Training Efficient Tree-Based Models for Document Ranking

Nima Asadi[1,2] and Jimmy Lin[1,2,3]

[1] Dept. of Computer Science
[2] Institute for Advanced Computer Studies
[3] The iSchool University of Maryland, College Park
nima@cs.umd.edu, jimmylin@umd.edu

Abstract. Gradient-boosted regression trees (GBRTs) have proven to be an effective solution to the learning-to-rank problem. This work proposes and evaluates techniques for training GBRTs that have efficient *runtime* characteristics. Our approach is based on the simple idea that compact, shallow, and balanced trees yield faster predictions: thus, it makes sense to incorporate some notion of execution cost during training to "encourage" trees with these topological characteristics. We propose two strategies for accomplishing this: the first, by directly modifying the node splitting criterion during tree induction, and the second, by stage-wise tree pruning. Experiments on a standard learning-to-rank dataset show that the pruning approach is superior; one balanced setting yields an approximately 40% decrease in prediction latency with minimal reduction in output quality as measured by NDCG.

1 Introduction

There is general consensus in the information retrieval community that the challenge of document ranking is best addressed using machine learning techniques, known as the "learning to rank" approach. In particular, gradient-boosted regression trees (GBRTs) have proven to be highly effective, as documented in both the academic literature [1, 2] and in production commercial search engines such as Bing [3]. In this work, we propose two novel extensions to the GBRT training regime that not only yields effective models (as measured by NDCG), but also those that are efficient *at runtime* (i.e., fast in making predictions on new test instances). This is an important problem because model execution forms the inner loop of search engines, and in the web context, may be invoked billions of times a day in a commercial system. Improvements in efficiency have a direct impact on the bottom line, in terms of fewer servers required to handle the query load or more computational resources to explore richer features. Thus, increasing the runtime efficiency of tree-based models is a worthwhile problem to tackle.

Our approach is based on the simple idea that models with compact, shallow, and balanced trees should generally be faster in making predictions—thus, we should "encourage" the trainer to build ensembles with these topological characteristics. This paper proposes two different strategies to accomplish this goal:

P. Serdyukov et al. (Eds.): ECIR 2013, LNCS 7814, pp. 146–157, 2013.

In the first, we directly modify the node splitting criterion for tree induction to incorporate efficiency cost. In the second, we run tree induction as normal in each boosting stage, but then prune the trees until they are better balanced before continuing to build the ensemble. Experiments on a standard learning-to-rank dataset show that the pruning approach is superior: one particular parameter setting yields an approximately 40% decrease in prediction latency with minimal reduction in output quality as measured by NDCG.

This work contributes to an emerging thread of research in learning to rank focused on better balancing effectiveness and efficiency. The primary contribution is a novel approach to training GBRTs that yields good *and* fast models. To our knowledge, we are the first to explore the topology of tree-based ensembles for learning to rank from the perspective of runtime performance.

2 Background and Related Work

Until recently, most research on learning to rank has focused exclusively on building effective models (e.g., that yield high NDCG), without regard to the runtime performance of those models (query execution speed); see [1] for a recent survey. However, there is an emerging thread of work, dubbed "learning to *efficiently* rank", whose goal is to train models that both deliver high-quality results and are fast in ranking documents [4–6]. Our work is in this spirit. However, there are important differences: Wang et al. [5] explore a cascade of linear models, whereas we focus on tree-based models. Xu et al. [6] take advantage of tree-based models, but their work aims to minimize feature extraction costs. The authors do not factor in the structure of the trees, whereas we specifically explore the effect of tree topology on runtime speed.

There has been work on efficiently *training* tree-based models on large datasets using distributed approaches [7, 8]. However, this work is orthogonal, since it is not concerned with the *runtime* performance of the learned models. Nevertheless, our proposed techniques can benefit from distributed training strategies to scale out, but scalable learning is beyond the scope of this paper.

In the remainder of this section, we first provide an overview of LambdaMART, which is the learning-to-rank model that underlies this work, and then discuss previous attempts at optimizing tree ensembles.

2.1 LambdaMART

The effectiveness of tree-based ensembles for learning to rank has been widely demonstrated: an example is the family of gradient-boosted regression trees (GBRTs) [9, 3, 10, 2]. In this context, our work uses LambdaMART [3], which is the combination of LambdaRank [11] and MART [12]—a class of boosting algorithms that performs gradient descent using regression trees.

LambdaMART learns a ranking model by sequentially adding new trees to an ensemble that best account for the remaining regression error (i.e., the residuals) of the training samples. More specifically, LambdaMART learns a linear predictor $H_{\boldsymbol{\beta}}(x) = \boldsymbol{\beta}^{\intercal}\mathbf{h}(x)$ that minimizes a given loss function $\ell(H_{\boldsymbol{\beta}})$, where the base

learners are limited-depth regression trees [13]: $\mathbf{h}(x) = [h_1(x), ..., h_T(x)]$, where $h_t \in \mathcal{H}$, and $\mathcal{H}$ is the set of all possible regression trees.

Assuming we have constructed $t-1$ regression trees in the ensemble, LambdaMART adds the t_{th} tree that greedily minimizes the loss function, given the current pseudo-responses. CART [13] is used to generate a regression tree with J terminal nodes, which works by recursively splitting the training data. At each step, CART computes the best split (a feature and a threshold) for all terminal nodes, and then applies the split that yields the most gain, thereby growing the tree one node at a time. Consider the following cost function:

$$C(N, \langle f, \theta \rangle_N) = \sum_{x_i \in L} (y_i - \bar{y}_L)^2 + \sum_{x_i \in R} (y_i - \bar{y}_R)^2, \tag{1}$$

where N denotes a node in the tree; $\langle f, \theta \rangle_N$ is a split, consisting of a feature and a threshold; L and R are the left and right sets containing the instances that fall into the left and right of node N after the split is applied, respectively; x_i and y_i denote a training instance and its associated pseudo-response; and finally, $\bar{y}_L$ and $\bar{y}_R$ are the average y (regression value) of instances that fall either to the left or the right branch of a node, respectively, after the split. Minimizing Equation (1) is equivalent to maximizing the difference in $C(\cdot)$ before and after a split is applied to node N. This difference can be computed as follows:

$$G(N, \langle f, \theta \rangle_N) = \sum_{x_i \in N} (y_i - \bar{y}_N)^2 - C(N, \langle f, \theta \rangle_N), \tag{2}$$

where $x_i \in N$ denotes the set of instances that are present in node N.

The final LambdaMART model has low bias but is prone to overfitting training data (i.e., the model has high variance). In order to reduce the variance of an ensemble model, bagging [14] and randomization can be utilized during training. Friedman [12] introduces the following randomization techniques:

- A weak learner is fit on a sub-sample of the training set drawn at random without replacement.
- Similar to Random Forests [15], to determine the best tree split, the algorithm picks the best feature from a random subset of all features.

Ganjisaffar et al. [2] take this one step further and construct multiple ensembles, each built using a random bootstrap of the training data (i.e., *bagging* multiple boosted ensembles). In this work, we do not explore bagging, primarily because it is embarrassingly-parallel from the runtime execution perspective and thus not particularly interesting.

2.2 Optimizing Tree Ensembles

Tree ensembles can comprise a large number of trees, which makes prediction slow. In this section, we discuss a number of existing techniques to address this issue. First and most obvious is to use a larger shrinkage parameter η, which

causes the algorithm to converge faster and hence yields ensembles containing fewer trees. However, this often comes at the cost of lower effectiveness in a difficult-to-control manner. In other words, the shrinkage parameter is too coarse-grained a "knob" to control the effectiveness/efficiency tradeoff.

Ganjisaffar [16] proposes several techniques to reduce the evaluation cost by discarding trees that contribute the least to document scores. In what is referred to as *Prefix* compression, only the first n trees in an ensemble are preserved, while the rest are removed from the ensemble. The value of n can be determined based on a time budget. This is similar to the early termination of training described by Margineantu and Dietterich [17], where the focus is on minimizing storage (as opposed to speed per se).

A different approach is to re-weight all trees *after* the entire ensemble is built and to create sparsity by assigning a weight of zero to trees whose absence does not significantly change effectiveness. Ganjisaffar [16] achieves this using the Lasso [18] method, in which the regularized least squares of the error between the original scores and the scores obtained through re-weighted trees are minimized. However, this approach does not yield more compact ensembles when the learning rate is small [16]. Similarly, Margineantu and Dietterich [17] select weak learners from an AdaBoost classifier that meet a diversity and classification accuracy criterion. Results suggest that 60–80 percent of the weak learners can be discarded without a significant loss in accuracy. For an overview of ensemble pruning techniques see [19] and the references therein.

At a different level, pruning can be applied to individual nodes rather than entire trees. Ganjisaffar [16] presents a node pruning approach that trims leaves in a tree and collapses them into a single terminal node after an ensemble has been built, if the impact of such trimmings on effectiveness is insignificant. However, despite the costly computations required to perform such pruning, it does not result in more compact ensembles.

There is, of course, much work on the pruning of non-ensemble tree models. These techniques require an estimation of error, such as [20]. However, since boosting is performed on residuals, pruning each tree alters what the learner would have produced for every subsequent tree. In general, the problem of adapting pruning techniques for non-ensemble trees to boosted ensembles has not been thoroughly explored. This is in part because trees in ensembles are much shallower than trees in non-ensemble models, and thus there is less need to prune.

Our work differs from previous work in two important respects: First, previous research on optimizing tree ensembles do not focus on efficiency explicitly—for the most part, pruning is thought of as a regularization technique to reduce overfitting. Second, all the techniques presented above represent *post hoc* approaches—i.e., pruning is performed as a post-processing step *after* the model has been learned. In contrast, we attempt to optimize runtime efficiency *while* constructing the ensemble. Thus, we view the above surveyed literature as complementary but orthogonal—there is nothing to prevent us from applying any of the above techniques *after* training tree ensembles using the methods proposed in this paper. Similarly, we also consider strategies such as early exits [21]

orthogonal to our work, since they focus on runtime characteristics after a model has been learned (and can be applied to our approach as well).

3 Training Efficient GBRTs

In this paper, we assume an implementation of LambdaMART using statically-generated if-else code blocks in C. That is, we directly translate a learned model into a prediction function in C using nested if-else blocks. This code is then compiled and linked to the rest of the code base that provides the evaluation framework. In a production environment, the compiled object files would be linked to the rest of the search engine. The input feature vector is assumed to be a densely-packed floating point array, so the boolean predicate at each if statement involves comparing an array element to a threshold, and then taking the "then" block or the "else" block based on the comparison.

We expect this approach to be fast. The entire model is statically specified; machines instructions are expected to be relatively compact and will fit into the processor's instruction cache, thus exhibiting good reference locality. Furthermore, we leverage decades of compiler optimizations that have been built into `gcc`. To our knowledge, this is a relatively common "trick" adopted in industry for building fast tree-based models.

Our approach to training efficient tree ensembles is based on the simple idea that compact, shallow, and balanced trees should yield faster runtime execution. It is perhaps intuitive why this should be the case, although our results show that the connection between tree topology and performance is more nuanced. By more compact trees we mean trees with fewer nodes: for trees of equal depth, we would expect more compact trees to be faster since there are fewer branch instructions overall (no matter what path is taken). For trees with equal numbers of nodes, we'd expect shallower trees to be faster, since the longest path through the tree is bounded by its depth. Also, for trees with equal numbers of nodes, we'd expect better balanced trees to be faster since that would imply shorter *average* paths through the tree. A side effect of better balanced trees might be lower variance on execution time, since the paths through the trees are more likely to have similar lengths. Compactness, tree depth, and degree of balance are all inter-related, but our results show that to some extent they are orthogonal characteristics: for example, more compact trees aren't necessarily shallower.

Based on the desirability of compact, shallow, and balanced trees, we wish to "encourage" the LambdaMART trainer to construct ensembles consisting of trees having these characteristics. We explore two techniques to accomplish this goal: One approach is to directly modify the splitting criterion during induction of each individual tree to grow leaves in a balanced manner. Alternatively, when generating each tree in the ensemble, we can proceed as normal, and then prune back the tree to a shallower and more balanced topology before proceeding to the next boosting stage. We explore both approaches, detailed below.

3.1 Cost-Sensitive Tree Induction

The cost of evaluating a tree ensemble is bound by $\boldsymbol{c}(H_{\boldsymbol{\beta}}) = \sum_{t=1}^{T} c \cdot d_t$, where c is the cost of evaluating one conditional, and d_t is the depth of the t_{th} tree h_t. We can view c as the marginal cost of evaluating a tree whose depth grows by one. Our goal is to jointly minimize the loss $\ell(H_{\boldsymbol{\beta}})$ as well as the cost $\boldsymbol{c}(H_{\boldsymbol{\beta}})$.

In order to solve this multi-objective optimization problem, we take a greedy approach in which we modify the CART splitting criterion. The intuition is that the algorithm should not only maximize the gain $G(\cdot)$, but should also penalize splits that result in an increase in the tree depth. That is, to select the next node to split, we want to maximize Equation (2) while minimizing $\boldsymbol{c}([h_t]) = c \cdot d_t$; we can safely ignore c since it's a constant.

We approach this problem by a priori articulating the preferences, where we determine the importance of each objective function and prioritize accordingly. We then solve the problem using the lexicographic approach [22, 23]. This method iterates through the functions sorted in increasing order of importance. At step i, it finds a solution for function f_i. A solution X' must fulfill f_i's constraints and must also satisfy $f_j(X') < \alpha f_j(X)\ \forall_{j<i}$, where X is the solution found so far, and α is the relaxation parameter. In this way, among all terminal nodes, we find the node and split (i.e., threshold) that maximizes Equation (2):

$$\underset{N,\langle f,\theta\rangle_N}{\arg\max}\, G(N, \langle f, \theta\rangle_N). \tag{3}$$

We denote the solution to Equation (3) as N^* and its split $\langle f^*, \theta^*\rangle_{N^*}$. Next, among all terminal nodes, we choose the node that minimizes the depth of the tree d_t, by relaxing a constraint on $G(\cdot)$. Our second optimization problem is:

$$\begin{aligned} &\underset{N,\langle f,\theta\rangle_N}{\arg\min}\, D(N, \langle f, \theta\rangle_N), \\ &\text{such that } G(N, \langle f, \theta\rangle_N) > (1-\lambda)\, G(N^*, \langle f^*, \theta^*\rangle_{N^*}), \end{aligned} \tag{4}$$

where $\lambda \in [0, 1]$. Function $D(\cdot)$ is defined as follows:

$$D(N, \langle f, \theta\rangle_N) = \begin{cases} d_t, & d_N + 1 \leq d_t \\ d_t + 1, & \text{otherwise} \end{cases}, \tag{5}$$

where d_t is the current depth of the tree, and d_N is the depth of node N. In the second optimization problem, in order to obtain a single solution, we break ties based on gain. Setting λ to 0 reduces this model to unmodified LambdaMART.

We can interpret this optimization problem as follows: If the split that results in maximum gain does not increase the maximum depth of the tree, then continue with the split. Otherwise, find a node closer to the root which, if split, would result in a gain larger than the discounted maximum gain.

3.2 Pruning While Boosting

In the pruning approach, we construct the t_{th} tree using the original CART algorithm, but before proceeding to add the tree to the ensemble, we prune

the tree with a focus on depth and balance. Our algorithm starts discarding the two deepest terminal nodes in the tree and turning their parent into a new terminal node (and restoring the original regression value at the leaf pre-split) until a stopping criterion is met (discussed below). However, unlike the method proposed in [16], we do not include effectiveness in our criterion and exclusively focus on tree depth and density. The intuition is that, since we are performing the pruning *while* boosting, additional stages compensate for the loss in effectiveness while at the same time reduce the average depth of trees in the ensemble.

In this work, we explore a simple criterion: we continue collapsing terminal nodes until the total number of nodes in the tree is greater than or equal to a fraction of the maximum possible number of nodes in the tree given its depth (i.e., in a perfectly-balanced tree), as follows:

$$|h_t| \geq \alpha \left(2^{d_t+1} - 1\right), \tag{6}$$

where $|h_t|$ is the number of nodes in tree h_t, d_t is the current depth of the tree, and $\alpha \in [0, 1]$ is a tuning parameter. Intuitively, α controls to what extent we want our trees to "look like" perfectly-balanced binary trees. Setting α to 0 reduces this model to the original LambdaMART algorithm. Increasing α translates into more aggressive pruning, and with $\alpha = 1.0$, we prune back the tree until we obtain a perfectly-balanced binary tree. Obviously, this pruning method does not preserve the target number of leaves learning parameter.

4 Experimental Setup

We conducted experiments on the MSLR-WEB10K learning-to-rank dataset.[1] The dataset is pre-folded, providing 720K training, 240K validation, and 240K test instances, with a total of 136 features. We repeated experiments on all five folds and report both per-fold and average results.

We implemented our proposed methods on top of the open-source jforests library[2] by Ganjisaffar et al. [2]. The code was ported to Hadoop, which allowed us to run multiple experiments in parallel. In order to capture the variance introduced by randomization, we ran many trials (see below). Following standard practice, NDCG [24] was used as the objective. In terms of parameter settings, we used the best values reported by Ganjisaffar et al. for number of leaves (70), feature and data sub-sampling parameters (0.3), minimum observations per leaf (0.5), and the learning rate (0.05). Thus, we are able to replicate the state-of-the-art effectiveness results reported in the previous work.

To train an ensemble, we initialized the LambdaMART algorithm with a random seed S and proceeded with learning. To capture variance, we repeated this process $E = 100$ times, and then again for each fold. To ensure a fair comparison against the unmodified LambdaMART algorithm, we used the same set of random seeds to construct ensembles with our proposed methods.

We evaluated the resulting ensembles using the following metrics:

[1] `http://research.microsoft.com/en-us/projects/mslr/`
[2] `http://code.google.com/p/jforests/`

- **Average maximum depth (d_{avg})**: the average maximum depth of all trees in the ensembles. Specifically, we first compute the average maximum depth of each tree in the ensemble, then compute the mean across all trials.
- **Average ensemble size (T_{avg})**: the average number of trees in each ensemble, which is important since shallower trees might require larger ensembles (i.e., with more stages) for the learner to converge.
- **Average maximum path length (p_{avg})**: the product of the above two numbers, which quantifies the average worst case performance of an ensemble—the maximum number of instructions that will be executed if all paths through each tree in the ensemble take the longest decision branch.
- **Average total number of nodes (n_{avg})**: the total number of nodes in the entire ensemble, averaged across all trials.
- **Latency**, our metric for efficiency (performance): the latency of evaluating a single instance using the tree ensemble. We measure the time between when a feature vector is provided to the model to when a relevance score is computed. Mean latency across all trials is reported in microseconds.
- **NDCG**, our metric for effectiveness: average NDCG (across all trials) at different cutoffs for each fold. We used a Wilcoxon signed-rank test ($p < 0.05$) to determine statistical significance.

5 Results

Table 1 summarizes the results for all our experiments, averaged across all trials and all five folds, for both the modified splitting criterion in Table 1(a) and the pruning technique in Table 1(b). The first column shows the λ and α settings; the next columns characterize the tree topology: average maximum depth (d_{avg}), average ensemble size (T_{avg}), average maximum path length (p_{avg}), and average total number of nodes in the ensemble (n_{avg}). The next set of columns show NDCG values at various cutoffs. The final two columns show the prediction latency (with 95% confidence interval in parentheses) and relative improvement over the baseline. The first entry in Table 1(a) and Table 1(b) show the unmodified LambdaMART baseline.

For the approach involving modifications to the CART splitting criterion: results suggest that increasing λ (i.e., the relaxation parameter) yields shallower trees—more precisely, trees with smaller average maximum depths. In some cases, the sizes of the ensembles increase slightly, but overall, the average maximum path length decreases. Note, however, that the average total number of nodes in the ensembles remains about the same—we get shallower and more balanced trees, but they are not more compact. While we observe no significant differences in effectiveness as a result of changing λ, latency does not improve either (i.e., confidence intervals overlap). Figures 1(a) and 2(a), which show NDCG@3 and query latency per fold for different values of λ, suggest that the findings are consistent across all cross-validation folds. For both figures, we show 95% confidence intervals across the trials in each condition. Setting λ to a value greater than 0.95 yields results that are indistinguishable from $\lambda = 0.95$, so for the sake of brevity we do not include them in the results table.

Table 1. Results on the MSLR-WEB10K dataset, averaged across five folds. Columns show topological properties of the ensembles, effectiveness (NDCG at various cutoffs), and latency (with 95% confidence intervals). For NDCG, * denotes a statistically significant difference wrt. to unmodified LambdaMART at $p < 0.05$.

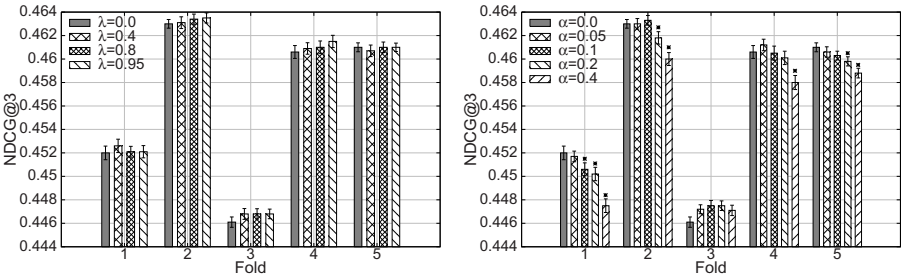
(a) Modified Splitting Criterion

					NDCG						
λ	d_{avg}	T_{avg}	p_{avg}	n_{avg}	@1	@3	@5	@10	@20	Latency	Δ
0.0	17.8	321.2	5716	44115	0.471	0.456	0.461	0.480	0.511	42.2 (±1.5)	
0.4	15.7	327.1	5178	45462	0.471	0.457	0.461	0.480	0.511	42.7 (±1.5)	+1.2%
0.8	12.3	317.8	3945	44173	0.471	0.457	0.461	0.480	0.511	41.7 (±1.4)	−1.2%
0.95	11.7	325.8	3796	44421	0.472	0.457	0.461	0.480	0.511	41.6 (±1.4)	−1.4%

(b) Pruning

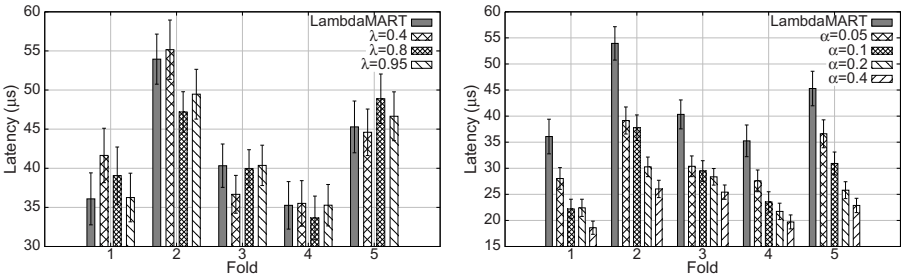
					NDCG						
α	d_{avg}	T_{avg}	p_{avg}	n_{avg}	@1	@3	@5	@10	@20	Latency	Δ
0.0	17.8	321.2	5716	44115	0.471	0.456	0.461	0.480	0.511	42.2 (±1.5)	
0.05	9.1	395.3	3592	30554	0.471	0.457	0.461	0.479	0.510*	32.4 (±1.1)	−23.2%
0.1	7.8	432.2	3211	26228	0.472	0.456	0.460	0.479	0.510*	28.8 (±1.1)	−31.7%
0.2	6.4	474.0	3005	22372	0.471	0.456	0.460	0.478*	0.509*	25.7 (±0.8)	−39.1%
0.4	4.8	567.2	2706	17524	0.469*	0.454*	0.458*	0.476*	0.507*	22.5 (±0.7)	−46.7%
0.6	3.6	675.8	2483	13660	0.467*	0.452*	0.456*	0.474*	0.505*	18.5 (±0.5)	−56.2%

For the pruning approach: increasing the parameter α improves latency significantly. There is a considerable increase in the number of trees in the ensembles as we increase α, but the individual trees are much shallower—leading to an overall decrease in average maximum path length through the ensembles (even less than using the modified splitting criterion). Furthermore, the average total number of nodes (n_{avg}) in the ensembles decreases significantly (unlike with the modified splitting criterion). With pruning, we obtain compact and shallow trees (though not necessarily balanced by the design of the α parameter). However, aggressive pruning comes at the cost of lower effectiveness. As we increase α, we obtain significantly lower NDCG values, first at larger cutoffs, then at smaller cutoffs. However, these reductions are very small (due to the large number of test instances, the dataset has great resolving power to detect significant but small differences)—one might even question whether these differences are meaningful from the user perspective in a real search engine. At $\alpha = 0.6$, we obtain significantly lower NDCG at all cutoff values, and thus we did not further explore larger values of α. Figures 1(b) and 2(b) show per-fold NDCG@3 and query latency with 95% confidence intervals for different values of α. Overall, $\alpha = 0.2$ appears to be a good setting, yielding approximately 40% decrease in latency while decreasing NDCG@10 and NDCG@20 by only a tiny amount and leaving NDCG at the other cutoffs unaffected.



(a) Modified Splitting Criterion (b) Pruning

Fig. 1. NDCG@3 across all trials per fold, with 95% confidence intervals. * denotes a stat. sig. difference wrt. to unmodified LambdaMART at $p < 0.05$.



(a) Modified Splitting Criterion (b) Pruning

Fig. 2. Mean latency (in μs) across all trials per fold, with 95% confidence intervals

6 Discussion

The starting point of this work is that we should "encourage" LambdaMART to learn compact, shallow, and balanced trees. It seems intuitive why trees with such topologies would be more efficient, but results do not fully support these intuitions. It doesn't appear that better balanced trees actually have an impact on latency and variance of latency: this is shown in Figure 2(a). Results also show that average maximum path length does not predict performance, since both strategies reduce that figure. In one case latency is unaffected (modified splitting criterion), yet in the other case (pruning) latency decreases significantly.

It seems of the topological features we explored, the only one that impacts efficiency is compactness—the total number of nodes in the ensemble. This is most poignantly illustrated by comparing $\lambda = 0.95$ with the modified splitting criterion and $\alpha = 0.05$ with the pruning approach. Both conditions yield roughly the same average maximum path length, but pruning is significantly faster. The critical difference is that modifying the splitting criterion preserves the number of nodes in each tree, whereas pruning reduces the total number of nodes. With the first, we've simply made the trees better balanced, thus decreasing the maximum

depth. Because of this, the average number of branch instructions needed to traverse each tree is close to the maximum depth. On the other hand, the pruning approach not only decreases the maximum depth, but also the total number of nodes per tree—this means that the average decision path requires fewer branch instructions. This, in short, explains why pruning is a superior strategy. One might suggest using average depth (i.e., average of the depths of all the terminal nodes in a tree) to model performance, but this makes the assumption that all paths through each are equally likely, which does not turn out to be the case—branches are taken with unequal probability, depending on the characteristics of the test data. Furthermore, modern processor architectures make this model overly simplistic—most of the prediction latency comes from branch mispredicts (which cause pipeline stalls), and it is not clear how we would model branch prediction in the context of trees.

Our experiments show that pruning is an effective technique for substantially increasing the performance of tree-based models, but one potential objection might be: are we measuring the right thing? In our experiments, prediction time is measured from when the feature vector is presented to the model to when the prediction is made. Critically, we assume that features have already been computed. What about an alternative architecture where features are computed lazily, i.e., when the predicate at a tree node needs to access a particular feature?

This alternative architecture where features are computed on demand is difficult to study, since results will be dependent on the implementation of the feature extraction algorithm. However, there is an easier way to study this issue—we can trace the execution of the full tree ensembles and keep track of the fraction of features that are accessed. If during the course of making a prediction, most of the features are accessed, then there is little waste in computing all the features ahead of time. Analysis shows that with unmodified LambdaMART, evaluating a test instance requires, on average, 95.6% ($\pm$1.6%) of the features. Therefore, it makes sense to separate feature extraction from prediction.

7 Conclusion

In this paper, we show how to obtain the "best of both worlds" (effectiveness and efficiency) with GBRTs for learning to rank—by modifying the learning algorithm to perform stagewise pruning during the boosting process. With this novel approach, one particular setting reduces prediction latency by approximately 40% with only a minimal impact on effectiveness. This serves as a good example of how we can learn to *efficiently* rank.

Acknowledgments. This work has been supported by NSF under awards IIS-0916043, IIS-1144034, and IIS-1218043. Any opinions, findings, or conclusions are the authors' and do not necessarily reflect those of the sponsor. The first author's deepest gratitude goes to Katherine, for her invaluable encouragement and wholehearted support. The second author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob.

References

1. Li, H.: Learning to Rank for Information Retrieval and Natural Language Processing. Morgan & Claypool Publishers (2011)
2. Ganjisaffar, Y., Caruana, R., Lopes, C.: Bagging gradient-boosted trees for high precision, low variance ranking models. In: SIGIR 2011 (2011)
3. Burges, C.: From RankNet to LambdaRank to LambdaMART: An overview. Technical Report MSR-TR-2010-82, Microsoft Research (2010)
4. Chapelle, O., Chang, Y., Liu, T.Y.: Future directions in learning to rank. In: JMLR: Workshop and Conference Proceedings 14 (2011)
5. Wang, L., Lin, J., Metzler, D.: A cascade ranking model for efficient ranked retrieval. In: SIGIR 2011 (2011)
6. Xu, Z., Weinberger, K., Chapelle, O.: The greedy miser: Learning under test-time budgets. In: ICML 2012 (2012)
7. Panda, B., Herbach, J., Basu, S., Bayardo, R.: PLANET: Massively parallel learning of tree ensembles with MapReduce. In: VLDB 2009 (2009)
8. Svore, K., Burges, C.: Large-scale learning to rank using boosted decision trees. In: Bekkerman, R., Bilenko, M., Langford, J. (eds.) Scaling Up Machine Learning. Cambridge University Press (2011)
9. Ye, J., Chow, J., Chen, J., Zheng, Z.: Stochastic gradient boosted distributed decision trees. In: CIKM 2009 (2009)
10. Tyree, S., Weinberger, K., Agrawal, K.: Parallel boosted regression trees for web search ranking. In: WWW 2011 (2011)
11. Burges, C., Ragno, R., Le, Q.: Learning to rank with nonsmooth cost functions. In: NIPS 2007 (2007)
12. Friedman, J.: Greedy function approximation: A gradient boosting machine. The Annals of Statistics 29(5), 1189–1232 (2001)
13. Breiman, L., Friedman, J., Stone, C., Olshen, R.: Classification and Regression Trees. Chapman and Hall (1984)
14. Breiman, L.: Bagging predictors. Machine Learning 24(2), 123–140 (1996)
15. Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001)
16. Ganjisaffar, Y.: Tree ensembles for learning to rank. PhD thesis, UC Irvine (2011)
17. Margineantu, D., Dietterich, T.: Pruning adaptive boosting. In: ICML 1997 (1997)
18. Tibshirani, R.: Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society, Series B 58, 267–288 (1994)
19. Martínez-Muñoz, G., Hernández-Lobato, D., Suárez, A.: An analysis of ensemble pruning techniques based on ordered aggregation. IEEE TPAMI 31(2) (2009)
20. Torgo, L.: Sequence-based methods for pruning regression trees. In: Technical Report, LIACC, Machine Learning Group (1998)
21. Cambazoglu, B.B., Zaragoza, H., Chapelle, O., Chen, J., Liao, C., Zheng, Z., Degenhardt, J.: Early exit optimizations for additive machine learned ranking systems. In: WSDM 2010 (2010)
22. Stadler, W.: Multicriteria Optimization in Engineering and in the Sciences. In: Mathematical Concepts and Methods in Science and Engineering. Springer (1988)
23. Osyczka, A.: Multicriterion Optimization in Engineering with FORTRAN Programs. E. Horwood (1984)
24. Järvelin, K., Kekäläinen, J.: Cumulative gain-based evaluation of IR techniques. ACM TOIS 20(4) (2002)