Gradient Boosted Decision Trees for High Dimensional Sparse Output

Si Si[1] Huan Zhang[2] S. Sathiya Keerthi[3] Dhruv Mahajan[4] Inderjit S. Dhillon[5] Cho-Jui Hsieh[2]

Abstract

In this paper, we study the gradient boosted decision trees (GBDT) when the output space is high dimensional and sparse. For example, in multilabel classification, the output space is a L-dimensional 0/1 vector, where L is number of labels that can grow to millions and beyond in many modern applications. We show that vanilla GBDT can easily run out of memory or encounter near-forever running time in this regime, and propose a new GBDT variant, GBDT-SPARSE, to resolve this problem by employing L_0 regularization. We then discuss in detail how to utilize this sparsity to conduct GBDT training, including splitting the nodes, computing the sparse residual, and predicting in sublinear time. Finally, we apply our algorithm to extreme multilabel classification problems, and show that the proposed GBDT-SPARSE achieves an order of magnitude improvements in model size and prediction time over existing methods, while yielding similar performance.

1. Introduction

Gradient boosted decision tree (GBDT) is a powerful machine-learning technique that has a wide range of commercial and academic applications and produces state-of-the-art results for many challenging data mining problems. The algorithm builds one decision tree at a time to fit the residual of the trees that precede it. GBDT has been widely used recently mainly due to its high accuracy, fast training and prediction time, and small memory footprint.

In this paper, we study the GBDT algorithm for problems with high-dimension and sparse output space. Extreme multi-label learning and multi-class classification belong to this problem, where the goal is to automatically assign one or a subset of relevant labels from a very large label set. Dealing with problems with high dimensional output leads to multiple computational challenges. In this paper we mainly focus on two important issues that limit the application of the existing methods to real world applications: **prediction time and model size**. As the output space size increases, these dimensions become the bottleneck, both during training and testing. As an example, if a one-versus-all model is used on a classification problem with 1 million labels, then we need to evaluate 1 million models for any testing sample. If these models cannot be kept in memory, reading them from disks will further increase the prediction time substantially. The linear dependency on number of labels makes most of the existing approaches very slow during testing, especially when we do not want to access the cloud for every test point.

The computation of GBDT is also prohibitively expensive for applications with high dimensional sparse output. At each iteration, GBDT builds a regression tree that fits the residuals from the previous trees. The density of the residual grows dramatically even after just one single iteration, and it will soon become an L by N dense matrix where N is number of samples and L is the number of labels (size of output space). As a consequence, at least $O(NL)$ time and memory are required to build GBDT trees. This makes GBDT infeasible for large scale applications where N and L can be both large, e.g., several millions.

Our goal is to develop a new approach for problems with high-dimensional and sparse output spaces that achieves faster prediction time and smaller model size than existing algorithms, but has similar prediction accuracy and training time. To this end, we develop the first Gradient Boosted Decision Tree (GBDT) algorithm for high dimensional and sparse output, with applications in extreme multilabel learning problems. We make the crucial observation that each data point has very few labels; based on that we solve a L_0 regularized optimization problem to enforce the prediction of each leaf node in each tree to have only a small number (k) of nonzero elements or labels. Hence, after T trees have been added during GBDT iterations, there will be at most Tk nonzero gradients for any data point. Another important challenge discussed in this paper is pre-

[1] Google Research, Mountain View, USA [2] University of California at Davis, Davis, USA [3] Microsoft, Mountain View, USA [4] Facebook, Menlo Park, USA [5] University of Texas at Austin, Austin, USA. Correspondence to: Cho-Jui Hsieh <chohsieh@ucdavis.edu>.

Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017. Copyright 2017 by the author(s).

diction time. Given the sparsified output, we discuss efficient algorithms to conduct prediction for both top-K recommendation or the whole sparse output vector. Finally, we discuss how to handle sparse data, where each feature is active only on a small fraction of training examples. To handle this, we use several unsupervised and supervised dimensional reduction algorithms as pre-processing steps. This also has the positive effect of reducing the search space of each node.

For extreme multi-label applications, our algorithm has competitive accuracy compared with existing state-of-the-art algorithms, while achieving substantial reductions in prediction time and model size. For example, on the Wiki10-31K dataset with 30938 labels, our method takes only 1.3 secs. for prediction and achieves 84.34% accuracy with a model size of 85.8MB, while the state-of-the-art fast multi-label method FASTXML takes more than 10 secs. to achieve 82.71% accuracy and uses 853.5MB memory to store the model. Our method can be efficiently parallelized and achieve almost linear speed up in multi-core settings.

The rest of the paper is outlined as follows. We present related work in Section 2. Traditional GBDT is explained in Section 3. Our main algorithm GBDT-SPARSE is proposed and analyzed in Section 4. Experimental results are given in Section 5. We present conclusions in Section 6.

2. Related Work

Ensemble methods have shown excellent performance in various machine learning applications and analytics competitions, e.g., Kaggle challenges. Common ensemble methods include random forests (Liaw & Wiener, 2002), bagging (Breiman, 1996), and boosting (Schapire, 1999; Friedman, 2001; 2002). Out of these, boosting is very effective in reducing model size and prediction time since it uses the output of previous models to train the next one.

Many classical boosting methods have shown their efficiency in practice. Among them, gradient boosted decision trees (GBDT) (Friedman, 2001; 2002) has received much attention because of its high accuracy, small model size and fast training and prediction. It been widely used for binary classification, regression, and ranking. In GBDT, each new tree is trained on the per-point residual defined as the negative of gradient of loss function wrt. output of previous trees. GBDT is well studied in the literature: some research has been done to speed up the computation of GBDT under different parallel settings (multi-core or distributed), e.g., XGBoost (Chen & Guestrin, 2016), LightGBM,[1] PLANET (Panda et al., 2009), PV-Tree (Meng et al., 2016), and YGGDRASIL(Abuzaid et al., 2016) or exploit its benefit for different machine learning applications, e.g., using GBDT for CRFs (Chen et al., 2015). However, to the best of our knowledge none of them can be efficiently applied to problems with high dimensional output.

Recently, machine learning problems with high dimensional output have drawn considerable attention. Two popular and representative problems are extreme multi-class classification and extreme multi-label learning problem (Prabhu & Varma, 2014; Bhatia et al., 2015; Yu et al., 2014; Agrawal et al., 2013; Jasinska et al., 2016; Si et al., 2016) and both deal with very large number of labels. LOMtree proposed in (Choromanska & Langford, 2015) constructs trees for extreme multi-class problem, and obtains training and test time complexity logarithmic in the number of classes, but its extension to multi-label case is not straightforward. Many algorithms have been developed to solve extreme multi-label learning problem. For instances, embedding based methods LEML (Yu et al., 2014) and SLEEC (Bhatia et al., 2015) project the labels and features to some low-dimensional space while preserving distances either with the neighboring label vectors or the full training set; PLT(Jasinska et al., 2016) considers using sparse probability estimates restricted to the most probable labels to speed up the F-measure maximization for extreme multi-label learning; PD-Sparse (Yen et al., 2016) formulates multilabel learning problem as a primal-dual sparse problem given by margin-maximizing loss with L_1 and L_2 penalties. Tree based methods (Prabhu & Varma, 2014; Agrawal et al., 2013) generalize the impurity measures defined for binary classification and ranking tasks to multi-label scenario for splitting the nodes, but require hundreds of trees to achieve good accuracy. FASTXML (Prabhu & Varma, 2014) uses NDCG based ranking loss function and solves a non-convex optimization problem to find a sparse linear separator for splitting each node. All the approaches discussed above either do not give good accuracy (Yu et al., 2014), or, require large sized models with high prediction times to do so (Prabhu & Varma, 2014).

In contrast, to solve extreme multi-label learning problem, our method is based on GBDT and hence requires only a few trees to build a good model. During training, we also enforce sparsity in the label vector at each leaf node to reduce the model size and prediction time. Our approach is different from FASTXML in three aspects:(1) we do not need to solve a non-convex optimization at each node, but, rather do a much simpler and faster feature selection; (2) we follow the idea of GBDT to build trees, while FASTXML is a random forest based method; (3) we can achieve similar accuracy as FASTXML, but with much faster prediction time and smaller model size.

3. Background

We first discuss the original GBDT algorithm, and present the difficulty when applying GBDT to solve problems with

[1] https://github.com/Microsoft/LightGBM

high dimensional output space.

GBDT for binary classification Let us explain the main idea behind GBDT using binary classification, in which a scalar score function is formed to distinguish the two classes. Given training data $X = \{\boldsymbol{x}_i\}_{i=1}^N$ with $\boldsymbol{x}_i \in R^D$ and their labels $Y = \{y_i\}_{i=1}^N$ with $y_i \in \{0, 1\}$, the goal is to choose a classification function $F(\boldsymbol{x})$ to minimize the aggregation of some specified loss function $\mathcal{L}(y_i, F(\boldsymbol{x}_i))$:

$$F^* = \operatorname*{argmin}_F \sum_{i=1}^N \mathcal{L}(y_i, F(\boldsymbol{x}_i)). \tag{1}$$

Gradient boosting considers the function estimation F in an additive form:

$$F(\boldsymbol{x}) = \sum_{m=1}^T f_m(\boldsymbol{x}), \tag{2}$$

where T is the number of iterations. The $\{f_m(\boldsymbol{x})\}$ are designed in an incremental fashion; at the m-th stage, the newly added function, f_m is chosen to optimize the aggregated loss while keeping $\{f_j\}_{j=1}^{m-1}$ fixed.

Each function f_m belongs to a set of parametrized 'base-learners'; let θ denote the vector of parameters of the the base-learner. GBDT uses decision trees to be the base learners. For this choice, θ consists of parameters that represent the tree structure, such as the feature to split in each internal node, the threshold for splitting each node, etc.

At stage m, we form an approximate function of the loss:

$$\begin{aligned}&\mathcal{L}(y_i, F_{m-1}(\boldsymbol{x}_i) + f_m(\boldsymbol{x}_i)) \approx \\ &\mathcal{L}(y_i, F_{m-1}(\boldsymbol{x}_i)) + g_i f_m(\boldsymbol{x}_i) + \frac{1}{2} f_m(\boldsymbol{x}_i)^2, \end{aligned} \tag{3}$$

where $F_{m-1}(\boldsymbol{x}_i) = \sum_{j=1}^{m-1} f_j(\boldsymbol{x}_i)$ and

$$g_i = \frac{\partial \mathcal{L}(y_i, F(\boldsymbol{x}_i))}{\partial F(\boldsymbol{x}_i)} \mid_{F(\boldsymbol{x}_i) = F_{m-1}(\boldsymbol{x}_i)} .$$

Note that throughout the paper we will only take differentiation with the second parameter of $\mathcal{L}(\cdot, \cdot)$, so we define $\mathcal{L}'(y_i, F_{m-1}(\boldsymbol{x}_i))$ to be the above differentiation.

We want to choose f_m to minimize the right hand side of (3), which can be written as the following minimization problem:

$$\arg\min_{f_m} \sum_{i=1}^N \frac{1}{2}(f_m(\boldsymbol{x}_i) - g_i)^2. \tag{4}$$

Since only the direction is fitted, a suitable step size (shrinkage parameter) is usually applied to f_m before it is added to F_{m-1}. The advantage of this gradient boosting approach is that only the expression of the gradient varies for different loss functions, while the rest of the procedure, and in particular the decision tree induction step, remains the same for different loss functions.

4. Proposed Algorithm (GBDT-SPARSE)

Now we discuss the problem with sparse high dimensional output. For input data $X = \{\boldsymbol{x}_i\}_{i=1}^N$ with $\boldsymbol{x}_i \in R^D$, we assume the corresponding output $Y = \{\boldsymbol{y}_i\}_{i=1}^N$ with $\boldsymbol{y}_i \in \mathbb{R}^L$ are high-dimensional and sparse—L is very large but each $\boldsymbol{y}_i$ only contains a few nonzero elements. We denote the average number of nonzero elements $S = \sum_i \|\boldsymbol{y}_i\|_0 / N$, and $S \ll L$. Multilabel learning is an example, where each $\boldsymbol{x}_i$ is the input features for a training sample, $\boldsymbol{y}_i \in \{0, 1\}^L$ where L is the number of labels, and $(\boldsymbol{y}_i)_q = 1$ if sample i has label q.

Now we discuss the proposed GBDT-SPARSE algorithm. For a general loss function with high dimensional output $\boldsymbol{y}_i$, we consider

$$F^* = \operatorname*{argmin}_F \sum_{i=1}^n \mathcal{L}(\boldsymbol{y}_i, F(\boldsymbol{x}_i)) + R(F), \tag{5}$$

where $R(F)$ is the regularization term. For simplicity we assume an L_2 regularization, so

$$R(F) = \lambda \sum_{m=1}^T \sum_{j=1}^{M_m} \|w_j^m\|^2, \tag{6}$$

where $f_m(\boldsymbol{x}) = w_{J(x)}^m$ with $J(x) : R^D \to M_m$ representing the tree structure which maps a data point $\boldsymbol{x}$ into one of the M_m leaves of the m-th tree, and $w_j^m \in \mathbb{R}^L$ is the prediction vector of the j-th leaf node in the m-th tree.

We assume $\mathcal{L}$ is differentiable and satisfies the following properties:

1. $\mathcal{L}(y, z)$ is decomposable:

$$\mathcal{L}(y, z) = \sum_{q=1}^L \ell(y_q, z_q). \tag{7}$$

2. Each $\ell(\cdot, \cdot)$ satisfies that

$$\ell'(y_q, z_q) = 0 \text{ if } y_q = z_q. \tag{8}$$

Examples include but not limited to the square loss: $\ell(y_q, z_q) = (y_q - z_q)^2$ and the square hinge loss (note that this is the square-hinge loss with center shifted to 0.5 and width scaled to 0.5):

$$\ell(y_q, z_q) = \begin{cases} \max(1 - z_q, 0)^2 & \text{if } y_q = 1 \\ \max(z_q, 0)^2 & \text{if } y_q = 0 \end{cases} \tag{9}$$

Using the same Taylor expansion, at each iteration we want to construct f_m by solving

$$\begin{aligned}&\mathcal{L}(\boldsymbol{y}_i, F_{m-1}(\boldsymbol{x}_i) + f_m(\boldsymbol{x}_i)) \approx \\ &\mathcal{L}(\boldsymbol{y}_i, F_{m-1}(\boldsymbol{x}_i)) + \langle \boldsymbol{g}_i, f_m(\boldsymbol{x}_i) \rangle + \frac{1}{2}\|f_m(\boldsymbol{x}_i)\|^2, \end{aligned} \tag{10}$$

where $\boldsymbol{g}_i$ is the L-dimensional gradient for the i-th sample with $(\boldsymbol{g}_i)_q = \ell'((\boldsymbol{y}_i)_q, (F_{m-1}(\boldsymbol{x}_i))_q)$. Following the same steps as the previous section, for each tree we want to find the cut value to minimize the following objective function:

$$\min_{f_m} \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{g}_i - f_m(\boldsymbol{x}_i)\|_2^2 + \lambda \sum_{j=1}^{M_m} \|w_j^m\|_2^2. \tag{11}$$

Vanilla extension of GBDT to high-dimensional output space. As in most decision tree induction methods, we follow a greedy approach, that is, starting from a single node and iteratively adding branches to the tree until some stopping conditions are met. At a general step, we want to split an existing leaf node e in the m-th tree. Let $V_e = \{i | J(\boldsymbol{x}_i) = e\}$ denote the set of examples that pass through the leaf e. Suppose we fix a split, $t = [feature\ id, threshold]$, consisting of the variable to split and at what threshold it has to be split. This partitions V_e into two disjoint sets: a set V_r associated with the right node and a set V_l associated with the left node. Then we can compute the prediction vectors (h_r and h_l) associated with the right and left nodes based on the loss function restricted to the corresponding sets of examples:

$$\begin{aligned} h_r &= \operatorname*{argmin}_{h_r} \frac{1}{N} \sum_{i \in V_r} \|\boldsymbol{g}_i - h_r\|_2^2 + \lambda \|h_r\|_2^2 \\ h_l &= \operatorname*{argmin}_{h_l} \frac{1}{N} \sum_{i \in V_l} \|\boldsymbol{g}_i - h_l\|_2^2 + \lambda \|h_l\|_2^2. \end{aligned} \tag{12}$$

Since the objectives follow a simple quadratic form, these problems can be solved in closed form as

$$h_r = \frac{1}{\lambda N + |V_r|} \sum_{i \in V_r} \boldsymbol{g}_i, \;\; h_l = \frac{1}{\lambda N + |V_l|} \sum_{i \in V_l} \boldsymbol{g}_i \tag{13}$$

Now we can use h_r and h_l to form prediction: the prediction for example i is $h_{e,i} = h_r$ if $i \in V_r$ and is h_l if $i \in V_l$. This leads to the objective, $obj(t)$ for the split t:

$$obj(t) = \frac{1}{N} \sum_{i \in V_e} \|\boldsymbol{g}_i - h_{e,i}\|^2 + \lambda(\|h_r\|^2 + \|h_l\|^2) \tag{14}$$

The best split is chosen to optimize $obj(t)$:

$$t^* = \min_t obj(t) \tag{15}$$

This completes a general step of the vanilla extension of GBDT for high dimensional sparse output.

Why vanilla GBDT fails on high dimensional sparse output? The vanilla GBDT extension described above faces several difficulties when it is applied on high dimensional sparse output:

1. The first issue is the size of gradient $\boldsymbol{g}_i$ in (11). Each $\boldsymbol{g}_i$ is an L-dimensional vector. Although in the first step $\boldsymbol{g}_i$ is sparse, after one step, h_l (h_r) in (12) will be the average of $|V_r|(|V_l|)$ sparse vectors, which will be dense. A dense prediction F_m will then lead to dense gradients in all the trees after the first step, and this NL space and time complexity is prohibitive in large scale applications where N and L can be both several millions.
2. The second issue is the model size. The prediction vector in each leaf of each tree is a dense vector of length L. This will result in a total model size of $O(TML)$, where T is the number of trees and M is the average number of leaves in each tree. Given that L is large in extreme multi-label learning, the model size will also become very large.
3. The third issue is also related to the dense prediction vector in the tree leaves, and concerns the prediction time. The prediction time for a test point is $O(T\bar{l} + TL)$,[2] where $\bar{l}$ is the average depth of the trees. Thus, when L is large, the prediction is very expensive.
4. The fourth issue relates to the sparsity and large dimension of the input vector $\boldsymbol{x}$. For many real-world problems, the input $\boldsymbol{x}$ is sparse. Induction on such data leads to very unbalanced decision trees with a large number of leaves; this in turn increases the model size and prediction time. It is worth noting that decision trees are generally found to be unsuitable for data with such sparsity.

4.1. Our proposed algorithm: GBDT-Sparse

We now propose a sparsified approach for resolving the above mentioned issues, which leads to the first effective GBDT algorithm for high dimensional sparse output. These modifications lead to models with high accuracy, small model size and fast prediction time.

We first discuss the case when the input features are dense. To handle the first three issues (dense residual vectors, model size, and prediction time), we use the fact that the labels $\boldsymbol{y}_i$ are high dimensional but very sparse. For the loss function satisfies our assumptions (Assumption (7) and (8)), and if both $\boldsymbol{y}_i$ and $\boldsymbol{z}_i$ are sparse, then the gradient vector $\boldsymbol{g}_i$ in (11) will also be a sparse vector, and the sparsity is at most $\|\boldsymbol{y}_i\|_0 + \|\boldsymbol{z}_i\|_0$.

Thus, we enforce a sparsity constraint on the prediction vector in each leaf of each tree and maintain non-zero prediction values only for a small number ($k \ll L$) of labels. Typically, after each tree induction, each leaf contains a coherent set of examples related to a small set of labels and thus the above sparsity constraint makes a lot of sense. Additionally, the constraint offers a nice form of regularization. Note that by definition of $\boldsymbol{g}_i$, it can have at most

[2] The first term is the cost of tree traversal while second is the cost of getting predictions from the leaf nodes.

$Tk + \|\boldsymbol{y}_i\|_0$ non-zeros after T iterations (the label vector $\boldsymbol{y}_i$ is also sparse). This strategy makes the computation very efficient and also reduces memory footprint substantially.

To enforce the sparsity, we add L_0 constraint into the objective function (11), and we have

$$\min_{f_m, w_j^m} \sum_{i=1}^{N} \|\boldsymbol{g}_i - f_m(\boldsymbol{x}_i)\|_2^2 + \lambda \sum_{j=1}^{M_m} \|w_j^m\|_2^2$$
$$\text{s.t. } \|w_j^m\|_0 \leq k, \ \ \forall j. \tag{16}$$

For each cut t, the objective of the left partition becomes:

$$\min_{\|h_l\|_0 = k} \left\{ \sum_{i \in V_l} \|g_i - h_l\|_2^2 + \lambda \|h_l\|_2^2 \right\} := f_l(h_l), \tag{17}$$

where, like before, V_l denotes the set of examples that fall in leaf l. Interestingly, (17) has a closed form solution, and there is no additional time cost by enforcing the sparse constraints. Let $p_q^l = \sum_{i \in V_l} (g_i)_q$ be sorted by the absolute values with the order to be π, such that

$$|p_{\pi(1)}^l| \geq |p_{\pi(2)}^l| \geq \ldots \geq |p_{\pi(|V_l|)}^l|, \tag{18}$$

then the optimal solution of (17) is

$$(h_l)_q^* = \begin{cases} p_q^l / (|V_l| + \lambda) & \text{if } \pi(q) \leq k \\ 0 & \text{otherwise ,} \end{cases} \tag{19}$$

and the objective function is

$$f_l(h_l^*) = f_l(0) - \sum_{q: \pi(q) \leq k} \frac{(p_q^l)^2}{|V_l| + \lambda}. \tag{20}$$

Similarly we can get the same h_r^* and $f_r(h_r^*)$ for the right child, and compute the objective function gain.

Using this closed form solution of the objective function, we want to find the best split $t = [feature\ id, threshold]$ for the current node by minimizing the objective function $f_l(h_l^*) + f_r(h_r^*)$. For simplicity, we assume all the data are in the current node (e.g. the root) in order to simplify the notation, while the same algorithm can be applied to a node with partial samples. Also, we assume a sorted list $\sigma_j(\cdot)$ according to each feature j's value is given, where

$$(\boldsymbol{x}_{\sigma_j(1)})_j \leq (\boldsymbol{x}_{\sigma_j(2)})_j \leq \cdots \leq (\boldsymbol{x}_{\sigma_j(N)})_j.$$

This can be typically done as a pre-processing step before building GBDT because the ordering will not be changed. We then test the decrease of objective function for each threashold according to this order, and select the best one. See Algorithm 1 for detail.

For each feature, although selecting the best threshold from all potential values can optimize objective function, we found this also leads to over-fitting. Therefore, in our implementation we consider the "inexact" version where we only test the threshold for every $\bar{S}$ values in the sorted list: $\{(\boldsymbol{x}_{\sigma_j(i)})_j\}_{i=1,1+\bar{S},1+2\bar{S},\ldots,n}$.

Algorithm 1: GBDT-SPARSE tree node splitting algorithm

```
Input: $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$, sorted list according to each feature
       $\{\sigma_j\}_{j=1}^D$, $\lambda$ (the regularization parameter), $k$
       (sparsity constraint)
Output: Best split $t = [feature\ id, threshold]$
Initial: $f^{\text{best}} = 0$ ;
for $j = 1, \cdots, D$ do
    $(p^l)_s = 0, \ \forall s = 1, \cdots, L$ ;
    $(p^r)_s = \sum_i (g_i)_s, \ \forall s = 1, \cdots, L$ ;
    for $i = 1, \ldots, N$ do
        for $s$ with $(\boldsymbol{g}_{\sigma_j(i)})_s \neq 0$ do
            $(p^l)_s \leftarrow (p^l)_s + (\boldsymbol{g}_{\sigma_j(i)})_s$ ;
            $(p^r)_s \leftarrow (p^r)_s - (\boldsymbol{g}_{\sigma_j(i)})_s$ ;
        Compute the $f = -\frac{\sum_{s \in Q_l} (p_s^l)^2}{i + \lambda} - \frac{\sum_{s \in Q_r} (p_s^r)^2}{N - i + \lambda}$,
        where $Q_l$ and $Q_r$ are the index set of top-$k$ $|p_s^l|$
        and $|p_s^r|$ values respectively;
        If $f < f^{best}$, set $f^{\text{best}} = f$, $t^{\text{best}} = [j, (\boldsymbol{x}_{\sigma_j(i)})_j]$ ;
```

Algorithm 1 can be implemented in $O(D\|G\|_0 \log(k))$ time, where $\|G\|_0 = \sum_{i=1}^N \|\boldsymbol{g}_i\|_0$ is the number of nonzero elements in the current gradient. The main trick is to use two priority queues to maintain two lists of k features with top-k p_s values (correspond to sum of gradient) for left tree and right tree. When scanning through one sample in the inner step, only one term of p_s will change, which has $O(\log k)$ complexity using a priority queue. However, in practice we set $\bar{S}$ to be very large (5% of samples), so a sorting algorithm for finding the top-k list is fast enough, since it only needs to be executed 20 times.

4.2. GBDT-SPARSE: Dealing with Sparse Features

Decision trees usually have difficulty handling sparse features. When feature vectors are sparse, e.g., only 100 out of 10,000 training samples have nonzero values on a feature, the tree will be always imbalanced and extremely deep.

To handle sparse input features, we consider several projection methods that transform sparse features to dense ones. The most simple yet useful one is to use random projection, that is, projecting the data point to $\bar{\boldsymbol{x}}_i = \bar{G}\boldsymbol{x}_i$ using a fixed random Gaussian matrix $\bar{G} \in \mathbb{R}^{d \times D}$ as projection matrix. To reduce reconstruction error, another approach is to use Principal Component Analysis (PCA) (Halko et al., 2011) via SVD (Si et al., 2014).

Both random projection and PCA are un-supervised learn-

Table 1: Comparison between traditional GBDT, our proposed GBDT-SPARSE, and FASTXML in terms of training time, prediction time, model size and accuracy. Prediction time includes feature projection time. All time in seconds.

Metrics	FASTXML	vanilla GBDT (LEML)	GBDT-SPARSE (Random Projection)	GBDT-SPARSE (PCA)	GBDT-SPARSE (LEML)
Dimension reduction time	N/A	100.74	4.97	99.86	100.74
Training Time	1275.9	41078.76	931.57	1025.03	1054.12
Prediction Time	9.1175	52.139	1.0766	1.0796	1.087
Accuracy *P@1*(%)	82.71	84.11	80.79	83.51	**84.36**
Accuracy *P@3*(%)	67.87	68.94	50.68	67.04	**69.49**
Model size	813MB	809.39M	79.01MB	79.23MB	79.26MB

Table 2: Data set statistics for multi-label learning problems.

Dataset	# Training samples	# Testing samples	# Features	# Labels	Avg. points per label	Avg. labels per point
Mediamill	30,993	12,914	120	101	1338.8	4.36
Delicious	12,920	3,185	500	983	250.06	19.02
NUS-WIDE	161,789	107,859	1,134	1,000	935.22	5.78
Wiki10-31K	14,146	6,616	101,938	30,938	8.52	18.64
Delicious-200K	196,606	100,095	782,585	205,312	72.34	75.54

ing approaches—in the sense that they do not use any label information; however, in our problem setting there is rich information in the high dimensional output space Y. Therefore, we can use a supervised algorithm LEML (Yu et al., 2014) to construct dense features, which solves the following optimization problem:

$$\min_{W \in R^{D \times \bar{d}}, H \in R^{L \times \bar{d}}} \|Y - XWH^T\|_F^2 + \gamma(\|W\|_F^2 + \|H\|_F^2)$$

where γ is a regularization term to control the over-fitting and $\bar{d}$ is the projected dimension. This has been discussed in (Yu et al., 2014) for solving the multi-label classification problems, and the resulting algorithm uses an alternating minimization algorithm to compute the solutions W and H. After we get W from LEML, we use the new features $\bar{X}$ as $\bar{X} = XW$ to construct the decision trees. Using this projection has two benefits:(1) the projection incorporates the label information; and, (2) the new data after projection, $\bar{X}$ is dense, and thus results in shallow and balanced trees.

We compare GBDT-SPARSE with different projection methods as well as vanilla GBDT for extreme multilabel learning problem in Table 1. We used the Wiki10-31K dataset with training parameters the same as the ones in section 5, except we terminate all methods (except vanilla GBDT) in about 1000 seconds. Three dimension reduction techniques, LEML, PCA and random projections are used to reduce the feature size to 100. We also include FASTXML as a comparison for training time. From Table 1 we can see that using LEML is more accurate than using PCA and random projections, but takes longer time to train the model. Different from vanilla GDBT, GBDT-SPARSE enforces the sparsity in the leaf nodes, which brings significant speedup (about 40x) for training. This table shows the benefits of using feature projection and enforcing sparsity in leaf nodes when applying GDBT on problems with high-dimensional sparse output.

4.3. GBDT-SPARSE: Fast Prediction

When performing prediction, the data points will go through each tree and then the prediction is $f(\boldsymbol{x}_i) = \sum_{m=1}^{T} h_m(\boldsymbol{x}_i)$. In vanilla GBDT, this requires $O(LT)$ time since we have to sum over the prediction for T trees, each one is an L-dimensional dense vector. Note that the tree traversal time can be omitted because each node only takes 1 comparison to look at whether a feature is larger or smaller than the threshold.

In GBDT-SPARSE, when making prediction for a new data point, we can utilize the sparsity structure of each prediction vector to achieve fast prediction time: adding up T of the k-sparse vectors together. The naive approach is to create an array of size Tk, copy all the index-value pairs to the array, and sort them by index. This has $O(Tk\log(Tk))$ time complexity. A more efficient approach is to use a min-heap data structure to merge these k lists which can reduce time complexity: first, sort each list according to the index orders, and then create a min heap of size k and insert the first element in all lists to the heap. Then repeatedly conduct the following process: (1) get the minimum element from heap, store to the output array, and (2) update the heap root value by the next index from the list that the element is fetched. The overall algorithm will take $O(Tk\log k)$ time.

In some real world applications, only top-B labels are needed with very small B (typically 1,3,5). In those cases, we can further reduce the prediction time to $O(Tk\log B)$ (see details in appendix B). Since we test on small k for all our experiments, we do not use this technique in practice.

4.4. Summary of GBDT-SPARSE

In summary, the training time of GBDT-SPARSE is $O(D\|G\|_0\log(k))$ for each node, where $\|G\|_0$ is total number of nonzeros of the samples belonging to the node. So each level of the tree requires $O(D\|X\|_0\log(k))$ time. If we build T trees and each with h levels, the total training time is $O(DTh\|X\|_0\log(k))$.

As discussed in the previous section, the prediction time is $O(Tk\log k)$ for prediction. k (sparsity constraint) is usually set to be less than 50; T (number of tress) is usually less than 100. Therefore GBDT-SPARSE has a sub-linear (constant) prediction time.

Now we discuss model size. Each intermediate node only stores the $[feature\ id, threshold]$ pair, which is one integer and one floating point. Each leaf node only stores the k index-value pairs. Therefore, the model size is $O(kT2^h)$. As long as tree depth h is not too large (usually less than 12), the model size is very small.

5. Experiments

We compare GBDT-SPARSE against other key methods for extreme multi-label classification problems and demonstrate its value with respect to model size, prediction time and performance.

Data: We conducted experiments on 5 standard and publicly available multi-label learning datasets.[3] Table 2 shows the associated details. Note the diversity in the number of training samples, label size and feature dimensionality. Delicious-200K has more than $200,000$ labels.

Baselines: We compare our method to four state-of-the-art extreme multi-label learning baselines.

1. LEML (Yu et al., 2014) is an embedding technique based on low-rank empirical risk minimization.
2. FASTXML (Prabhu & Varma, 2014) is a random forest based approach where each tree is constructed by jointly optimizing both $nDCG$ ranking loss and tree structure. A sparse linear separator is used as the splitting criteria at each node.
3. SLEEC (Bhatia et al., 2015) learns an ensemble of local distance preserving embeddings. Pairwise distances are preserved between only the nearest label vectors.
4. PD-SPARSE (Yen et al., 2016) proposes to solve L_1 regularized multi-class loss using Frank-Wolfe based algorithm. However, it needs to store weight vectors in size $O(DL)$, which is hard to scale to large datasets.

For the baselines, we use their highly optimized C++ implementation published along with the original papers. We also compare with DisMEC (Babbar & Schölkopf, 2017) in the Appendix.

Parameter Setting: For FASTXML and LEML, we use the default parameter settings in the code. SLEEC's code also has optimal parameter settings for all the datasets except NUS-WIDE. It has 7 parameters and their settings vary widely for different datasets. For PD-SPARSE, we use a grid search to find the best regularization parameter λ and cost C. For our method, we kept most of the parameters fixed for all the datasets: $h_{max} = 10, n_{leaf} = 100$, and, $\lambda = 5$, where h_{max} and n_{leaf} are the maximum level of the tree and the minimal number of data points in each leaf. Leaf node sparsity k was set to 100 for Delicious-200K and 20 for all others. This parameter can be very intuitively set as an increasing function of label set size. We hand tuned the projection dimensionality d and set it to 100 for Delicious and Wiki10-31K, and 50 for others.

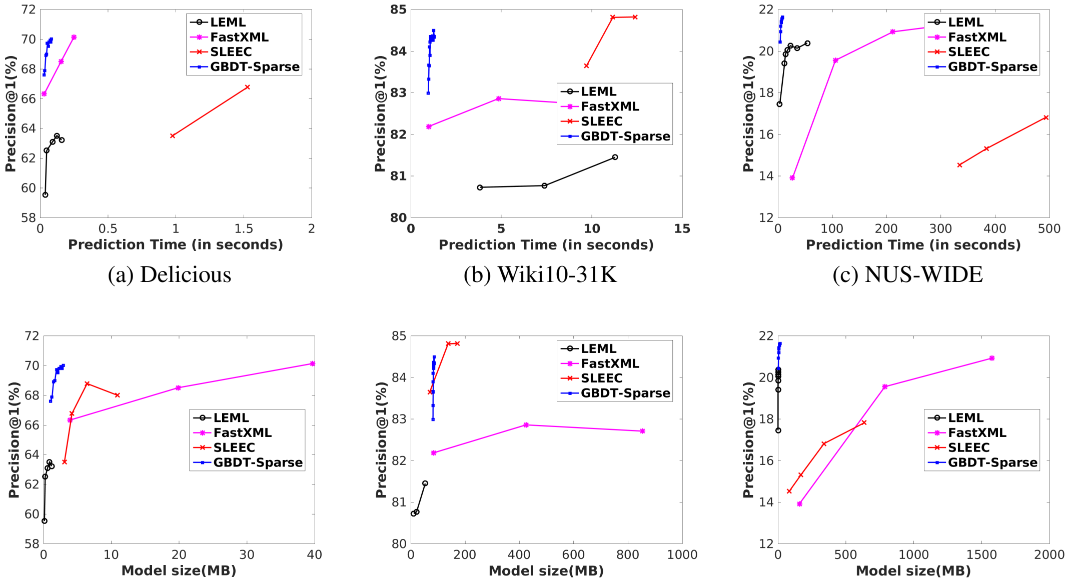
Results: Table 3 shows the performance of different methods along the dimensions of prediction time, model size and prediction accuracy (*Precision@1* (*P@1*) and *Precision@3*(*P@3*)). Note that the strength of our method is to achieve similar accuracy with smaller memory footprint and prediction time. Also note that LEML has inferior performance to all other methods. However, its prediction times are similar to our method on many datasets. FASTXML, SLEEC and GBDT-SPARSE achieve similar accuracy on almost all the datasets. For PD-SPARSE, we observe that its accuracy can fluctuate badly across iterations in dataset Delicious and Delicious-200K despite of trying different set of parameters, even though the reported dual objective is monotonically decreasing. Also, due to its linear nature, its model size is small, but accuracy is also limited by the capacity of the learner. In terms of accuracy *P@1* and *P@3*, there is no clear trend of GBDT-SPARSE being better or worse than others. However, GBDT-SPARSE gives an order of magnitude speedup in prediction times for almost all the datasets. For example, for Delicious-200K, our method is 10.58x and 14.72x faster than FASTXML and SLEEC respectively. Similar gains can be observed for the model size. It is worth noting that we do not fine-tune most hyper parameters for decision tree building process, and the set of parameters can get good accuracy on all of our datasets.

Figure 1(a)-(c) shows the *P@1* as a function of time for three datasets. For GBDT-SPARSE and FASTXML, we vary the number of trees to get different prediction times. For LEML and SLEEC, experiments are ran for different embedding sizes to generate the curve. The more the curve is towards top left, better is the performance. For GBDT-SPARSE, the curves sharply rise in performance; though not shown, they become stable at the highest performance values shown. Though GBDT-SPARSE does not always beat all methods on performance, we can observe that for any fixed prediction time our approach impressively outperforms all others. Figure 1(d)-(f) shows the corresponding curves as a function of model size. Again similar observations can be made, except for Wiki10-31K where SLEEC has a similar model size. In summary, we can see from Figure 1 that to achieve similar accuracy, GBDT-SPARSE takes much less prediction time and the model size

[3] NUS-WIDE is available at http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm. All other datasets are available at http://manikvarma.org/downloads/XC/XMLRepository.html.

Table 3: Comparison on five large-scale multi-label datasets. Time refers to prediction times in seconds. Size is the model size in megabytes. All experiments are conducted on a machine with an Intel Xeon X5440 2.83GHz CPU and 32GB RAM. For PD-Sparse we use a similar machine with 192GB memory due to its large memory footprint. Please zoom.

	LEML				FastXML				SLEEC				PD-Sparse				GBDT-Sparse (proposed)			
	Time	Size	*P@1*	*P@3*	Time	Size	*P@1*	*P@3*	Time	Size	*P@1*	*P@3*	Time	Size	*P@1*	*P@3*	Time	Size	*P@1*	*P@3*
Mediamill	0.28	0.17	82.83	66.29	3.44	7.25	83.13	66.39	65.16	92.04	85.02	68.40	0.034	0.005	82.99	62.32	0.60	3.54	84.23	67.85
Delicious	0.16	1.18	63.23	58.51	0.24	39.66	70.14	64.51	1.52	4.19	66.78	60.32	0.036	0.25	52.02	45.91	0.13	4.76	69.29	63.62
NUS-WIDE	22.77	1.70	20.26	15.58	211.88	1.57G	20.93	16.24	384.11	212.2	15.32	12.36	program crashed				8.86	14.46	21.65	16.73
Wiki10-31K	11.67	13.28	80.26	65.73	10.21	853.5	82.71	67.87	30.19	570.2	85.99	73.65	1.04	0.60	82.03	67.44	1.30	85.81	84.34	70.82
Delicious-200K	12.85	790.4	40.47	37.69	146.55	11.3G	42.99	38.50	203.86	7.98G	45.63	40.77	67.51	3.80	35.47	32.07	13.85	338.0	42.11	39.06



(a) Delicious

(b) Wiki10-31K

(c) NUS-WIDE

(d) Delicious

(e) Wiki10-31K

(f) NUS-WIDE

Figure 1: **Top:** *P@1* as a function of time. **Bottom:** *P@1* as a function of model size.

is much smaller than other methods.

Multicore Implementation: Unlike random-forest based methods, paralllelizing GBDT is not straightforward. In our problem, because L is large, existing frameworks like XGBoost (Chen & Guestrin, 2016) do not scale well as it needs $O(L)$ storage per leaf, and histogram based methods need $O(L)$ space per bin to accumulate gradients. We implement our algorithm by finding best splits for different features on a single leaf in parallel. Although this requires extra time to sort feature values on each leaf, we find that for datasets with a big L the sorting time is insignificant. We run our algorithm with Delicious-200K on a 28-core dual socket E5-2683v3 machine to build a GBDT with 5 trees, and record the average time for building one tree in Table 4. The good scaling shows that our algorithm is capable for handling big data. Also, the huge speedup from parallelization is a big advantage to use our algorithm in practice, comparing to algorithms that cannot be easily parallelized, like PD-Sparse.

6. Conclusion

We apply GBDT to solve problems with high dimensional sparse output. Applying GBDT to this setting has several challenges: large dense gradient/residual matrix, imbalanced trees due to data sparsity, and large memory footprint for leaf nodes. We made non-trivial modifications to GBDT (use embeddings to make features dense, introduce label vector sparsity at leaf nodes) to make it suitable for handling high dimensional output. These improvements can significantly reduce the prediction time and model size. As an application, we use our proposed method to solve extreme multi-label learning problem. Compared to the state-of-the-art baselines, our method shows an order of magnitude speed-up (reduction) in prediction time (model size) on datasets with label set size $1000 - 200000$.

Table 4: Average time (in seconds) for building one tree using GBDT-Sparse on dataset Delicious-200K.

Threads	1	4	8	10	14	28 (2 sockets)
Time (s)	1092.60	353.07	191.22	153.53	117.49	85.36
Speedup	baseline	3.09x	5.71x	7.12x	9.30x	12.80x

Acknowledgments This research was supported by NSF grants CCF-1320746, IIS-1546452 and CCF-1564000. Cho-Jui Hsieh also acknowledges support from XSEDE.

References

Abuzaid, Firas, Bradley, Joseph K., Liang, Feynman T., Feng, Andrew, Yang, Lee, Zaharia, Matei, and Talwalkar, Ameet S. Yggdrasil: An optimized system for training deep decision trees at scale. In *NIPS*, 2016.

Agrawal, Rahul, Gupta, Archit, Prabhu, Yashoteja, and Varma, Manik. Multi-label learning with millions of labels: Recommending advertiser bid phrases for web pages. In *WWW*, 2013.

Babbar, Rohit and Schölkopf, Bernhard. Dismec: Distributed sparse machines for extreme multi-label classification. In *WSDM*, pp. 721–729, 2017.

Bhatia, Kush, Jain, Himanshu, Kar, Purushottam, Varma, Manik, and Jain, Prateek. Sparse local embeddings for extreme multi-label classification. In *NIPS*, 2015.

Breiman, Leo. Bagging predictors. *Machine Learning*, 24 (2):123–140, 1996.

Chen, Tianqi and Guestrin, Carlos. Xgboost: A scalable tree boosting system. In *KDD*, 2016.

Chen, Tianqi, Singh, Sameer, Taskar, Ben, and Guestrin, Carlos. Efficient second-order gradient boosting for conditional random fields. In *AISTATS*, 2015.

Choromanska, Anna and Langford, John. Logarithmic time online multiclass prediction. In *NIPS*, pp. 55–63, 2015.

Friedman, Jerome H. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29 (5):1189–1232, 2001.

Friedman, Jerome H. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38(4):367–378, 2002.

Halko, Nathan, Martinsson, Per-Gunnar, and Tropp, Joel A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

Jasinska, Kalina, Dembczynski, Krzysztof, Busa-Fekete, Róbert, Pfannschmidt, Karlson, Klerx, Timo, and Hüllermeier, Eyke. Extreme f-measure maximization using sparse probability estimates. In *ICML*, pp. 1435–1444, 2016.

Liaw, Andy and Wiener, Matthew. Classification and regression by random forest. *R News*, 2(3):18–22, 2002.

Meng, Qi, Ke, Guolin, Wang, Taifeng, Chen, Wei, Ye, Qiwei, Ma, Zhi-Ming, and Liu, Tie-Yan. A communication-efficient parallel algorithm for decision tree. In *NIPS*, 2016.

Panda, Biswanath, Herbach, Joshua S., Basu, Sugato, and Bayardo, Roberto J. PLANET: massively parallel learning of tree ensembles with mapreduce. *Proceedings of VLDB*, 2(2):1426–1437, 2009.

Prabhu, Yashoteja and Varma, Manik. Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *KDD*, 2014.

Schapire, Robert E. A brief introduction to boosting. In *IJCAI*, 1999.

Si, Si, Shin, Donghyuk, Dhillon, Inderjit S., and Parlett, Beresford N. Multi-scale spectral decomposition of massive graphs. In *NIPS*, pp. 2798–2806, 2014.

Si, Si, Chiang, Kai-Yang, Hsieh, Cho-Jui, Rao, Nikhil, and Dhillon, Inderjit S. Goal-directed inductive matrix completion. In *ACM SIGKDD*, 2016.

Yen, Ian En-Hsu, Huang, Xiangru, Ravikumar, Pradeep, Zhong, Kai, and Dhillon, Inderjit S. Pd-sparse : A primal and dual sparse approach to extreme multiclass and multilabel classification. In *ICML*, pp. 3069–3077, 2016.

Yu, Hsiang-Fu, Jain, Prateek, Kar, Purushottam, and Dhillon, Inderjit S. Large-scale multi-label learning with missing labels. In *ICML*, 2014.

A. Additional Experiments on DisMEC

DisMEC (Babbar & Schölkopf, 2017) is a distributed extreme multi-label learning framework based on one-versus-rest linear classifiers with explicit model size controlled by pruning small weights. Unlike other methods we have compared in Section 5, DisMEC mainly focuses on parallelizing an extremely large number of one-versus-all classifiers in large-scale distributed settings with double layers of parallelization (multi-core and multi-machine).

DisMEC's primarily advantage is that it does not make any low-rank or sparsity assumption for the data and thus prediction performance is better, and its model size is reasonably small due to weight pruning. However, it has the same time complexity as the naive one-versus-all method and requires much more computation than all other methods we have compared in Section 5. For example, on dataset Wiki10-31K, our algorithm needs less than 20 minutes on 1 core (refer to Table 1), while DisMEC requires 10 minutes on 300 cores as reported in (Babbar & Schölkopf, 2017) and we record about 450 minutes training time using 4 cores.

Table 5: Experiments on DisMEC. Time refers to prediction times in seconds. Size is the modelsize in megabytes.

	DisMEC				GBDT-Sparse (proposed)			
	Time	Size	*P@1*	*P@3*	Time	Size	*P@1*	*P@3*
Mediamill	0.59	0.087	87.77	70.25	0.60	3.54	84.23	67.85
Delicious	0.24	3.4	66.80	61.79	0.13	4.76	69.29	63.62
Wiki10-31K	771.8	880	84.12	74.71	1.30	85.81	84.34	70.82

We use the DisMEC implementation from its authors [4]. We found that in their experiment implementation, a TF-IDF (term frequency inverse document frequency) feature conversion is used. Using TF-IDF features can improve prediction accuracy for text based datasets, and we found that it is necessary to use TF-IDF to get a good accuracy for Wiki10-31K. Therefore, we pre-process Wiki10-31K using TF-IDF for DisMEC in this section (we do not use TF-IDF pre-processing in all other experiments). Due to our limited computing resources, we only include Mediamill, Wiki10-31K and Delicious in this experiment. The result is shown in Table 5. DisMEC achieves similar performance with our method, but note that DisMEC requires much more computation resources than our method. Larger Datasets like Delicious-200K is practically unfeasible on a single machine (with only a few cores) using DisMEC.

B. Prediction time for GBDT-Sparse

In many real world applications, only top-B labels are needed with very small B (typically 1,3,5). In those cases, we can further reduce the prediction time to $O(Tk \log B)$. To do that, we need a hash (with $O(Tk)$ size) and a min-heap Q. The algorithm scans through all the elements in the prediction vectors (each contains k $(idx, value)$ pairs) for each tree $h_1(\boldsymbol{x}_i), \cdots, h_m(\boldsymbol{x}_i)$. For each $(idx, value)$ pair, we first use hash to add the value to the corresponding index p_{idx}. If the index is already in the heap, then update the corresponding value. If $Q.size()$ is smaller than B, then add the (idx, p_{idx}) pair to Q. Otherwise compare p_{idx} with $Q.min()$, and replace the minimum number in Q by p_{idx} if p_{idx} is larger than $Q.min()$. Since the size of Q is always $\leq B$, the complexity is $O(Tk \log B)$.

[4] https://sites.google.com/site/rohitbabbar/code/dismec