Lightning IR: Straightforward Fine-tuning and Inference of Transformer-based Language Models for Information Retrieval

Ferdinand Schlatt
Friedrich-Schiller-Universität Jena
Jena, Germany
ferdinand.schlatt@uni-jena.de

Maik Fröbe
Friedrich-Schiller-Universität Jena
Jena, Germany
maik.froebe@uni-jena.de

Matthias Hagen
Friedrich-Schiller-Universität Jena
Jena, Germany
matthias.hagen@uni-jena.de

Abstract

A wide range of transformer-based language models have been proposed for information retrieval tasks. However, including transformer-based models in retrieval pipelines is often complex and requires substantial engineering effort. In this paper, we introduce Lightning IR, an easy-to-use PyTorch Lightning-based framework for applying transformer-based language models in retrieval scenarios. Lightning IR provides a modular and extensible architecture that supports all stages of a retrieval pipeline: from fine-tuning and indexing to searching and re-ranking. Designed to be scalable and reproducible, Lightning IR is available as open-source: https://github.com/webis-de/lightning-ir.

CCS Concepts

• **Information systems → Retrieval models and ranking**.

Keywords

Software framework, Retrieval pipeline, Retrieval experiments

ACM Reference Format:
Ferdinand Schlatt, Maik Fröbe, and Matthias Hagen. 2025. Lightning IR: Straightforward Fine-tuning and Inference of Transformer-based Language Models for Information Retrieval. In *Proceedings of the Eighteenth ACM International Conference on Web Search and Data Mining (WSDM '25), March 10–14, 2025, Hannover, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3701551.3704118

1 Introduction

Pre-trained transformer-based language models have become a cornerstone in information retrieval (IR) research [15]. Many different architectures have been proposed, each with their own implementation and training procedure. This plethora makes fine-tuning and comparing different model architectures cumbersome. However, all these models use similar backbones, are fine-tuned in the same way, and have only minor differences in their inference procedure.

To unify the usage of transformer-based language models in IR, we present the *Lightning IR* framework. Lightning IR builds on and extends PyTorch Lightning [9] to provide several key features that set it apart from existing libraries for neural IR: (1) It is backbone agnostic, i.e., (almost) any HuggingFace [25] transformer-based language model can be used. (2) It supports the entire IR pipeline, from fine-tuning and indexing to searching and re-ranking. (3) It is flexible and supports, for example, multi-vector or sparse bi-encoders and pointwise or listwise cross-encoders. (4) It provides an easy to use API and CLI. (5) It is highly configurable, allowing for reproducible experiments and painless model comparison. In this paper, we compare Lightning IR to existing frameworks, describe its features and API, and demonstrate Lightning IR's capabilities.

Table 1: Comparison of different IR frameworks' supported stages (FT: fine-tuning, I: indexing, S: searching, RR: re-ranking) and model types (Bi-/Cr.-Enc.: bi-/cross-encoder, SV/MV: single-/multi-vector, DE: dense, SP: sparse, PW/LW: point-/listwise). (✓) denotes support for some model types.

Framework	**Stages**				**Model Types**					
	FT	I	S	RR	Bi-Enc.				Cr.-Enc.	
					SV	MV	DE	SP	PW	LW
baguetter [13]	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗
Capreolus [26]	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗
Experimaestro-IR [21]	✓	(✓)	(✓)	✓	✓	✓	✓	✓	✓	✗
FlexNeuART [2]	✓	✗	✓	✓	✓	✗	✓	✗	✓	✗
OpenNIR [16]	✓	✗	✗	✓	✓	✗	✓	✗	✓	✗
Patapasco [5]	✗	✓	✓	✓	✓	✗	✓	✗	✓	✗
Pyserini [14]	✗	✓	✓	✓	✓	✗	✓	✗	✓	✗
PyTerrier [18]	✗	(✓)	(✓)	✓	✓	✓	✓	✗	✓	✗
RAGatouille[1]	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗
rerankers [4]	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
retriv[2]	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗
Seismic [3]	✗	✓	✓	✗	✓	✗	✗	✓	✗	✗
SentenceBERT [22]	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗
Tevatron [11]	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗
Lightning IR (Ours)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

2 Comparison to Similar Frameworks

Several existing frameworks support fine-tuning and inference with neural retrieval models, but they differ in the supported stages of the retrieval pipeline and the types of models they can handle (see Table 1 for an overview). Frameworks like RAGatouille, SentenceBERT [22], or Seismic [3] focus on specific model types, while frameworks like PyTerrier [18] or Experimaestro-IR [21] support several model types, but not all are available for all stages.

[1] https://github.com/AnswerDotAI/RAGatouille
[2] https://github.com/AmenRa/retriv

WSDM '25, Hannover, Germany

ACM ISBN 979-8-4007-1329-3/25/03
https://doi.org/10.1145/3701551.3704118

Instead, Lightning IR is a framework that implements different models as generic, configurable, and extensible modules. In essence, a model is defined by its backbone encoder and how the contextualized embeddings of a query and document are combined to compute a relevance score. This design allows Lightning IR to support a wide range of model types for all stages. In addition, as all model types use the same API, they can use the same fine-tuning and inference procedures. This makes it easy to compare different model types and to experiment with new ones.

3 Lightning IR: Components

Lightning IR has four central components: (1) model, (2) dataset, (3) trainer, and (4) the CLI. We describe each component in detail in the following sections.

3.1 Model

Lightning IR supports two types of models: cross-encoders and bi-encoders. Both model types are built on top of backbone encoder models from HuggingFace [25]. For example, the following code shows how to initialize a new bi-encoder and cross-encoder model using some pre-trained model from HuggingFace, where `{HF_MODEL}` is a placeholder for the model name.

```python
from lightning_ir import BiEncoderModel, CrossEncoderModel
bi_encoder = BiEncoderModel.from_pretrained("{HF_MODEL}")
cross_encoder = CrossEncoderModel.from_pretrained("{HF_MODEL}")
```

A configuration class defines how each model type uses the contextualized embeddings generated by the backbone model to compute a relevance score. For example, a ColBERT-style model [12] does not pool the contextualized embeddings. It instead uses late interaction to compute relevance scores over the contextualized query and document token embeddings. A ColBERT-style model can be initialized in Lightning IR as follows.

```python
from lightning_ir import BiEncoderConfig, BiEncoderModel
config = BiEncoderConfig(
  similarty_function="dot",
  query_pooling_strategy=None,
  doc_pooling_strategy=None,
  embedding_dim=128,
)
colbert = BiEncoderModel.from_pretrained(
  "bert-base-uncased", config=config
)
```

For usability and reproducibility, we combine a Lightning IR model and tokenizer in a PyTorch Lightning module [9]. The module is responsible for handling training and inference logic, but it also provides convenience functions to quickly score queries and documents. For example, the following code snippet shows how to compute scores using a pre-trained BERT-based bi-encoder and an ELECTRA-based cross-encoder.

```python
from lightning_ir import BiEncoderModule, CrossEncoderModule
bi_encoder = BiEncoderModule("webis/bert-bi-encoder")
cross_encoder = CrossEncoderModule("webis/monoelectra-base")
query = "What is the capital of Germany?"
docs = [
  "Berlin is the capital of Germany.",
  "Paris is the capital of France."
]
print(bi_encoder.score(query, docs).scores.numpy().round(2))
# [39.37 31.4]
print(cross_encoder.score(query, docs).scores.numpy().round(2))
# [ 7.81 -4.13]
```

3.2 Dataset

Lightning IR tightly integrates with `ir_datasets` [17] to provide access to a wide range of common information retrieval datasets, but custom datasets are also supported. Datasets are split into four different classes: document, query, tuple, and run datasets. Document datasets iterate over a document collection and are used for indexing. Query datasets iterate over queries and are used for retrieval. Tuple datasets are used for fine-tuning as they contain samples consisting of a query and multiple documents. Run datasets contain ranked documents for a query (optional: relevance judgments) and are used for re-ranking; also fine-tuning is possible by sampling n-tuples from rankings. The following code snippet shows how to load the MS MARCO passage dataset [1] and the TREC Deep Learning 2019 passage ranking dataset [7].

```python
from lightning_ir import DocDataset, QueryDataset, RunDataset
print(next(iter(DocDataset("msmarco-passage/train"))))
# QuerySample(query_id='121352', query='define extreme')
print(next(iter(QueryDataset("msmarco-passage/train"))))
# DocSample(doc_id='0', doc='The presence of communication ...')
print(RunDataset("msmarco-passage/trec-dl-2019", depth=3)[0])
# RankSample(query_id='1037798', query='who is robert gray',
  doc_ids=('7134595', '7134596', '8402859'), docs=(..., ..., ...))
```

Multiple datasets can be combined into a single PyTorch Lightning datamodule. Similar to a model's Lightning module, datamodules make fine-tuning and inference easier by handling data sampling and batching. They also cleanly separate fine-tuning and evaluation data. The following code snippet shows how to create a datamodule for fine-tuning a bi-encoder model on MS MARCO triples and evaluating it on the TREC Deep Learning 2019 and 2020 passage ranking datasets.

```python
from lightning_ir import LightningIRDataModule, TupleDataset
bi_encoder = BiEncoderModule(...)
data_module = LightningIRDataModule(
  module=module,
  train_dataset=TupleDataset("msmarco-passage/train/triples-v2"),
  inference_datasets=[
    RunDataset("msmarco-passage/trec-dl-2019"),
    RunDataset("msmarco-passage/trec-dl-2020"),
  ]
)
```

3.3 Trainer

The trainer component builds on PyTorch Lightning's trainer class to provide flexible, scalable, and reproducible training. Lightning IR adds functionality to support indexing, retrieval, and re-ranking. The following code snippet shows how to fine-tune a bi-encoder model on the MS MARCO triples dataset. Hyperparameters (e.g., batch size, learning rate, number of epochs) should be adjusted in the module, datamodule, and trainer to the specific use case.

```python
from lightning_ir import LightningIRTrainer
module = BiEncoderModule("{HF_MODEL}", config=...)
data_module = LightningIRDataModule(train_dataset=...)
trainer = LightningIRTrainer(...)
trainer.fit(module, data_module)
```

After fine-tuning a model, the trainer can be used to run inference for all stages of a retrieval pipeline. Indexing, searching, and re-ranking are all implemented as PyTorch Lightning callbacks (but indexing and searching are only needed for bi-encoders). Lightning IR currently supports two indexing and searching methods: Faiss [8] for dense retrieval and a custom PyTorch-based [20] sparse retrieval method. The following code snippet shows how to index and search documents using a fine-tuned bi-encoder model.

```
from lightning_ir import FaissFlatIndexConfig, FaissFlatIndexer
module = BiEncoderModule("{PATH_TO_MODEL}")
data_module = LightningIRDataModule(
  inference_datasets=[DocDataset("msmarco-passage")]
)
index_callback = IndexCallback(
  index_dir="index", index_config=FaissFlatIndexConfig()
)
trainer = LightningIRTrainer(callbacks=[index_callback])
trainer.index(module, data_module)
```

To use an index for retrieval, the path of the index must be passed to a searcher class that matches the indexer class used to create the index. If a dataset has relevance judgments in `ir_datasets` and evaluation metrics are specified in the module's configuration, the trainer will automatically evaluate the retrieval effectiveness. The following code snippet shows how to retrieve documents for a query using the indexed documents.

```
from lightning_ir import FaissSearchConfig, SearchCallback
module = BiEncoderModule(
  "{PATH_TO_MODEL}",
  evaluation_metrics=["nDCG@10"]
)
search_callback = SearchCallback("index", FaissSearchConfig(k=10))
data_module = LightningIRDataModule(inference_datasets=[
  QueryDataset("msmarco-passage/trec-dl-2019/judged")])
trainer = LightningIRTrainer(callbacks=[search_callback])
trainer.search(module, data_module)
```

3.4 CLI

Lightning IR provides a command line interface (CLI) to simplify the usage of the framework. The CLI is built on top of PyTorch Lightning's CLI and provides commands for fine-tuning, indexing, searching, and re-ranking. All options are configurable via command-line arguments or a configuration YAML file. The configuration YAML files are especially useful for reproducibility. For example, to fine-tune a bi-encoder model on the MS MARCO triples dataset, the following command can be used.

```
# > train.yaml
# trainer:
#   ... # trainer hyperparameters
# model:
#   class_path: BiEncoderModule
#   init_args:
#     model_name_or_path: bert-base-uncased
#     config:
#       class_path: BiEncoderConfig
#       init_args:
#         ... # model hyperparameters
# data:
#   class_path: LightningIRDataModule
#   init_args:
#     ... # data hyperparameters
#     train_dataset:
#       class_path: TupleDataset
#       init_args:
#         tuples_dataset: msmarco-passage/train/triples-v2
# optimizer:
#   class_path: torch.optim.AdamW
#   init_args:
#     ... # optimizer hyperparameters
lightning-ir fit --config train.yaml
```

The CLI also supports indexing, searching, and re-ranking via the `index`, `search`, and `re_rank` commands. The configuration file and the command for indexing the MS MARCO passage collection using a fine-tuned bi-encoder model are shown below. The configuration files for searching and re-ranking are similar.

```
# > index.yaml
# trainer:
#   callbacks:
#   - class_path: IndexCallback
#     init_args:
#       index_dir: index
#       index_config:
#         class_path: FaissFlatIndexConfig
# model:
#   class_path: BiEncoderModule
#   init_args:
#     model_name_or_path: {PATH_TO_MODEL}
# data:
#   class_path: LightningIRDataModule
#   init_args:
#     inference_datasets:
#     - class_path: DocDataset
#       init_args:
#         doc_dataset: msmarco-passage
lightning-ir index --config index.yaml
```

4 Lightning IR: Supported Models

Lightning IR supports fine-tuning and running inference on a wide range of bi- and cross-encoder models but we have also added support for a number of popular models not natively fine-tuned in Lightning IR. This includes all bi-encoders from the sentence transformers library [22], SPLADE models released by Naver labs [10], the official ColBERT checkpoints [12], and monoT5 and RankT5 models [19, 27]. Further additional models can be easily added anytime by providing the corresponding configuration files and adding the model to the Lightning IR model registry.

Table 2 compares a selection of supported models when re-ranking the TREC 2019 and 2020 Deep Learning track data [6, 7] using the following configuration file and command.

```
# > re-rank.yaml
# trainer:
#   logger: false
# model:
#   class_path: BiEncoderModule # or CrossEncoderModule
#   init_args:
#     model_name_or_path: {PATH_TO_MODEL}
#     evaluation_metrics:
#     - nDCG@10
# data:
#   class_path: LightningIRDataModule
#   init_args:
#     inference_datasets:
#     - class_path: RunDataset
#       init_args:
#         run_path_or_id: msmarco-passage/trec-dl-2019/judged
#     - class_path: RunDataset
#       init_args:
#         run_path_or_id: msmarco-passage/trec-dl-2020/judged
lightning-ir index --config re-rank.yaml
```

5 Reproducibility Experiment

To demonstrate the capabilities of Lightning IR, we conducted a reproducibility experiment. Using the `bert-base-uncased`[1] model as the backbone, we fine-tuned a single-vector bi-encoder [22], a SPLADE model [10], and a ColBERT model [12]. The models are available in the HuggingFace model hub[2,3,4] along with the corresponding configuration files for reproducing the models and results using the Lightning IR command-line interface.

[1] https://huggingface.co/google-bert/bert-base-uncased
[2] https://huggingface.co/webis/bert-bi-encoder
[3] https://huggingface.co/webis/splade
[4] https://huggingface.co/webis/colbert

Table 2: Effectiveness (nDCG@10) of a selection of models supported by Lightning IR when re-ranking the TREC 2019 and 2020 Deep Learning track data.

Model	TREC DL 2019	TREC DL 2020
Cross-Encoders		
monoELECTRA Large [24]	0.750	0.791
monoT5 3B [19]	0.726	0.752
RankT5 3B [27]	0.721	0.776
Bi-Encoders		
SBERT [22]	0.705	0.735
ColBERT [10]	0.732	0.746
SPLADE [12]	0.715	0.749

Table 3: Effectiveness (nDCG@10) of our fine-tuned models and of the official checkpoints for first-stage retrieval on TREC DL 2019 and 2020. † denotes a statistically significant difference ($p < 0.05$) between our and the original model.

Model	TREC DL 2019	TREC DL 2020
SBERT (Ours)	0.705	0.696
SBERT [22] (Original)	0.705	0.726
SPLADE (Ours)	0.760†	0.720†
SPLADE [10] (Original)	0.722	0.754
ColBERT (Ours)	0.738	0.726
ColBERT [12] (Original)	0.722	0.723

Table 3 compares the effectiveness of our fine-tuned models with the official checkpoints[5,6,7] provided by the authors on the TREC 2019 and 2020 Deep Learning track data [6, 7]. Our fine-tuned models achieve competitive effectiveness compared to the official checkpoints. Minor differences between our results and the official checkpoints can be attributed to randomness in the training process and are not statistically significant, except for the SPLADE models (our SPLADE model is more effective on TREC Deep Learning 2019 and less effective on TREC Deep Learning 2020). These results demonstrate that reproducing the effectiveness of state-of-the-art models is possible with minimal effort in Lightning IR.

6 Conclusion

We have introduced Lightning IR, a PyTorch Lightning-based framework that enables straightforward fine-tuning and inference of transformer-based language models in retrieval tasks. By conducting a reproducibility experiment, we have demonstrated the capabilities of Lightning IR and the simplicity and flexibility of its API. With short code snippets and minimal effort, we were able to fine-tune and evaluate a variety of state-of-the-art models that achieve competitive effectiveness to their official checkpoints. Future work includes extending Lightning IR to support additional efficient dense, sparse, and multi-vector indexing and retrieval pipelines, such as PLAID [23] and Seismic [3], and to improve the latency and scalability of multi-vector and sparse retrieval models.

References

[1] P. Bajaj, D. Campos, N. Craswell, L. Deng, J. Gao, X. Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguyen, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, and T. Wang. 2018. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. arXiv:1611.09268

[2] L. Boytsov and E. Nyberg. 2020. Flexible Retrieval with NMSLIB and FlexNeuART. In *Proc. of NLP-OSS Workshop 2020.* 32–43.

[3] S. Bruch, F. M. Nardini, C. Rulli, and R. Venturini. 2024. Efficient Inverted Indexes for Approximate Retrieval over Learned Sparse Representations. In *Proc. of SIGIR 2023.* 152–162.

[4] B. Clavié. 2024. rerankers: A Lightweight Python Library to Unify Ranking Methods. arXiv:2408.17344

[5] C. Costello, E. Yang, D. Lawrie, and J. Mayfield. 2022. Patapasco: A Python Framework for Cross-Language Information Retrieval Experiments. In *Proc. of ECIR 2022.* 276–280.

[6] N. Craswell, B. Mitra, E. Yilmaz, and D. Campos. 2020. Overview of the TREC 2020 Deep Learning Track. In *Proc. of TREC 2020.* 13 pages.

[7] N. Craswell, B. Mitra, E. Yilmaz, D. Campos, and E. M. Voorhees. 2019. Overview of the TREC 2019 Deep Learning Track. In *Proc. of TREC 2019.* 22 pages.

[8] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. 2024. The Faiss Library. arXiv:2401.08281

[9] W. Falcon and the PyTorch Lightning team. 2024. *PyTorch Lightning v2.4.0.*

[10] T. Formal, B. Piwowarski, and S. Clinchant. 2021. SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking. In *Proc. of SIGIR 2021.* 2288–2292.

[11] L. Gao, X. Ma, J. Lin, and J. Callan. 2022. Tevatron: An Efficient and Flexible Toolkit for Dense Retrieval. arXiv:2203.05765

[12] O. Khattab and M. Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proc. of SIGIR 2020.* 39–48.

[13] X. Li, J. Lipp, A. Shakir, R. Huang, and J. Li. 2024. BMX: Entropy-weighted Similarity and Semantic-enhanced Lexical Search. arXiv:2408.06643

[14] J. Lin, X. Ma, S.-C. Lin, J.-H. Yang, R. Pradeep, and R. Nogueira. 2021. Pyserini: An Easy-to-Use Python Toolkit to Support Replicable IR Research with Sparse and Dense Representations. In *Proc. of SIGIR 2021.* 2356–2362.

[15] J. Lin, R. Nogueira, and A. Yates. 2022. *Pretrained Transformers for Text Ranking: BERT and Beyond.* Springer Nature.

[16] S. MacAvaney. 2020. OpenNIR: A Complete Neural Ad-Hoc Ranking Pipeline. In *Proc. of WSDM 2020.* 845–848.

[17] S. MacAvaney, A. Yates, S. Feldman, D. Downey, A. Cohan, and N. Goharian. 2021. Simplified Data Wrangling with ir_datasets. In *Proc. of SIGIR 2021.* 2429–2436.

[18] C. Macdonald, N. Tonellotto, S. MacAvaney, and I. Ounis. 2021. PyTerrier: Declarative Experimentation in Python from BM25 to Dense Retrieval. In *Proc. of CIKM 2021.* 4526–4533.

[19] R. Nogueira, Z. Jiang, R. Pradeep, and J. Lin. 2020. Document Ranking with a Pretrained Sequence-to-Sequence Model. In *Findings of EMNLP 2020.* 708–718.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proc. of NeurIPS 2019.* 8024–8035.

[21] B. Piwowarski. 2020. Experimaestro and Datamaestro: Experiment and Dataset Managers (for IR). In *Proc. of SIGIR 2020.* 2173–2176.

[22] N. Reimers and I. Gurevych. 2019. Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks. In *Proc. of EMNLP-IJCNLP 2019.* 3980–3990.

[23] K. Santhanam, O. Khattab, C. Potts, and M. Zaharia. 2022. PLAID: An Efficient Engine for Late Interaction Retrieval. In *Proc. of CIKM 2022.* 1747–1756.

[24] F. Schlatt, M. Fröbe, H. Scells, S. Zhuang, B. Koopman, G. Zuccon, B. Stein, M. Potthast, and M. Hagen. 2025. Rank-DistiLLM: Closing the Effectiveness Gap Between Cross-Encoders and LLMs for Passage Re-Ranking. In *Proc. of ECIR 2025.* 10 pages. (to appear).

[25] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771

[26] A. Yates, S. Arora, X. Zhang, W. Yang, K. M. Jose, and J. Lin. 2020. Capreolus: A Toolkit for End-to-End Neural Ad Hoc Retrieval. In *Proc. of CIKM 2020.* 861–864.

[27] H. Zhuang, Z. Qin, R. Jagerman, K. Hui, J. Ma, J. Lu, J. Ni, X. Wang, and M. Bendersky. 2022. RankT5: Fine-Tuning T5 for Text Ranking with Ranking Losses. arXiv:2210.10634

[5] https://huggingface.co/sentence-transformers/msmarco-bert-base-dot-v5

[6] https://huggingface.co/naver/splade-v3

[7] https://huggingface.co/colbert-ir/colbertv2.0