QuickScorer: a Fast Algorithm to Rank Documents with Additive Ensembles of Regression Trees

Claudio Lucchese
ISTI–CNR, Pisa, Italy
c.lucchese@isti.cnr.it

Franco Maria Nardini
ISTI–CNR, Pisa, Italy
f.nardini@isti.cnr.it

Salvatore Orlando
Univ. di Venezia, Italy
orlando@unive.it

Raffaele Perego
ISTI–CNR, Pisa, Italy
r.perego@isti.cnr.it

Nicola Tonellotto
ISTI–CNR, Pisa, Italy
n.tonellotto@isti.cnr.it

Rossano Venturini
Univ. di Pisa, Italy
rossano@di.unipi.it

ABSTRACT

Learning-to-Rank models based on additive ensembles of regression trees have proven to be very effective for ranking query results returned by Web search engines, a scenario where quality and efficiency requirements are very demanding. Unfortunately, the computational cost of these ranking models is high. Thus, several works already proposed solutions aiming at improving the efficiency of the scoring process by dealing with features and peculiarities of modern CPUs and memory hierarchies. In this paper, we present QuickScorer, a new algorithm that adopts a novel bitvector representation of the tree-based ranking model, and performs an interleaved traversal of the ensemble by means of simple logical bitwise operations. The performance of the proposed algorithm are unprecedented, due to its cache-aware approach, both in terms of data layout and access patterns, and to a control flow that entails very low branch mis-prediction rates. The experiments on real Learning-to-Rank datasets show that QuickScorer is able to achieve speedups over the best state-of-the-art baseline ranging from 2x to 6.5x.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

Keywords

Learning to Rank, Efficiency, Cache-aware Algorithms

1. INTRODUCTION

Ranking query results according to a relevance criterion is a fundamental problem in Information Retrieval (IR). Nowadays, an emerging research area named Learning-to-Rank (LtR) [7, 2] has shown that effective solutions to the ranking problem can leverage machine learning techniques. A LtR-based function, which *scores* a set of candidate documents according to their relevance to a given user query, is learned from a *ground-truth* composed of many training examples. The examples are basically a collection of queries $\mathcal{Q}$, where each query $q \in \mathcal{Q}$ is associated with a set of assessed documents $\mathcal{D} = \{d_0, d_1, \ldots\}$. Each pair (q, d_i) is in turn labeled by a *relevance judgment* y_i, usually a positive integer in a fixed range, stating the degree of relevance of the document for the query. These labels induce a partial ordering over the assessed documents, thus defining their *ideal ranking* [6]. The scoring function learned by a LtR algorithm aims to approximate the *ideal ranking* from the examples observed in the training set.

The ranking process is particularly challenging for Web search engines, which, besides the demanding requirements for result pages of high quality in response to user queries, have also to deal with efficiency constraints, which are not so common in other ranking-based applications. Indeed, two of the most effective LtR-based rankers are based on additive ensembles of regression trees, namely Gradient-Boosted Regression Trees (GBRT) [4], and Lambda-MART (λ-MART) [18]. Due to the thousands of trees to be traversed at scoring time for each document, these rankers are also the most expensive in terms of computational time, thus impacting on response time and throughput of query processing. Therefore, devising techniques and strategies to speed-up document ranking without losing in quality is definitely an urgent research topic in Web search [14, 3, 10, 5, 19].

Usually, LtR-based scorers are embedded in complex two-stage ranking architectures [3, 16], which avoid applying them to all the documents possibly matching a user query. The first stage retrieves from the inverted index a sufficiently large set of possibly relevant documents matching the user query. This phase is aimed at optimizing the recall and is usually carried out by using a simple and fast ranking function, e.g., BM25 combined with some document-level scores [9]. LtR-based scorers are used in the second stage to *re-rank* the candidate documents coming from the first stage, and are optimized for high precision. In this two-stage architecture, the time budget available to re-rank the candidate documents is limited, due to the incoming rate of queries and the users' expectations in terms of quality-of-service. Strongly motivated by time budget considerations, the IR community has started to investigate low-level optimizations to reduce the scoring time of the most effective LtR

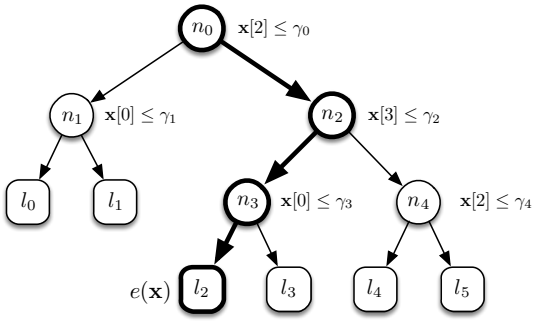


Figure 1: A decision tree.

rankers based on ensembles of regression trees, by dealing with features and peculiarities of modern CPUs and memory hierarchies [1, 12]. In this work we advance the state of the art in this field, and propose QuickScorer (QS), a new algorithm to score documents with an ensemble of regression trees. The main contributions of our proposal are:

- a novel representation of an ensemble of binary regression trees based on bitvectors, allowing QS to perform a fast interleaved traversal of the trees by using efficient logical bitwise operations. The performance benefits of the resulting traversal are unprecedented, due to a cache-aware approach, both in terms of data layout and access patterns, and to a program control flow that entails very low branch mis-prediction rates;
- an extensive experimental assessment conducted on publicly available LtR datasets with various λ-MART models, differing for both the size of the ensemble and the number of tree leaves. The results of the experiments show that QS achieves impressive speedups over the best state-of-the-art competitor, ranging from 2x up to 6.5x. Moreover, to motivate the very good performance of QS over competitors, we evaluate in-depth some CPU counters that measure important performance events, such as number of instructions executed, cache-misses suffered, or branches mis-predicted;
- a block-wise version of QS for scoring large tree ensembles and large sets of documents. BlockWise-QS (BWQS) splits the set of documents and the tree ensemble in disjoint groups that can be processed separately. Our experiments show that BWQS performs up to 1.55 times better than the original QS, thanks to cache reuse which reduces cache misses.

The rest of the paper is structured as follows: Section 2 provides background information and discusses the related work, while Section 3 details the QS algorithm and its features. Then, Section 4 reports on the results of our comprehensive evaluation. Finally, we conclude our investigation in Section 5 by reporting some conclusions and suggestions for future research.

2. BACKGROUND AND RELATED WORK

Gradient-Boosted Regression Trees (GBRT) [4] and Lambda-MART (λ-MART) [18] are two of the most effective Learning-to-Rank (LtR) algorithms. They both generate additive ensembles of regression trees aiming at predicting the relevance labels y_i of a query document pair (q, d_i). The GBRT algorithm builds a model by approximating the root mean squared error on a given training set. This loss function makes GBRT a point-wise LtR algorithm, i.e., query-document pairs are exploited independently. The λ-MART algorithm improves over GBRT by directly optimizing list-wise information retrieval measures such as NDCG [6]. Thus, λ-MART aims at finding a scoring function that generates an ordering of documents as close as possible to the ideal ranking. In terms of scoring process there is thus no difference between λ-MART and GBRT, since they both generate a set of weighted regression trees.

In this work, we discuss algorithms and optimizations for scoring efficiently documents by means of regression tree ensembles. Indeed, the findings of this work apply beyond LtR, and in any application where large ensembles of regression trees are used for classification or regression tasks.

Each query-document pair (q, d_i) is represented by a real-valued vector $\mathbf{x}$ of *features*, namely $\mathbf{x} \in \mathbb{R}^{|\mathcal{F}|}$ where $\mathcal{F} = \{f_0, f_1, \ldots\}$ is the set of features characterizing the candidate document d_i and the user query q, and $\mathbf{x}[i]$ stores feature f_i. Let $\mathcal{T}$ be an ensemble of trees representing the ranking model. Each tree $T = (N, L)$ in $\mathcal{T}$ is a decision tree composed of a set of internal nodes $N = \{n_0, n_1, \ldots\}$, and a set of leaves $L = \{l_0, l_1, \ldots\}$. Each $n \in N$ is associated with a *Boolean test* over a specific feature with id ϕ, i.e., $f_\phi \in \mathcal{F}$, and a constant threshold $\gamma \in \mathbb{R}$. This test is in the form $\mathbf{x}[\phi] \leq \gamma$. Each leaf $l \in L$ stores the *prediction* $l.\texttt{val} \in \mathbb{R}$, representing the potential contribution of tree T to the final score of the document.

All the nodes whose Boolean conditions evaluate to False are called *false nodes*, and *true nodes* otherwise. The scoring of a document represented by a feature vector $\mathbf{x}$ requires the traversing of all the trees in the ensemble, starting at their root nodes. If a visited node in N is a false one, then the *right* branch is taken, and the *left* branch otherwise. The visit continues recursively until a leaf node is reached, where the value of the *prediction* is returned. Such leaf node is named *exit leaf* and denoted by $e(\mathbf{x}) \in L$. We omit $\mathbf{x}$ when it is clear from the context.

Hereinafter, we assume that nodes of T are numbered in breadth-first order and leaves from left to right, and let ϕ_i and γ_i be the feature id and threshold associated with i-th internal node, respectively. It is worth noting that the same feature can be involved in multiple nodes of the same tree. For example, in the tree shown in Figure 1, the features f_0 and f_2 are used twice. Assuming that $\mathbf{x}$ is such that $\mathbf{x}[2] > \gamma_0$, $\mathbf{x}[3] \leq \gamma_2$, and $\mathbf{x}[0] \leq \gamma_3$, the exit leaf e of the tree in the Figure 1 is the leaf l_2.

The tree traversal process is repeated for all the trees of the ensemble $\mathcal{T}$, denoted by $\mathcal{T} = \{T_0, T_1, \ldots\}$. The score $s(\mathbf{x})$ of the whole ensemble is finally computed as a *weighted sum* over the contributions of each tree $T_h = (N_h, L_h)$ in $\mathcal{T}$ as:

$$s(\mathbf{x}) = \sum_{h=0}^{|\mathcal{T}|-1} w_h \cdot e_h(\mathbf{x}).\texttt{val}$$

where $e_h(\mathbf{x}).\texttt{val}$ is the predicted value of tree T_h, having weight $w_h \in \mathbb{R}$.

In the following we review state-of-the-art optimization techniques for the implementation of additive ensemble of regression trees and their use in document scoring.

Tree traversal optimization. A naïve implementation of a tree traversal may exploit a node data structure that stores the feature id, the threshold and the pointers to the left and right children nodes. The traversal starts from the root and moves down to the leaves accordingly to the results of the Boolean conditions on the traversed nodes. This method can be enhanced by using the optimized data layout in [1]. The resulting algorithm is named STRUCT+. This simple approach entails a number of issues. First, the next node to be processed is known only after the test is evaluated. As the next instruction to be executed is not known, this induces frequent *control hazards*, i.e., instruction dependencies introduced by conditional branches. As a consequence, the efficiency of a code strongly depends on the *branch misprediction rate* [8]. Finally, due to the unpredictability of the path visited by a given document, the traversal has low temporal and spatial locality, generating low *cache hit ratio*. This is apparent when processing a large number of documents with a large ensemble of trees, since neither the documents nor the trees may fit in cache.

Another basic, but well performing approach is IF-THEN-ELSE. Each decision tree is translated into a sequence of if-then-else blocks, e.g., in C++ language. The resulting code is compiled to generate an efficient document scorer. IF-THEN-ELSE aims at taking advantage of compiler optimization strategies, which can potentially re-arrange the tree ensemble traversal into a more efficient procedure. The size of the resulting code is proportional to the total number of nodes in the ensemble. This makes it impossible to exploit successfully the instruction cache. IF-THEN-ELSE was proven to be efficient with small feature sets [1], but it still suffers from *control hazards*.

Asadi *et al.* [1] proposed to rearrange the computation to transform *control hazards* into *data hazards*, i.e., data dependencies introduced when one instruction requires the result of another. To this end, node n_s of a tree stores, in addition to a feature id ϕ_s and a threshold γ_s, an array `idx` of two positions holding the addresses of the left and right children nodes data structures. Then, the output of the test $\mathbf{x}[\phi_s] > \gamma_s$ is directly used as an index of such array in order to retrieve the next node to be processed. The visit of a tree of depth d is then statically "un-rolled" in d operations, starting from the root node n_0, as follows:

$$d \text{ steps} \begin{cases} i \leftarrow n_0.\texttt{idx}\,[\mathbf{x}[\phi_0] > \gamma_0] \\ i \leftarrow n_i.\texttt{idx}\,[\mathbf{x}[\phi_i] > \gamma_i] \\ \quad\vdots \qquad\qquad\quad \vdots \\ i \leftarrow n_i.\texttt{idx}\,[\mathbf{x}[\phi_i] > \gamma_i] \end{cases}$$

Leaf nodes are encoded so that the indexes in `idx` generate self loops, with dummy ϕ_s and γ_s. At the end of the visit, the exit leaf is identified by variable i, and a look-up table is used to retrieve the prediction of the tree.

This approach, named PRED, removes control hazards as the next instruction to be executed is always known. On the other hand, data dependencies are not solved as the output of one instruction is required to execute the subsequent. Memory access patterns are not improved either, as they depend on the path along the tree traversed by a document. Finally, PRED introduces a new source of overhead: for a tree of depth d, even if document reaches a leaf early, the above d steps are executed anyway.

To reduce *data hazards* the same authors proposed a *vectorized* version of the scoring algorithm, named VPRED, by interleaving the evaluation of a small set of documents (16 was the best setting). VPRED was shown to be 25% to 70% faster than PRED on synthetic data, and to outperform other approaches. The same approach of PRED was also adopted in some previous works exploiting GPUs [11], and a more recent survey evaluates the trade-off among multi-core CPUs, GPUs and FPGA [13].

In this work we compare against VPRED which can be considered the best performing algorithm at the state of the art. In the experimental section, we show that the proposed QS algorithm has reduced *control hazard*, smaller *branch mis-prediction* rate and better *memory access patterns*.

Memory latency issues of scoring algorithms are tackled in Tang *et al.* [12]. In most cases, the cache memory may be insufficient to store the candidate documents to be scored and/or the set of regression trees. The authors proposed a cache-conscious optimization by splitting documents and regression trees in *blocks*, such that one block of documents and one block of trees can both be stored in cache at the same time. Computing the score of all documents requires to evaluate all the tree blocks against all the document blocks. Authors applied this computational scheme on top of both IF-THEN-ELSE and PRED, with an average improvement of about 28% and 24% respectively. The blocking technique is indeed very general and can be used by all algorithms. The same computational schema is applied to QS in order to improve the *cache hit ratio* when large ensembles are used.

Other approaches and optimizations. Unlike our method that aims to devise an efficient strategy for fully evaluating the ensemble of trees, other approaches tries to approximate the computation over the ensemble for reducing the scoring time. Cambazoglu *et al.* [3] proposed to early terminate the scoring of documents that are unlikely to be ranked within the top-k results. Their work applies to an ensemble of additive trees like ours, but the authors aims to save scoring time by reducing the number of tree traversals, and trades better efficiency for little loss in raking quality. Although our method is thought for globally optimizing the traversal of thousands of trees, the idea of early termination can be applied as well along with our method, by evaluating some proper exit strategy after the evaluation of some subsets of the regression trees.

Wang *et al.* [15, 17, 16] deeply investigated different efficiency aspects of the ranking pipeline. In particular, in [16] they propose a novel cascade ranking model, which unlike previous approaches, can simultaneously improve both top-k ranked effectiveness and retrieval efficiency. Their work is mainly related to the tuning of a two-stage ranking pipeline.

3. QUICKSCORER: AN EFFICIENT TREE ENSEMBLE TRAVERSAL ALGORITHM

In order to efficiently exploit memory hierarchies and to reduce the branch mis-prediction rate, we propose an algorithm based on a totally novel traversal of the trees ensemble, called QUICKSCORER (QS). The building block of our approach is an alternative method for tree traversal based on bitvector computations, which is presented in Subsection 3.1. Given a tree and a vector of document features, our traversal processes all its nodes and produces a bitvector which encodes the exit leaf for the given document. In isolation this traversal is not particularly advantageous over the

others, since in principle it requires to evaluate all the nodes of a tree. However, it has the nice property of being insensitive to the order in which the nodes are processed. This makes it possible to interleave the evaluation of the trees in the ensemble in a *cache-aware* fashion. In addition, the proposed bitvector encoding allows to save the computation of many test conditions.

The interleaved evaluation of a trees ensemble is discussed in Subsection 3.2. Intuitively, rather than traversing the ensemble tree after tree, our algorithm performs a global visit of the ensemble by traversing portions of all the trees together, feature by feature. For each feature, we store all the associated thresholds occurring anywhere in the ensemble in a sorted array, to easily to compute the result of all the test conditions involved. A bitvector for each tree is updated after each test, in such a way to encode, at the end of the process, the exit leaves in each tree for a given document. These bitvector are eventually used to lookup the predicted value of each tree.

3.1 Tree traversal using bitvectors

We start by presenting a simpler version of our tree traversal and, then, we introduce two crucial refinements for the performance of this algorithm when used in the interleaved evaluation of all the trees as described in Subsection 3.2.

Given an input feature vector $\mathbf{x}$ and a tree $T_h = (N_h, L_h)$, our tree traversal algorithm processes the internal nodes of T_h with the goal of identifying a set of *candidate exit leaves*, denoted by $\mathcal{C}_h$ with $\mathcal{C}_h \subseteq L_h$, which includes the actual exit leaf e_h. Initially $\mathcal{C}_h$ contains all the leaves in L_h, i.e., $\mathcal{C}_h = L_h$. Then, the algorithm evaluates one after the other in an arbitrary order the test conditions of all the internal nodes of T_h. Considering the result of the test for a certain internal node $n \in N_h$, the algorithm is able to infer that some leaves cannot be the exit leaf and, thus, it can safely remove them from $\mathcal{C}_h$. Indeed, if n is a *false node* (i.e., its test condition is false), the leaves in the left subtree of n cannot be the exit leaf and they can be safely removed from $\mathcal{C}_h$. Similarly, if n is a *true node*, the leaves in the right subtree of n can be removed from $\mathcal{C}_h$. It is easy to see that, once all the nodes have been processed, the only leaf left in $\mathcal{C}_h$ is the exit leaf e_h.

The first refinement turns the above algorithm into a lazy one. This lazy algorithm uses an oracle, called `FindFalse`, that, given T_h and $\mathbf{x}$, returns the false nodes in N_h without the need of evaluating all the associated test conditions. Then, the algorithm removes from $\mathcal{C}_h$ the leaves in the left subtrees of all the false nodes returned by the oracle. For the moment we concentrate on the set $\mathcal{C}_h$ obtained at the end of the algorithm and we defer the materialization of the above oracle to Subsection 3.2 where the interleaved evaluation of all the trees makes its implementation possible. Observe that $\mathcal{C}_h$ may now contain several leaves. As an extreme example, the set $\mathcal{C}_h$, in absence of false nodes, will contain all the leaves in L_h. Interestingly, we can prove (see Theorem 1 below) that the *exit leaf* e_h is always the one associated with the smallest identifier in $\mathcal{C}_h$, i.e., the leftmost leaf in the tree. A running example is reported in Figure 2 which shows the actual traversal (bold arrows) for a vector $\mathbf{x}$, and also the true and false nodes. The figure shows also the set $\mathcal{C}_h$ after the removal of the leaves of the left subtrees of false nodes: $\mathcal{C}_h$ is $\{l_2, l_3, l_5\}$ and, indeed, the exit leaf is the leftmost leaf in $\mathcal{C}_h$, i.e., $e_h = l_2$.

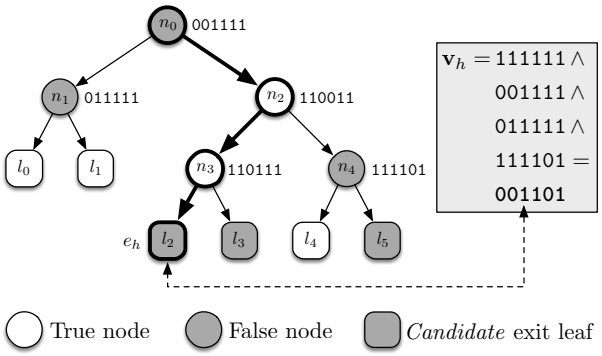


Figure 2: Tree traversal example.

The second refinement implements the operations on $\mathcal{C}_h$ with fast operations on compact bitvectors. The idea is to represent $\mathcal{C}_h$ with a bitvector $\mathbf{v}_h$, where each bit corresponds to a distinct leaf in L_h, i.e., $\mathbf{v}_h$ is the characteristic vector of $\mathcal{C}_h$. Every internal node n is associated with a *node bitvector* (of the same length), acting as a bitmask that encodes (with 0's) the set of leaves to be removed from $\mathcal{C}_h$ whenever n is a false node. This way, the bitwise logical AND between $\mathbf{v}_h$ and the node bitvector of a false node n corresponds to the removal of the leaves in the left subtree of n from $\mathcal{C}_h$. We finally observe that the exit leaf corresponds to the leftmost bit set to 1 in $\mathbf{v}_h$. Figure 2 shows how the initial bitvector $\mathbf{v}_h$ is updated by using bitwise logical AND operations.

The full approach is described in Algorithm 1. Given a binary tree $T_h = (N_h, L_h)$ and an input feature vector $\mathbf{x}$, let u.`bitvector` be the precomputed bitwise mask associated with a generic $n \in N_h$. First the result bitvector $\mathbf{v}_h$ is initialized with all bits set to 1. Then, `FindFalse`$(\mathbf{x}, T_h)$ returns all the false nodes in N_h. For each of such nodes, $\mathbf{v}_h$ is masked with the corresponding node bitvector. Finally, the position of the leftmost bit of $\mathbf{v}_h$ identifies the exit leaf e_h, whose output value is returned. The correctness of this approach is stated by the following theorem.

THEOREM 1. *Algorithm 1 is correct.*

PROOF. We prove that for each binary decision tree T_h and input feature vector $\mathbf{x}$, Algorithm 1 always computes

Algorithm 1: Scoring a feature vector $\mathbf{x}$ using a binary decision tree T_h

Input :
- $\mathbf{x}$: input feature vector
- $T_h = (N_h, L_h)$: binary decision tree, with
  - $N_h = \{n_0, n_1, \ldots\}$: internal nodes of T_h
  - $L_h = \{l_0, l_1, \ldots\}$: leaves of T_h
  - n.`bitvector`: node bitvector associated with $n \in N_h$
  - l_j.`val`: output value associated with $l_j \in L_h$

Output:
- tree traversal output value

`Score(`$\mathbf{x}$`,`T_h`):`

```
1  v_h ← 11...11
2  U ← FindFalse(x, T_h)
3  foreach node u ∈ U do
4      v_h ← v_h ∧ u.bitvector
5  j ← index of leftmost bit set to 1 of v_h
6  return l_j.val
```

a result bitvector $\mathbf{v}_h$, where the leftmost bit set to 1 corresponds to the exit leaf e_h.

First, we prove that the bit corresponding to the exit leaf e_h in the result bitvector $\mathbf{v}_h$ is always set to 1. Consider the internal nodes along the path from the root to e_h, and observe that only the bitvectors applied for those nodes may change the e_h's bit to 0. Since e_h is the exit leaf, it belongs to the left subtree of any true node and to the right subtree of any false node in this path. Thus, since the bitvectors are used to set to 0 leaves in the left subtrees of false nodes, the bit corresponding to e_h remains unmodified, and, thus, will be 1 at the end of Algorithm 1.

Second, we prove that the leftmost bit equal to 1 in $\mathbf{v}_h$ corresponds to the exit leaf e_h. Let $l_{\leftarrow}$ be the leaf corresponding to the leftmost bit set to 1 in $\mathbf{v}_h$. Assume by contradiction that e_h is not the leftmost bit set to 1 in $\mathbf{v}_h$, namely, $l_{\leftarrow} \neq e_h$. Let u be their lowest common ancestor node in the tree. Since $l_{\leftarrow}$ is smaller than e_h, the leaf $l_{\leftarrow}$ belongs to u's left subtree while the leaf e_h belongs to u's right subtree. This leads to a contradiction. Indeed, on one hand, the node u should be a true node otherwise its bitvector would have been applied setting $l_{\leftarrow}$'s bit to 0. On the other hand, the node u should be a false node since e_h is in its right subtree. Thus, we conclude that $l_{\leftarrow} = e_h$ proving the correctness of Algorithm 1. □

Algorithm 1 represents a general technique to compute the output value of a single binary decision tree stored as a set of precomputed bitvectors. Given an additive ensemble of binary decision trees, to score a document $\mathbf{x}$ we have to loop over all the trees $T_h \in \mathcal{T}$ by repeatedly applying Algorithm 1. Unfortunately, this naïve algorithm is inefficient, since this method does not permit us to implement efficiently `FindFalse`$(\mathbf{x}, T_h)$.

In the following section we present QS, which overcomes this issue by performing a global visit of the whole tree ensemble $\mathcal{T}$. The QS algorithm realizes the goal of identifying efficiently the false nodes of all the tree ensemble by exploiting an *interleaved* evaluation of all the trees in the ensemble.

3.2 The QS Algorithm

Our QS algorithm scores a feature vector $\mathbf{x}$ with an interleaved execution of several tree traversals, one for each tree in the ensemble. The algorithm does not loop over all the trees in $\mathcal{T}$ one at the time, as one would expect, but does loop instead over all the features in $\mathcal{F}$, hence incrementally discovering for each $f_k \in \mathcal{F}$ the false nodes involving f_k in any tree of the ensemble. This is a very convenient order for two reasons: i) we are able to identify all the false nodes for all the trees without even considering their true nodes, thus effectively implementing the oracle introduced in the previous section; ii) we are able to operate in a cache-aware fashion with a small number of Boolean comparisons and branch mis-predictions.

During its execution, QS has to maintain the bitvectors $\mathbf{v}_h$'s, encoding the set $\mathcal{C}_h$'s for all the tree T_h in the ensemble. The bitvector $\mathbf{v}_h$ of a certain tree is updated as soon as a false node for that tree is identified. Once the algorithm has processed all the features in $\mathcal{F}$, each of these $\mathbf{v}_h$ is guaranteed to encode the exit leaf in the corresponding tree. Now the algorithm can compute the overall score of $\mathbf{x}$ by summing up (and, possibly, weighting) the scores of all these exit leaves.

Let us concentrate on the processing of a feature f_k and describe the portion of the data structure of interest for this feature. The overall algorithm simply iterates this process over all features in $\mathcal{F}$. Each node involving f_k in any tree $T_h \in \mathcal{T}$ is represented by a *triple* containing: (i) the feature threshold involved in the Boolean test; (ii) the id of the tree that contains the node, where the id is used to identify the bitvector $\mathbf{v}_h$ to update; (iii) the node bitvector used to possibly update $\mathbf{v}_h$. We sort these triples in *ascending order* of their feature thresholds.

This sorting is crucial for obtaining a fast implementation of our oracle. Recall that all the conditions occurring in the internal nodes of the trees are of the form $\mathbf{x}[k] \leq \gamma_s^h$. Hence, given the sorted list of all the thresholds involving $f_k \in \mathcal{F}$, the feature value $\mathbf{x}[k]$ splits the list in two, possibly empty, sublists. The first sublist contains all the thresholds γ_s^h for which the test condition $\mathbf{x}[k] \leq \gamma_s^h$ evaluates to FALSE, while the second sublists contains all thresholds for which the test condition evaluates to TRUE. Thus, if we sequentially scan the sorted list of the thresholds associated with f_k, all the values in the first sublist will cause negative tests. Associated with these thresholds entailing false tests, we have false nodes belonging to the trees in $\mathcal{T}$. Therefore, for all these false nodes we can take in sequence the corresponding bitvector, and perform a bitwise logical AND with the appropriate result bitvector $\mathbf{v}_h$.

This large sequence of tests that evaluates to FALSE corresponds to the repeated execution of conditional branch instructions, whose behavior is indeed very predictable. This

Algorithm 2: The QUICKSCORER Algorithm

Input :
- $\mathbf{x}$: input feature vector
- $\mathcal{T}$: ensemble of binary decision trees, with
  - $w_0, \ldots, w_{|\mathcal{T}|-1}$: weights, one per tree
  - `thresholds`: sorted sublists of thresholds, one sublist per feature
  - `tree_ids`: tree's ids, one per threshold
  - `bitvectors`: node bitvectors, one per threshold
  - `offsets`: offsets of the blocks of triples
  - `v`: result bitvectors, one per each tree
  - `leaves`: output values, one per each tree leaf

Output:
- Final score of $\mathbf{x}$

QUICKSCORER($\mathbf{x}$,$\mathcal{T}$):

```
1  foreach h ∈ 0,1,...,|T|−1 do
2      v[h] ← 11...11
3  foreach k ∈ 0,1,...,|F|−1 do                    // Step ①
4      i ← offsets[k]
5      end ← offsets[k+1]
6      while x[k] > thresholds[i] do
7          h ← tree_ids[i]
8          v[h] ← v[h] ∧ bitvectors[i]
9          i ← i+1
10         if i ≥ end then
11             break
12 score ← 0
13 foreach h ∈ 0,1,...,|T|−1 do                    // Step ②
14     j ← index of leftmost bit set to 1 of v[h]
15     l ← h·|L_h| + j
16     score ← score + w_h · leaves[l]
17 return score
```

is confirmed by our experimental results, showing that our code incurs in very few branch mis-predictions.

We now present the layout in memory of the required data structure since it is crucial for the efficiency of our algorithm. The triples of each feature are stored in three separate arrays, one for each component: `thresholds`, `tree_ids`, and `bitvectors`. The use of three distinct arrays solves some data alignment issues arising when tuples of heterogeneous data types are stored contiguously in memory. The arrays of the different features are then juxtaposed one after the other as illustrated in Figure 3. Since arrays of different features may have different lengths, we use an auxiliary array `offsets` which marks the starting position of each array in the global array. We also juxtapose the bitvectors $\mathbf{v}_h$ into a global array `v`. Finally, we use an array `leaves` which stores the output values of the leaves of each tree (ordered from left to right) grouped by their tree id.

Algorithm 2 reports the steps of QS as informally described above. After the initialization of the result bitvectors of each tree (loop starting al line 1), we have the first step of QS that exactly corresponds to what we discussed above (loop starting at line 3). The algorithm iterates over all features, and inspects the sorted lists of thresholds to update the result bitvectors. Upon completion of the first step, we have the second step of the algorithm (loop starting at line 13), which simply inspects all the result bitvectors, and for each of them identifies the position of the leftmost bit set to 1, and uses this position to access the value associated with the corresponding leaf stored array `leaves`. The value of the leaf is finally used to update the final score.

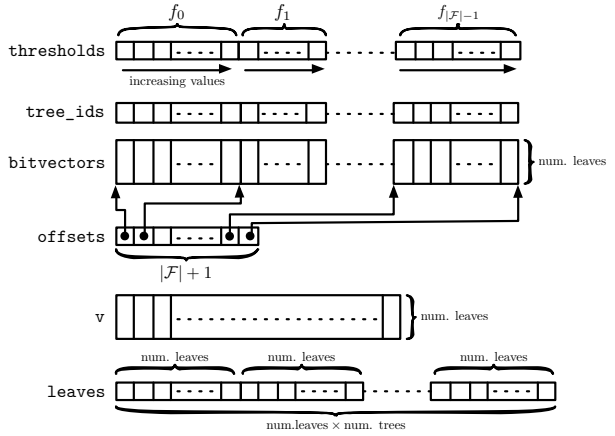


Figure 3: Arrays used by algorithm QS.

Implementation details. In the following we discuss some details about our data structures, their size and access modes.

A few important remarks concern the bitvectors stored in `v` and `bitvectors`. The learning algorithm controls the accuracy of each single tree with a parameter Λ, which determines the maximal number of leaves for each $T_h = (N_h, L_h)$ in $\mathcal{T}$, namely $|L_h| \leq \Lambda$. Usually, the value of Λ is kept small (≤ 64). Thus, the length of bitvectors, which have to encode tree leaves, is equal to (or less than) a typical machine word of modern CPUs (64 bits). As a consequence, the bitwise operations performed by Algorithm 2 on them can be realized very efficiently, because they involve machine words (or halfwords, etc).

We avoid any possible performance overhead due to shifting operations to align the operands of bitwise logical ANDs by forcing the bitvectors to have uniform length of B bytes. To this end, we pad each bitvector on its right side with a string of 0 bits, if necessary. We always select the minimum number of bytes $B \in \{1, 2, 4, 8\}$ fitting Λ.

Let us now consider Table 1, which shows an upper bound for the size of each linear array used by our algorithm. The array `offsets` has $|\mathcal{F}|$ entries, one entry for each distinct feature. The array `v`, instead, has an entry for each tree in $\mathcal{T}$, thus, $|\mathcal{T}|$ entries overall. The sizes of the other data structures depends on the number of total internal nodes or leaves in the ensemble $\mathcal{T}$, besides the datatype sizes. Any internal node of some tree of $\mathcal{T}$ contributes with an entry in each array `thresholds`, `bitvectors` and `tree_ids`. Therefore the total number of entries of each of these arrays, i.e., $\sum_0^{|\mathcal{T}|-1} |N_h|$, can be upper bounded by $|\mathcal{T}| \cdot \Lambda$, because for every tree T_h we have $|N_h| < |N_h| + 1 = |L_h| \leq \Lambda$. Finally, the array `leaves` has an entry for each leaf in a tree of $\mathcal{T}$, hence, no more than $|\mathcal{T}| \cdot \Lambda$ in total.

Table 1: Data structures used by QS, the corresponding maximum sizes, and the access modes.

Data structure	*Maximum Size (bytes)*	*Data access modes*
`thresholds`	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot$ `sizeof(float)`	*1. Sequential (R)*
`tree_ids`	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot$ `sizeof(uint)`	
`bitvectors`	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot B$	
`offsets`	$\lvert\mathcal{F}\rvert \cdot$ `sizeof(uint)`	
`v`	$\lvert\mathcal{T}\rvert \cdot B$	*1. Random (R/W)* *2. Sequential (R)*
`leaves`	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot$ `sizeof(double)`	*2. Seq. Sparse (R)*

The last column of Table 1 reports the *data access modes* to the arrays, where the leading number, either 1 or 2, corresponds to the step of the algorithm during which the data structures are read/written. Recall that the first step of QS starts at line 3 of Algorithm 2, while the second at line 13. We first note that `v` is the only array used in both phases of function QUICKSCORER($\mathbf{x}, \mathcal{T}$). During the first step `v` is accessed randomly in reading/writing to update the $\mathbf{v}_h$'s. During the second step the same array is accessed sequentially in reading mode to identify the exit leafs l_h of each tree T_h, and then to access the array `leaves` to read the contribution of tree T_h to the output of the regression function. Even if the trees and their leaves are accessed sequentially during the second step of QS, the reading access to array `leaves` is sequential, but very sparse: only one leaf of each block of $|L_h|$ elements is actually read.

Finally, note that the arrays storing the triples, i.e., `thresholds`, `tree_ids`, and `bitvectors`, are all sequentially read during the first step, though not completely, since for each feature we stop its inspection at the first test condition that evaluates to TRUE. The cache usage can greatly benefit from the layout and access modes of our data structures, thanks to the increased references locality.

We finally describe an optimization which aims at reducing the number of comparisons performed at line 6 of Algorithm 2. The (inner) while loop in line 6 iterates over the list of threshold values associated with a certain feature $f_k \in \mathcal{F}$ until we find the first index j where the test fails, namely, the value of the k^{th} feature of vector `x` is greater than `thresholds`$[j]$. Thus, a test on the feature

value and the current threshold is carried out at each iteration. Instead of testing each threshold in a prefix of `thresholds`$[i : end]$, our optimized implementation tests only one every Δ thresholds, where Δ is a parameter. Since the subvector `thresholds`$[i : end]$ is sorted in ascending order, if a test succeed the same necessarily holds for all the preceding $\Delta - 1$ thresholds. Therefore, we can go directly to update the result bitvector $\mathbf{v}_h$ of the corresponding trees, saving $\Delta - 1$ comparisons. Instead, if the test fails, we scan the preceding $\Delta - 1$ thresholds to identify the target index j and we conclude. In our implementation we set Δ equal to 4, which is the value giving the best results in our experiments. We remark that in principle one could identify j by binary searching the subvector `thresholds`$[i : end]$. Experiments have shown that the use of binary search is not profitable because in general the subvector is not sufficiently long.

4. EXPERIMENTS

In this section we provide an extensive experimental evaluation that compares our QS algorithm with other state-of-the-art competitors and baselines over standard datasets.

Datasets and experimental settings. Experiments are conducted by using publicly available LtR datasets: the MSN[1] and the Yahoo! LETOR[2] challenge datasets. The first one is split into five folds, consisting of vectors of 136 features extracted from query-document pairs, while the second one consists of two distinct datasets (Y!S1 and Y!S2), made up of vectors of 700 features. In this work, we focus on MSN-1, the first MSN fold, and Y!S1 datasets. The features vectors of the two selected datasets are labeled with relevance judgments ranging from 0 (irrelevant) to 4 (perfectly relevant). Each dataset is split in training, validation and test sets. The MSN-1 dataset consists of 6,000, 2,000, and 2,000 queries for training, validation and testing respectively. The Y!S1 dataset consists of 19,944 training queries, 2,994 validation queries and 6,983 test queries.

We exploit the following experimental methodology. We use training and validation sets from MSN-1 and Y!S1 to train λ-MART [18] models with 8, 16, 32 and 64 leaves. We use QuickRank[3] an open-source parallel implementation of λ-MART written in C++11 for performing the training phase. During this step we optimize NDCG@10. The results of the paper can be also applied to analogous tree-based models generated by different state-of-the-art learning algorithms, e.g., GBRT [4].

In our experiments we compare the scoring efficiency of QS[4] with the following competitors:

- If-Then-Else is a baseline that translates each tree of the forest as a nested block of if-then-else.
- VPred and Struct+ [1] kindly made available by the authors[5].

Given a trained model and a test set, all the above scoring methods achieve the same result in terms of effectiveness. Hence, we do not report quality measures.

All the algorithms are compiled with GCC 4.9.2 with the highest optimization settings. The tests are performed by using a single core on a machine equipped with an Intel Core i7-4770K clocked at 3.50Ghz, with 32GiB RAM, running Ubuntu Linux 3.13.0. The Intel Core i7-4770K CPU has three levels of cache. Level 1 cache has size 32 KB, one for each of the four cores, level 2 cache has size 256 KB for each core, and at level 3 there is a shared cache of 8 MB.

To measure the efficiency of each of the above methods, we run 10 times the scoring code on the test sets of the MSN-1 and Y!S1 datasets. We then compute the average per-document scoring cost. Moreover, to deeply profile the behavior of each method above we employ `perf`[6], a performance analysis tool available under Ubuntu Linux distributions. We analyze each method by monitoring several CPU counters that measure the total number of instructions executed, number of branches, number of branch mispredictions, cache references, and cache misses.

Scoring time analysis. The average time (in μs) needed by the different algorithms to score each document of the two datasets MSN-1 and Y!S1 are reported in Table 2. In particular, the table reports the per-document scoring time by varying the number of trees and the leaves of the ensemble employed. For each test the table also reports between parentheses the gain factor of QS over its competitors. At a first glance, these gains are impressive, with speedups that in many cases are above one order of magnitude. Depending on the number of trees and of leaves, QS outperforms VPred, the most efficient solution so far, of factors ranging from 2.0x up to 6.5x. For example, the average time required by QS and VPred to score a document in the MSN-1 test set with a model composed of $1,000$ trees and 64 leaves, are 9.5 and 62.2 μs, respectively. The comparison between QS and If-Then-Else is even more one-sided, with improvements of up to 23.4x for the model with $10,000$ trees and 32 leaves trained on the MSN-1 dataset. In this case the QS average per-document scoring time is 59.6 μs with respect to the 1396.8 μs of If-Then-Else. The last baseline reported, i.e., Struct+, behaves worst in all the tests conducted. Its performance is very low when compared not only to QS (up to 38.2x times faster), but even with respect to the other two algorithms VPred and If-Then-Else. The reasons of the superior performance of QS over competitor algorithms are manyfold. We analyse the most relevant in the following.

Instruction level analysis. We used the `perf` tool to measure the total number of instructions, number of branches, number of branch mis-predictions, L3 cache references, and L3 cache misses of the different algorithms by considering only their scoring phase. Table 3 reports the results we obtained by scoring the MSN-1 test set by varying the number of trees and by fixing the number of leaves to 64. Experiments on Y!S1 are not reported here, but they exhibited similar behavior. As a clarification, L3 cache references accounts for those references which are not found in any of the previous level of cache, while L3 cache misses are the ones among them which miss in L3 as well. Table 3 also reports the number of visited nodes. All measurements are per-document and per-tree normalized.

[1] `http://research.microsoft.com/en-us/projects/mslr/`
[2] `http://learningtorankchallenge.yahoo.com`
[3] `http://quickrank.isti.cnr.it`
[4] The C++11 implementation of QS is available at `http://github.com/hpclab/quickscorer`.
[5] `http://nasadi.github.io/OptTrees/`
[6] `https://perf.wiki.kernel.org`

Table 2: Per-document scoring time in μs of QS, VPRED, IF-THEN-ELSE and STRUCT+ on MSN-1 and Y!S1 datasets. Gain factors are reported in parentheses.

Method	Λ	Number of trees/dataset							
		1,000		5,000		10,000		20,000	
		MSN-1	Y!S1	MSN-1	Y!S1	MSN-1	Y!S1	MSN-1	Y!S1
QS	8	**2.2** (–)	**4.3** (–)	**10.5** (–)	**14.3** (–)	**20.0** (–)	**25.4** (–)	**40.5** (–)	**48.1** (–)
VPRED		7.9 (3.6x)	8.5 (2.0x)	40.2 (3.8x)	41.6 (2.9x)	80.5 (4.0x)	82.7 (3.3)	161.4 (4.0x)	164.8 (3.4x)
IF-THEN-ELSE		8.2 (3.7x)	10.3 (2.4x)	81.0 (7.7x)	85.8 (6.0x)	185.1 (9.3x)	185.8 (7.3x)	709.0 (17.5x)	772.2 (16.0x)
STRUCT+		21.2 (9.6x)	23.1 (5.4x)	107.7 (10.3x)	112.6 (7.9x)	373.7 (18.7x)	390.8 (15.4x)	1150.4 (28.4x)	1141.6 (23.7x)
QS	16	**2.9** (–)	**6.1** (–)	**16.2** (–)	**22.2** (–)	**32.4** (–)	**41.2** (–)	**67.8** (–)	**81.0** (–)
VPRED		16.0 (5.5x)	16.5 (2.7x)	82.4 (5.0x)	82.8 (3.7x)	165.5 (5.1x)	165.2 (4.0x)	336.4 (4.9x)	336.1 (4.1x)
IF-THEN-ELSE		18.0 (6.2x)	21.8 (3.6x)	126.9 (7.8x)	130.0 (5.8x)	617.8 (19.0x)	406.6 (9.9x)	1767.3 (26.0x)	1711.4 (21.1x)
STRUCT+		42.6 (14.7x)	41.0 (6.7x)	424.3 (26.2x)	403.9 (18.2x)	1218.6 (37.6x)	1191.3 (28.9x)	2590.8 (38.2x)	2621.2 (32.4x)
QS	32	**5.2** (–)	**9.7** (–)	**27.1** (–)	**34.3** (–)	**59.6** (–)	**70.3** (–)	**155.8** (–)	**160.1** (–)
VPRED		31.9 (6.1x)	31.6 (3.2x)	165.2 (6.0x)	162.2 (4.7x)	343.4 (5.7x)	336.6 (4.8x)	711.9 (4.5x)	694.8 (4.3x)
IF-THEN-ELSE		34.5 (6.6x)	36.2 (3.7x)	300.9 (11.1x)	277.7 (8.0x)	1396.8 (23.4x)	1389.8 (19.8x)	3179.4 (20.4x)	3105.2 (19.4x)
STRUCT+		69.1 (13.3x)	67.4 (6.9x)	928.6 (34.2x)	834.6 (24.3x)	1806.7 (30.3x)	1774.3 (25.2x)	4610.8 (29.6x)	4332.3 (27.0x)
QS	64	**9.5** (–)	**15.1** (–)	**56.3** (–)	**66.9** (–)	**157.5** (–)	**159.4** (–)	**425.1** (–)	**343.7** (–)
VPRED		62.2 (6.5x)	57.6 (3.8x)	355.2 (6.3x)	334.9 (5.0x)	734.4 (4.7x)	706.8 (4.4x)	1309.7 (3.0x)	1420.7 (4.1x)
IF-THEN-ELSE		55.9 (5.9x)	55.1 (3.6x)	933.1 (16.6x)	935.3 (14.0x)	2496.5 (15.9x)	2428.6 (15.2x)	4662.0 (11.0x)	4809.6 (14.0x)
STRUCT+		109.8 (11.6x)	116.8 (7.7x)	1661.7 (29.5x)	1554.6 (23.2x)	3040.7 (19.3x)	2937.3 (18.4x)	5437.0 (12.8x)	5456.4 (15.9x)

We first observe that VPRED executes the largest number of instructions. This is because VPRED always runs d steps if d is the depth of a tree, even if a document might reach an exit leaf earlier. IF-THEN-ELSE executes much less instructions as it follows the document traversal path. STRUCT+ introduces some data structures overhead w.r.t. IF-THEN-ELSE. QS executes the smallest number instructions. This is due to the different traversal strategy of the ensemble, as QS needs to process the *false nodes* only. Indeed, QS always visits less than 18 nodes on average, out of the 64 present in each tree of the ensemble. Note that IF-THEN-ELSE traverses between 31 and 40 nodes per tree, and the same trivially holds for STRUCT+. This means that the interleaved traversal strategy of QS needs to process less nodes than in a traditional root-to-leaf visit. This mostly explains the results achieved by QS.

As far as number of branches is concerned, we note that, not surprisingly, QS and VPRED are much more efficient than IF-THEN-ELSE and STRUCT+ with this respect. QS has a larger total number of branches than VPRED, which uses scoring functions that are branch-free. However, those branches are highly predictable, so that the mis-prediction rate is very low, thus, confirming our claims in Section 3.

Observing again the timings in Table 2 we notice that, by fixing the number of leaves, we have a super-linear growth of QS's timings when increasing the number of trees. For example, since on MSN-1 with $\Lambda = 64$ and $1,000$ trees QS scores a document in 9.5 μs, one would expect to score a document 20 times slower, i.e., 190 μs, when the ensemble size increases to $20,000$ trees. However, the reported timing of QS in this setting is 425.1 μs, i.e., roughly 44 times slower than with 1000 trees. This effect is observable only when the number of leaves $\Lambda = \{32, 64\}$ and the number of trees is larger than $5,000$. Table 3 relates this super-linear growth to the numbers of L3 cache misses.

Considering the sizes of the arrays as reported in Table 1 in Section 3, we can estimate the minimum number of trees that let the size of the QS's data structure to exceed the cache capacity, and, thus, the algorithm starts to have more cache misses. This number is estimated in $6,000$ trees when the number of leaves is 64. Thus, we expect that the number of L3 cache miss starts increasing around this number of trees. Possibly, this number is slightly larger, because portions of the data structure may be infrequently accessed at scoring time, due the the small fraction of false nodes and associated bitvectors accessed by QS.

These considerations are further confirmed by Figure 4, which shows the average per-tree per-document scoring time (μs) and percentage of cache misses QS when scoring the MSN-1 and the Y!S1 with $\Lambda = 64$ by varying the number of trees. First, there exists a strong correlation between QS's timings and its number of L3 cache misses. Second, the number of L3 cache misses starts increasing when dealing with $9,000$ trees on MSN and $8,000$ trees on Y!S1.

BWQS: a block-wise variant of QS

The previous experiments suggest that improving the cache efficiency of QS may result in significant benefits. As in Tang *et al.* [12], we can split the tree ensemble in disjoint blocks of size τ that are processed separately in order to let the corresponding data structures fit into the faster levels of the memory hierarchy. This way, we are essentially scoring each document over each tree blocks that partition the original ensemble, thus inheriting the efficiency of QS on smaller ensembles. Indeed, the size of the arrays required to score the documents over a block of trees depends now on τ instead of $|\mathcal{T}|$ (see Table 1 in Section 3). We have, however, to keep an array that stores the partial scoring computed so far for each document.

The temporal locality of this approach can be improved by allowing the algorithm to score blocks of documents together over the same block of trees before moving to the next block of documents. To allow the algorithm to score a block of δ documents in a single run we have to replicate in δ copies the array `v`. Obviously, this increases the space occupancy and may result in a worse use of the cache. Therefore, we need to find the best balance between the number of documents δ and the number of trees τ to process in the body of a nested loop that first runs over the blocks of trees (outer loop) and then over the blocks of documents to score (inner loop).

This algorithm is called BLOCKWISE-QS (BWQS) and its efficiency is discussed in the remaining part of this section.

Table 3: Per-tree per-document low-level statistics on MSN-1 with 64-leaves λ-MART models.

Method	Number of Trees				
	1,000	5,000	10,000	15,000	20,000
Instruction Count					
QS	**58**	**75**	**86**	**91**	**97**
VPRED	580	599	594	588	516
IF-THEN-ELSE	142	139	133	130	116
STRUCT+	341	332	315	308	272
Num. branch mis-predictions (above) Num. branches (below)					
QS	0.162	**0.035**	**0.017**	**0.011**	**0.009**
	6.04	7.13	8.23	8.63	9.3
VPRED	**0.013**	0.042	0.045	0.049	0.049
	0.2	**0.21**	**0.18**	**0.21**	**0.21**
IF-THEN-ELSE	1.541	1.608	1.615	1.627	1.748
	42.61	41.31	39.16	38.04	33.65
STRUCT+	4.498	5.082	5.864	6.339	5.535
	89.9	88.91	85.55	83.83	74.69
L3 cache misses (above) L3 cache references (below)					
QS	0.004	**0.001**	**0.121**	**0.323**	0.51
	1.78	**1.47**	**1.52**	**2.14**	**2.33**
VPRED	0.005	0.166	0.326	0.363	**0.356**
	12.55	12.6	13.74	15.04	12.77
IF-THEN-ELSE	**0.001**	17.772	30.331	29.615	29.577
	27.66	38.14	40.25	40.76	36.47
STRUCT+	0.039	12.791	17.147	15.923	13.971
	7.37	18.64	20.52	19.87	18.38
Num. Visited Nodes (above) Visited Nodes/Total Nodes (below)					
QS	**9.71**	**13.40**	**15.79**	**16.65**	**18.00**
	15%	**21%**	**25%**	**26%**	**29%**
VPRED	54.38	56.23	55.79	55.23	48.45
	86%	89%	89%	88%	77%
STRUCT+	40.61	39.29	37.16	36.15	31.75
IF-THEN-ELSE	64%	62%	59%	57%	50%

Table 4 reports average per-document scoring time in μs of algorithms QS, VPRED, and BWQS. The experiments were conducted on both the MSN-1 and Y!S1 datasets by varying Λ and by fixing the number of trees to 20,000. It is worth noting that our QS algorithm can be thought as a limit case of BWQS, where the blocks are trivially composed of 1 document and the whole ensemble of trees. VPRED instead vectorizes the process and scores 16 documents at the time over the entire ensemble. With BWQS the sizes of document and tree blocks can be instead flexibly optimized according to the cache parameters. Table 4 reports the best execution times, along with the values of δ and τ for which BWQS obtained such results.

The blocking strategy can improve the performance of QS when large tree ensembles are involved. Indeed, the largest improvements are measured in the tests conducted on models having 64 leaves. For example, to score a document of MSN-1, BWQS with blocks of 3,000 trees and a single document takes 274.7 μs in average, against the 425.1 μs required by QS with an improvement of 1.55x.

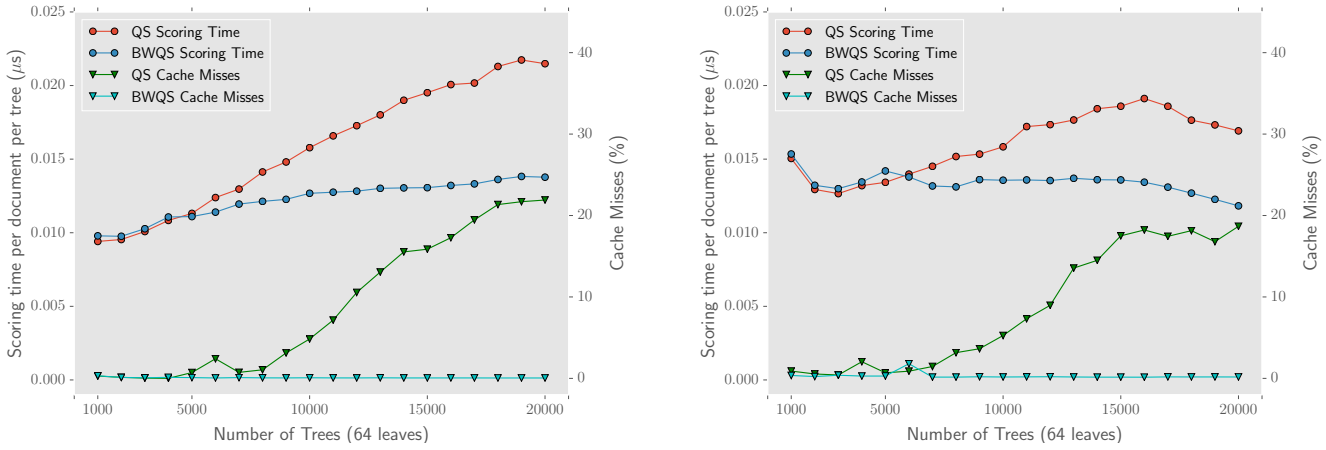
Table 4: Per-document scoring time in μs of BWQS, QS and VPRED algorithms on MSN-1.

Λ	Method	MSN-1			Y!S1		
		Block		Time	Block		Time
		δ	τ		δ	τ	
8	BWQS	8	20,000	**33.5** (–)	8	20,000	**40.5** (–)
	QS	1	20,000	40.5 (1.21x)	1	20,000	48.1 (1.19x)
	VPRED	16	20,000	161.4 (4.82x)	16	20,000	164.8 (4.07x)
16	BWQS	8	5,000	**59.6** (–)	8	10,000	**72.34** (–)
	QS	1	20,000	67.8 (1.14x)	1	20,000	81.0 (1.12x)
	VPRED	16	20,000	336.4 (5.64x)	16	20,000	336.1 (4.65x)
32	BWQS	2	5,000	**135.5** (–)	8	5,000	**141.2** (–)
	QS	1	20,000	155.8 (1.15x)	1	20,000	160.1 (1.13x)
	VPRED	16	20,000	711.9 (5.25x)	16	20,000	694.8 (4.92x)
64	BWQS	1	3,000	**274.7** (–)	1	4,000	**236.0** (–)
	QS	1	20,000	425.1 (1.55x)	1	20,000	343.7 (1.46x)
	VPRED	16	20,000	1309.7 (4.77x)	16	20,000	1420.7 (6.02x)

The reason of the improvements highlighted in the table are apparent from the two plots reported in Figure 4. These plots report for MSN-1 and Y!S1 the per-document and per-tree average scoring time of BWQS and its cache misses ratio. As already mentioned, the plot shows that the average per-document per-tree scoring time of QS is strongly correlated to the cache misses measured. The more the cache misses, the larger the per-tree per-document time needed to apply the model. On the other hand, the BWQS cache misses curve shows that the block-wise implementation incurs in a negligible number of cache misses. This cache-friendliness is directly reflected in the per-document per-tree scoring time, which is only slightly influenced by the number of trees of the ensemble.

5. CONCLUSIONS

We presented a novel algorithm to efficiently score documents by using a machine-learned ranking function modeled by an additive ensemble of regression trees. Our main contribution is a new representation of the tree ensemble based on bitvectors, where the tree traversal, aimed to detect the leaves that contribute to the final scoring of a document, is performed through efficient logical bitwise operations. In addition, the traversal is not performed one tree after another, as one would expect, but it is interleaved, feature by feature, over the whole tree ensemble. Our tests conducted on publicly available LtR datasets confirm unprecedented speedups (up to 6.5x) over the best state-of-the-art competitor. The motivations of the very good performance figures of our QS algorithm are diverse. First, linear arrays are used to store the tree ensemble, while the algorithm exploits cache-friendly access patterns (mainly sequential patterns) to these data structures. Second, the interleaved tree traversal counts on an effective oracle that, with a few branch mis-predictions, is able to detect and return only the internal node in the tree whose conditions evaluate to FALSE. Third, the number of internal nodes visited by QS is in most cases consistently lower than in traditional methods, which recursively visits the small and unbalanced trees of the ensemble from the root to the exit leaf. All these remarks are confirmed by the deep performance assessment conducted by also analyzing low-level CPU hardware counters. This analysis shows that QS exhibits very low cache misses and branch mis-prediction rates, while the instruction count is



(a) MSN-1 (b) Y!S1

Figure 4: Per-tree per-document scoring time in μs and percentage of cache misses of QS and BWQS on MSN-1 (left) and Y!S1 (right) with 64-leaves λ-MART models.

consistently smaller than the counterparts. When the size of the data structures implementing the tree ensemble becomes larger the last level of the cache (L3 in our experimental setting), we observed a slight degradation of performance. To show that our method can be made scalable, we also present BWQS, a block-wise version of QS that splits the sets of feature vectors and trees in disjoint blocks that entirely fit in the cache and can be processed separately. Our experiments show that BWQS performs up to 1.55 times better than the original QS on large tree ensembles.

As future work, we plan to apply the same devised algorithm to other contexts, when a tree-based machine learned model must be applied to big data for classification/prediction purposes. Moreover, we aim at investigating whether we can introduce further optimizations in the algorithms, considering that the same tree-based model is applied to a multitude of feature vectors, and thus we could have the chance of partially reusing some work. Finally, we plan to investigate the parallelization of our method, which can involve various dimensions, i.e., the parallelization of the scoring task of each single feature vector, or the parallelization of the simultaneous scoring of many feature vectors.

Acknowledgements

We acknowledge the support of Tiscali S.p.A. In particular, we wish to warmly thank Domenico Dato and Monica Mori (Istella) for fruitful discussions and valuable feedbacks helping us in concretizing this paper.

6. REFERENCES

[1] N. Asadi, J. Lin, and A. P. de Vries. Runtime optimizations for tree-based machine learning models. *IEEE Trans. Knowl. Data Eng.*, 26(9):2281–2292, 2014.

[2] C. J. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical Report MSR-TR-2010-82, 2010.

[3] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proc. ACM WSDM*, pages 411–420. ACM, 2010.

[4] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

[5] Y. Ganjisaffar, R. Caruana, and C. V. Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. In *Proc. ACM SIGIR*, pages 85–94, New York, NY, USA, 2011. ACM.

[6] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[7] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[8] D. Patterson and J. Hennessy. *Computer Organization and Design (4th ed.)*. Morgan Kaufmann, 2009.

[9] S. Robertson and H. Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, 2009.

[10] I. Segalovich. Machine learning in search quality at yandex. ACM SIGIR, Industry track, 2010.

[11] T. Sharp. Implementing decision trees and forests on a gpu. In *Proc. Computer Vision*, pages 595–608. Springer, 2008.

[12] X. Tang, X. Jin, and T. Yang. Cache-conscious runtime optimization for ranking ensembles. In *Proc. ACM SIGIR*, pages 1123–1126, 2014.

[13] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger. Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In *Proc. IEEE FCCM*, pages 232–239. IEEE, 2012.

[14] P. Viola and M. J. Jones. Robust real-time face detection. *Int. J. Comput. Vision*, 57(2):137–154, 2004.

[15] L. Wang, J. J. Lin, and D. Metzler. Learning to efficiently rank. In *Proc. ACM SIGIR*, pages 138–145, 2010.

[16] L. Wang, J. J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. ACM SIGIR*, pages 105–114, 2011.

[17] L. Wang, D. Metzler, and J. J. Lin. Ranking under temporal constraints. In *Proc. ACM CIKM*, pages 79–88, 2010.

[18] Q. Wu, C. Burges, K. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 2010.

[19] Z. Xu, K. Weinberger, and O. Chapelle. The greedy miser: Learning under test-time budgets. In *Proc. ICML*, pages 1175–1182, New York, NY, USA, 2012. ACM.