XpmIR: A Modular Library for Learning to Rank and Neural IR Experiments

Yuxuan Zong
yuxuan.zong@isir.upmc.fr
Sorbonne Université, CNRS, ISIR, F-75005 Paris, France
Paris, France

Benjamin Piwowarski
benjamin@piwowarski.fr
CNRS, Sorbonne Université, ISIR, F-75005 Paris, France
Paris, France

ABSTRACT

During past years, several frameworks for (Neural) Information Retrieval have been proposed. However, while they allow reproducing already published results, it is still very hard to re-use some parts of the learning pipelines, such as for instance the pre-training, sampling strategy, or a loss in newly developed models. It is also difficult to use new training techniques with old models, which makes it more difficult to assess the usefulness of ideas on various neural IR models. This slows the adoption of new techniques, and in turn, the development of the IR field. In this paper, we present XpmIR, a Python library defining a reusable set of experimental components. The library already contains state-of-the-art models and indexing techniques and is integrated with the HuggingFace hub.

CCS CONCEPTS

• **Information systems → Information retrieval**; *Learning to rank*; • **General and reference → Experimentation**.

KEYWORDS

neural information retrieval, learning to rank, experimental framework

ACM Reference Format:
Yuxuan Zong and Benjamin Piwowarski. 2023. XpmIR: A Modular Library for Learning to Rank and Neural IR Experiments. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23), July 23–27, 2023, Taipei, Taiwan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Developing (Neural) Information Retrieval models depends on a complex pipeline (e.g. pre-processing, pre-training, training, indexation, and evaluation). A series of (Python) libraries have been proposed to allow easy reproduction of experimental results. While those libraries are successful when trying to re-run one specific setting – or variations thereof, as it is possible to reproduce experimental results and manipulate some experimental parameters through a command line interface and/or parameter files, they are less useful when it comes to new models.

More precisely, modifying the pipeline is not easy since these projects are not modular enough. We think that this mostly explains why new models are not developed within those libraries – thereby preventing the adoption of some *components* of these new models into new ones. Designing new models that leverage recent innovations is time-consuming, and this in turn slows down research and code sharing.

The aim of the XpmIR library is to propose a framework of *reusable components* that allow designing new experiments for neural information retrieval (neural IR), as well as reproducing old ones. XpmIR achieves this goal by: (1) Adopting abstractions to describe the various data sources through dataset adapters and samplers; (2) Defining a standardized way to learn, using hooks to modify the learning behavior; (3) Providing a set of reusable components (samplers, neural models, text representation, evaluation toolchain); (4) Providing pre-trained models on HuggingFace.[1]

The library is open source (GPLv3 licence) and available on GitHub[2]. XpmIR is built upon experimaestro and datamaestro [23], which are generic frameworks (i.e. not specific to IR) that allows to (1) standardize access to datasets; (2) define modular experimental components (configuration and tasks); (3) express full experimental plans with automatic folder tracking (each folder corresponding to a unique set of experimental settings); (4) monitor the submitted tasks (through a scheduler).

2 RELATED WORKS

Firstly, there exist many libraries related to IR or implementing search indices for standard IR models. Among those, we can cite:

Datasets ir-datasets [15] propose an API to access many IR datasets and download them (if freely available);

Indexing and retrieval There are many libraries that deal with indexing. For standard IR models, we can cite Pyserini [29], PyTerrier [16] and [18], and for dense neural IR models, FAISS [9].

Evaluation ir-measures [15] is a recent library that provides direct access to a very diverse set of IR metrics.

When possible XpmIR re-uses existing libraries by providing components encapsulating the access to these libraries, so they can be reused in different experimental pipelines. More precisely, as of today, XpmIR provides components for ir-datasets, ir-measures, Pyserini, and FAISS. A helper library has been developed to tackle the

SIGIR '23, July 23–27, 2023, Taipei, Taiwan.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9408-6/23/07…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]The current list of pre-trained models is available at https://huggingface.co/models?library=xpmir

[2]The documentation can be found at https://experimaestro-ir.readthedocs.io/ and the source code at https://github.com/bpiwowar/experimaestro-ir

case of sparse neural IR models that do not rely on term frequency-based indices. More related to XpmIR, various libraries for neural IR have been proposed during past years, such as OpenNIR[3] [13], MatchMaker[4] [6], Capreolus[5] [30] – or more specific code repositories related to one set of models – e.g. for the ColBERT[6] [10], SPLADE models[7] [4] and dense models (Tevatron[8][5]).

For OpenNIR and MatchMaker, the experiments are all based on configuration files (or command line parameters) which allow changing some aspects of the training process but do not allow combining parts of the pipeline easily. Capreolus, which is the most related to XpmIR defines a set of components (*modules*), but relies on pre-defined tasks (e.g. train and evaluate) that define placeholders (e.g. which re-ranker to use) which might not fit all the needs. For instance, generating strong negatives during the learning process to define new training triplets, as needed in ANCE [28], is not doable without fully rewriting the learning script in Capreolus, MatchMaker or OpenNIR.

We argue that these libraries are not *modular* enough. For instance, what if we want to use a specific MLM pre-training step such as LexMAE [25] with ColBERT [10]? What if we want to use one model to generate hard negatives? In these libraries, this requires coding explicitly all the "glue" between different parts, but this is time-consuming and error-prone. In comparison, the XpmIR library has the objective of (i) proposing a set of reusable components to train and evaluate neural IR models; (ii) allowing to combine these components and design complex experiments thanks to experimaestro and datamaestro [23]; (iii) providing a set of high-level experiment components (e.g. those used in a typical MS-Marco-based re-ranking task) similar to what is done in Capreolus; (iv) providing integration with HuggingFace to re-use pre-trained IR models (as Tevatron).

3 XPMIR

This section describes the various XpmIR library components[9] grouped by category:

datasets components provide access to the ir-datasets library (through lightweight wrappers), as well as other IR-specific datasets (e.g. triplets to train neural models) and adapters that can process and transform datasets;

retrievers and scorers components define how to represent a text, define IR and neural IR models;

learning to rank components allow to build up a training pipeline

evaluation components allow the evaluation of trained models.

We illustrate some components by (modified) code excerpts corresponding to paper reproductions of two state-of-the-art models implemented within the library, namely MonoBERT [20] and SPLADE [4]. MonoBERT [20] is the well-established cross-encoder model for the Neural IR that uses a two-stage ranking. SPLADE [4] is a first-stage ranking model which benefits from a sparse representation of documents and queries. Full code can be found in the documentation[10] that illustrate somehow complex experimental plans composing pre/post-processing steps with learning to rank components, as well as evaluating the trained model and comparing it to baselines – illustrating the fact that XpmIR could be used for ensuring reproducibility of a research paper by providing the *full* code of the complete experiments (including baselines and variations).

3.1 Datasets

XpmIR is integrated with the text repository of datamaestro [23] that provides a standard way to access datasets of different types. Through datamaestro, XpmIR provides an interface to ir-datasets [15], allowing it to access easily most IR datasets. XpmIR also provides access to distillation samples such as those from [8]. Finally, each data type – documents, topics, assessments, learning triplets, runs – is associated with a Python interface allowing to access the data.

XpmIR also provides dataset adapters[11] needed to transform a dataset. For instance, RandomFold can be useful to sample topics from an Adhoc IR dataset. The following code shows how to sample 500 topics from the MS-Marco development dataset – leaving out topics in the "small development" dataset commonly used when reporting results[12]:

```
small = prepare_dataset("irds.msmarco-passage.dev.small")
dev = prepare_dataset("irds.msmarco-passage.dev")
ds_val = RandomFold(
    dataset=dev, fold=0, sizes=[500], exclude=small.topics
).submit()
```

Another useful dataset transformation is the creation of a retriever-based collection composed of all the documents retrieved by a given retriever – which can be used when computing validation metrics for first-stage rankers like SPLADE.

3.2 Retrievers and Neural IR models

3.2.1 Text Representation and Neural Models. Many models rely on some form of text representation. In XpmIR, we distinguish three types of text representations: (1) a `Tokenizer` that returns a list of token IDs; (2) a `TokensEncoder` that returns one embedding per token; (3) encoders that encode into a vector either a text (a `TextEncoder`, that can be used to transform a document collection into a FAISS index), pair of texts (`DualTextEncoder`, e.g. for a query/document pair), or a triplet (`TripletTextEncoder`, e.g. for query/document/document triplet). These encoders are the basis of dense models, cross-encoders [20], and dual cross-encoders like duoBERT [21]. This text representation is handled by two concrete sets of classes, those that correspond to word embeddings like GloVe [22] or word2vec [19], and can be used to define pre-BERT models, and those that leverage HuggingFace transformers [27].

[3]https://github.com/Georgetown-IR-Lab/OpenNIR
[4]https://github.com/sebastian-hofstaetter/matchmaker
[5]https://github.com/capreolus-ir/capreolus
[6]https://github.com/stanford-futuredata/ColBERT
[7]https://github.com/naver/splade
[8]https://github.com/texttron/tevatron
[9]These components either correspond to configurations or tasks in experimaestro: configurations describe experimental settings, while tasks correspond to executable code – as for instance, when indexing a collection, learning or evaluating a model

[10]https://experimaestro-ir.readthedocs.io/en/latest/papers/monobert.html for monoBERT and https://experimaestro-ir.readthedocs.io/en/latest/papers/splade.html for SPLADE
[11]https://experimaestro-ir.readthedocs.io/en/latest/data/adapters.html
[12]the `submit` sends the task to the experimaestro scheduler

3.2.2 Retrievers and Scorers. `Scorers` are responsible for computing a score for a given query and document – which means that all neural models are scorers. The neural models are organized within a hierarchy that allows factorizing as many common properties as possible. For instance, a `DualRepresentationScorer` is a neural model that represents both documents and queries separately before computing a similarity – dense models are part of this family, as well as late interaction models. Another example is a `DualVectorScorer` that relies on two vectorial representations (queries and documents), and provide ways to speed up learning when using batch-wise negatives [24]. In practice, scorers are often defined as a composition of a scoring function and a text representation. For instance, the monoBERT model can be defined as the composition of a `CrossScorer`, a classifier of query/document representations provided by a `DualTransformerEncoder`, as the code below[13]:

```
monobert = CrossScorer(encoder=DualTransformerEncoder(
    model_id="bert-base-uncased", trainable=True))
```

3.2.3 Retrievers. Retrievers allow searching a document collection *efficiently*. The simplest models are standard bag-of-words models, such as BM25. For such models, a retriever can be created based on an index. At the moment, a Pyserini [12] adapter is provided, but others like PyTerrier [16] would be straightforward to implement. The following code shows how to define a retriever for the BM25 model based on Pyserini [12]:

```
index = IndexCollection(documents=documents).submit()
retr = AnseriniRetriever(index=index, k=50, model=BM25())
```

Other types of indices are also supported to allow fast retrieval with neural models. Dense indices are handled through FAISS integration [9]. Sparse neural models produce indices with a weight distribution different from standard IR models [17]; to cope with those, a helper library has been written in rust[14] and allows indexing by providing sparse vectors to as document representations. For retrieval, WAND [1] and MaxScore [26] algorithms are currently implemented. As an illustration, the code below shows how to define a retriever for the SPLADE model:

```
index = SparseRetrieverIndexBuilder(
    encoder=DenseDocumentEncoder(scorer=scorer),
    ↪  documents=documents,
).submit()
retr = SparseRetriever(index=index, topk=100,
    encoder=DenseQueryEncoder(scorer=scorer),
)
```

Finally, for "heavy" neural models that can only be used to rerank the results, the `TwoStageRetriever` class uses a retriever to select a subset of documents before using a `Scorer` to rerank them. The definition of such a retriever is easy:

```
retr = TwoStageRetriever(retriever=retriever,
↪  scorer=monobert)
```

3.3 Learning to Rank

The learning process is handled by different components which are described below.

Optimization: optimizers and schedulers. The optimization part defines how to perform a gradient step. It relies on the definition of a series of optimizers, each responsible for optimizing a part of the model parameters. The latter can be useful when some parameters need to be optimized differently, as when fine-tuning transformers (i.e. normalization layers and biases parameters should not be included in the L2 regularization loss). Each optimizer learning rate can be controlled by a scheduler – which is again commonly used when fine-tuning transformers [27]. The following code illustrates how to define the optimizer during the training of MonoBERT, where the first optimizer uses Adam [11] avoiding L2 regularization for biases or normalization layers (parameters whose name finish by `bias` or containing `LayerNorm`), while the second handles all the other parameters with the AdamW optimizer – this example illustrates the flexibility of the components exposed by XpmIR:

```
scheduler = LinearWithWarmup(num_warmup_steps=1024)
optimizers = [
    ParameterOptimizer(
        scheduler=scheduler, optimizer=Adam(),
        filter=RegexParameterFilter(includes=[r"\.bias$",
        ↪  r"\.LayerNorm\."]),
    ),
    ParameterOptimizer(
        scheduler=scheduler,
        ↪  optimizer=AdamW(weight_decay=1e-2),
    ),
]
```

Trainers and samplers. The trainer is responsible for performing a *learning step* over the training data. Different trainers exist depending on the type of samples they can handle (i.e. pointwise, pairwise, batch-wise). Some trainers are designed to handle distillation, which is key to obtaining state-of-the-art first-stage rankers [3, 7]. Samplers are in charge of providing data samples whose type depends on the sampler (e.g. pointwise or pairwise). Finally, hooks can be used to modify the training process, which might be necessary to compute regularization losses.

The code below shows how to build a trainer based on knowledge distillation (used to train SPLADE_DistilMSE [3]), where `sampler` is an iterator over tuples composed of a query, two documents, and their scores computed by monoBERT:

```
distil_pairwise_trainer = DistillationPairwiseTrainer(
    batch_size=64,
    sampler=sampler,
    lossfn=MSEDifferenceLoss(),
    hooks=[FlopsRegularizer()],
)
```

[13] This code also shows how components – `CrossScorer` and `DualTransformerEncoder` – can be composed in XpmIR
[14] https://github.com/experimaestro/experimaestro-ir-rust

Learner. Finally, the `Learner` is the central class that handles the overall learning process. It relies on: (i) a neural model to be trained (the `Scorer`); (ii) a trainer that specifies how to train the model (e.g. pointwise, pairwise) and with which data (using samplers); (iii) an optimizer that specifies how to perform gradient descent; (iv) one or more listeners that monitor the training process – the most important is the validation listener allowing keeping the best checkpoints for a set of validation metrics. Moreover, the learner can be modified with hooks allowing them to alter some parts of the learning process. An example of a hook is to distribute the models on multiple GPUs; another one is to modify the model by "freezing" some weights. The learner process is divided into epochs – where each epoch is defined by a number of learning steps (i.e. batches) performed by the trainer. Note that one epoch does not correspond to a full pass over the dataset (which is necessary since some collections might be huge, e.g. if sampling from a retriever). The following code shows how we define a learner for monoBERT and get the model that maximizes the RR@10 metric over the validation set:

```
learner = Learner(
    trainer=monobert_trainer, scorer=monobert_scorer,
    steps_per_epoch=100, max_epochs=1000,
    optimizers=optimizers,
    listeners={"best_val": validation},
    hooks=[DistributedHook(models=[monobert_scorer])]
)
trained = learner.submit()
best_rr10 = trained["bestval"]["RR@10"]
```

3.4 Evaluation

The evaluation of the model is the final step of IR experiments. To ease the evaluation, an `EvaluationsCollection` class holds the different datasets and metrics on which evaluation should be performed for each evaluated model. The actual metrics are computed thanks to ir-measure [14]. The code below shows how to evaluate monoBERT – the `retriever_factory` defines how to build a retriever from monoBERT (using the `TwoStageRetriever` with `best_rr10` as the scorer) depending on the document collection:

```
measures = [AP, P @ 20, nDCG, nDCG @ 10, nDCG @ 20, RR @ 10]
tests = EvaluationsCollection(
    trec2019=Evaluations(
      prepare_dataset("irds.msmarco-passage.trec-dl-2019"),
        measures
    ),
    msmarco_dev=Evaluations(devsmall, measures),
)
tests.evaluate_retriever(retriever_factory)
```

4 PAPER REPRODUCTION AND HUGGINGFACE INTEGRATION

A part of the XpmIR library is dedicated to reproducing (partially) some IR papers. A command line interface is provided to reproduce these papers – automating the upload to HuggingFace Hub, including monitored training metrics and results on evaluation.[15]

[15] An example with monoBERT is available on https://huggingface.co/xpmir/monobert.

Table 1: Reproduction of monoBERT and SPLADE

	MS MARCO dev		TREC DL 2019	
	MRR@10	nDCG@10	MRR@10	nDCG@10
monoBERT XpmIR	0.364	0.426	0.937	0.705
monoBERT [20]	0.347	-	-	-
splade-max XpmIR	0.345	0.407	0.973	0.694
splade-max [2]	0.340	-	-	0.684
splade-doc XpmIR	0.321	0.404	0.934	0.667
splade-doc [2]	0.322	-	-	0.667
splade-DistilMSE XpmIR	0.356	0.421	0.961	0.730
splade-DistilMSE [3]	0.358	-	-	0.729

Two (partial) paper reproductions are actually implemented within the XpmIR library, namely MonoBERT [20] and SPLADE [3]. For the reproduction of MonoBERT, we use BERT-base as our pre-trained checkpoint. For SPLADE, we use the DistilBERT-base as the pre-trained checkpoint. In both cases, we use the MRR@10 and nDCG@10 metrics and evaluate the model on the MS-MARCO passage retrieval (development set) and TREC-DL-2019. Table 1 shows that the results obtained with XpmIR match those reported in the papers.

Besides defining components for defining, training, and evaluating neural IR models, XpmIR provides an integration with the HuggingFace Hub. This allows to upload and download pre-trained models.[16] These models can be used in other experiments and/or finetuned on specific datasets by leveraging XpmIR. The code below shows how to create a `Scorer` from the pre-trained tas-balanced dense model [8]:

```
tasb = AutoModel.load_from_hf_hub("xpmir/tas-balanced")
```

This scorer can then be re-used in another training or evaluation pipeline.

5 CONCLUSION

In this paper, we presented the XpmIR framework for neural IR training and evaluation. XpmIR is based on the idea to decompose dataset pre-processing, neural IR models, learning to rank, and evaluation into a set of re-usable components – which leverage existing IR libraries when possible. These components can be composed through the experimaestro library [23] to define complex neural IR experimental plans. XpmIR also provide high level components that ease the description of standard experiments (e.g. training a re-ranker on MS Marco). The interest of using such components is that it is much easier to re-use some of them for new models and to design experimental plans comparing various models. With our current code structure and the different components already presented, many other papers could be easily implemented. We are for example currently working on reproducing duoBERT [21] and ColBERT [10], and plan to include modern training procedures relying on IR-specific masked-LM pre-training (e.g. [25]) and self-distillation (e.g. used in [3]).

[16] At the time of writing, monoBERT [20], TAS-Balanced [8], as well as various versions of SPLADE [2] are available. The actual list of pre-trained models can be found on the HuggingFace Hub: https://huggingface.co/models?library=xpmir

REFERENCES

[1] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management (CIKM '03)*. Association for Computing Machinery, New York, NY, USA, 426–434. https://doi.org/10.1145/956863.956944

[2] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval. arXiv:2109.10086 (Sep 2021). http://arxiv.org/abs/2109.10086 arXiv:2109.10086 [cs].

[3] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2022. From Distillation to Hard Negative Sampling: Making Sparse Neural IR Models More Effective. arXiv:2205.04733 (May 2022). http://arxiv.org/abs/2205.04733 arXiv:2205.04733 [cs].

[4] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. 2021. *SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking*. Technical Report. http://arxiv.org/abs/2107.05720 arXiv: 2107.05720.

[5] Luyu Gao, Xueguang Ma, Jimmy J. Lin, and Jamie Callan. 2022. Tevatron: An Efficient and Flexible Toolkit for Dense Retrieval. *ArXiv* abs/2203.05765 (2022).

[6] Sebastian Hofstätter. 2019. Matchmaker. https://github.com/sebastian-hofstaetter/matchmaker

[7] 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 (Jan 2021). http://arxiv.org/abs/2010.02666 arXiv:2010.02666 [cs].

[8] Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. 2021. Efficiently Teaching an Effective Dense Retriever with Balanced Topic Aware Sampling. arXiv:2104.06967 (May 2021). http://arxiv.org/abs/2104.06967 arXiv:2104.06967 [cs].

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[10] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. *arXiv:2004.12832 [cs]* (April 2020). http://arxiv.org/abs/2004.12832 arXiv: 2004.12832.

[11] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[12] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python Toolkit for Reproducible Information Retrieval Research with Sparse and Dense Representations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, Canada) *(SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 2356–2362. https://doi.org/10.1145/3404835.3463238

[13] Sean MacAvaney. 2020. OpenNIR: A Complete Neural Ad-Hoc Ranking Pipeline. In *WSDM 2020*.

[14] Sean MacAvaney, Craig Macdonald, and Iadh Ounis. 2022. *Streamlining evaluation with ir-measures.* Lecture Notes in Computer Science, Vol. 13186. Springer International Publishing, Cham, 305–310. https://doi.org/10.1007/978-3-030-99739-7_38

[15] Sean MacAvaney, Andrew Yates, Sergey Feldman, Doug Downey, Arman Cohan, and Nazli Goharian. 2021. Simplified Data Wrangling with ir_datasets. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval.* Association for Computing Machinery, New York, NY, USA, 2429–2436. https://doi.org/10.1145/3404835.3463254

[16] Craig Macdonald and Nicola Tonellotto. 2020. Declarative Experimentation inInformation Retrieval using PyTerrier. In *Proceedings of ICTIR 2020.*

[17] Joel Mackenzie, Antonio Mallia, and Alistair Moffat. 2022. Accelerating Learned Sparse Indexes Via Term Impact Decomposition. In *Findings of the Association for Computational Linguistics: EMNLP 2022.*

[18] Antonio Mallia, Michal Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia. In *Proceedings of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, OSIRRC@SIGIR 2019, Paris, France, July 25, 2019.* 50–56. http://ceur-ws.org/Vol-2409/docker08.pdf

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality.. In *NIPS'14*, Vol. cs.CL. 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdfhttp://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality

[20] Rodrigo Nogueira and Kyunghyun Cho. 2020. Passage Re-ranking with BERT. https://doi.org/10.48550/arXiv.1901.04085 arXiv:1901.04085 [cs].

[21] Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy Lin. 2019. Multi-Stage Document Ranking with BERT. *arXiv:1910.14424 [cs]* (Oct. 2019). http://arxiv.org/abs/1910.14424 ZSCC: 0000001 arXiv: 1910.14424.

[22] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global Vectors for Word Representation.

[23] Benjamin Piwowarski. 2020. Experimaestro and Datamaestro: Experiment and Dataset Managers (for IR). In *ACM SIGIR 2020*. Xian, China. https://doi.org/10.1145/3397271.3401410 ZSCC: NoCitationData[s0].

[24] Yingqi Qu, Yuchen Ding, Jing Liu, Kai Liu, Ruiyang Ren, Wayne Xin Zhao, Daxiang Dong, Hua Wu, and Haifeng Wang. 2021. RocketQA: An Optimized Training Approach to Dense Passage Retrieval for Open-Domain Question Answering. In *In Proceedings of NAACL.*

[25] Tao Shen, Xiubo Geng, Chongyang Tao, Can Xu, Xiaolong Huang, Binxing Jiao, Linjun Yang, and Daxin Jiang. 2022. LexMAE: Lexicon-Bottlenecked Pretraining for Large-Scale Retrieval. In *ICLR*. arXiv. https://doi.org/10.48550/arXiv.2208.14754 arXiv:2208.14754 [cs].

[26] Howard Turtle and James Flood. 1995. Query evaluation: Strategies and optimizations. *Information Processing & Management* 31, 6 (Nov. 1995), 831–850. https://doi.org/10.1016/0306-4573(95)00020-H

[27] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi 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 M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. https://doi.org/10.48550/arXiv.1910.03771 arXiv:1910.03771 [cs].

[28] 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 (Oct 2020). http://arxiv.org/abs/2007.00808 arXiv:2007.00808 [cs].

[29] Peilin Yang, Hui Fang, and Jimmy Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. 10, 4 (2018). https://doi.org/10/ggmdws

[30] Andrew Yates, Siddhant Arora, Xinyu Zhang, Wei Yang, Kevin Martin Jose, and Jimmy Lin. [n. d.]. Capreolus: A Toolkit for End-to-End Neural Ad Hoc Retrieval *(WSDM '20)*. 861–864. https://doi.org/10/ggjnkm