# Quality versus Efficiency in Document Scoring with Learning-to-Rank Models

Gabriele Capannini[a], Claudio Lucchese[b], Franco Maria Nardini[b], Salvatore Orlando[c], Raffaele Perego[b], Nicola Tonellotto[b]

[a]*IDT, Mälardalens högskola, Västerås, Sweden*
[b]*"Istituto di Scienza e Tecnologie dell'Informazione" (ISTI) of the National Research Council of Italy (CNR), Pisa, Italy*
[c]*University Ca' Foscari of Venice, Italy*

**Abstract**

Learning-to-Rank (LtR) techniques leverage machine learning algorithms and large amounts of training data to induce high-quality ranking functions. Given a set of documents and a user query, these functions are able to precisely predict a score for each of the documents, in turn exploited to effectively rank them. Although the scoring efficiency of LtR models is critical in several applications – e.g., it directly impacts on response time and throughput of Web query processing – it has received relatively little attention so far.

The goal of this work is to experimentally investigate the scoring efficiency of LtR models along with their ranking quality. Specifically, we show that machine-learned ranking models exhibit a quality versus efficiency trade-off. For example, each family of LtR algorithms has tuning parameters that can influence both effectiveness and efficiency, where higher ranking quality is generally obtained with more complex and expensive models. Moreover, LtR algorithms that learn complex models, such as those based on forests of regression trees, are generally more expensive and more effective than other algorithms that induce simpler models like linear combination of features.

We extensively analyze the quality versus efficiency trade-off of a wide spectrum of state-of-the-art LtR, and we propose a sound methodology to devise the most effective ranker given a time budget. To guarantee reproducibility, we used publicly available datasets and we contribute an open source C++ framework providing optimized, multi-threaded implementations of the most effective tree-based learners: Gradient Boosted Regression Trees (GBRT), Lambda-Mart ($\lambda$-MART), and the first public-domain implementation of Oblivious Lambda-Mart ($\Omega_\lambda$-MART), an algorithm that induces forests of oblivious regression trees.

We investigate how the different training parameters impact on the quality versus efficiency trade-off, and provide a thorough comparison of several algorithms in the quality-cost space. The experiments conducted show that there is not an overall best algorithm, but the optimal choice depends on the time budget.

*Keywords:* Efficiency, learning to rank, document scoring

## 1. Introduction

Ranking is a central task of many information retrieval problems, in particular for document retrieval where documents must be ranked according to their relevance to a user query. Indeed, ranking is particularly challenging for large-scale Web Search Engines (WSEs), since it involves effectiveness requirements and efficiency constraints that are not common to other ranking-based applications.

From an effectiveness point of view, a number of machine learning algorithms have been proposed to automatically build high-quality ranking functions able to exploit a multitude of features characterizing the candidate documents and the user query. These algorithms fall under the Learning-to-Rank (LtR) framework [1]. The *models*, or *rankers*, generated by such methods are generally quite expensive to use for ranking large sets of documents. For example, methods based on forests of regression trees may generate thousands of trees to be evaluated on hundreds of features modeling a single query-document pair, in order to predict scores used to effectively rank all the candidate documents for a given query [2, 3]. Therefore, even if LtR models are able to provide high quality results, it is not possible to apply such rankers to all the documents matching a user query due to the resulting prohibitive ranking cost.

To overcome this issue, WSEs usually exploit multi-stage ranking architectures (Figure 1), where top-$K$ retrieval is carried out by a two-step process: (i) candidate retrieval and (ii) candidate re-ranking. The first step retrieves from the inverted index $N$ possibly relevant documents matching the user query, where $N \gg K$. This phase aims at optimizing the recall of the retrieval system, and is usually achieved by a simple and fast *base ranker*, e.g., BM25 combined with some document-level scores [4]. The assumption is that the base ranker is able to retrieve a large part of the most relevant documents, even it is not able to effectively rank them. In the second step, a complex scoring function is used by the *top ranker* to re-rank the candidate documents coming from the first step. The top ranker is optimized for high precision, i.e., to place the most relevant results in the top positions of the first page of results. LtR models are commonly used in this second step to achieve the desired precision of the top ranker.

Therefore, the top ranker is a crucial component for both effectiveness and efficiency of WSEs. First, the top ranker determines the quality of the results presented to the user. Second, it impacts on the response time of the WSE. We know that both quality and response time largely impact on the click behavior of users [5], and ultimately on the user satisfaction and WSE revenue. Devising a good trade-off between efficiency and effectiveness is thus very important.

This paper investigates such effectiveness vs. efficiency trade-offs. We believe that the problem of devising the right trade-off between the quality and the computational cost of LtR models at query processing time has not yet received enough attention from the Machine Learning (ML) and Information Retrieval (IR)

*Email addresses:* gabriele.capannini@mdh.se (Gabriele Capannini), c.lucchese@isti.cnr.it (Claudio Lucchese), f.nardini@isti.cnr.it (Franco Maria Nardini), orlando@unive.it (Salvatore Orlando), r.perego@isti.cnr.it (Raffaele Perego), n.tonellotto@isti.cnr.it (Nicola Tonellotto)

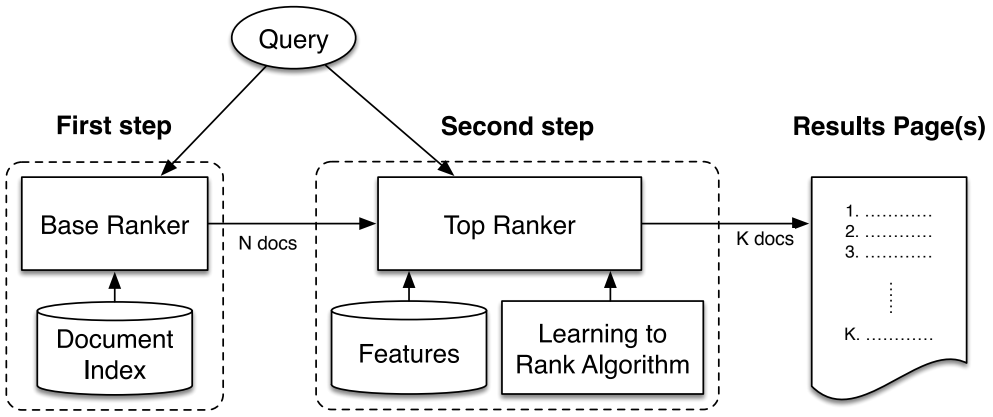

Figure 1: The architecture of a generic Machine-Learned Ranking pipeline.

communities. Traditionally, the ML community focused primarily on the accuracy of the learned model, or the scalability of the training phase, while the efficiency of the application of the learned model was considered as unimportant or negligible. On the other hand, strongly motivated by budget considerations that are very important for commercial WSEs, the IR community has started only recently to investigate low-level optimizations to reduce the execution time of some families of LtR rankers. The computational cost of the LtR models must be in fact strictly accounted in the time budget available for processing queries in the incoming stream, as it can impact to a large extent on the throughput of the system. In addition, since each family of algorithms has tuning parameters that can influence both effectiveness and efficiency (e.g., number of trees in tree-based models), even for a given family of algorithms a change in the setting of the parameters can have a deep impact on the performance of the learned model.

*Contributions and Research Questions*

To the best of our knowledge, this is the first work that compares a wide spectrum of LtR approaches through extensive experiments thus providing a comprehensive perspective on effectiveness vs. efficiency trade-offs offered by the different families of rankers. The main contributions of our work are:

- QuickRank[1], a public-domain framework for evaluating the efficiency at scoring time of the various LtR models. It is written in C++ and it allows a fair performance comparison of different ranking models.
- QuickRank includes the C++ multithreaded implementations of the most effective state-of-the-art LtR algorithms whose learned models are forests of additive regression trees: Gradient Boosted Regression Trees (GBRT) [6], Lambda-Mart ($\lambda$-MART) [7], and Oblivious Lambda-Mart ($\Omega_\lambda$-MART) [2]. We remark that no public-domain implementation of the $\Omega_\lambda$-MART solution, which generates forests of oblivious trees, was previously available;

[1]The source code of QuickRank is available under Reciprocal Public License 1.5 at `http://quickrank.isti.cnr.it/`

- an extensive and reproducible experimental analysis of efficiency/effectiveness trade-offs offered by nine different LtR approaches, conducted within the QuickRank framework using well-known IR metrics (NDCG[8]) and publicly available datasets;
- a new measure, named *AuQC*, aimed at estimating to which degree a given model can be tuned to provide high-quality rankers for any given time budget.

The experimental results of this work allow to answer the following research questions which are crucial in designing a machine-learned ranking pipeline for a large-scale search system:

- **Q1:** How does the effectiveness of the top ranker trained with a given LtR model varies with its computation cost at testing time?
- **Q2:** Given a time budget, what are the LtR model and the associated tuning parameters providing the ranker with the best quality at a cost not greater than the budget?
- **Q3:** How can we characterise the quality versus cost trade-off of a given LtR model?

The rest of the paper is structured as follows: Section 2 presents the related work. Section 3 details the experimental methodology to evaluate quality/cost trade-offs of LtR models and introduces the QuickRank framework. Then, Section 4 discusses the different LtR models and how we model their cost, while Section 5 reports on the results of our comprehensive evaluation. Finally, we conclude our investigation in Section 6.

## 2. Related Work

LtR models are usually classified in three broad categories: point-wise, pair-wise and list-wise [1]. Point-wise methods are regression or classification algorithms aiming at predicting the relevance label associated to each query-document pair in the training set. Pair-wise methods consider pairs of documents as a training instances, and they explore scoring functions that are able to discriminate the best document among the two. The learning process for point-wise models tries to optimize loss functions such as root mean squared error (RMSE) with respect to documents relevance labels, while pair-wise methods typically optimise the number of misclassified pairs. IR quality measures that are a function of the full set of results for a given query cannot be directly optimised by the above two approaches. To this end, list-wise methods have been introduced to directly optimise list-based metrics such as NDCG [9].

In this work we analyze trade-off properties of LtR algorithms belonging to all of the above categories. The considered algorithms are discussed in some detail in Section 4. However, our comparison does not aim at finding the algorithm achieving the best quality, or at assessing what is the best class of algorithms. Our analysis is cost-driven, and thus we classify the discussed algorithms into cost-driven categories: linear combination of features, artificial neural networks, forests of regression trees, and forests of oblivious trees.

Tax *et al.* propose a comparison of several LtR algorithms [10] by focusing on quality issues and without taking into consideration efficiency. A first step toward efficiency analysis is discussed in [11]. Here, the authors analyse the behaviour of different algorithms when varying the number of documents filtered by the *base ranker* of a two-stage ranking architecture. Time-efficiency of the models is not taken into consideration, but the authors show that a larger set of candidate documents can provide significantly better results. A straightforward implication of this result is that the *top ranker* should be sufficiently fast to process a large number of candidates. This work investigates the performance of the top ranker, that is assumed to include as input all relevant documents, i.e., the sample size provided to the second step ranker has maximum recall.

Schenkel *et al.* [12] discusses the concept of budget-aware learning, discussing an approach that limits the cost for evaluating a ranking model. They propose to only partially evaluate parts of the ranking model for the most promising documents by pruning posting lists to generate an execution plan before the actual query processing starts. In doing so, they assume that the LtR model is deployed as a base ranker, and that it needs to directly access posting lists. Instead, our work focuses on the deployment of the LtR algorithm as a top ranker, with no access to any index, but receiving a limited set of top documents from the base ranker. To the best of our knowledge, this is the first experimental analysis of LtR algorithms in a two-stage ranking architecture focused on the efficiency vs. effectiveness trade-off.

The efficiency of machine-learned ranking pipelines recently attracted increasing interest. Many researchers have examined the impact of several factors on the whole ranking pipeline. For example, Macdonald *et al.* [11] assessed the impact of the number of candidate documents and the objective metric to use when training the learning-to-rank model. Asadi *et al.* [13] and Tonellotto *et al.* [14] have specifically looked efficiency/effeciveness tradeoff of the candidate generation between the two phases by exploiting approximate techniques and dynamic pruning, respectively. The separation of the scoring process in these two phases has the advantage of providing better control over cost/quality trade-offs. The second phase of the scoring process is the fastest part of the ranking process [14]. A state-of-the-art processing in the first phase [15] employs $\sim 20$ ms to rank 10 documents, i.e., $\sim 2$ ms per document, while a state-of-the-art processing in the second phase [16] employs only $\sim 0.05$ ms per document. The time savings in the second phase can be exploited to retrieve a larger number of document or to compute expensive features such as term proximity or other bigram features to the candidates selected by the base ranker [17].

Most of the work published so far focused primarily on the optimization techniques that can be adopted to make the top ranker faster. Cambazoglu *et al.* [3], propose to use early exits in additive ensemble ranking pipelines to early terminate the scoring of documents that are unlikely to be ranked within the top-$K$ results. Wang *et al.* [18, 19, 20] deeply investigate different efficiency aspects of the ranking pipeline. In particular, in [20] authors propose a novel cascade ranking model that, unlike previous approaches, can simultaneously improve both effectiveness and efficiency. The model constructs a cascade of increasingly complex scorers that progressively prunes and refines the set of candidate documents to minimize scoring time and maximize result set quality. The authors present a novel boosting algorithm for learning such cascades by directly optimizing the trade-off between effectiveness and efficiency. Experimental results show that the proposed cascades are faster and return higher quality results than comparable ranking models. Asadi *et al.* [21, 22] also propose runtime optimizations for tree-based machine learning models. The authors focus on engineering a ranking pipeline that exploits cache and branch prediction features of modern processor architectures. The approach is based on training compact and balanced gradient boosted regression trees that yield efficient runtime and effective branch prediction. The experimental assessment show significant performance improvements over standard implementations. Tang *et al.* [23] discuss a simple cache-conscious 2D block-wise approach for traversing large ensemble of additive scorers achieving better cache utilization. Both the set of candidate documents and the set of scorers are partitioned so that a subset of documents and one of scorers can fit in the processor cache. The execution of each set of scorers on each set of candidates is orchestrated in a parallelizable nested loop achieving significant speedup without loss of accuracy.

The state-of-the-art algorithm for the evaluation of a forest of regression tree is provided by the QuickScorer algorithm, proposed by Lucchese *et al.* [16]. QuickScorer traverses tree-based models with an innovative strategy coupled with a new organization of the data able to reduce cache misses and branch mispredictions, providing 2x to 6x speed-up over [22].

The approaches discussed above are orthogonal to our as they address computational cost issues by plugging efficiency in the learned models or in their run-time behavior. We instead define a general framework for analyzing the cost-efficiency trade-off of a given ranker, and compare on a fair basis models belonging to different families of techniques. All the above optimizations can be easily accommodated in our framework. Moreover, unlike previous works, our study allows the different ranking models to be evaluated under time budget constraints.

The results of our investigation can be straightforwardly exploited in any machine-learned ranking pipeline as the one sketched in Fig. 1, where the time budget for ranking candidate documents is fixed to fulfill a constraint on the total query response time. However we can imagine even a more advanced two-stage ranking pipeline where a maximum time budget is still given but: (i) the first stage takes a variable amount of time to process a query, depending on the query and collection characteristics [14]; (ii) the time budget left for the second stage is computed on a per-query basis as the difference between the total budget and the time actually spent in the first stage. In this scenario, our analysis can be exploited to devise a set of rankers that behave "optimally" for different time budgets, and the ranker that best fits this query-specific time budget selected for each query. Similarly, this approach may address also distributed search engines where incoming queries are stored in a global queue waiting to be processed, and per-query time budgets can be dynamically established on the basis of the query scheduling algorithm adopted and the time spent by the query in the queue [24].

## 3. Evaluation methodology

In this section, we examine the overall evaluation methodology used to quantitatively compare the various LtR-based top rankers in the *quality vs. cost space*, where the cost is the scoring efficiency of a ranker, and the quality is the effectiveness measured on a test set with a standard IR quality measure.

First we discuss the measures devised to explore the behavior of the various scorers with respect to their quality/cost trade-off, and answer our three research questions. Among the various measures, it is worth remarking the use of a novel measure, named $AuQC$, inspired by the area under the ROC curve index adopted for comparing binary classifiers. We use $AuQC$ to estimate to which degree a given ranker $R$ can be tuned to provide good results for any given time budget $B$. Second, we discuss QuickRank, an optimized framework implementing the most effective LtR algorithms at the state of the art. In particular QuickRank provides multithreaded C++ implementations of GBRT, $\lambda$-MART, and $\Omega_\lambda$-MART, and allows to easily generate the scoring code exploiting models learned with these and other LtR algorithms in order to fairly measure their computational cost. Third, we present the datasets adopted for training the models and testing their effectiveness, and discuss the experimental settings.

### *3.1. Evaluating Quality vs. Cost*

Top rankers based on different LtR algorithms may differ significantly in the time taken to score the same set of candidates retrieved by the base ranker: computing a simple linear combination of features is trivially far less expensive than traversing a forest of 10,000 regression trees.

Hereinafter, given a top ranker $R$, we refer to the *cost* $C(R)$ as the average time in $\mu$s required to score a candidate document represented by a given set of features. We are interested in analysing rankers having a cost $C(R)$ smaller than a maximum *time budget* $B$. Also, we refer to the *quality* of $R$ as the average effectiveness measured on a test set according to standard IR quality measures. Since we focus on WSE systems, in this paper we consider NDCG@10 [8] as the metric used to assess the effectiveness of the system. We denote the *quality* of $R$ by $Q(R)$, $0 \leq Q(R) \leq 1$. The following analysis is however completely general and easily adaptable to different metrics and cut-offs.

Most LtR algorithms have tuning parameters that impact on the complexity of the produced rankers, and therefore on $Q$ and $C$. This is for example the case of tree-based algorithms, where the maximum size of a tree and the total number of trees can be chosen at training time. We denote by $\theta \in \Theta$ the set of parameters in the parameter space $\Theta$ used during the learning phase of a given LtR algorithm, and by $R_\theta$ the resulting ranker.

We now discuss the methodology to answer the questions **Q1**, **Q2**, and **Q3**, introduced in Section 1.

To answer **Q1**, i.e., how the effectiveness of the top ranker trained with a given LtR algorithm varies with its computation cost at testing time, we comprehensively evaluate different LtR algorithms, by sweeping their parameters in order to build different instances of top rankers $R_\theta$. For each ranker $R_\theta$ we measure its quality $Q(R_\theta)$ on a test set in terms of NDCG@10 [8]. We also determine its cost $C(R_\theta)$, by measuring the average time (in $\mu$sec) required by our QuickRank-based deployment of the specific ranker to score a candidate document. In order to fairly compare the different rankers, these are translated by QuickRank into C++ implementations with the same degree of code-level optimizations. We then provide a scatter plot, named *QC-plot*, where each point $P(R_\theta)$, corresponding to a specific ranker $R_\theta$, has coordinates $(C(R_\theta), Q(R_\theta))$ in the quality vs. cost space, for every parameter vector $\theta$ being tested.

The *QC-plot* provides a global view of the algorithm behaviour on varying $\theta$, and it helps in answering the above three questions. However, most of the points in the *QC-plot* are irrelevant because they are *dominated both in quality and cost* by others. For example, suppose two models $R_\theta$ and $R_{\theta'}$ have been learned by the same LtR algorithm by varying its parameters. If $R_\theta$ has a smaller cost and a better quality than $R_{\theta'}$ then we can discard $P(R_{\theta'})$. We say that $R_\theta$ *dominates* $R_{\theta'}$ if $C(R_{\theta'}) \geq C(R_\theta)$ and $Q(R_{\theta'}) \leq Q(R_\theta)$.

In order to focus on the dominant points only, we thus introduce the concept of *QC-curve*, which contains all the dominant points of the *QC-plot* of a given LtR algorithm $R$. The *QC-curve* of a LtR algorithm $R$ is defined by the following function in the Quality-Cost space:

$$QC_R(B) = \max_{\theta \in \Theta \ | \ C(R_\theta) \leq B} Q(R_\theta) \tag{1}$$

where $B$ is a time budget. Given a time budget $B$, the function thus returns the quality of the most effective ranker the given LtR algorithm can provide. We assume that for every LtR algorithm there exists $\theta \in \Theta$ such that $C(R_\theta) = 0$ and $Q(R_\theta) = 0$, meaning that quality cannot be achieved without paying a cost. This also implies that the function is defined for all budgets $B$. Note that the *QC-curve* is monotonically increasing.

By comparing the *QC-curve* of two different LtR algorithms we can identify which is best for a given budget $B$. The *QC-curve* is thus able to answer question **Q2**, i.e., given a time budget, what are the LtR algorithm and the associated tuning parameters providing the top ranker with the best quality at a cost smaller than the budget.

Finally, we introduce the *area under QC-curve*, or *AuQC*, as follows:

$$AuQC_R(B) = \frac{1}{B} \int_0^B QC_R(x) \, \mathrm{d}x \tag{2}$$

which is computed up to a given time budget $B$. Note that function $QC_R(x)$ ranges in the interval [0,1], so that the maximum value of the above integral is $B$, normalized in the equation above by $\frac{1}{B}$. Therefore, $0 \leq AuQC_R(B) \leq 1$.

*AuQC* measures how well a given LtR algorithm $R$ can be tuned by varying the time budget, from 0 to $B$. High values of $AuQC_R(B)$ imply that we can learn instances of the ranker $R$ ($R_\theta$) with high quality for any time budget smaller than $B$. The ideal case, i.e., $AuQC_R(B) = 1$, corresponds to a ranker achieving NDCG@10 $= 1$ for any given time budget in $[0, B]$. We exploit this single measure $AuQC$ to answer the third question **Q3**, i.e., how can we characterise the quality versus cost tradeoff of a given LtR algorithm.

Compared to related measures, such as MEET [18], $AuQC$ is better suited to measure the quality of a LtR algorithm across the set of its training parameters and across the efficiency spectrum of the models it can generate.

## *3.2. Efficient Scorers and QuickRank*

To apply the evaluation methodology discussed above, for each learned model corresponding to a top ranker $R$, we compile the model and generate an optimized code, whose running time $C(R)$ is finally collected. Specifically, we measure the average time in $\mu$s taken by $R$ to score a document, given its feature-based representation. The efficiency analysis is carried out by means of QuickRank, a novel C++ framework allowing the computational cost of rankers exploiting different learned models to be fairly compared.[2]

Besides the scoring framework, QuickRank also provides efficient, multithreaded implementations of state-of-the-art LtR learning algorithms that induce complex tree-based models, namely GBRT [6], $\lambda$-MART [7], and $\Omega_\lambda$-MART [2]. In particular the regression trees produced by $\Omega_\lambda$-MART are *oblivious*, so that the same splitting predicate, i.e., the same test *"feature is less than threshold"*, is applied by all the nodes at the same level of each tree, which is also balanced. It is worth mentioning that no implementation of the $\Omega_\lambda$-MART algorithm was previously publicly available. For all these algorithms, QuickRank accepts training sets in the SVM-light format (as in [25]) and produces a ranking model in an XML format.[3]

All the ranking models learned by the algorithms discussed in Section 4 have a C++ plugin within the QuickRank framework. These plugin read the input feature vectors of the candidate documents, store them in dense floating-point arrays, use feature ids as indexes to directly access feature values, while the learned models are compiled as in-memory data structures that can be accessed very efficiently. Some of the algorithms discussed in Section 4 generate scorers that are simple linear combinations of input features. The plugins for these ranking models turn out to be very simple and inexpensive to use at run time. On the other hand, QuickRank provides plugins also for ranking models based on *forests of regression trees*, that provide state-of-the-art ranking effectiveness but involves computationally expensive scoring functions.

The QuickRank C++ plugin that scores tree-based models implements a state-of-the-art strategy, namely QuickScorer[4], proposed by Lucchese *et al.* [16]. Given a query-document pair, 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

[2]Publicly available at `http://quickrank.isti.cnr.it`

[3]To allow repeatability of experiments, we made all the learned model in XML format publicly available at `http://quickrank.isti.cnr.it/ipm-submission/ipm-ltr-submission.tar.gz`

[4]Publicly available at `https://github.com/hpclab/quickscorer`

**Algorithm 1:** The QuickScorer Algorithm

```
QuickScorer(x,T):
1   foreach $T_h \in \mathcal{T}$ do
2       leafindexes[$h$] ← 11...11
3   foreach $f_\phi \in \mathcal{F}$ do
4       foreach $(\gamma, h, n) \in \mathcal{N}_\phi$ in ascending order do
5           if $\mathbf{x}[\phi] > \gamma$ then
6               leafindexes[$h$] ← leafindexes[$h$] $\wedge$ nodemasks[$n$]
            else
                break
7   $score \leftarrow 0$
8   foreach $T_h \in \mathcal{T}$ do
9       $j \leftarrow$ index of leftmost bit set to 1 of leafindexes[$h$]
10      $l \leftarrow h \cdot \Lambda + j$
11      $score \leftarrow score +$ leafvalues[$l$]
12  return $score$
```

by $\mathcal{T} = \{T_0, T_1, \ldots\}$. Each interal (or branching) node $n \in T_h$ 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 tree predictions.

QuickScorer scores tree-based models with an innovative strategy coupled with a new organization of the data to reduce the cache misses and the branch misprediction. Algorithm 1 illustrates QuickScorer. One important result of QuickScorer is that to compute $s(\mathbf{x})$ it needs to identify only the branching nodes whose tests evaluate to false, called *false nodes*. To do so, QuickScorer maintains for each tree $T_h \in \mathcal{T}$ a bitvector `leafindexes[`$h$`]`, made of $\Lambda$ bits, one per leaf. Initially, every bit in `leafindexes[`$h$`]` is set to 1. Moreover, each branching node $n$ is associated with a binary mask `nodemasks[`$n$`]` idendifying the set of unreachable leaves in case the corresponding test evaluates to false. Whenever a false node is visited, the set of unreachable leaves `leafindexes[`$h$`]` is updated through a *logical and* with `nodemasks[`$n$`]`. Eventually, the leftmost bit set in `leafindexes[`$h$`]` 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, QuickScorer processes the branching nodes of all the trees *feature by feature* and in ascending order of their predicate thresholds. Specifically, for each feature $f_\phi$, QuickScorer builds a list $\mathcal{N}_\phi$ of tuples $(\gamma, h, n)$, where $\gamma$ is the predicate threshold of node $n$ occurring in tree $T_h$. When processing $\mathcal{N}_\phi$ in ascending order by $\gamma$, 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.

### 3.3. Datasets and Experimental Settings

Experiments are conducted by using publicly available LtR datasets: the MSN Learning to Rank[5] and the Yahoo! Learn to Rank Challenge version 2.0[6].

The first one consists of vectors of 136 features extracted from query-url pairs, while the second one consists of two distinct datasets (Y!S1 and Y!S2), made up of vectors of 700 features. Query-url pairs of all the three 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 dataset consists of 6,000, 2,000, and 2,000 queries for training, validation and testing respectively. In addition, the Y!S1 dataset consists of 19,944 training queries, 2,994 validation queries and 6,983 test queries while Y!S2 is of a smaller size: it contains 1,266 training queries, 1,266 validation queries and 3,798 test queries.

Without loss of generality, the quality of a ranker $Q(R_\theta)$ is measured by computing its NDCG@10. Note that LtR algorithms either optimize the root mean squared error w.r.t. documents' relevance labels (e.g., GBRT), or they optimize a proxy function of NDCG@10 (e.g., $\lambda$-MART). we highlight that the proposed framework could be used to evaluate a different measure of interest.

Finally, to measure the costs $C(R_\theta)$ of each top ranker analyzed, we run 3 times the scorers produced on the test sets of the three datasets. We then compute the average per-document scoring cost. The tests were performed on a machine equipped with an Intel Xeon CPU E5-2630 v3 clocked at 2.40GHz with 20 MB of cache L3 and 64GB RAM. We use the GCC 5.3.0 compiler with the highest optimization settings.

## 4. Modeling LtR Rankers

We now introduce a cost-based categorization of LtR algorithms, corresponding to the following four broad families: *linear combinations*, *artificial neural networks*, *forests of regression trees*, and *forests of oblivious trees*. For example, SVM-RANK falls in the *linear combination* family as the computational cost of a SVM-RANK ranker is that of computing a scalar product between two vectors: the set of feature values used to represent a query-document pair, and the set of weights representing the learned model. On the other hand, since $\lambda$-MART belongs to the *forest of regression trees* family, the cost of a $\lambda$-MART ranker is proportional to the number of trees and corresponding levels traversed.

We present a general description of each family of LtR algorithms, and, for each family, we analyze in detail some of the most relevant members. We then derive a cost model based on the tuning parameters characterizing each family, and the actual computation time measured experimentally with our framework. Table 1 summarizes, for each LtR family and algorithms, the tuning parameters exploited. In particular, whereas all the parameters impacts on the quality of learned models, the table also identifies which of these

[5]`http://research.microsoft.com/en-us/projects/mslr/`
[6]`http://webscope.sandbox.yahoo.com/catalog.php?datatype=c`

Table 1: Roles of the training parameters used to tune each LtR algorithm.

| Algorithm | Parameters | Role | Impact on model complexity and cost |
|---|---|---|---|
| | | **Linear Combinations** | |
| CA | – | – | – |
| Ridge | $\theta_\alpha$ | *Shrinking parameter* for reducing overfitting | **No** |
| SVM-rank | $\theta_c$ | *Soft margin* parameter, for weighing the regularization term | **No** |
| | | **Artificial Neural Networks** | |
| ListNet | $\theta_s$ | *Learning rate* | **No** |
| | $\theta_h$ | Number of *hidden nodes* | **Yes**: each hidden node applies the activation function to a weighted sum of the input features. |
| | $\theta_e$ | Number of *learning iterations* | **No** |
| | | **Forests of Regression Trees** | |
| RankBoost | $\theta_t$ | Number of one-level decision trees (*decision stumps*) | **Yes**: linear cost in the number of trees. |
| RF | $\theta_t$ | Number of *trees* | **Yes**: as for the previous $\theta_t$. |
| | $\theta_l$ | Number of *tree leaves* | **Yes**: it impacts on the number of internal nodes, and thus on the number of per-tree tests. |
| GBRT | $\theta_t$ | Number of *trees* | **Yes**,: as for the previous $\theta_t$. |
| | $\theta_l$ | Number of *tree leaves* | **Yes**: as for the previous $\theta_l$. |
| | $\theta_s$ | *Learning rate* | **No** |
| $i$GBRT | $\theta_t, \theta_l, \theta_s$ | The same as GBRT | Analogously to GBRT. |
| $\lambda$-MART | $\theta_t, \theta_l, \theta_s$ | The same as GBRT | Analogously to GBRT. |
| | | **Forests of Oblivious Trees** | |
| $\Omega_\lambda$-MART | $\theta_t, \theta_l, \theta_s$ | The same as GBRT | Analogously to GBRT. Specifically, since branching nodes at the same level of each (balanced) tree perform the same test, the number of per-tree tests is exactly $\log(\theta_l)$. |

Table 2: Best quality results of each LtR algorithm.

| Algorithm | $\theta$ | Dataset | Cost ($\mu$s.) | NDCG@10 |
|---|---|---|---|---|
| **Linear Combinations** | | | | |
| CA | – | MSN | 0.15 | 0.4102 |
| | – | Y!S1[*] | 0.67 | 0.7049 |
| | – | Y!S2 | 0.67 | 0.7208 |
| RIDGE ($\theta_c$) | (0.9) | MSN | 0.15 | 0.3677 |
| | (0.8) | Y!S1 | 0.67 | 0.7294 |
| | (1.0) | Y!S2 | 0.67 | 0.7397 |
| SVM-RANK ($\theta_c$) | (50) | MSN | 0.15 | 0.4012 |
| | (200) | Y!S1 | 0.67 | 0.7238 |
| | (10) | Y!S2 | 0.67 | 0.7306 |
| **Artificial Neural Networks** | | | | |
| LISTNET ($\theta_l, \theta_h, \theta_e$) | $(0.05, 50, 75)$ | MSN | 7.10 | 0.4381 |
| | $(0.005, 50, 75)$ | Y!S1 | 32.27 | 0.7477 |
| | $(0.01, 10, 50)$ | Y!S2 | 6.76 | 0.7340 |
| **Forests of Regression Trees** | | | | |
| RANKBOOST ($\theta_t$) | (500) | MSN | 1.60 | 0.3435 |
| | (500) | Y!S1[*] | 2.11 | 0.7146 |
| | (500) | Y!S2 | 2.27 | 0.7316 |
| RF ($\theta_t, \theta_l$) | $(100, 100)$ | MSN | 4.15 | 0.4163 |
| | $(100, 100)$ | Y!S1 | 4.95 | 0.7296 |
| | $(700, 100)$ | Y!S2 | 26.26 | 0.7548 |
| GBRT ($\theta_t, \theta_l, \theta_s$) | $(1500, 50, 0.05)$ | MSN | 24.07 | 0.4602 |
| | $(1000, 50, 0.05)$ | Y!S1 | 17.00 | **0.7555** |
| | $(100, 50, 0.05)$ | Y!S2 | 5.40 | **0.7620** |
| $i$GBRT ($\theta_t, \theta_l, \theta_s$) | $(1500, 8, 0.1)$ | MSN | 6.07 | 0.4405 |
| | $(1500, 32, 0.1)$ | Y!S1 | 21.84 | 0.7452 |
| | $(800, 32, 0.1)$ | Y!S2 | 14.12 | 0.7525 |
| $\lambda$-MART ($\theta_t, \theta_l, \theta_s$) | $(1500, 50, 0.05)$ | MSN | 13.63 | 0.4618 |
| | $(1000, 50, 0.05)$ | Y!S1 | 15.94 | 0.7529 |
| | $(800, 25, 0.05)$ | Y!S2 | 10.01 | 0.7531 |
| **Forests of Oblivious Trees** | | | | |
| $\Omega_\lambda$-MART ($\theta_t, \theta_l, \theta_s$) | $(1500, 32, 0.1)$ | MSN | 8.66 | **0.4644** |
| | $(1500, 32, 0.1)$ | Y!S1 | 8.55 | 0.7467 |
| | $(1500, 64, 0.1)$ | Y!S2 | 8.21 | 0.7504 |

[*]Due to size of the training set, we trained these models using a 10% random sample of the original dataset.

parameters also affects the scoring performance. With the exception of the parameters reported in the table, we used the default parameters of the different tools without any further modification.

In the following, according to the notation introduced in Sec. 3, we refer to a top ranker with $R_\theta$, the latter denoting a ranker that depends on the specific values of the parameters used to train it. Moreover, $C(R_\theta)$ denotes the cost of the top ranker $R_\theta$, i.e., the average time in $\mu$s required to score a candidate document represented by a given set of features.

## *4.1. Linear Combinations*

This class includes the LtR algorithms producing a ranking model based on a linear combination of features.

Coordinate Ascent (CA) is an (sub)optimization technique for unconstrained optimization. Given a multivariate function to optimize, it performs a univariate optimization on each parameter, keeping fixed all the other parameters. This technique can be applied to learn a linear combination of the features of query-document pairs [26]. The derived linear feature-based model (sub)optimizes directly an effectiveness function on a training set, such as MAP or NDCG. The cost of the learned model is not explicitely tunable at learning time. Some experimental tuning, such as shuffling and random restart, is required during the learning in order to avoid being trapped in local optima. In our tests we perform 5 restarts from random initial weights with 25 iterations to perform line search optimization for every parameter and restart. In all tests, the tolerance was set to 0.001 with no parameters regularization. We evaluated every model obtained due to restarting, and we kept the one having highest effectiveness. The cost of the model is then just a single scalar product between the features vector and the learned weights vector. In this work, we employ the Coordinate Ascent implementation provided by RankLib [27] to learn the models. Results are reported in Table 2. The quality achieved by the CA rankers is not high. In terms of NDCG@10 it achieves 0.4102 on the MSN dataset. The same behaviour is confirmed on the Y!S1 and Y!S2 datasets where CA achieves 0.7049 and 0.7208, respectively. As shown in the following sections, more complex models can bring significant improvement. On the other hand, being CA rankers very simple, they are very fast in scoring candidate documents. Their average per-document scoring cost is one order of magnitude lower than the one shown by the fastest algorithms belonging to the other families of rankers considered in this work.

Ridge Regression (Ridge), also known as Tikhonov regularization [28], solves a regression model where the loss function is defined as the linear least squares function plus a regularization term. The regularization term imposes a penalty on the size of coefficients of the ordinary least squares. As a consequence, Ridge regression regularized a penalized residual sum of squares,

$$min_w ||Xw - y||_2^2 + \theta_c ||w||_2^2$$

The regularization term of the loss function allows RIDGE to be more robust. Specifically, $\theta_c \geq 0$ is the shrinking parameter: the larger the value of $\theta_c$, the greater the regularization term thus allowing the coefficients to become more robust to collinearity. In this paper, we trained RIDGE models by using the `scikit-learn`[7] Python library. We trained several models by varying the value of $\theta_c \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. Note that the variation of $\theta_c$ does not impact the cost of applying the learned ranker.

The best results are reported in Table 2. The RIDGE regressor achieves the best performance among linear models on both Y!S1 and Y!S2 datasets, i.e., the dataset with the largest number of features. Here, RIDGE achieves its best results of NDCG@10 of 0.7294 and 0.7397 by using $\theta_c = 0.8$ and $\theta_c = 1.0$ for Y!S1 and Y!S2 datasets, respectively. This is not true for the MSN dataset, where RIDGE is the worst performing method among linear models. Here, the regressor achieves 0.3677 in terms of NDCG@10 by using $\theta_c = 0.9$.

SVM-RANK [25] is a learning to rank algorithm that adopts a Support Vector Machine (SVM) approach for learning top rankers. The method aims at learning a retrieval function maximizing the expected Kendall's $\tau$. SVM-RANK directly addresses the maximization problem by taking an regularized risk minimization approach. It means that, given a training sample, the learner will select a ranking function $f$ from a family of ranking functions $F$ that minimizes a linear combination of the empirical error over the training pairs and the regularization term, weighted by the *soft margin* parameter $\theta_c$. In this paper, we learn the SVM-RANK models by means of an efficient implementation for training linear SVMs in linear time [29] described by Joachims in [30]. We learn several linear models by varying the value of $\theta_c$. In particular, in our test we tested the values of $\theta_c \in \{0.1, 0.5, 1, 5, 20, 50, 100, 200, 500\}$. Note that the variation of the *soft margin* parameter does not impact on the cost of applying the learned ranker.

Table 2 reports the results corresponding to the largest quality achieved. As expected, the *soft margin* parameter plays an important role in learning an effective model. SVM-RANK shows a better performance in terms of NDCG@10 corresponding to different values of $\theta_c$ in the three datasets. On the MSN dataset, the method achieves 0.4012 in terms of NDCG@10 with $\theta_c = 50$. Here, CA outperforms SVM-RANK of about 2%. The same is not true for Y!S1 and Y!S2 where SVM-RANK achieves 0.7238 and 0.7306 respectively. Here, it shows good performance as it outperforms CA of about 2% on Y!S1 and 1% on Y!S2. As for CA, SVM-RANK (employing a linear kernel) is very simple, thus very fast in scoring candidate documents. Results in terms of average per-document scoring cost are similar to the ones of CA.

[7] `http://scikit-learn.org/stable/modules/linear_model`

**Cost Model**. All rankers based on a linear combination of feature values are extremely simple and fast. Their efficiency does not depend on any learning algorithm parameter, and the scalar product of two vectors is highly optimized on any modern CPU architecture. The per document cost of the ranker depends only on the number $\phi_R$ of features present in the input data, with no additional tuning parameters in the training phase. In order to estimate the scoring cost of linear combination rankers as a funtion of $\phi_R$, we run a linear regression on our training data, i.e., all the training experiments across the three datasets, with both CA and SVM-RANK on varying its parameter $\theta_c$, for a total of 30 scoring time measurements. The following estimate was obtained:

$$C(R_\theta) \ = \ 0.92{\cdot}10^{-3} \cdot \phi_R \ + \ 26.6{\cdot}10^{-3}$$

with coefficient of determination $R^2 = 0.99$.

## *4.2. Artificial Neural Networks*

This class includes all the LtR algorithms producing a ranking model based on artificial neural networks having an input for each one of the features.

ListNet. Artificial Neural Networks were exploited by RankNet [31], one of the first LtR algorithms. The new cost function exploited by RankNet depends on the number of document pairs being correctly ordered. Unlike IR quality measures, this cost function is continuous and differentiable. Unfortunately, a scoring function that minimizes the number of mis-ordered pairs does not necessary optimises IR quality measures. For this reason, the RankNet-based rankers do not obtain a high effectiveness. To overcome this limitation, ListNet [32] addresses a different optimisation problem. A results permutation probability distribution is devised on the basis of the learned score and the relevance labels. By doing so, it is possible to estimate the error via the Kullback—Leibler divergence between the two. Then, such error is minimised with an artificial neural network. Note that ListNet does not optimise directly an IR measure. Unlike RankNet, it optimises the ordering of the full result list according to its own defined cost function. ListNet is thus a list-wise method.

The implementation[8] used in this paper uses a neural network with one hidden layer. We extended the library with some functionalities allowing to save a learned model in XML format as discussed in Section 3.2. We tested several parameter combinations $\theta = (\theta_s, \theta_h, \theta_e)$, which include the learning rate $\theta_s \in \{0.005, 0.01, 0.05, 0.1\}$, the number of hidden nodes $\theta_h \in \{10, 25, 50, 100\}$, and the number of learning iterations $\theta_e \in \{25, 50, \ldots, 200\}$.

The best results are reported in Table 2, while the results on varying $\theta$ are shown in Figure 2. Unexpectedly, the best results are achieved with a small number of hidden nodes. Note that only the number

[8]`http://www.dmi.usherb.ca/~larocheh/mlpython/`

of hidden nodes impacts on the cost of the learned ranker, while the other two parameters only impact on the quality. The four groups of points in Figure 2 with the same cost correspond to the four hidden nodes settings. The resulting ranker is more time consuming than previous solutions. We observe that ListNet provides better performance than linear models with the only exception of the Ridge algorithm on the Y!S2 dataset. The improved performance is due to the capability of neural networks to model non-linear functions and also to the list-wise cost function being optimised. On the other hand, ListNet is at least one order of magnitude more expensive than previous methods based on linear models.

**Cost Model**. The cost of a ranker based on an artificial neural network depends on the number of input nodes and hidden nodes. The number of input nodes equals to the number of features in the data, while the number of hidden nodes is a learning parameter. The per-document scoring cost in $\mu$s of a neural network can thus be easily estimated as follows:

$$C(R_\theta) \ = \ 11{\cdot}10^{-3} \cdot \theta_h \ + \ 0.9{\cdot}10^{-3} \cdot \theta_w \ + \ 0.1$$

where $\theta_h$ is the number of hidden nodes and $\theta_w$ is the total number of weights. The coefficients of the linear combination were found through linear regression on the experimental data of ListNet on all the three dataset used on varying the training parameters, for a total of 384 scoring time measurements. Note that the number of input nodes is not considered as it is implicitly encompassed by $\theta_w$. This model is very accurate as the coefficient of determination of the regression is $R^2 = 0.99$.

### *4.3. Forests of Regression Trees*

This class includes all the LtR algorithms producing a ranking model based on an additive ensemble of regression trees.

RankBoost [33] is an algorithm for combining preferences based on the boosting approach. Boosting is a method of producing highly accurate prediction rules by combining many "weak" rules which may be only moderately accurate. RankBoost, like all boosting algorithms, operates in rounds. At each round, a weak learner is trained and it is combined with already trained weak rankers through a weight, with the idea that each weak learner is able to discriminate effectively on a subset of the input sample space, while their weighted combination is able to discriminate effectively on the whole input sample space. The weak rankers typically used are one-level decision trees (decision stumps). Specifically, the root of each tree encompasses a Boolean comparison between a feature $f_i$ of a given instance $x$ and a learned threshold $\tau$. RankBoost is a pair-wise algorithm.

The quality and cost of the learned model are tunable at learning time by selecting the number $\theta_t$ of weak learners to train. We selected $\theta_t = \{100, 200, 300, 400, 500\}$.

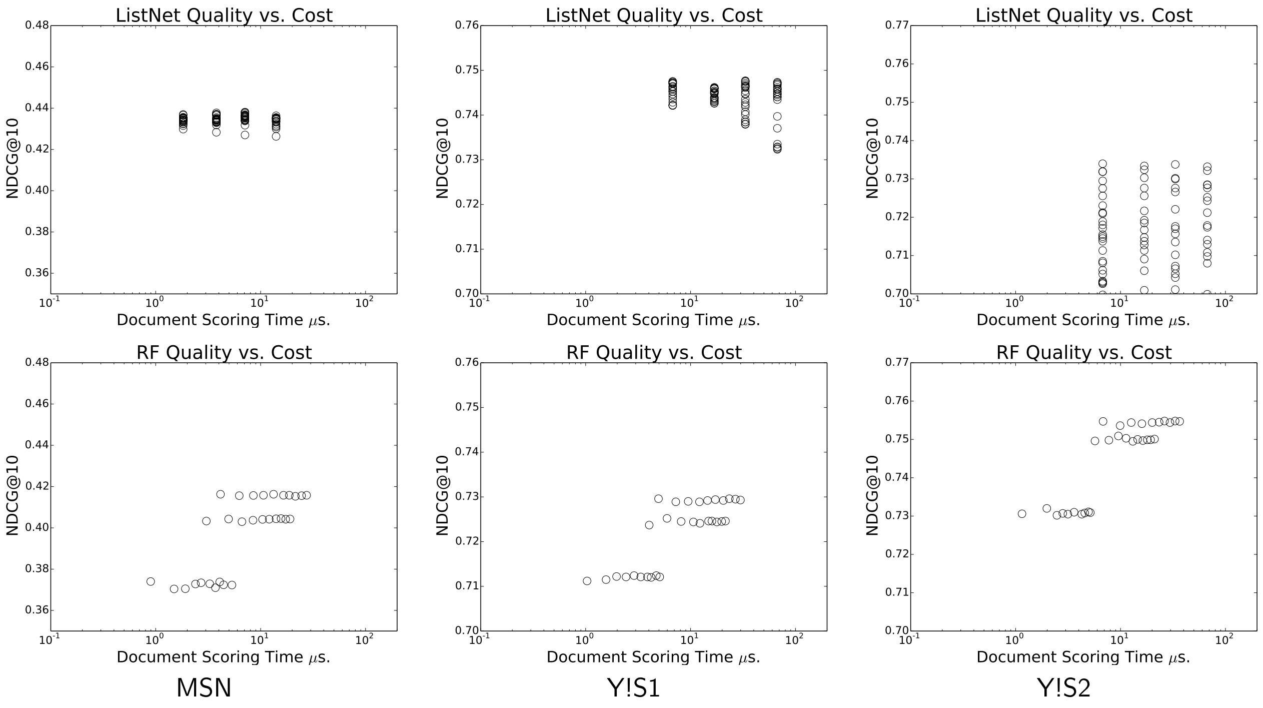

Figure 2: Effectiveness in terms of NDCG@10 of ListNet, and RF top rankers as a function of average per-document scoring cost.

The results reported in Table 2 show that RankBoost is never a good choice compared with linear algorithms such as Ridge: it is both more expensive and less accurate on each dataset considered.

Random Forests (RF) [34] are defined as a collection of tree-based regressors where each of them casts a prediction for a given input instance, and the final prediction is the arithmetic mean of the scores produced by the regressors in the forest. Each regression tree is trained on a random subset of the feature set so as to reduce overfitting. Random forests can optimise several information retrieval measures, depending on the tree regressor wrapped. In our experiments we used a $\lambda$-MART tree, which is described below.

The cost and quality of the learned model depends on a few parameters $\theta = (\theta_t, \theta_l)$. The number of trees $\theta_t$ and the number of leaves $\theta_l$ per tree determine the number of steps needed to traverse the forest at scoring time. Larger and more complex trees produce better results, as long as overfitting is avoided, e.g., by using a validation set. For evaluation purposes, $\theta_t = \{100, 200, \ldots, 1000\}$ and $\theta_l = \{10, 50, 100\}$.

In this work, we employ the implementation provided by RankLib [27] to learn the RF models. We extend the library with some functionalities allowing to save a learned model in XML format as discussed in Section 3.2. Points in Figure 2 on the same horizontal line correspond to random forests with varying number of trees, while the different number of leaves impacts on the quality of the ranker. As reported in Table 2, RF can achieve good quality but with a non trivial cost. On both the MSN and Y!S1 datasets, it is between 7 and 27 times less efficient than linear models, but the achieved quality is only marginally larger. On the same datasets, ListNet provides larger quality figures at a comparable cost. On the Y!S2 dataset, RF exhibits the best quality observed so far, still being 4 times slower than ListNet.

Gradient-Boosted Regression Trees (GBRT) [6] is a general function approximation technique aiming at finding the best function $f$ minimising a given loss function $L(f)$. $f$ is defined as a weighted sum of weak-learners functions, i.e., $f = \sum_i w_i f_i$. The basic assumption is that if we can compute the gradient $\partial L(f)/\partial f$, then we can solve the minimisation problem of finding the best $f_i$ via gradient descent. In fact, if the gradient is computed for a set of data points $\mathbf{x}$, it is enough to find a function $f_i$ able to approximate the gradient value at the given $\mathbf{x}$. This is a regression problem which is solved with a regression tree. Therefore, the ranking function produced by GBRT is indeed a forest of (weighted) trees. Usually, and in this work, the loss function adopted is the root mean squared error (RMSE), meaning that GBRT tries to predict the relevance labels of the document in the training set. This loss function makes GBRT a point-wise algorithm. We learn the GBRT models assessed in this paper by exploiting our QuickRank framework.

In terms of cost at scoring time, GBRT is equivalent to RF since they are both forests of trees of tunable size. For GBRT we evaluated different parameter sets $\theta = (\theta_t, \theta_l, \theta_s)$ by varying the number of trees $\theta_t = \{100, 200, \ldots, 1500\}$, the number of leaves per tree $\theta_l = \{5, 10, 25, 50\}$ and the shrinkage (or learning rate) $\theta_s = \{0.05, 0.1, 0.5, 1.0\}$.

The rankers generated by GBRT largely outperform all of the previous on every dataset tested. This quality comes at a cost. On the MSN dataset, GBRT generates the third most expensive model observed in the full set of experiments. This is because the best rankers employ more than 1,000 trees with 50 leaves each on both MSN and Y!S1. Finally, note that the smallest learning rate and the largest number of leaves were the optimal setting on all datasets. Nevertheless, as shown in Figure 3, by varying the parameter set a large number of models can be produced, and many of them have a smaller cost and comparable quality. One exception is Y!S2, where the best model encompasses 100 trees only. In this case, a larger number of trees leads to overfitting, thus producing expensive models with poor performance. This overfitting behaviour is visible in the GBRT plots of Figure 3 (top row) where the most costly rankers decrease their effectiveness. As usual, the use of a validation set is mandatory to avoid such situations.

Initialized Gradient-Boosted Regression Trees.
The GBRT algorithm is an iterative process aiming at finding a weighted sum of weak learners functions miniming a given loss function, e.g., RMSE. GBRT is traditionally initialized with the all-zero function, meaning that the initial prediction of each document is zero. Mohan et al. [35] propose to initialize GBRT with the predictions produced by a previously learned RF ranker. They call this approach Initialized Gradient-Boosted Regression Trees ($i$GBRT). The proposal constitutes an enhancement of the GBRT

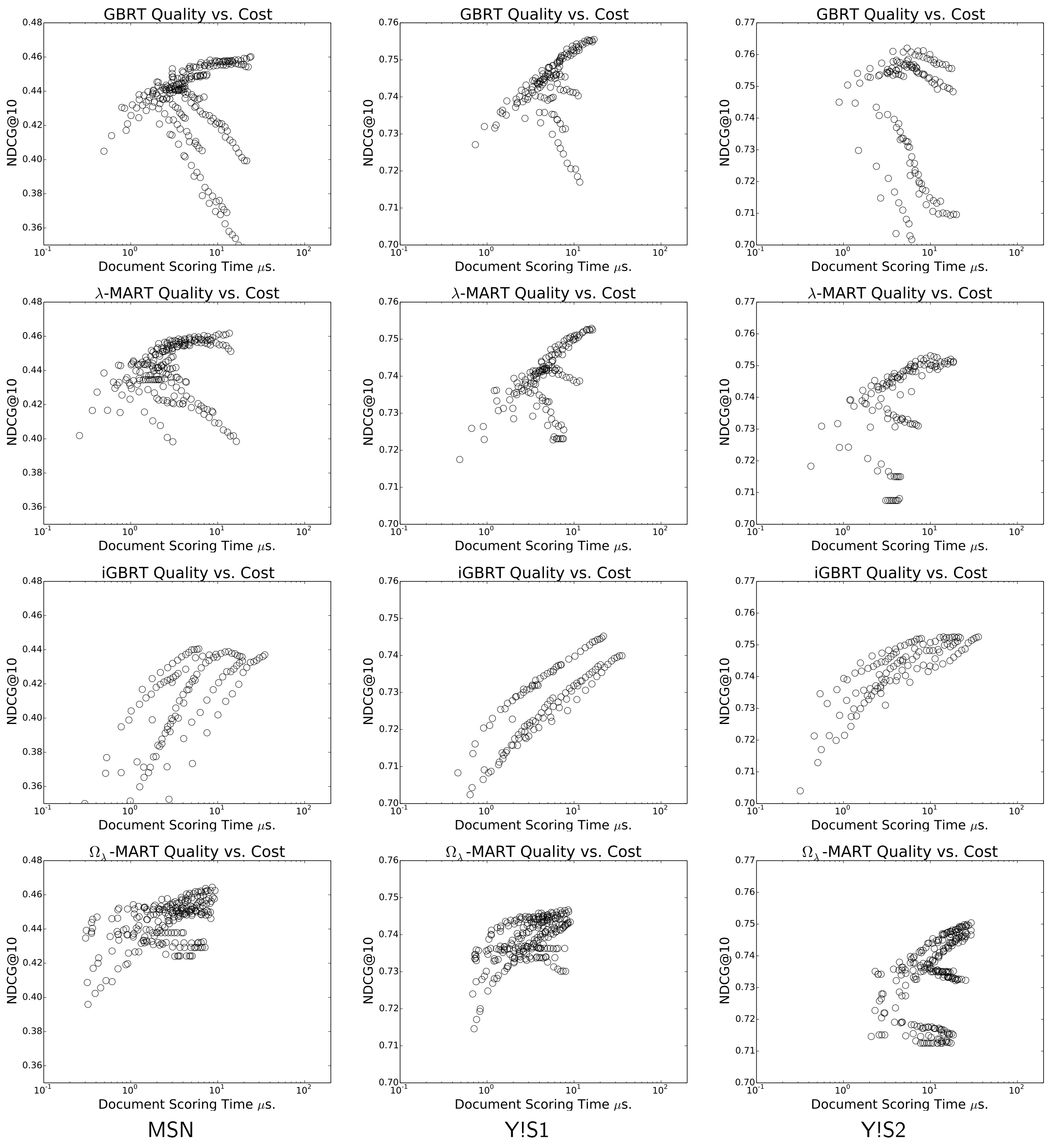

Figure 3: Effectiveness in terms of NDCG@10 of GBRT, $\lambda$-MART, $i$GBRT, and $\Omega_\lambda$-MART top rankers as a function of average per-document scoring cost.

algorithm as RF is a good initialization point. The reasons of that are: i) RF is known to be robust to overfitting as it is the result of a bagging process, ii) it is insensitive to parameter settings. Moreover, the training overhead coming from determining the initialization scores is not big as the training step of RF is embarrassingly parallel.

In terms of cost at scoring time, $i$GBRT is equivalent to other algorithms belonging to the *forests of regression trees* family. We evaluate different parameter sets $\theta = (\theta_t, \theta_l, \theta_s)$ by varying the number of trees $\theta_t = \{100, 200, \ldots, 1500\}$, the number of leaves per tree $\theta_l = \{4, 8, 16, 32, 64\}$ and the shrinkage (or learning rate) $\theta_s = \{0.05, 0.1\}$. In addition, we perform the initialization step by learning on the training queries of each dataset a RF model consisting of 1000 bags and by following the common rule of thumb of setting to 10% the number of features to sample in each bag [35].

We learn the $i$GBRT models by exploiting the RT-rank library (v1.5-alpha2) [36]. We extend the library with some functionalities allowing to save a learned model in XML format as discussed in Section 3.2.

Note that this implementation generates balanced trees only. This is why we experimented a number of leaves different from other algorithms.

Results in Table 2 show that $i$GBRT does not improve over GBRT on any dataset. The maximum difference in terms of NDCG@10 is observed on the MSN dataset where GBRT outperforms $i$GBRT by 4%. However, $i$GBRT is able to produce fast rankers with sufficient quality. We believe that this behaviour is due to the RF-based initialization, which allows the algorithm to start from an initial solution of good quality. This strategy avoids overfitting: Figure 3 shows that quality keeps increasing with the complexity of the model. But, we must conclude that $i$GBRT is not able to reach the most promising regions of the search space.

The LambdaMART ($\lambda$-MART) algorithm [7] is an improvement over GBRT. The main issues in machine-learned document scoring functions that optimise information retrieval measures is that such measures involve a sorting of documents, and sorting is not a differentiable function. The $\lambda$-MART approach exploits the fact that GBRT only requires the gradient to be computed at the given set of data points $\mathbf{x}$. The gradient at a given data point describes how much the score of a document should be increased/decreased to improve the loss function. Given two documents, this quantity is estimated by computing the loss function variation when swapping their current score. Every document is compared with any other document, and the loss function variation is accumulated. The resulting value is named $\lambda$, and it can be considered as the gradient of the loss function computed at the given document. Indeed, the $\lambda$ values are slightly more complex, since they include a factor related to the RankNet [31] cost. This gradient estimation is plugged into a GBRT algorithm, thus obtaining $\lambda$-MART. $\lambda$-MART can optimise several information retrieval measures, e.g., NDCG, and for this reason it can be considered a list-wise algorithm. Note that in optimising the cost function, the score produced by $\lambda$-MART can be distant from the training relevance labels, as the algorithms aims at finding whatever score generates a good ordering of documents. We trained the $\lambda$-MART models assessed in this paper by using the implementation included in our QuickRank framework.

In terms of cost at scoring time, there is no difference among RF, $\lambda$-MART and GBRT, since they are all forests of trees of tunable size. For $\lambda$-MART we evaluated the same parameter sets as for GBRT. The rankers generated by $\lambda$-MART are comparable to that of GBRT, with $\lambda$-MART performing better on MSN and worse on Y!S1 and Y!S2. Similarly to GBRT and as shown in Figure 3, several rankers can be learned with smaller cost and high quality. Also in $\lambda$-MART, a large number of leaves and trees coupled with a small learning rate provide the best results.

The $\Omega_\lambda$-MART [2] algorithm can be seen as a variation of $\lambda$-MART, where oblivious regression trees [37] are used instead of standard regression trees. In oblivious regression trees, the same splitting criterion is used across an entire level of a tree. As a consequence, the resulting trees are balanced. The goal of reducing the degrees of freedom of the algorithm at training time is to minimize the risk of overfitting. Indeed, $\lambda$-MART and $\Omega_\lambda$-MART resulted to be the most effective LtR algorithms among the ones participating in the Yahoo! Learning to Rank Challenge [38].

We trained $\Omega_\lambda$-MART models by exploiting our QuickRank framework that includes the first public-domain implementation of this interesting LtR algorithm. Our implmentation acccepts input dataset in the SVM format and store the learned model in XML format as discussed in Section 3.2. Specifically, we experimented with models obtained with different parameter sets $\theta = (\theta_t, \theta_l, \theta_s)$ by varying the number of trees $\theta_t = \{100, 200, \ldots, 1500\}$, the number of leaves per tree $\theta_l = \{8, 16, 32, 64\}$ and the shrinkage $\theta_s = \{0.05, 0.1, 0.5, 1.0\}$.

According to the results reported in Table 2, $\Omega_\lambda$-MART performs similarly to the other algorithms in the same family. Note that it is the best performing algorithm on the MSN dataset. As shown in Figure 3, $\Omega_\lambda$-MART is able to generate rankers of good quality and small cost. This is due to the balanced trees used. Recall that the cost of scoring a tree is proportional to its depth $\theta_d$. Even if also $i$GBRT generates balanced trees, $\Omega_\lambda$-MART produces better models, thanks to the optimization of the list-wise cost function employed by $\lambda$-MART.

**Cost Model**. All the rankers in this class are based on forests of regression trees, even if we have simple decision stumps for RankBoost. The cost of such rankers is mainly affected by the number of trees $\theta_t$ and the number of leaves $\theta_l$. In addition, it results to be useful to consider also the maximum depth of each tree in the forest, averaged across all the trees, denoted with $\theta_d$. This is because trees are not balanced, and even if parameter $\theta_l$ limits the number of nodes, the length of the longest path varies from tree to tree.

We estimated the per-document scoring cost of a regression tree model by fitting a linear regression of all the scoring time measurements taken across the three datasets for each of the aforementioned methods by varying their training parameters (for a total of 2715 samples), with the following result:

$$C(R_\theta) \ = \ 0.341 \cdot \theta_t \ + \ 0.005 \cdot \theta_l \ + \ 0.22 \cdot \theta_d \ - \ 1.8$$

This cost model is sufficiently accurate with a coefficient of determination of the regression $R^2 = 0.90$. We observe that $\theta_t$ has a significant impact as trees are increased by hundreds. Also the average maximum depth $\theta_d$ as a strong impact. Note that $\theta_d$ is expected to increase logarithmically with the number of leaves (i.e., nodes) in the tree, but this may not hold for unbalanced trees. By restricting to the MSN dataset and fixing $\theta_t = 1000$, we observed that the scoring time varies between 3.0$\mu$s ($\theta_d = 4.9$, $\theta_l = 5$) and 17.2$\mu$s ($\theta_d = 39$, $\theta_l = 50$) for GBRT, while the variance is much smaller, between 2.7$\mu$s ($\theta_d = 3.4$, $\theta_l = 8$) and 6.3$\mu$s ($\theta_d = 5$, $\theta_l = 64$) for $\Omega_\lambda$-MART, as also shown in Figure 3 (recall that $\Omega_\lambda$-MART may create less than $2^{\theta_d - 1}$ leaves if a good splitting criterion is not found). We conclude that GBRT grows quite unbalanced trees having average depth close to the number of leaves, while $\Omega_\lambda$-MART grows balanced and thus less expensive trees.

## 5. Quality vs. Cost Tradoffs

After reviewing the results of each algorithm independently, we now compare different methods on the basis of their *QC-curve*. We then present a budget-based analysis where we discuss, for a set of time budgets, which are the best rankers to be adopted. Finally, we evaluate the capability of the rankers to provide high quality results for any given budget threshold on the basis of *AuQC*, i.e., their area under the *QC-curve*.

### *5.1. Comparison through Dominant Curve*

Figure 4 compares the different LtR approaches on the basis of the *QC*-curve derived from their respective rankers. Since each curve includes the dominant points in terms of ranking effectiveness of each learned model, it is actually used to answer our research question **Q1**, i.e., how the effectiveness of the top ranker trained with a given LtR algorithm varies with its computation cost at testing time.

The first observation is that the LtR approaches do not show the same behavior on all datasets. For instance, *i*GBRT provides interesting results on Y!S2 dataset, comparable to that of $\lambda$-MART, but its performance is unsatisfactory on the other two datasets. Another interesting observation, is that tree-based models can be less expensive than linear models when the dataset at hand has a large number of features, as with Y!S1 and Y!S2. In these two cases, it is possible to build cheap regression forests with almost the same performance. This is because 700 multiply-add operations required by linear models may have a cost larger than that of a simple forest with 100 trees having 4 leaves each, for a total of 400 nodes.

As highlighted before, $\lambda$-MART is often outperformed by GBRT. Even if $\lambda$-MART is a list-wise approach, it may not improve over the point-wise algorithm GBRT. This might be due to the fact that

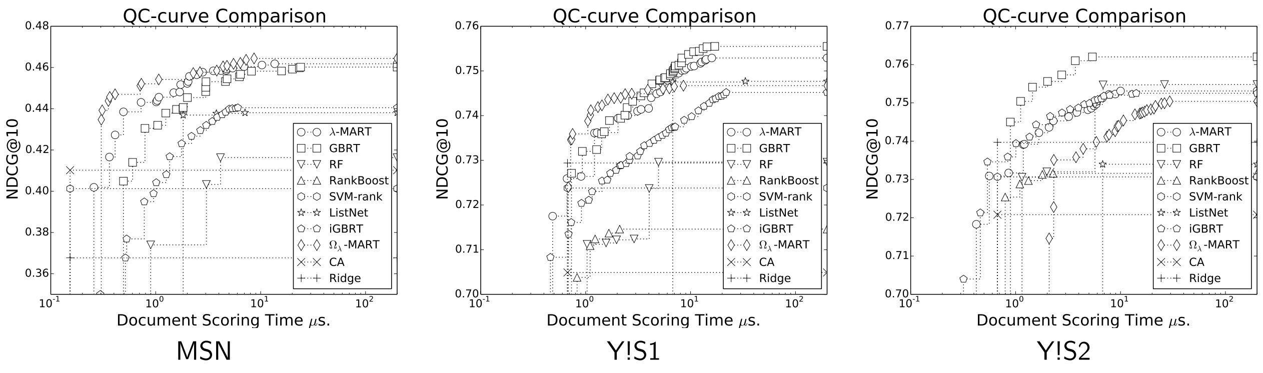

Figure 4: *QC*-curves of the various LtR algorithms on the MSN, Y!S1, Y!S2 datasets.

λ-MART encompasses the cost function of RankNet, which is known to perform poorly. The $\Omega_\lambda$-MART algorithm shows good performance especially on MSN, where it provides the best quality for any scoring time larger than 0.25 $\mu$s.

Finally, on each dataset, the choice of the best ranker depends on the desired cost. For instance, on the MSN dataset, CA is the best algorithm for scoring times smaller that 0.2 $\mu$s. On the Y!S1 dataset, $\Omega_\lambda$-MART is the best performing for budgets up to 4 $\mu$s, while GBRT provides best quality for large time budgets. Finally, GBRT is the best algorithm on Y!S2 for scoring times larger than 1 $\mu$s, while Ridge and *i*GBRT are the best choices for smaller time budgets.

This supports the claim of this work: quality and cost should be analysed all together in order to draw significant conclusions.

We highlight that the proposed evaluation methodology can be applied to any quality measure of interest. In this work, we limit our analysis to NDCG@10, which is one of the most widely adopted. We also conducted experiments with the ERR@10 measure and achieved similar results. It is beyond the scope of this work to understand which algorithm is best suited to optimize a given quality measure. Nevertheless, we found that the parameter settings that best optimized NDCG, were also the best in optimizing ERR, i.e., large number of leaves and small learning rate. This might suggest a rule of thumb for parameter tuning when a large number of trees is an acceptable solution.

## 5.2. Comparison under Budget Constraints

In the following, we answer the research question **Q2**, i.e., given a time budget, what are the LtR algorithm and the associated tuning parameters providing the top ranker with the best quality at a cost smaller than the budget, by analysing the performance of the LtR algorithms on varying the given cost budget $B$. We perform the analysis by assuming different values of the average per-document cost. We test the following values (in $\mu$s) of $B$: $\{0.5, 0.75, 1, 2, 4, 8, 16, 32, 64\}$. Table 3 confirms that there is not a best (LtR) algorithm for any given time budget.

Results highlight that GBRT and $\Omega_\lambda$-MART are the best performing algorithms in most experiments, with the former being superior on the Y!S2 dataset and the latter on MSN. The MSN dataset has a smaller number of features (136 vs. 700), suggesting the $\Omega_\lambda$-MART strategy of exploiting oblivious tree may prevent the algorithm to explore complex features interaction in Y!S1 and Y!S2.

Dealing with small time budgets is more challenging, and in this case different algorithms may provide the desired quality. With the smallest time budget $B = 0.5\mu$s, each dataset suggests a different best performing algorithm. When $B$ increases to $B = 0.75\mu$s, also Ridge is able to provide a good model on the Y!S2 dataset. Finally, on the Y!S1 and Y!S2 datasets, three different algorithms are shown to be the best performing depending on the time budget.

The experiments conducted show that the choice of the best algorithm across different time budgets is not unique, and this opens to novel load-sensitive strategies for processing queries. We highlight that different queries may take advantage of a different amount of time to be spent for the second ranking stage. This is the case for queries with a different cost during the first stage [39]. For instance, in both Y!S1 and Y!S2, our analysis proposed three different models depending on the given time budget, which can be even decided dynamically at run time for every single query. Therefore, the proposed analysis may lead to the choice of multiple rankers corresponding to multiple time budgets.

The global behavior of any given algorithm on varying the time budget $B$ is captured by the *AuQC* measure, whose results are discussed below.

Table 3: Best quality results by varying the time budget $B$.

| $B$ | MSN | | Y!S1 | | Y!S2 | |
|---|---|---|---|---|---|---|
| $\mu$s | Algorithm | NDCG@10 | Algorithm | NDCG@10 | Algorithm | NDCG@10 |
| 0.5 | $\Omega_\lambda$-MART | 0.4470 | $\lambda$-MART | 0.7175 | $i$GBRT | 0.7213 |
| 0.75 | $\Omega_\lambda$-MART | 0.4521 | $\Omega_\lambda$-MART | 0.7347 | Ridge | 0.7397 |
| 1 | $\Omega_\lambda$-MART | 0.4521 | $\Omega_\lambda$-MART | 0.7359 | GBRT | 0.7450 |
| 2 | $\Omega_\lambda$-MART | 0.4542 | $\Omega_\lambda$-MART | 0.7439 | GBRT | 0.7541 |
| 4 | $\Omega_\lambda$-MART | 0.4585 | $\Omega_\lambda$-MART | 0.7458 | GBRT | 0.7610 |
| 8 | $\Omega_\lambda$-MART | 0.4637 | GBRT | 0.7513 | GBRT | 0.7620 |
| 16 | $\Omega_\lambda$-MART | 0.4644 | GBRT | 0.7554 | GBRT | 0.7620 |
| 32 | $\Omega_\lambda$-MART | 0.4644 | GBRT | 0.7555 | GBRT | 0.7620 |
| 64 | $\Omega_\lambda$-MART | 0.4644 | GBRT | 0.7555 | GBRT | 0.7620 |

### *5.3. Area under QC-curve*

We now address **Q3**, i.e., how can we characterise the quality versus cost tradeoff of a given LtR algorithm, by analyzing the results obtained by the proposed measure *area under QC-curve* (*AuQC*). This measure takes into account the capability of a given algorithm to provide high effectiveness for any given time budget up to a maximum one. Since a *QC*-curve encompasses all the dominant rankers generated by a given LtR algorithm for the various tuning parameters of the algorithm, a very large value of $AuQC$ reveals that the training step of the given LtR algorithm can be tuned in a way to produce a high quality ranker for any given time budget up to a maximum one.

In Table 4, we report the $AuQC$ value for the maximum time budget $B = 100\mu$s for all the LtR algorithms analyzed on the three datasets. $\Omega_\lambda$-MART and GBRT turned out to be globally the best choices, namely $\Omega_\lambda$-MART for the MSN dataset, and GBRT for both Y!S1 and Y!S2. Looking at the $QC$-curves in Figure 4, we can actually observe that $\Omega_\lambda$-MART and GBRT are not only the best when, as expected, a very large number of trees is exploited and thus the time budget is used almost completely to score each document, but also when simpler models with smaller scoring time, down to $1\mu$s, are exploited. We highlight that $\lambda$-MART exhibits quality in between the two algorithms on the three datasets, and it has the second best average performance across the three datasets, showing that it is a robust and high performing approach. Therefore, we can conclude that our $AuQC$ proposed metric is able to capture the global behaviour of a LtR algorithm in the whole quality vs. cost space.

Table 4: $AuQC_R(B)$ with $B = 100\mu$s.

| | MSN | Y!S1 | Y!S2 | *avg.* |
|---|---|---|---|---|
| CA | 0.4096 | 0.7002 | 0.7160 | 0.6086 |
| Ridge | 0.3671 | 0.7245 | 0.7347 | 0.6088 |
| SVM-rank | 0.4006 | 0.7189 | 0.7258 | 0.6151 |
| ListNet | 0.4300 | 0.6971 | 0.6844 | 0.6038 |
| RankBoost | 0.3418 | 0.7086 | 0.7257 | 0.5920 |
| RF | 0.4115 | 0.7215 | 0.7450 | 0.6260 |
| GBRT | 0.4568 | **0.7491** | **0.7550** | **0.6536** |
| $i$GBRT | 0.4382 | 0.7405 | 0.7496 | 0.6428 |
| $\lambda$-MART | 0.4600 | 0.7483 | 0.7493 | 0.6525 |
| $\Omega_\lambda$-MART | **0.4625** | 0.7414 | 0.7333 | 0.6457 |

## 6. Conclusions

In this paper we studied the performance of ten LtR algorithms in the quality vs. cost space. The spectrum of ranking models analyzed ranges from the simplest ones, which at scoring time exploit a simple linear combination of features values associated with each query-document pair, to the most complex ones, based on forests of regression trees.

We present a comprehensive and reproducible analysis done by means of three well-known publicly available datasets for Learning to Rank. Our analysis employs QuickRank, an open-source C++ framework which allowed us to fairly compare the effectiveness and the efficiency of the different LtR algorithms analyzed. Notably, QuickRank includes the optimized, multi-threaded implementations of the most effective tree-based learners: GBRT [6], $\lambda$-MART [7], and the first public-domain implementation of Oblivious Lambda-Mart ($\Omega_\lambda$-MART) [2]. QuickRank also provides a C++ plugin that scores tree-based models by using a state-of-the-art strategy, namely QuickScorer [16]. Experiments were performed by sweeping the training parameters of each LtR algorithm to build instances of ranking models behaving differently in the quality vs. cost space.

We introduced a novel measure, called *area under QC-curve* (*AuQC*), to evaluate which LtR algorithm learns the most effective ranker (measured in terms of NDCG@10) for a given time budget. The analysis conducted by considering *AuQC* shows that, for a maximum time budget of 100 $\mu$s, the LtR algorithms $\Omega_\lambda$-MART and GBRT are globally the most effective. The two algorithms show the best effectiveness when a large number of deep trees are exploited and thus the time budget is used almost completely to score each document. However, these two algorithms turned out to be very effective also when small and not expensive ranking models are used. Therefore, the area under the *QC*-curve is a useful measure to capture the global behaviour of a LtR algorithm in the whole quality vs. cost space.

## References

## References

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

[2] I. Segalovich, Machine learning in search quality at yandex, Invited Talk, SIGIR.

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

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

[5] I. Arapakis, X. Bai, B. B. Cambazoglu, Impact of response latency on user behavior in web search, in: Proc. SIGIR, ACM, 2014, pp. 103–112.

[6] J. H. Friedman, Greedy function approximation: a gradient boosting machine, Annals of Statistics (2001) 1189–1232.

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

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

[9] C. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, New York City, NY, USA, 2008.

[10] N. Tax, S. Bockting, D. Hiemstra, A cross-benchmark comparison of 87 learning to rank methods, Information Processing & Management 51 (6) (2015) 757 – 772.

[11] C. Macdonald, R. L. Santos, I. Ounis, The whens and hows of learning to rank for web search, Information Retrieval 16 (5) (2013) 584–628.

[12] C. Pölitz, R. Schenkel, Learning to rank under tight budget constraints, in: Proc. SIGIR, ACM, 2011, pp. 1173–1174.

[13] N. Asadi, J. Lin, Fast candidate generation for two-phase document ranking: Postings list intersection with bloom filters, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12, ACM, New York, NY, USA, 2012, pp. 2419–2422.

[14] N. Tonellotto, C. Macdonald, I. Ounis, Efficient and effective retrieval using selective pruning, in: Proceedings of WSDM 2013, 2013, pp. 63–72.

[15] G. Ottaviano, R. Venturini, Partitioned elias-fano indexes, in: Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, ACM, New York, NY, USA, 2014, pp. 273–282.

[16] C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, R. Venturini, Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees, in: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '15, ACM, New York, NY, USA, 2015, pp. 73–82.

[17] N. Asadi, J. Lin, Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures, in: Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '13, ACM, New York, NY, USA, 2013, pp. 997–1000.

[18] L. Wang, J. J. Lin, D. Metzler, Learning to efficiently rank, in: Proc. SIGIR, ACM, 2010, pp. 138–145.

[19] L. Wang, D. Metzler, J. J. Lin, Ranking under temporal constraints, in: Proc. CIKM, ACM, 2010, pp. 79–88.

[20] L. Wang, J. J. Lin, D. Metzler, A cascade ranking model for efficient ranked retrieval, in: Proc. SIGIR, ACM, 2011, pp. 105–114.

[21] N. Asadi, J. Lin, Training efficient tree-based models for document ranking, in: Proc. ECIR, Springer, 2013, pp. 146–157.

[22] N. Asadi, J. Lin, A. P. de Vries, Runtime optimizations for prediction with tree-based models, IEEE Transactions on Knowledge and Data Engineering 99 (PrePrints) (2013) 1.

[23] X. Tang, X. Jin, T. Yang, Cache-conscious runtime optimization for ranking ensembles, in: Proc. SIGIR, ACM, 2014, pp. 1123–1126.

[24] D. Broccolo, C. Macdonald, S. Orlando, I. Ounis, R. Perego, F. Silvestri, N. Tonellotto, Load-sensitive selective pruning for distributed search, in: Proceedings of CIKM 2013, 2013, pp. 379–388.

[25] T. Joachims, Optimizing search engines using clickthrough data, in: Proc. SIGKDD, ACM, 2002, pp. 133–142.

[26] D. Metzler, W. B. Croft, Linear feature-based models for information retrieval, Information Retrieval 10 (3) (2007) 257–274.

[27] RankLib, `http://sourceforge.net/p/lemur/wiki/RankLib/`.

[28] A. N. Tikhonov, A. Goncharsky, V. Stepanov, A. G. Yagola, Numerical methods for the solution of ill-posed problems, Vol. 328, Springer Science & Business Media, 2013.

[29] SVM-rank, `http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html`.

[30] T. Joachims, Training linear svms in linear time, in: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2006, pp. 217–226.

[31] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, G. Hullender, Learning to rank using gradient descent, in: Proc. ICML, ACM, 2005, pp. 89–96.

[32] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, H. Li, Learning to rank: from pairwise approach to listwise approach, in: Proc. ICML, ACM, 2007, pp. 129–136.

[33] Y. Freund, R. Iyer, R. E. Schapire, Y. Singer, An efficient boosting algorithm for combining preferences, Journal of Machine Learning Research 4 (2003) 933–969.

[34] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32.

[35] A. Mohan, Z. Chen, K. Q. Weinberger, Web-search ranking with initialized gradient boosted regression trees., in: Yahoo! Learning to Rank Challenge, 2011, pp. 77–89.

[36] RT-rank, `https://sites.google.com/site/rtranking/`.

[37] R. Kohavi, Bottom-up induction of oblivious read-once decision graphs, in: Proc. ECML, Springer, 1994, pp. 154–169.

[38] O. Chapelle, Y. Chang, Yahoo! learning to rank challenge overview., Journal of Machine Learning Research-Proceedings Track 14 (2011) 1–24.

[39] C. Macdonald, N. Tonellotto, I. Ounis, Learning to predict response times for online query scheduling, in: Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '12, ACM,

New York, NY, USA, 2012, pp. 621–630.