Learning Nonlinear Functions Using Regularized Greedy Forest

Rie Johnson
RJ Research Consulting

Tong Zhang
Rutgers University

January 27, 2012

Abstract

We apply the concept of structured sparsity to improve boosted decision trees with general loss functions. The existing approach to this problem is Friedman's gradient boosting procedure using a regression tree base learner. Although this method has led to many successful industrial applications, it suffers from several theoretical and practical drawbacks. By employing the idea of structured greedy search, we are able to design a regularized greedy forest procedure to address these issues. The resulting method constructs tree ensembles more effectively than gradient boosting, and achieves better performance on most datasets we have tested on. This work suggests that we can view boosted decision trees as procedures that construct high order nonlinear interactions through the concept of structured sparsity regularization, and this general view can guide us to design nonlinear learning algorithms that are more effective than existing methods.

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[21]), which has an important advantage for handling heterogeneous data with ease when different features come from different sources. This makes decision trees a popular "black box" 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 the "boosting" idea [11, 14, 22], 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 black-box 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 Forest [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 computer science (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 improving Friedman's gradient boosting decision tree (GBDT) approach [14] which has the ability to handle general loss functions (while other boosting methods such as Adaboost are specific to certain loss functions), and has a wider range of applicability. We show that RGF can deliver better results than GBDT on many datasets.

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 $h(\mathbf{x})$ from a function class $\mathcal{H}$ that minimizes a risk function

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

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 $h \in \mathbb{R}^n$.

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 [17, 12]), 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 $h(\mathbf{x}) \leq h(\mathbf{x}') + 1$. 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 learn-

ing/optimization algorithm. This paper achieves nonlinearity by using additive models of the form:

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

where each $\alpha_j \in \mathbb{R}$ is a coefficient that can be optimized, and each $g_j(\mathbf{x})$ 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

$$\mathcal{C} = \left\{ g(\cdot) : g(\mathbf{x}) = \prod_{j} \mathcal{I}(\mathbf{x}[i_j] \leq t_j) \prod_{k} \mathcal{I}(\mathbf{x}[i_k] > t_k) \right\}, \tag{3}$$

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:

$$\mathcal{I}(p) = \begin{cases} 1 & \text{if } p \text{ is true} \\ 0 & \text{otherwise} \end{cases}.$$

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). Since this procedure is specifically targeted at improving the popular gradient boosting algorithm [14], in the next section we shall briefly describe this algorithm as well as its pros and cons.

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 output a nonlinear function $\hat{g} = \mathcal{A}(\tilde{X}, \tilde{Y})$ that approximately solve the regression problem:*

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

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). The general algorithm is described in Algorithm 1. 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, Algorithm 1 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 and implemented in our

experimental comparisons. We shall refer to this modification as gradient boosted decision tree (GBDT), and the details are listed in Algorithm 5.

Algorithm 1: Generic Gradient Boosting [14]

$h_0(\mathbf{x}) \leftarrow 0$
for $k = 1$ **to** K **do**
  $\tilde{Y}_k \leftarrow -\partial \mathcal{L}(h, Y)/\partial h|_{h=h(X)}$
  $g_k \leftarrow \mathcal{A}(X, \tilde{Y}_k)$
  $\beta_k \leftarrow \arg\min_{\beta \in \mathbb{R}} \mathcal{L}(h_{k-1}(X) + \beta \cdot g_k(X), Y)$
  $h_k(\mathbf{x}) \leftarrow h_{k-1}(\mathbf{x}) + s\beta_k g_k(\mathbf{x})$ // s is a shrinkage parameter
end
return $h(\mathbf{x}) = h_K(\mathbf{x})$

Gradient boosting may be regarded as a functional generalization of gradient descent method $h_k \leftarrow h_{k-1} - s_k \partial \mathcal{L}(h_t)/\partial h_t$, where s_t (corresponding to the shrinkage parameter s) is the step-size, and $\tilde{Y}_k = -\partial \mathcal{L}(h_t)/\partial h_t$ is approximated using the regression tree output $\mathcal{A}(X, \tilde{Y}_k)$. 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 [25], 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 proven to be 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, which we address in this work. The first disadvantage is that there is no explicit regularization in the algorithm, and in fact, it is argued in [25] 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 term of regularization is unclear, and the resulting implicit regularization may not be effective. The second disadvantage is also a consequence of using small step size s as implicit regularization, as this can lead to huge number of trees, which is very undesirable in applications due to the increased computational cost. Note that in order to achieve good performance, it is often necessary to choose a small shrinkage parameter s and hence large K; in the extremely scenario of ϵ-boosting, one needs an infinite number of trees. The third disadvantage is that 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. The purpose of this work is to remedy these drawbacks by presenting a method that directly learns a decision forest using structured greedy search and structured sparsity regularization.

4 Fully-Corrective Greedy Update and Structured Sparsity Regularization

One disadvantage of gradient boosting is that in order to achieve good performance in practice, the shrinkage parameter s often needs to be small, and Friedman himself argued for infinitesimal step size. This practical observation is supported by the theoretical analysis in [25] 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 [23]

$h_0(\mathbf{x}) \leftarrow 0$
for $k = 1$ **to** K **do**
  $\tilde{Y}_k \leftarrow -\partial\mathcal{L}(h, Y)/\partial h|_{h=h(X)}$
  $g_k \leftarrow \mathcal{A}(X, \tilde{Y}_k)$
  let $\mathcal{H}_k = \{\sum_{j=1}^k \beta_j g_j(\mathbf{x}) : \beta_j \in \mathbb{R}\}$
  $h_k(\mathbf{x}) \leftarrow \arg\min_{h\in\mathcal{H}_k} \mathcal{L}(h(X), Y)$ // fully-corrective step
end
return $h(\mathbf{x}) = h_K(\mathbf{x})$

In gradient boosting of Algorithm 1 (or its variation with tree base learner of Algorithm 5), the algorithm only does a partial corrective step that optimizes either the coefficient of the last basis function g_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 $\{g_j\}_{j=1}^k$ obtained so far at each iteration k. It was noticed empirically that such fully-corrective step can significantly accelerate the convergence of boosting procedures [24]. This observation was theoretically justified in [23] 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

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

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 [25]). 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 [3, 20, 19, 1, 2, 18]. That is, we will include a regularization term in our learning formulation that not only encourages using a small number of

decision rules (sparsity), but also takes advantage of the underlying forest structure. In our setting, we are interested in an additive model $h(\mathbf{x})$ that is a linear combination of decision rules $g(\mathbf{x})$ in (3). Each decision rule is a basis function or atom, and the additive model $h(\mathbf{x})$ is a sparse combination of $g(\mathbf{x})$. This problem can be considered as a variable selection problem. However, search over all nonlinear interactions (atoms) over $\mathcal{C}$ is computationally difficult or infeasible; one has to impose structured search over atoms, and in our case, a forest structure in Figure 2. This naturally corresponds to the recently investigated concept of structured sparsity. The ideas 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.

Algorithmically, one can explore the use of an underlying graph structure that connects different variables in order to search over sparsity patterns. The general concept of graph-structured sparsity were considered in [1, 2, 18]. A simple example is presented in Figure 1, where each node of the graph indicates a variable (non-linear decision rule), and each gray node denotes a selected variable. The graph structure is used to select variables that form a *connected region*; that is, we may grow the region by following the edges from the variables that have been selected already, and new variables are selected one by one. Figure 1 indicates the order of selection. This approach reduces both statistical complexity and algorithmic complexity. The algorithmic advantage is quite obvious; statistically, using the information theoretical argument in [18], one can show that the generalization performance is characterized by $O(\sum_{j\in\{\text{selected nodes}\}} \log_2 \text{degree}(\text{parent}(j)))$, while without structure, it will be $O(\#\{\text{selected nodes}\} \cdot \ln p)$, where p is the total number of atoms in $\mathcal{C}$. Based on this general idea, [2] considered the problem of learning with nonlinear kernels induced by an underlying graph.

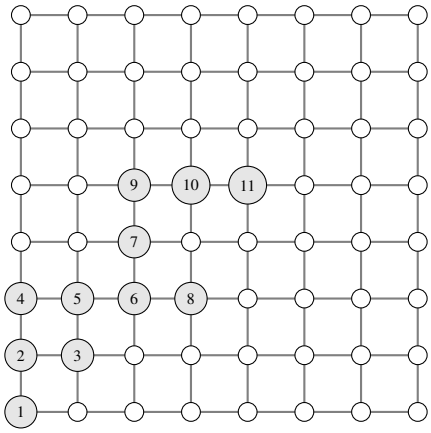

Figure 1: Graph Structured Sparsity

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 improve 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 [13], 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

Our solution, regularized greedy forest (RGF), is designed to overcome GBDT's shortcomings. The first problem of GBDT is the lack of explicit regularization and complex control. Our solution is to 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). The regularizer we introduce explicitly depends on the underlying forest structure, and the numerical algorithm also takes advantage of such structure. The second problem of GBDT is the shrinkage parameter s, which is important for performance; however, when s is small, a huge number of trees is necessary, and this raises computational complexity. Our solution is to employ *fully-corrective greedy algorithm* which repeatedly performs greedy search and re-optimization of all coefficients. This does not lead to overfitting because explicit regularization is introduced in our formulation, and thus we can avoid the problem introduced by using small s. The third problem of GBDT is treating tree learning as a black-box, which is decoupled from the boosting (wrapper) step. Our solution is to employ the concept of graph structured sparsity, and perform structured greedy search directly over forest nodes. 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 graph sparsity structure (i.e. forest structure). Conceptually gradient boosting algorithm is designed for an additive model on trees (tree ensemble) even though at a finer level it is also an additive model of forest nodes. As a result, the structure of the finer level (i.e., node level) is not sufficiently explored by the GBDT algorithm.

In summary, the proposed regularized greedy forest (RGF) approach is designed to learn a decision forest directly. Before going into more details, 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 $T_1, \ldots, T_K$ that are connected with a common root. The forest shown in Figure 2 contains three trees T_1, T_2, and T_3. The common root node is equivalent to the root nodes for the trees T_1, T_2, T_3, and they are duplicated in the graphical representation for convenience. 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 common root to this node. The common root itself denotes the constant function. Although the underlying graph of this decision forest is a "tree" from the graph theoretical point of view, we call this structure a forest instead of tree due to the semantics of the node described above.

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

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

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 $g_v(\mathbf{x}) = g_{v_1}(\mathbf{x}) + g_{v_2}(\mathbf{x})$. This means that any internal node is redundant in

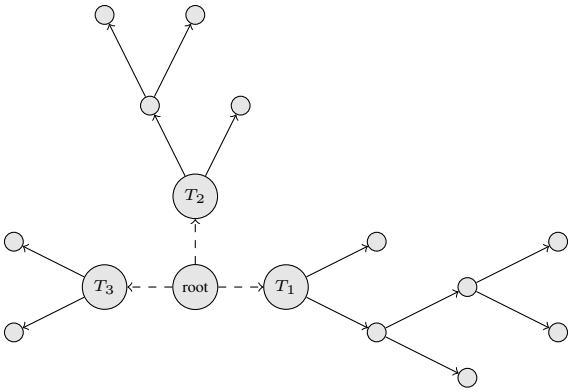

Figure 2: Decision Forest

the sense that an additive model with basis functions $g_v(\mathbf{x}), g_{v_1}(\mathbf{x}), g_{v_2}(\mathbf{x})$ can be represented as an additive model over basis functions $g_{v_1}(\mathbf{x})$ and $g_{v_2}(\mathbf{x})$. 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. However, internal nodes are still useful conceptually as we will see later that we can define tree-structured regularization based on all nodes including internal nodes.

Now, let $\mathcal{F}$ represent a forest, and each node v of $\mathcal{F}$ is associated with $(g_v, \alpha_v, \theta_v)$. Here g_v is the basis function that this node represents; α_v is the *weight* or coefficient assigned to this node; and θ_v represents other *attributes* of this node such as the depth and all the information sufficient for reconstructing the forest and defining tree-structured regularization introduced later. The additive model of this forest $\mathcal{F}$ considered in this paper is:

$$h_{\mathcal{F}}(\mathbf{x}) = \sum_{v \in \mathcal{F}} \alpha_v \cdot g_v(\mathbf{x}) \ ,$$

or if we emphasize the fact that our model is over leaf nodes only,

$$h_{\mathcal{F}}(\mathbf{x}) = \sum_{v \in L_{\mathcal{F}}} \alpha_v \cdot g_v(\mathbf{x}) \ ,$$

where $L_{\mathcal{F}}$ represents the set of leaf nodes in forest $\mathcal{F}$. We can then rewrite the regularized loss in (4) as follows to emphasize that the regularizer depends on the underlying forest structure, by replacing the regularization term $\mathcal{R}(h_{\mathcal{F}})$ with $\mathcal{G}(\mathcal{F})$:

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

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 is difficult to find, we *greedily* select the *basis functions* and optimize the *weights*. The key algorithmic ideas can be described as follows: first we build a forest by growing possibly multiple trees simultaneously; second, we use structured greedy search algorithm to search over nonlinear decision rules one by one based on the underlying forest (graph) structure; third, perform fully-corrective greedy search so that all the coefficients of the nonlinear decision rules are re-optimized repeatedly while the rules are added

into the forest by greedy search; fourth, we use individual tree structures in the forest to define regularizers on additive models associated with each tree. Finally all computation is performed using leaf-nodes only, although internal nodes can be used to define the regularization condition. This is possible because, as we have pointed out earlier, an additive model over all forest nodes can be equivalently represented by an additive model over only leaf-nodes of the same forest.

At the 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 2–4).
- Fix the *structure* of the forest, and change the *weights* so that loss $\mathcal{Q}(\mathcal{F})$ is minimized (Line 5).

Algorithm 3: Regularized greedy forest framework

1 $\mathcal{F} \leftarrow \{\}$.
repeat
2 $\quad \hat{o} \leftarrow \arg\min_{o \in O(\mathcal{F})} \mathcal{Q}(o(\mathcal{F}))$ where $O(\mathcal{F})$ is a set of all the structure-changing operations applicable to $\mathcal{F}$.
3 $\quad$ **if** $(\mathcal{Q}(\hat{o}(\mathcal{F})) \geq \mathcal{Q}(\mathcal{F}))$ **then** break // Leave the loop if $\hat{o}$ does not reduce the loss.
4 $\quad \mathcal{F} \leftarrow \hat{o}(\mathcal{F})$. // Perform the optimum operation.
5 $\quad$ **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 $h_{\mathcal{F}}(\mathbf{x})$

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).

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, Figure 3 shows that at the same stage as Figure 2, we may either consider splitting leaf nodes marked with symbol X or grow a new tree T_4.

Consequently, 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.

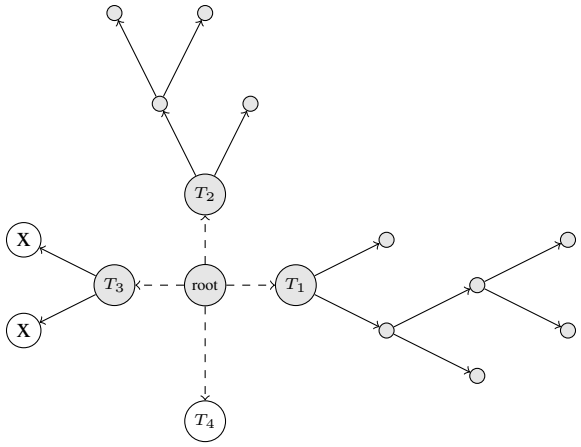

Figure 3: Decision Forest Splitting Strategy (we may either split leaf nodes X or grow a new tree T_4)

Computation Consider the evaluation of loss reduction by splitting a node associated with (g, α, θ) into the nodes associated with $(g_{u_1}, \alpha + \delta_1, \theta_{u_1})$ and $(g_{u_2}, \alpha + \delta_2, \theta_{u_2})$. Then the model associated with the new forest $\tilde{\mathcal{F}} = o(\mathcal{F})$ after splitting the node can be written as:

$$h_{\tilde{\mathcal{F}}}(\mathbf{x}) = h_{\mathcal{F}}(\mathbf{x}) - \alpha \cdot g(\mathbf{x}) + \sum_{k=1}^{2} (\alpha + \delta_k) g_{u_k}(\mathbf{x}) = h_{\mathcal{F}}(\mathbf{x}) + \sum_{k=1}^{2} \delta_k \cdot g_{u_k}(\mathbf{x}). \tag{6}$$

Recall that our additive models are over leaf nodes only. The node that was split is no longer leaf and therefore $\alpha \cdot g(\mathbf{x})$ is removed from the model. The second equality is from $g(\mathbf{x}) = g_{u_1}(\mathbf{x}) + g_{u_2}(\mathbf{x})$ due to the parent-child relationship. To emphasize that δ_1 and δ_2 are the only variables in $\tilde{\mathcal{F}}$ for the current computation, let us write $\tilde{\mathcal{F}}(\delta_1, \delta_2)$ for the new forest. 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 $\mathcal{Q}(\mathcal{F})$. In general, there may not be an analytical 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:

$$\hat{\delta}_k = \frac{-\frac{\partial \mathcal{Q}(\tilde{\mathcal{F}}(\delta_1,\delta_2))}{\partial \delta_k}|_{\delta_1=0,\delta_2=0}}{\frac{\partial^2 \mathcal{Q}(\tilde{\mathcal{F}}(\delta_1,\delta_2))}{\partial \delta_k^2}|_{\delta_1=0,\delta_2=0}}. \tag{7}$$

In particular, suppose that loss function is for either regression or classification tasks, then $\mathcal{Q}(\mathcal{F})$ can be written in the following form

$$\mathcal{Q}(\mathcal{F}) = \sum_{i=1}^{n} \ell(h_{\mathcal{F}}(\mathbf{x}_i), y_i)/n + \mathcal{G}(\mathcal{F}) \, .$$

In this case, $\frac{\partial h_{\tilde{\mathcal{F}}}(\mathbf{x})}{\partial \delta_k} = g_{u_k}(\mathbf{x})$, $\frac{\partial^2 h_{\tilde{\mathcal{F}}}(\mathbf{x})}{\partial \delta_k^2} = 0$, and $h_{\mathcal{F}}(\mathbf{x}) = h_{\tilde{\mathcal{F}}(0,0)}(\mathbf{x})$ by (6); thus $\hat{\delta}_k$ in (7) can be rewritten as:

$$\hat{\delta}_k = \frac{-\sum_{g_{u_k}(\mathbf{x}_i)=1} \frac{\partial \ell(h,y)}{\partial h}|_{h=h_{\mathcal{F}}(\mathbf{x}_i),y=y_i} - n\frac{\partial \mathcal{G}(\tilde{\mathcal{F}}(\delta_1,\delta_2))}{\partial \delta_k}|_{\delta_1=0,\delta_2=0}}{\sum_{g_{u_k}(\mathbf{x}_i)=1} \frac{\partial^2 \ell(h,y)}{\partial h^2}|_{h=h_{\mathcal{F}}(\mathbf{x}_i),y=y_i} + n\frac{\partial^2 \mathcal{G}(\tilde{\mathcal{F}}(\delta_1,\delta_2))}{\partial \delta_k^2}|_{\delta_1=0,\delta_2=0}}.$$

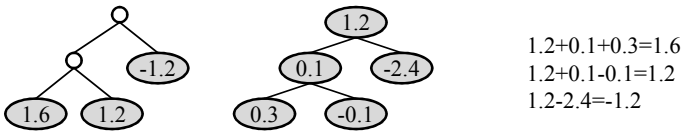

Figure 4: Example of Equivalent Models

For example, with square loss $\ell(h, y) = (h - y)^2/2$ and L_2 regularization penalty $\mathcal{G}(\mathcal{F}) = \lambda \sum_{v \in \mathcal{F}} \alpha_v^2/2$, we have

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

which is the exact optimum for the given split.

5.3.2 Weight optimization/correction (Line 5)

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:

$$\alpha_v \leftarrow \alpha_v + \eta \cdot \frac{-\frac{\partial \mathcal{Q}(\mathcal{F}(\delta_v))}{\partial \delta_v}\big|_{\delta_v=0}}{\frac{\partial^2 \mathcal{Q}(\mathcal{F}(\delta_v))}{\partial \delta_v^2}\big|_{\delta_v=0}}, \tag{8}$$

where δ_v is the additive change to α_v. Computation of the Newton step is similar to Section 5.3.1.

Since the initial weights of new leaf nodes set in Line 4 are approximately optimum at the moment, it is not necessary to perform weight correction in every iteration, which is relatively expensive. We found that the strategy of "correcting the weights once every few new leaf nodes are added" works well. The interval between fully-corrective updates is not crucial as long as it is not too large.

5.4 Tree-structured regularization

In our formulation, which minimizes the regularized loss $\mathcal{Q}(\mathcal{F}) = \mathcal{L}(\cdot) + \mathcal{G}(\mathcal{F})$, one can express preference or prior knowledge about how the ideal forest should look via the regularization penalty term or *regularizer* $\mathcal{G}(\mathcal{F})$. In this section, to simplify notation, we define regularizers over a single tree instead of the entire forest. The regularizer over a forest can be obtained by adding the regularizers described here over all the trees in the forest. Therefore, suppose that we are given a tree T with an additive model over leaf nodes:

$$h_T(\mathbf{x}) = \sum_{v \in T} \alpha_v \cdot g_v(\mathbf{x}) = \sum_{v \in L_T} \alpha_v \cdot g_v(\mathbf{x}) \quad \text{(i.e., } \alpha_v = 0 \text{ for } v \notin L_T\text{)},$$

where L_T denotes the set of leaf nodes in T.

To consider useful regularizers defined on the given tree T, 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. Then if for all

$u \in L_T$, $\sum_{v \in A(u)} \beta_v = \alpha_u$, as illustrated in Figure 4, the additive model defined with $\{\beta_v\}$ is equivalent to the given leaf-only model according to the underlying tree topology:

$$\sum_{v \in T} \beta_v \cdot g_v(\mathbf{x}) \equiv \sum_{v \in L_T} \alpha_v \cdot g_v(\mathbf{x}) \; .$$

Our basic idea is that it is natural to give the same regularization penalty to all equivalent additive models defined on the same tree topology. One way to define a regularizer that satisfies this condition is to choose an additive 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.

More concretely, we consider the following form of regularization:

$$\mathcal{G}(T) = \sum_{v \in T} r(\rho_v, \theta_v) \;:\; \forall u \in L_T. \left[\sum_{v \in A(u)} \rho_v = \alpha_u \right] \; .$$

Here node v includes both internal and leaf nodes; the additive model defined by $\{\rho_v\}_{v \in T}$ serves as the unique representation of the set of equivalent models; and $r(\cdot, \cdot)$ is a penalty function of a node weight and node attributes; Each ρ_v is a function of given leaf weights $\{\alpha_u\}_{u \in L_T}$, though the function may not be a closed form. Since this formulation allows the entire tree including its topological structure to play a role in regularization, we call regularizers in this form *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 (namely, sets $\rho_v = \alpha_v$) and sets $r(\rho_v, \theta_v) = \lambda \rho_v^2 / 2$. This penalizes large weights (and therefore more expressive/complex models) using the standard L_2-regularization and leads to

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

where λ is a constant for controlling the strength of regularization. Among the equivalent models, the leaf-only model is often (but not always[1]) the one with the smallest number of basis functions and therefore least complex in that regard.

5.4.2 Minimum-penalty regularization

The regularizer above is simple to implement and works reasonably well, but it does not fully utilize information available through the tree structure. A more dynamic approach would be to first define an arbitrary regularizer for any given additive model, and then define the final regularization of a tree as the minimum regularization value over all equivalent additive models. In other words, we choose the model that minimizes the penalty as representative of the equivalent models, as it is the most preferable model according to the defined penalty. We call the regularizers based on this approach *min-penalty regularizers*.

[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.

The following is a min-penalty regularizer, in which bigger penalties are imposed on larger weights and more complex basis functions associated with deeper nodes:

$$\mathcal{G}(T) = \lambda \cdot \min_{\{\beta_v\}} \left\{ \sum_{v \in T} \gamma^{d_v} \beta_v^2 / 2 : \ \forall u \in L_T . \left[\sum_{v \in A(u)} \beta_v = \alpha_u \right] \right\} . \tag{9}$$

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, and we assume that $\gamma \geq 1$. This new regularizer explicitly regularizes the complexity of basis functions via the node depth.

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

$$\begin{cases} \bar{\beta}_{o_T} = \beta_{o_T} \\ \bar{\beta}_v = \beta_v + \bar{\beta}_{p(v)} \end{cases} ,$$

where o_T denotes the root node of T, and $p(v)$ denotes node v's parent node. Then, the equivalence condition $\forall u \in L_T . \left[\sum_{v \in A(u)} \beta_v = \alpha_u \right]$ is satisfied iff:

$$\forall v \in L_T . \left[\bar{\beta}_v = \alpha_v \right] . \tag{10}$$

Using these auxiliary variables, (9) can be rewritten as:

$$\mathcal{G}(T) = \lambda \cdot \min_{\{\bar{\beta}_v\}} \left\{ f(\{\bar{\beta}_v\}) : \forall v \in L_T . [\ \bar{\beta}_v = \alpha_v] \right\} \tag{11}$$

$$\text{where } \ f(\{\bar{\beta}_v\}) = \sum_{v \neq o_T} \gamma^{d_v} (\bar{\beta}_v - \bar{\beta}_{p(v)})^2 / 2 + \bar{\beta}_{o_T}^2 / 2 \, . \tag{12}$$

Setting f's partial derivatives to zero, we obtain that at the optimum, $\bar{\beta}_v$ of the internal node (except for the root) is the weighted average of neighbors $\bar{\beta}_v$:

$$\forall v \notin L_T : \ \bar{\beta}_v = \begin{cases} \frac{\bar{\beta}_{p(v)} + \sum_{p(w)=v} \gamma \bar{\beta}_w}{1+2\gamma} & v \neq o_T \\ \frac{\sum_{p(w)=v} \gamma \bar{\beta}_w}{1+2\gamma} & v = o_T \end{cases} , \tag{13}$$

which leads to an iterative algorithm summarized in Algorithm 4. Convergence of this algorithm and some more computational detail of this regularizer are shown in the Appendix.

5.4.3 Min-penalty regularization with sum-to-zero sibling constraints

We introduce another regularizer that is based on the same basic idea as above but is computationally simpler. As a unique representation for equivalent models, we choose the model in which the sum of weights for every sibling pair is zero, as illustrated in Figure 5. The intuition behind the 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.

Algorithm 4:

for $v \in T$ **do** $\bar{\beta}_{v,0} \leftarrow \begin{cases} \alpha_v & v \in L_T \\ 0 & v \notin L_T \end{cases}$

for $i = 1$ **to** m **do**

  for $v \in L_T$ **do** $\bar{\beta}_{v,i} \leftarrow \alpha_v$

  for $v \notin L_T$ **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 $\{\bar{\beta}_{v,m}\}$

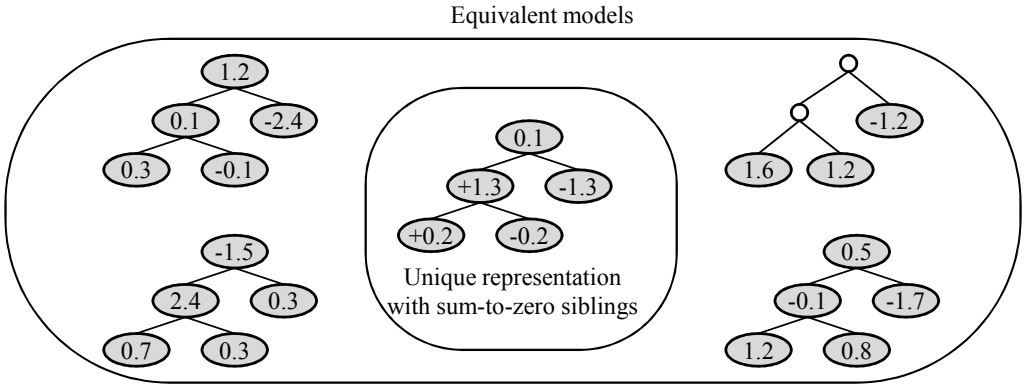

Figure 5: Example of a Sum-to-zero Sibling Model

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, whose coefficients are $\{\beta_v\}$, can be obtained through the following recursive computation:

$$\bar{\beta}_v = \begin{cases} \alpha_v & v \in L_T \\ \sum_{p(w)=v} \bar{\beta}_w / 2 & v \notin L_T \end{cases} . \tag{14}$$

Formally, this regularizer can be written as

$$\mathcal{G}(T) = \lambda \cdot \min_{\{\beta_v\}} \left\{ \sum_{v \in T} \gamma^{d_v} \beta_v^2 / 2 : \forall u \in L_T . \left[\sum_{v \in A(u)} \beta_v = \alpha_u \right] ; \forall v \notin L_T . \left[\sum_{p(w)=v} \beta_w = 0 \right] \right\} .$$

Since for a given leaf-only model, exactly one model satisfies the sibling constraints, that model trivially minimizes the constrained objective. Therefore, this regularizer is also a min-penalty regularizer. Computational detail of this regularization method is described in the Appendix.

6 Experiments

This section reports empirical studies of RGF in comparison with GBDT and some other tree ensemble methods including AdaBoost. For simplicity, we focus on square loss for regression and binary classification

Name	#test	#attribute	Task
Houses	5,000	6	Target: log(median house price)
CT slices	24,564	384	Target: relative location of CT slices
Adult	16,281	14(168)	Is income $>$ \$50K?
Letter	4,000	16	A-M vs N-Z
MNIST shrunk	10,000	49	Odd digits vs even digits
Musk2	4,598	166	Musk or not
Nursery	5,000	8(24)	"Special priority" or not
Waveform2	3,000	40	Class2 vs. Class1&3

Figure 6: Real-world Datasets. 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.

tasks, although some results with exponential loss are also shown. All experimental results in this paper can be reproduced using the RGF software available from `http://riejohnson.com/rgf_download.html`.

6.1 Data

Our experiments used synthesized data and real-world data. The purpose of the synthesized data experiments was to control the complexity of target functions (which are unknown in real-world applications) and observe the empirical behavior of the methods.

6.1.1 Synthesized data generation

In the following data generation procedure, complexity of target functions is controlled by parameter q.

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, $0, 1, 2, \cdots, q-1$, to the q leaf nodes of each of 100 trees generated.
3. Generate data points of 10 dimensions so that the components distribute uniformly over $\{0, 1, \cdots, 99\}$.
4. Apply the tree ensemble generated above to each data point. The obtained value is an interim target value.
5. To create binary classification problems, assign classes based on whether the interim target value is greater than the median. That is, the final target value is set to +1 or -1 depending on whether the interim target value is above or below the median.
6. To create regression problems, normalize the interim target value by subtracting the mean and dividing by the standard deviation.

A larger q makes the target function more complex. The datasets were generated with $q \in \{64, 16, 4, 2\}$. For each value of q, 10 datasets were synthesized with different random seeds. All the results are the average over the 10 datasets for each data type.

6.1.2 Real-world datasets

Figure 6 summarizes the tested real-world datasets. They are from a variety of domains such as real estate, health care, census records, and so on. They are mostly from the UCI Machine Learning Repository [10].

The Houses dataset[2] is for the regression task. As suggested by the creators of the data, we predict the logarithm of the median house price of the region, based on the median income, the median house age, total rooms/population, total bedrooms/population, population/households, and households. The task associated with the CT slice dataset is to predict the relative location of the CT scan images along the axial axis of the human body, based on some image attributes. MNIST[3] is for hand-written digit recognition containing intensity of 28×28 pixels. We reduced the dimensionality to 7×7 by replacing each of 4×4 regions with the average intensity of the region. The Musk Version 2 dataset contains information of conformations of molecules. The original task associated with this dataset is multi-instance learning. We use this dataset for single-instance learning by regarding all the conformations independently.

Categorical attributes, contained in Adult and Nursery, were converted to binary indicator vectors whose i-th component is 1 if the value of the attribute is the i-th category and other components are all zero.

The official test sets were used as test sets if any (Letter, MNIST, and Adult). For Nursery and Houses, which are relatively large, 5K data points were held out as test sets, and training data points were randomly drawn from the rest. For Musk2 and Waveform2, which are relatively small, in each run, 2K data points were randomly chosen as training sets, and the rest were used as test sets. The CT slice dataset was divided into the training and test sets based on whether the patent ID is odd or even. All the results are the average of 3 runs, each of which used 2K training data points that were randomly chosen from the sets disjoint from the test sets.

6.2 Parameter settings and some computational detail

6.2.1 Regularized greedy forest

RGF-L_2 (RGF with the regularizer in Section 5.4.1) was tested with the regularization parameter λ set to one of $\{1, 0.1, 0.01\}$, with a few exceptions in exponential loss experiments (described later). RGF with min-penalty regularization in Sections 5.4.2 and 5.4.3 was tested with $\gamma \in \{2, 4\}$ and $\lambda \in \{\frac{1}{\gamma}, \frac{0.1}{\gamma}, \frac{0.01}{\gamma}\}$. In all the configurations, the search for the best node split was limited to the most recently-created tree, and weight optimization was done after every 100 leaf nodes. Weight optimization was done by coordinate descent with step size set to 0.5. The number of iterations was 10 for square loss and 5 for exponential loss. On the regression task, $h_{\mathcal{F}}(\mathbf{x})$ was fitted to $\{(\mathbf{x}_i, y_i - \bar{y})\}_i$, where $\bar{y} = \sum_{i=1}^n y_i/n$, and the final output was set to $h_{\mathcal{F}}(\mathbf{x}) + \bar{y}$.

6.2.2 Gradient Boosted Decision Tree

Of special interests 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 a decision rule of the form (3). We can represent this J-leaf decision tree T by its leaf node as $\{g_j\}_{j=1}^J$, and gradient boosted decision tree (GBDT) in Algorithm 5, as proposed in [14], is an adaption of the generic gradient boosting method in Algorithm 1. This is the

[2] http://lib.stat.cmu.edu.

[3] http://yann.lecun.com/exdb/mnist/

algorithm implemented in this work for comparison.

Algorithm 5: Gradient Boosted Decision Tree (GBDT) [14]

$h_0(\mathbf{x}) \leftarrow 0$
for $k = 1$ **to** K **do**
  $\tilde{Y}_k \leftarrow -\partial \mathcal{L}(h, Y)/\partial h|_{h=h(X)}$
  Build a J-leaf decision tree $T_k \leftarrow \mathcal{A}(X, \tilde{Y}_k)$ with leaf-nodes $\{g_{k,j}\}_{j=1}^{J}$
  for $j = 1$ **to** J **do** $\beta_{k,j} \leftarrow \arg\min_{\beta \in \mathbb{R}} \mathcal{L}(h_{k-1}(X) + \beta \cdot g_{k,j}(X), Y)$
  $h_k(\mathbf{x}) \leftarrow h_{k-1}(\mathbf{x}) + s \sum_{j=1}^{J} \beta_{k,j} \cdot g_{k,j}(\mathbf{x})$ // s is a shrinkage parameter
end
return $h(\mathbf{x}) = h_K(\mathbf{x})$

With square loss, $\beta_{k,j}$ coincides with the leaf weight computed by the tree builder, which does not need to be recomputed. With exponential loss, approximation by the Newton step $\beta_{k,j} \leftarrow \frac{\sum_{g_{k,j}(\mathbf{x})=1} y_i \exp(-h_{k-1}(\mathbf{x}_i) y_i)}{\sum_{g_{k,j}(\mathbf{x})=1} \exp(-h_{k-1}(\mathbf{x}_i) y_i)}$ was performed. Following [14], the maximum number of leaf nodes of each regression tree ('tree size' in short) was parameterized, and the nodes were split in the best-first manner. For tree size, six values were tested: $\{2, 4, 8, 12, 16, 64\}$. For the shrinkage parameter s, seven values $\{1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}$ were tested.

6.2.3 Common settings

Whenever applicable, node split was prohibited if it would cause a leaf node to have training data points fewer than 10, as is commonly done.

In the computation of the optimum δ_k (7) for exponential loss, the following was done for numerical stability:

$$\mu \leftarrow \max\left(-500, \min\left(500, \frac{1}{n}\sum_{i=1}^{n} y_i h_{\mathcal{F}}(\mathbf{x}_i)\right)\right), \quad \hat{\delta}_k \leftarrow \frac{\sum_{g_{u_k}(\mathbf{x})=1} y_i e^{-y_i h_{\mathcal{F}}(\mathbf{x}_i)+\mu} - n\lambda\alpha \cdot e^{\mu}}{\sum_{g_{u_k}(\mathbf{x})=1} e^{-y_i h_{\mathcal{F}}(\mathbf{x}_i)+\mu} + n\lambda \cdot e^{\mu}}.$$

Similar protection was done in the weight correction procedure and for GBDT with exponential loss.

6.3 Comparison with GBDT

Due to our interest in sparse models, we plot performance against the number of leaf nodes, which is equal to the number of basis functions of the model. As the number of tested configurations for GBDT is too large (42) to fit in one graph, only a few best-performing configurations (measured by the peak performances) are shown for GBDT. In the legends, the first number following "gb" is the tree size and the second number is the shrinkage parameter, e.g., "gb8:s=0.1" means GBDT with 8-leaf trees with shrinkage parameter 0.1.

The first results (Figure 7) are on the datasets synthesized from 64-leaf trees and 16-leaf trees for regression. The target functions of these datasets are relatively complex. RGF-L_2 (RGF with L_2 regularization on leaf-only models; blue solid lines) achieves smaller error than GBDT (black dotted lines) with all the tested values of regularization parameter λ.

The next results are on the datasets synthesized from 4- and 2-leaf trees (stumps) in Figure 8, whose target functions are less complex than those of the previous figure. RGF-L_2 (blue solid lines) achieves smaller error than GBDT when λ is appropriate, but the merit somewhat diminishes on the stump datasets. RGF-L_2 seems to be particularly effective for relatively complex target functions, on which regularization can help to avoid overfitting.

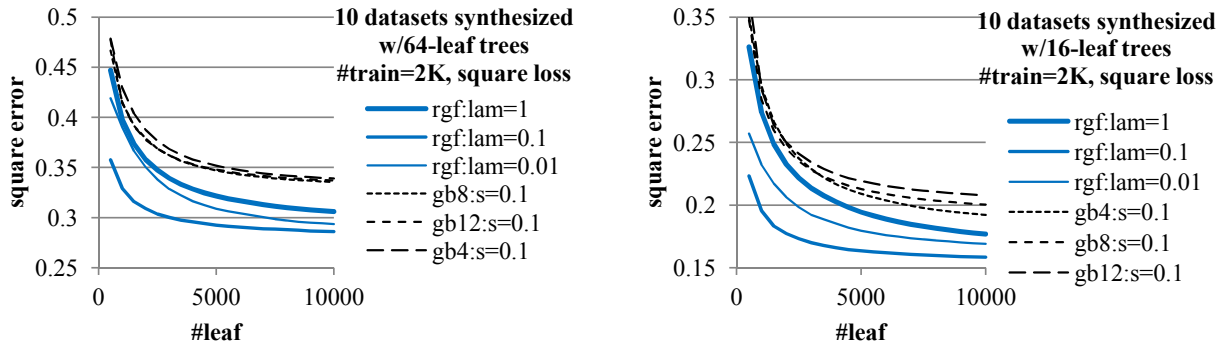

Figure 7: Performance Comparison on Synthetic Data (regression with complex targets)

On the datasets in Figure 8, RGF with the min-penalty regularization (green lines with long dashes) is proven to be effective. Recall that the min-penalty regularizer uses the weights β_v of the equivalent model over all nodes as in $\sum_v \lambda \gamma^{d_v} \beta_v^2/2$. The configurations shown here set $\gamma = 4$ or 2 and $\lambda = 0.01/\gamma$, which encourages smaller trees, and outperform not only GBDT but also RGF-L_2. The two min-penalty regularizers (with and without the sibling constraint) achieved similar performance on these datasets and only the results with the sibling constraints are shown in the figure. Similar results were obtained on the synthesized classification datasets (Figure 9).

In Figure 10, we show regression and binary classification results on the real-world datasets. RGF-L_2 (blue solid lines) shows merit over GBDT in terms of higher accuracy or sparser models on all but the Adult dataset. The min-penalty regularization with $\gamma > 1$ was found to be effective on Musk2 but not on the other datasets. Our conjecture based on the synthesized data experiments is that the unknown target functions underlying these real-world datasets are relatively complex except for Musk2.

Finally, Figure 11 shows the results of RGF and GBDT with exponential loss ("xrgf" and "xgb"). The square loss results and AdaBoost performances ("ada") are also shown for comparison. The base learners for AdaBoost were regression trees generated in the best-first manner with tree size in $\{2, 4, 8, 12, 16, 64\}$ as in the GBDT experiments. For AdaBoost, the best-performing configuration among these six configurations is shown. It appears that exponential loss generally requires weaker regularization than square loss. The effective range of regularization parameters are smaller; e.g., $\lambda = 0.01$ with square loss vs. $\lambda = 1e-20$ with exponential loss on MNIST. Consequently, GBDT's lack of explicit regularization is less harmful with exponential loss, and on some datasets GBDT and RGF (and AdaBoost) achieved similar performance. However, RGF with either square loss or exponential loss generally produced more accurate/sparse models than GBDT or AdaBoost. On some datasets, RGF with exponential loss performs better than RGF with square loss while it does not on other datasets. There may be other regularization methods that are more suitable to exponential loss, but we did not explore this direction in this paper.

GBDT with post processing of fully-corrective updates A two-stage approach was proposed in [16] that 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. Figure 12 shows representative results of their two-stage approach. As is well known, L_1 regularization has "feature selection" effects, assigning zero weights to more and more features with stronger regularization.

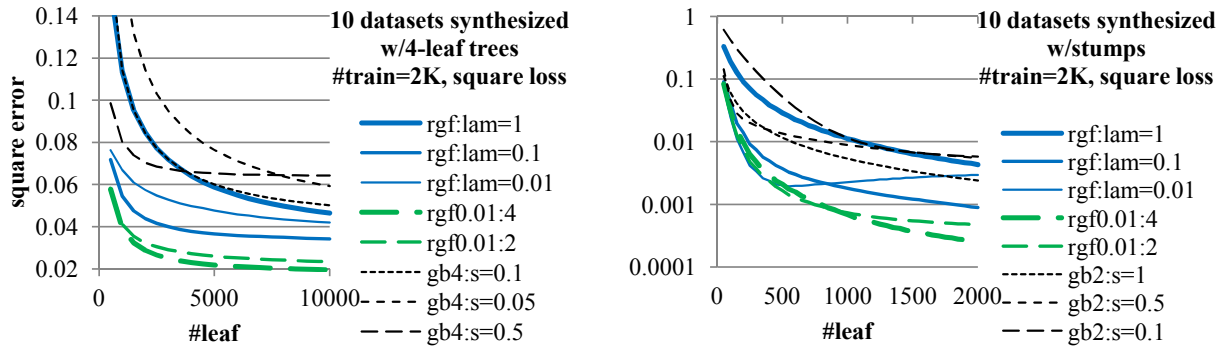

Figure 8: Performance Comparison on Synthetic Data (regression with simple targets)

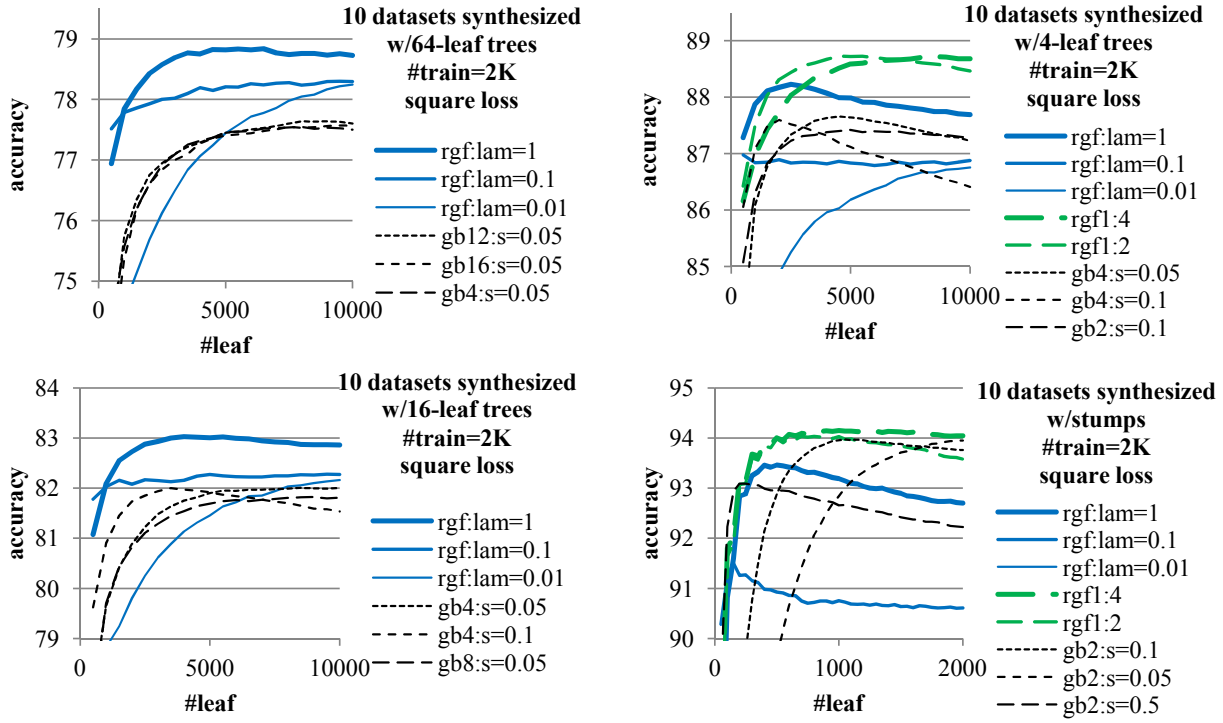

Figure 9: : Performance Comparison on Synthetic Data (classification with complex targets (left) and simple targets (right))

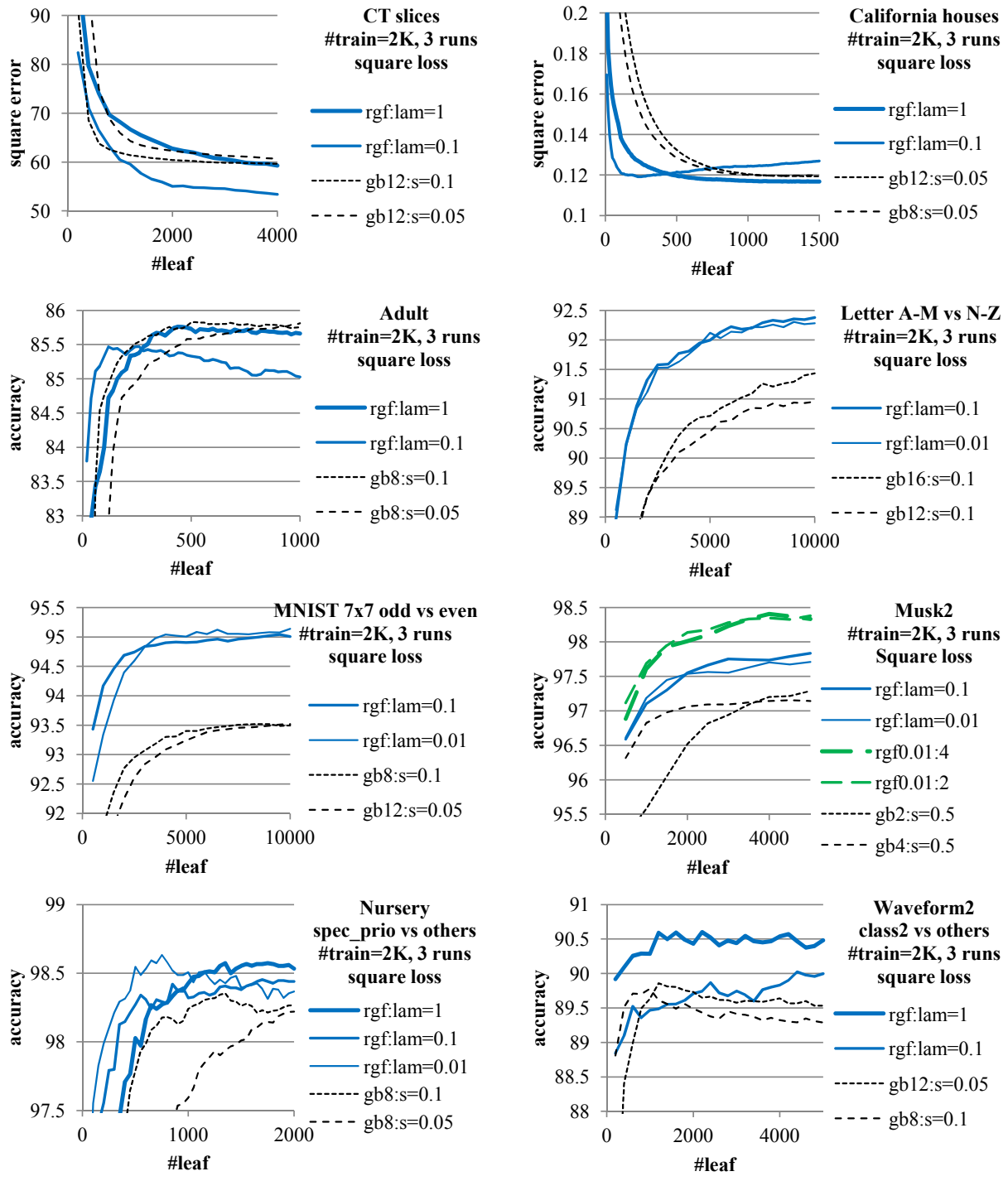

Figure 10: Performance Comparison on Real Data (square loss). Average of 3 runs. Each run used 2K training data points.

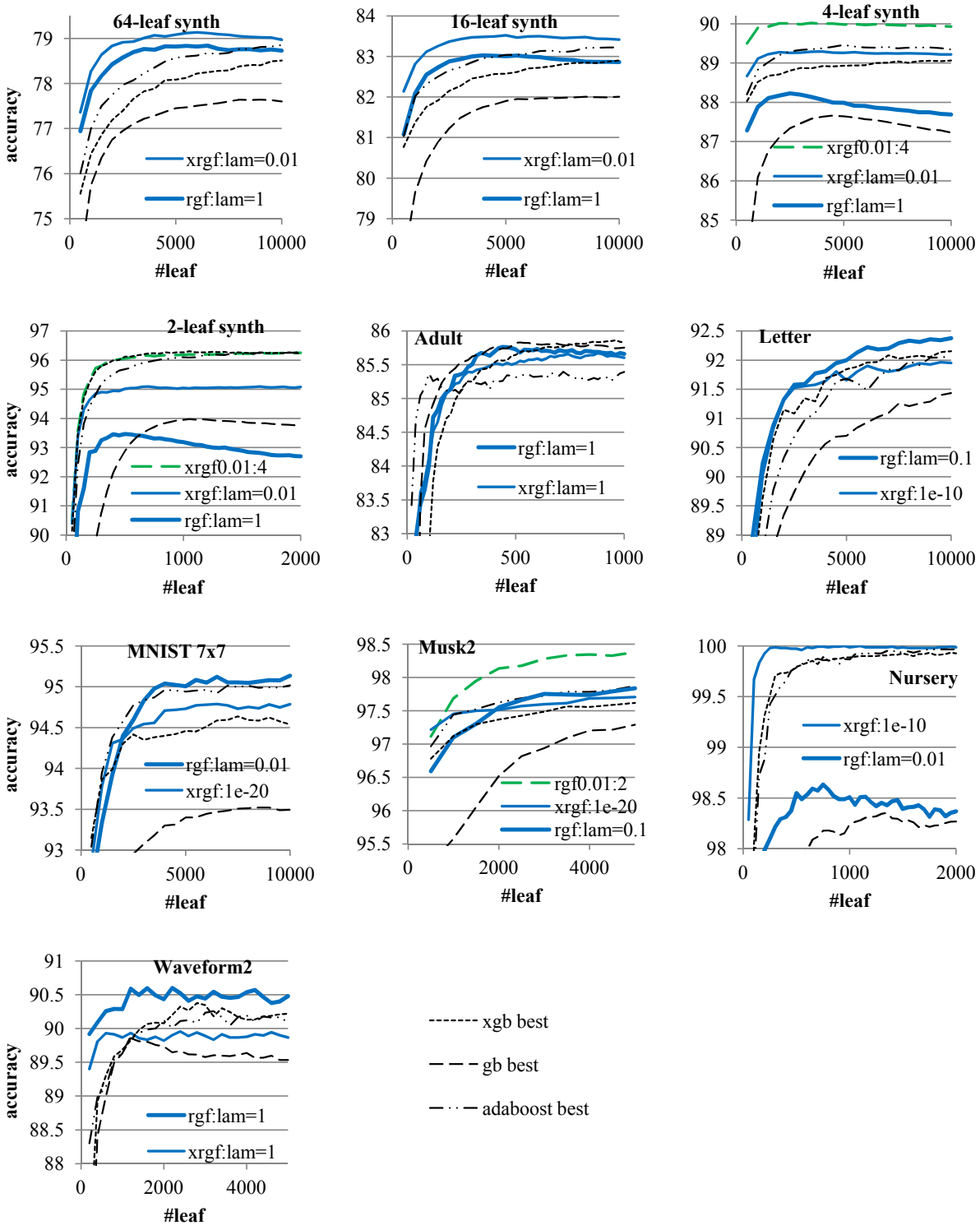

Figure 11: Performance Comparison on Synthetic Data and Real Data (exponential loss). Average of 10 (synthetic) or 3 (real) runs. Each run used 2K training data points.

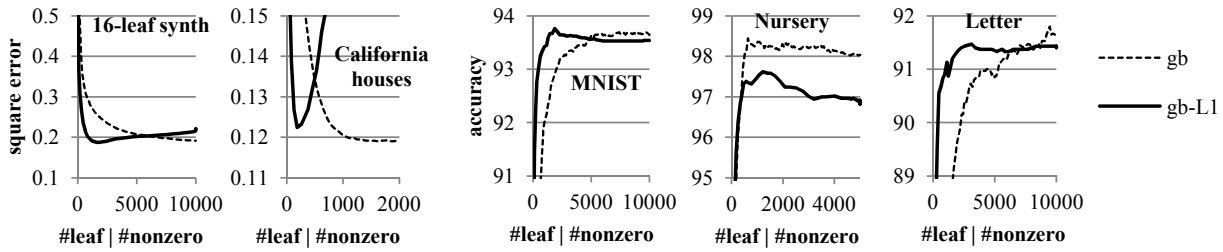

Figure 12: GBDT with L_1 regularized post-processing.

After performing GBDT until the maximum number of leaf nodes in the graphs (e.g., 10000 on MNIST) was obtained, we used the R package `glmnet` [15] 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. The solid lines plot the performance of the entire L_1 path in relation to the number of nonzero weights. The dotted lines are GBDT *without* post-processing for comparison. In both, GBDT was performed with the best-performing parameters shown in Figures 7 and 10. The graphs show that the L_1 post processing makes the models sparser so that the performance peak is obtained with fewer basis functions. But it does not necessarily improve the accuracy of the models; rather, accuracy is degraded in some cases. We view that the results support RGF's interleaving approach.

Discussion Using synthesized datasets, we have shown that RGF can perform well on the datasets whose target functions range from simple to complex. In particular, RGF with L_2 regularization on leaf-only models is effective when underlying target functions are composed of relatively complex basis functions. The min-penalty regularization with $\gamma > 1$, which encourages simpler basis functions, is effective when underlying target functions are composed of relatively simple basis functions. We have also shown that there are a number of real-world datasets on which RGF can produce sparser and more accurate models than GBDT. On the Adult, Houses, and 2-leaf classification datasets, RGF appeared to have limited success in that its peak performance rivaled those of GBDT but provided no significant improvement. Interestingly, on these datasets performances appear to converge with a smaller number of basis functions (leaf nodes) than other datasets. The indication is that RGF is particularly useful when a relatively large number of basis functions is required to model the underlying target functions, as this is one of the situations when RGF's fully-corrective update and explicit regularization can make real difference. Overall, these results are consistent with our expectation.

6.4 Comparison with other tree ensemble methods

In the previous section, we studied RGF's performance in comparison with GBDT. This section reports empirical comparison with other types of tree ensemble methods. As before, all the results are the average of either 10 or 3 runs (10 on the synthesized data and 3 on the others), each of which used 2K training data points.

6.4.1 Random forests

Random forests generate K trees from K random inputs generated by random draw of training samples and features. We used the R package `randomForest` [7] and performed random forest training with the

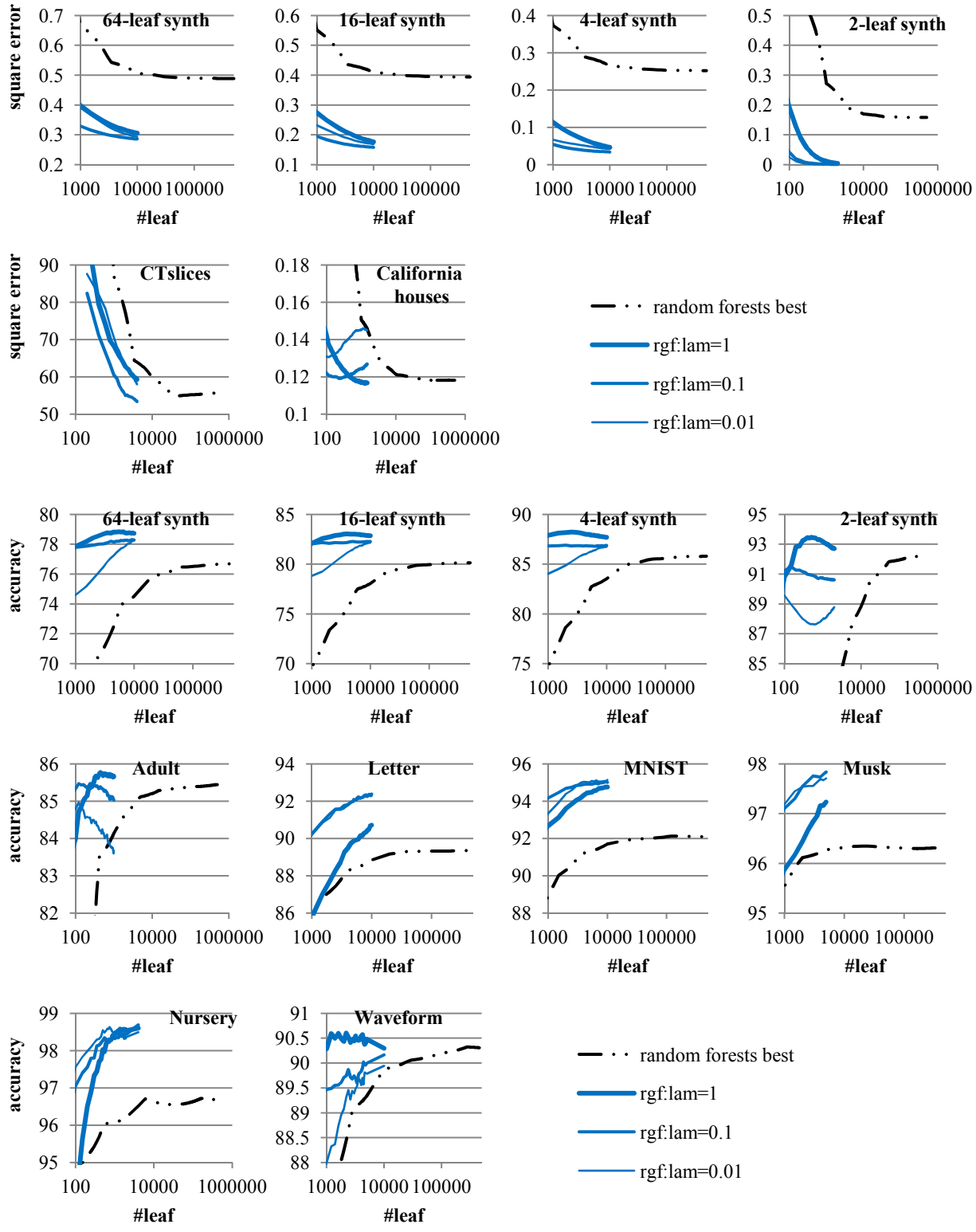

Figure 13: Comparison with Random forests. Average of 10 or 3 runs. Each run used 2K training data points.

number of randomly-drawn features k in $\{1, \frac{d}{4}, \frac{d}{2}, \frac{3d}{4}, \text{default}\}$, where d is the feature dimensionality and the 'default' is the value recommended by the system. For each of these five configurations, the number of trees was varied from 1 to 10000. In Figure 13, we show the results of the best (at the peak) random forest configuration among the five configurations in black lines. Since we are interested in compact models, we plot performance in relation to the number of leaf nodes of the forests (in the log-scale). The blue lines are RGF-L_2 with square error. On most of the datasets, RGF achieves higher accuracy (or lower error) with much fewer leaf nodes, compared with the best-performing random forests configuration.

6.4.2 Bayesian Additive Regression Trees (BART)

Bayesian Additive Regression Trees (BART) [9] is a Bayesian approach to tree ensemble learning. We used the R package `BayesTree` [8] for the experiments. It is interesting to see how RGF empirically compares to BART, as they share some high-level strategies such as explicit regularization and non-black-box approaches to tree learners. BART starts with m trees each of which consists of the root node only, and in each iteration, the m trees are either grown or pruned. This process is viewed as generating a sequence of additive models of m trees, which is converging to the posterior distribution on the "true" model. Prediction is done by taking the average of the predictions of K models from K iterations after some number of burn-in iterations. Therefore, the final model consists of $m \times K$ trees.

In addition to m and K above, BART has five parameters to control regularization. We did not attempt to tune all the parameters as [9] reports that the default/recommended values work well, and as BART training is relatively time-consuming (roughly 100 times longer than GBDT according to [9]). After preliminary experiments, we found that performance can often be improved by appropriately choosing the weight shrinkage parameter k from $\{1, 2, 3\}$ while the others fixed to the default values. Below we report the results of the best BART configuration (at its peak) among these three configurations.

Figure 14 shows performance in relation to the number of trees in the final model (in the log-scale), which is $m \times K$ for BART where $m = 200$ (default) and K (the number of the additive models to be averaged) varies from 1 to 1000. Unlike previous figures, the model size is represented by the number of trees instead of leaf nodes since the BART package does not provide the number of leaf nodes. The black lines represent the best BART configuration, and the blue solid lines are RGF-L_2 with square error. The CT slices results are not shown as BART could not complete the training due to memory shortage. The BART performance is highest with 200,000 trees (the tested maximum) and relatively low with a small number of trees. On most of the tested datasets, at least one of the RGF configurations exceeds the BART's best performance with 100 to 1000 trees.

Although BART and RGF share some high-level strategies, an interesting difference is that BART does not attempt to keep models compact. As a result, in our experiments, RGF required roughly 100 times fewer trees to achieve performance either better or comparable accuracy compared with BART. This means that prediction using the models obtained by RGF could be 100 times faster than prediction using the models obtained by BART, which may be significant in some applications.

6.5 Running time

We have shown that RGF often achieves higher accuracy than GBDT, but this is done at the cost of additional computational complexity mainly for fully-corrective weight updates. In this section 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

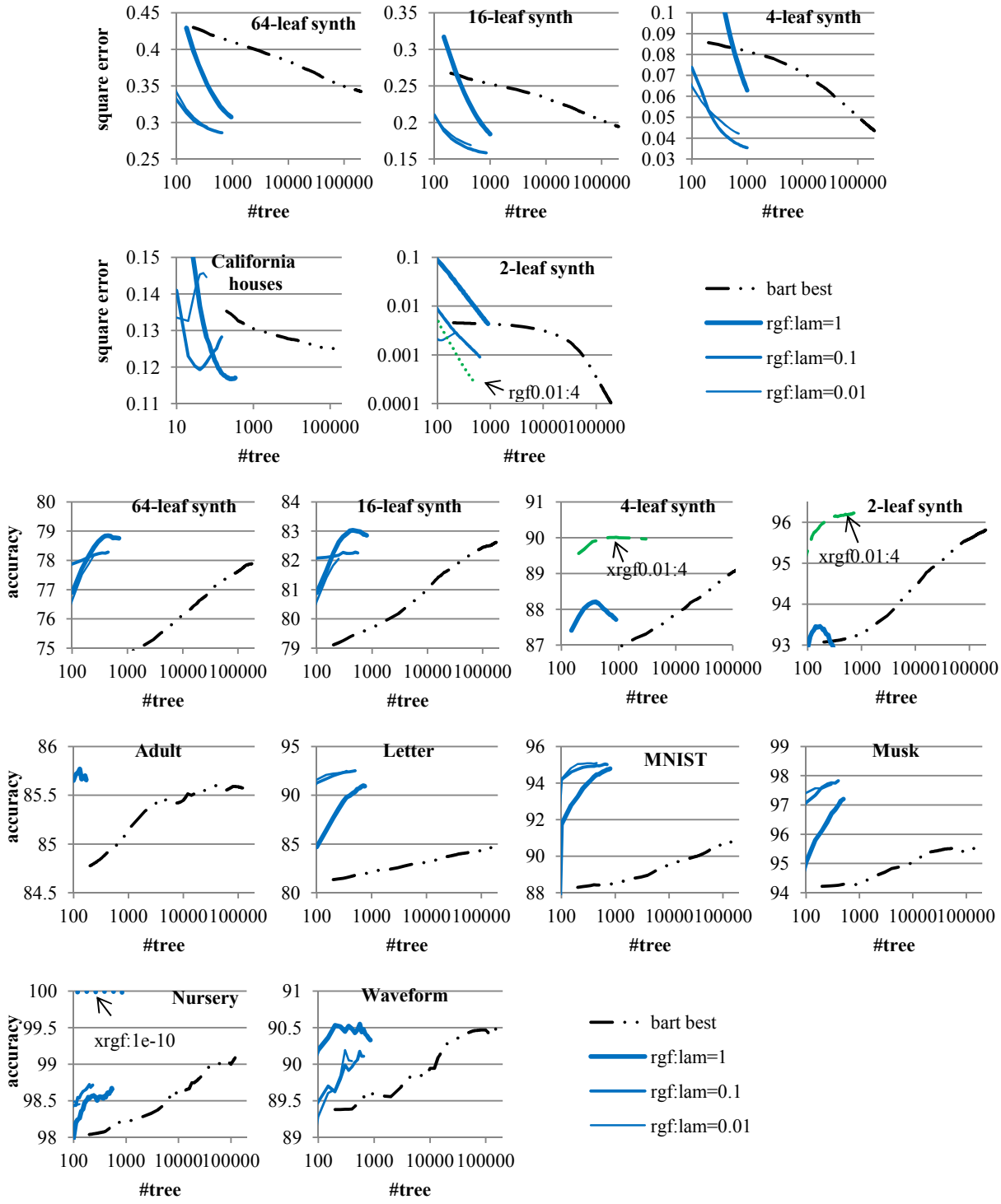

Figure 14: Comparison with BART. Average of 10 or 3 runs. Each run used 2K training data points.

data	RGF	GBDT	ratio
Nursery	0.60	0.36	1.7
Houses	0.92	0.41	2.2
Adult	0.60	0.37	1.6
Letter	5.43	1.40	3.9
16-leaf(C)	6.40	1.36	4.7

data	RGF	GBDT	ratio
Waveform	8.36	1.41	5.9
16-leaf(R)	10.7	3.70	2.9
MNIST 7x7	23.6	8.75	3.6
Musk	28.7	11.2	2.6
CT slices	50.2	22.6	2.2

Figure 15: Average training time in seconds. 2000 data points. RGF: average over 3 configurations of RGF-L_2. GBDT: average over 9 best-performing configurations.

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. As in our experiments, assume that only the most recently-grown tree is searched during forest building and weight optimization is done by coordinate descent with a fixed number of iterations. First, we consider running time excluding the processing for regularization as it varies with the type of regularizer. The part to grow the forest 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 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 with respect to one leaf weight is contained in the node of interest and runs in $O(1)$. The extra running time for that 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 $O(nd\ell) + O(\ell z^2)$ (instead of $O(nd\ell z)$, which could be large with a large amount of training data). The extra running time during weight optimization is $O(\ell cz)$, and the constant part can be substantially reduced by efficient implementation. Details are shown in the Appendix.

Empirical running time In Figure 15, we show the elapsed time of RGF on the tested datasets in comparison with GBDT. The RGF column shows the average over three RGF-L_2 configurations with $\lambda \in \{1, 0.1, 0.01\}$. The GBDT column shows the average over the best-performing configurations, which are 9 combinations of tree size and the shrinkage parameter that achieved the best performance on at least one of the 8 synthesized data types or the 8 real-world datasets on average. For both, the loss function was square loss; ℓ was set to the maximum value of the x-axis in Figure 7–10; and n was set to 2000 as before. RGF training mostly took 2–5 times longer than GBDT, as shown in the right-most column. But even so, RGF training runs conveniently fast in less than a minute in most of these settings. Even if training data size is increased by 10 times, it will mostly take less than 5 minutes as RGF training time is linear in n.

Figure 16 plots running time in relation to tree size z which is approximated by the average tree size. The

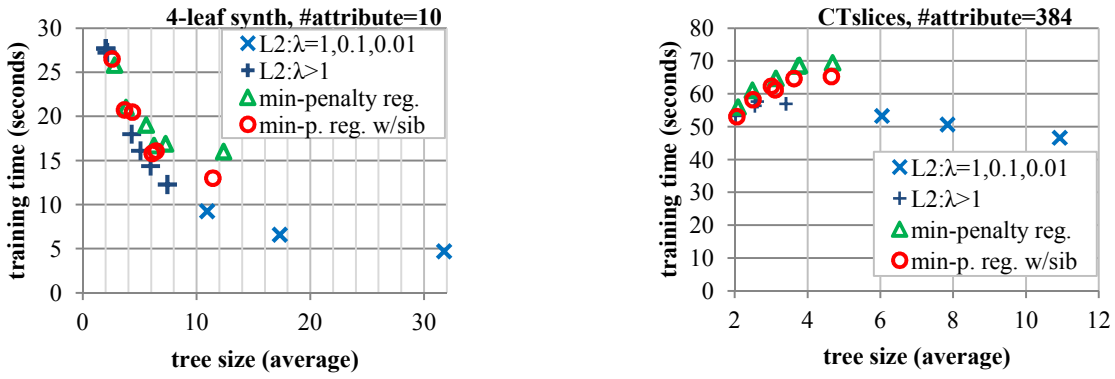

Figure 16: Training Time of RGF. The circles and triangles are RGF with min-penalty regularizers with and without the zero-to-sum sibling constraints, respectively.

points X's are RGF-L_2 with λ set to practically useful values $\{1, 0.1, 0.01\}$. The points +'s are RGF-L_2 with much larger λ, which would not be used in practical applications but are shown for running time analysis. The triangles and circles are with min-penalty regularizers with $\lambda \in \{\frac{1}{\gamma}, \frac{0.1}{\gamma}, \frac{0.01}{\gamma}\}$ with $\gamma \in \{2, 4\}$. On each dataset, n, ℓ, and c were fixed to the same values as in Figure 15. On the 4-leaf synthesized dataset, on which d is relatively small, weight optimization in $O(\frac{n\ell c}{z})$ dominates over forest building in $O(nd\ell)$, so the running time is nearly inversely proportional to z. The additional running time for min-penalty regularization seems negligible with small tree sizes z and more prominent with larger z. The running time with min-penalty regularization tends to be longer than RGF-L_2 with practical values of λ, but that is not only because of additional computation but also because setting $\gamma > 1$ penalizes deeper nodes and makes tree size z smaller. By contrast, on CT slices, whose d is relatively large, forest building in $O(nd\ell)$ dominates over weight optimization in $O(\frac{n\ell c}{z})$, so the influence of the tree size z on RGF-L_2 is small.

As seen above, training could take longer with min-penalty regularizers. To save training time, rather than mechanically training combinations of λ and γ, first one should try RGF-L_2 and find a good value for λ (based on validation data) and then try min-penalty regularizers with λ no greater than the found value, as $\gamma > 1$ has an effect of further strengthening regularization.

7 Conclusion

This paper introduced a new method that learns a nonlinear function by using an additive model over nonlinear decision rules. Instead of 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 more effective and principled than boosted decision trees. Specially, the new method can achieve smaller sized forests with more accurate predictions, especially for more complex nonlinear functions that require stronger regularization. Our empirical studies also demonstrated these advantages.

References

[1] Francis Bach. Exploring large feature spaces with hierarchical multiple kernel learning. In *NIPS' 2008*, 2008.

[2] Francis Bach. High-dimensional non-linear variable selection through hierarchical kernel learning. Technical Report 00413473, HAL, 2009.

[3] Richard Baraniuk, Volkan Cevher, Marco F. Duarte, and Chinmay Hegde. Model based compressive sensing. *IEEE Transactions on Information Theory*, 56:1982–2001, 2010.

[4] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth Advanced Books and Software, Belmont, CA, 1984.

[5] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, August 1996.

[6] Leo Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[7] Leo Brieman, Adele Cutler, Andy Liaw, and Matthew Wiener. Package 'randomForest', 2010.

[8] Hugh Chipman and Robert mcCulloch. Package 'BayesTree', 2010.

[9] Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. BART: Bayesian additive regression trees. *The Annals of Applied Statistics*, 4(1):266–298, 2010.

[10] A. Frank and A. Asuncion. UCI machine learning repository [http://archive.ics.uci.edu/ml], 2010. University of California, Irvine, School of Information and Computer Sciences.

[11] Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, 1997.

[12] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *JMLR*, 4:933–969, 2003.

[13] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *ICML' 99*, pages 124–133, 1999.

[14] Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29, 2001.

[15] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Package 'glmnet', 2011.

[16] Jerome H. Friedman and Bogdan E. Popescu. Importance sampled learning ensembles. Technical report, Tech Report, 2003.

[17] R. Herbrich, T. Graepel, and K. Obermayer. Large margin rank boundaries for ordinal regression. In B. Schölkopf A. Smola, P. Bartlett and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 115–132. MIT Press, 2000.

[18] Junzhou Huang, Tong Zhang, and Dimitris Metaxas. Learning with structured sparsity. *JMLR*, 2011. to appear.

[19] L. Jacob, G. Obozinski, and J. Vert. Group lasso with overlap and graph lasso. In *Proceedings of ICML*, 2009.

[20] R. Jenatton, J.-Y. Audibert, and F. Bach. Structured variable selection with sparsity-inducing norms. Technical report, Tech Report: arXiv:0904, 2009.

[21] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[22] Robert E. Schapire. The boosting approach to machine learning: An overview. *Nonlinear Estimation and Classification*, 2003.

[23] Shai Shalev-Shwartz, Nathan Srebro, and Tong Zhang. Trading accuracy for sparsity in optimization problems with sparsity constraints. *Siam Journal on Optimization*, 20:2807–2832, 2010.

[24] M. Warmuth, J. Liao, and G. Ratsch. Totally corrective boosting algorithms that maximize the margin. In *Proceedings of the 23rd international conference on Machine learning*, 2006.

[25] Tong Zhang and Bin Yu. Boosting with early stopping: Convergence and consistency. *The Annals of Statistics*, 33:1538–1579, 2005.

A Appendix

A.1 Convergence of Algorithm 4

To show that Algorithm 4 converges, let us express the algorithm using matrix multiplication as follows. Let J be the number of internal nodes, and define a matrix $\mathbf{A} \in \mathbb{R}^{J \times J}$ and a column vector $\mathbf{b} \in \mathbb{R}^J$ so that:

$$\mathbf{A}[v,w] = \begin{cases} \frac{1}{1+2\gamma} & w = p(v) \\ \frac{\gamma}{1+2\gamma} & p(w) = v \\ 0 & \text{otherwise} \end{cases}, \quad \mathbf{b}[v] = \sum_{p(w)=v, w \in L_T} \frac{\gamma \alpha_w}{1+2\gamma}$$

Define $\mathbf{B} \in \mathbb{R}^{(J+1)\times(J+1)}$ and $\bar{\beta} \in \mathbb{R}^{n+1}$ so that:

$$\mathbf{B} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ 0 & 1 \end{bmatrix}, \quad \bar{\beta} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

Then since we have:

$$\mathbf{B}^m = \begin{bmatrix} \mathbf{A}^m & \sum_{i=0}^{m-1} \mathbf{A}^i \mathbf{b} \\ 0 & 1 \end{bmatrix},$$

m iterations of Algorithm 4 is equivalent to:

$$\bar{\beta}_{v,m} \leftarrow \begin{cases} \alpha_v & v \in L_T \\ \left(\mathbf{B}^m \bar{\beta}\right)[v] = \left(\sum_{i=0}^{m-1} \mathbf{A}^i \mathbf{b}\right)[v] & v \notin L_T \end{cases}$$

It is well known that for any square matrix $\mathbf{Z}$, if $\|\mathbf{Z}\|_p < 1$ for some $p \geq 1$ then $\mathbf{I} - \mathbf{Z}$ is invertible and $(\mathbf{I} - \mathbf{Z})^{-1} = \sum_{k=0}^{\infty} \mathbf{Z}^k$. Therefore, it suffices to show that $\|\mathbf{A}\|_p < 1$ for some $p \geq 1$.

First, consider the case that $\gamma = 1$. In this case, $\mathbf{A}$ is symmetric and column v (and row v) has $|N(v)|$ non-zero entries, where $N(v)$ denote the set of the internal nodes adjacent to v, and all the non-zero entries are $1/3$.

Using the fact that $|N(v)| \leq 3$ for $v \neq o_T$ and $|N(o_T)| \leq 2$, for any $\mathbf{x} \in \mathbb{R}^J$, we have:

$$\begin{aligned} \|\mathbf{x}\|_2^2 - \|\mathbf{A}\mathbf{x}\|_2^2 =& \sum_j \mathbf{x}[j]^2 - \frac{1}{9} \sum_j \left(\sum_{k \in N(j)} \mathbf{x}[k] \right)^2 > \frac{2}{3} \sum_j \mathbf{x}[j]^2 - \frac{1}{9} \sum_j \sum_{k,\ell \in N(j), k<\ell} 2\mathbf{x}[k] \cdot \mathbf{x}[\ell] \\ >& \frac{1}{9} \sum_j \sum_{k,\ell \in N(j), k<\ell} (\mathbf{x}[k] - \mathbf{x}[\ell])^2 \geq 0 \end{aligned}$$

Therefore, $\|\mathbf{A}\|_2 < 1$.

Next suppose that $\gamma > 1$. Then we have:

$$\|\mathbf{A}\|_1 = \max_j \sum_{i=1}^{J} |A[i,j]| \leq 2 \cdot \frac{1}{1+2\gamma} + \frac{\gamma}{1+2\gamma} < 1$$

Hence, Algorithm 4 converges with $\gamma \geq 1$.

Another way to look at this is that (10) and (13) can be expressed using the matrix notation above as: $\bar{\beta} = \mathbf{A}\bar{\beta} + \mathbf{b}$. As shown above, $\mathbf{I} - \mathbf{A}$ is invertible, and therefore $\{\bar{\beta}_v\}$ with desired properties can be obtained by $\bar{\beta} = (\mathbf{I} - \mathbf{A})^{-1}\mathbf{b}$. Algorithm 4 computes it iteratively.

Our implementation used its slight variant (Algorithm 6) as it converges faster.

Algorithm 6:

forall the $v \in T$ **do** $\bar{\beta}_v \leftarrow \begin{cases} \alpha_v & v \in L_T \\ 0 & v \notin L_T \end{cases}$

for $i = 1$ **to** m **do**

  for $v \notin L_T$ *in some fixed order* **do** $\bar{\beta}_v \leftarrow \begin{cases} \frac{\bar{\beta}_{p(v)} + \sum_{p(w)=v} \gamma \bar{\beta}_w}{1+2\gamma} & v \neq o_T \\ \frac{\sum_{p(w)=v} \gamma \bar{\beta}_w}{1+2\gamma} & v = o_T \end{cases}$

end

A.2 Computational detail of the min-penalty regularization in Section 5.4.2

To optimize weights according to (8), we need to obtain the derivatives of the regularization penalty, $\frac{\partial \mathcal{G}(T(\delta_u))}{\partial \delta_u}_{|\delta_u=0}$ and $\frac{\partial^2 \mathcal{G}(T(\delta_u))}{\partial \delta_u^2}_{|\delta_u=0}$, where δ_u is the additive change to α_u, the weight of a leaf node u, and T is the tree to which node u belongs. Let $\{\bar{\rho}_v\} = \arg\min_{\{\bar{\beta}_v\}} \left\{ f(\{\bar{\beta}_v\}) : \forall v \in L_T . [\bar{\beta}_v = \alpha_v] \right\}$ so that $\mathcal{G}(T) = \lambda \cdot f(\{\bar{\rho}_v\})$, where f and $\mathcal{G}(T)$ are defined in terms of auxiliary variables as in (11) and (12).

From the derivation in the previous section, we know that $\bar{\rho}_w$ is linear in leaf weights $\{\alpha_v\}$. In particular, $\bar{\rho}_w$ for an internal node w can be written in the form of $\bar{\rho}_w = \sum_{v \in L_T} c_{w,v} \alpha_v$ with coefficients $c_{w,v}$ that are independent of leaf weights and only depend on the tree topology. Also considering $\bar{\rho}_v = \alpha_v$ for $v \in L_T$, we have:

$$\frac{\partial \bar{\rho}_w}{\partial \alpha_u} = \begin{cases} c_{w,u} & w \notin L_T \\ 1 & w = u \\ 0 & w \neq u \;\&\; w \in L_T \end{cases} .$$

$c_{w,u}$ can be obtained by Algorithm 4 (or 6) with input of: $\alpha_u = 1$; $\alpha_t = 0$ for $t \neq u$; and the topological structure of T. Also using $\frac{\partial^2 \bar{\rho}_w}{\partial \alpha_v^2} = 0$, it is straightforward to derive from the definition of f in (12) that:

$$\frac{\partial \mathcal{G}(T(\delta_u))}{\partial \delta_u}_{|\delta_u=0} = \lambda \sum_{w \in T} \gamma^{d_w} \rho_w \frac{\partial \rho_w}{\partial \alpha_u} \;\; , \;\; \frac{\partial^2 \mathcal{G}(T(\delta_u))}{\partial \delta_u^2}_{|\delta_u=0} = \lambda \sum_{w \in T} \gamma^{d_w} \left(\frac{\partial \rho_w}{\partial \alpha_u} \right)^2 , \tag{15}$$

$$\text{where } \rho_w = \bar{\rho}_w - \bar{\rho}_{p(w)} \text{ if } w \neq o_T; \bar{\rho}_w \text{ otherwise.} \tag{16}$$

The optimum change to the leaf weight α_u can be computed using these quantities. Loss reduction caused by node split can be similarly estimated.

For efficient implementation The key to efficient implementation is to make use of the fact that in the course of training certain quantities are locally invariant, by storing and reusing the invariant quantities. First, since the iterative algorithm is relatively complex, it should be executed as infrequently as possible. As noted above, $\bar{\rho}_w$'s partial derivatives only depend on the topological structure of the tree, so they need to be computed only when the tree topology changes. When δ_v is added to α_v, $\bar{\rho}_w$ should be updated through $\bar{\rho}_w \leftarrow \bar{\rho}_w + \delta_v \frac{\partial \bar{\rho}_w}{\partial \alpha_v}$ instead of running the iterative algorithm. Second, consider the process of evaluating loss reduction of all possible splits of some node, which is fixed during this process. Using the notation in Section 6.5, the partial derivatives of the regularization penalty similar to (15) are referred to $O(nd)$ times in this process, but they are invariant and need to be computed just once. The change in penalty caused by node split is evaluated also $O(nd)$ times, and one can make each evaluation run in $O(1)$ instead of $O(z)$ by storing invariant quantities. To see this, as in Section 5.3.1, consider splitting a node associated with weight

α, and let u_k for $k = 1, 2$ be the new leaf nodes after split with weight $\alpha + \delta_k$. Write $\tilde{T}(\delta_1, \delta_2)$ for the new tree. Define $\bar{\rho}_w$ and ρ_w as above but on $\tilde{T}(0, 0)$ instead of T. To simplify notation, let $\dot{\rho}_{w,k} = \frac{\partial \rho_w}{\partial \alpha_{u_k}}$. Due to symmetry, we have: $\dot{\rho}_{w,1} = \dot{\rho}_{w,2}$ for $w \neq u_1$ & $w \neq u_2$. The change in penalty is: $\mathcal{G}(\tilde{T}(\delta_1, \delta_2)) - \mathcal{G}(T) = \Big(\mathcal{G}(\tilde{T}(\delta_1, \delta_2)) - \mathcal{G}(\tilde{T}(0, 0))\Big) + \Big(\mathcal{G}(\tilde{T}(0, 0)) - \mathcal{G}(T)\Big)$. Since the second term is invariant during this process, it needs to be computed only once. To ease notation, let

$$\Delta_v = \gamma^{d_v} \left(\rho_v \sum_{k=1}^{2} \delta_k \dot{\rho}_{v,k} + \frac{1}{2} \left(\sum_{k=1}^{2} \delta_k \dot{\rho}_{v,k} \right)^2 \right) .$$

Then the first term in the penalty change can be written as:

$$\Big(\mathcal{G}(\tilde{T}(\delta_1, \delta_2)) - \mathcal{G}(\tilde{T}(0, 0))\Big) = \frac{1}{2} \sum_{v \in \tilde{T}} \gamma^{d_v} \left(\rho_v + \sum_{k=1}^{2} \delta_k \dot{\rho}_{v,k} \right)^2 - \frac{1}{2} \sum_{v \in \tilde{T}} \gamma^{d_v} \rho_v^2 = \sum_{v \in \tilde{T}} \Delta_v$$

$$= (\delta_1 + \delta_2) \sum_{v \in \tilde{T} - \{u_1, u_2\}} \gamma^{d_v} \rho_v \dot{\rho}_{v,1} + \frac{1}{2} (\delta_1 + \delta_2)^2 \sum_{v \in \tilde{T} - \{u_1, u_2\}} \gamma^{d_v} \dot{\rho}_{v,1}^2 + \Delta_{u_1} + \Delta_{u_2} .$$

Here $\sum_{v \in \tilde{T} - \{u_1, u_2\}} \gamma^{d_v} \rho_v \dot{\rho}_{v,1}$ and $\sum_{v \in \tilde{T} - \{u_1, u_2\}} \gamma^{d_v} \dot{\rho}_{v,1}^2$ are invariant during this process and can be precomputed; therefore, each evaluation of penalty differences runs in $O(1)$.

A.3 Computational detail of the regularizer in Section 5.4.3

The same basic ideas above can be applied to make efficient implementation of the regularizer with sum-to-zero sibling constraints. To optimize the additive change δ_u of the leaf weight α_u, let $\{\rho_v\}$ be the arguments that minimize (9) so that $\mathcal{G}(T) = \lambda \cdot \sum_{v \in T} \gamma^{d_v} \rho^2 / 2$. Then the partial derivatives of $\mathcal{G}(T(\delta_u))$ are obtained by (15). From (14), we have:

$$\frac{\partial \rho_w}{\partial \alpha_u} = \begin{cases} 2^{-d_{u_k}} & w = o_T \\ 2^{d_w - d_{u_k} - 1} & w \neq o_T, w \in A(u) \ \ (w \text{ is either } u \text{ or } u\text{'s ancestor}) \\ -2^{d_w - d_{u_k} - 1} & w \notin A(u), p(w) \in A(u) \ \ (w \text{ is } u\text{'s ancestor's sibling}) \\ 0 & \text{otherwise} \end{cases} .$$

Optimization can be done using these quantities.

Regarding efficient implementation, one difference from the regularizer without the sibling constraints above is that $\mathcal{G}(\tilde{T}(0, 0)) = \mathcal{G}(T)$ with this regularizer and therefore $\mathcal{G}(\tilde{T}(0, 0))$ does not have to be computed.