Mach Learn (2013) 93:293–320
DOI 10.1007/s10994-013-5364-5

BoostingTree: parallel selection of weak learners in boosting, with application to ranking

Levente Kocsis · András György · Andrea N. Bán

Received: 24 January 2012 / Accepted: 18 April 2013 / Published online: 7 June 2013

Abstract Boosting algorithms have been found successful in many areas of machine learning and, in particular, in ranking. For typical classes of weak learners used in boosting (such as decision stumps or trees), a large feature space can slow down the training, while a long sequence of weak hypotheses combined by boosting can result in a computationally expensive model. In this paper we propose a strategy that builds several sequences of weak hypotheses in parallel, and extends the ones that are likely to yield a good model. The weak hypothesis sequences are arranged in a *boosting tree*, and new weak hypotheses are added to promising nodes (both leaves and inner nodes) of the tree using some randomized method. Theoretical results show that the proposed algorithm asymptotically achieves the performance of the base boosting algorithm applied. Experiments are provided in ranking web documents and move ordering in chess, and the results indicate that the new strategy yields better performance when the length of the sequence is limited, and converges to similar performance as the original boosting algorithms otherwise.

Keywords Boosting · Random search · Ranking

Editors: Eyke Hüllermeier and Johannes Fürnkranz.

L. Kocsis (✉) · A.N. Bán
Data Mining and Web Search Research Group, Informatics Laboratory, Institute for Computer Science and Control, Hungarian Academy of Sciences (MTA SZTAKI), Budapest, Hungary
e-mail: kocsis@sztaki.hu

A.N. Bán
e-mail: bana@ilab.sztaki.hu

L. Kocsis
Eötvös Loránd University, Budapest, Hungary

A. György
Department of Computing Science, University of Alberta, Edmonton, Canada
e-mail: gyorgy@ualberta.ca

1 Introduction

Boosting algorithms (Schapire 2002) have been found successful in many areas of machine learning and, in particular, in ranking (Burges et al. 2011). For typical classes of weak learners used in boosting (such as decision stumps or trees), a large feature space can slow down the training considerably. A natural way to accelerate the training is to limit the set of features accessible to the weak learner at certain decision points. For example, when the weak learner attempts to create a decision stump, the selection of feature is limited to a subset of all features (or even to one particular feature). Such, an approach was followed by Busa-Fekete and Kégl (2009, 2010) using multi-armed bandit algorithms (see, e.g., Auer et al. 2002a, 2002b) to narrow the freedom of the weak learners.

Another drawback of boosting algorithms is that in order to improve on the performance of individual weak learners, such algorithms tend to cumulate a long sequence of (weak) hypotheses resulting in a computationally expensive model. This problem is exacerbated when bandit algorithms are used to speed up the training, since the exploring nature of bandit algorithms will include much weaker hypotheses, and a larger number of these are needed to achieve similar performance. One way to avoid the inclusion of poor hypotheses in the final ensemble is to revisit some of these decisions later. In this way effectively a tree of weak hypotheses (and their combination weights) is built, where, for any node, the path from the root to the node determines a weighted sequence of hypotheses. The number of possible children of a node is determined by the number of possible next hypotheses to be chosen. While a standard boosting algorithms just selects a hypothesis in each step, here first a node (a leaf or an inner node that does not have the maximum number of children) has to be selected, and then a weak hypothesis extending the hypothesis sequence leading to this node.

The above procedure essentially runs several (dependent) boosting algorithms in parallel. This idea is very similar to improving local search algorithms by having run several search instances in parallel. The latter problem is discussed in our previous paper (György and Kocsis 2011). There we proposed an efficient algorithm, called MetaMax, that dynamically starts instances of a local search algorithm and allocates resources to the instances based on their (potential) performance. The major difference between the two problems is that in our current set-up new sequences are created from old ones, instead of starting each algorithm instance from scratch, introducing serious inter-dependence between the different boosting runs. Furthermore, the boosting methods considered must differ to produce different hypothesis sequences. Therefore, constraints on selecting the next weak hypothesis in each node at each time instant are necessary. In this paper, we propose a new algorithm, BoostingTree, that borrows the core ideas behind selecting the promising algorithm instances from the MetaMax algorithm, and adapts them to the tree-based structure of hypotheses sequences.

While the proposed algorithm can be of interest in many applications of machine learning, ranking problems are particularly suited for this approach. First, in many ranking problems, such as web search or move ordering in games, a fast model is essential. Moreover, the ranking evaluation measures are notoriously non-smooth, therefore virtually all learning algorithms optimize a different measure, and combining these algorithms with a strategy that 'corrects' their decisions when alternative decisions could improve performance with respect to the 'real' evaluation measure should only improve their performance.

The rest of the paper is organized as follows. Section 2 summarizes related research. The proposed algorithm is described in Sect. 3, while theoretical considerations are given in Sect. 3.1. Simulation results on web search benchmarks and move ordering in chess are described in Sect. 4. Conclusions and future work are provided in Sect. 5.

2 Related work

The importance of restricting the number of hypotheses combined by boosting was noted by Margineantu and Dietterich (1997), proposing several methods to achieve this including early stopping, pruning techniques that aim to diversify the hypotheses included in the final model (such as Kappa pruning), and the Reduce-Error pruning that greedily adds hypotheses (from the larger pool) to reduce error on a pruning (or validation) set. While the boosting pruning problem was proven to be NP-complete by Tamon and Xiang (2000), several algorithms have been proposed to select a small set of hypotheses from a larger ensemble (see e.g. Martínez-Muñoz et al. 2009; Tsoumakas et al. 2009 for an overview). It is worth noting that most of the pruning algorithms are more effective when the ensembles are created by bagging rather than boosting, although the latter provides a natural ordering of the hypotheses (Martínez-Muñoz et al. 2009).

A more effective way of achieving good performance in boosting with a reduced number of hypotheses is through l_1 regularization of the weights attached to hypotheses, combined often with early stopping. This approach was chosen by, for example, Xi et al. (2009), Xiang and Ramadge (2009). While we believe that l_1 regularization can improve boosting algorithms for ranking problems as well, adjusting these algorithms are out of the scope of this paper (moreover, such regularization can be used in addition to the BOOSTINGTREE algorithm as well) and therefore we do not attempt to compare the proposed algorithm to these techniques. For simplicity, we restrict our attention in the experimental work to early stopping for limiting the number hypotheses, while keeping in mind that alternative approaches can improve any of the tested algorithms.

Accelerating the training of boosting algorithms was discussed by Escudero et al. (2000) proposing several algorithms, including LazyBoost that reduces the set of features considered in each iteration to a random subset. Busa-Fekete and Kégl (2009) improved on this algorithm by posing the feature selection as a multi-armed bandit problem, and using the bandit algorithm UCB (Auer et al. 2002a) to focus the selection on more informative features while keeping some exploration in the process. Since, the selection of features in subsequent boosting iterations is non-stationary, Busa-Fekete and Kégl (2010) argued for the more sensible use of adversarial bandit algorithms such as EXP3 (Auer et al. 2002b). The latter will be revisited in Sect. 4.

As we mentioned before, the BOOSTINGTREE algorithm is based on the METAMAX algorithm (György and Kocsis 2011), an algorithm specifically designed to speed up local search algorithms by running several search instances in parallel. Therefore, it is natural to consider the alternatives to METAMAX discussed by György and Kocsis (2011) as alternatives to BOOSTINGTREE as well. In particular, the bandit based approach by Streeter and Smith (2006) could be used to alternate between boosting sequences, however, it is not clear how to adapt the algorithm beyond selecting amongst sequences with a fixed start-up (such as varying the initial model). Luby et al. (1993) proposed an anytime algorithm that, in our context, would translate to running (possibly randomized) boosting algorithms repeatedly and stopping them after a prescribed number of iterations (that varies from instance to instance). While the algorithm would produce a plethora of hypothesis sequences with varying length, some randomization needs to be introduced in the boosting algorithm, otherwise all sequences would be just prefixes of the longest sequence. If, in the spirit of the experiments described by György and Kocsis (2011), the sequences would vary only in the initial model that would limit the improvement potential on later hypotheses.

The allocation strategy of the METAMAX algorithm is inspired by the DIRECT optimization algorithm of Jones et al. (1993). Recently, Munos (2011) provided a generalization

of the DIRECT algorithm, called the simultaneous optimistic optimization (SOO) algorithm. Both algorithms are designed to find the optimum of a smooth function, without knowledge of the actual smoothness. In order to do so, the algorithms build a search tree, partitioning the input space, where each leaf node represents a set from the actual partition. Since the selection mechanism of the METAMAX algorithm is based on that of the DIRECT algorithm, the mechanism of Munos (2011) could also be used to improve METAMAX, and by association, the BOOSTINGTREE algorithm. While the DIRECT and the SOO algorithms build trees, just like BOOSTINGTREE, an important difference is that the former algorithms build full trees, while the BOOSTINGTREE adds the children of a node one by one.

While multi-armed bandit approaches to boosting have been discussed above, it could be interesting to look at tree based variants of the bandit algorithms, such as UCT (Kocsis and Szepesvári 2006). Some of our preliminary empirical analysis (not reported in this paper) indicated that UCT would spend too much effort in getting the first few hypotheses right (a natural behavior for games, where the move in the current position has to be decided), and too little attention is paid to subsequent weak learners that can have the same influence in the combined performance. Nested Monte-Carlo search (Cazenave 2009) has been applied to one-player games, and could be used for the same purpose as the BOOSTINGTREE algorithm, but we leave the analysis of these Monte-Carlo algorithms for future research.

Another approach that builds a tree of alternatives is the alternating decision tree (ADT) algorithm (Freund and Mason 1999). While both ADT and BOOSTINGTREE are applied to boosting, a major difference is that while in ADT the constructed tree is one model, in our approach the leaves of the tree are individual models.

3 The BOOSTINGTREE algorithm

In this paper we consider speeding up boosting algorithms. Given a set of instances $\mathcal{X}$, the goal of a boosting algorithm is to create a model $M : \mathcal{X} \to \mathbb{R}$ from weak hypotheses (generated by weak learners) $w_j \in \mathcal{W}$, where $\mathcal{W} \subset \{w : \mathcal{X} \to \mathbb{R}\}$ is a set of weak hypotheses. Each $x \in \mathcal{X}$ is usually described by a feature vector, and with a slight abuse of notation the feature vector representing x will also be denoted by x. The goal of constructing the model can be any supervised learning task; in this paper we will be concerned about creating models for ranking. The model M is built iteratively in a linear fashion from the weak hypotheses: a model M is an ensemble $M = \sum_{j=1}^{l(M)} w_j$, where $l(M)$ is the number of weak hypotheses used in constructing M, which will often be referred to later as the 'length of the boosting sequence', and in each step of the algorithm a weak hypothesis w_j is added to the existing model. While typically a model is a *weighted* combination of weak hypotheses, to simplify the notation we assume that the weight is already included in the hypotheses w_i in the above formulation. The selection of the next hypothesis depends on the current model M, the training data D, and optionally on a constraint C that restricts the new weak hypothesis to be an element of a restricted set $\mathcal{W}_C \subseteq \mathcal{W}$. Thus, one step of the boosting algorithm A encapsulates the reweighting of the data, the computation of a hypothesis by the weak learner (subject to the constraint), and the weighting of the hypothesis. That is, if the actual model is $M_i = \sum_{j=1}^{l} w_j$, the next weak hypothesis selected by the standard boosting algorithm is $w_{i+1} = A(M_i, D)$.

As mentioned before, a single step of boosting algorithms may be speeded up by imposing some constraint on the next weak hypothesis. Given a constraint C_i, the constrained boosting algorithm A selects the next weak hypothesis $w_{i+1} = A(M_i, C_i, D)$ (the unconstrained version can be described in this way as $A(M_i, \emptyset, D)$). A typical case is when the

weak hypotheses are decision stumps (depending only on a single feature of the observations), and the constraint C_i selects a single feature: then the step $A(M_i, C_i, D)$ is nothing but optimizing a weak learner w_{i+1} over the set of decision stumps corresponding to the feature selected by C_i. We denote the measure of the performance of the model M with data D by $f(M, D)$. Throughout this paper this measure will typically be some ranking measure $f(M, D) = V(M, D)$ that evaluates the model M on the particular dataset D, but the provided method works for any other performance measure. Since it is possible that the performance of a model is decreased by adding more weak hypotheses, we also define the function $\hat{f}$ to be the maximum of the ranking measure over the possible prefixes of an ensemble, that is,

$$\hat{f}(M) = \max_{0 \le j \le l(M)} f\left(\sum_{i=1}^{j} w_i, D\right).$$

While standard boosting algorithms typically produce a deterministic sequence of weak learners (for a given data D), allowing several different constraint values for each model leads to a tree: the edges of the tree correspond to (weighted) weak hypotheses, while any node in the tree is a model that sums the hypotheses that are on the path from the root to the node. Assuming constraints $C_1, \ldots, C_k$ can be selected at a node M, the children of M correspond to the weak hypotheses $\{A(M, C_i, D)\}_{i=1,\ldots,k}$ as edges. Assuming the available constraints at each node are fixed in advance, the tree can be built in several ways. In this section we propose an algorithm, called BOOSTINGTREE, that provides a systematic way of exploring the tree. The key component of BOOSTINGTREE is the way it selects the nodes (i.e., the models) to extend with a new hypothesis. Before describing this selection mechanism, we first revisit the METAMAX algorithm (György and Kocsis 2011) that forms the basis of the selection procedure.

The goal of the METAMAX algorithm is to find a maximizer x^* of a function f, using a given local search algorithm. As we discussed in the introduction, METAMAX runs several instances of the local search algorithm in parallel, and allocates in every step additional resources to the most promising instances. The key idea of METAMAX is the assumption of bounding the convergence of the local search instances by a function $cg(n)$, where n is the number of steps taken by the local search algorithm, g is typically an exponential function, and c is an unknown constant. The algorithm then steps every local search instance that has the highest optimistic bound (the actual best value plus the confidence bound $cg(n)$) on the performance for some particular range of values of c. The idea behind the algorithm, first introduced in the context of Lipschitz optimization without the knowledge of the Lipschitz constant (Jones et al. 1993), is that if the convergence rate $g(n)$ is only known up to a constant factor, then instead of selecting an arbitrary value of this constant, all algorithms are considered that are optimal for some values of the constant. As the function g is also unknown, it is replaced by a sequence of functions $\{h_r\}$ chosen such that it aids the convergence of the METAMAX algorithm. The METAMAX algorithm is shown in Fig. 1, for completeness. At the beginning of each round r, a new instance is created randomly (instance A_r), and the most promising instances are selected in step (b), where n_i denotes the number of steps made by A_i, $\hat{f}_i$ is its current estimate of the optimum, and $\hat{f}_i + \hat{c}h_r(n_i)$ is the optimistic estimate on the performance of A_i with assumed convergence rate $\hat{c}h_r(n)$ (note that both n_i and $\hat{f}_i$ may change from round to round). The selection procedure will be explained in more details in the context of the BOOSTINGTREE algorithm. The most promising algorithms are selected and used to make another step. Finally, in step (c′), it is ensured that the instance with the currently best estimate of the optimum makes the most

METAMAX: A MULTI-START STRATEGY WITH INFINITELY MANY ALGORITHM INSTANCES.

Parameters: $\{h_r\}$, a set of positive, monotone decreasing functions with $\lim_{n\to\infty} h_r(n) = 0$.
For each round $r = 1, 2, \ldots$

(a) Initialize algorithm A_r by choosing uniformly random starting point $X_{i,0}$, evaluating $f(X_{i,0})$, and by setting $n_r = 0$, $\hat{f}_r = f(X_{i,0})$.
(b) For $i = 1, \ldots, r-1$ select algorithm A_i if there exists a $\hat{c} > 0$ such that

$$\hat{f}_i + \hat{c}h_{r-1}(n_i) > \hat{f}_j + \hat{c}h_{r-1}(n_j)$$

for all $j = 1, \ldots, r-1$ such that $(n_i, \hat{f}_i) \neq (n_j, \hat{f}_j)$. If there are several values of i selected that have the same step number n_i then keep only one of these that has the smallest index.
(c) Step each selected A_i obtaining the sample point X_{i,n_i}, and update variables. That is, for each selected A_i, set $n_i = n_i + 1$, evaluate $f(X_{i,n_i})$ and if $f(X_{i,n_i}) > \hat{f}_i$ then set $\hat{X}_i = X_{i,n_i}$, $\hat{f}_i = f(X_{i,n_i})$.
(c′) Let $I_r = \text{argmax}_{i=1,\ldots,r} \hat{f}_i$ denote the index of the algorithm with the currently largest estimate of f^* (in case I_r is not unique, choose the one with the smallest number of steps n_i). If $I_r \neq I_{r-1}$, step algorithm A_{I_r} $(n_{I_{r-1}} - n_{I_r} + 1)$ times and set $n_{I_r} = n_{I_{r-1}} + 1$, and $\hat{f}_{I_r}$ and X_{I_r} as the largest value of f and its location, respectively, encountered by algorithm A_{I_r}.
(d) Estimate the location of the maximum with $\hat{X}_r = \hat{X}_{I_r}$ and its value with $\hat{f}_r = \hat{f}_{I_r}$.

Fig. 1 The METAMAX algorithm. The algorithm attempts to maximize a real-valued function $f(X)$ with the help of local search instances A_i that differ in their initial point $X_{i,0}$. The algorithm allocates further time to instances that appear more promising (step (b) and (c)) and gives high priority to instances that suddenly provide the best estimate having being less promising previously (step (c′)). New instances are started in every round (step (a))

steps. While introducing this step does not have much effect in practice (typically, if the best algorithm is not the one with the most steps, its step number would increase very quickly in short rounds, even without (c′)), it helps the algorithm in certain pathological cases, and also aids its theoretical analysis (see György and Kocsis 2011 for a more detailed discussion).

The boosting tree algorithm employs this idea in the node selection procedure. Nodes (models) are added to the tree in rounds. In each round r, for each node M_i already added to the tree, the performance of M_i is evaluated as $\hat{f}_i = \hat{f}_i(M_i)$, and an estimate of the potential performance of extending M_i is formed as

$$\hat{f}_i + \hat{c}h_r(n_i) \tag{1}$$

where h_r is some decreasing positive function (typically exponential) and n_i is the cumulative length of M_i, which is defined as the sum of the length of the model M_i and the number of children of M_i already in the tree. Then the algorithm expands those nodes which maximize the performance estimate (1) for some values of $\hat{c}$. While it may seem that almost all models will be selected in this way (for different values of $\hat{c}$), it is easy to see that exactly those models are selected that lie on the corners of the upper convex hull H of the set of

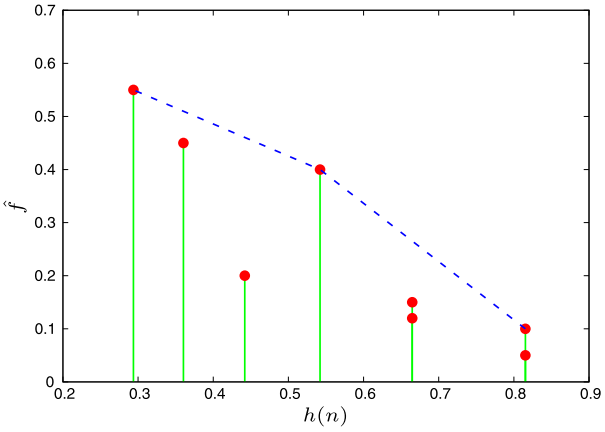

Fig. 2 Selecting models for expansion in BOOSTINGTREE: the points represent the models, and the models that lie on the corners of the upper convex hull (drawn with *dashed lines*) are selected

points $\{(h_r(n_i), \hat{f}_i)\} \cup \{(0, \max_i \hat{f}_i)\}$ where i ranges over all models M_i that are already in the tree, see Fig. 2 (the reason for this is that $\hat{f}_i + \hat{c}(h_r(n_i) - x)$ is a tangent line of H if i maximizes (1) for $\hat{c}$).

Thinking of a sequence of weak hypotheses as a local search algorithm, the above selection rule for leafs is the same as that of the METAMAX algorithm, as for leaf nodes the cumulative length is exactly the number of hypotheses in the corresponding model. For internal nodes some adjustment needs to be made to avoid excessive attention to models that are hard to improve: Note that for many local search algorithms, including boosting, there are situations when the current model can only be improved slightly or cannot be improved at all by a single step. Then, if h_r sufficiently prefers lower complexity models (i.e., ensembles that are sums of fewer hypotheses), and we considered the model length instead of the cumulative length, a hardly improvable model would typically be selected if at least one of its children were selected. Our solution to this problem is to introduce the notion of cumulative length of a node, which is the length of the path to the node plus the number of children of the node. Intuitively, this modification yields that the probability of expanding a child of a node (even with a higher potential than other far descendants, grandchildren and more) decreases as more and more children of a given node are expanded.

The BOOSTINGTREE algorithm is shown in Fig. 3. Note that for a model M_i, l_i denotes the depth of the corresponding node, the cumulative (or extended) length n_i is the depth plus the number of children already generated, f_i is the actual performance of the model, $\hat{f}_i$ is the performance of the best ancestor of M_i. Furthermore, t_{r-1} is the number of models in the tree generated before round r. Note that, unlike to the METAMAX algorithm, most values defined in the algorithm are static, and only the cumulative length variables n_i depend implicitly on r, for example, f_i and $\hat{f}_i$ remain unchanged once node i is created.

The algorithm starts with an empty tree, and in every phase expands the nodes that have the potential of leading to the highest performing model (step (a)). For each selected node, a constraint is set on the weak learners and the selected node is extended by a new hypotheses according to the boosting algorithm. The variables of the selected node and the new node are updated according to step (b). The final step (c) enforces after each iteration that we expand the model with best ranking measure until it leads to the longest path in the tree, an idea also borrowed from the METAMAX algorithm. Although this last step usually does not influence strongly the algorithm, it slightly speeds up the convergence when a new and shorter 'leader' is found (in fact, if such a step would not be present such an 'overtake' would be followed by very short phases, until the leading path is expanded for a sufficient

Parameters: $\{h_r\}$, a set of positive, monotone decreasing functions with $\lim_{n\to\infty} h_r(n) = 0$, training data D.
Initialization: $M_0 = \emptyset$, $t_0 = 1$, $\hat{f}_0 = 0$, $n_0 = 0$, and $l_0 = 0$.
For each round $r = 1, 2, \ldots$

(a) For $i = 0, \ldots, t_{r-1}$ select model M_i if there exists a $\hat{c} > 0$ such that

$$\hat{f}_i + \hat{c}h_{r-1}(n_i) > \hat{f}_j + \hat{c}h_{r-1}(n_j)$$

for all $j = 0, \ldots, r$ such that $(n_i, \hat{f}_i) \neq (n_j, \hat{f}_j)$. If there are several values of i selected that have the same cumulative length n_i then keep only the one with the smallest length l_i, and in case of tie, with the highest f_i.

(b) Set $t_r = t_{r-1}$. For each selected M_i:
- set $t_r = t_r + 1$;
- select constraint C_{t_r};
- compute weak hypothesis $w_{t_r} = A(M_i, C_{t_r}, D)$;
- construct new model $M_{t_r} = M_i + w_{t_r}$;
- evaluate the new model: $f_{t_r} = V(M_{t_r}, D)$, $\hat{f}_{t_r} = \max(\hat{f}_i, f_{t_r})$;
- compute (and update) length variables: $n_i = n_i + 1$, $n_{t_r} = l_i + 1, l_{t_r} = l_i + 1$.

(c) Let $I_r = \text{argmax}_{i=1,\ldots,t_r} \hat{f}_i$ denote the index of the algorithm with the currently largest value of $\hat{f}_i$ (in case I_r is not unique, choose the one with the largest length l_i). While either there exists $j \neq I_r$ such that $l_{I_r} \leq l_j$ or M_{I_r} has not been expanded in the current round, select M_{I_r} and proceed as in step (b) and repeat the current step (step (c)).

Fig. 3 The BOOSTINGTREE algorithm

number of times). Furthermore, this step helps to handle some pathological situations, and simplifies the theoretical analysis of the algorithm.

Note that one could also try to approach the problem by running several instances of a boosting algorithm independently (with some randomization or with a diverse set of constraints applied for the different runs) to obtain efficient models. Then, for example, one could run these instances in parallel, and use the METAMAX algorithm to select which instances should be continued. However, such an approach would neglect the strong correspondence between the different runs. Taking into account these correspondences leads to the tree structure and the BOOSTINGTREE algorithm described above. One can, of course, run METAMAX over runs of *different* boosting algorithms (including even BOOSTINGTREE), but this is an orthogonal issue and is beyond the scope of the present paper.

3.1 Some theoretical results

In this section we provide some basic properties of the BOOSTINGTREE algorithm, following the analog analysis for the METAMAX algorithm (György and Kocsis 2011). First note that the base boosting algorithm A used inside BOOSTINGTREE would provide a sequence of models $M_r^A = M_{r-1} + A(M_{r-1}, \emptyset, D)$, $r = 1, 2, \ldots$ (with $M_0 = \emptyset$), thus, in each iteration of the algorithm, the model is augmented with a weak hypothesis computed without any constraints. It is reasonable to expect that our BOOSTINGTREE algorithm can mimic the performance of A if a node is expanded sufficiently many times, that is, a sufficiently rich

set of constraints are used in the selection process. The following assumption ensures this property.

Assumption 1 For any node M of the BOOSTINGTREE assume that the node can have only finitely many children, each defined by a different constraint, such that for one of these constraints, C, we have $A(M, C, D) = A(M, \emptyset, D)$.

It is easy to see by induction that the above assumption implies that the full infinite boosting tree (where each node has its maximum number of children) contains all the models M_r^A generated by the boosting algorithm A such that M_{r+1}^A is a child of M_r^A, for each $r = 0, 1, 2, \ldots$.

Furthermore, notice that the selection procedure (a) implies that a model M_i with the smallest step number n_i is always selected (for the largest values of $\hat{c}$). Since each node can have only a finite number of children, for any n there is a bounded number of nodes in the tree whose step number can be at most n (if a node at depth l can have at most k children, then its step number is always at most $l + k$). This implies that each node of the infinite boosting tree is expanded by a maximum number of times if BOOSTINGTREE is run for a sufficiently long time.

The above two facts imply the following consistency result:

Theorem 1 *Suppose Assumption 1 holds. Then any model M_r^A generated by the original boosting algorithm A is also generated by the* BOOSTINGTREE *algorithm if it is run for a sufficiently long time. As a consequence, the asymptotic training performance of* BOOSTINGTREE *is at least as good as that of A:* $\lim_{r\to\infty} \hat{f}(M_{I_r}) \geq \lim_{r\to\infty} \hat{f}(M_r^A)$.[1]

The goal of BOOSTINGTREE is to find the best model possible on the training data. Since it is not clear which constraints are going to be the most relevant, the algorithm tries to explore several paths in the boosting tree while keeping the most promising ones as long as possible (in an ideal setting the algorithm would explore just the optimal path, spending all resources on it, not wasting effort on suboptimal combinations of weak hypotheses).

The trade-off in this exploration-exploitation problem can be best analyzed by evaluating the length of M_{I_r}, that is, how many weak hypotheses are combined in M_{I_r} at the end of round r. Note that M_{I_r} is the longest extension of the actually best model of the boosting tree (which is typically also the actually best model); therefore we will refer to M_{I_r} as the best leaf-model of the tree. The following theorem shows that the length l_{I_r} of M_{I_r} is of the order of the square root of t_r, the total number of times a weak hypothesis is computed in BOOSTINGTREE (note that the number of nodes in the tree is $t_r + 1$). The proof of this result is a straightforward modification of that of Theorem 15 of György and Kocsis (2011). Note that in practice we have found that the BOOSTINGTREE algorithm behaves better in the sense that the length of M_{I_r} is typically much larger, about $\Omega(t_r / \ln t_r)$; for more details see Sect. 4.7.

Theorem 2 *At the end of any round r the number of weak hypotheses in the best leaf-model M_{I_r} is between r and $2r$. That is,*

$$r \leq n_{I_r} = l_{I_r} < 2r. \tag{2}$$

[1] Note that the limits exist (may be infinite) as they are taken over nondecreasing sequences.

Furthermore,

$$n_{I_r} = l_{I_r} \geq \frac{\sqrt{2t_r + 7} - 1}{2}. \tag{3}$$

Proof The first statement of the lemma is very simple, since in any round the length of the best leaf-model increases by either one or two: If the best leaf-model is expanded in round r according to step (a) and it remains the best-leaf model then trivially $n_{I_r} = n_{I_{r-1}} + 1$. If the best leaf model is expanded but another model becomes better during the round, then the latter model is expanded so many times that the corresponding leaf-model (which, in turn, will become M_{I_r}) be of depth $n_{I_r} = l_{I_r} = l_{I_{r-1}} + 2$. Finally, if $M_{I_{r-1}}$ is not expanded according to step (a), then the length of M_{I_r} has to be set to $l_{I_{r-1}} + 1$. Thus

$$1 \leq n_{I_r} - n_{I_{r-1}} = n_{I_r} - n_{I_{r-1}} \leq 2$$

in all situations. Since in the first round clearly exactly one node of the boosting tree is generated, that is, $n_{I_1} = 1$, (2) follows.

To prove the second part, notice that in any round r, at most $n_{I_{r-1}} + 1$ nodes can be expanded according to step (a) as no algorithm can be used that has taken more steps than the currently best one (and hence the currently best leaf-node). Also, no extra weak hypotheses have to be computed if the conditions in step (c) are met. If not, then at most $n_{I_{r-1}} + 1$ weak hypotheses have to be computed in addition (i.e., the best model has to be extended with at most this many weak hypothesis). Therefore,

$$t_r \leq t_{r-1} + 2n_{I_{r-1}} + 2.$$

Thus, since step (c) has no effect in round 1, we obtain

$$t_r \leq 1 + \sum_{s=2}^{r} 2(n_{I_{s-1}} + 1).$$

Then, by (2) we have

$$t_r \leq 1 + 4\sum_{s=2}^{r} s = 1 + 2(r+2)(r-1) \leq 1 + 2(n_{I_r} + 2)(n_{I_r} - 1)$$

which yields (3). □

3.2 An illustrative example

We illustrate the BOOSTINGTREE algorithm on a synthetic binary classification problem combined with the ADABOOST algorithm (Freund and Schapire 1997) with decision stumps. The classification problem is shown in Fig. 4. A stump $w : \mathbb{R} \to \mathbb{R}$ is defined by a split of the real line represented by a half line H and a weight v such that $w(u) = v$ if $u \in H$ and $w(u) = -v$ if $u \notin H$. The constraints at the nodes restrict the choice of the stumps to a single feature. Denoting the feature selected for the stump w_j by i_j, and the ith coordinate of a feature vector $x \in \mathcal{X}$ by x_i, the model after l steps becomes $M_l(x) = \sum_{j=1}^{l} w_j(x_{i_j})$, and the classifier decides to class $+$ if $M(x) \geq 0$, and to class $-$ otherwise.

The model generated on this problem by a regular ADABOOST algorithm is shown in Table 1, left. While the model classifies all data points correctly, it includes five decision

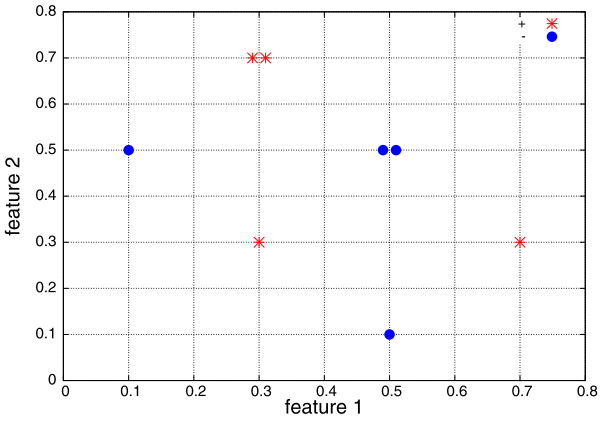

Fig. 4 Synthetic binary classification problem with two features. The positive data points are marked with +, and the negative points with −

Table 1 Models for the synthetic classification problem. The left table shows the sequence of decision stumps generated by a regular ADABOOST algorithm with the corresponding weigths. For each decision stump, the top row specifies the feature to be splitted, and the second row specifies the interval for which positive label is assigned. The table on the right specifies the model generated if in each step the decision stump is constrained to feature 1. For this model, if we sum the weights over the intervals, we have $+0.55 - 0.55 - 0.80 = -0.8$ if feature 1 is less than 0.2 (thus negative label), $+0.8$ between 0.2 and 0.4 (positive), -0.3 between 0.4 and 0.6 (negative), and $+0.8$ above 0.6 (positive)

feature	1	2	2	1	1
split	≤ 0.4	≤ 0.4	> 0.6	> 0.2	> 0.6
weight	0.55	0.55	0.62	0.76	0.67

feature	1	1	1
split	≤ 0.4	> 0.6	> 0.2
weight	0.55	0.55	0.80

stumps, and perfect classification can be obtained with only three decision stumps as shown in Table 1, right, if we split always feature 1. A similarly small model can be obtained by splitting three times feature 2.

In each step of the ADABOOST algorithm, first, the data are re-weighted, then a decision stump is chosen (a feature and a split for that feature is selected), and a weight is computed for the decision stump. When the BOOSTINGTREE algorithm is combined with ADABOOST, it selects a model (a combination of decision stumps) to be extended, and selects the feature to be added to the model. The latter is the constraint on the construction of the new weak hypothesis (i.e., the decision stump to be added). The re-weighting of the data, the selection of the split, and the formula for the weight of the new decision stump remains unchanged from the ADABOOST algorithm.

Two possible runs of the BOOSTINGTREE algorithm are shown in Table 2. The differences in the possible runs are due to the different selection of the features for the new decision stumps (which is the same as the selection of the constraint in step (b) of the algorithm; cf. Fig. 3). Normally, the selection is done by selecting randomly from the features that has not been expanded for the model chosen for expansion in step (a). In the worst case scenario of Table 2, top, instead of random, the feature that least likely to lead to a small optimal model in the algorithm is selected. While for the best case scenario of Table 2, bottom, the feature that leads more likely to such a model is selected.

Table 2 BOOSTINGTREE runs combined with ADABOOST on the synthetic problem of Fig. 4. The separate runs are provided for a worst case (top) and a best case scenario (bottom). Each row corresponds to a feature sequence, that is, to the model obtained with ADABOOST by sequentially constraining the decision stumps in the ensemble with the particular feature from the sequence. The first row in each table corresponds to an empty model. In the error column classification error corresponding to the obtained model (i.e., feature sequence) is given (the error equals $1-f$), while the minimum error ($1-\hat{f}$) over the prefixes of the particular sequence is given in brackets if differs from f. In the columns corresponding to the consecutive rounds (r) the cumulative length (n) is provided. The number is given in bold if the sequence is selected for extension. The letter e specifies the round in which the sequence is evaluated, c is attached if the sequence is extended due to step (c) of the BOOSTINGTREE algorithm, while the letter t marks a tie situation. When a sequence is extended with both features, it will not be considered for further expansion, and, therefore, the corresponding cell in the table is left empty for the following rounds

model	error	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$	$r=6$	$r=7$
–	0.5	**0**	1	**1**				
2	0.25	e	**1**	**2**				
2,1	0.375 (0.25)		e	2	2	2	2	2 (t)
1	0.25			e	**1**	**2**		
2,2	0.375 (0.25)			e	2	2	2	2 (t)
1,2	0.25				e	2	2	**2** (t)
1,1	0.125					e	**2** (c)	**3**
1,1,2	0.125						e	3
1,2,1	0.125							e
1,1,1	0							e
–	0.5	**0**	1	**1**				
1	0.25	e	**1**	2				
1,1	0.125		e	**2**				
1,2	0.25			e				
1,1,1	0			e				

As shown in Table 2, the BOOSTINGTREE algorithm starts with an empty model (the first row in each table). In each round, the 'active' models[2] are considered for expansion with their error ($1-\hat{f}$) and cumulative length (n). Of these candidate models, only those are expanded that obey the properties outlined in step (a) of the BOOSTINGTREE algorithm. In this discussion we will not make the functions h_r more specific since, due to the simplicity of the considered problem, the steps shown in the tables would be independent of the choice of h_r as long as it obeys the specified requirement (i.e., being positive, monotone decreasing function). To understand better how the selection works, consider the candidates in round 3 of the top table, the empty model, the model that includes a decision stump with feature 2, and the model with the feature sequence (2,1). The empty model that has cumulative length equal to 1 (it has model length equal to 0, and has been expanded once) will be selected for expansion since the inequality holds for large $\hat{c}$ (e.g., for $\hat{c}=\infty$). For the other two candidate models the inequality holds for small values of $\hat{c}$ (e.g., for $\hat{c}=0$), but both have the same value $n=2$, and according to the description of the algorithm, we select for expansion only the one with smaller model length. Subsequently the empty model is extended with feature 1

[2]Active models are those that have been evaluated already and have not been expanded with all features available, in this problem by both feature 1 and 2.

(fourth row), given that it was already extended with feature 2. Similarly, the model with the single feature 2 is extended with the same feature 2. For both models the AdaBoost steps follow, thus the data is reweighted, the split is computed for the added feature, the model is augmented with the new decision stump (consisting of the feature and the split), and the new model is evaluated.

Considering how well the BoostingTree algorithm performs on this classification problem, we observe that even in the worst case, the number of steps necessary to obtain a 0-error model for the BoostingTree algorithm is not much higher than for the regular AdaBoost (nine vs. five), with the BoostingTree yielding a smaller model (three decision stumps instead of five). Moreover, even in this case the number of times a split has to be selected is smaller for the BoostingTree combination (nine) compared to how many times the regular AdaBoost looks for a split (ten, since in each step the split is computed for both features). In the best case scenario, only four steps are needed to find the smallest 0-error model.

Looking at some finer details of the BoostingTree runs, in particular the one corresponding to the worst case scenario, we note that step (c) of the algorithm is activated in the sixth round. In this case, it does not alter the course of the algorithm since the (1,1) sequence would have been the only one to be selected anyway, but it would have given a further push if more complex models (with larger length l) had been generated previously, which is favorable since exactly this model was extended subsequently to the optimal model. In the seventh round there is a tie broken based on the classification error of the model, with the sequence (1,2) having the smallest f. Ties with identical $\hat{f}$ and n occurred also in round 3 and 5, and were broken according to the smallest length (l). Finally, in a somewhat more favorable scenario the sequence (2,2) could be extended to the alternative smallest 0-error model, which is (2,2,2). This is an example for the situation of the error first increasing and then decreasing mentioned in Sect. 3.

3.3 Implementation issues

Note that, as shown in the example in the previous section, the training error becomes zero exponentially fast in boosting algorithms. However, when the labels are noisy, it is typically worth to run the training longer, and set the actual number of training iterations using a validation set. When the BoostingTree includes several models with optimal performance on the training set, the shortest models are selected, and the optimal models are grown at the same rate, building a full tree from any optimal node. One can avoid this, for example, by using a validation set to compute $\hat{f}_i$. Note however, that we have not encountered such situations in our experiments.

Boosting algorithms work by weighting the data points, and, in order to reduce the computation time, most implementations take advantage of the possibility to compute incrementally the weights of the data points, and store some auxiliary record for each point as the algorithm proceeds. In the case of AdaBoost, the auxiliary records consist of the current weights and, for the ranking algorithms discussed in Sect. 4.3, they consist of the current score of each document. Since a boosting tree can be expanded at any node, in order to use the same speed-up technique, one would need to keep the auxiliary records for each node of the boosting tree, which may result in prohibitive memory consumption. It is clear that for nodes corresponding to short boosting sequences, recomputing these records is sufficiently fast, and there is no real need to store such auxiliary records. To balance between memory usage and computation time, one can develop heuristics to store the records for the most promising and/or most recently used models only, and recompute them for other models

when necessary. In the experiments, presented in the next section, we have simply decided to always recompute the weights; since most of the tree is shallow (we aim for short models in any case), this has not introduced a large computational overhead, and the observable speed-ups include this overhead.

4 Experiments

In this section we describe the empirical evaluation of the BOOSTINGTREE algorithm using two boosting algorithms for ranking: LAMBDAMART (Sect. 4.3.1) and NDCGBOOST (Sect. 4.3.2). The two boosting algorithms are used either standalone (we refer to these as the standard variants), combined with adversarial bandits (EXP3.P; Busa-Fekete and Kégl 2010), or combined with the BOOSTINGTREE algorithm. Thus in total we have six algorithms. The evaluation measure used in the experiments is the Normalized Discounted Cumulative Gain (NDCG; Järvelin and Kekäläinen 2000), described in Sect. 4.1. Two standard webpage ranking benchmark datasets are used for evaluation (Sect. 4.4 and Sect. 4.5), with an additional benchmark constructed with move ordering in chess (Sect. 4.6). The section ends with a discussion on the results and some observations regarding how BOOSTINGTREE behaves in practice.

4.1 Ranking measure: NDCG

NDCG is one of the most popular measures to evaluate ranking, and has been used in several ranking challenges (see, e.g., Chapelle and Chang 2011). Here we consider a typical ranking problem, where documents should be provided to answer queries. We denote the set of queries by $Q = \{q_1, \ldots, q_n\}$. For each query q_i, a ranker is faced with a set of m_i documents $D_i = \{d_{i1}, \ldots, d_{im_i}\}$. Each document d_{ij} is labeled by a number l_{ij} indicating its relevance with respect to the ith query. The gain of a document is typically defined as $g_{ij} = 2^{l_{ij}} - 1$, with irrelevant documents being labeled 0, and more relevant ones having higher valued labels $(1, 2, 3, \ldots)$. Faced with a query q_i, a ranking algorithm outputs a permutation π of the documents, with π_{ik} denoting the document ranked on the kth position, and conversely, r_{ij} denotes the rank of the document d_{ij}.

The Discounted Cumulative Gain (DCG) of the ith query is usually defined up to K documents as follows:

$$DCG@K_i(\pi) = \sum_{k=1}^{\min(K,m_i)} \gamma_k \, g_{i\pi_{ik}},$$

where γ_k is a discount factor. Järvelin and Kekäläinen (2000) defines the discount factor by $\gamma_k = 1$, if $k = 1$, and $\gamma_k = 1/\log_2 k$, otherwise. This definition is used by the benchmark described in Sect. 4.4, while the benchmark of Sect. 4.5 uses a slightly different definition, with $\gamma_k = 1/(\log_2(k+1))$ that results in a strictly decreasing discount sequence (as opposed to the first definition, where $\gamma_1 = \gamma_2$).

Let $maxDCG@K_i = \max_\pi DCG@K_i(\pi)$ denote the discounted cumulative gain corresponding to an optimal ranking for the ith query. Then normalized discounted cumulative gain (NDCG) of the ith query is defined as

$$NDCG@K_i(\pi) = \frac{DCG@K_i(\pi)}{maxDCG@K_i},$$

and the average NDCG is defined by

$$NDCG@K(\pi) = \frac{1}{n}\sum_{i=1}^{n} NDCG@K_i(\pi).$$

Finally, $MeanNDCG(\pi)$ is defined as the average of the normalized discounted cumulative gain up to the number of documents:

$$MeanNDCG(\pi) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{m_i}\sum_{k=1}^{m_i} NDCG@k_i(\pi).$$

Both ranking measures, $NDCG@K(\pi)$ and $MeanNDCG(\pi)$, need to be maximized. They reach their maxima at 1, for an optimal ranking, and are always non-negative as the lowest value for $g_{i\pi_{ik}}$ is assumed to be non-negative.

4.2 Algorithms

In this section we revisit the two boosting algorithms used in the experiments, and some details of the specific implementation of BOOSTINGTREE and EXP3.P algorithms are also given.

In a typical ranking problem documents related to a query have to be ordered, and each query-document pair is described by a feature vector. Since a model obtained by a boosting algorithm returns a real valued score for each pair, the ordering for a particular query is obtained by sorting the documents according to their scores.

The two boosting algorithms for ranking used in the experiments (LAMBDAMART and NDCGBOOST) apply trees as weak hypotheses (regression and decision trees, respectively). When selecting a weak hypothesis, the tree construction algorithms build a tree by recursively partitioning the input space and assigning a value (or label) to each leaf. When a partition corresponding to a node is being subpartitioned, the tree construction algorithm selects a feature, and partitioning is made by thresholding this feature value. The feature and the corresponding threshold in each internal node, as well as the label of each leaf is selected to minimize some measure, such as the mean squared error for regression trees.

In our algorithms the selection of the weak learner, a regression or decision tree, is often constrained. In these experiments we choose to use constraints that allow partitioning a node of a tree based on a single feature, and the constraint determines for each node which feature should be used. Thus a constraint can be seen as a tree of features, where each node in the constraint tree restricts the choice of feature in the corresponding internal node of the decision tree. The leaves in the decision tree hold the labels for a particular subpartition, and have no corresponding node in the constraint tree. It may happen that a tree construction algorithm decides not to split the data further at some node (for instance, when only one document falls in the node), then some elements of the constraint tree are ignored.

Next we specify details concerning the algorithms applied in the experiments.

4.2.1 The BOOSTINGTREE *algorithm*

There are two important details that have to be specified when implementing the BOOSTINGTREE algorithm: selecting appropriate functions for h_r and choosing how to set constraints on the weak learners. For the former, we opted to the functions $h_r(n) = e^{-n/\sqrt{r}}$, successfully applied in György and Kocsis (2011).

Parameters: exploration rate $0 \leq \lambda \leq 1$, smoothing parameter $\eta > 0$, training data D, set of K features.
Initialization: $M_0 = \emptyset$, $t = 1$, $\omega_k^{(1)} = 1$, with $k = 1, \ldots, K$, and evaluate initial model $f_o = V(M_0, D)$.
For each iteration $t = 1, 2, \ldots$

(a) For each feature k, compute selection probabilities $p_k^{(t)}$:

$$p_k^{(t)} = (1 - \lambda) \frac{\omega_k^{(t)}}{\sum_{k'} \omega_{k'}^{(t)}} + \frac{\lambda}{K}.$$

(b) Select constraint C_t using the set of probabilities $p_k^{(t)}$.
(c) compute weak hypothesis $w_t = A(M_{t-1}, C_t, D)$.
(d) Construct new model $M_t = M_{t-1} + w_t$.
(e) Evaluate the new model: $f_t = V(M_t, D)$, and compute reward function $\tilde{f}_t$.
(f) For each feature k, compute $\rho_k^{(t)}$, $\omega_k^{(t+1)}$:

$$\rho_k^{(t)} = \begin{cases} \frac{\tilde{f}_t}{p_k^{(t)}} & \text{if feature } k \text{ is included in } C_t, \\ 0 & \text{otherwise,} \end{cases}$$

$$\omega_k^{(t+1)} = \omega_k^{(t)} \exp\left(\frac{\lambda}{3K} \left(\rho_k^{(t)} + \frac{\eta}{p_k^{(t)} \sqrt{K}} \right) \right).$$

Fig. 5 The EXP3.P algorithm applied for boosting

The constraint tree is constructed by selecting features randomly for all nodes. The size of the tree depends on how large the decision tree to be used in the base boosting algorithm is intended to be. In practice, the constraints on each node can be selected during the construction of the decision tree. When expanding a particular model, constraint trees that have been already selected for that model are not repeated.

4.2.2 The EXP3.P *algorithm*

The EXP3.P algorithm is an algorithm originally designed for multi-armed bandit problems (Auer et al. 2002b). The use of the algorithm to improve the speed of a boosting algorithm was proposed by Busa-Fekete and Kégl (2010). The algorithm applied for boosting is described in Fig. 5. Note that the description differs somewhat from that of Busa-Fekete and Kégl (2010) in the initialization of the weights $\omega_k^{(1)}$, however the algorithm performs exactly the same way independently of the initial values of the weights as long as they are equal. Moreover, in step (f) of the original algorithm, η is divided by the square root of the maximum iteration number. We absorb this into the smoothing parameter η in order to simplify notation.

There are several variants proposed by Busa-Fekete and Kégl (2010) on how to select the constraint in step (b). In our implementation, the constraint is similar to the one described in Sect. 4.2.1 for the BOOSTINGTREE algorithm, except that features are not selected uniformly at random, but according to the probability distribution $p_k^{(t)}$. If the weak hypotheses

are decision stumps this choice of the constraint is very similar to the multi-armed bandit set-up. For decision trees with several nodes, the choice of the constraint is less trivial. Busa-Fekete and Kégl (2010) discussed several choices, and decided in their empirical evaluations for a similar approach to ours. This is perhaps the simplest approach without having to solve a more complicated credit assignment problem over the involved features.

A further choice has to be made on the reward function $\tilde{f}_t$. When combined with ADABOOST Busa-Fekete and Kégl (2010) suggested the use of a function based on the edge of the weak hypothesis. For the boosting algorithms for ranking discussed in Sect. 4.3, this is a less natural choice. Experimentally we found that the difference $\tilde{f}_t = f_t - f_{t-1}$ is an appropriate choice in our case. In Sect. 4.4 we revisit this issue in a short discussion.

Although we tested several values of the constants λ and η, we have not noticed any strong influence on the performance, except for some extreme values (such as $\lambda = 1$ or $\lambda = 0$). Therefore all reported experiments with EXP3.P are with $\lambda = 0.3$ and $\eta = 0.1$.

4.3 Boosting algorithms

Next we describe the two boosting algorithms used in our ranking problems.

4.3.1 The LAMBDAMART *algorithm*

LAMBDAMART (Wu et al. 2010) is a gradient tree boosting algorithm that approximates the gradient of the ranking measure by combining a pairwise gradient, based on the scores given by the generated model, and the change in the ranking measure when two documents are swapped. Empirically, this approximation was shown to be close to the true gradient in LAMBDARANK (Donmez et al. 2009). LAMBDAMART is one of the most successful ranking algorithm, forming the core of the winning entry at the Yahoo! learning to rank challenge (Burges et al. 2011).

One iteration of the LAMBDAMART algorithm that represents the computation of a weak hypothesis given the current model is shown in Fig. 6. The standard version of the algorithm starts with an initial model (possibly empty as in our implementation), and iteratively adds new weak hypotheses. No constraints are placed on the generation of the regression tree (step (c)), in this variant. When called from the BOOSTINGTREE or the EXP3.P algorithm, the selection of features is constrained in the manner discussed above.

In the experiments, we optimize for *NDCG*, thus, the change in the ranking measure, ΔZ, after swapping documents d_{ij} and $d_{ij'}$ becomes

$$\Delta NDCG_{ijj'} = \frac{1}{n\ maxDCG_i}(\gamma_{r_{ij}} - \gamma_{r_{ij'}})(g_{ij} - g_{ij'}).$$

The combination coefficient α in the algorithm is set to 0.2, which appears to be a good choice for all three benchmarks, although the value was selected only after a few test runs. σ is set to 1, while the number of leaves L varies with the problem, depending mostly on the size of the data, but, as we discuss in Sect. 4.6, also on the complexity of the features.

When analyzing the time complexity of LAMBDAMART, we can split the algorithm into two parts: (1) the computation of λ_{ij} and γ_{ij}, and (2) the training the regression tree. The first part scales linearly with the product of the number of documents and the number of documents per query (the latter term can be smaller if there are only a few relevant documents for each query). The second part scales linearly with the product of the number of documents, the number of leaves, and the number of features considered in each internal node. If the number of leaves is small and, more importantly, the selection in an internal node is constrained to one particular feature, then the first part may dominate the computation time, otherwise the second part can be far more expensive.

Parameters: current model M, training data D, constraint C, ranking measure Z, constants $\sigma, \alpha > 0$, number of leaves L.

(a) For each query i, and each distinct document pairs $d_{ij}, d_{ij'} \in D$, compute the change in the ranking measure by swapping the two documents, $Z_{ijj'}$, and the corresponding approximate pairwise gradients, $\lambda_{ijj'}$:

$$\lambda_{ijj'} = \frac{-\sigma|\Delta Z_{ijj'}|}{1 + e^{\sigma(M(d_{ij}) - M(d_{ij'}))}}.$$

(b) For each query i, and each document $d_{ij} \in D$, compute approximate first and second order gradients, λ_{ij} and γ_{ij}, as follows:

$$\lambda_{ij} = \sum_{j':l_{ij} > l_{ij'}} \lambda_{ijj'} - \sum_{j':l_{ij} < l_{ij'}} \lambda_{ij'j},$$

$$\gamma_{ij} = -\sum_{j':l_{ij} > l_{ij'}} \lambda_{ijj'}\big(\sigma|\Delta Z_{ijj'}| + \lambda_{ijj'}\big) - \sum_{j':l_{ij} < l_{ij'}} \lambda_{ij'j}\big(\sigma|\Delta Z_{ij'j}| + \lambda_{ij'j}\big).$$

(c) Train an L leaf regression tree ψ on $\{d_{ij}, \lambda_{ij}\}$, subject to constraint C.

(d) Create regression tree ψ', by replacing each lth leaf values of ψ, with $\sum_{d_{ij} \in \psi_l} \lambda_{ij} / \sum_{d_{ij} \in \psi_l} \gamma_{ij}$, where the summations are over the documents that fall into the lth leaf.

(e) Return weak hypothesis $\alpha\psi'$.

Fig. 6 Computing the weak hypothesis in the LAMBDAMART algorithm

4.3.2 *The* NDCGBOOST *algorithm*

NDCGBOOST (Valizadegan et al. 2009) is a boosting algorithm that optimizes the expectation of NDCG over all possible permutations of documents. The computation of a weak hypothesis given a current model in the NDCGBOOST algorithm is provided in Fig. 7. The binary classifier used in step (b) of the algorithm is a decision tree with L leaves, where the number L varies with the problem (similarly as in the case of LAMBDAMART). The constraint imposed on the tree building algorithm is also similar to the constraint imposed in LAMBDAMART, with the standard NDCGBOOST iteratively adding unconstrained weak hypotheses, while the BOOSTINGTREE and the EXP3.P algorithms constrain every added tree by a feature tree. Although we are not aware of any strong performance obtained in any ranking challenge with NDCGBOOST, the main reason it is included in our experiments is that it appears to us as a highly performing ranking algorithm, which is also illustrated in Sects. 4.4 and 4.5.

When analyzing the time complexity of LAMBDAMART, we can split the algorithm into three parts: (1) the computation of γ_{ij}, (2) the training of the decision tree (assumed to be the classifier), and (3) the computation of the combination weight. The first and third parts scale linearly with the product of the number of documents and the number of documents per query (and as for LAMBDAMART, the latter term can be smaller if there are only a few relevant documents for each query). The second part scales linearly with the product of the number of documents, the number of leaves, and the number of features considered in each

Parameters: current model M, training data D, constraint C.

(a) For each query i, and each document pair $d_{ij}, d_{ij'} \in D_i$, $j \neq j'$, compute $\theta_{ijj'}$, and the weight γ_{ij} for each document $d_j \in D_j$, as follows:

$$\theta_{ijj'} = \frac{e^{M(d_{ij})-M(d_{ij'})}}{(1+e^{M(d_{ij})-M(d_{ij'})})^2},$$

$$\gamma_{ij} = \sum_{j'=1:j'\neq j}^{m_i} \frac{2^{l_{ij}} - 2^{l_{ij'}}}{maxDCG_i} \theta_{ijj'}.$$

(b) Train a binary classifier ψ that maximizes $\sum_{i=1}^{n} \sum_{j=1}^{m_i} \gamma_{ij} \psi(d_{ij})$, subject to constraint C.

(c) Compute the combination weight α:

$$\alpha = \frac{1}{2} \log_2 \left(\frac{\sum_{i=1}^{n} \sum_{j,j':\psi(d_{ij})<\psi(d_{ij'})} \theta_{ijj'} \frac{2^{l_{ij}}-1}{maxDCG_i}}{\sum_{i=1}^{n} \sum_{j,j':\psi(d_{ij})>\psi(d_{ij'})} \theta_{ijj'} \frac{2^{l_{ij}}-1}{maxDCG_i}} \right).$$

(d) Return weak hypothesis $\alpha\psi$.

Fig. 7 Computing the weak hypothesis in the NDCGBOOST algorithm

internal node. Similarly to LAMBDAMART, the first and third parts take a significant time compared to the second part only if the number of leaves is small and in the presence of a strong constraint (when the optimization in the training of the decision tree is rather limited, e.g., when the tree is constrained by a feature tree).

4.4 The LETOR benchmark

The LETOR benchmark collection[3] is perhaps the most frequently used benchmark dataset for learning to rank. One of the most significant feature of LETOR is that the results of several reference algorithms are available for the datasets included. For our experiments we selected the largest dataset, MQ2007, from the Letor 4.0 collection, which is 46 dimensional, with 1692 queries and 69,623 documents (thus, approximately 41 documents/query in average), and is split in five folds. The relevance labels are ranging from 2 (most relevant) to 0 (irrelevant). On this dataset the RANKBOOST (Freund et al. 1998) algorithm has roughly the best results out of the algorithms enlisted on the website associated with the benchmark collection, and therefore, in the following we also show the performance of RANKBOOST as a baseline (note that the standard performance measure for this dataset is *MeanNDCG*).

On this dataset, for both LAMBDAMART and NDCGBOOST small decision trees with only 2 leaves appeared to be a good choice for the weak learners, thus, in fact, we use stumps. For learning with EXP3.P accelerated boosting, the reward function for the EXP3.P algorithm has to be defined. In the experiments we used the change in *NDCG* on the training set (after adding a new weak hypothesis) as the reward. Using changes in *MeanNDCG* gave

[3] http://research.microsoft.com/en-us/um/beijing/projects/letor/.

Table 3 *MeanNDCG* on the MQ2007 dataset from the Letor 4.0 benchmark collection for the five folds, and their average. The RANKBOOST results are provided as baseline on the LETOR website. For each fold, the best model for the standard variant of the LAMBDAMART and the NDCGBOOST algorithms is selected based on the *MeanNDCG* on the validation set in 1,000 iterations (i.e., ensembles of at most 1,000 hypotheses), and in the table the *MeanNDCG* of the selected model on the corresponding test set is reported. Similarly, the *MeanNDCG* on the test sets for the EXP3.P variants correspond to the ensembles with the best validation *MeanNDCG* in 5,000 iterations, while for BOOSTINGTREE it corresponds to the best validation *MeanNDCG* of the first 5,000 hypothesis sequences

	Fold1	Fold2	Fold3	Fold4	Fold5	average
RANKBOOST	0.5267	0.4913	0.5118	0.4687	0.5030	0.5003
LAMBDAMART	0.5380	0.4832	0.5248	0.4781	0.5060	0.5060
NDCGBOOST	0.5356	0.4852	0.5193	0.4765	0.5094	0.5052
EXP3.P(LAMBDAMART)	0.5247	0.4801	0.5120	0.4691	0.5025	0.4977
EXP3.P(NDCGBOOST)	0.5426	0.4918	0.5123	0.4700	0.5001	0.5033
BOOSTINGTREE(LAMBDAMART)	0.5400	0.4837	0.5257	0.4770	0.5060	0.5065
BOOSTINGTREE(NDCGBOOST)	0.5383	0.4897	0.5202	0.4785	0.5089	0.5071

similar results, while using quantities related to the training of the weak classifiers as rewards fared worse. Finally, BOOSTINGTREE optimizes for *MeanNDCG* on the training set.

For this problem there is only one decision stump for each weak hypothesis, and the number of documents for each query is in the same order as the dimension of the data, while a quarter of the documents are relevant, therefore, for both boosting algorithms the time required for building the tree does not dominate outright the running time of the other steps of the respective boosting algorithm. In our implementation, both EXP3.P and BOOSTINGTREE were approximately five to ten times faster per iteration than the standard variants.[4] Consequently, we run these two algorithms for five times more iterations then the standard variants.

The *MeanNDCG* performance on the five folds for the six algorithms are provided in Table 3, in addition to the RANKBOOST baseline, while the *NDCG@k* performance are shown in Fig. 8. Overall, the standard variants of both LAMBDAMART and NDCGBOOST performed better than RANKBOOST, while the difference between the two standard variants is negligible. EXP3.P performs poorly when combined with LAMBDAMART and somewhat worse than the standard variant when combined with NDCGBOOST. BOOSTINGTREE performs slightly better than the corresponding standard algorithms, with a more pronounced improvement over NDCGBOOST, which might be related to the fact that EXP3.P is also more successful in this combination (on this dataset, randomized moves appear more successful with NDCGBOOST).

The *MeanNDCG* performance of the six algorithms is shown in Fig. 9 for varying limitations on the length of the boosting sequence. Note that the length of the boosting sequence is equal to the number of (boosting) iterations for the regular variant and the EXP3.P algorithm, but it differs from the number of iterations (or rounds) in the case of the BOOSTINGTREE algorithm. We are interested in the length of the sequence more, since we aim for

[4]We discussed, so far, only the time complexity of the various components of the boosting algorithms. Selecting the models to expand in a round of the BOOSTINGTREE algorithm (Fig. 3, step (a)) takes more time than computing the probabilities in the EXP3.P algorithm (Fig. 5, step (f) and (a)), however, for all datasets described in the paper, these take still considerably less time than the reweighting of the data in either of the two boosting algorithms, that is, in LAMBDAMART (Fig. 6, step (b)) and in NDCGBOOST (Fig. 7, step (a)).

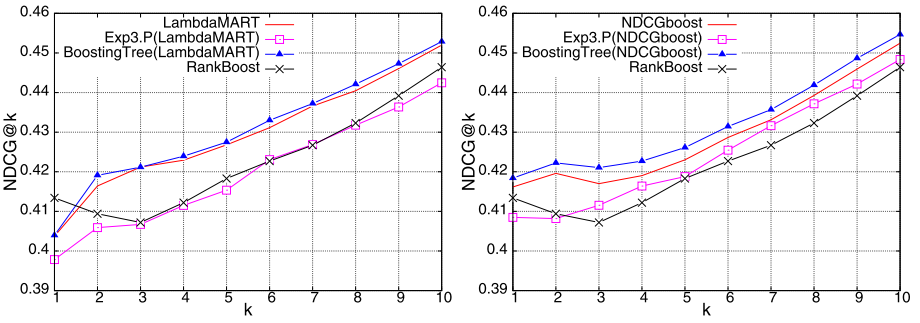

Fig. 8 *NDCG@k* performances for varying *k* on the LETOR data set. The *NDCG@k* test results are averaged over the five folds, and correspond to the ensembles with the best *MeanNDCG* on the validation sets. As in Table 3, the standard variant of LAMBDAMART and NDCGBOOST are run for 1,000 iterations, while EXP3.P and BOOSTINGTREE for 5,000

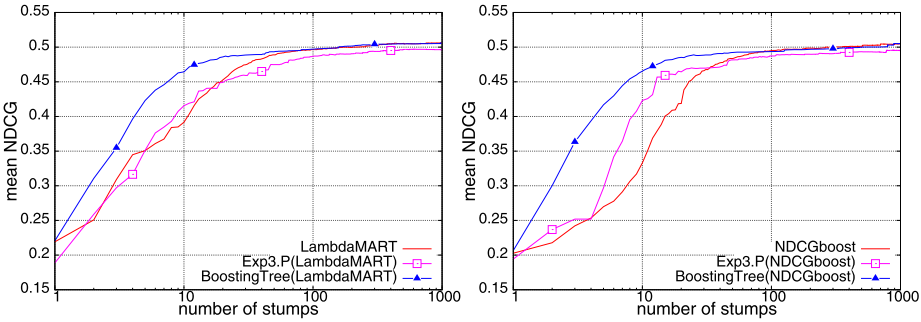

Fig. 9 Average *MeanNDCG* on the test sets of the five folds of the LETOR benchmark. The *MeanNDCG* test results correspond to the ensembles with varying maximum number of stumps that have the best validation *MeanNDCG*

shorter (i.e., low-complexity) models with good performances. The asymptotic performance of the three variants (standard, EXP3.P and BOOSTINGTREE) are quite close to each other, but there is a clear difference when less than 100 trees are included in the final model. Interestingly, EXP3.P performs on par with the standard variant, when combined with LAMBDAMART, and even better, when combined with NDCGBOOST. BOOSTINGTREE has a clear advantage in the early phase over both variants with more pronounced advantage combined with NDCGBOOST.

4.5 Yahoo! ranking challenge

The dataset used in this section was used in the Yahoo! Learning to Rank Challenge (Chapelle and Chang 2011). The set is 519 dimensional, with 29,921 queries, and 709,877 documents (thus, approximately 29 documents/query in average). The relevance labels range from 4 (most relevant) to 0 (irrelevant). The winning entry in the challenge was a combination of models resulting from the LAMBDAMART and the LAMBDARANK algorithms (Burges et al. 2011) (the evaluation metric in the challenge was *NDCG@10*).

On this dataset, a good choice of the number of leaves of the decision trees for LAMBDAMART appears to be 16, while for NDCGBOOST it is in the range between 16 and 64

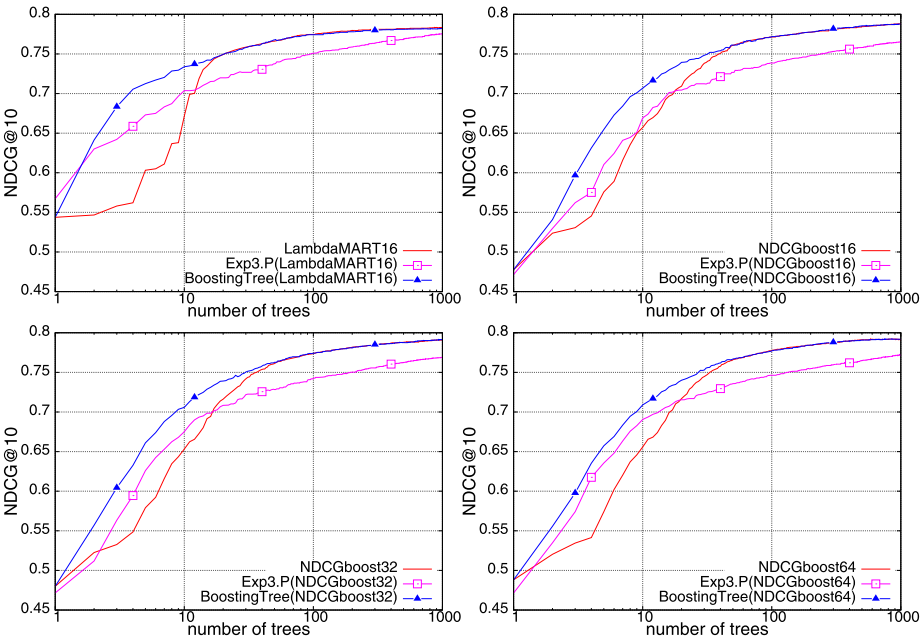

Fig. 10 *NDCG*@10 results on the Yahoo! benchmark for LAMBDAMART with 16 leaves (*top-left*), NDCGBOOST with 16 leaves (*top-right*), NDCGBOOST with 32 leaves (*bottom-left*), and NDCGBOOST with 64 leaves (*bottom-right*). The *NDCG*@10 test results correspond to the ensembles with varying maximum number of decision trees that have the best validation *NDCG*@10. BOOSTINGTREE was run for 5,000 iterations (i.e., until 5,000 sequences were generated)

(with a large number of iterations it appeared to be 32). In EXP3.P, we use the change in *NDCG* on the training set as reward, while BOOSTINGTREE optimizes for *NDCG* on the training set.

Given that the number of leaves is somewhat larger for this problem (compared to the LETOR benchmark), and the dimension of the data is 20 times larger than the number of documents per query, the EXP3.P and the BOOSTINGTREE algorithms are approximately 500 times faster per iteration than the regular variants (with 32 leaves). By parallelizing the standard variants (by distributing the features on different processors), we managed to bring down their running time significantly and as for the LETOR benchmark we run the BOOSTINGTREE algorithm only for five times more iterations than the standard variants.

The *NDCG*@10 performance of the six algorithms is shown in Fig. 10 for varying limitations on the length of the boosting sequence. We observe that, asymptotically, BOOSTINGTREE achieves similar performance as the standard variant, for both LAMBDAMART and NDCGBOOST, while EXP3.P has a relatively weak performance even when a large number of trees are included. We expect EXP3.P to converge eventually to the performance of the standard variant, but that may require a considerably larger number of trees. It is interesting that the performance drop of EXP3.P is not so much related to the number of leaves, and varies more with the boosting algorithm (it performs considerably better with LAMBDAMART than with NDCGBOOST). When the final model is limited to a small number of trees BOOSTINGTREE has clearly an edge, and the difference to the standard version correlates with the performance of EXP3.P.

4.6 Move ordering in chess

The efficiency of search algorithms in games heavily depends on the order in which the moves are examined. Although the strong chess programs are unlikely to benefit from move ordering generated by boosting algorithms, we include chess datasets in the experiments, since chess records are easily accessible, and some properties specific to games are highlighted by these sets as well. We expect that, for instance, using such move ordering in the Monte-Carlo simulations in Go (Gelly and Silver 2008), or in RTS games can offer more improvement (than it would do in chess).

Two datasets were constructed including middle-game positions originating either from the opening line B84 or E97 (Matanović et al. 1971). The first, B84 (Classical Scheveningen variation of Sicilian Defence), results in more open positions, while the second, E97 (Aronin-Taimanov variation of King's Indian), leads usually to positions with a closed centre. Each entry of the datasets describes a position-move pair, where the position describes the locations of all pieces on the board. The goal of the ranking algorithm is to provide partial ordering for moves for the same position. Thus, to cast the problem in our previous document-query framework, the positions correspond to queries, and moves correspond to documents. Similarly, each position-move pair is described by a feature vector: an integer value is reserved for each square of the board (representing the piece occupying the location), describing the position, and further seven features encode the move (including: the moving piece, the file and rank, i.e., the coordinates, of the origin and the destination of the move, if the move is a capture, captured piece if any). Thus, the datasets are 71 dimensional. The B84 dataset includes 2,000 positions and 85,380 moves (thus, a branching factor of around 42), while the E97 set includes 3,000 positions and 109,341 moves (branching factor of around 36). The lower branching factor of the second set is due to the closeness of the opening line.

For each position of the datasets all legal moves were included, and their score was computed by a 1 second search using the game program CRAFTY (Hyatt and Newborn 1997). The scores are converted to ranking labels as follows: the move with the best score, or with a score not worse than that by a tenth of a pawn (the best moves) have their relevance labeled with the value 4, the moves scored lower but not by more than a fifth of a pawn are labeled 3, moves with scores lower by less than three-tenth of a pawn are labeled 2, moves with scores lower by less than half a pawn are labeled 1, and the rest are labeled 0. The latter group includes moves that are likely to be poor, so they should not be considered for investigation.

On this dataset both LAMBDAMART and NDCGBOOST needed regression, and, respectively, decision trees with up to 32 leaves to achieve a reasonable performance. Although the dimensionality of the data is not considerably higher than that of the LETOR benchmark set, the features representing the positions and moves are very 'raw' for this set, and therefore it is not surprising that a deeper representation is needed to combine the features. In EXP3.P, we use the change in *NDCG* on the training set as reward, while BOOSTINGTREE optimizes for *NDCG* on the training set.

For these two datasets the number of features is not much higher than the branching factor (a quarter of the moves being relevant), but the number of leaves is significantly higher than in the case of the LETOR benchmark, and therefore, the regular variants are again significantly slower per iteration (approximately 60 times slower). As for the Yahoo! benchmark, we parallelized the regular variants, and run the BOOSTINGTREE algorithm for five times more iterations.

The *NDCG*@10 performance of the six algorithms on the dataset B84 and E97 is shown in Fig. 11 for varying limitation on the length of the boosting sequence. It is striking how

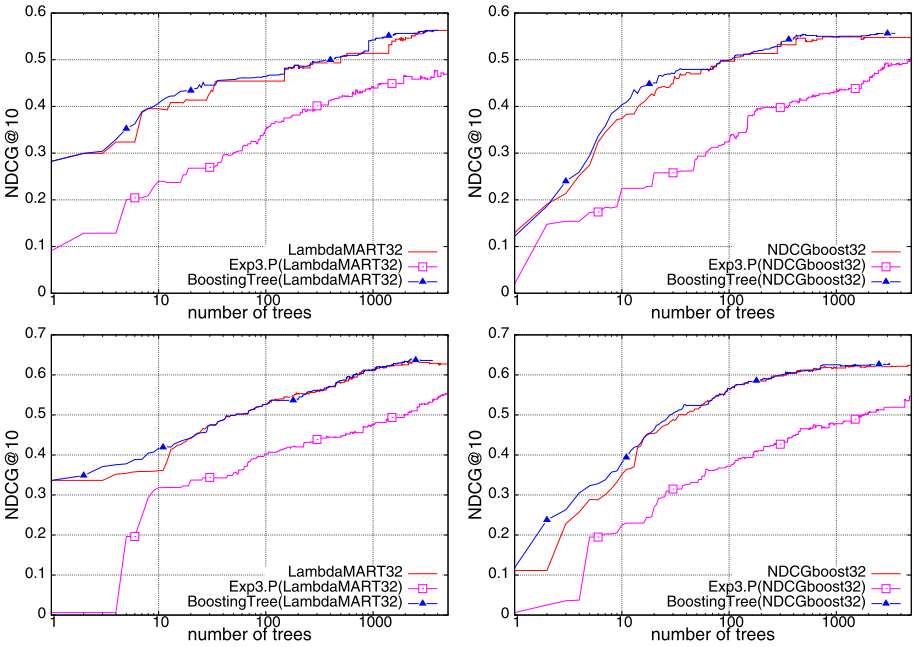

Fig. 11 *NDCG*@10 results on the B84 (*top*) and E97 (*bottom*) chess benchmark sets. The *NDCG*@10 test results correspond to the ensembles with varying maximum number of decision trees that have the best validation *NDCG*@10. BOOSTINGTREE was run for 25,000 iterations (i.e., until 25,000 sequences were generated)

poor the performance of EXP3.P is compared to the standard variant. And the weak performance is most stringent when only a small number of trees are included in the final model. The performance of EXP3.P is catching up slowly, but it is clear that finding a good combination of features (almost) randomly is slow. For the two previous benchmarks, with more complex features, this did not appear to be a problem. The performance of BOOSTINGTREE (relative to the standard boosting algorithms) is also weaker compared to the previous benchmarks, but it is clear that by attempting to find alternatives to the randomly chosen feature sequences, the algorithm eventually converges to better choices, and to performances that are comparable to that of the standard boosting algorithms.

4.7 Practical growth rate in the BOOSTINGTREE algorithm

György and Kocsis (2011) provided a bound that the number of algorithm instances in METAMAX grows with at least $\Omega(\sqrt{t_r})$, while showing that in practice it grows at a rate of $\Omega(t_r/\ln t_r)$. Conversely, this implies that the length of a METAMAX round in practice grows at a rate of $\Omega(\ln t_r)$ instead of $\Omega(\sqrt{t_r})$.

We revisit this problem here by plotting the length of a round in BOOSTINGTREE for various ranking benchmark sets (Fig. 12, left). We observe that the number of sequences grows at a slightly larger rate, but it appears to be still $\Omega(\ln t_r)$, rather than $\Omega(\sqrt{t_r})$, although there appears to be a slight increase compared to METAMAX. Note, however, that in BOOSTINGTREE the number of sequences is increased by 1 every time a sequence is extended (although a model is not considered anymore once all of its possible children nodes

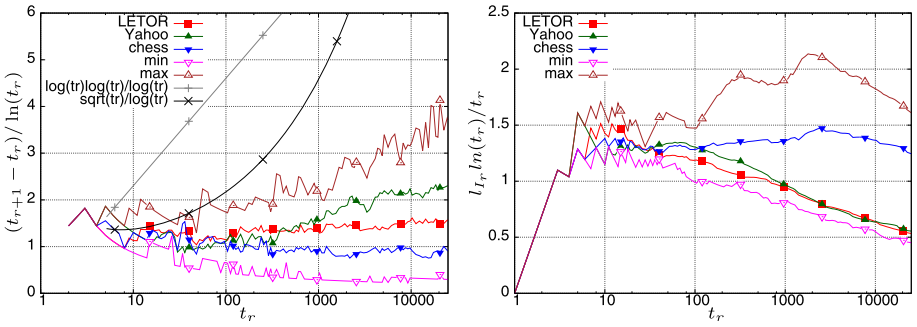

Fig. 12 The length of a round (*left*) and the number of hypothesis in the best model (*right*) for varying number of generated hypothesis sequences (t_r). The $LETOR$ curve averages over runs on the five folds of LETOR benchmark set when BOOSTINGTREE was combined with the LAMBDAMART and NDCGBOOST algorithms with decision stumps as weak hypotheses. The *Yahoo* curve averages over runs on the Yahoo! dataset including results when BOOSTINGTREE was applied with the LAMBDAMART algorithm with 16-leaf decision trees as weak hypotheses, and the NDCGBOOST algorithm with 16-, 32-, and 64-leaf decision trees. The *chess* curve averages over runs on the B84 and the E97 move ordering datasets when the LAMBDAMART and the NDCGBOOST algorithms (with 32-leaf decision trees) where used as the base algorithm in BOOSTINGTREE. The minimum (min) and the maximum (max) of the curves over all above runs is also included. The curve for $\sqrt{t_r}$ (corresponding to the theoretical bound) is added for comparison to the *left figure*

are generated, which rarely happens in practice), while in METAMAX a new algorithm instance is added in every round. The increase in the round length is most likely due to this increase in the number of hypothesis sequences (compared to the number of algorithm instances in METAMAX), which is most prominent when the leading sequence is forced to be the longest (step (c) of the algorithm), while in the similar step of the METAMAX algorithm no new algorithm instances are added.

In Theorem 2 we provided a bound of order $\Omega(\sqrt{t_r})$ on the number of weak hypotheses l_{I_r} combined in the best model in a round r. In practice, the best model seems to grow at a higher rate, $\Omega(t_r/\ln(t_r))$, as shown in Fig. 12, right.

4.8 Discussion

In the experiments, we have used three benchmark sets, a smaller and a larger in the web search domain, and a medium size in move ordering in chess. Other than the size and the domain, there was one more difference between the sets, notably the complexity of the features (with the chess dataset having rather raw features). Not surprisingly, for larger data more complex models are suitable (decision trees with more leaves), moreover, raw features need to be enhanced by multiple combination levels (again, more leaves in the decision trees). The performance of boosting algorithms accelerated by bandit algorithms appears to be less sensitive to the size of the data (and as such, in some respect, to the size of the decision tree), however, their performance appears to degrade significantly when individual features are too raw to improve the model, and the right combination of features are to be found instead. The behavior of this algorithm seems to depend on how good weak hypotheses can be constructed from a randomly selected constrained tree, since even for regression/decision trees of moderate size the number of constraints becomes huge, and therefore the bandit algorithm essentially just samples uniformly from the constraints. BOOSTINGTREE relates to bandit algorithms in the sense that it performs better when bandit algorithms perform

well, while mitigating the problem of raw features by revisiting the selection choices on the feature set. For the web search datasets the BOOSTINGTREE algorithm manages to build better performing models when the number of hypotheses has to be limited, while achieving similar performance as the base boosting algorithm when the number of hypotheses is unbounded.

5 Conclusions

In this paper we provided a tree based strategy, BOOSTINGTREE, that builds several sequences of weak hypotheses in parallel, and extends the ones that are likely to yield a good model. The new strategy aims to improve the training speed by constraining the weak learners, and simultaneously, it generates small models that have better performance than the ones obtained by the standard variant of the boosting algorithm with early stopping. Theoretical analysis shows that the BOOSTINGTREE algorithm is consistent. Moreover, at least $\Omega(\sqrt{t_r})$ of the weak hypotheses are allocated to the leading model, which in practice seems to be even closer to $\Omega(t_r/\ln t_r)$ (where t_r is the number of weak hypotheses used in building the tree up to the end of the rth round of the algorithm). Experimental results are provided for two standard learning to rank benchmarks, and one additional benchmark constructed for move ordering in chess. On these benchmarks we conclude that the new algorithm performs favorably in comparison to the standard boosting algorithms, and to some randomized variants that use the theory of multi-armed bandits to speed up the learning process. At early stages of the training process, BoostingTree provides more accurate solutions, while later the resulting models are computationally less expensive with similar (ranking) performance.

There are several open questions left in the theoretical analysis, in particular on the tightness of the bound on the number of weak hypotheses allocated to the leading model. We expect that for certain boosting algorithms and well-designed constraints on the weak learner, the rate at which the BOOSTINGTREE algorithm converges to the 'optimal' model can be bounded. Comparing and combining the algorithm with pruning and l_1 regularization techniques applied for boosting are also to be investigated, as well as the performance of the algorithm for more general machine learning tasks. Furthermore, smart, adaptive choices of the constraints used in extending existing models in BOOSTINGTREE would also be of natural interest.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped to improve the presentation of the paper. This research was supported in part by the Hungarian Scientific Research Fund and the Hungarian National Office for Research and Technology (OTKA-NKTH CNK 77782 and OTKA NK 105645), by the PASCAL2 Network of Excellence (EC grant no. 216886), by the Alberta Innovates Technology Futures and by the Natural Sciences and Engineering Research Council of Canada. The work was also carried out in part under the EITKIC_12-1-2012-0001 project, supported by the Hungarian Government, managed by the National Development Agency, financed by the Research and Technology Innovation Fund, and was performed in cooperation with the EIT ICT Labs Budapest Associate Partner Group (www.ictlabs.elte.hu).

References

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002a). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*(2–3), 235–256.

Auer, P., Cesa-Bianchi, N., Freund, Y., & Schapire, R. (2002b). The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, *32*, 48–77.

Burges, C. J. C., Svore, K. M., Bennett, P. N., Pastusiak, A., & Wu, Q. (2011). Learning to rank using an ensemble of lambda-gradient models. *Journal of Machine Learning Research*, *14*, 25–35.

Busa-Fekete, R., & Kégl, B. (2009). Accelerating AdaBoost using UCB. *Journal of Machine Learning Research Workshop and Conference Proceedings*, *7*, 111–122.

Busa-Fekete, R., & Kégl, B. (2010). Fast boosting using adversarial bandits. In J. Fürnkranz & T. Joachims (Eds.), *Proceedings of the 27th international conference on machine learning (ICML-10)* (pp. 143–150).

Cazenave, T. (2009). Nested Monte-Carlo search. In C. Boutilier (Ed.), *Proceedings of the 21st international joint conference on artificial intelligence (IJCAI)* (pp. 456–461).

Chapelle, O., & Chang, Y. (2011). Yahoo! Learning to rank challenge overview. *Journal of Machine Learning Research Workshop and Conference Proceedings*, *14*, 1–24.

Donmez, P., Svore, K. M., & Burges, C. J. (2009). On the local optimality of lambdarank. In *Proceedings of the 32nd international ACM SIGIR conference on research and development in information retrieval (SIGIR)* (pp. 460–467).

Escudero, G., Màrquez, L., & Rigau, G. (2000). Boosting applied toe word sense disambiguation. In R. L. de Mántaras & E. Plaza (Eds.), *Lecture notes in computer science: Vol. 1810. 11th European conference on machine learning (ECML)* (pp. 129–141). Berlin: Springer.

Freund, Y., & Mason, L. (1999). The alternating decision tree algorithm. In *Proceedings of the 16th international conference on machine learning* (pp. 124–133).

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, *55*(1), 119–139.

Freund, Y., Iyer, R. D., Schapire, R. E., & Singer, Y. (1998). An efficient boosting algorithm for combining preferences. In J. W. Shavlik (Ed.), *Proceedings of the 15th international conference on machine learning (ICML)* (pp. 170–178). San Mateo: Morgan Kaufmann.

Gelly, S., & Silver, D. (2008). Achieving master level play in 9×9 computer go. In D. Fox & C. P. Gomes (Eds.), *Proceedings of the 23rd AAAI conference on artificial intelligence* (pp. 1537–1540). Menlo Park: AAAI Press.

György, A., & Kocsis, L. (2011). Efficient multi-start strategies for local search algorithms. *The Journal of Artificial Intelligence Research*, *41*, 407–444.

Hyatt, R. M., & Newborn, M. (1997). CRAFTY goes deep. *ICCA Journal*, *20*(2), 79–86.

Järvelin, K., & Kekäläinen, J. (2000). IR evaluation methods for retrieving highly relevant documents. In *Proceedings of the 23rd international ACM SIGIR conference on research and development in information retrieval (SIGIR)* (pp. 41–48).

Jones, D. R., Perttunen, C. D., & Stuckman, B. E. (1993). Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, *79*(1), 157–181.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, & M. Spiliopoulou (Eds.), *Proceedings of the 17th European conference on machine learning (ECML)* (pp. 282–293).

Luby, M., Sinclair, A., & Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, *47*, 173–180.

Margineantu, D. D., & Dietterich, T. G. (1997). Pruning adaptive boosting. In D. H. Fisher (Ed.), *Proceedings of the 14th international conference on machine learning (ICML 1997)* (pp. 211–218). San Mateo: Morgan Kaufmann.

Martínez-Muñoz, G., Hernández-Lobato, D., & Suárez, A. (2009). An analysis of ensemble pruning techniques based on ordered aggregation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *31*(2), 245–259.

Matanović, A., Molorović, M., & Božić, A. (1971). *Classification of chess openings*. Beograd: Chess Informant.

Munos, R. (2011). Optimistic optimization of deterministic functions without the knowledge of its smoothness. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 24* (pp. 783–791).

Schapire, R. E. (2002). The boosting approach to machine learning: an overview. In D. D. Denison, M. H. Hansen, C. Holmes, B. Mallick, & B. Yu (Eds.), *Nonlinear estimation and classification*. Berlin: Springer.

Streeter, M. J., & Smith, S. F. (2006). A simple distribution-free approach to the max k-armed bandit problem. In *Proceedings of the 12th international conference on principles and practice of constraint programming (CP2006)*, Nantes, France, September 25–29, 2006 (pp. 560–574).

Tamon, C., & Xiang, J. (2000). On the boosting pruning problem. In R. L. de Mántaras & E. Plaza (Eds.), *Proceedings of the 11th European conference on machine learning (ECML)* (pp. 404–412). Barcelona.

Tsoumakas, G., Partalas, I., & Vlahavas, I. P. (2009). An ensemble pruning primer. In O. Okun & G. Valentini (Eds.), *Studies in computational intelligence: Vol. 245. Applications of supervised and unsupervised ensemble methods* (pp. 1–13). Berlin: Springer.

Valizadegan, H., Jin, R., Zhang, R., & Mao, J. (2009). Learning to rank by optimizing ndcg measure. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, & A. Culotta (Eds.), *Advances in neural information processing systems 22* (pp. 1883–1891).

Wu, Q., Burges, C. J. C., Svore, K. M., & Gao, J. (2010). Adapting boosting for information retrieval measures. *Information Retrieval*, *13*(3), 254–270.

Xi, Y. T., Xiang, Z. J., Ramadge, P. J., & Schapire, R. E. (2009). Speed and sparsity of regularized boosting. *Journal of Machine Learning Research Workshop and Conference Proceedings*, *5*, 615–622.

Xiang, Z. J., & Ramadge, P. J. (2009). Sparse boosting. In *Proceedings of the 34th IEEE international conference on acoustics, speech, and signal processing (ICASSP)* (pp. 1625–1628).