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:

1 INTRODUCTION
Developing (Neural) Information Retrieval models depends on a complex pipeline (e.g. pre-processing, pre-training, training, indexing, 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.

The library is open source (GPLv3 licence) and available on GitHub. XpmIR is built upon experimaestro and datamaestro, which are generic frameworks (i.e. not specific to IR) that allow designing new experiments for neural information retrieval (neural IR), as well as reproducing old ones. 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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.nnnnnn

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/bpiwowarski/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\(^4\) [13], MatchMaker\(^5\) [6], Capreolus\(^5\) [30] – or more specific code repositories related to one set of models – e.g. for the ColBERT\(^7\) [10], SPLADE modules\(^7\) [4] and dense models (Tevatron\(^5\)[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 re-writing 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 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 transform and process 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\(^{20}\). 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\(^12\):

```python
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)
```

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 (DuoTextEncoder, 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].
3.2.2 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\(^3\):

```
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\(^1\) adapter is provided, but others like PyTerrier\(^2\) 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\(^4\) 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=retr, 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:

```
distill_pairwise_trainer = DistillationPairwiseTrainer(
    batch_size=64,
    sampler=sampler,
    lossfn=MSEDifferenceLoss(),
    hooks=[FlopsRegularizer()])
```
3.4 Evaluation

The evaluation of the model is the final step of IR experiments. To evaluate the model, the 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_rr@10 as the scorer) depending on the document collection:

```python
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"]
```

### 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

#### Table 1: Reproduction of monoBERT and SPLADE

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

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]:

```python
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 experimastor 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]).

---

14At 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

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