Dockerising Terrier for The Open-Source IR Replicability Challenge (OSIRRC 2019)

Arthur Câmara
A.BarbosaCamara@tudelft.nl
Delft University of Technology
Delft, the Netherlands

Craig Macdonald
craig.macdonald@glasgow.ac.uk
University of Glasgow
Glasgow, UK

ABSTRACT

Reproducibility and replicability are key concepts in science, and it is therefore important for information retrieval (IR) platforms to aid in reproducing and replicating experiments. In this paper, we describe the creation of a Docker container for Terrier within the framework of the OSIRRC 2019 challenge, which allows typical runs to be reproduced on TREC Test Collections such as Robust04, GOV2, Core2018. In doing so, it is hoped that the produced Docker image can be of aid to other (re)producing baseline experiments on these test collections. Initiatives like OSIRRC are key in advancing these key concepts in the IR area. By making not only the source code available, but also the exact same environment and standardising inputs and outputs, it is possible to easily compare approaches and thereby improve the quality of the research for Information Retrieval.

1 OVERVIEW

Terrier (Terabyte Retriever) is an information retrieval (IR) toolkit, initiated by the University of Glasgow, which has been developed since 2001 [5]. It implements a number of retrieval and indexing methods, ready to be used in both research and production.

Given its open source nature, Terrier has been used in a number of papers in the field of IR and others over the years, particularly using standard IR test collections such as those from the Text REtrieval Conference (TREC). For this reason, we agreed to join the OSIRRC 2019 challenge, to create a Docker container image to allow standard baseline results to be obtained using Terrier in a manner that can be easily cross-compared with other platforms implementing the OSIRRC design. Moreover, it is hoped that the produced Docker image can be of aid to others in (re)producing baseline experiments on these test collections.

This paper describes the implementation of the Terrier Docker image. In particular: Section 2 describes the various scripts implemented, as well as the obtained retrieval performances; Section 3 describes the lessons learned in this implementation; Concluding remarks and outlook follow in Section 4.

2 TECHNICAL DESIGN

The nature of the OSIRRC challenge is that implementing systems should provide a Dockerfile that can be used to create a Docker container image that can be run on any Docker installation. The container image is required to implement to a number of "hooks" - simply put, an executable at a known location in the image filesystem. These hooks are then called by the OSIRRC jig, which also mounts any additional files required into the container image (for instance, the corpus files for indexing, or the topic files for retrieval). In the following, we describe the Dockerfile used to build the container image, and the hooks that we implemented for Terrier within the image.

2.1 Dockerfile

The Dockerfile builds a container image with the necessary prerequisites for Terrier. As Terrier is developed in Java, we base the Terrier container image on the OpenJDK standard image for Java 8. A number of other libraries are installed, including Python (for interacting with the OSIRRC jig); gcompat (standard C libraries for trec_eval); and the Jupyter pre-requisites. We chose not to install Terrier using the Dockerfile, to maintain a lightweight container image.

2.2 Standard Hooks

2.2.1 init. This hook is used to prepare the container. We use this hook to download and extract Terrier. We provide example code for both downloading a pre-built "tarball" Terrier from the Terrier.org website, or checking out a version from the Github repository. This hook is configured to use Terrier latest stable version, 5.2. It can, however, be easily configured for fetching other Terrier versions, by changing the variable `version` in the `init` script.

The hook is also compatible with git, making it possible to fetch bleeding-edge versions of Terrier directly from the `terrier-core` Github repository[1]. This can be controlled by the variable `github` in the same `init` script. Note that when using the Github version the *init* hook may take longer to run, since the code will be compiled manually for your system. In our experiments, this could take up to 3 extra minutes.

2.2.2 index. This hook is used to index the corpus that has been mounted by the jig. The jig provides the name of the corpus (see Table 1 for supported corpora), as well as the format (`trectext`, `trecweb` or `json`). The index jig uses the corpora name to configure the indexing process. For example, for the `robust04` corpus which uses TREC Disks 4 & 5, we remove the Congressional Record from the indexing manifest (the `collection.spec` file that lists the files Terrier should index) as well READMEs and other unnecessary files, and configure an additional decompression plugin. Similarly, for the `core18` corpus, we configure Terrier to download[2] an additional indexing plugin to support parsing of the TREC Washington Post corpus.

[1] https://github.com/terrier-org/terrier-core
[2] Indeed, inspired by Apache Spark, since version 5.0, Terrier supports downloading additional plugins from MavenCentral, based on the value of the `terrier.mvn.coords` configuration property.

Name	Description
robust04	TREC Disks 4 & 5, minus CR
gov2	TREC GOV2
core18	TREC Washington Post
cw09b	TREC ClueWeb09 part B
cw12b	TREC ClueWeb12 part B

Table 1: Supported corpora.

Finally, we note that Terrier supports a variety of indexing configurations - we document here our choices and the alternatives available:

- **Positions:** Terrier, by default, does not record positions (also called blocks) in the direct or inverted index. Passing the optional argument of block.indexing to the jig will result in positions being saved. This allows proximity weighting models to be run.
- **Indexer:** Terrier's "classical" indexer creates a direct index (also called a forward index) before inversion of the direct index to create the inverted index. Indeed, the direct index allows pseudo-relevance feedback query expansion. However, the classical indexer is known to be slower than the alternative "single-pass" indexer that Terrier also provides. If the direct index is not required, the single-pass indexer could be used by passing a `-j` flag to Terrier during indexing.
- **Fields:** Including fields in the index allow the frequency of query terms within different parts of the document (e.g. TITLE tags) to be recorded separately. This allows the use of field-based weighting models such as BM25F or PL2F, or use of fields for features within a learning-to-rank setting.
- **Stemming & Stopwords:** We retain Terrier's default setting of Porter Stemming and standard stopword removal. Terrier's stopword list has 733 words.

2.2.3 search. This hooks makes use of Terrier's `batchretrieve` command to execute a 'run', i.e. to extract 1000 results for each of a batch of information needs (topics/queries). The jig mounts the topics into the container image. In addition to learning-to-rank (discussed further below), we provide off-the-shelf support for 12 retrieval configurations. These are broken down by three orthogonal components: weighting model (BM25 [8], as well as PL2 [1] and DPH [2] from the Divergence from Randomness framework); proximity (pBiL Divergence from Randomness model [7]); and query expansion (Bo1 Divergence from Randomness model [1]).

The combination of these three components yields the following possible configurations, to be passed to the hook using the `--opts config=<setting>` parameter:

- BM25
  - `bm25`: Vanilla BM25
  - `bm25_qe`: BM25 with query expansion
  - `bm25_prox`: BM25 with proximity
  - `bm25_prox_qe`: BM25 with proximity and query expansion
- PL2
  - `pl2`: Vanilla PL2
  - `pl2_qe`: PL2 with query expansion
  - `pl2_prox`: PL2 with proximity
  - `pl2_prox_qe`: PL2 with proximity and query expansion
- DPH
  - `dph`: Vanilla DPH
  - `dph_qe`: DPH with query expansion
  - `dph_prox`: DPH with proximity
  - `dph_prox_qe`: DPH with proximity and query expansion

The expected results for each of these runs can be found in Table 2, together with the relative improvements of each configuration over the vanilla version. On analysing the results in the table, it is of interest to note that query expansion always improves the final result considerably, reaching up to 27.90% of improvement over the vanilla versions. However, the same cannot be said about the proximity option. While yielding improvements of up to 3.61% (`bm25_prox`), it can, sometimes, decrease the performance of the vanilla version up to 1.70% (`pl2_prox`). These results are interesting, since they show that different methods can result in diverse observations across the multiple corpora and query sets.

When combining both proximity and query expansion, the results, overall, show improvements over both the vanilla and qe or proximity alone, with improvements of up to 27.26% over the vanilla versions. In the cases where proximity decreased the original results, combining query expansion and proximity search does not improve the results, as expected. However, the results from scenarios like the DPH model on Core18 and GOV2 (topics 701-750) show that, even if both query expansion and proximity search are combined, the overall result may not improve over only the stronger of the two methods (usually, query expansion).

This again reinforces the knowledge that each dataset is different, and that a practitioner should be aware not to simply stack methods that provides marginal gains, but instead test multiple combinations, and understand how each method may behave when used in combination with others.

2.3 Learning To Rank Hooks

Terrier provides support for learning-to-rank in several manners - the ability to integrate additional features during ranking, including additional query dependent features without having to re-traverse the inverted or direct index [6], as well as providing integration of the Jforests [3] implementation of LambdaMART [9].

Learning to Rank integration is demonstrated through two hooks, train and search.

2.3.1 train. This hook extracts features for the training and validation topics, before calling Jforests to build the learned model. To aid implementation, train calls the search hook internally to obtain results for the training and validation sets, specifying the `bm25_ltr_features` retrieval configuration. The retrieval features to use are configurable by specifying the `features` argument to the jig.

2.3.2 search. Search also supports generation of the final learning-to-rank run, using the `bm25_ltr_jforest` retrieval configuration. This configuration assumes that train has already been called and hence a Jforests learned model file already exists.

[3] https://github.com/yasserg/jforests/

Table 2: Expected performance per method and corpus. The best result for each corpus and query set is emphasised.

Method		Robust04	Core18	GOV2		
				701-750	751-800	801-850
BM25	Vanilla	0.2363	0.2326	0.2461	0.3081	0.2629
	+QE	0.2762 (+16.89%)	0.2975 (+27.90%)	0.2621 (+6.50%)	0.3506 (+13.79%)	0.3118 (+18.60%)
	+Proximity	0.2404 (+1.74%)	0.2369 (+1.85%)	0.2537 (+3.09%)	0.3126 (+1.46%)	0.2724 (+3.61%)
	+QE +Proximity	0.2781 (+17.69%)	0.2960 (+27.26%)	0.2715 (+10.32%)	0.3507 (+13.83)	0.3085 (+17.34%)
PL2	Vanilla	0.2241	0.2225	0.2334	0.2884	0.2363
	+QE	0.2538 (+13.25%)	0.2787 (+25.26%)	0.2478 (+6.17%)	0.3160 (+9.57%)	0.2739 (+15.91%)
	+Proximity	0.2283 (+1.87%)	0.2248 (+1.03%)	0.2347 (+0.056%)	0.2835 (-1.70%)	0.2361 (-0.08%)
	+QE +Proximity	0.2575 (+14.90%)	0.2821 (+26.79%)	0.2455 (+5.18%)	0.3095 (+7.32%)	0.2628 (+11.21%)
DPH	Vanilla	0.2479	0.2427	0.2804	0.3311	0.2917
	+QE	0.2821 (+13.80%)	**0.3055 (+25.88%)**	**0.3120 (+11.27%)**	**0.3754 (+13.38%)**	**0.3439 (+17.90%)**
	+Proximity	0.2501 (+0.89%)	0.2428 (+0.04%)	0.2834 (+1.07%)	0.3255 (-1.69%)	0.2904 (-0.45%)
	+QE +Proximity	**0.2869 (+15.73%)**	0.3035 (+25.05%)	0.3064 (+9.27%)	0.3095 (-6.52%)	0.3288 (+12.72)

2.4 Interaction

In the interact hook, we provide three HTTP-accessible methods that allow a researcher to interact with the Terrier instance. Two of these provide access to the results of the search engine, while the third allows the user to conduct further experiments within a Jupyter notebook environment, making use of Terrier-Spark [3]. Each HTTP server is made available on a separate port, as detailed below[4].

2.4.1 Port 1980: Simple search interface. This provides a user-friendly simple web presentation of the search results, allowing the user to enter queries, and receive ranked search results.

2.4.2 Port 1981: REST API. This provides a REST endpoint for Terrier to provide search results from. This can be used directly, or can be used by another instance of Terrier to query the index in a running container (i.e. Terrier can be both a server or a client). Figure 1 shows an example of using Terrier from the command line of another machine to access an index hosted within a Docker container.

2.4.3 Port 1982: Terrier-Spark Jupyter Notebook. Finally, port 1982 starts a Jupyter notebook with Apache Toree's Scala kernel installed. This allows use of Terrier-Spark - a Scala interface built on top of Apache Spark that allows Terrier retrieval experiments to be conducted, including in a Jupypter notebook [3, 4]. An example notebook is provided that allows the user to run more experiments on the available indices. Functionalities include querying and evaluating outcomes (as shown in Figure 2), as well as combining Terrier's learning-to-rank feature support with Apache Spark's machine learning capabilities.

3 LESSONS LEARNED

While developing this work, a number of roadblocks appeared, prompting new insights and workarounds that ended up improving the overall reproducibility of the work. Some of these roadblocks, formulated as questions, are described in this section.

[4]Note that the ports on the host machine may differ, due to the way that Docker assigns ports. It is foreseen that this will be resolved in future versions of the OSIRRC jig - see https://github.com/osirrc/jig/issues/112.

```
#this starts the REST endpoint on port 1981
[dockerhost]$ cd jig
[dockerhost]$ python run.py interact --repo terrier --tag latest
------
#this demonstrates access to that index from another machine
[anotherhost]$ cd terrier
[anotherhost]$ bin/terrier interactive -I http://dockerhost:1981/
terrier query> information retrieval end:5
  Displaying 1-6 results
0 FBIS4-20699 10.268754805435458
1 FBIS4-20702 9.768490153503198
2 FR941027-2-00046 9.491347902606723
3 FBIS4-20701 9.456022500508775
4 FBIS3-24510 9.31403481019499
5 FBIS4-20700 8.792342494849281
```

Figure 1: Accessing an index hosted on the Terrier Docker container via the Terrier REST API.

3.1 Do you really have the original version of the corpus?

We discovered, like several other research labs involved in the OSIRRC challenge, that TREC Disks 4 & 5 had been originally compressed using the archaic Unix `compress` utility, resulting in .z .1z and .2z filename suffices. Our own copies in Glasgow and Delft had at sometime been recompressed using more contemporary Gzip compression (with a resulting .gz filename suffix).

We made some minor adjustments in Terrier version 5.2 that allowed decompression of .z files using an Apache Commons package to be integrated into Terrier on-the-fly.

3.2 How much memory is in this container?

Like any Java process, Terrier is limited in the amount of memory available in the Java Virtual Machine (JVM). We worked hard to ensure that the JVM is allowed to use as much memory once a container is running. This allows Terrier potential speed improvements for both indexing and retrieval.

In [17]:

```
//change this for your topics file
val topicsFile = "file:/path/to/topics.txt"
val qrelsFile = "file:/path/to/qrels.txt"

val topics = TopicSource.extractTRECTopics(topicsFile).toList.toDF("qid", "query").repartition(1)

val r1 = queryTransform.transform(topics)
//r1 is a dataframe with results for queries in topics
val qrelTransform = new QrelTransformer()
    .setQrelsFile(qrelsFile)

val r2 = qrelTransform.transform(r1)
//r2 is a dataframe as r1, but also includes a label column
val ndcg = new RankingEvaluator(Measure.NDCG, 20).evaluateByQuery(r2).toList

val newSchema = StructType(topics.schema.fields ++ Array(StructField("ndcg", DoubleType, false)))
val rtr = spark.createDataFrame(topics.rdd.zipWithIndex.map{ case (row, index) => Row.fromSeq(row.toSeq ++ Array(ndcg(i
```

```
Querying concurrent:/work/indexes/robust04.properties for 250 queries
Got for 242108 results total
We have 311410 qrels
```

Out[17]:

In [18]:

```
%%dataframe
rtr
```

Out[18]:

qid	query	ndcg
301	international organized crime	0.0
302	poliomyelitis and post polio	0.17502679579397282
303	hubble telescope achievements	0.11854207483654515
304	endangered species mammals	0.03829285746486456
305	most dangerous vehicles	0.14376931608695356
306	african civilian deaths	0.08111548628241008
307	new hydroelectric projects	0.16194241901521403
308	implant dentistry	0.252750465141966
309	rap and crime	0.2849008613713492
310	radio waves and brain cancer	0.9157513515137172

Figure 2: An example of evaluating a run from Terrier-Spark

3.3 Can the classical indexer be more aggressive in using the available memory?

In OSIRRC, we elected to default to Terrier's classical indexer, as this allows more flexibility in the index due to the creation of a direct index compared to the faster single-pass indexer. However, it was recognised that the classical indexer had seen less attention in recent years, and hence could be further optimised. In particular, in Terrier 5.2, we made changes to the classical indexer to recognise the available memory, and be more aggressive in its use of that RAM. In particular, we have observed significant efficiency improvements when building a block index for GOV2 (an 11% reduction in indexing time for a Docker host machine with many CPU cores, with larger benefit observed for less powerful hosts).

4 CONCLUSIONS & OUTLOOK

This paper has described the implementation of the Terrier-Docker container image within the OSIRRC replicability challenge. This has been a worthwhile effort that has allowed many IR platforms and toolkits to be made available within a standardised Docker environment. We have aimed to provide a range of standard retrieval configurations that Terrier can provide for the relevant test collections. Meanwhile, participation in the challenge has allowed some improvements to the Terrier platform, that will be released in version 5.2.

On the other hand, while the Docker image is a step in the direction of allowing replication of IR experiments, we believe that it should be combined with a notebook-like environments that facilitate the scripting of advanced experiments. We have provided one example Terrier-Spark notebook, which demonstrates the possible functionality of conducting an IR experiment within a notebook. However, we acknowledge the overheads of operating in a Spark environment (both in efficiency and in code complexity). In the future, we seek better integration of Terrier into a Python environment, to allow easier scripting of complex retrieval experiments.

REFERENCES

[1] Giambattista Amati. 2003. *Probabilistic Models for Information Retrieval based on Divergence from Randomness.* Ph.D. Dissertation. Department of Computing Science, University of Glasgow.

[2] Giambattista Amati. 2006. Frequentist and Bayesian Approach to Information Retrieval. In *ECIR (Lecture Notes in Computer Science)*, Vol. 3936. Springer, 13–24.
[3] Craig Macdonald. 2018. Combining Terrier with Apache Spark to create Agile Experimental Information Retrieval Pipelines. In *SIGIR*. ACM, 1309–1312.
[4] Craig Macdonald, Richard McCreadie, and Iadh Ounis. 2018. Agile Information Retrieval Experimentation with Terrier Notebooks. In *DESIRES (CEUR Workshop Proceedings)*, Vol. 2167. CEUR-WS.org, 54–61.
[5] Craig Macdonald, Richard McCreadie, Rodrygo L. T. Santos, and Iadh Ounis. 2012. From puppy to maturity: Experiences in developing Terrier. *Proc. of OSIR at SIGIR* (2012), 60–63.
[6] Craig Macdonald, Rodrygo L. T. Santos, Iadh Ounis, and Ben He. 2013. About learning models with multiple query-dependent features. *ACM Trans. Inf. Syst.* 31, 3 (2013), 11.
[7] Jie Peng, Craig Macdonald, Ben He, Vassilis Plachouras, and Iadh Ounis. 2007. Incorporating term dependency in the dfr framework. In *SIGIR*. ACM, 843–844.
[8] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389.
[9] Qiang Wu, Chris J. C. Burges, Krysta M. Svore, and Jianfeng Gao. 2008. *Ranking, Boosting, and Model Adaptation.* Technical Report MSR-TR-2008-109. Microsoft.