QuickRank: a C++ Suite of Learning to Rank Algorithms

Gabriele Capannini[4], Domenico Dato[3], Claudio Lucchese[1]
Monica Mori[3], Franco Maria Nardini[1], Salvatore Orlando[2]
Raffaele Perego[1], and Nicola Tonellotto[1]

[1] ISTI-CNR, Pisa, Italy
[2] University Ca' Foscari of Venice, Italy
[3] Tiscali S.p.A., Cagliari, Italy
[4] IDT, Mälardalens högskola, Västerås, Sweden.

Abstract. Ranking is a central task of many Information Retrieval (IR) problems, particularly challenging in the case of large-scale Web collections where it involves effectiveness requirements and efficiency constraints that are not common to other ranking-based applications. This paper describes QuickRank, a C++ suite of efficient and effective Learning to Rank (LtR) algorithms that allows high-quality ranking functions to be devised from possibly huge training datasets. QuickRank is a project with a double goal: i) answering industrial need of Tiscali S.p.A. for a flexible and scalable LtR solution for learning ranking models from huge training datasets; ii) providing the IR research community with a flexible, extensible and efficient LtR framework to design LtR solutions and fairly compare the performance of different algorithms and ranking models. This paper presents our choices in designing QuickRank and report some preliminary use experiences.

1 Introduction

In the last years 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) [11] framework. It is known that several commercial Web Search Engines (WSEs) exploit LtR solutions using a large number of relevance signals as features of the learned models. The effectiveness of these WSE rankers relies on huge training datasets containing gigabytes of annotated query-document examples and efficient and scalable machine learning solutions [15].

In this paper, we describe QuickRank, a high-performance Learning to Rank (LtR) toolkit that provides C++ multithreaded implementations of several LtR algorithms[5]. In particular it aims at efficiently training highly effective, production-ready ranking models based on forests of regression trees such as GBRT [8],

[5] QuickRank source code is publicly available for non commercial uses under a Reciprocal Public License 1.5 (RPL-1.5, see `http://opensource.org/licenses/RPL-1.5`) at URL `http://quickrank.isti.cnr.it`.

λ-MART [16] and O-λ-MART [15]. These algorithms represent the state of the art of LtR algorithms [13], and QuickRank is able to train complex models with tens of thousands regressions trees in a few hours. QuickRank is written in C++ (using C++11 and Boost C++[6] features) and it exploits OpenMP[7] to improve runtime performance in multithreaded environments. In addition to speeding up the training phase, the framework is able to produce for the model learned a state-of-the-art C++ implementation of the associated scoring function, ready to be deployed in production environments. Finally, the QuickRank framework is designed to be modular and extensible, so that new machine learning algorithms and new optimized implementations of the scorers can be easily included.

The rest of the paper is organized as follows. Section 2 discusses the main motivations of our work. Section 3 describes the distinguishing features of QuickRank, including the learning algorithms provided, a glimpse of code organization, preliminary usage experiences and experimental results. Finally, Section 4 draws some conclusions and proposes some future directions of research.

2 Motivations and Aims

Besides being the most effective LtR algorithms [13], GBRT, λ-MART and O-λ-MART are computationally expensive solutions at both learning and scoring time.

At learning time they typically produce ensembles involving (tens of) thousands of regression trees. The construction of the ensemble at training time is an iterative and expensive process, where a relatively small regression tree is generated at each iteration. In order to choose the best feature and threshold associated with each tree node, the loss function is evaluated over the possibly huge training. In large-scale WSEs, the adoption of LtR solutions are really effective only if the exploited LtR models are fresh, and reflect large and novel ground truth datasets. Regarding this, we have to consider that WSEs continuously collect, index, and process billions of documents, and process millions of queries per day. During this time, specific documents and query topics may become important for users, due to unpredictable or new events attracting their interests. Therefore LtR models have to be periodically re-trained in order to encompass all these changes, and avoid loss in ranking effectiveness due to the model aging. The time needed to learn a new ranking model thus becomes an important factor to consider in the design of a WSE. In addition, different settings of the LtR algorithms used may result in different performance of the learned models, and a careful parameter tuning can require an expensive training of different models to choose the most effective one.

While the issues discussed above are related to the generation and maintenance of *effective* LtR models, we further motivate QuickRank by considering that WSEs must be very *efficient* at query processing time, thus producing result pages in sub-second times. Since the most effective LtR models are based on

[6] `http://www.boost.org`
[7] `http://www.openmp.org`

ensembles of regression trees, at scoring time the ensemble must be traversed in order to score each single candidate document by accumulating the scores stored in every leaf of the trees reached during the visit.

Therefore, due to the resulting prohibitive ranking cost [15, 3] it is not possible to apply such rankers to all the documents matching a user query in the case of large collections of documents. To overcome this issue, WSEs usually exploit multi-stage ranking architectures, where top-K retrieval is carried out by a two-step process: (i) candidate retrieval and (ii) candidate re-ranking. WSEs partition their huge full index in shards, each managing a subset of documents, which are evenly distributed across many index server nodes which process any query concurrently. The partial results retrieved from each shard are merged at the index server level, and sent to an aggregator to compute the final results [7, 9], as shown in Figure 1.

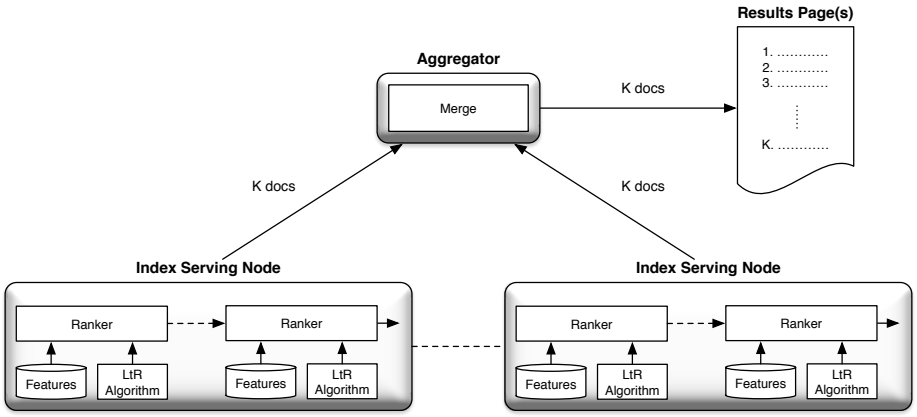

Fig. 1. The architecture of a generic distributed machine-learning Web search engine.

The first step at each index serving node retrieves N possibly relevant documents matching the user query from all the shards of the inverted index managed by the node. 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 [14]. The assumption is that the base ranker is able to efficiently retrieve most of the relevant documents, even if it is not able to effectively rank them. In the second step at each index serving node a complex scoring function is used to re-rank the candidate documents extracted from the shards. A large number of query-document features must be retrieved or computed on the fly during this step, requiring different time constraints. Hence, the second step is possibly composed of a pipeline of machine-learned rankers, collectively called *top ranker*, where each ranking stage exploits a potentially different set of features. LtR models used in this second step are trained to maximize the precision at small cuts, e.g., P@10. In such a partition-aggregate scheme, both the base ranker and the top rankers pipeline must not introduce long latencies in order to achieve good response times at the aggregator. Hence, while the num-

ber of per-shard candidate documents N retrieved by the base ranker usually ranges from $1,000$ to over $10,000$ [6, 5, 4, 12], batches of documents of different sizes may pass through the stages of the top ranker. Eventually, the aggregator typically merges the top-k documents collected in parallel by the index serving nodes and select the globally top-K results.

This architecture requires the training and test of several LtR models. Every model requires an iterative process to be trained, and every model additionally requires several steps of parameters tuning to select the best trade-off between effectiveness and efficiency. The QuickRank framework has been designed and implemented to timely manage the activities for the training and the tuning of such a large set of learning to rank models.

3 The QuickRank Framework

To the best of our knowledge, QuickRank is the first framework addressing both the learning and the scoring processes. Below, we describe these two steps in detail, and introduce the organization of the code. Finally, we report about some experiments aimed at assessing the efficiency and scalability of QuickRank training phase.

Learning Algorithms. QuickRank provides C++ multithreaded implementations of GBRT [8], λ-MART [16], and O-λ-MART [15]. It is worth mentioning that no implementation of the O-λ-MART algorithm was previously made publicly available. For all these algorithms, QuickRank accepts training sets in the SVM-light format and produces an XML file storing the learned ranking model. A short description of the LtR algorithms currently supported by QuickRank is reported below:

Gradient-Boosted Regression Trees (GBRT) [8] 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 practice, 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, 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.

LambdaMART (λ-MART) [16] 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 derivable function. The λ-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 λ, and it can be considered as the gradient of the loss function computed at the given document. Indeed, the λ values are slightly more complex, since they include a factor related to the RankNet [2] cost. This gradient estimation is plugged into a GBRT algorithm, thus obtaining λ-MART. λ-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 λ-MART can be distant from the training relevance labels, as the algorithms aims at finding whatever score generates a good ordering of documents.

Oblivious LambdaMART [15] can be seen as a variation of λ-MART, where oblivious regression trees [10] are used instead of standard regression trees. In oblivious regression trees, the same splitting criterion is used across an entire level of the tree. As a consequence, the resulting trees are balanced and the cost model slightly simplified with respect to the one adopted for forests of regression trees. Regardless this constraint, O-λ-MART provides good performance and it was proven to be less prone to overfitting.

Document Scoring Functions. QuickRank also provides a framework for the cost evaluation *at scoring time* of the learned models. The evaluation is implemented as a two-step process.

During the first step, a tree-based model is converted in a C++ plugin function implementing the scoring of a given document. QuickRank implements three code generation strategies.

The first strategy is named If-Then-Else. Each decision tree is translated into a sequence of C++ if-then-else blocks. If-Then-Else aims at taking advantage of compiler optimization strategies, which can potentially re-arrange the tree ensemble traversal into a more efficient procedure. The size of the resulting code is proportional to the total number of nodes in the ensemble. This makes it impossible to exploit successfully the instruction cache. If-Then-Else was proven to be efficient only with small feature sets [1].

QuickRank also implements the state-of-the-art VPred algorithm, proposed by Asadi *et al.* [1]. The goal of VPred is to reduce the control hazards of the If-Then-Else algorithm caused by the large number of conditional branches. A decision tree of maximum depth d is converted into a d-step traversal of an array storing the tree's nodes. At each step, a node's predicate is tested, and, on the basis of the test's outcome, the next element of the array to be visited is retrieved. The output of QuickRank is a model description file in the format required by the VPred implementation.

The third strategy only applies to O-λ-MART. Recall that an oblivious tree of depth d contains only d distinct predicates and 2^d leaves, containing the possible outcomes. In this case, the d tests are used to set a bitmask of d bits. The integer value of the final bitmask is then exploited to look up the value from a vector of 2^d outcomes, i.e., to select the proper leaf of the oblivious tree.

QuickRank generates a C++ plugin function implementing this scoring strategy, which, by exploiting the properties of oblivious trees, provides more compact data structure and cache-friendly memory access patterns.

During the second step, the generated source code is compiled and linked with the QuickRank framework. The compiled function is invoked by the framework on each document of the given test set, and the total scoring time is measured.

Evaluating the cost of scoring time of a ranking model is important in several application scenarios, and it is receiving more attention in the IR community. We believe that QuickRank can both provide implementation of state-of-the-art algorithms and serve as a a fair evaluation framework.

Code organization. In Figure 2 we show the class diagram of a few classes of the QuickRank framework. We want to highlight the extensibility of the framework to new learning algorithms. Within QuickRank, an LtR algorithm uses the two `Metric` and `Dataset` classes.

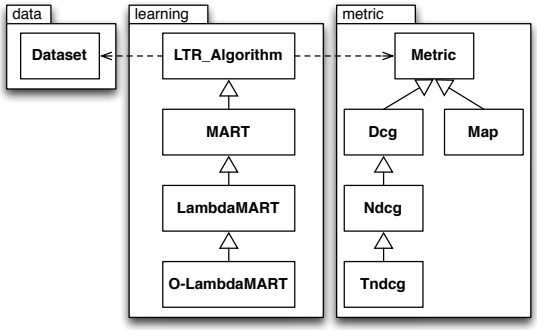

Fig. 2. Class diagram of a subset of QuickRank main classes.

The former implements the IR evaluation measure exploited during the learning process. QuickRank already provides the implementation of MAP, DCG, NDCG and Tied-NDCG. Other measures can be easily provided and automatically used to guide the construction of the ranking model. The latter provides access to the training, evaluation and test datasets. As mentioned above, QuickRank already provides the implementation of GBRT, λ-MART and O-λ-MART algorithms. Additional algorithms can be included in the framework by simply implementing a few methods (such as `learn`, `save`, `load`) or by overriding existing classes. A more detailed documentation is provided at `http://quickrank.isti.cnr.it/doxygen/index.html`.

Experimental assessment. We report about some scalability experiments conducted by using the Yahoo! LETOR[8] challenge Y!S1 dataset. The Y!S1 dataset is publicly available and consists of 19,944 training queries, 2,994 validation queries and 6,983 test queries. Each query is associated with a set (of variable size) of candidate documents represented by vectors of 700 features. The training sam-

[8] `http://learningtorankchallenge.yahoo.com`

ples are 473,134 totally. Query-url pairs in the dataset are labeled with relevance judgments ranging from 0 (irrelevant) to 4 (perfectly relevant).

We present an analysis of the time needed by QuickRank to train LambdaMART models with 1,000 trees, each having up to 16 leaves (learning rate equal to 0.1). Effectiveness results are not reported since they depend only on the learning algorithm and not on its efficiency. Tests are executed on a server equipped with two AMD Opteron^{TM}Processors 6276 (32 cores in total) and 128GiB of RAM. The system runs Ubuntu Server 14.04 LTS with a Linux 3.5.0-49-generic kernel and the GCC 4.8 compiler.

# Threads	Size of Dataset					
	100%		50%		25%	
1	363	(-)	192	(-)	101	(-)
4	114	(3,2x)	63	(3,0x)	35	(2,9x)
8	71	(5x)	42	(4,6x)	24	(4,1x)
16	51	(7x)	31	(6,2x)	19	(5,3x)
32	41	(9x)	25	(7,7x)	16	(6,4x)

Table 1. Performance of QuickRank in terms of training time (minutes) and speedup by varying the number of threads and the size of the training set, i.e., full-size training set, 50% and 25% of it.

Table 1 reports the learning time (minutes) needed by QuickRank to train the λ-MART model by varying the number of threads and the size of the dataset. We experiment 1, 4, 8, 16, and 32 threads. Moreover, performance figures are reported for the full Y!S1 training set (473,134 training samples), 50% of it (236,567 training samples), and 25% of it (118,283 training samples). The results show that by exploiting 32 threads, QuickRank is able to train a λ-MART model of 1,000 trees on the full Y!S1 dataset in about 41 minutes. On the smaller datasets the time required decreases to 25 and 16 minutes, respectively. Even if the speedup grows sub-linearly due to some steps of the λ-MART training algorithm that cannot be parallelized, results confirm that QuickRank can be fruitfully used within a real-world scenario to train machine-learned models on huge datasets. As an example, it is used routinely by Tiscali S.p.A. to train the ranking models used within the istella[9] search engine.

4 Conclusions and Future Work

We presented QuickRank, a C++ suite of efficient and effective LtR algorithms that allows to train high-quality ranking functions from huge training datasets. QuickRank aims at: i) answering Tiscali industrial need for a flexible and scalable LtR solution for learning ranking models from huge training datsets; ii) providing

[9] `http://www.istella.it`

the Information Retrieval research community with a flexible, extensible and efficient LtR framework to design LtR solutions. We presented some preliminary results about the efficiency of QuickRank in training λ-MART models by using the Yahoo! LETOR challenge dataset. As future work, we intend to include in the framework further implementations of relevant LtR algorithms, and develop innovative high-performance solutions addressing the efficiency of both (online) learning and document scoring.

References

1. Asadi, N., Lin, J., de Vries, A.P.: Runtime optimizations for tree-based machine learning models. IEEE Trans. Knowl. Data Eng. 26(9), 2281–2292 (2014)
2. Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.: Learning to rank using gradient descent. In: Proc. ICML. ACM (2005)
3. Cambazoglu, B.B., Zaragoza, H., Chapelle, O., Chen, J., Liao, C., Zheng, Z., Degenhardt, J.: Early exit optimizations for additive machine learned ranking systems. In: Proc. WSDM. pp. 411–420. ACM (2010)
4. Chapelle, O., Chang, Y., Liu, T.Y.: Future directions in learning to rank. In: Yahoo! Learning to Rank Challenge. pp. 91–100 (2011)
5. Craswell, N., Fetterly, D., Najork, M., Robertson, S., Yilmaz, E.: Microsoft Research at TREC 2009: Web and Relevance Feedback Tracks. Tech. rep. (2009)
6. Craswell, N., Robertson, S., Zaragoza, H., Taylor, M.: Relevance weighting for query independent evidence. In: Proc. SIGIR. pp. 416–423. ACM (2005)
7. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM 56(2), 74–80 (Feb 2013), http://doi.acm.org/10.1145/2408776.2408794
8. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Annals of Statistics pp. 1189–1232 (2001)
9. Kim, S., He, Y., Hwang, S.w., Elnikety, S., Choi, S.: Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. In: Proc. WSDM'15. ACM (2015)
10. Kohavi, R.: Bottom-up induction of oblivious read-once decision graphs. In: Proc. ECML. pp. 154–169. Springer (1994)
11. Liu, T.Y.: Learning to rank for information retrieval. Foundations and Trends in Information Retrieval 3(3), 225–331 (2009)
12. Macdonald, C., Santos, R.L., Ounis, I.: The whens and hows of learning to rank for web search. Information Retrieval 16(5), 584–628 (2013)
13. Mohan, A., Chen, Z., Weinberger, K.Q.: Web-search ranking with initialized gradient boosted regression trees. In: Yahoo! Learning to Rank Challenge. pp. 77–89 (2011)
14. Robertson, S., Zaragoza, H.: The probabilistic relevance framework: Bm25 and beyond. Found. Trends Inf. Retr. 3(4), 333–389 (Apr 2009), http://dx.doi.org/10.1561/1500000019
15. Segalovich, I.: Machine learning in search quality at yandex. Invited Talk, SIGIR (2010)
16. Wu, Q., Burges, C., Svore, K., Gao, J.: Adapting boosting for information retrieval measures. Information Retrieval (2010)