A software library for conducting large scale experiments on Learning to Rank algorithms

Nicola Ferro
Dept. of Information Engineering,
University of Padua, Italy
ferro@dei.unipd.it

Paolo Picello
Dept. of Information Engineering,
University of Padua, Italy
paolopicelloit@gmail.com

Gianmaria Silvello
Dept. of Information Engineering,
University of Padua, Italy
silvello@dei.unipd.it

ABSTRACT

This paper presents an efficient application for driving large scale experiments on *Learning to Rank (LtR)* algorithms. We designed a software library that exploits caching mechanisms and efficient data structures to make the execution of massime experiments on LtR algorithms as fast as possible in order to try as many combinations of components as possible. This presented software has been tested on different algorithms as well as on different implementations of the same algorithm in different libraries. This software is highly configurable and extensible in order to enable the seamless addition of new features, algorithms, and libraries.

CCS CONCEPTS

•Information systems →Evaluation of retrieval results; Test collections;

KEYWORDS

learning to rank; component-based evaluation; grid of points

1 INTRODUCTION

LtR is a branch of *Information Retrieval (IR)* that employs machine learning techniques to improve the *effectiveness* of IR systems, by taking as input the ranked result list generated by an IR system and producing as output a new *re-ranked* list of documents [4]. LtR techniques are extremely popular nowadays in IR and they are used by almost all the commercial search engines.

Our long term goal is to study how LtR algorithms behave and interact with other components typically present in an IR systems, such as stemmers or different IR models. To this end, we will rely on and extend our methodology based on the use of *Grid of Points (GoP)* and *General Linear Mixed Model (GLMM)* [1, 2], where a factorial combination of all the components under experimentation is leveraged to estimate the main and interaction effects of the different components as well as their effect size. Therefore, we will basically need to test each LtR algorithm with all the combinations of the other IR system components; if you consider that typically GoPs just combining different stop lists, stemmers and IR models consist of thousands of IR systems [3], you can imagine the explosion in the number of combinations to be tested when you add alternative LtR algorithms on top of them.

Unfortunately, when it comes to test LtR algorithms, they are usually evaluated in *isolation*, i.e. outside of a typical IR system pipeline. Indeed, instead of ranked list of documents, documents

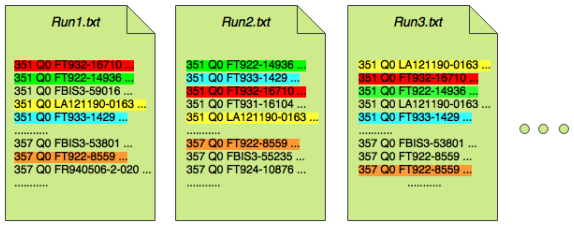

Figure 1: Repetition of query/document pairs within multiple runs.

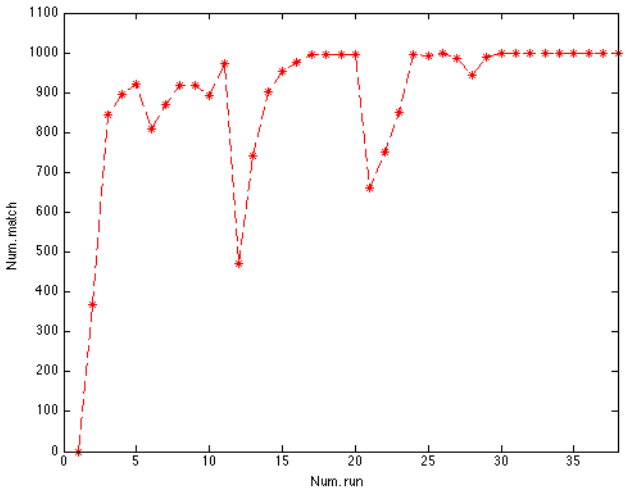

Figure 2: Number of repeated query/document pairs as the number of runs increases.

features, usually in LETOR format [5], are given as input and performance *scores* are directly produced as output.

Even when an LtR library is directly integrated in an IR system, as it happens with JForest[1] and Terrier[2], they are designed to run a single experiment at time and, if you have to run thousands of experiments as in our case, you have to re-start from scratch each time, while the same query/document pair is typically found by many different runs, as shown in Figure 1. As a consequence, these approaches compute the same document and query features again and again with a consequent waste of time and resources.

Figure 2 shows the number of already found query/document pairs, i.e. the number of feature vectors already computed, as the

LEARNER'17, October 1, 2017, Amsterdam, The Netherlands

[1] https://github.com/jeromepaul/jForest
[2] http://terrier.org/

number of considered runs increases. If instead of recomputing these features each time we encounter them again, we somehow cache and re-use them, we will obtain a significant performance improvement. For example, you can note from Figure 2 that, after just 30 runs, we have already computed the features for almost all the possible query/document pairs.

Therefore, our objective is to build an application that allows us to evaluate LtR algorithms performance in an *end-to-end pipeline*, configuring different components and evaluating the re-ranking process as a whole, and that optimizes the costs, in terms of computational load and execution times[3].

The paper is organized as follows: Section 2 describes the proposed solution; Section 3 shows the performance of the proposed solution in terms of execution costs; finally, Section 4 wraps up the discussion and outlooks future work.

2 PROPOSED SOLUTION

The application we propose is modular and presents a logical separation between two successive experimental stages:

- Features Extraction
- Learning To Rank Algorithms Execution

The first module, *Features Extraction*, is responsible of the computation of the features for each query/document pair in the input runs enabling fast retrieval when these features are required by the *Learning To Rank Algorithms Execution* module. This second module is responsible for retrieving the desired features from memory, constructing the required LETOR files, and executing the desired LtR algorithm.

This division of the main tasks allows us to drop down the total execution time and facilitate the separation of the functions in the software. Indeed, the first phase is time consuming because we need to compute the features for all the considered input runs. Afterwards we execute the second module as many times as we want and with different parameters, without re-computing the features. This is where we improve the efficiency of the process.

2.1 Features Extraction

For each query/document pair we calculate the relative features only the first time a pair is processed.

We employ a *caching* mechanism based on a *features matrix* allowing us to store the features for the already computed query/document pairs. In this matrix each line is a document and the columns contain its features. Figure 3 shows this data structure.

We can see that when a document processed for the first time is found in the input run, its features are computed and stored in a matrix. This structure is a `<key,value>` map where the `key` is the document identifier and the `value` is its associated vector of features. Once a new document is found, we need to verify if its entry is already in the table and if it is not we just go on to the next document.

When we need to compute the features for the given document/query pair, first of all we get the relevance judgment from the pool file and we check if it is the first document we consider for the given query. Then, we compute every other required feature and save the computed values in a byte buffer to avoid unnecessary memory occupation. Once all the features have been computed we proceed by populating the feature matrix.

[3]The code is available here https://bitbucket.org/tesisti-unipd/picello as open-source.

2.2 Learning To Rank Algorithms Execution

For each run we have several algorithm/library configurations, but for all these configurations we need the same LETOR files; so, we extract the required features from the previously generated data structure. We parallelize the features extraction process and the LETOR file generation by writing one different LETOR file for each query and then merging these files accordingly to the train/validation/test split specified in configuration parameters.

We realized three different feature extraction alternatives: (i) the first one employs no parallelism and each task is processed sequentially; (ii) the second one employs a different thread for each different task, one thread is responsible for writing train file, one for the validation file and another for the test file; and, (iii) the third one employs a *Thread Pool*, that works like many different threads, but minimizing the overhead due to thread creation. This solution requires a locking mechanism to avoid *readers-writers* problems when different threads access the same file. An advantage of this last approach is that if we change the train/validation/test split we do not need to perform the LETOR extraction phase again, but we only need to perform the faster merging operation.

In the Figure 4 we see the time required for the LETOR creation by the three different approaches. In this case the Thread Pool generates only 4 threads because it was limited by the computer's architecture used for testing. If we employ more threads the gain will be even higher. As we can see, the Thread Pool solution reduces the time by more than 50% w.r.t. the single thread execution.

The execution of this module is repeated for each input run. The steps it follows are: (i) to create the LETOR text files according to the desired train/validation/test split; (ii) to merge the generated text files; and, (iii) execute the LTR algorithms.

3 PERFORMANCE EVALUATION

We conducted the experiment by using a MacBook Air (Mid 2013) with a 1,3 GHz Intel Core i5 3MB cache L3, Hyper-Threading (up to 4 threads) and 4 GB DDR3 a 1600 MHz. We employed Terrier v4.1 for extracting the features and the LtR algorithms reported in Figure 5 where we also report the open-source libraries implementing these algorithms.

As we can see, several algorithms like *MART* or *LambdaMART* are implemented by all the considered libraries, while others as *AdaRank* or *LineSearch* are specific only to a single library. We created a single property file where the parameters of the algorithms are specified. Since different implementations of the same algorithm use different nomenclature, we used a map that associates a parameter value with its corresponding parameter in a given library. As an example, let us consider MART where all the libraries have a different parameter name to indicate the number of trees: `tree` for RankLib, `num-trees` for QuickRank and `boosting.num-trees` for JForests. We gathered all these parameters under the same entry in our `properties` file to simplify the testing phase.

All the tests have been conducted by using the TREC7 corpus (TIPSTER disk 4 and 5 minus CR) and its 50 topics (number 351-400).

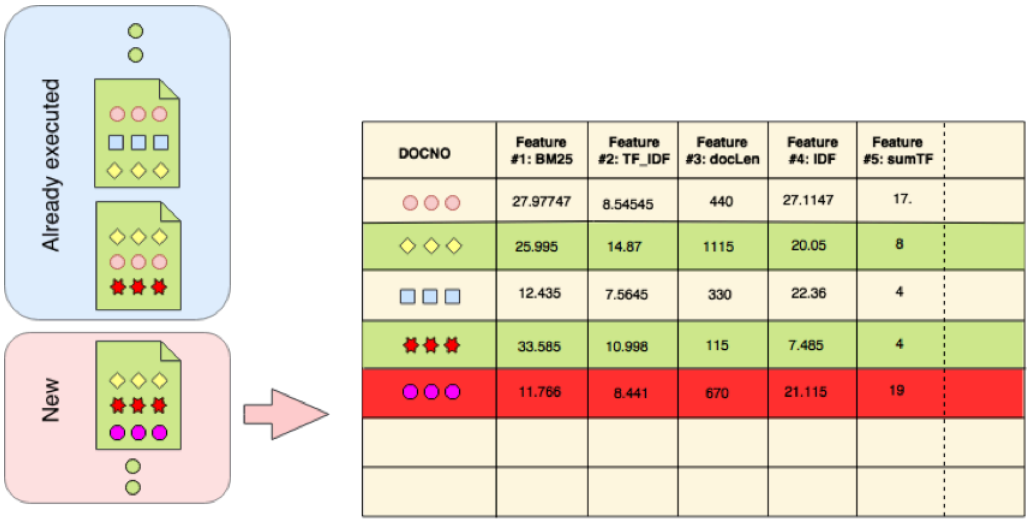

Figure 3: The data structure used to store the calculated features. We see that a new row is added every time a new document is processed.

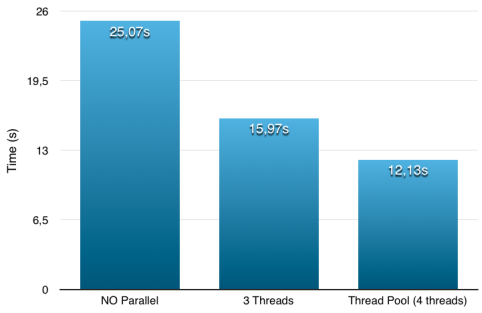

Figure 4: Time for generating LETOR text files with different approaches.

We created a GoP of 1990 runs by using Terrier as detailed in [2]; for each run in this set we performed a re-ranking with all the available LtR algorithms.

3.1 Efficiency

In Figure 6 we see the execution time for about 40 different runs for the same topic. As we could expect from previous considerations, the execution time decreases as the number of processed runs increases, saturating after about 30 runs. For the first run we have the maximum execution time since we need to compute the features for all the documents in the run.

In Table 1, we show the main stages involved in features computation and the respective execution time. This example is about features computation for document FBIS4-33167 for topic 351.

Finally, in Figure 7 we see the total execution time for the Features Extraction phase for the considered topics.

There is an average computation time of about 2, 000 seconds (33 minutes) for each topic. The total execution time for this test was about 99683 seconds (27.68 hours). We recall from above that this computation has to be performed only once for a given set of runs,

Algorithm \ Library	RankLib	QuickRank	JForests
MART	✓	✓	✓
RankNet	✓		
RankBoost	✓	✓	
AdaRank	✓		
Coordinate Ascent	✓	✓	
LambdaMART	✓	✓	✓
ListNet	✓		
RandomForest	✓		
LineSearch		✓	
Gradient Boosting Binary Classifier			✓

Figure 5: Summary of the employed algorithms and the tested open-source libraries implementing them.

Action	Time
Time to get docid from docno	2.0 ms
Time to compute arrays term frequency	61.0 ms
Time to compute arrays TF_IDF	1.0 ms
Time to compute other features	1.0 ms
Time to write whole byte array	12.3 ms
Total time for features computation	68.0 ms

Table 1: Example of execution times for features computation.

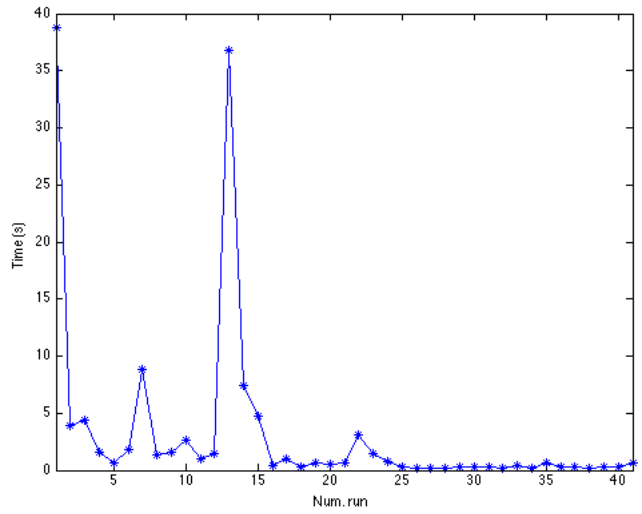

Figure 6: Time for extracting features and generating LETOR files for different runs.

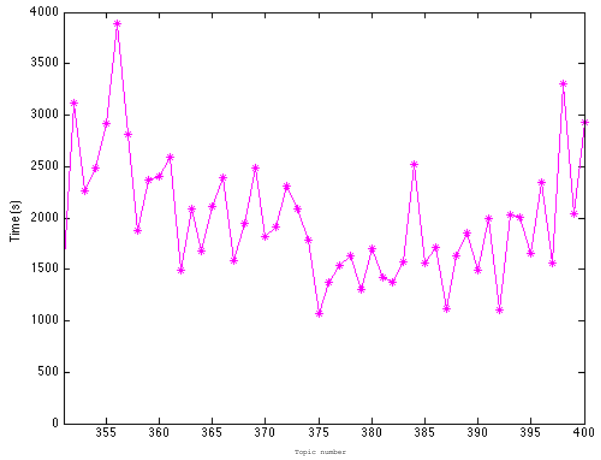

Figure 7: Time for generating LETOR for different topics. On the x-axis there are topic numbers and on the y-axis there is the execution time expressed in seconds.

while the second phase where the LtR algorithms are executed is possibly repeated many times.

For what concerns the LtR algorithms execution module, the execution time depends by the LTR libraries. As an example, in Table 2 we report the execution times of the LtR algorithms by employing their default parameter settings.

3.2 Effectiveness

In this subsection we give an initial glance over the effectiveness performances of the tested LtR and we point out where different implementations of the same algorithm lead to different performances; we leave a deeper and extensive analysis for future works. In Table 3 we report the MAP and precision at cutoff five and ten for a given run re-ranked with different LtR algorithms. The base run employs the `DFIZ` ranking model, the standard Terrier stopword list and a 8-grams lexical unit generator. As we can see

Algorithm	Library	Time(s)
MART	RankLib	10.29
	QuickRank	6.32
	JForests	9.54
LambdaMART	RankLib	37.34
	QuickRank	13.81
	JForests	11.16
ListNet	RankLib	67.27
RankBoost	RankLib	353.79
	QuickRank	208.01
Gradientboostingbinaryclassifier	JForests	10.38
RankNet	RankLib	504.80
Coordinate Ascent	RankLib	201.42
	QuickRank	11.21
AdaRank	RankLib	69.20
Random Forest	RankLib	69.78
LineSearch	QuickRank	12.17

Table 2: Example of execution times of the tested LtR algorithms with default parameters.

Algorithm	Library	MAP	P@5	P@10
Original	-	0.1387	0.5167	0.4583
MART	RankLib	0.1477	0.4333	0.4083
	QuickRank	0.1153	0.3500	0.3333
	JForests	0.1288	0.3667	0.3417
LambdaMART	RankLib	0.1388	0.4167	0.3500
	QuickRank	0.1323	0.4167	0.4083
	JForests	0.1363	0.4500	0.3833
ListNet	RankLib	0.0443	0.0833	0.1167
RankBoost	RankLib	0.1299	0.4167	0.3583
	QuickRank	0.1449	0.4833	0.4333
Gradient Boosting Binary Classifier	JForests	0.1329	0.4000	0.3333
RankNet	RankLib	0.0444	0.1167	0.1000
Coordinate Ascent	RankLib	0.1412	0.4667	0.4083
	QuickRank	0.0306	0.0167	0.0083
AdaRank	RankLib	0.1564	0.5000	0.4500
Random Forest	RankLib	0.1488	0.4333	0.3750
LineSearch	QuickRank	0.0331	0.0167	0.0083

Table 3: MAP, P@5, P@10 and P@15 for different algorithms

`ListNet`, `LineSearch`, `RankNet` and the QuickRank implementation of `Coordinate Ascent` give the lowest results, suggesting that some improves are in order or that they need a thorough parameter tuning phase. In this situation and with default parameters `AdaRank` gives the better results in terms of MAP.

We analyzed the performance of the same algorithm implemented by different libraries in terms of DCG values, to understand if there are differences between different implementations. Again, these are preliminary tests and we report the analysis for a single topic. In particular, we present the results we get for `LambdaMART`, `RankBoost` and `Coordinate Ascent`.

Figure 8 shows the DCG of the `LambdaMART` algorithm for topic 390.

As we can see all the implementations have similar performances. In the case of RankLib's implementation we have some little improvements with respect to the original run. This behaviour is the same for most of the topics.

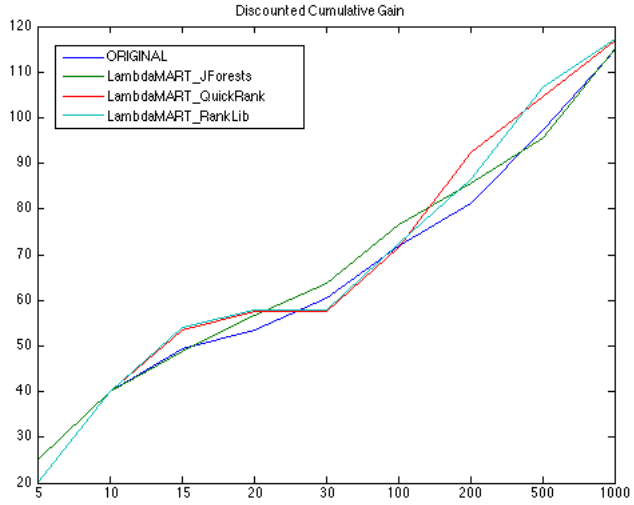

Figure 8: DCG for LambdaMART runs for topic 390.

`RankBoost` is implemented by both QuickRank and RankLib; in Figure 9 we show the DCG curves for the topic 390 where we report also the original run. We can see how QuickRank slightly

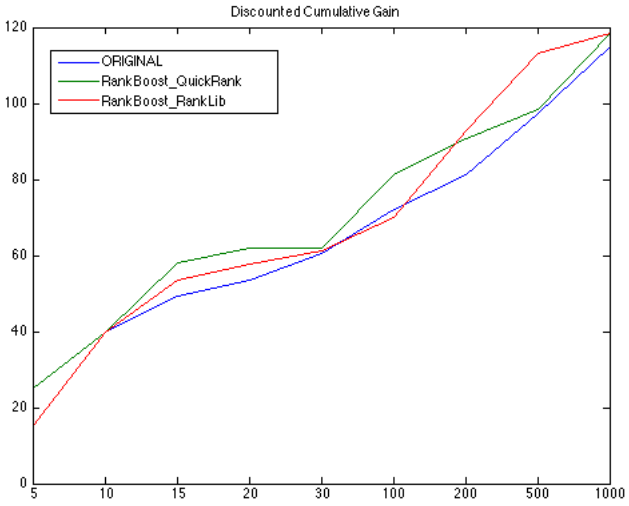

Figure 9: DCG for RankBoost runs for topic 390.

outperforms both RankLib and the original run. In Figure 10 we analyze `Coordinate Ascent`, where RankLib performs similarly to the original run, while QuickRank is slightly worse.

4 FINAL REMARKS

In this paper we described a software library that enables us to run large-scale experiments over many LtR algorithms. We have designed a library that, separating the execution in two different modules, avoid to repeat unnecessary computations. This allows us to efficiently run batch experiments for studying how different parameters affect the results of the models and how the results differ for the same algorithms implemented by different libraries.

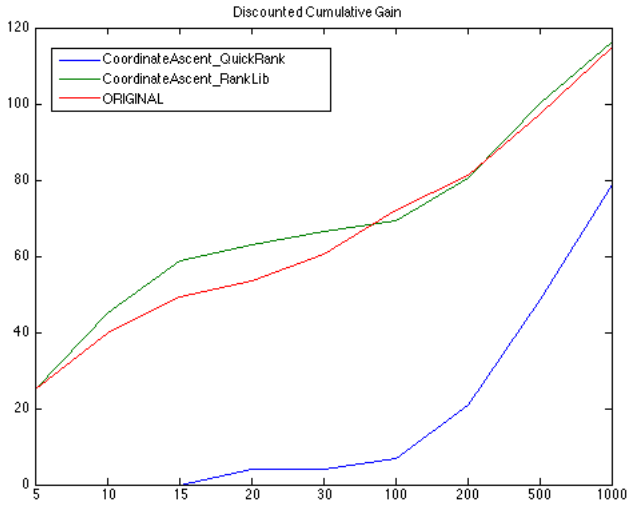

Figure 10: DCG for Coordinate Ascent runs for topic 390.

As future works, one of the first improvements is to employ the Hadoop MapReduce implementation of Terrier to index large document collections in a distributed way [6].

REFERENCES

[1] N. Ferro and D. Harman. 2010. CLEF 2009: Grid@CLEF Pilot Track Overview. In *Multilingual Information Access Evaluation Vol. I Text Retrieval Experiments – Tenth Workshop of the Cross–Language Evaluation Forum (CLEF 2009). Revised Selected Papers*, C. Peters, G. M. Di Nunzio, M. Kurimo, T. Mandl, D. Mostefa, A. Peñas, and G. Roda (Eds.). Lecture Notes in Computer Science (LNCS) 6241, Springer, Heidelberg, Germany, 552–565.

[2] N. Ferro and G. Silvello. 2016. A General Linear Mixed Models Approach to Study System Component Effects. In *Proc. 39th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2016)*, R. Perego, F. Sebastiani, J. Aslam, I. Ruthven, and J. Zobel (Eds.). ACM Press, New York, USA, 25–34.

[3] N. Ferro and G. Silvello. 2017. Towards an Anatomy of IR System Component Performances. *Journal of the American Society for Information Science and Technology (JASIST)* (2017).

[4] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval (FnTIR)* 3, 3 (March 2009), 225–331. http://dx.doi.org/10.1561/1500000016

[5] T.-Y. Y. Liu, J. Xu, T. Qin, W. Xiong, and H. Li. 2007. LETOR: Benchmark Dataset for Research on Learning to Rank for Information Retrieval. In *SIGIR 2007 Workshop on Learning to Rank for Information Retrieval*, T. Joachims, H. Li, T.-Y. Liu, and C. Zhai (Eds.).

[6] R. McCreadie, C. Macdonald, and I. Ounis. 2012. MapReduce Indexing Strategies: Studying Scalability and Efficiency. *Information Processing & Management* 48, 5 (September 2012), 873–888.