v$$
$$\quad \textbf{for } v \notin L_T \textbf{ do } \bar{\beta}_{v,i} \leftarrow \begin{cases} \frac{\bar{\beta}_{p(v),i-1} + \sum_{p(w)=v} \gamma \bar{\beta}_{w,i-1}}{1+2\gamma} & v \neq o_T \\ \frac{\sum_{p(w)=v} \gamma \bar{\beta}_{w,i-1}}{1+2\gamma} & v = o_T \end{cases}$$
end
return $\{\bar{\beta}_{v,m}\}$

and (8) can be rewritten as:

$$\mathcal{R}(h_T) = \lambda \cdot \min_{\{\bar{\beta}_v\}} \left\{ f(\{\bar{\beta}_v\}) : \forall v \in L_T . [\ \bar{\beta}_v = \alpha_v] \right\}$$
$$\text{where } f(\{\bar{\beta}_v\}) = \sum_{v \neq o_T} \gamma^{d_v} (\bar{\beta}_v - \bar{\beta}_{p(v)})^2 / 2 + \bar{\beta}_{o_T}^2 / 2 \ .$$

Setting f's partial derivatives to zero, we obtain that at the optimum,

$$\forall v \notin L_T : \bar{\beta}_v = \begin{cases} \frac{\bar{\beta}_{p(v)} + \sum_{p(w)=v} \gamma \bar{\beta}_w}{1+2\gamma} & v \neq o_T \\ \frac{\sum_{p(w)=v} \gamma \bar{\beta}_w}{1+2\gamma} & v = o_T \end{cases} , \qquad (10)$$

i.e., essentially, $\bar{\beta}_v$ is the weighted average of the neighbors. This naturally leads to an iterative algorithm summarized in Algorithm 4.

5.4.3 Min-penalty Regularization with Sum-to-zero Sibling Constraints

Another regularizer we introduce is based on the same basic idea as above but is computationally simpler. We add to (8) the constraint that the sum of weights for every sibling pair must be zero,

$$\mathcal{R}(h_T) = \lambda \cdot \min_{\{\beta_v\}} \left\{ \sum_{v \in T} \gamma^{d_v} \beta_v^2 / 2 : h_{T(\beta)}(\mathbf{x}) \equiv h_T(\mathbf{x}) \ ; \right.$$
$$\left. \forall v \notin L_T . \left[\sum_{p(w)=v} \beta_w = 0 \right] \right\} ,$$

as illustrated in Fig. 5. The intuition behind this sum-to-zero sibling constraints is that less redundant models are preferable and that the models are the *least redundant* when branches at every internal node lead to completely opposite actions, namely, 'adding x to' versus 'subtracting x from' the output value.

Using the auxiliary variables $\{\bar{\beta}_v\}$ as defined above, it is straightforward to show that any set of equivalent models has exactly one model that satisfies the sum-to-zero sibling constraints. This model can be obtained through the following recursive computation on the auxiliary variables:

$$\bar{\beta}_v = \begin{cases} \alpha_v & v \in L_T \\ \sum_{p(w)=v} \bar{\beta}_w / 2 & v \notin L_T \end{cases} .$$

5.5 Extension of Regularized Greedy Forest

We introduce an extension, which allows the process of forest growing and the process of weight correction to have different regularization parameters. The motivation is that the regularization parameter optimum for weight correction may not necessarily be optimal for forest growing, as the former is fully-corrective and therefore global whereas the latter is greedy and is localized to the leaf nodes of interest. Therefore, it is sensible to allow distinct regularization parameters for these two distinct processes. Furthermore, there could be an extension that allows one to change the strength of regularization as the forest grows, though we did not pursue this direction in the current work.

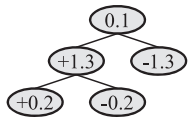

Fig. 5. Example of sum-to-zero sibling model.

6 Experiments

This section reports empirical studies of RGF in comparison with GBDT and several tree ensemble methods. In particular, we report the results of entering competitions using RGF. Our implementation of RGF used for the experiments is available from http://riejohnson.com/rgf_download.html.

For clarity, the experiments focus on regression tasks and binary classification tasks. However, note that since the method is designed for optimizing general loss, there are other applicable tasks. For example, multi-class categorization can be performed by combining binary classification tasks in the "one-vs-others" or other encoding schemes, as is commonly done with the methods that optimize general loss. In addition, there are multi-class training methods for, for example, GBDT and AdaBoost, and RGF can be extended similarly.

6.1 On the Synthesized Datasets Controlling Complexity of Target Functions

First we study the performance of the methods in relation to the complexity of target functions using synthesized datasets. To synthesize datasets, first we defined the target function by randomly generating 100 q-leaf regression trees; then we randomly generated data points and applied the target function to them to assign the output/target values. In more detail, (1) generate 100 trees of q leaf nodes by randomly choosing a node to split and also randomly choosing features and threshold values for split; (2) assign weights $0, 1, \ldots, q$ to the leaf nodes of each tree; (3) generate data points of 10 dimensions so that the components distribute uniformly over $\{0, 1, \ldots, 99\}$; (4) apply the tree ensemble generated above to each data point. The obtained value is an interim target value. To generate regression problems, normalize the interim target value by subtracting the mean and dividing by the standard deviation. Note that a larger tree size q makes the target function more complex.

The results shown in Table 1 are in the root mean square error (RMSE) averaged over three runs. In each run, randomly chosen 2K data points were used for training and the number of test data points was 20K. The parameters were chosen by 2-fold cross validation on the training data. Since the task is regression, the loss function for RGF and GBDT

TABLE 1
Regression Results on Synthesized Datasets

	RGF-L_2	GBDT	RGF min-penalty			
			w/sib. constraint		w/o sib. constraint	
5-leaf data	**0.2200**	0.2597	0.1885	(0.0315)	0.1890	(0.0310)
10-leaf data	**0.3480**	0.3968	0.3270	(0.0210)	0.3266	(0.0214)
20-leaf data	**0.4578**	0.4942	0.4545	(0.0033)	0.4538	(0.0040)

RMSE. Average of 3 runs, each of which used randomly-drawn 2K training data points. RGF-L_2 outperforms GBDT. RGF min-penalty (with or without the sibling constraint) further improves accuracy; the numbers in parentheses are accuracy improvements over RGF-L_2.

were set to square loss. RGF used here is the most basic version, which does L_2 regularization with one parameter λ for both forest growing and weight correction. λ was chosen from $\{1, 0.1, 0.01\}$. For GBDT, we used R package `gbm`[2] [24]. The tree size (in terms of the number of leaf nodes) and the shrinkage parameter were chosen from $\{5, 10, 15, 20, 25\}$ and $\{0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}$, respectively. Table 1 shows that RMSE achieves smaller error than GBDT on all types of datasets.

RGF with min-penalty regularizer with the sibling constraints further improves RMSE over RGF-L_2 by 0.0315, 0.0210, 0.0033 on the 5-leaf, 10-leaf, and 20-leaf synthesized datasets, respectively. RGF with min-penalty regularizer without the sibling constraints also achieved the similar performances. Based on the amount of improvements, min-penalty regularizer appears to be more effective on simpler targets. Fig. 6 plots RMSE in relation to the model size in terms of the number of basis functions or leaf nodes. RGF produces better RMSE at all the model sizes; in other words, to achieve similar RMSE, RGF requires a smaller model than GBDT.

The synthesized datasets used in this section are provided with the RGF software.

6.2 Regression and 2-way Classification Tasks on the Real-world Datasets

The first suite of real-world experiments use relatively small training data of 2K data points to facilitate experimenting with a wide variety of datasets. The criteria of data choice were (1) having over 5000 data points in total to ensure a decent amount of test data and (2) to cover a variety of domains. The datasets and tasks are summarized in Table 2. All except Houses (downloaded from http://lib.stat.cmu.edu) are from the UCI repository [11]. All the results are the average of 3 runs, each of which used randomly-drawn 2K training data points. For multi-class data, binary tasks were generated as in Table 2. The official test sets were used as test sets if any (Letter, Adult, and MSD). For relatively large Nursery and Houses, 5K data points were held out as test sets. For relatively small Musk and Waveform, in each run, 2K data points were randomly chosen as training sets, and the rest were used as test sets (4598 data points for Musk and 3000 for Waveform). The exact partitions of training and test data are provided with the RGF software.

All the parameters were chosen by 2-fold cross validation on the training data. The RGF tested here is RGF-L_2 with the extension in which the processes of forest growing and weight correction can have regularization parameters of different values, which we call λ_g ('g' for 'growing') and λ, respectively. The value of λ was chosen from $\{10, 1, 0.1, 0.01\}$ with square loss, and from $\{10, 1, 0.1, 0.01, 1e-10, 1e-20, 1e-30\}$ with logistic loss and exponential loss. λ_g was chosen from $\{\lambda, \frac{\lambda}{100}\}$. The tree size for GBDT was chosen from $\{5, 10, 15, 20, 25\}$, and the shrinkage parameter was from $\{0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}$.

In addition to GBDT, we also tested two other tree ensemble methods: *random forests* [7] and *Bayesian additive regression trees (BART)* [10]. We used the R package `randomForest` [7] and performed random forest training with the number of randomly-drawn features in $\{\frac{d}{4}, \frac{d}{3}, \frac{d}{2}, \frac{3d}{5}, \frac{7d}{10}, \frac{4d}{5}, \frac{9d}{10}, \sqrt{d}\}$, where d is the feature dimensionality; the number of trees set to 1000; and other parameters set to default values. BART is a Bayesian approach to tree ensemble learning. The motivation to test BART was that it shares some high-level strategies with RGF such as explicit regularization and non-black-box approaches to tree learners. We used the R package `BayesTree` [9] and chose the parameter k, which adjusts the degree of regularization, from $\{1, 2, 3\}$.

Table 3 shows the regression results in RMSE. RGF achieves lower error than all others.

Table 4 shows binary classification results in accuracy(%). RGF achieves the best performance on the three datasets, whereas GBDT achieves the best performance on only one dataset.

The min-penalty regularizer was found to be effective on Musk, improving the accuracy of RGF-L_2 with square loss from 97.83% to 98.39%, but it did not improve performance on other datasets. Based on the synthesized data experiments in the previous section, we presume that this

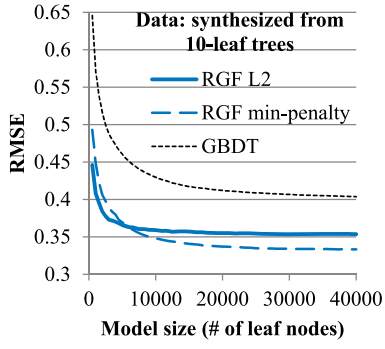

Fig. 6. Regression results in relation to model size. One particular run on the data synthesized from 10-leaf trees.

2. In the rest of the paper, `gbm` was used for the GBDT experiments unless otherwise specified.

TABLE 2
Real-World Datasets

Name	Dim	Regression tasks
CT slices	384	Target: relative location of CT slices
California Houses	6	Target: log(median house price)
YearPredictionMSD	90	Target: year when the song was released
Name	**Dim**	**Binary classification tasks**
Adult	14(168)	Is income $>$ \$50K?
Letter	16	A-M vs N-Z
Musk	166	Musk or not
Nursery	8(24)	"Special priority" or not
Waveform	40	Class2 vs. Class1&3

We report the average of 3 runs, each of which uses 2K training data points. The numbers in parentheses indicate the dimensionality after converting categorical attributes to indicator vectors.

TABLE 3
Regression Results

	RGF-L_2	GBDT	RandomF.	BART
CT slices	**7.2037**	7.6861	*7.5029*	8.6006
California Houses	**0.3417**	0.3454	*0.3453*	0.3536
YearPredictionMSD	**9.5523**	9.6846	9.9779	*9.6126*

*RMSE. Average of 3 runs, each of which used randomly-drawn 2K training data points. The best and second best results are in **bold** and italic, respectively.*

is because the target functions underlying these real-world datasets are mostly complex.

On the binary classification tasks, AdaBoost with three configurations was also tested[3]: AdaBoost with decision stumps both with and without unregularized fully-corrective weight update to minimize exponential loss as post processing, and a publicly available AdaBoost implementation with tree ensembles. For the third configuration (labeled as 'AdaBoost reg.' in the table) we used the R package `ada` and set the parameter "cp", which controls the degree of regularization of the tree learner, from $\{0.1, 0.01, 0.001\}$ by cross validation. AdaBoost is a meta learner known to produce highly accurate classifiers, and in particular, AdaBoost with decision stumps has been intensively studied. The unregularized fully-corrective weight update of AdaBoost is discussed in the Appendix of [26].

As shown in Table 4, the accuracy of AdaBoost with decision stumps turned out to be generally poor, for example, the accuracy on Letter is about 12% lower than the other methods. Among the three configurations of AdaBoost, 'AdaBoost reg.' is the most competitive, which indicates that the success of the meta learner AdaBoost relies on the appropriate regularization of the base learner. Apparently, the degree of regularization implicitly provided by restricting the base learner to decision stumps is not the optimum on the three (Letter, Musk, and Nursery) out of five datasets, causing accuracy to degrade by 1%, 6%, and 12% compared with RGF. The unregularized fully-corrective update (as suggested in [26]) was found to degrade accuracy on all the datasets. This is not surprising because it is known that the exponential loss used in Adaboost is prone to overfitting, especially without regularization. These AdaBoost results provide further support for

3. Note that AdaBoost cannot be used for regression tasks since the loss function associated with AdaBoost is specifically the exponential loss.

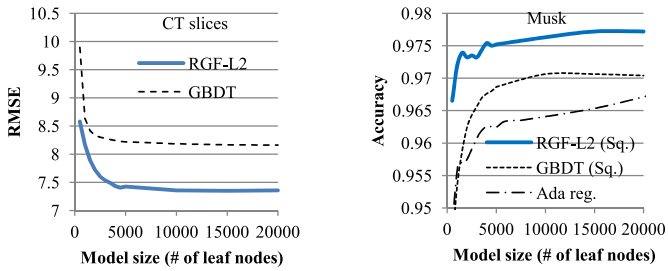

Fig. 7. RMSE/Accuracy in relation to model size. One particular run on the representative datasets.

our methodology of incorporating fully-corrective weight updates with explicit regularization.

Regarding model sizes, we noticed that random forests and BART require far larger models than RGF to achieve the performances shown in the Table 4; for example, all the BART's models consist of over 400K leaf nodes whereas the RGF models reach the best performance with 20K leaf nodes or fewer. Similarly, AdaBoost with stumps requires far larger models (200K leaf nodes) on Letter and Musk and yet it achieves lower accuracy than RGF. Fig. 7 shows the RMSE/accuracy of RGF and GBDT (and AdaBoost for classification) in relation to the model sizes on the representative datasets. Similar to Fig. 6 (on the synthesized data), RGF is more accurate than GBDT (and AdaBoost) at all model sizes; in other words, to achieve similar accuracy, RGF only requires a smaller model than GBDT.

6.3 GBDT with Post Processing of Fully-Corrective Updates

A *two-stage* approach was proposed in [17], [18][4] that, in essence, first performs GBDT to learn basis functions and then fits their weights with L_1 penalty in the post-processing stage. Note that by contrast RGF generates basis functions and optimizes their weights in an *interleaving* manner so that fully-corrected weights can influence generation of the next basis functions.

Table 5 shows the performance results of the two-stage approach on the regression and 2-way classification tasks described in Section 6.2. As is well known, L_1 regularization has "feature selection" effects, assigning zero weights to more and more features with stronger regularization. After performing GBDT[5] with the parameter chosen by cross validation on the training data, we used the R package `glmnet` [16] to compute the entire L_1 path in which the regularization parameter goes down gradually and thus more and more basis functions obtain nonzero weights, and chose the L_1 regularization parameter by 3-fold cross validation using the cross validation functionality of `glmnet`. In the table, the numbers in the parentheses compare the sizes of the models with and without post-processing of the two-stage approach; for example, on Adult, the size of the model after post-processing is 13.5% compared with

4. Although [18] discusses various techniques regarding rules, we focus on the aspect of the two-stage approach which [18] derives from [17], since it is the most relevant portion to our work due to its contrast with our interleaving approach.

5. We used our own implementation of GBDT for this purpose, as `gbm` does not have the functionality to output the features generated by tree learning.

TABLE 4
Binary Classification Results

	RGF-L_2			GBDT			Random	BART	AdaBoost		
	Sq.	Log.	Expo.	Sq.	Log.	Expo.	forests		stumps	w/full	reg.
Adult	85.62	85.63	85.20	85.62	**85.72**	*85.75*	85.29	85.62	85.51	85.10	84.68
Letter	**92.50**	92.19	*92.48*	91.20	91.72	91.93	90.33	85.06	80.65	79.67	92.12
Musk	97.83	**97.91**	*97.83*	97.14	96.79	97.27	96.23	95.56	96.91	95.63	97.13
Nursery	98.63	**99.97**	*99.95*	98.13	99.90	99.85	97.44	99.12	93.31	93.11	99.51
Waveform	90.28	90.21	90.06	89.56	89.97	90.18	90.20	*90.49*	90.19	88.08	**90.52**

Accuracy (%). Average of 3 runs, each of which used randomly-drawn 2K training data points. "Sq.", "Log.", "Expo." stand for the square loss, logistic loss, and exponential loss, respectively. "w/full" is AdaBoost with stumps with fully-corrective update as post-processing. "Reg." is AdaBoost with the regularized tree learner. The best and second best results are in ***bold*** *and italic, respectively.*

the GBDT model without post-processing and accuracy is 0.52% lower. The results show that the L_1 post processing makes the models smaller, but it noticeably degrades accuracy on all but one dataset. We view that for achieving better accuracy, RGF's interleaving approach has a clear advantage.

6.4 RGF in the Competitions

To further test RGF in practical settings, we entered three machine learning competitions (listed in Table 6) and obtained good results. The competitions were held in the "Netflix Prize" style. That is, participants submit predictions on the test data (whose labels are not disclosed) and receive performance results on the *public* portion of the test data as feedback on the *public Leaderboard*. The goal is to maximize the performance on the *private* portion of the test data, and neither the private score nor the standing on the *private Leaderboard* is disclosed until the competition ends.

In all of the three competitions, RGF produced more accurate models than GBDT. This demonstrates that RGF can achieve performance superior to GBDT even in the most competitive situation.

Bond Price Prediction

We were awarded with the First Place Prize in Benchmark Bond Trade Price Challenge (www.kaggle.com/c/benchmark-bond-trade-price-challenge). The task was to predict bond trade prices based on the information such as past trade recordings.

The evaluation metric was weighted mean absolute error, $\sum_i w_i|y_i - f(\mathbf{x}_i)|$, with the weights set to be larger for the bonds whose price prediction is considered to be harder. We trained RGF with L1-L2 hybrid loss [8], $\sqrt{1+r^2} - 1$ where r is the residual, which behaves like L_2 loss when $|r|$ is small and L_1 when $|r|$ is large. Our winning submission was the average of 62 RGF runs, each of which used different data pre-processing.

	Leaderboard WMAE	
	Public	Private
Our winning entry (Average of 62 RGF-L_2 runs)	**0.68548**	**0.68031**
Second best team (GBDT and random forests)	0.69380	0.69062
RGF-L_2 (single run)	**0.69273**	**0.68847**
GBDT (single run)	0.69504	0.69582

In the table above, "RGF-L_2 (single run)" is one of the RGF runs used to make the winning submission, and "GBDT (single run)" is GBDT[6] using exactly the same features as "RGF-L_2 (single run)". RGF produces smaller error than GBDT on both public and private portions. Furthermore, by comparison with the performance of the second best team (which blended random forest runs and GBDT runs), we observe that not only the average of the 62 RGF runs but also the single RGF run could have won the first place prize. whereas the single GBDT run would have fallen behind the second best team.

In Fig. 8, accuracy (in terms of WMAE) is shown in relation to model sizes on the 2-to-1 split of the data provided for training. RGF is more accurate than GBDT at all the model sizes; in other words, to achieve similar accuracy, RGF requires a smaller model than GBDT.

Biological Response Prediction

The task of Predicting a Biological Response (www.kaggle.com/c/bioresponse) was to predict a biological response (1/0) of molecules from their chemical properties. We were in the fourth place with a small difference from the first place.

Our best submission combined the predictions of RGF and other methods with some data conversion. For the purpose of this paper, we show the performance of RGF and GBDT on the original data for easy reproduction of

6. As `gbm` does not support the L1-L2 loss, we used our own implementation.

TABLE 5
Comparison of GBDT with and without Fully-Corrective Post Processing Proposed by [18]

	GBDT	GBDT w/post-proc.	
CT slices	7.6861	*7.6371*	(2.9%)
Houses	*0.3454*	0.3753	(11.8%)
MSD	*9.6846*	10.0624	(16.1%)

	GBDT	GBDT w/post-proc.	
Adult	*85.62*	85.10	(13.5%)
Letter	*91.20*	90.69	(8.7%)
Musk	*97.14*	96.62	(7.3%)
Nursery	*98.13*	97.62	(8.8%)
Waveform	*89.56*	88.71	(12.8%)

RMSE/accuracy(%) and model sizes (in parentheses) relative to those without post-processing. Square loss. Average of 3 runs. The post-processing decreases the model size, but noticeably degrades accuracy on all but one dataset.

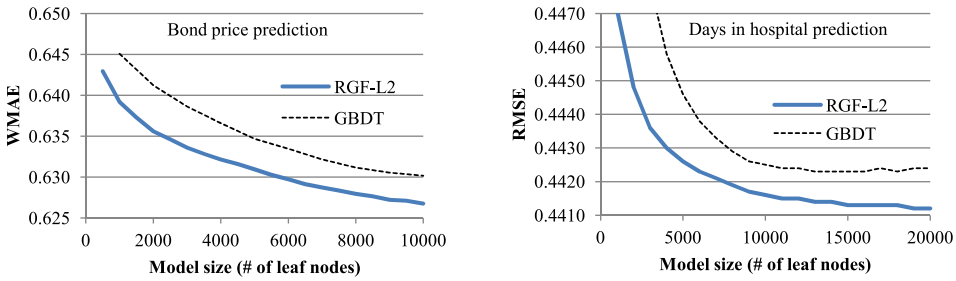

Fig. 8. Accuracy in relation to model size.

the results. Although the evaluation metric was log loss, $-\frac{1}{n}\sum_{i=1}^{n} y_i \log(f(\mathbf{x}_i)) + (1-y_i)\log(1-f(\mathbf{x}_i))$, we found that with both RGF and GBDT, better results can be obtained by training with square loss and then calibrating the predictions by: $g(x) = (0.05 + x)/2$ if $x < 0.05$; $(0.95 + x)/2$ if $x > 0.95$; x otherwise. The log loss results shown in the table below were obtained this way. RGF produces better results than GBDT on both public and private sets. The same model size was used for both RGF and GBDT, which was found to generally produce the best accuracy for both.

	Leaderboard Log loss	
	Public	Private
RGF-L_2	**0.42224**	**0.39361**
GBDT	0.43576	0.40105

Predicting Days in a Hospital—$3M Grand Prize

After two kaggle competitions with good results, we decided to enter the highest profile kaggle competition at the time—Heritage Provider Network Health Prize (www.heritagehealthprize.com/c/hhp). This was a two-year-long competition with $3,000,000 Grand Prize and three milestones, which attracted 1660 teams. We entered the competition right before the 3rd/final milestone, in which we achieved the 2nd place, and then we were asked by other milestone winners to merge with them. The competition has concluded, and our team won the 1st place (though the threshold for the $3M was not achieved).

The task was to predict the number of days people will spend in a hospital in the next year based on "historical claims data". We show the public Leaderboard performance of an RGF run and a GBDT run applied to the same features in the table below. Both runs were part of the winning submission. We also show the 5-fold cross validation results of our testing using training data on the same features. Again RGF achieves lower error than GBDT in both comparisons.

	Public LB RMSE	cross validation
RGF-L_2	**0.459877**	**0.440874**
GBDT	0.460997	0.441612

Furthermore, we have 5-fold cross validation results of RGF and GBDT on 53 datasets each of which uses features composed differently. Their corresponding official runs are all part of the winning submission. On *all* of the 53 datasets, RGF produced lower error than GBDT with the average of error differences 0.0005, which is significant on this data. The superiority of RGF is consistent on these datasets. This provides us competitive advantage to do well in all three competitions we have entered.

In Fig. 8, accuracy (in terms of RMSE) is shown in relation to model sizes on the 4-to-1 split of the data provided for training. RGF is more accurate than GBDT at all the model sizes; in other words, to achieve similar accuracy, RGF requires a smaller model than GBDT.

7 Running Time

Compared with GBDT, computation of RGF involves additional complexity mainly for fully-corrective weight updates; however, running time of RGF is linear in the number of training data points. Below we analyze running time in terms of the following factors: ℓ, the number of leaf nodes generated during training; d, dimensionality of the original input space; n, the number of training data points; c, how many times the fully-corrective weight optimization is done; and z, the number of leaf nodes in one tree, or tree size. In RGF, tree size depends on the characteristics of data and strength of regularization. Although tree size can differ from tree to tree, for simplicity we treat it as one quantity, which should be approximated by the average tree size in applications.

In typical tree ensemble learning implementation, for efficiency, the data points are sorted according to feature values at the beginning of training. The following analysis assumes that this "pre-sorting" has been done. Pre-sorting runs in $O(nd\log(n))$, but its actual running time seems practically negligible compared with the other part of training even when n is as large as 100,000.

Recall that RGF training consists of two major parts: one grows the forest, and the other optimizes/corrects the weights of leaf nodes. The part to grow the forest excluding regularization runs in $O(nd\ell)$, same as GBDT. Weight optimization takes place c times, and each time we have an optimization problem of n data points each of

TABLE 6
Competition Data Statistics

	Data size		dim	#team
	#train	#test		
Bond prices	762,678	61,146	91	265
Bio response	3,751	2,501	1,776	703
Health Prize	71,435	70,942	50468	1,660

The "Dim" (feature dimensionality) and #train are shown for the data used by one particular run for each competition for which we show the leaderboard performance in Section 6.4.

which has at most $\frac{\ell}{z}$ nonzero entries; therefore, the running time for optimization, excluding regularization, is $O(\frac{n\ell c}{z})$ using coordinate descent implemented with sparse matrix representation.

During forest building, the partial derivatives and the reduction of regularization penalty are referred to $O(nd\ell)$ times. During weight optimization, the partial derivatives of the penalty are required $O(\ell c)$ times. With RGF-L_2, computation of these quantities is practically negligible. Computation of min-penalty regularizers involves $O(z)$ nodes; however, with efficient implementation that stores and reuses invariant quantities, extra running time for min-penalty regularizers during forest building can be reduced to $O(nd\ell) + O(\ell z^2)$ from $O(nd\ell z)$. The extra running time during weight optimization is $O(\ell cz)$, but the constant part can be substantially reduced by efficient implementation.

Actual execution time for training depends on not only the data but also the design of the parameter selection process. In our experiments in the previous section, we performed 2-fold cross validation for parameter selection from 8 (RGF-L_2) and 30 (GBDT) parameter combinations for square loss; the total time for parameter selection on, for example, Letter, was 128 seconds with RGF-L_2 and 191 seconds with GBDT. That is, even though RGF training tends to take longer than GBDT individually, the total time for parameter selection could be shorter with RGF. On the same Letter dataset, parameter selection for AdaBoost with stumps (which was simply for deciding how large the model should be) took only 33 seconds; however, its accuracy is 12% lower than RGF, which makes longer training time for RGF worthwhile.

8 Conclusion

This paper introduced a new method that learns a nonlinear function by using an additive model over nonlinear decision rules. Unlike the traditional boosted decision tree approach, the proposed method directly works with the underlying forest structure. The resulting method, which we refer to as regularized greedy forest (RGF), integrates two ideas: one is to include tree-structured regularization into the learning formulation; and the other is to employ the fully-corrective regularized greedy algorithm. Since in this approach we are able to take advantage of the special structure of the decision forest, the resulting learning method is effective and principled. Our empirical studies showed that the new method can achieve more accurate predictions than existing methods which we tested.

Acknowledgments

T. Zhang is supported by the following grants: NSF IIS-1016061, NSF DMS-1007527, and NSF IIS-1250985.

References

[1] F. Bach, "Exploring large feature spaces with hierarchical multiple kernel learning," in *Proc. NIPS*, 2008.
[2] F. Bach, "High-dimensional non-linear variable selection through hierarchical kernel learning," Tech. Rep. 00413473, HAL, 2009.
[3] R. Baraniuk, V. Cevher, M. F. Duarte, and C. Hegde, "Model based compressive sensing," *IEEE Trans. Inf. Theory*, vol. 56, no. 4, pp. 1982–2001, Apr. 2010.
[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Belmont, CA, USA: Wadsworth Advanced Books and Software, 1984.
[5] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.
[6] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[7] L. Breiman, A. Cutler, A. Liaw, and M. Wiener, *Package 'RandomForest'*, 2010.
[8] K. Bube and R. Langan, "Hybrid ℓ^1/ℓ^2 minimization with application to tomography," *Geophysics*, vol. 62, no. 4, pp. 1183–1195, 1997.
[9] H. Chipman and R. McCulloch, Package '*BayesTree*', 2010.
[10] H. A. Chipman, E. I. George, and R. E. McCulloch, "BART: Bayesian additive regression trees," *Ann. Appl. Statist.*, vol. 4, no. 1, pp. 266–298, 2010.
[11] A. Frank and A. Asuncion. (2010). "UCI machine learning repository," School Inform. Comput. Sci., Univ. California, Irvine, CA, USA [Online]. Available: http://archive.ics.uci.edu/ml
[12] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, 1997.
[13] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," *JMLR*, vol. 4, pp. 933–969, Nov. 2003.
[14] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *Proc. ICML*, San Francisco, CA, USA, 1999, pp. 124–133.
[15] J. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1536, 2001.
[16] J. Friedman, T. Hastie, and R. Tibshirani, Package *'glmnet'*, 2011.
[17] J. H. Friedman and B. E. Popescu, "Importance sampled learning ensembles," Dept. Stat., Stanford Univ., Tech. Rep., 2003.
[18] J. H. Friedman and B. E. Popescu, "Predictive learning via rule ensembles," *Ann. Appl. Statist.*, vol. 2, no. 3, pp. 916–954, 2008.
[19] R. Herbrich, T. Graepel, and K. Obermayer, "Large margin rank boundaries for ordinal regression," in *Advances in Large Margin Classifiers*, B. Schölkopf A. Smola, P. Bartlett and D. Schuurmans, Eds. Cambridge, MA, USA: MIT Press, 2000, pp. 115–132.
[20] J. Huang, T. Zhang, and D. Metaxas, "Learning with structured sparsity," *JMLR*, vol. 12, pp. 3371–3412, Nov. 2011.
[21] L. Jacob, G. Obozinski, and J. Vert, "Group lasso with overlap and graph lasso," in *Proc. 26th ICML*, Montreal, QC, Canada, 2009.
[22] R. Jenatton, J.-Y. Audibert, and F. Bach, "Structured variable selection with sparsity-inducing norms," *JMLR*, vol. 12, pp. 2777–2824, Oct. 2011.
[23] J. Ross Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA, USA: Morgan Kaufmann, 1993.
[24] G. Ridgeway, Package 'gbm' v1, 2006.
[25] R. E. Schapire, "The boosting approach to machine learning: An overview," in *Nonlinear Estimation Classification.* New York, NY, USA: Springer, 2003.
[26] R. E. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions," *Mach. Learn.*, vol. 37, no. 3, pp. 297–336, 1997.
[27] S. Shalev-Shwartz, N. Srebro, and T. Zhang, "Trading accuracy for sparsity in optimization problems with sparsity constraints," *SIAM J. Optim.*, vol. 20, no. 6, pp. 2807–2832, Aug. 2010.
[28] M. Warmuth, J. Liao, and G. Ratsch, "Totally corrective boosting algorithms that maximize the margin," in *Proc. 23rd ICML*, Pittsburgh, PA, USA, 2006.
[29] T. Zhang and B. Yu, "Boosting with early stopping: Convergence and consistency," *Ann. Statist.*, vol. 33, no. 4, pp. 1538–1579, 2005.

Rie Johnson received the Ph.D. degree in computer science from Cornell University, Ithaca, NY, USA, in 2001. She was a research scientist with the IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, until 2007. Her current research interests include machine learning and its applications.

Tong Zhang received the B.A. degree in mathematics and computer science from Cornell University, Ithaca, NY, USA, in 1994 and the Ph.D. degree in computer science from Stanford University, Stanford, CA, USA, in 1999. After graduation, he was with the IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, and Yahoo Research, New York, NY, USA. Currently, he is a professor of statistics with Rutgers University, New Brunswick, NJ, USA. His current research interests include machine learning, statistical algorithms, their mathematical analysis and applications.