Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge

Jimmy Lin[1](✉), Matt Crane[1], Andrew Trotman[2], Jamie Callan[3], Ishan Chattopadhyaya[4], John Foley[5], Grant Ingersoll[4], Craig Macdonald[6], and Sebastiano Vigna[7]

[1] University of Waterloo, Waterloo, Canada
jimmylin@uwaterloo.ca
[2] eBay Inc., San Jose, USA
[3] Carnegie Mellon University, Pittsburgh, USA
[4] Lucidworks, Redwood City, USA
[5] University of Massachusetts Amherst, Amherst, USA
[6] University of Glasgow, Glasgow, UK
[7] Università degli Studi di Milano, Milan, Italy

Abstract. The Open-Source IR Reproducibility Challenge brought together developers of open-source search engines to provide reproducible baselines of their systems in a common environment on Amazon EC2. The product is a repository that contains all code necessary to generate competitive *ad hoc* retrieval baselines, such that with a single script, anyone with a copy of the collection can reproduce the submitted runs. Our vision is that these results would serve as widely accessible points of comparison in future IR research. This project represents an ongoing effort, but we describe the first phase of the challenge that was organized as part of a workshop at SIGIR 2015. We have succeeded modestly so far, achieving our main goals on the Gov2 collection with seven open-source search engines. In this paper, we describe our methodology, share experimental results, and discuss lessons learned as well as next steps.

Keywords: *ad hoc* retrieval · Open-source search engines

1 Introduction

As an empirical discipline, advances in information retrieval research are built on experimental validation of algorithms and techniques. Critical to this process is the notion of a competitive baseline against which proposed contributions are measured. Thus, it stands to reason that the community should have common, widely-available, reproducible baselines to facilitate progress in the field. The Open-Source IR Reproducibility Challenge was designed to address this need.

In typical experimental IR papers, scant attention is usually given to baselines. Authors might write something like "we used BM25 (or query likelihood) as the baseline" without further elaboration. This, of course, is woefully under-specified. For example, Mühleisen et al. [13] reported large differences in effectiveness across

© Springer International Publishing Switzerland 2016
N. Ferro et al. (Eds.): ECIR 2016, LNCS 9626, pp. 408–420, 2016.
DOI: 10.1007/978-3-319-30671-1_30

four systems that all purport to implement BM25. Trotman et al. [17] pointed out that BM25 and query likelihood with Dirichlet smoothing can actually refer to at least half a dozen different variants; in some cases, differences in effectiveness are statistically significant. Furthermore, what are the parameter settings (e.g., k_1 and b for BM25, and μ for Dirichlet smoothing)?

Open-source search engines represent a good step toward reproducibility, but they alone do not solve the problem. Even when the source code is available, there remain many missing details. What version of the software? What configuration parameters? Tokenization? Document cleaning and pre-processing? This list goes on. Glancing through the proceedings of conferences in the field, it is not difficult to find baselines that purport to implement the same scoring model from the same system on the same test collection (by the same research group, even), yet report different results.

Given this state of affairs, how can we trust comparisons to baselines when the baselines themselves are ill-defined? When evaluating the merits of a particular contribution, how can we be confident that the baseline is competitive? Perhaps the effectiveness differences are due to inadvertent configuration errors? This is a worrisome issue, as Armstrong et al. [1] pointed to weak baselines as one reason why *ad hoc* retrieval techniques have not really been improving.

As a standard "sanity check" when presented with a purported baseline, researchers might compare against previously verified results on the same test collection (for example, from TREC proceedings). However, this is time consuming and not much help for researchers who are trying to reproduce the result for their own experiments. The Open-Source IR Reproducibility Challenge aims to solve both problems by bringing together developers of open-source search engines to provide reproducible baselines of their systems in a common execution environment on Amazon's EC2 to support comparability both in terms of effectiveness *and* efficiency. The idea is to gather everything necessary in a repository, such that with a single script, anyone with a copy of the collection can reproduce the submitted runs. Two longer-term goals of this project are to better understand how various aspects of the retrieval pipeline (tokenization, document processing, stopwords, etc.) impact effectiveness and how different query evaluation strategies impact efficiency. Our hope is that by observing how different systems make design and implementation choices, we can arrive at generalizations about particular classes of techniques.

The Open-Source IR Reproducibility Challenge was organized as part of the SIGIR 2015 Workshop on Reproducibility, Inexplicability, and Generalizability of Results (RIGOR). We were able to solicit contributions from the developers of seven open-source search engines and build reproducible baselines for the Gov2 collection. In this respect, we have achieved modest success. Although this project is meant as an ongoing exercise and we continue to expand our efforts, in this paper we share results and lessons learned so far.

2 Methodology

The product of the Open-Source IR Reproducibility Challenge is a repository that contains everything needed to reproduce competitive baselines on standard

IR test collections[1]. As mentioned, the initial phase of our project was organized as part of a workshop at SIGIR 2015: most of the development took place between the acceptance of the workshop proposal and the actual workshop. To begin, we recruited developers of open-source search engines to participate. We emphasize the selection of developers—either individuals who wrote the systems or were otherwise involved in their implementation. This establishes credibility for the quality of the submitted runs. In total, developers from seven open-source systems participated (in alphabetical order): ATIRE [16], Galago [6], Indri [10,12], JASS [9], Lucene [2], MG4J [3], and Terrier [14]. In what follows, we refer to the developer(s) from each system as a separate team.

Once commitments of participation were secured, the group (on a mailing list) discussed the experimental methodology and converged on a set of design decisions. First, the test collection: we wished to work with a collection that was large enough to be interesting, but not too large as to be too unwieldy. The Gov2 collection, with around 25 million documents, seemed appropriate; for evaluation, we have TREC topics 701–850 from 2004 to 2006 [7].

The second major decision concerned the definition of "baseline". Naturally, we would expect different notions by each team, and indeed, in a research paper, the choice of the baseline would naturally depend on the techniques being studied. We sidestepped this potentially thorny issue by pushing the decisions onto the developers. That is, the developers of each system decided what the baselines should be, with this guiding question: "If you read a paper that used your system, what would you like to have seen as the baseline?" This decision allowed the developers to highlight features of their systems as appropriate. As expected, everyone produced bag-of-words baselines, but teams also produced baselines based on term dependence models as well as query expansion.

The third major design decision focused around parameter tuning: proper parameter settings, of course, are critical to effective retrieval. However, we could not converge on an approach that was both "fair" to all participants and feasible in terms of implementation given the workshop deadline. Thus, as a compromise, we settled on building baselines around the default "out of the box" experience—that is, what a naïve user would experience downloading the software and using all the default settings. We realize that in most cases this would yield sub-optimal effectiveness and efficiency, but at least such a decision treated all systems equitably. This is an issue we will revisit in future work.

The actual experiments proceeded as follows: the organizers of the challenge started an EC2 instance[2] and handed credentials to each team in turn. The EC2 instance was configured with a set of standard packages (the union of the needs of all the teams), with the Gov2 collection (stored on Amazon EBS) mounted at a specified location. Each team logged into the instance and implemented their baselines within a common code repository cloned from GitHub. Everyone agreed on a directory structure and naming conventions, and checked in their

[1] https://github.com/lintool/IR-Reproducibility/.

[2] We used the r3.4xlarge instance, with 16 vCPUs and 122 GiB memory, Ubuntu Server 14.04 LTS (HVM).

code when done. The code repository also contains standard evaluation tools (e.g., `trec_eval`) as well as the test collections (topics and qrels).

The final product for each system was an execution script that reproduced the baselines from end to end. Each script followed the same basic pattern: it downloaded the system from a remote location, compiled the code, built one or more indexes, performed one or more experimental runs, and printed evaluation results (both effectiveness and efficiency).

Each team got turns to work with the EC2 instance as described above. Although everyone used the same execution environment, they did not necessarily interact with the same instance, since we shut down and restarted instances to match teams' schedules. There were two main rounds of implementation—all teams committed initial results and then were given a second chance to improve their implementations. The discussion of methodology on the mailing list was interleaved with the implementation efforts, and some of the issues only became apparent after the teams began working.

Once everyone finished their implementations, we executed all scripts for each system from scratch on a "clean" virtual machine instance. This reduced, to the extent practical, the performance variations inherent in virtualized environments. Results from this set of experiments were reported at the SIGIR workshop. Following the workshop, we gave teams the opportunity to refine their implementations further and to address issues discovered during discussions at the workshop and beyond. The set of experiments reported in this paper incorporated all these fixes and was performed in December 2015.

3 System Descriptions

The following provides descriptions of each system, listed in alphabetical order. We adopt the terminology of calling a "count index" one that stores only term frequency information and a "positions index" one that stores term positions.

ATIRE. ATIRE built two indexes, both stemmed using an s-stripping stemmer; in both cases, SGML tags were pruned. The postings lists for both indexes were compressed using variable-byte compression after delta encoding. The first index is a frequency-ordered count index that stores the term frequency (capped at 255), while the second index is an impact-ordered index that stores pre-computed quantized BM25 scores at indexing time [8].

For retrieval, ATIRE used a modified version of BM25 [16] ($k_1 = 0.9$ and $b = 0.4$). Searching on the quantized index reduces ranking to a series of integer additions (rather than floating point calculations in the non-quantized index), which explains the substantial reduction in query latencies we observe.

Galago (Version 3.8). Galago built a count index and a positions index, both stemmed using the Krovetz stemmer and stored in document order. The postings consist of separate segments for documents, counts, and position arrays (if included), with a separate structure for skips every 500 documents or so. The indexes use variable-byte compression with delta encoding for ids and positions. Query evaluation uses the document-at-a-time MaxScore algorithm.

Galago submitted two sets of search results. The first used a query-likelihood model with Dirichlet smoothing ($\mu = 3000$). The second used a sequential dependence model (SDM) based on Markov Random Fields [11]. The SDM features included unigrams, bigrams, and unordered windows of size 8.

Indri (Version 5.9). The Indri index contains both a positions inverted index and `DocumentTerm` vectors (i.e., a forward index). Stopwords were removed and terms were stemmed with the Krovetz stemmer.

Indri submitted two sets of results. The first was a query-likelihood model with Dirichlet smoothing ($\mu = 3000$). The second used a sequential dependence model (SDM) based on Markov Random Fields [11]. The SDM features were unigrams, bigrams, and unordered windows of size 8.

JASS. JASS is a new, lightweight search engine built to explore score-at-a-time query evaluation on quantized indexes and the notion of "anytime" ranking functions [9]. It does not include an indexer but instead post-processes the quantized index built from ATIRE. The reported indexing times include both the ATIRE time to index and the JASS time to derive its index. For retrieval, JASS implements the same scoring model as ATIRE, but requires an additional parameter ρ, the number of postings to process. In the first submitted run, ρ was set to one billion, which equates to exhaustive processing. In the second submitted run, ρ was set to 2.5 million, corresponding to the "10 % of document collection" heuristic proposed by the authors [9].

Lucene (Version 5.2.1). Lucene provided both a count and a positions index. Postings were compressed using variable-byte compression and a variant of delta encoding; in the positions index, frequency and positions information are stored separately. Lucene submitted two runs, one over each index; both used BM25, with the same parameters as in ATIRE ($k_1 = 0.9$ and $b = 0.4$). The `English Analyzer` shipped with Lucene was used with the default settings.

MG4J. MG4J provided an index containing all tokens (defined as maximal subsequences of alphanumerical characters) in the collection stemmed using the Porter2 English stemmer. Instead of traditional gap compression, MG4J uses quasi-succinct indices [18], which provide constant-time skipping and uses the least amount of space among the systems examined.

MG4J submitted three runs. The first used BM25 to provide a baseline for comparison, with $k_1 = 1.2$ and $b = 0.3$. The second run utilized Model B, as described by Boldi et al. [4], which still uses BM25, but returns first the documents containing all query terms, then the documents containing all terms but one, and so on; quasi-succinct indices can evaluate these types of queries very quickly. The third run used Model B+, similar to Model B, but using positions information to generate conjunctive subqueries that are within a window two times the length of the query.

Terrier (Version 4.0). Terrier built three indexes, the count and positions indexes both use the single-pass indexer, while the "Count (inc direct)"—which includes a direct file (i.e., a forward index)—uses a slower classical indexer.

Table 1. Indexing results

System	Type	Size	Time	Threading	Terms	Postings	Tokens
ATIRE	Count	12 GB	41 m	Multi	39.9M	7.0B	26.5B
ATIRE	Count + Quantized	15 GB	59 m	Multi	39.9M	7.0B	26.5B
Galago	Count	15 GB	6 h 32 m	Multi	36.0M	5.7B	-
Galago	Positions	48 GB	26 h 23 m	Multi	36.0M	5.7B	22.3B
Indri	Positions	92 GB	6 h 42 m	Multi	39.2M		23.5B
JASS	ATIRE Quantized	21 GB	1 h 03 m	Multi	39.9M	7.0B	26.5B
Lucene	Count	11 GB	1 h 36 m	Multi	72.9M	5.5B	-
Lucene	Positions	40 GB	2 h 00 m	Multi	72.9M	5.5B	17.8B
MG4J	Count	8 GB	1 h 46 m	Multi	34.9M	5.5B	-
MG4J	Positions	37 GB	2 h 11 m	Multi	34.9M	5.5B	23.1B
Terrier	Count	10 GB	8 h 06 m	Single	15.3M	4.6B	-
Terrier	Count (inc direct)	18 GB	18 h 13 m	Single	15.3M	4.6B	-
Terrier	Positions	36 GB	9 h 44 m	Single	15.3M	4.6B	16.2B

The single-pass indexer builds partial posting lists in memory, which are flushed to disk when memory is exhausted, and merged to create the final inverted index. In contrast, the slower classical indexer builds a direct (forward) index based on the contents of the documents, which is then inverted through multiple passes to create the inverted index. While slower, the classical indexer has the advantage of creating a direct index which is useful for generating effective query expansions. All indexes were stemmed using the Porter stemmer and stopped using a standard stopword list. Both docids and term positions are compressed using gamma delta-gaps, while term frequencies are stored in unary. All of Terrier's indexers are single-threaded.

Terrier submitted four runs. The first was BM25 and used the parameters $k_1 = 1.2$, $k_3 = 8$, and $b = 0.75$ as recommended by Robertson [15]. The second run used the DPH ranking function, which is a hypergeometric parameter-free model from the Divergence from Randomness family of functions. The query expansion in the "DPH + Bo1 QE" was performed using the Bo1 divergence from randomness query expansion model, from which 10 terms were added from 3 pseudo-relevance feedback documents. The final submitted run used positions information in a divergence from randomness model called pBiL, which utilizes sequential dependencies.

4 Results

Indexing results are presented in Table 1, which shows both indexing time, the size of the generated index (1 GB = 10^9 bytes), as well as a few other statistics: the number of terms denotes the vocabulary size, the number of postings is equal to the sum of document frequencies of all terms, and the number of tokens

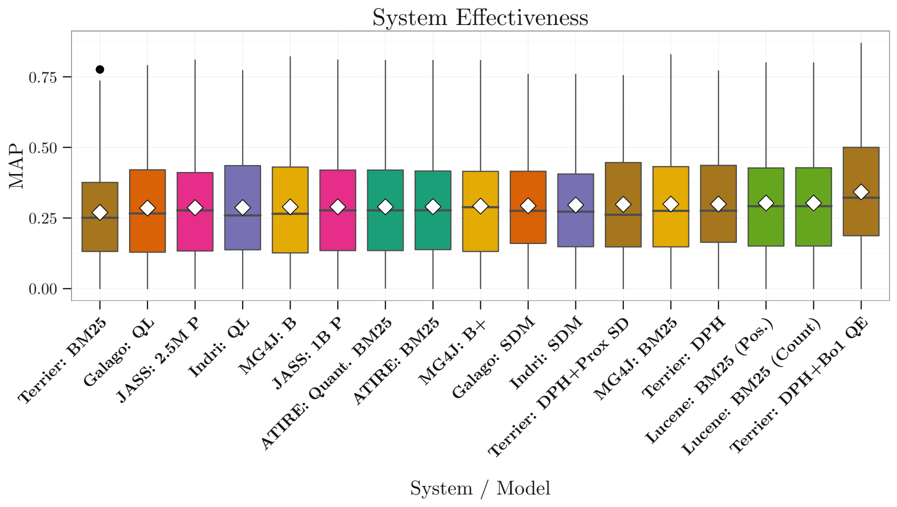

Fig. 1. Box-and-whiskers plot of MAP (all queries) ordered by mean (diamonds).

is the collection length (relevant only for positions indexes). Not surprisingly, for systems that built both positions and count indexes, the positions index took longer to construct. We observe a large variability in the time taken for index construction, some of which can be explained by the use of multiple threads. In terms of index size, it is unsurprising that the positions indexes are larger than the count indexes, but even similar types of indexes differed quite a bit in size, likely due to different tokenization, stemming, stopping, and compression.

Table 2 shows effectiveness results in terms of MAP (at rank 1000). Figure 1 shows the MAP scores for each system on all the topics organized as a box-and-whiskers plot: each box spans the lower and upper quartiles; the bar in the middle represents the median and the white diamond represents the mean. The whiskers extend to 1.5× the inter-quartile range, with values outside of those plotted as points. The colors indicate the system that produced the run.

We see that all the systems exhibit large variability in effectiveness on a topic-by-topic basis. To test for statistical significance of the differences, we used Tukey's HSD (honest significant difference) test with $p < 0.05$ across all 150 queries. We found that the "DPH + Bo1 QE" run of Terrier was statistically significantly better than all other runs and both Lucene runs significantly better than Terrier's BM25 run. All other differences were not significant. Despite the results of the significance tests, we nevertheless note that the systems exhibit a large range in scores, even though from the written descriptions, many of them purport to implement the same model (e.g., BM25). This is true even in the case of systems that share a common "lineage", for example, Indri and Galago. We believe that these differences can be attributed to relatively uninteresting differences in document pre-processing, tokenization, stemming, and stopwords. This further underscores the importance of having reproducible baselines to control for these effects.

Table 2. MAP at rank 1000.

System	Model	Index	Topics			
			701–750	751–800	801–850	All
ATIRE	BM25	Count	0.2616	0.3106	0.2978	0.2902
ATIRE	Quantized BM25	Count + Quantized	0.2603	0.3108	0.2974	0.2897
Galago	QL	Count	0.2776	0.2937	0.2845	0.2853
Galago	SDM	Positions	0.2726	0.2911	0.3161	0.2934
Indri	QL	Positions	0.2597	0.3179	0.2830	0.2870
Indri	SDM	Positions	0.2621	0.3086	0.3165	0.2960
JASS	1B Postings	Count	0.2603	0.3109	0.2972	0.2897
JASS	2.5M Postings	Count	0.2579	0.3053	0.2959	0.2866
Lucene	BM25	Count	0.2684	0.3347	0.3050	0.3029
Lucene	BM25	Positions	0.2684	0.3347	0.3050	0.3029
MG4J	BM25	Count	0.2640	0.3336	0.2999	0.2994
MG4J	Model B	Count	0.2469	0.3207	0.3003	0.2896
MG4J	Model B+	Positions	0.2322	0.3179	0.3257	0.2923
Terrier	BM25	Count	0.2432	0.3039	0.2614	0.2697
Terrier	DPH	Count	0.2768	0.3311	0.2899	0.2994
Terrier	DPH + Bo1 QE	Count (inc direct)	0.3037	0.3742	0.3480	0.3422
Terrier	DPH + Prox SD	Positions	0.2750	0.3297	0.2897	0.2983

Efficiency results are shown in Table 3: we report mean query latency (over three trials). These results represent query execution on a single thread, with timing code contributed by each team. Thus, these figures should be taken with the caveat that not all systems may be measuring exactly the same thing, especially with respect to overhead that is not strictly part of query evaluation (for example, the time to write results to disk). Nevertheless, to our knowledge this is the first large-scale efficiency evaluation of open-source search engines. Previously, studies typically consider only a couple of systems, and different experimental results are difficult to compare due to underlying hardware differences. In our case, a common platform moves us closer towards fair efficiency evaluations across many systems.

Figure 2 shows query evaluation latency in a box-and-whiskers plot, with the same organization as Fig. 1 (note the y axis is in log scale). We observe a large variation in latency: for instance, the fastest systems (JASS and MG4J) achieved a mean latency below 50 ms, while the slowest system (Indri's SDM model) takes substantially longer. It is interesting to note that we observe different amounts of per-topic variability in efficiency. For example, the fastest run (JASS 2.5M Postings) is faster than the second fastest (MG4J Model B) in terms of mean latency, but MG4J is actually faster if we consider the median—the latter is hampered by a number of outlier slow queries.

Table 3. Mean query latency (across three trials).

System	Model	Index	Topics			
			701–750	751–800	801–850	All
ATIRE	BM25	Count	132 ms	175 ms	131 ms	146 ms
ATIRE	Quantized BM25	Count + Quantized	91 ms	93 ms	85 ms	89 ms
Galago	QL	Count	773 ms	807 ms	651 ms	743 ms
Galago	SDM	Positions	4134 ms	5989 ms	4094 ms	4736 ms
Indri	QL	Positions	1252 ms	1516 ms	1163 ms	1310 ms
Indri	SDM	Positions	7631 ms	13077 ms	6712 ms	9140 ms
JASS	1B Postings	Count	53 ms	54 ms	48 ms	51 ms
JASS	2.5M Postings	Count	30 ms	28 ms	28 ms	28 ms
Lucene	BM25	Count	120 ms	107 ms	125 ms	118 ms
Lucene	BM25	Positions	121 ms	109 ms	127 ms	119 ms
MG4J	BM25	Count	348 ms	245 ms	266 ms	287 ms
MG4J	Model B	Count	39 ms	48 ms	36 ms	41 ms
MG4J	Model B+	Positions	91 ms	92 ms	75 ms	86 ms
Terrier	BM25	Count	363 ms	287 ms	306 ms	319 ms
Terrier	DPH	Count	627 ms	421 ms	416 ms	488 ms
Terrier	DPH + Bo1 QE	Count (inc. direct)	1845 ms	1422 ms	1474 ms	1580 ms
Terrier	DPH + Prox SD	Positions	1434 ms	1034 ms	1039 ms	1169 ms

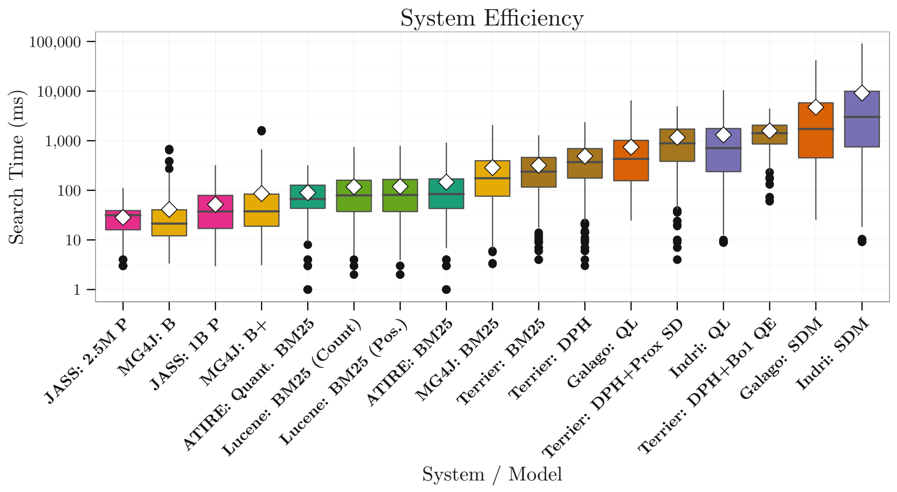

Fig. 2. Box-and-whiskers plot for query latency (all queries); diamonds are means.

Finally, Fig. 3 summarizes effectiveness/efficiency tradeoffs in a scatter plot. As expected, we observe a correlation between effectiveness and efficiency: $R^2 = 0.8888$ after a multi-variate regression of both MAP and system against log(time). Not surprisingly, faster systems tend to compromise quality.

5 Lessons Learned

Overall, we believe that the Open-Source IR Reproducibility Challenge achieved modest success, having accomplished our main goals for the Gov2 test collection. In this section, we share some of the lessons learned.

This exercise was a lot more involved than it would appear and the level of collective effort required was much more than originally expected. We were relying on the volunteer efforts of many teams around the world, which meant that coordinating schedules was difficult to begin with. Nevertheless, the implementations generally took longer than expected. To facilitate scheduling, the organizers asked the teams to estimate how long it would take to build their implementations at the beginning. Invariably, the efforts took more time than the original estimates. This was somewhat surprising because Gov2 is a standard test collection that researchers surely must have previously worked with before.

The reproducibility efforts proved more difficult than imagined for a number of reasons. In at least one case, the exercise revealed a hidden dependency—a pre-processing script that had never been publicly released. In at least two cases, the exercise exposed bugs in systems that were subsequently fixed. In multiple cases, the EC2 instance represented a computing environment that made different assumptions than the machines the teams originally developed on. It seemed that the reproducibility challenge helped the developers improve their systems, which was a nice side effect.

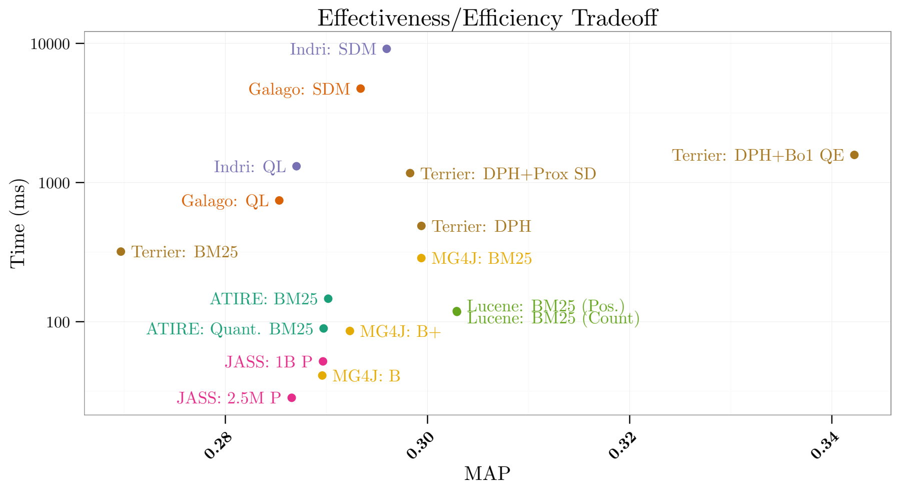

Fig. 3. Tradeoff between effectiveness and efficiency across all systems.

Another unintended consequence of the reproducibility challenge (that was not one of the original goals) is that the code repository serves as a useful teaching resource. In our experience, students new to information retrieval often struggle with basic tasks such as indexing and performing baseline runs. Our resource serves as an introductory tutorial that can teach students about the basics of working with IR test collections: indexing, retrieval, and evaluation.

6 Ongoing Work

The Open-Source IR Reproducibility Challenge is not intended to be a one-off exercise but a living code repository that is maintained and kept up to date. The cost of maintenance should be relatively modest, since we would not expect baselines to rapidly evolve. We hope that sufficient critical mass has been achieved with the current participants to sustain the project. There are a variety of motivations for the teams to remain engaged: developers want to see their systems "used properly" and are generally curious to see how their implementations stack up against their peers. Furthermore, as these baselines begin appearing in research papers, there will be further incentive to keep the code up to date. However, only time will tell if we succeed in the long term.

There are a number of ongoing efforts in the project, the most obvious of which is to build reproducible baselines for other test collections—work has already begun for the ClueWeb collections. We are, of course, always interested in including new systems into the evaluation mix.

Beyond expanding the scope of present efforts, there are two substantive (and related) issues we are currently grappling with. The first concerns the issue of training—from simple parameter tuning (e.g., for BM25) to a complete learning-to-rank setup. In particular, the latter would provide useful baselines for researchers pushing the state of the art in retrieval models. We have not yet converged on a methodology for including "trained" models that is not overly burdensome for developers. For example, would the developers also need to include their training code? And would the scripts need to train the models from scratch? Intuitively, the answer seems to be "yes" to both, but asking developers to contribute code that accomplishes all of this seems overly demanding.

The issue of model training relates to the second issue, which concerns the treatment of external resources. Many retrieval models (particularly in the web context) take advantage of sources such as anchor text, document-level features such as PageRank, spam score, etc. Some of these (e.g., anchor text) can be derived from the raw collection, but others incorporate knowledge outside the collection. How shall we handle such external resources? Since many of them are quite large, it seems impractical to store in our repository, but the alternative of introducing external dependencies increases the chances of errors.

A final direction involves efforts to better understand the factors that impact retrieval effectiveness. For example, we suspect that a large portion of the effectiveness differences we observe can be attributed to different document preprocessing regimes and relatively uninteresting differences in tokenization, stemming, and stopwords. We could explore this hypothesis by, for example, using a single document pre-processor. Such an experiment could be straightforwardly set up by creating a derived collection that every system then ingests, but it would be more efficient and architecturally cleaner to agree on a set of interfaces that allows different retrieval systems to inter-operate. This is similar to the proposal of Buccio et al. [5]: one difference, though, is that we would not prescribe these interfaces, but rather let them evolve based on community consensus. This might perhaps be a fanciful scenario, but the ability to mix-and-match different IR components would greatly accelerate research progress.

The Open-Source IR Reproducibility Challenge represents an ambitious effort to build reproducible baselines for use by the community. Although we have achieved modest success, there is much more to be done. We sincerely encourage participation from the community: both developers in contributing additional systems and everyone in terms of adopting our baselines in their work.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation under IIS-1218043 and by Amazon Web Services. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsors.

References

1. Armstrong, T.G., Moffat, A., Webber, W., Zobel, J.: Improvements that don't add up: Ad-hoc retrieval results since 1998. In: CIKM, pp. 601–610 (2009)
2. Białecki, A., Muir, R., Ingersoll, G.: Apache lucene 4. In: SIGIR 2012 Workshop on Open Source Information Retrieval (2012)
3. Boldi, P., Vigna, S.: MG4J at TREC 2005. In: TREC (2005)
4. Boldi, P., Vigna, S.: MG4J at TREC 2006. In: TREC (2006)
5. Buccio, E.D., Nunzio, G.M.D., Ferro, N., Harman, D., Maistro, M., Silvello, G.: Unfolding off-the-shelf IR systems for reproducibility. In: SIGIR 2015 Workshop on Reproducibility, Inexplicability, and Generalizability of Results (2015)
6. Cartright, M.A., Huston, S., Field, H.: Galago: A modular distributed processing and retrieval system. In: SIGIR 2012 Workshop on Open Source IR (2012)
7. Clarke, C., Craswell, N., Soboroff, I.: Overview of the TREC 2004 terabyte track. In: TREC (2004)
8. Crane, M., Trotman, A., O'Keefe, R.: Maintaining discriminatory power in quantized indexes. In: CIKM, pp. 1221–1224 (2013)
9. Lin, J., Trotman, A.: Anytime ranking for impact-ordered indexes. In: ICTIR, pp. 301–304 (2015)
10. Metzler, D., Croft, W.B.: Combining the language model and inference network approaches to retrieval. Inf. Process. Manage. **40**(5), 735–750 (2004)
11. Metzler, D., Croft, W.B.: A Markov random field model for term dependencies. In: SIGIR, pp. 472–479 (2005)
12. Metzler, D., Strohman, T., Turtle, H., Croft, W.B.: Indri at TREC 2004: Terabyte track. In: TREC (2004)
13. Mühleisen, H., Samar, T., Lin, J., de Vries, A.: Old dogs are great at new tricks: Column stores for IR prototyping. In: SIGIR, pp. 863–866 (2014)

14. Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., Lioma, C.: Terrier: A high performance and scalable information retrieval platform. In: SIGIR 2006 Workshop on Open Source IR (2006)
15. Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M., Gatford, M.: Okapi at TREC-3. In: TREC (1994)
16. Trotman, A., Jia, X.F., Crane, M.: Towards an efficient and effective search engine. In: SIGIR 2012 Workshop on Open Source IR (2012)
17. Trotman, A., Puurula, A., Burgess, B.: Improvements to BM25 and language models examined. In: ADCS, pp. 58–65 (2014)
18. Vigna, S.: Quasi-succinct indices. In: WSDM, pp. 83–92 (2013)