# QuickScorer: Efficient Traversal of Large Ensembles of Decision Trees

Claudio Lucchese[1], Franco Maria Nardini[1], Salvatore Orlando[2,1], Raffaele Perego[1], Nicola Tonellotto[1], and Rossano Venturini[3,1]

[1] ISTI–CNR, Italy, `name.surname@isti.cnr.it`
[2] Ca' Foscari Univ. of Venice, Italy, `orlando@unive.it`
[3] Univ. of Pisa, Italy, `rossano.venturini@unipi.it`

**Abstract.** Machine-learnt models based on additive ensembles of binary regression trees are currently deemed the best solution to address complex classification, regression, and ranking tasks. Evaluating these models is a computationally demanding task as it needs to traverse thousands of trees with hundreds of nodes each. The cost of traversing such large forests of trees significantly impacts their application to big and stream input data, when the time budget available for each prediction is limited to guarantee a given processing throughput. Document ranking in Web search is a typical example of this challenging scenario, where the exploitation of tree-based models to score query-document pairs, and finally rank lists of documents for each incoming query, is the state-of-art method for ranking (a.k.a. *Learning-to-Rank*). This paper presents QuickScorer, a novel algorithm for the traversal of huge decision trees ensembles that, thanks to a cache- and CPU-aware design, provides a $\sim 9\times$ speedup over best competitors.

**Keywords:** Learning to Rank, Ensemble of Decision Trees, Efficiency.

## 1 Introduction

In this paper we discuss QuickScorer (QS), an algorithm developed to speedup the application of machine-learnt forests of binary regression trees to score and finally rank lists of candidate documents for each query submitted to a Web search engine. QuickScorer was thus developed in the field of *Learning-to-Rank* (`LtR`) within the IR community. Nowadays, `LtR` is commonly exploited by Web search engines within their query processing pipeline, by exploiting massive training datasets consisting of collections of query-document pairs, in turn modeled as vectors of hundreds features, annotated with a relevance label.

The interest in exploiting forests of binary regression trees to rank lists of candidate documents is due to the success of gradient boosting tree algorithms [4]. This kind of algorithms is considered the state-of-the-art `LtR` solution for addressing complex ranking problems [5]. In search engines, these forests are exploited within a two-stage architecture. While the first stage retrieves a set of possibly relevant documents matching the user query, such expensive `LtR`-based scorers, optimized for high precision, are exploited in the second stage to *re-rank* the set of candidate documents coming from the first stage. 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 response time. Therefore, devising techniques and strategies to speed up document ranking without losing in quality is definitely an urgent research topic in Web search [9].

Strongly motivated by these considerations, the IR community has started to investigate computational optimizations to reduce the scoring time of the most effective `LtR` rankers based on ensembles of regression trees, by exploiting advanced features of modern CPUs and carefully exploiting memory hierarchies. Among those, the best competitor of QuickScorer is vPRED [1].

We argue that QuickScorer can also be exploited in different *time-sensitive* scenarios and each time it is needed to use a large forest of binary decision trees, e.g., random forest, for classification/regression purposes and apply it to big and stream data with strict processing throughput requirements.

## 2 QuickScorer

Given a query-document pair $(q, d_i)$, represented by a feature vector $\mathbf{x}$, a `LtR` model based on an additive ensemble of regression trees predicts a relevance score $s(\mathbf{x})$ used for ranking a set of documents. Typically, a tree ensemble encompasses several binary decision trees, denoted by $\mathcal{T} = \{T_0, T_1, \ldots\}$. Each internal (or branching) node in $T_h$ is associated with a Boolean test over a specific feature $f_\phi \in \mathcal{F}$, and a constant threshold $\gamma \in \mathbb{R}$. Tests are of the form $\mathbf{x}[\phi] \leq \gamma$, and, during the visit, the left branch is taken *iff* the test succeeds. Each leaf node stores the tree prediction, representing the potential contribution of the tree to the final document score. The scoring of $\mathbf{x}$ requires the traversal of all the ensemble's trees and it is computed as a *weighted sum* of all the tree predictions.

Algorithm 1 illustrates QS [7,3]. One important result is that QS computes $s(\mathbf{x})$ by only identifying the branching nodes whose test evaluates to false, called *false nodes*. For each false node detected in $T_h \in \mathcal{T}$, QS updates a bitvector associated with $T_h$, which stores information that is eventually exploited to identify the *exit leaf* of $T_h$ that contributes to the final score $s(\mathbf{x})$. To this end, QS maintains for each tree $T_h \in \mathcal{T}$ a bitvector `leafidx[`$h$`]`, made of $\Lambda$ bits, one per leaf. Initially, every bit in `leafidx[`$h$`]` is set to 1. Moreover, each branching node is associated with a bitvector `mask`, still of $\Lambda$ bits, identifying the set of unreachable leaves of $T_h$ in case the corresponding test evaluates to false. Whenever a false node is visited, the set of unreachable leaves `leafidx[`$h$`]` is updated through a *logical AND* ($\wedge$) with `mask`. Eventually, the leftmost bit set in `leafidx[`$h$`]` identifies the leaf corresponding to the score contribution of $T_h$, stored in the lookup table `leafvalues`.

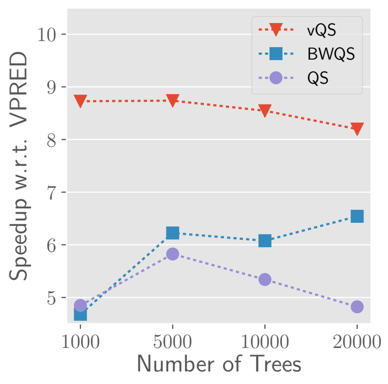
To efficiently identify all the *false nodes* in the ensemble, QS processes the branching nodes of all the trees *feature by feature*. Specifically, for each feature $f_\phi$, QS builds a list $\mathcal{N}_\phi$ of tuples $(\gamma, \mathtt{mask}, h)$, where $\gamma$ is the predicate threshold of a branching node of tree $T_h$ performing a test over the feature $f_\phi$, denoted by

$\mathbf{x}[\phi]$, and `mask` is the pre-computed mask that identifies the leaves of $T_h$ that are un-reachable when the associated test evaluates to false. $\mathcal{N}_\phi$ is statically sorted in ascending order of $\gamma$. Hence, when processing $\mathcal{N}_\phi$ sequentially, as soon as a test evaluates to true, i.e., $\mathbf{x}[\phi] \leq \gamma$, the remaining occurrences surely evaluate to true as well, and their evaluation is thus safely skipped.

**Algorithm 1:** QuickScorer

```
QuickScorer($\mathbf{x}$,$\mathcal{T}$):
    foreach $T_h \in \mathcal{T}$ do
        leafidx[$h$]$\leftarrow$ 11...11
    foreach $f_\phi \in \mathcal{F}$ do                // Mask Computation
        foreach $(\gamma, \texttt{mask}, h) \in \mathcal{N}_\phi$ do
            if $\mathbf{x}[\phi] > \gamma$ then
                leafidx[$h$]$\leftarrow$ leafidx[$h$] $\wedge$ mask
            else
                break
    $score \leftarrow 0$     // Score Computation
    foreach $T_h \in \mathcal{T}$ do
        $j \leftarrow$ leftmost bit set in leafidx[$h$]
        $l \leftarrow h \cdot \Lambda + j$
        $score \leftarrow score +$ leafvalues[$l$]
    return $score$
```

Fig. 1: QS performance.



We call *mask computation* the first step of the algorithm during which all the bitvectors `leafidx[`$h$`]` are updated, and *score computation* the second step where such bitvectors are used to retrieve tree predictions.

Compared to the classic tree traversal, QuickScorer introduces a main novelty. The cost of the traversal does not depend on the average length of the root-to-leaf paths, but rather on the average number of false nodes in the trees of the forest. Experiments on large public datasets with large forests, with 64 leaves per tree and up to 20,000 trees, show that a classic traversal evaluates between 50% and 80% of the branching nodes. This is due to the imbalance of the trees built by state-of-the-art `LtR` algorithms. On the other hand, on the same datasets, QuickScorer always visits less than 30% of the nodes. This results in a largely reduced number of operations and number of memory accesses.

Moreover, QuickScorer exploits a cache- and CPU-aware design. For instance, the values of $(\gamma, \texttt{mask}, h)$ are accessed through a linear scan of the QuickScorer data structures, which favours cache prefetching and limits data dependencies. For each feature, QuickScorer visits only one true node, thus easing the CPU branch predictor and limiting control dependencies. This makes QuickScorer to perform better than competitors also with a special kind of perfectly balanced trees named *oblivious* [6].

The design of QuickScorer makes it possible to introduce two further improvements. Firstly, for large `LtR` models, the forest can be split into multiple *blocks* of trees, sufficiently small to allow the data structure of a single block to entirely fit into the third-level CPU cache. We name BlockWise-QS (BWQS) the resulting variant. This cache-aware algorithm reduces the cache miss ratio from more than 10% to less than 1%. Secondly, the scoring can be vectorized so as to score multiple documents simultaneously. In V-QuickScorer (vQS) [8] vectorization is achieved through AVX 2.0 instructions and 256-bits wide registers. In such setting, up to 8 documents can be processed simultaneously.

Figure 1 compares QS, BWQS, and vQS against the best competitor vPRED. The test was performed on a large dataset, with a model with 64 leaves per tree and varying the number of trees of the forest.

## 3 Discussion

In this work, we focused on tree ensembles to tackle the `LtR` problem. Decision tree ensembles are a popular and effective machine learning tool beyond `LtR`. Their success is witnessed by the *Kaggle 2015* competitions, where most of the winning solutions exploited MART models, and by the *KDD Cup 2015*, where MART-based algorithms were used by all the top 10 teams [2].

In the `LtR` scenario, the time budget available for applying a model is limited and must be satisfied. Therefore large models, despite being more accurate, cannot be used because of their high evaluation cost. QS, a novel algorithm for the traversal of decision trees ensembles, is an answer to this problem as it provides $\sim 9\times$ speedup over state-of-the-art competitors. Moreover, the need of efficient traversal strategies goes beyond the `LtR` scenario, for instance when such models are used to classify big data collections. For all these reasons, we believe that QS can help scientists from the data mining community to speed-up the process of evaluating highly effective tree-based models over big and stream datasets.

## References

1. Asadi, N., Lin, J., de Vries, A.P.: Runtime optimizations for tree-based machine learning models. IEEE TKDE. 26(9), 2281–2292 (2014)
2. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proc. SIGKDD. pp. 785–794. ACM (2016)
3. Dato, D., Lucchese, C., Nardini, F.M., Orlando, S., Perego, R., Tonellotto, N., Venturini, R.: Fast ranking with additive ensembles of oblivious and non-oblivious regression trees. ACM TOIS 35(2), 15:1–15:31 (2016)
4. Friedman, J.H.: Greedy function approximation: A gradient boosting machine. Annals of Statistics 29, 1189–1232 (2000)
5. Gulin, A., Kuralenok, I., Pavlov, D.: Winning The Transfer Learning Track of Yahoo!'s Learning To Rank Challenge with YetiRank. In: Yahoo! Learning to Rank Challenge. pp. 63–76 (2011)
6. Langley, P., Sage, S.: Oblivious decision trees and abstract cases. In: Working Notes of the AAAI-94 Work. on Case-Based Reasoning. pp. 113–117. AAAI Press (1994)
7. Lucchese, C., Nardini, F.M., Orlando, S., Perego, R., Tonellotto, N., Venturini, R.: QuickScorer: A Fast Algorithm to Rank Documents with Additive Ensembles of Regression Trees. In: Proc. SIGIR. pp. 73–82. ACM (2015)
8. Lucchese, C., Nardini, F.M., Orlando, S., Perego, R., Tonellotto, N., Venturini, R.: Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles. In: Proc. SIGIR. pp. 833–836. ACM (2016)

9. Segalovich, I.: Machine learning in search quality at Yandex. Presentation at the Industry Track of SIGIR. (2010)