Pyserini: A Python Toolkit for Reproducible Information Retrieval Research with Sparse and Dense Representations

Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin,
Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira
David R. Cheriton School of Computer Science
University of Waterloo, Ontario, Canada

ABSTRACT

Pyserini is a Python toolkit for reproducible information retrieval research with sparse and dense representations. It aims to provide effective, reproducible, and easy-to-use first-stage retrieval in a multi-stage ranking architecture. Our toolkit is self-contained as a standard Python package and comes with queries, relevance judgments, pre-built indexes, and evaluation scripts for many commonly used IR test collections. We aim to support, out of the box, the entire research lifecycle of efforts aimed at improving ranking with modern neural approaches. In particular, Pyserini supports sparse retrieval (e.g., BM25 scoring using bag-of-words representations), dense retrieval (e.g., nearest-neighbor search on transformer-encoded representations), as well as hybrid retrieval that integrates both approaches. This paper provides an overview of toolkit features and presents empirical results that illustrate its effectiveness on two popular ranking tasks. Around this toolkit, our group has built a culture of reproducibility through shared norms and tools that enable rigorous automated testing.

CCS CONCEPTS

• **Information systems → Information retrieval**.

KEYWORDS

Open-Source Search Engine; First-Stage Retrieval

1 INTRODUCTION

The advent of pretrained transformers has led to many exciting recent developments in information retrieval [16]. In our view, the two most important research directions are transformer-based re-ranking models and learned dense representations for ranking. Despite many exciting opportunities and rapid research progress, the

need for effective, reproducible baselines has remained a constant. In particular, the importance of stable first-stage retrieval within a multi-stage ranking architecture [5, 25, 34] has become even more important, as it provides the foundation for increasingly-complex modern approaches that leverage hybrid techniques.

We present Pyserini, our Python IR toolkit designed to serve this role: it aims to provide a solid foundation to help researchers pursue work on modern neural approaches to information retrieval. The toolkit is designed to support the complete "research lifecycle" of systems-oriented inquiries aimed at building better ranking models, where "better" can mean more effective, more efficient, or some tradeoff thereof. This typically involves working with one or more standard test collections to design ranking models as part of an end-to-end architecture, iteratively improving components and evaluating the impact of those changes. In this context, our toolkit provides the following key features:

- Pyserini is completely self-contained as a Python package, available via `pip install`. The package comes with queries, collections, and qrels for standard IR test collections, as well as pre-built indexes and evaluation scripts. In short, batteries are included. Pyserini supports, out of the box, the entire research lifecycle of efforts aimed at improving ranking models.
- Pyserini can be used as a standalone module to generate batch retrieval runs or be integrated as a library into an application designed to support interactive retrieval.
- Pyserini supports sparse retrieval (e.g., BM25 scoring using bag-of-words representations), dense retrieval (e.g., nearest-neighbor search on transformer-encoded representations), as well hybrid retrieval that integrates both approaches.
- Pyserini provides access to data structures and system internals to support advanced users. This includes access to postings, document vectors, and raw term statistics that allow our toolkit to support use cases that we had not anticipated.

Pyserini began as the Python interface to Anserini [38, 39], which our group has been developing for several years, with its roots in a community-wide reproducibility exercise dating back to 2015 [15]. Anserini builds on the open-source Lucene search library and was motivated by the desire to better align academic research with the practice of building real-world search applications; see, for example, Grand et al. [9]. More recently, we recognized that Anserini's reliance on the Java Virtual Machine (due to Lucene) greatly limited its reach [2, 3], as Python has emerged as the language of choice for both data scientists and researchers. This is particularly the case for work on deep learning today, since the major toolkits (PyTorch [29] and Tensorflow [1]) have both adopted Python as their front-end

language. Thus, Pyserini aims to be a "feature-complete" Python interface to Anserini.

Sparse retrieval support in Pyserini comes entirely from Lucene (via Anserini). To support dense and hybrid retrieval, Pyserini integrates Facebook's Faiss library for efficient similarity search over dense vectors [12], which in turns integrates the HNSW library [23] to support low-latency querying. Thus, Pyserini provides a superset of features in Anserini; dense and hybrid retrieval capabilities are entirely missing from the latter.

This paper is organized as follows: After a preamble on our design philosophy, we continue with a tour of Pyserini, highlighting its main features and providing the reader with a sense of how it might be used in a number of common scenarios. This is followed by a presentation of empirical results illustrating the use of Pyserini to provide first-stage retrieval for two popular ranking tasks today.

2 DESIGN PHILOSOPHY

The design of Pyserini emphasizes ease of use and reproducibility. Larry Wall, the creator of the Perl programming language, once remarked that "easy things should be easy, and hard things should be possible." While aspects of the lifecycle for systems-oriented IR research are not difficult per se, there are many details that need to be managed: downloading the right version of a corpus, building indexes with the appropriate settings (tokenization, stopwords, etc.), downloading queries and relevance judgments (deciding between available "variants"), manipulating runs into the correct output format for the evaluation script, selecting the right metrics to obtain meaningful results, etc. The list goes on. These myriad details often trip up new researchers who are just learning systems-oriented IR evaluation methodology (motivating work such as Akkalyoncu Yilmaz et al. [2]), and occasionally subtle issues confuse experienced researchers as well.[1] The explicit goal of Pyserini is to make these "easy things" easy, supporting common tasks and reducing the possibility of confusion as much as possible.

At the other end of the spectrum, "hard things should be possible". In our context, this means that Pyserini provides access to data structures and system internals to support researchers who may use our toolkit in ways we had not anticipated. For sparse retrieval, the Lucene search library that underlies Anserini provides interfaces to control various aspects of indexing and retrieval, and Pyserini exposes a subset of features that we anticipate will be useful for IR researchers. These include, for example, traversing postings lists to access raw term statistics, manipulating document vectors to reconstruct term weights, and fine-grained control over document processing (tokenization, stemming, stopword removal, etc.). Pyserini aims to sufficiently expose Lucene internals to make "hard things" possible.

There is a long tradition in the information retrieval community of sharing open-source search engines, which dates back to Cornell's SMART system from the mid-1980s. Today, there is a vibrant ecosystem of search engines built by academic research groups such as Terrier [21] and PISA [24], common infrastructure such as ir_datasets [20], and neural ranking libraries as Capreolus [40, 41], OpenNIR [19], and OpenMatch [36].

There are two main aspects of our design philosophy that set Pyserini apart from other available packages: First, Pyserini explicitly aims to bridge the gap between academic information retrieval research and the practice of building real-world search applications. This is accomplished by its reliance on Lucene for sparse retrieval, which also powers industry-standard search platforms such as Elasticsearch and Solr. Such an alignment means that academic innovations built with Pyserini can be transitioned into production-ready deployments with relative ease. Second, Pyserini does not aspire to be a complete end-to-end neural search engine, but is rather meant to provide a building block as part of a larger software stack. This stands in contrast to, for example, PyTerrier, which aims to provide end-to-end search capabilities via a domain-specific language that allows advanced retrieval pipelines to be expressed in a declarative manner [22].

Finally, the focal use case of Pyserini as first-stage retrieval in a multi-stage ranking architecture means that reproducibility is of utmost concern, since it is literally the foundation that complex reranking pipelines are built on. In our view, reproducibility can be divided into technical and social aspects: an example of the former is an internal end-to-end regression framework that automatically validates experimental results. The latter includes a commitment to "eat our own dog food" and the adoption of shared norms.

3 PYSERINI TOUR

Pyserini is packaged as a Python module available on the Python Package Index. Thus, the toolkit can be installed via pip, as follows:

```
$ pip install pyserini==0.12.0
```

In this paper, we are explicitly using v0.12.0, which is presently our most recent release. Readers are advised, however, to use the latest version available at pyserini.io. For users who may be interested in contributing to Pyserini, we recommend a "development" installation, i.e., cloning the source repository itself. However, for researchers interested only in *using* Pyserini, the module installed via pip suffices.

In this section, we will mostly use the MS MARCO passage ranking dataset [6] as our running example. The dataset has many features that make it ideal for highlighting various aspects of our toolkit: the corpus, queries, and relevance judgments are all freely downloadable; the corpus is manageable in size and thus experiments require only modest compute resources (and time); the task is popular and thus well-studied by many researchers.

3.1 Interactive Retrieval

In Figure 1, we begin with a simple example of using Pyserini to perform bag-of-words ranking with BM25 (the default ranking model) on the MS MARCO passage corpus (comprising 8.8M passages). To establish a parallel with "dense retrieval" techniques using learned transformer-based representations (see below), we refer to this as "sparse retrieval", although this is not common parlance in the IR community at present.

The SimpleSearcher class provides a single point of entry for sparse retrieval functionality. In (L3), we initialize the searcher with a pre-built index. For many commonly used collections where there are no data distribution restrictions, we have built indexes that can

---

[1] As a concrete example, TREC-COVID has (at least) 12 different sets of qrels. All of them are useful for answering different research questions. Which one do you use?

```
1  from pyserini.search import SimpleSearcher
2
3  searcher = SimpleSearcher.from_prebuilt_index('msmarco-passage')
4  hits = searcher.search('what_is_a_lobster_roll?', 10)
5
6  for i in range(0, 10):
7      print(f'{i+1:2}_{hits[i].docid:7}_{hits[i].score:.5f}')
```

Figure 1: Simple example of interactive sparse retrieval (i.e., bag-of-word BM25 ranking).

```
1  from pyserini.dsearch import SimpleDenseSearcher, \
2                                TctColBertQueryEncoder
3
4  encoder = TctColBertQueryEncoder('castorini/tct_colbert-msmarco')
5  searcher = SimpleDenseSearcher.from_prebuilt_index(
6      'msmarco-passage-tct_colbert-hnsw',
7      encoder
8  )
9  hits = searcher.search('what_is_a_lobster_roll')
```

Figure 2: Simple example of interactive dense retrieval (i.e., approximate nearest-neighbor search on dense learned representations).

be directly downloaded from our project servers. For researchers who simply want an "out-of-the-box" keyword retrieval baseline, this provides a simple starting point. Specifically, the researcher does not need to download the collection and build the index from scratch. In this case, the complete index, which includes a copy of all the texts, is a modest 2.6GB.

Using an instance of SimpleSearcher, we issue a query to retrieve the top 10 hits (L4), the results of which are stored in the array hits. Naturally, there are methods to control ranking behavior, such as setting BM25 parameters and enabling the use of pseudo-relevance feedback, but for space considerations these options are not shown here. In (L6–7), we iterate through the results and print out rank, docid, and score. If desired, the actual text can be fetched from the index (e.g., to feed a downstream reranker).

Figure 2 shows an example of interactive retrieval using dense learned representations. Here, we are using TCT-ColBERT [17], a model our group has constructed from ColBERT [14] using knowledge distillation. As with sparse retrieval, we provide pre-built indexes that can be directly downloaded from our project servers. In this case, the pre-built dense index is 25GB, which is substantially larger than the sparse index; this is a known weakness of dense retrieval techniques. The SimpleDenseSearcher class serves as the entry point to nearest-neighbor search functionality that provides top k retrieval on dense vectors. Here, we are taking advantage of HNSW [23], which has been integrated into Faiss [12] to enable low latency interactive querying (L6).

The final component needed for dense retrieval is a query encoder that converts user queries into the same representation space as the documents. We initialize the query encoder in (L4), which is passed into the method that constructs the searcher. The encoder itself is a lightweight wrapper around the Transformers library by HuggingFace [35]. Retrieval is performed in the same manner (L9), and we can manipulate the returned hits array in a manner similar to sparse retrieval (Figure 1). In addition to TCT-ColBERT [17], we presently support a number of other models, including DPR [13], ANCE [37], and DistilBERT KD [10]. Note that our goal here is

```
1  from pyserini.search import SimpleSearcher
2  from pyserini.dsearch import SimpleDenseSearcher, \
3                                TctColBertQueryEncoder
4  from pyserini.hsearch import HybridSearcher
5
6  ssearcher = SimpleSearcher.from_prebuilt_index('msmarco-passage')
7  encoder = TctColBertQueryEncoder('castorini/tct_colbert-msmarco')
8  dsearcher = SimpleDenseSearcher.from_prebuilt_index(
9      'msmarco-passage-tct_colbert-hnsw',
10     encoder
11 )
12 hsearcher = HybridSearcher(dsearcher, ssearcher)
13 hits = hsearcher.search('what_is_a_lobster_roll', 10)
```

Figure 3: Simple example of interactive search with hybrid sparse–dense retrieval.

```
1  from pyserini.search import get_topics, get_qrels
2
3  topics = get_topics('msmarco-passage-dev-subset')
4  qrels = get_qrels('msmarco-passage-dev-subset')
5
6  # Compute the average length of queries:
7  sum([len(topics[t]['title'].split()) for t in topics])/len(topics)
8
9  # Compute the average number of relevance judgments per query:
10 sum([len(qrels[t]) for t in topics])/len(topics)
```

Figure 4: Simple example of working with queries and qrels from the MS MARCO passage ranking test collection.

to provide retrieval capabilities based on existing encoder models; quite explicitly, representational learning lies outside the scope of our toolkit.

Of course, the next step is to combine sparse and dense retrieval, which is shown in Figure 3. Our HybridSearcher takes as its constructor the sparse retriever and the dense retriever and performs weighted interpolation on the individual results to arrive at a final ranking. This is a standard approach and Pyserini adopts the specific implementation in TCT-ColBERT [17], but similar techniques are used elsewhere as well [13, 18].

3.2 Test Collections

Beyond the corpus, topics (queries) and relevance judgments (qrels) form indispensable components of IR test collections to support systems-oriented research aimed at producing better ranking models. Many topics and relevance judgments are freely available for download, but at disparate locations (in various formats)—and often it may not be obvious to a newcomer where to obtain these resources and which exact files to use.

Pyserini tackles this challenge by packaging together these evaluation resources and providing a unified interface for accessing them. Figure 4 shows an example of loading topics via get_topics (L3) and loading qrels via get_qrels (L4) for the standard 6980-query subset of the development set of the MS MARCO passage ranking test collection. We have taken care to name the text descriptors consistently, so the associations between topics and relevance judgments are unambiguous.

Using Pyserini's provided functions, the topics and qrels are loaded into simple Python data structures and thus easy to manipulate. A standard TREC topic has different fields (e.g., title, description, narrative), which we model as a Python dictionary. Similarly,

qrels are nested dictionaries: query ids mapping to a dictionary of docids to (possibly graded) relevance judgments. Our choice to use Python data structures means that they can be manipulated using standard constructs such as list comprehensions. For example, we can straightforwardly compute the avg. length of queries (L7) and the avg. number of relevance judgments per query (L10).

3.3 Batch Retrieval

Putting everything discussed above together, it is easy in Pyserini to perform an end-to-end batch retrieval run with queries from a standard test collection. For example, the following command generates a run on the development queries of the MS MARCO passage ranking task (with BM25):

```
$ python -m pyserini.search --topics msmarco-passage-dev-subset \
    --index msmarco-passage --output run.msmarco-passage.txt \
    --bm25 --output-format msmarco
```

Following completion, we can evaluate the effectiveness of the run with another simple command:

```
$ python -m pyserini.eval.msmarco_passage_eval \
    msmarco-passage-dev-subset run.msmarco-passage.txt
```

The expected MRR@10 score is 0.1874 over 6980 queries. Pyserini includes a copy of the official evaluation script and provides a lightweight convenience wrapper around it. The toolkit manages qrels internally, so the user simply needs to provide the name of the test collection, without having to worry about downloading, storing, and specifying external files. Otherwise, the usage of the evaluation module is exactly the same as the official evaluation script; in fact, Pyserini simply dispatches to the underlying script after it translates the qrels mapping internally.

The above result corresponds to an Anserini baseline on the MS MARCO passage leaderboard. This is worth emphasizing and illustrates our goal of making Pyserini easy to use: with one simple command, it is possible to reproduce a run that serves as a common baseline on a popular leaderboard, providing a springboard to experimenting with different ranking models in a multi-stage architecture. Similar commands allow anyone to reproduce baseline results using batch retrieval with dense representations as well as hybrid retrieval.

3.4 Working with Custom Collections

Beyond existing corpora and test collections, a common use case for Pyserini is users who wish to search their own collections. For bag-of-words sparse retrieval, we have built in Anserini (written in Java) custom parsers and ingestion pipelines for common document formats used in IR research, for example, the TREC SGML format used in many newswire collections and the WARC format for web collections. However, exposing the right interfaces and hooks to support custom implementations in Python is awkward. Instead, we have implemented support for a generic and flexible JSON-formatted collection in Anserini, and Pyserini's indexer directly accesses the underlying capabilities in Anserini. Thus, searching custom collections in Pyserini necessitates first writing a simple script to reformat existing documents into our JSON specification, and then invoking the indexer. For dense retrieval, support for custom collections is less mature at present, but we provide utility

```
1   from pyserini.index import IndexReader
2
3   # Initialize from a pre-built index:
4   reader = IndexReader.from_prebuilt_index('robust04')
5
6   # Iterate over index terms and fetch term statistics:
7   import itertools
8   for term in itertools.islice(reader.terms(), 10):
9       print(f'{term.term} (df={term.df}, cf={term.cf})')
10
11  # Analyze a term:
12  term = 'atomic'
13  analyzed = reader.analyze(term)
14  print(f'The analyzed form of "{term}" is "{analyzed[0]}"')
15
16  # Directly fetch term statistics for a term:
17  df, cf = reader.get_term_counts(term)
18  print(f'term "{term}": df={df}, cf={cf}')
19
20  # Traverse postings for a term:
21  postings_list = reader.get_postings_list(term)
22  for p in postings_list:
23      print(f'docid={p.docid}, tf={p.tf}, pos={p.positions}')
24
25  # Examples of manipulating document vectors:
26  tf = reader.get_document_vector('LA071090-0047')
27  tp = reader.get_term_positions('LA071090-0047')
28  df = {
29      term: (reader.get_term_counts(term, analyzer=None))[0]
30      for term in tf.keys()
31  }
32  bm25_vector = {
33      term: reader.compute_bm25_term_weight('LA071090-0047',
34                                              term,
35                                              analyzer=None)
36      for term in tf.keys()
37  }
```

Figure 5: Examples of using Pyserini to access system internals such as term statistics and postings lists.

scripts that take an encoder model to convert documents into dense representations, and then build indexes that support querying.

The design of Pyserini makes it easy to use as a standalone module or to integrate as a library in another application. In the first use case, a researcher can reproduce a baseline (first-stage retrieval) run with a simple invocation, take the output run file (which is just plain text) to serve as input for downstream reranking, or as part of ensembles [7, 8]. As an alternative, Pyserini can be used as a library that is tightly integrated into another system.

3.5 Access to System Internals

Beyond simplifying the research lifecycle of working with standard IR test collections, Pyserini provides access to system internals to support use cases that we might not have anticipated. A number of these features for sparse retrieval are illustrated in Figure 5 and available via the IndexReader object, which can be initialized with pre-built indexes in the same way as the searcher classes. For these examples, we use the Robust04 index because access to many of the features requires positional indexes and storing document vectors. Due to size considerations, this information is not included in the pre-built MS MARCO indexes.

In (L7−9), we illustrate how to iterate over all terms in a corpus (i.e., its dictionary) and access each term's document frequency and

collection frequency. Here, we use standard Python tools to select and print out the first 10 terms alphabetically. In the next example, (L12–14), we show how to "analyze" a word (what Lucene calls tokenization, stemming, etc.). For example, the analyzed form of "atomic" is "atom". Since terms in the dictionary (and document vectors, see below) are stored in analyzed form, these methods are necessary to access system internals. Another way to access collection statistics is shown in (L17–18) by direct lookup.

Pyserini also provides raw access to index structures, both the inverted index as well as the forward index (i.e., to fetch document vectors). In (L21–23), we show an example of looking up a term's postings list and traversing its postings, printing out term frequency and term position occurrences. Access to the forward index is shown in (L26–27) based on a docid: In the first case, Pyserini returns a dictionary mapping from terms in the document to their term frequencies. In the second case, Pyserini returns a dictionary mapping from terms to their term positions in the document. From these methods, we can, for example, look up document frequencies for all terms in a document using a list comprehension in Python (L28–31). This might be further manipulated to compute tf–idf scores. Finally, the toolkit provides a convenience method for computing BM25 term weights, using which we can reconstruct the BM25-weighted document vector (L32–37).

At present, access to system internals focuses on manipulating sparse representations. Dense retrieval capabilities in Pyserini are less mature. It is not entirely clear what advanced features would be desired by researchers, but we anticipate adding support as the needs and use cases become clearer.

4 EXPERIMENTAL RESULTS

Having provided a "tour" of Pyserini and some of the toolkit's features, in this section we present experimental results to quantify its effectiveness for first-stage retrieval. Currently, Pyserini provides support for approximately three dozen test collections; here, we focus on two popular leaderboards.

Pyserini provides baselines for two MS MARCO datasets [6]: the passage ranking task (Table 1) and the document ranking task (Table 2). In both cases, we report the official metric (MRR@10 for passage, MRR@100 for document). For the development set, we additionally report recall at rank 1000, which is useful in establishing an upper bound on reranking effectiveness. Note that evaluation results on the test sets are only available via submissions to the leaderboard, and therefore we do not have access to recall figures. Furthermore, since the organizers discourage submissions that are "too similar" (e.g., minor differences in parameter settings) and actively limit the number of submissions to the leaderboard, we follow their guidance and hence do not have test results for all of our experimental conditions.

For the passage ranking task, Pyserini supports sparse retrieval, dense retrieval, as well as hybrid dense–sparse retrieval; all results in rows (1) through (3) are reproducible with our toolkit. Row (1a) reports the effectiveness of sparse bag-of-words ranking using BM25 with default parameter settings on the original text; row (1b) shows results after tuning the parameters on a subset of the dev queries via simple grid search to maximize recall at rank 1000. Parameter tuning makes a small difference in this case. Pyserini

| Method | MS MARCO Passage | | |
| | Development | | Test |
	MRR@10	R@1k	MRR@10
Pyserini: sparse			
(1a) Original text BM25, default ($k_1 = 0.9, b = 0.4$)	0.184	0.853	0.186
(1b) Original text BM25, tuned ($k_1 = 0.82, b = 0.68$)	0.187	0.857	0.190
(1c) doc2query–T5 BM25, default ($k_1 = 0.9, b = 0.4$)	0.272	0.947	0.277
(1d) doc2query–T5 BM25, tuned ($k_1 = 2.18, b = 0.86$)	0.282	0.951	-
Pyserini: dense			
(2a) TCT-ColBERT (brute-force)	0.335	0.964	-
(2b) TCT-ColBERT (HNSW)	0.335	0.962	-
Pyserini: dense–sparse hybrid			
(3a) TCT-ColBERT + original text	0.353	0.970	-
(3a) TCT-ColBERT + doc2query–T5	0.365	0.975	-
(4a) BM25 (Microsoft Baseline)	0.167	-	0.165
(4b) ANCE [37]	0.330	0.959	-
(4c) DistilBERT$_{DOT}$ [11]	0.323	0.957	-
Pyserini: multi-stage pipelines			
(4d) monoBERT [27]	0.372	-	0.365
(4e) Expando-Mono-DuoT5 [32]	0.420	-	0.408

Table 1: Results on the MS MARCO passage ranking task.

also provides document expansion baselines using our doc2query method [28]; the latest model uses T5 [33] as described in Nogueira and Lin [26]. Bag-of-words BM25 ranking over the corpus with document expansion is shown in rows (1c) and (1d) for default and tuned parameters. We see that doc2query yields a large jump in effectiveness, while still using bag-of-words retrieval, since neural inference is applied to generate expansions prior to indexing. With doc2query, parameter tuning also makes a difference.

For dense retrieval, results using TCT-ColBERT [17] are shown in rows (2) using different indexes. Row (2a) refers to brute-force scans over the document vectors in Faiss [12], which provides exact nearest-neighbor search. Row (2b) refers to approximate nearest-neighbor search using HNSW [23]; the latter yields a small loss in effectiveness, but enables interactive querying. We see that retrieval using dense learned representations is much more effective than retrieval using sparse bag-of-words representations, even taking into account document expansion techniques.

Results of hybrid techniques that combine sparse and dense retrieval using weighted interpolation are shown next in Table 1. Row (3a) shows the results of combining TCT-ColBERT with BM25 bag-of-words search over the original texts, while row (3b) shows results that combine document expansion using doc2query with the T5 model. In both cases we used a brute-force approach. Results show that combining sparse and dense signals is more effective than either alone, and that the hybrid technique continues to benefit from document expansion.

To put these results in context, rows (4) provide a few additional points of comparison. Row (4a) shows the BM25 baseline provided by the MS MARCO leaderboard organizers, which appears to be less effective than Pyserini's implementation. Rows (4b) and (4c) refer to two alternative dense-retrieval techniques; these results show that our TCT-ColBERT model performs on par with competing models. Finally, rows (4d) and (4e) show results from two

| Method | MS MARCO Document | | |
| | Development | | Test |
	MRR@100	R@1k	MRR@100
Pyserini: sparse			
(1a) Original text (doc)	0.230	0.886	0.201
BM25, default ($k_1 = 0.9, b = 0.4$)			
(1b) Original text (doc)	0.277	0.936	-
BM25, tuned ($k_1 = 4.46, b = 0.82$)			
(1c) Original text (passage)	0.268	0.918	-
BM25, default ($k_1 = 0.9, b = 0.4$)			
(1d) Original text (passage)	0.275	0.931	0.246
BM25, tuned ($k_1 = 2.16, b = 0.61$)			
(1e) doc2query–T5 (doc)	0.327	0.955	0.291
BM25, tuned ($k_1 = 4.68, b = 0.87$)			
(1f) doc2query–T5 (passage)	0.321	0.953	0.290
BM25, tuned ($k_1 = 2.56, b = 0.59$)			
Pyserini: dense			
(2) TCT-ColBERT	0.332	-	-
Pyserini: dense–sparse hybrid			
(3a) TCT-ColBERT + original text	0.370	-	-
(3b) TCT-ColBERT + doc2query–T5	0.378	-	-
(4a) BM25 (Microsoft Baseline)	-	-	0.192
(4b) ANCE [37]	0.384	-	0.342
Pyserini: multi-stage pipelines			
(4c) Expando-Mono-DuoT5 [32]	0.426	-	0.370

Table 2: Results on the MARCO document ranking task.

of our own reranking pipelines built using Pyserini for first-stage retrieval: monoBERT, a standard BERT-based reranker [27], and our "Expando-Mono-Duo" design pattern with T5 [32]. These illustrate how Pyserini can serve as the foundation for further explorations in neural ranking techniques.

Results on the MS MARCO document ranking task are shown in Table 2. For this task, there are two common configurations, what we call "per-document" vs. "per-passage" indexing. In the former, each document in the corpus is indexed separately (as is standard). In the latter, each document is first segmented into multiple passages, and each passage is indexed as a separate "document". For the "per-passage" index, a document ranking is constructed by simply taking the maximum of per-passage scores; the motivation for this design is to reduce the amount of text that computationally expensive downstream rerankers need to process. Rows (1a)–(1d) show the per-document and per-passage approaches on the original texts, using default parameters and after tuning for recall@100 using grid search. With default parameters, there appears to be a large effectiveness gap between the per-document and per-passage approaches, but with properly tuned parameters, (1b) vs. (1d), we see that they achieve comparable effectiveness. As with passage retrieval, we can include document expansion with either the per-document or per-passage approaches (the difference is whether we append the expansions to each document or each passage); these results are shown in (1e) and (1f). Similarly, the differences in effectiveness between the two approaches are quite small.

Dense retrieval using TCT-ColBERT is shown in row (2); this is a new experimental condition that was not reported in Lin et al. [17]. Here, we are simply using the encoder that has been trained on the MS MARCO passage data in a zero-shot manner. Since these encoders were not designed to process long segments of text, only the per-passage condition makes sense. In row (3a), we combine

row (2) with the per-passage sparse retrieval results on the original text, and in row (3b), with the per-passage sparse retrieval results using document expansion. Overall, the findings are consistent with the passage ranking task: Dense retrieval is more effective than sparse retrieval (although the improvements for document ranking are smaller, likely due to zero-shot application). Dense and sparse signals are complementary, shown by the effectiveness of the dense–sparse hybrid, which further benefits from document expansion (although the gains appear to be smaller).

Similar to passage ranking, Table 2 provides a few additional points of comparison. Row (4a) shows the effectiveness of the BM25 baseline provided by the leaderboard organizers (which we see are worse than Pyserini). Row (4b) shows results from ANCE [37], which appears to be more effective than TCT-ColBERT, although the comparison isn't quite fair since our models were not trained on MS MARCO document data. Finally, Row (4c) shows the results of applying our "Expando-Mono-Duo" design pattern with T5 [32] in a zero-shot manner.

In summary, Pyserini "covers all the bases" in terms of providing first-stage retrieval for modern research on neural ranking approaches: sparse retrieval, dense retrieval, as well as hybrid techniques combining both approaches. Experimental results on two popular leaderboards show that our toolkit provides a firm foundation for building multi-stage ranking architectures for a range of retrieval tasks, for example, searching the COVID-19 literature [43] and podcasts [42]. Beyond search, Pyserini has also been used for other applications such as question answering [43], fact verification [30], and combating misinformation [31].

5 CONCLUSIONS

Due to page limits, we do not have sufficient space to discuss two important aspects of Pyserini: (1) reproducibility, both the social and technical aspects thereof in the context of our project, and (2) future developments, including discussions of what our toolkit explicitly *isn't* going to do. For these details, we refer readers to an extended version of this paper on arXiv.

Our group's efforts to promote and support reproducible research dates back to at least 2015 [4, 15], and the landscape has changed quite a bit since then. Today, there is much more awareness of reproducibility issues; norms such as the sharing of source code have become more entrenched than before, and we have access to better tools now (e.g., Docker, package mangers, etc.) than we did before. At the same time, however, today's software ecosystem has become much more complex; ranking models have become more sophisticated and modern multi-stage ranking architectures involve more complex components than before. In this changing environment, the need for stable foundations on which to build remains. With Pyserini, it has been and will remain our goal to provide effective, easy-to-use tools in support of reproducible research in information retrieval and beyond.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 265–283.

[2] Zeynep Akkalyoncu Yilmaz, Charles L. A. Clarke, and Jimmy Lin. 2020. A Lightweight Environment for Learning Experimental IR Research Practices. In *Proceedings of the 43rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2020)*. 2113–2116.

[3] Zeynep Akkalyoncu Yilmaz, Shengjin Wang, Wei Yang, Haotian Zhang, and Jimmy Lin. 2019. Applying BERT to Document Retrieval with Birch. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations*. Hong Kong, China, 19–24.

[4] Jaime Arguello, Matt Crane, Fernando Diaz, Jimmy Lin, and Andrew Trotman. 2015. Report on the SIGIR 2015 Workshop on Reproducibility, Inexplicability, and Generalizability of Results (RIGOR). *SIGIR Forum* 49, 2 (2015), 107–116.

[5] Nima Asadi and Jimmy Lin. 2013. Effectiveness/Efficiency Tradeoffs for Candidate Generation in Multi-Stage Retrieval Architectures. In *Proceedings of the 36th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2013)*. Dublin, Ireland, 997–1000.

[6] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2018. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. *arXiv:1611.09268v3* (2018).

[7] Michael Bendersky, Honglei Zhuang, Ji Ma, Shuguang Han, Keith Hall, and Ryan McDonald. 2020. RRF102: Meeting the TREC-COVID Challenge with a 100+ Runs Ensemble. *arXiv:2010.00200* (2020).

[8] Andre Esteva, Anuprit Kale, Romain Paulus, Kazuma Hashimoto, Wenpeng Yin, Dragomir Radev, and Richard Socher. 2020. CO-Search: COVID-19 Information Retrieval with Semantic Search, Question Answering, and Abstractive Summarization. *arXiv:2006.09595* (2020).

[9] Adrien Grand, Robert Muir, Jim Ferenczi, and Jimmy Lin. 2020. From MaxScore to Block-Max WAND: The Story of How Lucene Significantly Improved Query Evaluation Performance. In *Proceedings of the 42nd European Conference on Information Retrieval, Part II (ECIR 2020)*. 20–27.

[10] Sebastian Hofstätter, Sophia Althammer, Michael Schröder, Mete Sertkan, and Allan Hanbury. 2020. Improving Efficient Neural Ranking Models with Cross-Architecture Knowledge Distillation. *arXiv:2010.02666* (2020).

[11] Sebastian Hofstätter, Sophia Althammer, Michael Schröder, Mete Sertkan, and Allan Hanbury. 2021. Improving Efficient Neural Ranking Models with Cross-Architecture Knowledge Distillation. *arXiv:2010.02666* (2021).

[12] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv:1702.08734* (2017).

[13] 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 the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6769–6781.

[14] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2020)*. 39–48.

[15] Jimmy Lin, Matt Crane, Andrew Trotman, Jamie Callan, Ishan Chattopadhyaya, John Foley, Grant Ingersoll, Craig Macdonald, and Sebastiano Vigna. 2016. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *Proceedings of the 38th European Conference on Information Retrieval (ECIR 2016)*. Padua, Italy, 408–420.

[16] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2020. Pretrained Transformers for Text Ranking: BERT and Beyond. *arXiv:2010.06467* (2020).

[17] Sheng-Chieh Lin, Jheng-Hong Yang, and Jimmy Lin. 2020. Distilling Dense Representations for Ranking using Tightly-Coupled Teachers. *arXiv:2010.11386* (2020).

[18] Xueguang Ma, Kai Sun, Ronak Pradeep, and Jimmy Lin. 2021. A Replication Study of Dense Passage Retriever. *arXiv:2104.05740* (2021).

[19] Sean MacAvaney. 2020. OpenNIR: A Complete Neural Ad-Hoc Ranking Pipeline. In *Proceedings of the 13th ACM International Conference on Web Search and Data Mining (WSDM 2020)*. Houston, Texas, 845–848.

[20] Sean MacAvaney, Andrew Yates, Sergey Feldman, Doug Downey, Arman Cohan, and Nazli Goharian. 2021. Simplified Data Wrangling with ir_datasets. *arXiv:2103.02280* (2021).

[21] Craig Macdonald, Richard McCreadie, Rodrygo L.T. Santos, and Iadh Ounis. 2012. From Puppy to Maturity: Experiences in Developing Terrier. In *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*. Portland, Oregon.

[22] Craig Macdonald and Nicola Tonellotto. 2020. Declarative Experimentation in Information Retrieval using PyTerrier. In *Proceedings of the 2020 International Conference on the Theory of Information Retrieval (ICTIR 2020)*. 161–168.

[23] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.

[24] Antonio Mallia, Michał Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia. In *Proceedings of the Open-Source IR Replicability Challenge (OSIRRC 2019): CEUR Workshop Proceedings Vol-2409*. Paris, France, 50–56.

[25] Irina Matveeva, Chris Burges, Timo Burkard, Andy Laucius, and Leon Wong. 2006. High Accuracy Retrieval with Multiple Nested Ranker. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*. Seattle, Washington, 437–444.

[26] Rodrigo Nogueira and Jimmy Lin. 2019. From doc2query to docTTTTTquery.

[27] Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy Lin. 2019. Multi-Stage Document Ranking with BERT. *arXiv:1910.14424* (2019).

[28] Rodrigo Nogueira, Wei Yang, Jimmy Lin, and Kyunghyun Cho. 2019. Document Expansion by Query Prediction. *arXiv:1904.08375* (2019).

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach 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 *Advances in Neural Information Processing Systems*. 8024–8035.

[30] Ronak Pradeep, Xueguang Ma, Rodrigo Nogueira, and Jimmy Lin. 2021. Scientific Claim Verification with VerT5erini. In *Proceedings of the 12th International Workshop on Health Text Mining and Information Analysis*. 94–103.

[31] Ronak Pradeep, Xueguang Ma, Rodrigo Nogueira, and Jimmy Lin. 2021. Vera: Prediction Techniques for Reducing Harmful Misinformation in Consumer Health Search. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*.

[32] Ronak Pradeep, Rodrigo Nogueira, and Jimmy Lin. 2021. The Expando-Mono-Duo Design Pattern for Text Ranking with Pretrained Sequence-to-Sequence Models. *arXiv:2101.05667* (2021).

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[34] Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A Cascade Ranking Model for Efficient Ranked Retrieval. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*. Beijing, China, 105–114.

[35] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 38–45.

[36] Chenyan Xiong, Zhenghao Liu, Si Sun, Zhuyun Dai, Kaitao Zhang, Shi Yu, Zhiyuan Liu, Hoifung Poon, Jianfeng Gao, and Paul Bennett. 2020. CMT in TREC-COVID Round 2: Mitigating the Generalization Gaps from Web to Special Domain Search. *arXiv:2011.01580* (2020).

[37] 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).

[38] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proceedings of the 40th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2017)*. Tokyo, Japan, 1253–1256.

[39] Peilin Yang, Hui Fang, and Jimmy Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. *Journal of Data and Information Quality* 10, 4 (2018), Article 16.

[40] Andrew Yates, Siddhant Arora, Xinyu Zhang, Wei Yang, Kevin Martin Jose, and Jimmy Lin. 2020. Capreolus: A Toolkit for End-to-End Neural Ad Hoc Retrieval. In *Proceedings of the 13th ACM International Conference on Web Search and Data Mining (WSDM 2020)*. Houston, Texas, 861–864.

[41] Andrew Yates, Kevin Martin Jose, Xinyu Zhang, and Jimmy Lin. 2020. Flexible IR Pipelines with Capreolus. In *Proceedings of the 29th International Conference on Information and Knowledge Management (CIKM 2020)*. 3181–3188.

[42] Yongze Yu, Jussi Karlgren, Hamed Bonab, Ann Clifton, Md Iftekhar Tanveer, and Rosie Jones. 2020. Spotify at the TREC 2020 Podcasts Track: Segment Retrieval. In *Proceedings of the Twenty-Ninth Text REtrieval Conference (TREC 2020)*.

[43] Edwin Zhang, Nikhil Gupta, Raphael Tang, Xiao Han, Ronak Pradeep, Kuang Lu, Yue Zhang, Rodrigo Nogueira, Kyunghyun Cho, Hui Fang, and Jimmy Lin. 2020. Covidex: Neural Ranking Models and Keyword Search Infrastructure for the COVID-19 Open Research Dataset. In *Proceedings of the First Workshop on Scholarly Document Processing*. 31–41.