Flexible IR Pipelines with Capreolus

Andrew Yates,[1] Kevin Martin Jose,[1,2] Xinyu Zhang,[3*] Jimmy Lin[3]
[1] Max Planck Institute for Informatics, Saarland Informatics Campus
[2] Saarland University, Saarland Informatics Campus
[3] David R. Cheriton School of Computer Science, University of Waterloo

ABSTRACT

While a number of recent open-source toolkits for training and using neural information retrieval models have greatly simplified experiments with neural reranking methods, they essentially hard code a "search-then-rerank" experimental pipeline. These pipelines consist of an efficient first-stage ranking method, like BM25, followed by a neural reranking method. Deviations from this setup often require hacks; some improvements, like adding a second reranking step that uses a more expensive neural method, are infeasible without major code changes. In order to improve the flexibility of such toolkits, we propose implementing experimental pipelines as dependency graphs of functional "IR primitives," which we call modules, that can be used and combined as needed.

For example, a neural IR pipeline may rerank results from a Searcher module that efficiently retrieves results from an Index module that it depends on. In turn, the Index depends on a Collection to index, which is provided by the pipeline. This Searcher module is self-contained: the pipeline does not need to know about or interact with the Index of the Searcher, which is transparently shared among Searcher modules when possible (e.g., a BM25 and a QL Searcher might share the same Index). Similarly, a Reranker module might depend on a Trainer (e.g., Tensorflow), feature Extractor, Tokenizer, etc. In both cases, the pipeline needs to interact only with the Reranker or Searcher directly; the complexity of their dependencies is hidden and intelligently managed. We rewrite the Capreolus toolkit to take this approach and demonstrate its use.

ACM Reference Format:
Andrew Yates, Kevin Martin Jose, Xinyu Zhang, and Jimmy Lin. 2020. Flexible IR Pipelines with Capreolus. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3340531.3412780

1 INTRODUCTION

Modern ad-hoc IR architectures are complex, with multi-stage (often called telescoping) pipelines consisting of several reranking methods run in a series [22]. Each method makes a different efficiency vs. effectiveness trade-off and potentially operates on different features or document representations. With the growing popularity of computationally expensive BERT-based models (e.g., [2, 6, 13, 17]) and substantially more expensive models based on T5 [18], the telescoping approach becomes particularly appealing as a means for reducing the number of documents these models evaluate.

Recently, several toolkits have been proposed that implement reranking pipelines in the context of neural models [12, 29]. In this setting, an efficient first-stage ranking method like BM25 uses an inverted index to identify a pool of candidate documents, and a neural model then reranks these candidate documents to form a final ranked list. Such toolkits greatly reduce the difficulty of experimentation and of reproducing prior results by making a specific pipeline highly configurable (e.g., by easily changing datasets, folds, first-stage and neural ranking methods, etc). For example, in OpenNIR [12] pipelines, first-stage retrieval is performed by Anserini [27] and followed by a reranking step with a neural model implemented in PyTorch [21]. Anserini's first-stage ranking method and the neural model are configurable, with options for parameters like the batch size, learning rate, loss function, and embeddings used with the neural model.

While these neural reranking toolkits are no doubt an improvement over manually stitching together different components in an ad hoc manner for each experiment, they essentially hard code a "search-then-rerank" (most often, with PyTorch) pipeline and thus cannot flexibly support explorations of different architectural alternatives. For example, it would require substantial code changes to add a second reranking step, or to extend the pipeline to compute field similarity scores with neural IR (NIR) models that are then used as learning-to-rank (LTR) features. An unanticipated use case like this is all but impossible to handle with a rigid pipeline, so the tight coupling of these toolkits' components means that they are largely an "all-or-nothing" proposition. The user's only options are to abandon the toolkit entirely or to code a "pipeline of pipelines" that repeatedly runs pipelines and glues the output together in order to compute the desired features. While this approach looks acceptable at first glance, it means that the experimental advantages provided by the toolkit are lost. The user is again responsible for managing the inputs and outputs of both each neural feature pipeline and of the final LTR pipeline, which includes computing and caching other features in a way that is compatible with the experimental setup. Such burdens do not exist when the toolkit is used with the intended "search-then-rerank-with-PyTorch" paradigm.

Our Approach. Rather than defining *a* rigid experimental pipeline, we propose defining "IR primitives" that may be used and composed in arbitrary ways. Each primitive, called a module in our terminology, declares configuration options and dependencies on other modules to form a self-contained directed acyclic graph (DAG). The

*This work was conducted while the author was an intern at the Max Planck Institute for Informatics.

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10.
https://doi.org/10.1145/3340531.3412780

primary primitives we define are a combination of wrappers for existing toolkits, such as interfaces for running Anserini retrieval methods, and implementations of NIR models. Each may be used independently or in conjunction with other modules to form a pipeline.

By combining these modules with a configuration system that describes their operation, Capreolus provides modules with utility functions for determining when the output of a module's graph or subgraph can be safely cached and re-used.

For example, a first-stage retrieval model (Searcher module) depends on an Index module, and an Index module depends on a document Collection. To construct any Searcher (that is supported by the associated index type), the user simply instantiates a Searcher module while providing either an Index module or a document collection to be indexed. Neural reranking models are constructed similarly, though their dependency graphs are more complex as they also include additional modules, like a Trainer and an Extractor for transforming queries and documents into suitable representations. This approach fully retains the configurability and ease of use of the aforementioned pipelines while giving the user flexibility to compose modules in any desired way.

Importantly, this approach greatly increases the reproducibility and inspectability of experiments while substantially lowering the barrier to implementing experiments that do not easily fit into the aforementioned pipeline. Inspectability of the experimental pipeline is a natural consequence of the modules' decoupled nature: while the user may primarily interact directly with a Searcher or a Reranker module, they can also easily interact with their dependencies (the Index, Extractor, Tokenizer, Trainer, etc) to inspect their operation. Interacting with any module in the graph is straightforward, because notebooks are first-class citizens that can instantiate and use modules using the same APIs as any other experiment. This is difficult with toolkits that only support running the entire pipeline via a command line interface.

We rewrite Capreolus v0.1 [29], a rigid "search-then-rerank" pipeline, to follow this modular approach.[1] In addition, Capreolus v0.2 adds support for TensorFlow on both TPUs and GPUs, TF-Ranking integration, supports many more first-stage retrieval methods, and substantially increases the amount of automated testing. Capreolus v0.2 is available at https://capreolus.ai.

2 ARCHITECTURE

The Capreolus v0.2 architecture is based on the key idea that a modern IR pipeline can be broken down into loosely-coupled subgraphs consisting of tightly coupled nodes (modules). For example, consider the standard "search-then-rerank" pipeline implemented by both OpenNIR and Capreolus v0.1. The associated dependency graph in our approach is illustrated in Figure 1, with nodes representing modules and arrows representing dependencies. This reranking pipeline consists of two subgraphs: *(1)* a Rank Task in which a Searcher queries an inverted Index, which is built on top of a document Collection; and *(2)* a Rerank Task in which a supervised method reranks a set of candidate documents with the help of a Trainer and a feature Extractor, which in turn requires a Tokenizer and an inverted Index to access documents and term statistics. In this example, the Rerank Task (right) depends on the Rank Task (left) to identify a set of candidate documents to rerank. The Benchmark and Collection modules (middle) are shown separately to indicate that both Tasks depend on them to provide data, such as relevance judgments and a document collection.

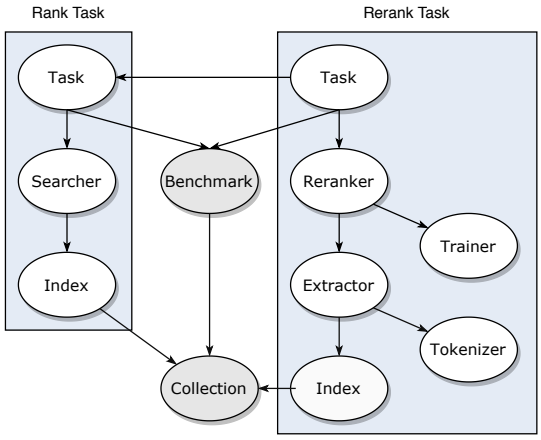

Figure 1: An example pipeline consisting of a Rerank Task that depends on a Rank Task to obtain results to rerank. Arrows indicate dependencies, which also define API access. For example, the Searcher can access both the Index's API and Collection's API, but neither the Index nor the Collection can access the Searcher. The Benchmark's and Collection's shading indicate they are shared by both Tasks.

Following this observation, our system's architecture consists of modules ("IR primitives") that declare dependencies on other modules.[2] For example, in Figure 1, Searcher depends on an Index (which depends on a Collection), and Reranker depends on a Trainer and Extractor. Dependencies may specify both a module type (e.g., Searcher) and a default module class (e.g., BM25), which can be overridden by the user via the configuration. In this example, the same shaded Benchmark and Collection models are used by both Tasks, whereas the user may configure other modules of the same type to be different (e.g., an Index with stemming to be used with the Searcher and an Index without stemming to be used with the Reranker). The Task modules primarily access the APIs of their dependencies to perform a task, such as ranking or reranking documents.

Both the input and output of the Rerank Task are a ranked list of documents, so it is clear how this design could be expanded to include a second reranking step (e.g., with a computationally expensive BERT model). A second, independently-configured Rerank Task would simply depend on the first Rerank Task and consume its output. This is already more flexibility than existing NIR toolkits provide. Similarly, a learning-to-rank (LTR) Task could be created that depends on a combination of (independently-configured) Rank or Rerank Tasks.

[1] https://mpi-inf.mpg.de/departments/databases-and-information-systems/research/neural-ir/capreolus

[2] Throughout this work, we use the term *module* to refer to *Capreolus* modules. These are distinct from *Python* modules, which are devices for organizing Python code.

2.1 Configuration

Capreolus provides a configuration system for choosing module classes to include in a pipeline (e.g., `searcher.name=BM25`) and for configuring each module (e.g., `searcher.b=0.5`). Module classes implementing a given module type are largely interchangeable[3] and every module class declares its own dependencies. While potentially verbose, a Capreolus configuration fully describes a pipeline (or experiment), including random seeds, preprocessing decisions, module classes, etc. In order to prevent this flexibility from becoming a burden to the user, reasonable defaults are automatically provided for any configuration options the user did not specify.

2.2 Design Goals and Module Requirements

We enumerate several design goals to formalize our statements that pipelines should be flexible, configurable, and inspectable:

(1) Pipelines should consist of self-contained modules that encapsulate basic functionality.
(2) Pipelines should be fully described by their configuration. Running the same pipeline with the same configuration should always yield the same result (to the degree possible when using GPUs[4]).
(3) Pipelines should cache their intermediate results, so that the same operations are not needlessly rerun.
(4) Capreolus should support the creation of new pipelines by arbitrarily combining modules together.

In order to satisfy these design goals, we impose some requirements on how modules are implemented. Succinctly, a module's state cannot change after the module has been created, and a module's methods should always return the same output given the same inputs. In more detail, the requirements are:

(1) A module's state should be a function of its configuration values and its dependencies.
(2) Methods of modules may additionally be functions of the method's arguments. Calling a method cannot modify a module's state.
(3) A module's state is independent of its parents' states, and a module cannot modify its children's states.

These requirements are partially enforced by the system, such as by making the configuration of modules immutable once they have been instantiated. The requirements limit the ways in which a module's state can change, which allows Capreolus to identify how outputs should be cached and when subgraphs can be safely re-used. For example, if `searcher.index` and `reranker.extractor.index` share the same configuration and Collection dependency, only one on-disk index will be built and used by both modules. This is transparently handled by the configuration system, which creates a cache path for the Index based on its configuration and dependency graph.

3 MODULES

Capreolus defines a variety of modules to interact with existing tools, to provide data needed for experiments, to implement NIR models, and to group modules together to perform a function. Modules are organized into *module types*, which define a minimal API a module class should provide. We later describe in Section 5 how Capreolus can be easily extended with new modules.

3.1 Module Types

The module types are listed in Table 1 in order of their complexity and typical dependencies. Collection and Benchmark modules primarily provide data for other modules to depend on and refer to (e.g., a document collection and qrels). Using modules to represent this data rather than directly passing paths around is unusual, but it serves two important purposes. First, including the data in the dependency graph simplifies modules' ability to cache and re-use intermediate outputs. Second, it allows the experimental data to be configured in the same way as other experimental parameters (e.g., configuring whether keyword or natural language queries are used). The Index module is responsible for building an inverted index on a Collection and providing access to the on-disk index. The Searcher module is responsible for querying an Index. Combined with a Task module (last row), which provides a pipeline entry point, these four module types are sufficient to implement a configurable ranking pipeline as shown in Figure 1 (left).

The remaining module types are primarily used with supervised reranking models, which Capreolus currently supports PyTorch and TensorFlow implementations of. The Tokenizer module tokenizes text. The Extractor module converts queries and documents into representations appropriate for a reranking model (e.g., using a WordPiece Tokenizer to prepare input for a BERT-based model). The Trainer module interacts with a Reranker, which is treated as a black box, to train the reranking model and make predictions. Finally, the Reranker module implements a reranking model, which depends on an appropriate Extractor and Trainer. As shown in Figure 1, a Task implementing a reranking pipeline interacts primarily with a Reranker module and a Benchmark module after identifying a list of documents to be reranked.

3.2 Module Classes

Examples of specific module classes of each type are shown in Table 2. Supported Collections and Benchmarks include `Robust04`,[6] `ANTIQUE` [10], `NFCorpus` [3], and the recent `TREC-COVID`[7] dataset. While the two module types are closely related, treating them as separate modules encodes the fact that Index modules are built only on a document collection and thus can be reused with different Benchmarks (e.g., with a different set of folds or queries). Capreolus primarily uses an Anserini Index with Anserini Searchers.

Tokenizer and Extractor modules are generally closely tied to each other and a specific Reranker. For example, `AnseriniTokenizer` is used by the `EmbedText` Extractor to preprocess documents before representing them as similarity matrices. The `PytorchTrainer` and `TensorFlowTrainer` classes allow Reranker modules to be implemented in either toolkit. Additionally, `TensorFlowTrainer` supports running models on a Google TPU[8] in addition to GPU and

[3] Modules are completely interchangeable from the perspective of the configuration system. Some configurations may not be runnable, however, such as if a Reranker's PyTorch Trainer is replaced with a TensorFlow Trainer.
[4] https://pytorch.org/docs/stable/notes/randomness.html
[6] https://trec.nist.gov/data/robust/04.guidelines.html
[7] https://ir.nist.gov/covidSubmit/
[8] https://cloud.google.com/tpu

Module Type	Purpose	Example API	Typical Dependencies
Collection	Specifies a document collection's location and format	• `get_path_and_types()`: returns a path to the collection and information about its format	none
Benchmark	Provides the data needed to run an experiment on a `Collection`, such as queries and judgments	• `download_if_missing()`: downloads any benchmark data that does not exist • `qrels`: relevance judgments • `topics`: queries • `folds`: topic splits indicating train, validation, and test sets • `relevance_level`: minimum label that indicates a relevant document (for non-graded metrics)	`Collection`
Index	Builds an inverted index on a `Collection`	• `create_index()`: creates an index on `Collection` if none exists • `get_doc(docid)`: returns a raw document • `get_df(term)`: returns a term's document frequency	`Collection`
Searcher	Searches an inverted index with an efficient retreival method	• `query(string)`: returns a ranked list of documents	`Index`
Tokenizer	Converts raw text into tokens for an NIR model	• `tokenize(string)`: returns a tokenized string	none
Extractor	Converts queries and documents into representations suitable for an NIR model	• `id2vec(query_docs)`: returns query-document representations (e.g., embedding similarity matrices)	`Index`, `Tokenizer`
Trainer	Trains an NIR reranking model and obtains predictions from it	• `train(reranker, data, output_path)`: trains the given `reranker` and saves its weights to `output_path` • `predict(reranker, data, output_path)`: predicts document relevance scores using the given `reranker` and saves the resulting ranked lists to `output_path`	none
Reranker	Re-ranks candidate documents with an NIR model	• `score(query_docs)`: predicts document relevance scores using the `Reranker`'s current weights • `save_weights(file)`: saves the `Reranker`'s current weights • `load_weights(file)`: loads weights into the `Reranker`	`Extractor`, `Trainer`
Task	Describes an experimental pipeline and runs experiments	• `train()` or `search()`: trains a `Reranker` or runs a `Searcher` • `evaluate()`: reports cross-validated effectiveness metrics • `print_config()`: displays the current pipeline's configuration • `print_pipeline()`: displays the current pipeline's structure • `list_modules()`: enumerates all known module types and classes	`Benchmark`, `Searcher`, `Reranker`

Table 1: Each module type, its purpose, an example of its API, and typical dependencies for the module type. API examples are intended to be illustrations and have been simplified to convey key ideas. See the documentation for a current reference.

CPU. Reranker modules provide model-specific configuration options.The `RankTask` and `RerankTask` are used to create and run experimental pipelines as illustrated in Figure 1. The `ReRerankTask` is a wrapper that runs a `RerankTask` followed by a second `RerankTask` on its output, demonstrating the flexibility of Capreolus pipelines. This task might be used, for example, to perform first-stage retrieval with BM25 to return 1,000 documents optimized for recall, followed by a second-stage KNRM reranker optimized for recall@100, and finally a final-stage BERT reranker that reranks KNRM's top 100 documents while optimizing for nDCG@20.

4 USAGE

Capreolus supports two primary use cases: running experimental pipelines via the command line interface or a notebook, and providing configurable interfaces to any of the "IR primitives" Capreolus implements as modules.

```
# install pip package (requirements: Java 11 and Python 3.7+)
$ pip install capreolus
# display information about a the 'rerank' pipeline
$ capreolus rerank.describe
# [output showing the pipeline and describing configuration options]
# run the reranking pipeline while specifying several options
$ capreolus rerank.traineval with reranker.name=KNRM
  ↪ rank.searcher.name=BM25
  ↪ benchmark.collection.path=/path/to/robust04
  ↪ reranker.extractor.embeddings=fasttext
  ↪ reranker.trainer.niters=3
# output includes metrics for the searcher and reranker
capreolus.task.rank.evaluate -            map: 0.2531
# ...
capreolus.task.rerank.evaluate -          map: 0.2435
```

Figure 2: Running a reranking pipeline via the CLI.

Type	Example Classes	Example Configuration Options
Collection	`Robust04`, `ANTIQUE`, `TREC-COVID`	• `round`: (`TREC-COVID`) round to include documents for
Benchmark	`Robust04`, `ANTIQUE`, `TREC-COVID`	• `round`: (`TREC-COVID`) round to use topics and qrels for • `udelqexpand`: (`TREC-COVID`) whether to use UDel's query generator [5] • `excludeknown`: (`TREC-COVID`) whether to omit judged documents from output
Index	`AnseriniIndex`	• `indexstop`: whether to keep stopwords • `stemmer`: stemmer type (e.g., porter, krovetz, none)
Searcher	`BM25`, `BM25RM3`, `DirichletQL`	• `hits`: number of documents to return for each query • `k1`: (`BM25`, `BM25RM3`) k1 parameter • `b`: (`BM25`, `BM25RM3`) b parameter • μ: (`DirichletQL`) μ parameter
Tokenizer	`AnseriniTokenizer`, `BertTokenizer`	• `keepstops`: (`AnseriniTokenizer`) whether stopwords should be kept • `pretrained`: (`BertTokenizer`) pretrained BERT vocabulary to load
Extractor	`EmbedText`, `BertText`	• `maxqlen`: maximum length for query • `maxdoclen`: maximum length for document • `embeddings`: (`EmbedText`) pretrained embeddings to use (e.g., `glove6b`)
Trainer	`PytorchTrainer`, `TensorFlowTrainer`	• `lr`: learning rate • `batch`: batch size • `niters`: number of iterations • `tpuname`: (`TensorFlowTrainer`) name of TPU device to use
Reranker	`DRMM` [8], `KNRM` [26], `PACRR` [11], `TFVanillaBert` [17]	• `nbins`: (`DRMM`) number of histogram bins • `pretrained`: (`TFVanillaBert`) pretrained BERT model to load
Task	`RankTask`, `RerankTask`, `ReRerankTask`	• `fold`: the train-validation-test data split • `seed`: the seed of all random generators used in the current experiment • `optimize`: metric used to identify the best run on the validation data

Table 2: Examples of module classes and example configuration options. Only a subset of those available are shown.

Figure 2 illustrates using the command line interface to train and evaluate a reranking pipeline, which corresponds to the graph shown in Figure 1. The configuration options indicate that the BM25 Searcher and the KNRM Reranker will be used with the default Robust04 Benchmark. The user provides a path to the Robust04 documents using the `benchmark.collection.path` option. Additionally, the Reranker is configured to use `fasttext` embeddings and to train for only three iterations. The command outputs training information and metrics for the Searcher and Reranker, which does not improve over the BM25 Searcher in this small example.

Figure 3 illustrates how a similar reranking pipeline can be created using Python in place of the command line interface. The `print_pipeline()` method displays the Task's full module graph, which corresponds to the graph shown in Figure 1. Calling `train()` and `eval()` is equivalent to the `traineval` command used in the previous example.

While the previous examples illustrated how Task modules can be used to run a full pipeline, Capreolus modules can also be used directly. Figure 4 demonstrates how individual modules can be used to create and query an index. After creating a Collection object, an Index is created on line 5 that receives the Collection as a dependency. When `create_index()` is called, the Index uses this Collection to locate ANTIQUE's documents on disk, which involves

```
>>> from capreolus.task.rerank import RerankTask
>>> config_string = "benchmark.name=antique rank.searcher.name=BM25
↪  reranker.name=KNRM reranker.trainer.niters=3"
>>> task = RerankTask(config_string)
>>> task.print_pipeline()
task=rerank
    benchmark=antique
        collection=antique
    task=rank
        benchmark=antique
            collection=antique
        searcher=BM25
            index=anserini
                collection=antique
    reranker=KNRM
        extractor=embedtext
            index=anserini
                collection=antique
            tokenizer=anserini
        trainer=pytorch
>>> task.train()
# ... ANTIQUE collection is downloaded and indexed
# ... the searcher produces a list of candidate docs
# ... the KNRM reranker is trained for 3 iterations
>>> task.evaluate()  # produce metrics
```

Figure 3: Running a reranking pipeline via Python.

```
>>> from capreolus.collection import ANTIQUE
>>> from capreolus.index import AnseriniIndex
>>> from capreolus.searcher import BM25
>>> collection = ANTIQUE()
>>> index = AnseriniIndex({"stemmer": "porter"},  {"collection": collection})
>>> index.create_index()
>>> collection.find_document_path()
'/home/ayates/.capreolus/cache/collection-antique/documents'
>>> index.get_df("retrieval")
0
>>> index.get_df("retriev")
155
>>> searcher = BM25({"b": 0.75, "hits": 3}, {"index": index})
>>> searcher.query("information retrieval")
OrderedDict([('746919_5', 8.655200004577637),
             ('1169146_0', 7.959799766540527),
             ('2011132_1', 7.939199924468994)])
>>> index.get_doc('746919_5')
'as intertainment, to retrieve information, and a writing tool.....and much more of course'
```

Figure 4: Indexing and searching a collection.

```
1  >>> from capreolus.searcher import Searcher
2  >>> from capreolus.task.rerank import RerankTask
3  >>> run = Searcher.load_trec_run("/path/to/existing/antique.run")
4  >>> task = RerankTask({"reranker": {"name": "DRMM", "nbins": 10, "trainer": {"niters": 1}}, "benchmark": {"name": "antique"}})
5  >>> results = task.rerank_run(run, "out_path", include_train=True)
6  >>> results.keys()
7  dict_keys(['dev', 'test', 'train'])
8  >>> Searcher.write_trec_run(results["dev"], "reranked_run.dev")
```

Figure 5: Reranking an existing set of results.

downloading ANTIQUE to the cache if a valid path cannot be found. Similarly, on line 13 a Searcher is created that receives the Index object as a dependency. Calling `query()` causes the Searcher to query this Index object and return the top 3 results. While it may seem counter-intuitive for `hits` to be one of the Searcher's configuration options rather than an argument to `query()`, Searcher modules are lightweight, and this approach is convenient because it allows the option to be easily configured by a pipeline. Lines 9, 11, and 18 illustrate how the Index can be interacted with directly to retrieve collection statistics or documents.

Similarly, Figure 5 demonstrates how the reranking pipeline can be used with an existing result set. After loading a TREC-format run from disk, `rerank_run()` is called to rerank these results using a DRMM NIR model trained for one iteration. The results returned are split into the validation (dev), test, and train queries in order to help the user ensure they match those used in the input run.

5 IMPLEMENTING NEW MODULES

New module classes must register themselves with Capreolus' module system and implement the API required by their module type. Implementations of a `VeryNew` Collection and Benchmark are shown in Figure 6. The Collection specifies a name to be used in the configuration system (e.g., `collection.name=verynew`), the necessary Anserini collection and generator types, and a configuration option allowing the user to specify a path. The Benchmark

```
@Collection.register
class VeryNewCollection(Collection):
    module_name = "verynew"
    config_spec = [ConfigOption("path", "/default", "document path")]
    collection_type = "TrecCollection"
    generator_type = "DefaultLuceneDocumentGenerator"

@Benchmark.register
class VeryNewBenchmark(Benchmark):
    module_name = "verynew"
    dependencies = [Dependency(key="collection", module="collection", name="verynew")]
    qrel_file = PACKAGE_PATH / "data" / "qrels.verynew.txt"
    topic_file = PACKAGE_PATH / "data" / "topics.verynew.txt"
    fold_file = PACKAGE_PATH / "data" / "verynew_folds.json"
```

Figure 6: Implementing Collection & Benchmark modules.

```
@Reranker.register
class NewModel(Reranker):
    module_name = "newmodel"
    config_spec = [ConfigOption("finetune", False, "train the embedding layer")]
    dependencies = [Dependency("extractor", module="extractor", name="embedtext"), Dependency(key="trainer", module="trainer", name="pytorch")]

    def build_model(self):
        if not hasattr(self, "model"):
            self.model = NewModel(self.extractor, self.config)
        return self.model

    def score(self, d):
        return [self.model(d["posdoc"], d["query"]).view(-1), self.model(d["negdoc"], d["query"]).view(-1)]

    def test(self, d):
        return self.model(d["posdoc"], d["query"]).view(-1)

class NewModel(pytorch.nn.Module)
  def __init__(self, extractor, config):
        self.embedding = create_emb_layer(extractor.embeddings, non_trainable=config["finetune"])

  def forward(docidxs, queryidxs):
      doc = self.get_embedding(docidxs)
      query = self.get_embedding(queryidxs)
      # ... neural model using the doc and query term embeddings ...
      return query_doc_scores
```

Figure 7: Implementing a new Reranker module.

declares a dependency on the new Collection we declared and specifies paths to data files. Both modules can optionally implement a `download_if_missing()` method. Once registered, the modules can be used from Python or via the command line interface.

Figure 7 implements a new Reranker module. As before, the module declares its name, configuration options, and dependencies, which include `PytorchTrainer` and an Extractor that converts tokens to term embeddings. The module provides `score()` and `test()` methods that receive query-document features produced by the Extractor.

Device	Batch Size	Time (seconds) Average	Std Dev
GPU Quadro 8000 (2018)	2	200.6	2.42
GPU Quadro 8000 (2018)	18 (max)	1570.6	10.37
GPU Tesla K80 (2014)	2 (max)	1333.0	5.25
GPU Tesla M40 (2015)	2 (max)	506.4	1.95
GPU Tesla V100 (2017)	2 (max)	199.6	0.48
GPU Titan Xp (2016)	2 (max)	287.6	1.20
TPU v2-8 (2017)	2	223.2	3.00
TPU v2-8 (2017)	64 (max)	17.6	0.01
TPU v3-8 (2018)	2	166.9	0.71
TPU v3-8 (2018)	128 (max)	9.1	0.01

Table 3: Average run time taken to train 2048 instances with TensorFlow on different GPU and TPU architectures.

6 EFFICIENCY

While training neural reranking models is inherently computationally expensive, Capreolus uses several strategies to mitigate this to the extent possible. First, representing pipelines as dependency graphs allows many intermediate outputs to be cached and reused, which limits repetitive computations. Second, making all aspects of a pipeline configurable allows many different experiments to be run in parallel across any machines available, which can include experiments with different folds, first-stage ranking models, neural reranking models, hyperparameters, etc. For example, a shell script can loop over different configurations and send each to a workload manager like Slurm. Finally, providing a `Trainer` module compatible with TensorFlow allows Capreolus to run computationally expensive models on Google Cloud TPUs. All `Reranker` modules using `TensorFlowTrainer` automatically support both GPU and TPU devices.

To estimate how training time varies across devices for a BERT model on a common collection, we conducted an experiment training VanillaBERT [17] on various TPUs and GPUs. Specifically, we initialized `TFVanillaBert` with the pre-trained BERT-base model, which consists of 12 Transformer encoder layers and 12 attention heads, and measured the time required to train (fine-tune) the model on training instances consisting of a query, a relevant document, and a non-relevant document. Instances were taken from the TREC Robust 2004 dataset, with the maximum query length set to 8 tokens and documents truncated to 500 tokens. The model was trained with pairwise hinge loss for three iterations consisting of 2048 training instances each. We ran the model five times on each device and report the average amount of time taken for the second training iteration. Additionally, we measured performance both with a batch size of two and with the maximum batch size that could fit in a device's RAM. This is the maximum for GPUs with 16GB RAM, but the TPUs and Quadro 8000 can support larger batch sizes.

As show in Table 3, both TPU devices are faster than most GPU devices that we tested on regardless of batch size. The Tesla GPUs become substantially faster between 2014 (K80) and 2017 (V100), with the training time decreasing from approximately 22 to 3 minutes. Only the most powerful GPU (Tesla V100) was able to finish training faster than the v2 TPU. TPUs are more fully utilized with higher batch sizes, so v3 TPUs are able to process the 2048 training instances in under 10 seconds when using the maximum batch size. We note that the TPUs and GPUs are fundamentally different, and this benchmark compares expected training times rather than fundamental properties of the devices themselves. That is, the TPUs automatically shared each batch across their 8 cores, so this benchmark is comparing one GPU to many TPU cores. TPUs additionally receive their training instances directly from a Google Cloud Storage bucket. While Capreolus automatically converts the data to the appropriate binary format and uploads it, this preprocessing can be time consuming.

7 RELATED WORK

The information retrieval community has a long history of providing open source toolkits that facilitate performing experiments, such as the Anserini, Galago, Indri, PISA, and Terrier platforms [5, 14, 16, 23, 27]. At a minimum, these toolkits support building an inverted index and retrieving documents from it using an efficient retrieval method (e.g., QL, BM25). Toolkits like Anserini [27] additionally aim to provide reproducible baselines on commonly used document collections.

Toolkits to support reranking methods using learning to rank, such as FastRank, RankLib, SVMRank, and Terrier [4, 7, 14, 24], are also available. This reranking is typically applied to results retrieved from a first-stage ranking method. With the exception of Terrier, these toolkits are independent of the inverted index and first-stage retrieval method, and the user is responsible for integrating the two in an appropriate way.

With the growing popularity of neural reranking (NIR) models, several toolkits such as MatchZoo [9] and OpenNIR [12] have recently provided implementations of popular NIR models. MatchZoo focuses on TensorFlow and PyTorch (MatchZoo-py) implementations of neural reranking models, with experimental considerations like cross-validation and earlier parts of the pipeline left up to the user. OpenNIR handles both and defines a rigid pipeline similar to the "search-then-rerank" pipeline implemented by Capreolus v0.1 [29]. This pipeline orchestrates Ranker, Dataset, Trainer and Predictor modules to obtain first-stage retrieval results and then to rerank them. MatchZoo provides separate TensorFlow and PyTorch codebases, while OpenNIR supports PyTorch.

Rather than implementing NIR models or a full pipeline, the TF-Ranking library [20] provides functions to facilitate implementing new models in TensorFlow. For example, it provides optimized pairwise and listwise loss functions that can be leveraged by TensorFlow models. The loss functions provided by TF-Ranking are exposed as a configuration option in Capreolus' `TensorFlowTrainer`, allowing users to leverage these optimized implementations.

All the aforementioned NIR toolkits (i.e., MatchZoo, OpenNIR, Capreolus v0.1, and TF-Ranking) are written in Python, while first-stage ranking methods are usually implemented by toolkits written in Java or C++. For example, both Capreolus and OpenNIR uses Anserini [27], which is written in Java, for the search stage of the "search-then-rerank" pipeline. Anserini simplifies this integration by providing Pyserini [1], which is a wrapper for much of Anserini's functionality. Similarly, Pyndri provides a Python interface to the Indri toolkit [25]. In addition to providing an interface for the Terrier toolkit, the PyTerrier toolkit [15] allows pipelines to be described

using Python operators. We note that PyTerrier is a contemporaneous approach that shares some of the same aims as our work. TREC Tools is a utility for helping researchers run "TREC-like campaigns" [19]. It provides a Python interface for querying with first-stage retrieval methods (e.g., using Indri, Terrier, or PISA) as well as methods for manipulating results (i.e., creating assessment pools and performing rank fusion) and evaluating them.

Though we have left it for a future release, our Capreolus modules allow such wrappers to be used to integrate with other IR toolkits, such as with Indri (through Pyndri) or with Terrier (through PyTerrier). Rather than replacing these toolkits, we aim to provide a configurable platform for conducting IR experiments using a variety of toolkits, and encourage leveraging existing software where possible. Moreover, we provide PyTorch implementations of prominent NIR models and some TensorFlow implementations to take advantage of TPUs.

Previous efforts to examine the validity and reproducibility of gains achieved by NIR models have found that pre-BERT NIR models are often outperformed by strong baselines [28]. We argue that our pipeline composed of IR primitives helps address this issue by making it easier to reproduce and to build upon existing work. For example, it is straightforward for the user to create a pipeline that extracts query-document similarity scores from a Transformer-based model (e.g., BERT [6]) and another similarity score from an n-gram based convolutional model (e.g., PACRR [11]), which are then input as auxiliary signals to a third model.

8 CONCLUSION AND FUTURE WORK

We described how Capreolus v0.2 can be used to construct flexible IR pipelines by providing "IR primitives" as configurable modules. While the flexible pipeline is the most prominent change introduced in Capreolus v0.2, we have also improved the general usability of the toolkit. This includes adding a `TensorFlowTrainer` module to enable training TensorFlow models on GPUs and Google TPUs, integrating with the TF-Ranking library to leverage the optimized loss functions that it provides, and extending our Anserini integration to include all of Anserini's first-stage ranking methods.

As future work, we hope to support additional first-stage ranking methods (e.g., via Terrier integration), to support non-neural learning-to-rank methods (e.g., FastRank), and to provide tools for directly measuring a method's efficiency (e.g., resource consumption and run time). We additionally plan to incorporate more Collections and Benchmarks for supporting the datasets that are already integrated in Anserini.

9 ACKNOWLEDGMENTS

This work was supported in part by Google Cloud, the TensorFlow Research Cloud, and the Natural Sciences and Engineering Research Council (NSERC) of Canada. We thank Siddhant Arora and Sean Macavaney for code contributions.

REFERENCES

[1] Zeynep Akkalyoncu Yilmaz, Charles L. A. Clarke, and Jimmy Lin. 2020. A Lightweight Environment for Learning Experimental IR Research Practices. In *SIGIR*.
[2] Zeynep Akkalyoncu Yilmaz, Wei Yang, Haotian Zhang, and Jimmy Lin. 2019. Cross-Domain Modeling of Sentence-Level Evidence for Document Retrieval. In *EMNLP*.
[3] Vera Boteva, Demian Gholipour, Artem Sokolov, and Stefan Riezler. 2016. A Full-Text Learning to Rank Dataset for Medical Information Retrieval. In *ECIR*.
[4] Yunbo Cao, Jun Xu, Tie-Yan Liu, Hang Li, Yalou Huang, and Hsiao-Wuen Hon. 2006. Adapting Ranking SVM to Document Retrieval. In *SIGIR*.
[5] W. Croft, D. Metzler, and T. Strohman. 2009. *Search Engines: Information Retrieval in Practice*. Pearson, New York, NY, USA.
[6] Zhuyun Dai and Jamie Callan. 2019. Deeper Text Understanding for IR with Contextual Neural Language Modeling. In *SIGIR*.
[7] John Foley. 2019. FastRank alpha release. https://jjfoley.me/2019/10/11/fastrank-alpha.html.
[8] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W. Bruce Croft. 2016. A Deep Relevance Matching Model for Ad-hoc Retrieval. In *CIKM*.
[9] Jiafeng Guo, Yixing Fan, Xiang Ji, and Xueqi Cheng. 2019. MatchZoo: A Learning, Practicing, and Developing System for Neural Text Matching. In *SIGIR*.
[10] Helia Hashemi, Mohammad Aliannejadi, Hamed Zamani, and W. Bruce Croft. 2019. ANTIQUE: A Non-Factoid Question Answering Benchmark. *arXiv:1905.08957* (2019).
[11] Kai Hui, Andrew Yates, Klaus Berberich, and Gerard de Melo. 2017. PACRR: A Position-Aware Neural IR Model for Relevance Matching. In *EMNLP*.
[12] Sean MacAvaney. 2020. OpenNIR: A Complete Neural Ad-Hoc Ranking Pipeline. In *WSDM*.
[13] Sean MacAvaney, Andrew Yates, Arman Cohan, and Nazli Goharian. 2019. CEDR: Contextualized Embeddings for Document Ranking. In *SIGIR*.
[14] Craig Macdonald, Richard McCreadie, Rodrygo L.T. Santos, and Iadh Ounis. 2012. From puppy to maturity: Experiences in developing Terrier. *Proc. of OSIR at SIGIR* (2012), 60–63.
[15] Craig Macdonald and Nicola Tonellotto. 2020. Declarative Experimentation in Information Retrieval using PyTerrier. In *ICTIR*.
[16] Antonio Mallia, Michal Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia. In *Proc. of the Open-Source IR Replicability Challenge at SIGIR*.
[17] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. *arXiv:1901.04085* (2019).
[18] Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. 2020. Document ranking with a pretrained sequence-to-sequence model. *arXiv:2003.06713* (2020).
[19] Joao Palotti, Harrisen Scells, and Guido Zuccon. 2019. TrecTools: an open-source Python library for Information Retrieval practitioners involved in TREC-like campaigns. In *SIGIR*.
[20] Rama Kumar Pasumarthi, Sebastian Bruch, Xuanhui Wang, Cheng Li, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, and Stephan Wolf. 2019. TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank. In *KDD*.
[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 8024–8035.
[22] Corby Rosset, Damien Jose, Gargi Ghosh, Bhaskar Mitra, and Saurabh Tiwary. 2018. Optimizing Query Evaluations Using Reinforcement Learning for Web Search. In *SIGIR*.
[23] Trevor Strohman, Donald Metzler, Howard Turtle, and W. Bruce Croft. 2005. Indri: A Language Model-Based Search Engine for Complex Queries. In *Proceedings of the International Conference on Intelligent Analysis*.
[24] Van Dang. 2019. Ranklib. https://sourceforge.net/p/lemur/wiki/RankLib/.
[25] Christophe Van Gysel, Evangelos Kanoulas, and Maarten de Rijke. 2017. Pyndri: a Python Interface to the Indri Search Engine. In *ECIR*.
[26] Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. 2017. End-to-End Neural Ad-hoc Ranking with Kernel Pooling. In *SIGIR*.
[27] Peilin Yang, Hui Fang, and Jimmy Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. *J. Data and Information Quality* 10, 4 (2018), Article 15.
[28] Wei Yang, Kuang Lu, Peilin Yang, and Jimmy Lin. 2019. Critically Examining the "Neural Hype": Weak Baselines and the Additivity of Effectiveness Gains from Neural Ranking Models. In *SIGIR*.
[29] Andrew Yates, Siddhant Arora, Xinyu Zhang, Wei Yang, Kevin Martin Jose, and Jimmy Lin. 2020. Capreolus: A Toolkit for End-to-End Neural Ad Hoc Retrieval. In *WSDM*.