PyTerrier: Declarative Experimentation in Python from BM25 to Dense Retrieval

Craig Macdonald
University of Glasgow
United Kingdom
craig.macdonald@glasgow.ac.uk

Nicola Tonellotto
University of Pisa
Italy
nicola.tonellotto@unipi.it

Sean MacAveney
University of Glasgow
United Kingdom
sean.macaveney@glasgow.ac.uk

Iadh Ounis
University of Glasgow
United Kingdom
iadh.ounis@glasgow.ac.uk

ABSTRACT

PyTerrier is a Python-based retrieval framework for expressing simple and complex information retrieval (IR) pipelines in a declarative manner. While making use of the long-established Terrier IR platform for basic text indexing and retrieval, its salient utility comes from its expressive Python operators, which allow for individual IR operations to be pipelined and combined in different flexible manners as requested by the search application. Each operation applies a transformation upon a dataframe, while operators are defined with clear semantics in relational algebra. Going further, we have recently expanded the PyTerrier framework to include additional support for state-of-the-art BERT-based text re-rankers (such as EPIC) and dense retrieval implementations (such as ANCE and ColBERT). Transformer pipelines can be tuned and evaluated in a declarative manner. To increase the reusability of this framework as a resource for the IR community, PyTerrier provides easy access to a variety of standard benchmark datasets, including pre-built indices. Finally, we highlight the advantages of such a framework for information retrieval researchers and educators.

CCS CONCEPTS

• **Information systems** → **Information retrieval**.

KEYWORDS

Experimentation, Neural Ranking, Dense Retrieval

ACM Reference Format:
Craig Macdonald, Nicola Tonellotto, Sean MacAveney, and Iadh Ounis. 2021. PyTerrier: Declarative Experimentation in Python from BM25 to Dense Retrieval. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21), November 1–5, 2021, Virtual Event, QLD, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3459637.3482013

CIKM '21, November 1–5, 2021, Virtual Event, QLD, Australia

1 INTRODUCTION

As a programming language, Python has gained enormous popularity in recent years.[1] Indeed, it has a syntax that is easy to understand and write and handy data structure abstractions such as Pandas dataframes. Moreover, it is also readily supported in interactive environments such as Jupyter/Google Colab notebooks. This makes it easy to develop and refine code in Python without heavyweight development environments, while avoiding recompilation cycles (as typified by compiled languages such as Java and C/C++).

Python's popularity and expressiveness have made it extremely popular for conducting machine learning tasks. For instance, both Tensorflow [11] and PyTorch [33] are widely accessible through Python. One advantage is Python's ability to support operator overloading for objects, which allows numerical expressions (such as tensor operations) to be easily expressed in Python using common mathematical operators. As a result, Python is widely used for the implementation of neural networks, including within the information retrieval (IR) community.

At the same time, the growing interest of the IR community in neural-based approaches for investigating and addressing typical and emerging IR tasks (such as ad hoc search and conversational search) have led to a further focus on the development and application of Python-based interfaces that leverage support libraries from existing IR research systems such as Terrier [27] and Anserini [42]. Pyserini [19] has been recently proposed as a Python interface to Anserini, a Java-based search system built on Lucene. Similarly, PyTerrier [28] builds on top of the Java-based Terrier indexing and retrieval platform, but proposes a radically new formalism to express retrieval pipelines in a declarative manner and close to their conceptual design.

With the rise of neural ranking approaches, it has become a common practice to release a codebase for reproducing the training and evaluation of a proposed approach.[2] These repositories usually function as a demonstration of an approach, rather than a tool that can be used for conducting future research. Several toolkits have been built as experimental testbeds for neural IR research, such as MatchZoo [13], OpenNIR [22], Capreolus [43], and OpenMatch [21]. These toolkits differ in their available interfaces (Python and/or command-line) and their capabilities (supported models, indexes,

[1] https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies
[2] https://github.com/microsoft/ANCE, https://github.com/stanford-futuredata/ColBERT, https://github.com/castorini/birch, https://github.com/Georgetown-IR-Lab/cedr, etc.

Table 1: Classes of PyTerrier transformers.

Input		Output	Transformer
Q	$\rightarrow$	Q'	Query rewriting
Q	$\rightarrow$	R	Retrieval
R	$\rightarrow$	Q'	Query expansion
R	$\rightarrow$	R'	Re-ranking
R	$\rightarrow$	R_f	Feature extraction

etc.). However, there is little compatibility between these toolkits and none offers flexible declarative pipelines.

This paper contributes a summary of the key features of the PyTerrier framework for expressing IR experiments in a declarative manner, as well as describing PyTerrier's recent advances in supporting and integrating state-of-the-art neural dense retrieval and re-ranking techniques. Indeed, compared to our first published description of PyTerrier in [28], this paper summarises the central data model and operator notations, but goes further, for instance by demonstrating its support and application to the most recent neural re-rankers and dense retrieval approaches. We also describe recent improvements including a resource website of downloadable pre-built indices that increase the usability and reproducibility of the framework. Some of these features were recently "tutorialised" to attendees at ECIR,[3] and we describe them here to provide a succinct resource for the community that PyTerrier now represents. Indeed, demonstrations of PyTerrier's features are made accessible to the community through interactive Colab notebooks that can be viewed or executed by the interested reader at https://github.com/terrier-org/pyterrier/blob/master/examples/notebooks.md. We have found that PyTerrier is an effective tool for both research and teaching; we summarise our observations in using this framework for IR research, and for the teaching of IR to undergraduate and post-graduate students. In particular, we note Pyterrier's suitability and utility for online teaching during the pandemic.

2 DATA MODEL AND OPERATORS

PyTerrier's fundamental feature is its transparent data model. We use Pandas dataframes (a Python implementation of relations) to represent standard sets of objects in PyTerrier, namely: D, a set of documents; Q, a set of queries; R, a set of documents retrieved for each query; and R_f extends R, with an extra column for features.

Next, PyTerrier's key building blocks are function objects that apply transformations to these dataframes, called transformers.[4] For instance, retrieval can be seen as a transformation of a Q dataframe, containing queries, into a dataframe of retrieved documents (R). Indeed, many common IR operations (other than retrieval) can be characterised as transformations, as summarised by Table 1.

Combining these transformers in different manners is supported through the overloading of Python's math operators for transformer objects. For instance, the binary >> operator (normally used for bit-shift operations on integers) is overloaded to create a pipeline of transformations. The input of >> is two transformers, while the output is a composite transformer, which firstly invokes the first transformer on its own input dataframe, then the second, i.e.:

$$(T_1 >> T_2)(R) := T_2(T_1(R)).$$

Table 2: PyTerrier operators for combining transformers.

Op.	Name	Description
>>	*then*	Pass the output from one transformer to the next transformer
+	*linear combine*	Sum the query-document scores of the two retrieved result lists
*	*scalar product*	Multiply the query-document scores of a retrieved result list by a scalar
**	*feature union*	Combine two retrieved result lists as features
|	*set union*	Make the set union of documents from the two retrieved result lists
&	*set intersection*	Make the set intersection of the two retrieved result lists
%	*rank cutoff*	Shorten a retrieved result list to the first K elements
^	*concatenate*	Add the retrieved result list from one transformer to the bottom of the other

Table 2 summarises the available PyTerrier's transformer operators. These include operations for obtaining linear combinations, as well as combining feature transformers to create a learning-to-rank pipeline. We refer the reader to [28] for a clear definition of the relational algebra semantics of each operator.

Using this notation, retrieval pipelines such as pseudo-relevance feedback become easily expressed in a declarative manner that clearly demonstrates their underlying conceptual behaviour. For example, the composition of BM25 with the RM3 pseudo-relevance feedback model is written in Python as follows:

```
# Compose the retrieval pipeline
bm25 = pt.TerrierRetrieve(index, wmodel="BM25")
prf = bm25 >> pt.rewrite.RM3(index) >> bm25
# Run the pipeline for the specified query
prf.search("query text")
```

3 TRANSFORMER EXAMPLES & USES

In this section, we describe some of the standard and recently integrated transformers, from retrieval & query rewriting operations to the most recent advances in neural re-rankers and dense retrieval.

3.1 Retrieval

To retrieve documents from an existing index with a specific weighting model such as BM25, it is sufficient to instantiate a `TerrierRetrieve` transformer, as `first_pass` in the following example:

```
first_pass = pt.TerrierRetrieve(index, wmodel="BM25")
```

Other possible retrieval implementations are possible – for instance, we also provide one for Anserini, as well as for dense retrieval (as discussed in Section 3.6 below).

[3] https://github.com/terrier-org/ecir2021tutorial [4] We note the use of the transformers name in neural models to represent a family of self-attention based networks.

3.2 Query Rewriting

In many cases, a query might be rewritten by the IR system before being passed to the retrieval component. In the following example, the sequential dependence proximity model [29] transformer adds operators such as the Indri #1 and #uw8 operators containing pairs and sequences of query terms, in order to boost the scores of documents where the query terms appear in close proximity. The output of that transformer can be pipelined with a retrieval transformer to create the new `sdm` composite transformer:

```
sdm = pt.rewrite.SequentialDependence() >> first_pass
```

PyTerrier also includes transformers for pseudo-relevance feedback mechanisms such as RM3 [1] and Bo1 [2].

3.3 Creating Custom Transformers

The extensibility of PyTerrier to create custom transformers is a unique feature among the existing IR toolkits. Recognising that many IR techniques apply a transformation in one of the patterns enumerated in Table 1, we aim to make it possible to write custom transformers as easily as possible, by simply defining a function that takes the query or document and applies the transformation represented by that function. Inspired by Pandas' apply operations, we denote these as *apply transformers*, which allow a user-defined function to be made into a transformer. For instance, to create a URL length feature, we might create a simple lambda (one-line) function, and pass that to `pt.apply.doc_score()`, as follows:

```
url_len = pt.apply.doc_score(lambda row : len(row['url']))
```

The resulting transformer `url_len` can be used in the same way as any other transformer. PyTerrier has several forms of `apply` transformers for operations such as query rewriting and document scoring, as well as arbitrary "generic" transformations.

In doing so, we aim to make it easy for researchers to integrate their techniques into a PyTerrier ranking pipeline. For instance, a neural re-ranker might be integrated through the creation of a Python function that, given the text of a query and the text of a document, returns a relevance score.[5] This can be instantiated into a transformer using `pt.apply.doc_score()`.

3.4 Learning-to-Rank

PyTerrier supports the computation of features to be used in Learning-to-Rank scenarios, including query-dependent features (e.g., weighting models), query-independent features (e.g., spam scores), and query features (e.g., query length). In the following example, the new `fields` transformer computes the per-field scores of the input query-document pairs, and its results are merged with the results produced by the `url_len` transformer through the `**` operator. Then, the instantiation of a learned model can be easily achieved by appending final transformers for learned methods from Scikit-Learn, e.g., Random Forests or LambdaMART [4] (as implemented by XGBoost [5] or LightGBM [16]). The training of the declared model is triggered by the `fit()` functions, by providing as input both training and validation topics and their corresponding relevance judgements, as illustrated below:

```
fields = pt.TerrierRetrieve(index, wmodel="BM25F")
ltr = pt.ltr.apply_learned_model(xgBoost({"rank": "ndcg"}))
ltr_pipe = first_pass >> (fields ** url_len) >> ltr
ltr_pipe.fit(tr_topics, tr_qrels, va_topics, va_qrels)
```

`ltr_pipe` can then be used for search, or indeed evaluation (as discussed in Section 3.7 below). In this way, the three stages [20, 36] of applying a learning-to-rank model in a cascading fashion are visible, separated by `>>` operators: `first_pass` defines the candidate set of documents for each query; `**` defines the features; and finally, the learned model is applied. There is no need to learn an advanced query language, since the entire configuration is clearly expressed in Python. New features can be easily tested though custom transformers, as discussed in Section 3.3.

3.5 Neural Re-ranking

PyTerrier also supports neural re-ranking models from the OpenNIR package [22]. These models can be created from scratch, or loaded from an existing checkpoint file. The neural re-ranking transformers support both training and inference:

```
reranker = onir_pt.reranker('knrm', 'wordvec') # from scratch
reranker = onir_pt.reranker.from_checkpoint( # pre-trained
    '/path/to/tuned_bert_model')
pipeline = first_pass >> pt.text.get_text(index) >> reranker
pipeline.fit(tr_topics, tr_qrels, va_topics, va_qrels)
```

Note that since these models use the document source text, a transformer must be included in the pipeline to retrieve this text from the retrieved documents (c.f. `pt.text.get_text()`). Alternatively, one can instruct `TerrierRetrieve` to include the text, if it was stored in the metadata index.

Through OpenNIR, PyTerrier can easily use pre-trained models from the popular Huggingface Transformers package [40]. A common paradigm is to encode the query and document text jointly (i.e., concatenated) and to score the sequence using the output of a control token (e.g., the so-called $\text{BERT}_{[\text{CLS}]}$ approach, used in works like [25, 30]). In the following example, such a model is constructed from a pre-trained ELECTRA [6] base model by providing the base model name:

```
reranker = onir_pt.reranker('hgf4_joint', ranker_config={
  'model': 'google/electra-base-discriminator'})
```

Some neural re-ranking models benefit at query-time from pre-computing document representations, such as PreTTR [23] and EPIC [24]. To perform this process in PyTerrier, the document collection is first indexed. Then, re-ranker transformers can make use of this index to avoid excessive computation at query time.

```
indexer = onir_pt.indexed_epic.from_checkpoint(
    '/path/to/tuned_epic_model', '/path/to/epic_index')
epic_index = indexer.index(dataset.get_corpus_iter())
epic_pipeline = first_stage >> epic_index.reranker()
```

[5] Clearly this misses the advantages of tighter integration, such as the efficiency benefits brought by the batching of GPU computations. This can be addressed by using the generic `pt.apply.by_query()` instead. We discuss more advanced neural integrations in Sections 3.5 & 3.6 below.

3.6 Dense Retrieval

The advent of BERT-related retrieval models has spawned a new line of research—called *dense passage retrieval* [15]—where the embedded term or document representations are stored directly and used for initial retrieval. We have integrated two such dense retrieval approaches into PyTerrier, namely ANCE [41] and ColBERT [17]. These are provided as separate repositories,[6] which can be installed and used in PyTerrier. Both implementations follow the same syntax as PyTerrier's indexers and retrieval transformers, as shown below:

```
indexer = ANCEIndexer("/path/to/ance_checkpoint/",
                      "/path/to/ance_index")
indexer.index(dataset.get_corpus_iter())
ance = ANCERetrieve("/path/to/ance_checkpoint/",
                    "/path/to/ance_index")
```

```
indexer = ColBERTIndexer("/path/to/colbert_checkpoint",
                         "/path/to/colbert_index")
cfactory = indexer.index(dataset.get_corpus_iter())
colbert_e2e = cfactory.end_to_end(query_encoded=False)
```

Going further, our newly proposed ColBERT PRF approach [38], can easily be expressed as the transformer pipeline shown below:

```
class ColBERT_PRF(TransformerBase):
    def __init__(self, factory, fb_docs, fb_terms):
        # initialisation code
        ...

    def transform(self, topics_and_docs):
        # Input dataframe transformation
        # based on ColBERT PRF
        ...

# reusing factory and colbert_e2e from listing above
colbert_prf = ColBERT_PRF(cfactory, fb_docs=10, fb_terms=5)

cprf_pipe = (colbert_e2e % 10) >> \
            colbert_prf >> \
            (cfactory.end_to_end(query_encoded=True) % 1000)
```

In this way, we demonstrate how new retrieval approaches can be proposed and easily expressed using PyTerrier's pipeline operators.

3.7 Using Transformers

A transformer, composed or otherwise, can be used for conducting a search for a query, to obtain a dataframe of retrieved documents:

```
(bm25 % 10).search("chemical reactions")
```

However, given the focus on experimentation, it is more typical to pass sets of queries through a pipeline:

```
allres = (bm25 % 10).transform(dataset.get_topics())
```

Indeed, the most common use case is to ask PyTerrier to evaluate different retrieval pipelines in a comparative fashion, through the `pt.Experiment()` function, which we describe in the next section.

[6] https://github.com/terrierteam/pyterrier_ance, https://github.com/terrierteam/pyterrier_colbert

4 DECLARATIVE EVALUATION AND TUNING OF TRANSFORMERS

Following our declarative style, PyTerrier seeks to permit evaluation without the need to create result files that are thereafter evaluated using the `trec_eval`[7] tool. In particular, a common experiment might comprise running multiple system variants using a common file containing topics, saving the output in result files, evaluating using the evaluation tool and the qrels file, and then analysing the results (for instance, conducting significance testing).

Instead of this manual and file-centric approach, PyTerrier's `pt.Experiment()` allows an entire evaluation process to be described by specifying the transformer pipelines (systems) to test, the topics to execute, and the criteria for evaluation (including relevance assessments, measures, and significance tests). PyTerrier's declarative nature avoids writing result files, and abstracts away from dealing with the results of transformers directly. Semantically, `pt.Experiment()` executes each transformer pipeline on the specified topics, and measures the effectiveness of the results in terms of IR evaluation metrics. The topics and qrels are provided by PyTerrier's Dataset API, which downloads the topics and qrels before ingesting them as Pandas dataframes. A typical usage of `pt.Experiment()` is illustrated below:

```
pt.Experiment(
    [bm25, sdm, ltr_pipe],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=['map', 'ndcg_cut_10'])
```

A `pt.Experiment()` returns a dataframe of metric values for each input transformer that can be further analysed, for instance plotting interpolated precision-recall graphs, per-query delta bar charts, etc. It also supports as input the use of dataframes containing pre-calculated results - this can be useful for comparing with previous systems or with the results of papers that released system outputs. Significance testing (using a paired t-test by default), and application of correction for multiple comparisons, can also be performed by `pt.Experiment()` – thereby emphasising best practices in IR [12] and simplifying their application by researchers. This is exemplified by the additional `baseline` and `correction` options in the following invocation:

```
# load a dataframe of existing results
bm25_baseline = pt.io.read_results("/path/to/baseline.res")
pt.Experiment(
    [bm25_baseline, sdm],
    dataset.get_topics(),
    dataset.get_qrels(),
    baseline=0, # apply significance testing viz. first system
    correction='b', # apply Bonoferroni testing correction
    eval_metrics=['map', 'ndcg_cut_10'])
```

PyTerrier also showcases a new and simplified way to specify the desired evaluation measures. Specifically, in addition to being able to specify the well-known measure names from `trec_eval` (e.g., '`map`', '`ndcg_cut_10`'), one can provide Python objects, which describe measures and their parameters. For instance, rather than

[7] https://github.com/usnistgov/trec_eval

Table 3: Supported measures from `ir_measures`.

Measure	Name/Description
(M)AP	(Mean) Average Precision
Bpref	Binary Preference
ERR	Expected Reciprocal Rank
infAP	Inferred Average Precision
IPrec@r	Interpolated Precision at r
Judged@k	Judgement rate at cutoff k
nDCG	Normalised Discounted Cumulative Gain
NumQ	Number of queries
NumRel	Number of relevant documents (from qrels)
NumRet	Number of results returned
P(recision)@k	Precision at cutoff k
R(ecall)@k	Recall at cutoff k
RBP	Rank Biased Precision
Rprec	Precision at R (total number of rel docs)
(M)RR	(Mean) Reciprocal Rank
SetAP	Unranked Set Average Precision
SetF	Set F-measure
SetP	Set Precision
SetR	Set Recall
Success@k	Relevant document found in top k

specifying the measure `'ndcg_cut_10'` as a string, it can alternatively be expressed naturally as `nDCG@10`. This functionality works for other measure parameters as well, such as the minimum relevance level for measures that use binary judgements (e.g., by specifying `MAP(rel=2)`). This is particularly useful because the `trec_eval` string-based measure names do not include this option; it is instead a global setting that requires a separate invocation for each relevance level. PyTerrier is able to handle this transparently by invoking the tool multiple times if needed for each requested relevance level. Furthermore, this declarative interface for specifying the evaluation criteria allows PyTerrier to support measures provided by tools other than `trec_eval`, such as such as Expected Reciprocal Rank (`ERR`, from `gdeval`) and Rank Biased Precision (`RBP`, from `cwl-eval`).

This functionality is provided by the standalone *ir_measures* package,[8] which allows others to easily take advantage of this simplified evaluation interface. It also allows the community to easily contribute new measures and implementations. At the time of writing, the software provides access to 20 measures from the `pytrec_eval` [37] tool, `gdeval`,[9] `trectools` [32], `cwl-eval` [3], and the evaluation script for MS MARCO.[10] Table 3 lists the supported measures. In the future, we plan provide a function that provides the "official" evaluation metrics for a dataset, just as one can currently get a dataset's topics and qrels.

It is considered best practice to tune the parameters of unsupervised approaches, such as BM25 and RM3 [1], when they are used as baselines [18]. PyTerrier provides several mechanisms for tuning these models, even when they are used in pipelines. All of these follow a declarative style that has a very similar API to

[8] https://github.com/terrierteam/ir_measures

[9] https://trec.nist.gov/data/web/12/gdeval.pl

[10] https://github.com/microsoft/MSMARCO-Passage-Ranking/blob/master/ms_marco_eval.py

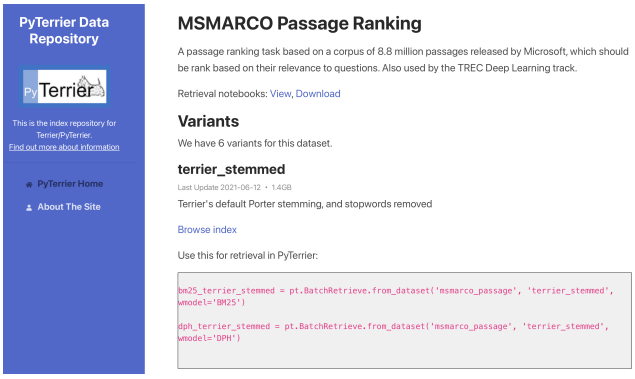

Figure 1: Screenshot of the PyTerrier data repository containing prebuilt indices for MSMARCO.

that of `pt.Experiment`—this aids the user in recalling the API to use. Below we show the `pt.GridSearch` function in action. First, a pipeline is built that retrieves using BM25, performs RM3 expansion, and then retrieves again using the expanded query. The parameters to search are provided as a dictionary, including BM25's b parameter, and the number of feedback docs/terms for RM3.

```
bm25 = pt.TerrierRetrieve(index,
    wmodel="BM25", controls={"bm25.b": 0.5})
rm3 = pt.rewrite.RM3(index, fb_terms=10, fb_docs=3)
pipeline = bm25 >> rm3 >> bm25

pipe_qe = pt.GridSearch(
    pipeline,
    # the range of parameters to search:
    {"bm25" : {"bm25.b": np.linspace(0, 1, 11)},
     "rm3" : {"fb_terms": np.arange(1, 12, 3),
            "fb_docs" : np.arange(2, 30, 6)}},
    dataset.get_topics(), # the topics and qrels to tune with
    dataset.get_qrels(),
    MAP) # search over MAP
```

PyTerrier also supports grid searches over multiple folds, which is beneficial when tuning a model on a test collection without designated training/test partitions.

5 TOWARDS PRE-BUILT INDICES WITH NOTEBOOKS AS A REPRODUCIBLE RESOURCE

Recently, we have focused some of our efforts on providing pre-built indices for commonly-used, downloadable test collections such as MSMARCO. In particular, by providing pre-built indices, we aim to reduce the entry barriers to conducting experiments on these corpora, while also making these readily available as baselines.

Downloads of pre-built indices are conducted automatically from our online data repository, found at http://data.terrier.org. The following example demonstrates the few lines of code necessary to obtain a baseline performance on the TREC 2019 Deep learning track topics of the MSMARCO passage ranking dataset:

```
dataset = pt.get_dataset("msmarco_passage")
bm25 = pt.TerrierRetrieve.from_dataset(dataset, "terrier_stemmed")
pt.Experiment(
    [bm25],
    dataset.get_topics('test-2019'),
    dataset.get_qrels('test-2019'),
    eval_metrics=['ndcg_cut_10'])
```

Indeed, thus far, we have Terrier indices covering MSMARCO passage and document ranking corpora [7], as well as the ANTIQUE QA corpus [14], and the small Vaswani test corpora (11k abstracts).[11] We shortly plan to add support in the PyTerrier plugins for ColBERT and ANCE to allow pre-built dense retrieval indices to be downloaded for some corpora, though the large size of dense indices can preclude the scale of corpora for which these can be reasonably downloaded. For instance, ~ 29GB for an ANCE index of the MSMARCO passage corpus could be reasonably downloaded, however 373GB for a ColBERT index would be too large.

For each corpus, we provide a few standard index types – for instance, with and without Porter stemming, with and without position information (to support phrasal/proximity queries), as well as with and without the raw text of documents (the former to to aid in applying neural re-rankers). Adding text approximately doubles the size of the index. For the MSMARCO corpora, we also provide Terrier indices of the output of the common DeepCT [10] and docT5query [31] neural document expansion techniques. We call each of these settings an index *variant*.

Going further, the PyTerrier Data Repository, for each corpus, provides a page of information that lists the available index variants. An example of such a page is shown in Figure 1. In particular, each page also provides sample PyTerrier code snippets to obtain retrieval pipeline(s) using each index variant. Finally, a complete and pre-executed Jupyter notebook is provided, complete with pipeline and `pt.Experiment()` invocations to demonstrate the effectiveness of the various pipelines on different index variants, and across several topic sets. An example of such a notebook is provided in Figure 2. The example compares sparse retrieval approaches, though we plan to add neural re-ranking baselines to the notebooks in the future. It should be noted that the uniform API of PyTerrier, makes it easy to add corpora, index variants, as well as additional pipelines to these baseline notebooks. Common templates can be used across datasets and allow configuration on a per-corpus, per-variant or per-topic set basis.

6 ADVANTAGES FOR IR RESEARCHERS

We have been using PyTerrier in our research groups for the last 14 months, and our experiences allow us to elicit several advantages for researchers. Firstly, the transformer operators described in Section 2 make it easy to combine different retrieval techniques in different manners. For instance, creating a weighted linear combination of BM25 scores with those from ANCE would be written in an easily understood declarative fashion, e.g., `2 * ance + first_pass`. Furthermore, a pipeline expressed in a declarative way can be evaluated

[11] NB: Due to licensing restrictions, we are not able to provide publicly accessible indices for test collection with more restrictive licenses, such as Robust04 or GOV2.

PyTerrier demonstration for msmarco_passage

In [1]:
```
import pyterrier as pt
if not pt.started():
    pt.init()
systems=[]
names=[]
dataset = pt.get_dataset('msmarco_passage')
```

Systems using index variant terrier_stemmed

In [2]:
```
bm25_terrier_stemmed = pt.BatchRetrieve.from_dataset(
    'msmarco_passage', 'terrier_stemmed', wmodel='BM25')
systems.append(bm25_terrier_stemmed)
names.append('bm25_terrier_stemmed')

dph_terrier_stemmed = pt.BatchRetrieve.from_dataset(
    'msmarco_passage', 'terrier_stemmed', wmodel='DPH')
systems.append(dph_terrier_stemmed)
names.append('dph_terrier_stemmed')
```

Systems using index variant terrier_stemmed_docT5query

In [3]:
```
bm25_terrier_stemmed_docT5query = pt.BatchRetrieve.from_dataset(
    'msmarco_passage', 'terrier_stemmed_docT5query', wmodel='BM25')
systems.append(bm25_terrier_stemmed_docT5query)
names.append('bm25_terrier_stemmed_docT5query')
```

Systems using index variant terrier_stemmed_deepct

In [4]:
```
bm25_terrier_stemmed_deepct = pt.BatchRetrieve.from_dataset(
    'msmarco_passage', 'terrier_stemmed_deepct', wmodel='BM25')
systems.append(bm25_terrier_stemmed_deepct)
names.append('bm25_terrier_stemmed_deepct')
```

Evaluation on trec-2019 topics and qrels

43 topics used in the TREC Deep Learning track Passage Ranking task, with deep judgements

In [5]:
```
pt.Experiment(
    systems,
    dataset.get_topics('test-2019'),
    dataset.get_qrels('test-2019'),
    batch_size=200,
    drop_unused=True,
    eval_metrics=['ndcg_cut_10'],
    names=names)
```

Out[5]:

	name	ndcg_cut_10
0	bm25_terrier_stemmed	0.479540
1	dph_terrier_stemmed	0.502513
2	bm25_terrier_stemmed_docT5query	0.630835
3	bm25_terrier_stemmed_deepct	0.534393

Evaluation on trec-2020 topics and qrels

43 topics used in the TREC Deep Learning track Passage Ranking task, with deep judgements

In [6]:
```
pt.Experiment(
    systems,
    dataset.get_topics('test-2020'),
    dataset.get_qrels('test-2020'),
    batch_size=200,
    drop_unused=True,
    eval_metrics=['ndcg_cut_10'],
    names=names)
```

Out[6]:

	name	ndcg_cut_10
0	bm25_terrier_stemmed	0.493627
1	dph_terrier_stemmed	0.450545
2	bm25_terrier_stemmed_docT5query	0.622808
3	bm25_terrier_stemmed_deepct	0.550106

Evaluation on dev.small topics and qrels

6800 topics with sparse judgements

In [7]:
```
pt.Experiment(
    systems,
    dataset.get_topics('dev.small'),
    dataset.get_qrels('dev.small'),
    batch_size=200,
    drop_unused=True,
    eval_metrics=['recip_rank'],
    names=names)
```

Out[7]:

	name	recip_rank
0	bm25_terrier_stemmed	0.196240
1	dph_terrier_stemmed	0.184982
2	bm25_terrier_stemmed_docT5query	0.284830
3	bm25_terrier_stemmed_deepct	0.254635

Figure 2: Example notebook showing how to load a prebuilt index for the MSMARCO passage collection.

on multiple datasets, allowing to easily investigate the robustness and generalisability of techniques.

PyTerrier has been designed to be entirely usable in Python — this means that experiments can be run within a notebook environment

(such as Jupyter or Google Colab), and without the need to resort to external shell scripting. Indeed, notebooks have great benefits for reproducibility [34]. Going further, we have noted a prevalence within the IR community when distributing source code for recent approaches to provide shell scripts that have to be run in particular manners and orders, with particular inputs, creating various data files. In this way, research is progressing without making it easy to combine an approach into an IR system pipeline, e.g., allowing to mix & match as easily as changing a single transformer variable in an expression. It is our opinion that authors would experience higher impact if there was a clearly defined function to obtain the key output of their approach — for instance, a neural re-ranker should expose a Python function that, given the text of a query and the text of a document, returns a relevance score. This can be easily integrated in PyTerrier, as discussed in Section 3.3. Techniques such as BERT-QE [44], DeepCT [9] and ANCE [41] have required non-trivial efforts for us to integrate, as they are not designed for interactive applications.[12]

We believe that the declarative formulation of experiments that PyTerrier strives for enforces good empirical research practice in information retrieval. Indeed, `pt.Experiment()` guides researchers towards clearly stating their baselines as transformers, and comparing to those baselines for significance testing. We also include correction for multiple testing, as is now recommended in IR [12].

Managing datasets (corpora, topics and qrels) for IR experiments can also be burdensome to researchers. PyTerrier's Dataset API automatically manages a variety of datasets, including those available from the `ir_datasets` library [26]. This has several advantages for researchers. First, it reduces the burden of manually finding and downloading datasets used in their experiments, and ensures that the data is processed properly. In some settings – such as in Google Colab – storage is ephemeral, hence a quick and simple way to get started with these datasets is particularly helpful. The Dataset API also simplifies experimentation on multiple datasets, either for the entire training pipeline, or for transferring learned relevance signals to another dataset. Finally, with a variety of pre-built indices available for several datasets, researchers can skip the indexing process entirely and start running retrieval experiments with ease.

7 ADVANTAGES FOR TEACHING IR

For the first time, we used PyTerrier to deliver the practical component of two IR courses at the University of Glasgow. One course involved a small class (~30 students) of bachelor students, and another was delivered to a large class of master students (~240 students). Both courses were delivered online due to the ongoing pandemic. PyTerrier's ability to run within a notebook environment and without access to dedicated computers proved to be both a key feature and a very popular choice with both cohorts of students. We also used PyTerrier's Dataset API to seamlessly provide the students with the required datasets and indexes to conduct their experiments on Google Colab.

One of the pedagogical appeals of PyTerrier is that it is straightforward to connect concepts taught in the course with the practical implementation details. In particular, PyTerrier offers the students the ability to easily deploy, configure and manipulate the full IR system pipeline from indexing to retrieval through conducting scientific experiments and analysing their results. Indeed, a key advantage of PyTerrier is that students can easily see the stages of an IR system pipeline, and how various IR functionalities operate logically and in practice through a series of transformers. For example, a pseudo-relevance feedback component (PRF) operates after an initial ranking of documents (first pass retrieval), and before another ranking component is deployed (final ranking of documents).

The 20-hour practical coursework required students to use PyTerrier to index and access the indexing structures of several test collections. It then asked the students to conduct a number of experiments using several classical and probabilistic IR weighting models, e.g., TF-IDF, BM25, PL2, with and without pseudo-relevance feedback, and to evaluate their results on several query test sets, including applying significance testing, interpolated precision-recall graphs, per-query performance analysis, etc. They were also asked to implement and evaluate a word2vec-based query expansion technique [35] and to compare it to a classical PRF approach. The final practical coursework component involved students deploying a learning-to-rank approach, comparing it to a non-learned model, as well as adding their own implementations of proximity (inspired by [8]) and URL length features ([39]). The students had then to analyse feature importance, evaluate the effectiveness of their deployed features, conduct statistical significance testing, report a failure analysis, and draw conclusions on what they learnt from conducting the experiments.

The feedback received from the students in relation to their practical work using PyTerrier was overwhelmingly positive. The students reported that they found the use of the framework to be both straightforward and intuitive. From a pedagogical perspective, the students reported that the framework and the conducted exercises allowed them to reinforce the lectures' content and to have a deeper understanding of the taught concepts. The students also commented that the platform encouraged them to further explore and experiment with concepts taught in the course, and to better appreciate the technical implementations of theoretical concepts in IR. Finally, the students praised the quality of the provided Python notebooks, which were deemed to be simple to follow and easy to navigate as well as the quality of the documentation, which they found to compare favourably with other popular Python machine learning kits they are using in other coursework such as scikit-learn. Furthermore, compared to previous cohorts undertaking exercises of similar difficulty using Terrier alone, we have observed higher engagement, completion and attainment rates.[13] Overall, we have been particularly encouraged with the students' feedback, and plan to add a neural network component to the practical exercises building on the recent support of PyTerrier for deep learning approaches in IR.

8 CONCLUSIONS

This paper presented PyTerrier, a tool for building flexible retrieval pipelines. These pipelines are composed in a declarative manner and support many of the latest retrieval techniques though integration with OpenNIR, ColBERT, and ANCE. PyTerrier offers benefits to researchers and educators alike. For researchers, it supports the

[12] We do not entirely blame the respective authors – it is clear that TensorFlow's development patterns are creating an emphasis on developing in a way that is oriented towards batched experimentation rather than interactive usage and integration.

[13] We observed a roughly 8% increase in scores on a comparable assignment.

ability to easily reproduce results, combine methodologies, and conduct experiments in a declarative manner. It also reduces the burden of dataset management and enables effective experimentation on numerous datasets. For educators, it provides a platform for students to easily link theoretical concepts covered in an IR course to a practical implementation. Being able to run easily in a Google Colab environment, it eliminates the need to provide dedicated hardware for students. Overall, we argue that the PyTerrier way of thinking allows researchers to easily expose their work, thereby facilitating progress in IR. Indeed, we have shown that the PyTerrier data model and operators are extensible, since newly proposed IR techniques can be easily integrated as new transformers. Furthermore, we anticipate that PyTerrier pipelines expressed in Python are sufficiently legible to be included in research papers, in the same way that examples of Indri query operators have been included in papers.

ACKNOWLEDGEMENTS

Nicola Tonellotto was partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence). Craig Macdonald, Sean MacAvaney and Iadh Ounis acknowledge EPSRC grant EP/R018634/1: Closed-Loop Data Science for Complex, Computationally- & Data-Intensive Analytics.

REFERENCES

[1] Nasreen Abdul-Jaleel, James Allan, W Bruce Croft, Fernando Diaz, Leah Larkey, Xiaoyan Li, Mark D Smucker, and Courtney Wade. 2004. UMass at TREC 2004: Novelty and HARD. In *Proceedings of TREC*.

[2] Gianni Amati and Cornelis Joost Van Rijsbergen. 2002. Probabilistic Models of Information Retrieval Based on Measuring the Divergence from Randomness. *TOIS* 20, 4 (2002).

[3] Leif Azzopardi, Paul Thomas, and Alistair Moffat. 2019. cwl_eval: An Evaluation Tool for Information Retrieval. In *Proceedings of SIGIR*.

[4] Christopher J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview*. Technical Report MSR-TR-2010-82.

[5] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of SIGKDD*.

[6] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. *ArXiv* abs/2003.10555 (2020).

[7] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, and Daniel Campos. 2020. Overview of the TREC 2020 Deep Learning Track. In *Proceedings of TREC*.

[8] Ronan Cummins and Colm O'Riordan. 2009. Learning in a Pairwise Term-Term Proximity Framework for Information Retrieval. In *Proceedings of SIGIR*.

[9] Zhuyun Dai and Jamie Callan. 2019. Context-Aware Sentence/Passage Term Importance Estimation For First Stage Retrieval. *arXiv 1910.10687* (2019).

[10] Zhuyun Dai and Jamie Callan. 2020. Context-aware Document Term Weighting for Ad-hoc Search. In *Proceedings of WWW*.

[11] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of OSDI*.

[12] Norbert Fuhr. 2020. Proof by Experimentation? Towards Better IR Research. In *Proceedings of SIGIR*.

[13] Jiafeng Guo, Yixing Fan, Xiang Ji, and Xueqi Cheng. 2019. MatchZoo: A Learning, Practicing, and Developing System for Neural Text Matching. In *Proceedings of SIGIR*.

[14] Helia Hashemi, Mohammad Aliannejadi, Hamed Zamani, and W. Croft. 2020. ANTIQUE: A Non-factoid Question Answering Benchmark. In *Proceedings of ECIR*.

[15] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of EMNLP*.

[16] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Proceedings of NeurIPS*.

[17] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of SIGIR*.

[18] Jimmy Lin. 2019. The Neural Hype and Comparisons Against Weak Baselines. *SIGIR Forum* 52, 2 (Jan. 2019), 40–51. https://doi.org/10.1145/3308774.3308781

[19] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: An Easy-to-Use Python Toolkit to Support Replicable IR Research with Sparse and Dense Representations. *arXiv 2102.10073* (2021).

[20] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009).

[21] Zhenghao Liu, Kaitao Zhang, Chenyan Xiong, and Zhiyuan Liu. 2021. OpenMatch: An Open-Source Package for Information Retrieval. *arXiv 2102.00166* (2021).

[22] Sean MacAvaney. 2020. OpenNIR: A Complete Neural Ad-Hoc Ranking Pipeline. *Proceedings of WSDM*.

[23] Sean MacAvaney, Franco Maria Nardini, Raffaele Perego, Nicola Tonellotto, Nazli Goharian, and Ophir Frieder. 2020. Efficient Document Re-Ranking for Transformers by Precomputing Term Representations. In *Proceedings of SIGIR*.

[24] Sean MacAvaney, Franco Maria Nardini, Raffaele Perego, Nicola Tonellotto, Nazli Goharian, and Ophir Frieder. 2020. Expansion via Prediction of Importance with Contextualization. In *Proceedings of SIGIR*.

[25] Sean MacAvaney, Andrew Yates, Arman Cohan, and Nazli Goharian. 2019. CEDR: Contextualized Embeddings for Document Ranking. In *Proceedings of SIGIR*.

[26] Sean MacAvaney, Andrew Yates, Sergey Feldman, Doug Downey, Arman Cohan, and Nazli Goharian. 2021. Simplified Data Wrangling with ir_datasets. In *Proceedings of SIGIR*.

[27] Craig Macdonald, Richard McCreadie, Rodrygo LT Santos, and Iadh Ounis. 2012. From Puppy to Maturity: Experiences in Developing Terrier. *Proceedings of OSIR at SIGIR*.

[28] Craig Macdonald and Nicola Tonellotto. 2020. Declarative Experimentation in Information Retrieval Using PyTerrier. In *Proceedings of ICTIR*.

[29] Donald Metzler and W Bruce Croft. 2005. A Markov Random Field Model for Term Dependencies. In *Proceedings of SIGIR*.

[30] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. *arXiv 1901.04085* (2019).

[31] Rodrigo Nogueira, Jimmy Lin, and AI Epistemic. 2019. From doc2query to docTTTTTquery. *Online preprint* (2019).

[32] Joao Palotti, Harrisen Scells, and Guido Zuccon. 2019. TrecTools: An Open-source Python Library for Information Retrieval Practitioners Involved in TREC-like Campaigns. In *Proceedings of SIGIR*.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of NeurIPS*.

[34] Fernando Perez and Brian E Granger. 2015. *Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science*. Technical Report. http://archive.ipython.org/JupyterGrantNarrative-2015.pdf

[35] Dwaipayan Roy, Debjyoti Paul, Mandar Mitra, and Utpal Garain. 2016. Using Word Embeddings for Automatic Query Expansion. *arXiv 1606.07608* (2016).

[36] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. 2018. Efficient Query Processing for Scalable Web Search. *Foundations and Trends in Information Retrieval* 12, 4-5 (2018).

[37] Christophe Van Gysel and Maarten de Rijke. 2018. Pytrec_eval: An Extremely Fast Python Interface to trec_eval. In *Proceedings of SIGIR*.

[38] Xiao Wang, Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. 2021. Pseudo-Relevance Feedback for Multiple Representation Dense Retrieval. In *Proceedings of ICTIR*.

[39] Thijs Westerveid, Wessel Kraaij, and Djoerd Hiemstra. 2002. Retrieving Web Pages using Content, Links, Urls and Anchors. In *Proceedings of TREC*.

[40] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *ArXiv* abs/1910.03771 (2019).

[41] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. 2020. Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval. *arXiv 2007.00808* (2020).

[42] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proceedings of SIGIR*.

[43] Andrew Yates, Siddhant Arora, Xinyu Zhang, W. Yang, K. M. Jose, and Jimmy Lin. 2020. Capreolus: A Toolkit for End-to-End Neural Ad Hoc Retrieval. *Proceedings of WSDM*.

[44] Zhi Zheng, Kai Hui, Ben He, Xianpei Han, Le Sun, and Andrew Yates. 2020. BERT-QE: Contextualized Query Expansion for Document Re-ranking. In *Proceedings of EMNLP (Findings)*.