Actes de CORIA-TALN 2023
Paris, France, 5 au 9 juin 2023

XPMIR: Une bibliothèque modulaire pour l'apprentissage d'ordonnancement et les expériences de RI neuronale

Yuxuan Zong[1] Benjamin Piwowarski[1, 2]
(1) ISIR, Sorbonne Université, 4, Place Jussieu, 75005 Paris, France
(2) CNRS
yuxuan.zong@isir.upmc.fr, benjamin@piwowarski.fr

RÉSUMÉ

Ces dernières années, plusieurs librairies pour la recherche d'information (neuronale) ont été proposées. Cependant, bien qu'elles permettent de reproduire des résultats déjà publiés, il est encore très difficile de réutiliser certaines parties des chaînes de traitement d'apprentissage, comme par exemple le pré-entraînement, la stratégie d'échantillonnage ou la définition du coût dans les modèles nouvellement développés. Il est également difficile d'utiliser de nouvelles techniques d'apprentissage avec d'anciens modèles, ce qui complique l'évaluation de l'utilité des nouvelles idées pour les différents modèles de RI neuronaux. Cela ralentit l'adoption de nouvelles techniques et, par conséquent, le développement du domaine de la RI. Dans cet article, nous présentons XPMIR, une librairie Python définissant un ensemble réutilisable de composants expérimentaux. La bibliothèque contient déjà des modèles et des techniques d'indexation de pointe, et est intégrée au hub HuggingFace.

ABSTRACT

XPMIR : A Modular Library for Learning to Rank and Neural IR experiments

Those last 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, 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 re-usable set of experimental components. The library already contains state-of-the-art models and indexation techniques, and is integrated with the HuggingFace hub.

MOTS-CLÉS : recherche d'information neuronale, apprentissage d'ordonnancement, cadre expérimental.

KEYWORDS: neural information retrieval, learning to rank, experimental framework.

1 Introduction

Le développement de modèles (neuronaux) de recherche d'information dépend d'une chaîne de traitement complexe (par exemple, le pré-traitement, l'entraînement, l'indexation et l'évaluation). Une série de bibliothèques (en Python) a été proposée pour faciliter la reproduction des résultats expérimentaux. Bien que ces bibliothèques soient pratiques lorsqu'il s'agit de ré-exécuter une expérience en changeant un ou des paramètres spécifiques, elles sont moins utiles lorsqu'il s'agit de créer de

Actes de la 18e Conférence en Recherche d'Information et Applications (CORIA), pages 222–233.

nouveaux modèles.

Plus précisément, il n'est pas facile de modifier la chaîne de traitement car ces projets ne sont pas assez modulaires. Nous pensons que cela explique pourquoi les nouveaux modèles ne sont pas développés dans ces bibliothèques – empêchant ainsi l'adoption de certains *composants* dans de nouveaux modèles. Concevoir de nouveaux modèles qui tirent parti des innovations récentes prend du temps, ce qui ce qui ralentit la recherche et le partage des codes.

L'objectif de la bibliothèque XPMIR est de proposer un ensemble de *composants réutilisables* permettant de concevoir de nouvelles expériences pour la recherche d'information (neuronale), ainsi que de reproduire les anciennes. XPMIR atteint cet objectif en :

1. Adoptant des abstractions pour décrire les différentes sources de données avec des adaptateurs de jeux de données et d'échantillonneurs ;
2. Définir une manière standardisée d'apprendre, en utilisant des *hooks* (crochets) pour modifier le comportement d'apprentissage ;
3. Fournir un ensemble de composants réutilisables (échantillonneurs, modèles neuronaux, représentation textuelle, chaîne d'outils d'évaluation) ;
4. Fournir des modèles pré-entraînés sur HuggingFace. [1]

La bibliothèque est open source (licence GPLv3) et disponible sur GitHub [2]. XPMIR s'appuie sur le cadre experimaestro/datamaestro (21) qui permet (1) de standardiser l'accès aux jeu de données ; (2) de définir des composants expérimentaux modulaires (configuration et tâches) ; (3) d'exprimer des plans expérimentaux complets avec suivi automatique des dossiers (chaque dossier correspondant à un ensemble unique de paramètres expérimentaux) ; (4) de contrôler les tâches soumises (par un planificateur). Par rapport à ces librairies qui sont génériques, XPMIR apporte un ensemble de composants spécialisés *pour la recherche d'information* qui sont décrits dans la suite.

2 Travaux connexes

Tout d'abord, il existe de nombreuses bibliothèques liées à la RI. Parmi celles-ci, nous pouvons citer :

Datasets ir-datasets (13) propose une API pour accéder à de nombreux jeux de données de RI et les télécharger (s'ils sont disponibles gratuitement) ;

Indexation et recherche Il existe de nombreuses bibliothèques qui traitent de l'indexation. Pour les modèles de RI standard, nous pouvons citer PYSERINI (26), PYTERRIER (14) et Pisa (16), et pour les modèles neuronaux denses de RI, FAISS (7).

Evaluation ir-measures (13) est une bibliothèque récente qui fournit un accès direct à une grande diversité de métriques de RI.

Lorsque cela est possible, XPMIR réutilise les bibliothèques existantes en fournissant des composants qui encapsulent l'accès à ces bibliothèques, de sorte qu'elles puissent être réutilisées dans différentes expériences. Plus précisément, à ce jour, XPMIR fournit des composants pour ir-datasets, ir-measures, Pyserini, et FAISS. Une bibliothèque en RUST a également été développée pour traiter le cas des modèles neuronaux de RI parcimonieux.

1. La liste actuelle des modèles pré-entraînés est disponible sur https://huggingface.co/models?library=xpmir
2. La documentation se trouve à https://experimaestro-ir.readthedocs.io/ et le code source à https://github.com/experimaestro/experimaestro-ir

Plus en rapport avec XPMIR, diverses bibliothèques pour la RI neuronale ont été proposées ces dernières années, telles que OPENNIR [3] (11), MATCHMAKER [4] (5) ou CAPREOLUS [5] (Yates *et al.*) – ou des librairies de code plus spécifiques liés à une classe de modèles, comme par exemple pour les modèles ColBERT [6] ou SPLADE [7].

Pour OPENNIR et MATCHMAKER, les expériences sont toutes configurées par un fichier de paramètres (ou via la ligne de commande) qui permet de modifier certains aspects du processus d'apprentissage, mais pas de combiner facilement des parties de la chaîne de traitement. CAPREOLUS, qui est le plus proche de XPMIR, définit un ensemble de composants (*modules*), mais s'appuie sur des tâches prédéfinies (par exemple, apprendre et évaluer) dont certaines parties sont configurables. Toutefois, cette librairie est relativement rigide car il n'y a aucun moyen facile de modifier une chaîne de traitement.

Nous pensons que ces bibliothèques ne sont pas suffisamment modulaires. Par exemple, que se passe-t-il si nous voulons utiliser une étape de pré-entraînement MLM (Masked Language Model) spécifique tel que LexMAE (23) pour le modèle ColBERT (8) ? Que se passe-t-il si nous voulons utiliser un modèle pour générer des exemples négatifs difficiles à partir d'un nouveau modèle de RI neuronale ? Dans ces bibliothèques, il faut coder explicitement toute la "colle" entre les différentes parties, mais cela prend du temps et est source d'erreurs car les composants n'ont pas forcément été conçu pour être ré-utilisés de manière indépendante. En comparaison, la bibliothèque XPMIR a pour objectif de

— Proposer un ensemble de composants réutilisables pour apprendre et évaluer les modèles neuronaux de RI ;

— Permettre de combiner ces composants et de concevoir des expériences complexes grâce à experimaestro et datamaestro (21) ;

— Fournir un ensemble de composants d'expérience de haut niveau (par exemple ceux utilisés dans une tâche typique de ré-ordonnancement sur MS-Marco) similairement à ce qui est fait dans CAPREOLUS ;

— Fournir une intégration avec HuggingFace pour réutiliser les modèles de RI pré-entraînés.

3 XPMIR

Cette section décrit les différents composants de la bibliothèque XPMIR [8], regroupés par catégorie :

Jeux de données Ces composants permettent d'accéder à la bibliothèque ir-datasets, ainsi qu'à d'autres jeux de données spécifiques à la RI (par exemple, des triplets pour apprendre des modèles neuronaux) et à des adaptateurs qui peuvent traiter et transformer les jeux de données ;

Retrievers et Scorer Ces composants définissent comment représenter un texte, un modèle classique de RI et enfin les modèles neuronaux de RI ;

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. Ces composants correspondent soit à des configurations, soit à des tâches dans experimaestro : les configurations décrivent simplement les paramètres expérimentaux, tandis que les tâches correspondent au code réel qui peut être exécuté - par exemple, lors de l'indexation d'une collection, de l'apprentissage ou de l'évaluation d'un modèle

Apprentissage d'ordonnancement Ces composants permettent de constituer une chaîne d'entraînement ;

Évaluation Les composants permettent l'évaluation des modèles appris.

Nous illustrons certains composants par des extraits de code (modifiés) correspondant à des reproductions de deux modèles état de l'art dans XPMIR, à savoir MonoBERT (18) et SPLADE (4). MonoBERT (18) est un modèle d'encodeur joint bien établi pour la RI neuronale, qui ordonne les documents en deux étapes : sélection de candidats avec BM25 ou un modèle neuronal léger, puis ré-ordonnancement avec monoBERT. L'autre modèle qui nous sert de source pour nos exemples est SPLADE (4), un modèle d'ordonnancement qui est basé sur une représentation parcimonieuse des documents et des questions. Le code complet, qui se trouve dans la documentation [9], illustre des plans expérimentaux complexes comprenant des étapes de pré/post-traitement avec des composants d'apprentissage d'ordonnancement, ainsi que l'évaluation du modèle appris et sa comparaison avec d'autres modèles – illustrant le fait que XPMIR pourrait être utilisé pour garantir la reproductibilité d'un article de recherche en fournissant le code *complet* des expériences (y compris les modèles de base et les variations).

3.1 Jeux de données

XPMIR fournit des jeux de données au format défini par datamaestro (21), une bibliothèque qui permet d'accéder aux jeux de données de différents types de manière homogène. Grâce à datamaestro, XPMIR permet de ré-utiliser les jeux de données de ir-datasets (13), ce qui lui permet d'accéder facilement à la plupart des jeux de données de la RI. XPMIR permet également d'accéder à des jeux de données utilisés pour la distillation tels que ceux de (6). Enfin, chaque type de données – documents, questions, jugements de pertinence, triplets d'apprentissage – est associé à une interface Python permettant d'accéder aux données sous-jacentes.

XPMIR fournit également des adaptateurs [10] permettant de transformer un jeu de données. Par exemple, `RandomFold` permet échantillonner des questions d'un jeu de données de RI (et de conserver l'information sur les documents et les jugements de pertinence correspondants aux questions sélectionnées). Le code suivant montre comment échantillonner 500 questions à partir du jeu de données MS-Marco dev - en excluant les questions du jeu de données "dev small" utilisé pour évaluer les modèles dans la plupart des articles :

```
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()
```

Une autre transformation utile du jeu de données est la création d'une collection basée sur des moteurs de recherche, qui est composée de tous les documents renvoyés par un modèle de RI donné. Ceci est utile pour le calcul des mesures de validation pour les modèles neuronaux utilisés dès la première

9. https://experimaestro-ir.readthedocs.io/en/latest/papers/monobert.html pour monoBERT et https://experimaestro-ir.readthedocs.io/en/latest/papers/splade.html pour SPLADE
10. https://experimaestroir.readthedocs.io/en/latest/data/adapters.html

étape de recherche (comme SPLADE), i.e. pour rechercher de manière efficace des documents dans une grande collection.

3.2 Moteurs de recherche et les modèles neuronaux de RI

3.2.1 Représentation du texte et modèles neuronaux

De nombreux modèles s'appuient sur une certaine forme de représentation du texte. Dans XPMIR, nous distinguons trois types de représentations textuelles : (1) un `Tokenizer` qui renvoie une liste d'ID de tokens ; (2) un `TokensEncoder` qui renvoie une représentation par token ; (3) des encodeurs qui associent un vecteur à un texte (`TextEncoder`, qui peut être utilisé pour transformer une collection de documents en un index FAISS), à une paire de textes (`DualTextEncoder`, par exemple pour une paire requête/document), ou à un triplet (`TripletTextEncoder`, par exemple pour un triplet requête/document/document). Ces encodeurs sont à la base des modèles denses, des encoders joints comme MonoBERT (18) ou comme duoBERT (19). Cette représentation du texte est gérée par deux ensembles concrets de classes, celles qui correspondent à des représentations de mots comme GloVe (20) ou word2vec (17), et peuvent être utilisés pour définir des modèles pré-BERT, et ceux qui exploitent les Transformers de HuggingFace (25).

3.2.2 Retrievers and Scorers

Les `Scorers` sont chargés de calculer un score pour une question et un document donné – i.e. tous les modèles neuronaux sont des `Scorers`. Les modèles neuronaux sont organisés au sein d'une hiérarchie qui permet de factoriser autant de propriétés communes que possible. Par exemple, un `DualRepresentationScorer` est un modèle neuronal qui représente séparément les documents et les questions avant de calculer une similarité. Les modèles denses font partie de cette famille, de même que les modèles d'interaction tardive comme ColBERT (8). Un autre exemple est un `DualVectorScorer` qui s'appuie sur deux représentations vectorielles (questions et documents), et fournit des moyens d'accélérer l'apprentissage lors de l'utilisation de négatifs de batch (22). Dans la pratique, les `Scorers` sont souvent définis comme une composition d'une fonction qui calcule un score avec un modèle qui permet de représenter un texte dans un espace vectoriel. Par exemple, le modèle monoBERT peut être défini comme la composition d'un `CrossScorer`, un classificateur de représentations de couples question/document fournies par un `DualTransformerEncoder`, comme le montre le code ci-dessous [11] :

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

11. Ce code montre également comment les composants – `CrossScorer` et `DualTransformerEncoder` – peuvent être composés dans XPMIR

3.2.3 Retrievers

Les `Retrievers` permettent d'effectuer des recherches dans une *grande* collection de documents *de manière efficace*. Les modèles les plus simples sont les modèles de RI standard, comme par exemple BM25. Pour ces modèles, un `Retriever` peut être créé à partir d'un index. Pour l'instant, un adaptateur Pyserini (10) est fourni, mais d'autres comme pour PyTerrier (14) seraient faciles à mettre en œuvre. Le code suivant montre comment définir un `Retriever` pour le modèle BM25 basé sur Pyserini (10) :

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

D'autres types d'indices sont également pris en charge pour permettre une recherche rapide à l'aide de modèles neuronaux. Les indices denses sont possibles via l'intégration de la librairie FAISS (7). Les modèles neuronaux parcimonieux produisent des indices avec une distribution de poids différente des modèles de RI standard (15). Une librairie associée a été écrite en Rust [12] et permet d'indexer une collection de documents à partir des vecteurs parcimonieux qui représentent les documents. Au niveau la recherche, les algorithmes WAND (1) et MaxScore (24) sont actuellement implémentés. À titre d'illustration, le code ci-dessous montre comment définir un `Retriever` pour le modèle SPLADE :

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

Enfin, pour les modèles neuronaux basés sur l'interaction, et qui ne peuvent être utilisés que pour ré-ordonner les documents, la classe `TwoStageRetriever` utilise un `Retriever` pour sélectionner un sous-ensemble de documents (ex. avec BM25) avant d'utiliser un `Scorer` pour les ré-ordonner. La définition d'un tel `Retriever` est simple :

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

3.3 Apprentissage d'ordonnancement

Le processus d'apprentissage est géré par différents composants que nous décrivons ci-dessous.

Optimisation : optimiseurs et planificateurs La partie relative à l'optimisation définit la manière d'effectuer un pas de gradient. Elle repose sur la définition d'une série d'optimiseurs, chacun étant chargé d'optimiser une partie des paramètres du modèle, ce qui est utile lorsque les paramètres

12. https://github.com/experimaestro/experimaestro-ir-rust

doivent être optimisés différemment, comme lors de l'affinage des Transformers (les couches de normalisation et les paramètres des biais ne doivent pas être inclus dans le coût de régularisation L_2). Chaque taux d'apprentissage de l'optimiseur peut être contrôlé par un planificateur – ce qui est encore une fois couramment utilisé lors de l'affinage des Transformers (25).

Le code suivant illustre comment définir l'optimiseur pour l'apprentissage de monoBERT, où le premier optimiseur utilise Adam (9) en évitant la régularisation L2 pour les biais ou les couches de normalisation (paramètres dont le nom se termine par `bias` ou contenant `LayerNorm`), tandis que le second traite tous les autres paramètres avec l'optimiseur AdamW. Cet exemple illustre de plus la flexibilité des composants exposés par 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),
    ),
]
```

Entraîneur and échantillonneur L'entraîneur (*Trainer*) est chargé d'effectuer une étape d'apprentissage. Il existe différents entraîneurs en fonction du type d'échantillons qu'ils peuvent traiter (ponctuelle, par paire, ou par lot). Certains entraîneurs sont conçus pour gérer la distillation, qui est essentielle pour obtenir des modèles avec des performances au niveau de l'état de l'art (**?** 3). Les données sont transmises aux entraîneurs par le biais d'échantillonneurs, qui sont chargés de fournir des échantillons de données dont le type est variable (par exemple, par points ou par paires). Enfin, des hooks peuvent être utilisés pour modifier le processus d'apprentissage, ce qui permet par exemple de calculer des coûts de régularisation.

Le code ci-dessous montre comment construire un entraîneur basé sur la distillation (utilisé pour entraîner SPLADE_DistilMSE (3)), où `sampler` est un itérateur sur les tuples composés d'une question, de deux documents et de leurs scores calculés par monoBERT :

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

Learner Enfin, la classe `Learner` est la classe qui gère l'ensemble du processus d'apprentissage. Elle s'appuie sur :

1. un modèle neuronal dont les paramètres doivent être appris (le `Scorer`);
2. un entraîneur qui spécifie comment apprendre le modèle (ex. en utilisant un coût de classification ponctuel ou par paire) et avec quelles données (à l'aide d'échantillonneurs);
3. un optimiseur qui spécifie comment effectuer la descente de gradient;
4. un ou plusieurs `Listeners` qui surveillent le processus d'apprentissage – le plus important étant celui de validation, qui permet de conserver les paramètres du modèle qui maximisent un certain nombre de métriques de validation.

En outre, le `Learner` peut être modifié à l'aide de *hooks* permettant de modifier certaines parties du processus d'apprentissage. Un exemple de *hook* permet de distribuer les modèles sur plusieurs GPU. Un autre exemple est de modifier le modèle en "gelant" certains poids – ces paramètres ne seront pas modifiés lors de l'apprentissage. Le processus d'apprentissage est divisé en époques, chaque époque étant définie par un certain nombre d'étapes d'apprentissage (c'est-à-dire par un nombre d'échantillons) effectuées par l'entraîneur. Une époque ne correspond pas à un passage complet sur le jeu de données (ce qui est nécessaire car certaines collections peuvent être très grandes, par exemple si elles sont échantillonnées à partir d'un `Retriever`). Le code suivant montre comment nous définissons un `Learner` pour monoBERT, et obtenons le modèle qui maximise la métrique `RR@10` sur un ensemble de validation :

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

3.4 Évaluation

L'évaluation du modèle est l'étape finale des expériences de RI. Pour faciliter l'évaluation, une classe `EvaluationsCollection` référence les différents jeux de données et métriques sur lesquels l'évaluation doit être effectuée pour chaque modèle. Les métriques réelles sont calculées grâce à `ir-measures` (12). Le code ci-dessous montre un exemple d'évaluation de monoBERT – la `retriever_factory` définit comment construire un `Retriever` à partir de monoBERT (en utilisant un `TwoStageRetriever` avec `best_rr10` comme `Scorer`) en fonction de la collection de documents :

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

TABLE 1 – Reproduction de monoBERT et de 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 (18)	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

```
)
tests.evaluate_retriever(retriever_factory)
```

4 Reproduction d'articles et intégration avec HuggingFace

Une partie de la bibliothèque XPMIR est dédiée à la reproduction (partielle) de certains articles de RI. Une interface (ligne de commande) est fournie pour permettre reproduire les expériences effectuées, et permettent d'automatiser le téléversement vers HuggingFace Hub, en incluant les métriques d'apprentissage et résultats de l'évaluation [13]. Deux reproductions (partielles) d'articles sont actuellement mises en œuvre dans la bibliothèque XPMIR, à savoir MonoBERT (18) et SPLADE (3). Pour la reproduction de MonoBERT, nous utilisons BERT-base comme point de départ, et pour SPLADE, DistilBERT. Dans les deux cas, nous utilisons les métriques MRR@10 et nDCG@10 et évaluons le modèle sur la recherche de passages MS-MARCO (ensemble de développement) et TREC Deep Learning 2019. Le tableau 1 montre que les résultats obtenus avec XPMIR correspondent à ceux rapportés dans les articles.

Outre la définition de composants pour la définition, l'entraînement et l'évaluation de modèles neuronaux de RI, XPMIR fournit une intégration avec le HuggingFace Hub. Cela permet de téléverser et de télécharger des modèles pré-entraînés [14]. Ces modèles peuvent être utilisés dans d'autres expériences et/ou affinés sur des jeux de données spécifiques en tirant parti de XPMIR. Le code ci-dessous montre comment créer un `Scorer` à partir du modèle dense pré-entraîné tas-balanced (6) :

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

13. Un exemple avec monoBERT est disponible sur https://huggingface.co/xpmir/monobert.
14. Au moment de la rédaction, monoBERT (18), TAS-Balanced (6), ainsi que diverses versions de SPLADE (2) sont disponibles. La liste actuelle des modèles pré-entraînés est disponible sur le HuggingFace Hub : https://huggingface.co/models?library=xpmir

Ce `Scorer` peut ensuite être réutilisé dans une autre expérience – pour servir à générer de nouveaux échantillons négatifs, pour être évalué sur une nouvelle collection d'évaluation RI, ou bien pour être affiné sur d'autres jeux de données.

5 Conclusion

Dans cet article, nous avons présenté la librairie XPMIR qui permet d'entraîner et d'évaluer des modèles de RI (neuronaux). XPMIR est basé sur l'idée de décomposer le pré-traitement des ensembles de données, la représentation du texte, les modèles neuronaux de RI, l'apprentissage d'ordonnancement et l'évaluation en un ensemble de composants réutilisables – qui exploitent les bibliothèques de RI existantes dans la mesure du possible. Ces composants peuvent être composés par l'intermédiaire de la bibliothèque experimaestro (21) pour définir des plans expérimentaux complexes de RI neuronale. XPMIR fournit également des composants de haut niveau qui facilitent la description d'expériences standard (par exemple, l'entraînement d'un ré-ordonnanceur pour MS Marco).

L'intérêt d'utiliser de tels composants est qu'il est beaucoup plus facile de réutiliser certains d'entre eux pour de nouveaux modèles et de concevoir des plans expérimentaux comparant différents modèles. Avec notre structure de code actuelle et les différents composants déjà présentés, de nombreux autres articles pourraient être facilement mis en œuvre. Par exemple, nous travaillons actuellement à la reproduction de duoBERT (19) et ColBERT (8), et nous prévoyons d'inclure des procédures de pré-entraînement récentes et spécifiques à la RI comme par exemple (23) ainsi que de l'auto-distillation comme utilisée dans (3). Finalement, la librairie est open-source (licence GPLv3) et les contributions et commentaires sont les bienvenus.

Références

[1] BRODER A. Z., CARMEL D., HERSCOVICI M., SOFFER A. & ZIEN J. (2003). Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, p. 426–434, New York, NY, USA : Association for Computing Machinery. DOI : 10.1145/956863.956944.

[2] FORMAL T., LASSANCE C., PIWOWARSKI B. & CLINCHANT S. (2021a). Splade v2 : Sparse lexical and expansion model for information retrieval. (arXiv :2109.10086). arXiv :2109.10086 [cs].

[3] FORMAL T., LASSANCE C., PIWOWARSKI B. & CLINCHANT S. (2022). From distillation to hard negative sampling : Making sparse neural ir models more effective. (arXiv :2205.04733). arXiv :2205.04733 [cs].

[4] FORMAL T., PIWOWARSKI B. & CLINCHANT S. (2021b). *SPLADE : Sparse Lexical and Expansion Model for First Stage Ranking*. Rapport interne. arXiv : 2107.05720.

[5] HOFSTÄTTER S. (2019). Matchmaker.

[6] HOFSTÄTTER S., LIN S.-C., YANG J.-H., LIN J. & HANBURY A. (2021). Efficiently teaching an effective dense retriever with balanced topic aware sampling. (arXiv :2104.06967). arXiv :2104.06967 [cs].

[7] JOHNSON J., DOUZE M. & JÉGOU H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, **7**(3), 535–547.

[8] KHATTAB O. & ZAHARIA M. (2020). ColBERT : Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. *arXiv :2004.12832 [cs]*. arXiv : 2004.12832.

[9] KINGMA D. P. & BA J. (2017). Adam : A method for stochastic optimization.

[10] LIN J., MA X., LIN S.-C., YANG J.-H., PRADEEP R. & NOGUEIRA R. (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*, SIGIR '21, p. 2356–2362, New York, NY, USA : Association for Computing Machinery. DOI : 10.1145/3404835.3463238.

[11] MACAVANEY S. (2020). OpenNIR : A complete neural ad-hoc ranking pipeline. In *WSDM 2020*.

[12] MACAVANEY S., MACDONALD C. & OUNIS I. (2022). *Streamlining evaluation with ir-measures*, In M. HAGEN, S. VERBERNE, C. MACDONALD, C. SEIFERT, K. BALOG, K. NØRVÅG & V. SETTY, Éds., *Advances in Information Retrieval*, volume 13186 de *Lecture Notes in Computer Science*, p. 305–310. Springer International Publishing : Cham. DOI : 10.1007/978-3-030-99739-7_38.

[13] MACAVANEY S., YATES A., FELDMAN S., DOWNEY D., COHAN A. & GOHARIAN N. (2021). Simplified Data Wrangling with ir_datasets. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, p. 2429–2436. New York, NY, USA : Association for Computing Machinery.

[14] MACDONALD C. & TONELLOTTO N. (2020). Declarative experimentation ininformation retrieval using pyterrier. In *Proceedings of ICTIR 2020*.

[15] MACKENZIE J., MALLIA A. & MOFFAT A. (2022). Accelerating Learned Sparse Indexes Via Term Impact Decomposition. In *Findings of the Association for Computational Linguistics : EMNLP 2022*.

[16] MALLIA A., SIEDLACZEK M., MACKENZIE J. & SUEL T. (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.*, p. 50–56.

[17] MIKOLOV T., SUTSKEVER I., CHEN K., CORRADO G. & DEAN J. (2013). Distributed Representations of Words and Phrases and their Compositionality. In *NIPS'14*, volume cs.CL, p. 3111–3119.

[18] NOGUEIRA R. & CHO K. (2020). Passage Re-ranking with BERT. arXiv :1901.04085 [cs], DOI : 10.48550/arXiv.1901.04085.

[19] NOGUEIRA R., YANG W., CHO K. & LIN J. (2019). Multi-Stage Document Ranking with BERT. *arXiv :1910.14424 [cs]*. ZSCC : 0000001 arXiv : 1910.14424.

[20] PENNINGTON J., SOCHER R. & MANNING C. (2014). Glove : Global Vectors for Word Representation.

[21] PIWOWARSKI B. (2020). Experimaestro and Datamaestro : Experiment and Dataset Managers (for IR). In *ACM SIGIR 2020*, Xian, China. ZSCC : NoCitationData[s0], DOI : 10.1145/3397271.3401410.

[22] QU Y., DING Y., LIU J., LIU K., REN R., ZHAO W. X., DONG D., WU H. & WANG H. (2021). Rocketqa : An optimized training approach to dense passage retrieval for open-domain question answering. In *In Proceedings of NAACL*.

[23] SHEN T., GENG X., TAO C., XU C., HUANG X., JIAO B., YANG L. & JIANG D. (2022). LexMAE : Lexicon-Bottlenecked Pretraining for Large-Scale Retrieval. In *ICLR* : arXiv. arXiv :2208.14754 [cs], DOI : 10.48550/arXiv.2208.14754.

[24] TURTLE H. & FLOOD J. (1995). Query evaluation : Strategies and optimizations. *Information Processing & Management*, **31**(6), 831–850. DOI : 10.1016/0306-4573(95)00020-H.

[25] WOLF T., DEBUT L., SANH V., CHAUMOND J., DELANGUE C., MOI A., CISTAC P., RAULT T., LOUF R., FUNTOWICZ M., DAVISON J., SHLEIFER S., VON PLATEN P., MA C., JERNITE Y., PLU J., XU C., SCAO T. L., GUGGER S., DRAME M., LHOEST Q. & RUSH A. M. (2020). HuggingFace's Transformers : State-of-the-art Natural Language Processing. arXiv :1910.03771 [cs], DOI : 10.48550/arXiv.1910.03771.

[26] YANG P., FANG H. & LIN J. (2018). Anserini : Reproducible Ranking Baselines Using Lucene. **10**(4). DOI : 10/ggmdws.

[Yates *et al.*] YATES A., ARORA S., ZHANG X., YANG W., JOSE K. M. & LIN J. Capreolus : A Toolkit for End-to-End Neural Ad Hoc Retrieval. WSDM '20, p. 861–864. DOI : 10/ggjnkm.