PyTerrier Documentation

Release 0.8.0-alpha

Contributors to PyTerrier

Jan 14, 2022

GUIDES

CHAPTER

ONE

INSTALLING AND CONFIGURING

PyTerrier is a declarative platform for information retrieval experiemnts in Python. It uses the Java-based Terrier information retrieval platform internally to support indexing and retrieval operations.

1.1 Pre-requisites

PyTerrier requires Python 3.6 or newer, and Java 11 or newer.

PyTerrier is natively supported on Linux and Mac OS X. PyTerrier uses Pytrec_eval for evaluation, and the latter does not install automatically on Windows.

1.2 Installation

Installing PyTerrier is easy - it can be installed from the command-line in the normal way using Pip:

```
pip install python-terrier
```

If you want the latest version of PyTerrier, you can install direct from the Github repo:

```
pip install --upgrade git+https://github.com/terrier-org/pyterrier.git#egg=python-terrier
```

NB: There is no need to have a local installation of the Java component, Terrier. PyTerrier will download the latest release on startup.

1.3 Configuration

You must always start by importing PyTerrier and running init():

```
import pyterrier as pt
pt.init()
```

PyTerrier uses PyJnius as a "glue" layer in order to call Terrier's Java classes. PyJnius will search the usual places on your machine for a Java installation. If you have problems, set the *JAVA_HOME* environment variable:

```
import os
os.environ["JAVA_HOME"] = "/path/to/my/jdk"
import pyterrier as pt
pt.init()
```

pt.init() has a multitude of options, for instance that can make PyTerrier more notebook friendly, or to change the underlying version of Terrier, as described below.

1.4 API Reference

All usages of PyTerrier start by importing PyTerrier and starting it using the *init()* method:

```
import pyterrier as pt
pt.init()
```

PyTerrier uses some of the functionality of the Java-based Terrier IR platform for indexing and retrieval functionality. Calling *pt.init()* downloads, if necessary, the Terrier jar file, and starts the Java Virtual Machine (JVM). It also configures the Terrier so that it can be more easily used from Python, such as redirecting the stdout and stderr streams, logging level etc.

Below, there is more documentation about method related to starting Terrier using PyTerrier, and ways to change the configuration.

1.4.1 Startup-related methods

`pyterrier.init()`

Function necessary to be called before Terrier classes and methods can be used. Loads the Terrier .jar file and imports classes. Also finds the correct version of Terrier to download if no version is specified.

Parameters

- **version** (*str*) – Which version of Terrier to download. Default is *None*.
  - If None, find the newest Terrier released version in MavenCentral and download it.
  - If *"snapshot"*, will download the latest build from Jitpack.
- **mem** (*str*) – Maximum memory allocated for the Java virtual machine heap in MB. Corresponds to java *-Xmx* commandline argument. Default is 1/4 of physical memory.
- **boot_packages** (*list(str)*) – Extra maven package coordinates files to load before starting Java. Default=`[]`. There is more information about loading packages in the Terrier documentation
- **packages** (*list(str)*) – Extra maven package coordinates files to load, using the Terrier classloader. Default=`[]`. See also *boot_packages* above.
- **jvm_opts** (*list(str)*) – Extra options to pass to the JVM. Default=`[]`. For instance, you may enable Java assertions by setting *jvm_opts=['-ea']*
- **redirect_io** (*boolean*) – If True, the Java *System.out* and *System.err* will be redirected to Pythons sys.out and sys.err. Default=True.
- **logging** (*str*) – the logging level to use:
  - Can be one of *'INFO'*, *'DEBUG'*, *'TRACE'*, *'WARN'*, *'ERROR'*. The latter is the quietest.
  - Default is *'WARN'*.
- **home_dir** (*str*) – the home directory to use. Default to PYTERRIER_HOME environment variable.
- **tqdm** – The tqdm instance to use for progress bars within PyTerrier. Defaults to tqdm.tqdm. Available options are *'tqdm'*, *'auto'* or *'notebook'*.

- **helper_version** (*str*) – Which version of the helper.

Locating the Terrier .jar file: PyTerrier is not tied to a specific version of Terrier and will automatically locate and download a recent Terrier .jar file. However, inevitably, some functionalities will require more recent Terrier versions.

- If set, PyTerrier uses the *version* init kwarg to determine the .jar file to look for.
- If the *version* init kwarg is not set, Terrier will query MavenCentral to determine the latest Terrier release.
- If *version* is set to *"snapshot"*, the latest .jar file build derived from the Terrier Github repository will be downloaded from Jitpack.
- Otherwise the local (*~/.mvn*) and MavenCentral repositories are searched for the jar file at the given version.

In this way, the default setting is to download the latest release of Terrier from MavenCentral. The user is also able to use a locally installed copy in their private Maven repository, or track the latest build of Terrier from Jitpack.

If you wish to run PyTerrier in an offline enviroment, you should ensure that the "terrier-assemblies-{your version}-jar-with-dependencies.jar" and "terrier-python-helper-{your helper version}.jar" are in the "~/.pyterrier" (if they are not present, they will be downloaded the first time). Then you should set their versions when calling `init()` function. For example: `pt.init(version = 5.5, helper_version = "0.0.6")`.

`pyterrier.`**`started`**`()`

Returns *True* if *init()* has already been called, false otherwise. Typical usage:

```
import pyterrier as pt
if not pt.started():
    pt.init()
```

`pyterrier.`**`run`**`()`

Allows to run a Terrier executable class, i.e. one that can be access from the *bin/terrier* commandline programme.

1.4.2 Methods to change PyTerrier configuration

`pyterrier.`**`extend_classpath`**`()`

Allows to add packages to Terrier's classpath after the JVM has started.

`pyterrier.`**`logging`**`()`

Set the logging level. Equivalent to setting the logging= parameter to init(). The following string values are allowed, corresponding to Java logging levels:

- *'ERROR'*: only show error messages
- *'WARN'*: only show warnings and error messages (default)
- *'INFO'*: show information, warnings and error messages
- *'DEBUG'*: show debugging, information, warnings and error messages

`pyterrier.`**`redirect_stdouterr`**`()`

Ensure that stdout and stderr have been redirected. Equivalent to setting the redirect_io parameter to init() as *True*.

`pyterrier.`**`set_property`**`()`

Allows to set a property in Terrier's global properties configuration. Example:

```
pt.set_property("termpipelines", "")
```

While Terrier has a variety of properties – as discussed in its indexing and retrieval configuration guides – in PyTerrier, we aim to expose Terrier configuration through appropriate methods or arguments. So this method should be seen as a safety-valve - a way to override the Terrier configuration not explicitly supported by PyTerrier.

`pyterrier.`**`set_properties`**`()`

Allows to set many properties in Terrier's global properties configuration

`pyterrier.`**`set_tqdm`**`()`

Set the tqdm progress bar type that Pyterrier will use internally. Many PyTerrier transformations can be expensive to apply in some settings - users can view progress by using the verbose=True kwarg to many classes, such as BatchRetrieve.

The tqdm progress bar can be made prettier when using appropriately configured Jupyter notebook setups. We use this automatically when Google Colab is detected.

Allowable options for type are:

- *'tqdm'*: corresponds to the standard text progresss bar, ala *from tqdm import tqdm*.
- *'notebook'*: corresponds to a notebook progress bar, ala *from tqdm.notebook import tqdm*
- *'auto'*: allows tqdm to decide on the progress bar type, ala *from tqdm.auto import tqdm*. Note that this works fine on Google Colab, but not on Jupyter unless the ipywidgets have been installed.

CHAPTER

TWO

IMPORTING DATASETS

The datasets module allows easy access to existing standard test collections, particulary those from TREC. In particular, each defined dataset can download and provide easy access to:

- files containing the documents of the corpus
- topics (queries), as a dataframe, ready for retrieval
- relevance assessments (aka, labels or qrels), as a dataframe, ready for evaluation
- ready-made Terrier indices, where appropriate

`pyterrier.datasets.`**`list_datasets`**`()`

Returns a dataframe of all datasets, listing which topics, qrels, corpus files or indices are available. By default, filters to only datasets with both a corpus and topics in English.

`pyterrier.datasets.`**`find_datasets`**`()`

A grep-like method to help identify datasets. Filters the output of list_datasets() based on the name containing the query

`pyterrier.datasets.`**`get_dataset`**`()`

Get a dataset by name

`class pyterrier.datasets.`**`Dataset`**

Represents a dataset (test collection) for indexing or retrieval. A common use-case is to use the Dataset within an Experiment:

```
dataset = pt.get_dataset("trec-robust-2004")
pt.Experiment([br1, br2], dataset.get_topics(), dataset.get_qrels(), eval_metrics=[
↪"map", "recip_rank"])
```

`get_corpus``()`

Returns the location of the files to allow indexing the corpus, i.e. it returns a list of filenames.

`get_corpus_iter``(`*verbose=True*`)`

Returns an iter of dicts for this collection. If verbose=True, a tqdm pbar shows the progress over this iterator.

Return type `Iterator[Dict[str, Any]]`

`get_corpus_lang``()`

Returns the ISO 639-1 language code for the corpus, or None for multiple/other/unknown

Return type `Optional[str]`

`get_index``(`*variant=None*, ***kwargs*`)`

Returns the IndexRef of the index to allow retrieval. Only a few datasets provide indices ready made.

`get_topics``(`*variant=None*`)`

Returns the topics, as a dataframe, ready for retrieval.

Return type `DataFrame`

get_topics_lang()

Returns the ISO 639-1 language code for the topics, or None for multiple/other/unknown

Return type `Optional[str]`

get_qrels(*variant=None*)

Returns the qrels, as a dataframe, ready for evaluation.

Return type `DataFrame`

get_topicsqrels(*variant=None*)

Returns both the topics and qrels in a tuple. This is useful for pt.Experiment().

Return type `Tuple[DataFrame, DataFrame]`

info_url()

Returns a url that provides more information about this dataset.

get_results(*variant=None*)

Returns a standard result set provided by the dataset. This is useful for re-ranking experiments.

Return type `DataFrame`

2.1 Examples

Many of the PyTerrier unit tests are based on the Vaswani NPL test collection, a corpus of scientific abstract from ~11,000 documents. PyTerrier provides a ready-made index on the Terrier Data Repository. This allows experiments to be easily conducted:

```
dataset = pt.get_dataset("vaswani")
bm25 = pt.BatchRetrieve.from_dataset(dataset, "terrier_stemmed", wmodel="BM25")
dph = pt.BatchRetrieve.from_dataset(dataset, "terrier_stemmed", wmodel="DPH")
pt.Experiment(
    [bm25, dph],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map"]
)
```

Indexing and then retrieval of documents from the MSMARCO document corpus can be achieved as follows:

```
dataset = pt.get_dataset("trec-deep-learning-docs")
indexer = pt.TRECCollectionIndexer("./index")
# this downloads the file msmarco-docs.trec.gz
indexref = indexer.index(dataset.get_corpus())
index = pt.IndexFactory.of(indexref)

DPH_br = pt.BatchRetrieve(index, wmodel="DPH") % 100
BM25_br = pt.BatchRetrieve(index, wmodel="BM25") % 100
# this runs an experiment to obtain results on the TREC 2019 Deep Learning track queries␣
↪and qrels
pt.Experiment(
    [DPH_br, BM25_br],
    dataset.get_topics("test"),
```

(continues on next page)

(continued from previous page)

```
    dataset.get_qrels("test"),
    eval_metrics=["recip_rank", "ndcg_cut_10", "map"])
```

For more details on use of MSMARCO, see our MSMARCO leaderboard submission notebooks.

You can also index datasets that include a corpus using IterDictIndexer and get_corpus_iter:

```
dataset = pt.datasets.get_dataset('irds:cord19/trec-covid')
indexer = pt.index.IterDictIndexer('./cord19-index')
indexref = indexer.index(dataset.get_corpus_iter(), fields=('title', 'abstract'))
index = pt.IndexFactory.of(indexref)

DPH_br = pt.BatchRetrieve(index, wmodel="DPH") % 100
BM25_br = pt.BatchRetrieve(index, wmodel="BM25") % 100
# this runs an experiment to obtain results on the TREC COVID queries and qrels
pt.Experiment(
    [DPH_br, BM25_br],
    dataset.get_topics('title'),
    dataset.get_qrels(),
    eval_metrics=["P.5", "P.10", "ndcg_cut.10", "map"])
```

2.2 Available Datasets

The table below lists the provided datasets, detailing the attributes available for each dataset. In each column, True designates the presence of a single artefact of that type, while a list denotes the available variants. Datasets with the `irds:` prefix are from the ir_datasets package; further documentation on these datasets can be found here.

dataset	corpus	index	topics
50pct		[‘ex1’, ‘ex2’]	[training, validation]
antique	True		[train, test]
vaswani	True	True	True
msmarco_document	True	True	[train, dev, test, test-2020, leaderboard
msmarcov2_document		True	[train, dev1, dev2, valid1, valid2, trec_
msmarco_passage	True	True	[train, dev, dev.small, eval, eval.small,
msmarcov2_passage		True	[train, dev1, dev2, trec_2021]
trec-robust-2004			True
trec-robust-2005			True
trec-terabyte			[2004, 2005, 2006, 2004-2006, 2006-r
trec-precision-medicine			[2017, 2018, 2019, 2020]
trec-covid	[round4, round5]	True	[round1, round2, round3, round4, rou
trec-wt2g			True
trec-wt10g			[trec9, trec10-adhoc, trec10-hp]
trec-wt-2002			[td, np]
trec-wt-2003			[td, np]
trec-wt-2004			[all, np, hp, td]
trec-wt-2009			True
trec-wt-2010			True
trec-wt-2011			True
trec-wt-2012			True

dataset	corpus	index	topics
irds:antique	True		
irds:antique/test	True		True
irds:antique/train	True		True
irds:aquaint	True		
irds:aquaint/trec-robust-2005	True		[title, description, narrative]
irds:argsme			
irds:argsme/1.0	True		
irds:argsme/1.0-cleaned	True		
irds:argsme/2020-04-01/debateorg	True		
irds:argsme/2020-04-01/debatepedia	True		
irds:argsme/2020-04-01/debatewise	True		
irds:argsme/2020-04-01/idebate	True		
irds:argsme/2020-04-01/parliamentary	True		
irds:argsme/2020-04-01	True		
irds:beir			
irds:beir/arguana	True		True
irds:beir/climate-fever	True		True
irds:beir/cqadupstack/android	True		[text, tags]
irds:beir/cqadupstack/english	True		[text, tags]
irds:beir/cqadupstack/gaming	True		[text, tags]
irds:beir/cqadupstack/gis	True		[text, tags]
irds:beir/cqadupstack/mathematica	True		[text, tags]
irds:beir/cqadupstack/physics	True		[text, tags]
irds:beir/cqadupstack/programmers	True		[text, tags]
irds:beir/cqadupstack/stats	True		[text, tags]
irds:beir/cqadupstack/tex	True		[text, tags]
irds:beir/cqadupstack/unix	True		[text, tags]
irds:beir/cqadupstack/webmasters	True		[text, tags]
irds:beir/cqadupstack/wordpress	True		[text, tags]
irds:beir/dbpedia-entity	True		True
irds:beir/fever	True		True
irds:beir/fiqa	True		True
irds:beir/hotpotqa	True		True
irds:beir/msmarco	True		True
irds:beir/nfcorpus	True		[text, url]
irds:beir/nq	True		True
irds:beir/quora	True		True
irds:beir/scidocs	True		[text, authors, year, cited_by, reference
irds:beir/scifact	True		True
irds:beir/trec-covid	True		[text, query, narrative]
irds:beir/webis-touche2020	True		[text, description, narrative]
irds:beir/webis-touche2020/v2	True		[text, description, narrative]
irds:c4			
irds:c4/en-noclean-tr	True		
irds:c4/en-noclean-tr/trec-misinfo-2021	True		[text, description, narrative, disclaime
irds:car			
irds:car/v1.5	True		
irds:car/v1.5/test200	True		[text, title, headings]
irds:car/v1.5/train/fold0	True		[text, title, headings]

dataset	corpus	index	topics
irds:car/v1.5/train/fold1	True		[text, title, headings]
irds:car/v1.5/train/fold2	True		[text, title, headings]
irds:car/v1.5/train/fold3	True		[text, title, headings]
irds:car/v1.5/train/fold4	True		[text, title, headings]
irds:car/v1.5/trec-y1	True		[text, title, headings]
irds:car/v1.5/trec-y1/auto	True		[text, title, headings]
irds:car/v1.5/trec-y1/manual	True		[text, title, headings]
irds:highwire	True		
irds:highwire/trec-genomics-2006	True		True
irds:highwire/trec-genomics-2007	True		True
irds:medline			
irds:medline/2004	True		
irds:medline/2004/trec-genomics-2004	True		[title, need, context]
irds:medline/2004/trec-genomics-2005	True		True
irds:medline/2017	True		
irds:medline/2017/trec-pm-2017	True		[disease, gene, demographic, other]
irds:medline/2017/trec-pm-2018	True		[disease, gene, demographic]
irds:clinicaltrials			
irds:clinicaltrials/2017	True		
irds:clinicaltrials/2017/trec-pm-2017	True		[disease, gene, demographic, other]
irds:clinicaltrials/2017/trec-pm-2018	True		[disease, gene, demographic]
irds:clinicaltrials/2019	True		
irds:clinicaltrials/2019/trec-pm-2019	True		[disease, gene, demographic]
irds:clinicaltrials/2021	True		
irds:clinicaltrials/2021/trec-ct-2021	True		True
irds:clirmatrix			
irds:clueweb09/catb	True		
irds:clueweb09/catb/trec-web-2009	True		[query, description, type, subtopics]
irds:clueweb09/catb/trec-web-2010	True		[query, description, type, subtopics]
irds:clueweb09/catb/trec-web-2011	True		[query, description, type, subtopics]
irds:clueweb09/catb/trec-web-2012	True		[query, description, type, subtopics]
irds:clueweb09/en	True		
irds:clueweb09/en/trec-web-2009	True		[query, description, type, subtopics]
irds:clueweb09/en/trec-web-2010	True		[query, description, type, subtopics]
irds:clueweb09/en/trec-web-2011	True		[query, description, type, subtopics]
irds:clueweb09/en/trec-web-2012	True		[query, description, type, subtopics]
irds:clueweb12	True		
irds:clueweb12/b13	True		
irds:clueweb12/b13/clef-ehealth	True		True
irds:clueweb12/b13/ntcir-www-1	True		True
irds:clueweb12/b13/ntcir-www-2	True		[title, description]
irds:clueweb12/b13/ntcir-www-3	True		[title, description]
irds:clueweb12/b13/trec-misinfo-2019	True		[title, cochranedoi, description, narrat
irds:clueweb12/trec-web-2013	True		[query, description, type, subtopics]
irds:clueweb12/trec-web-2014	True		[query, description, type, subtopics]
irds:cord19	True		
irds:cord19/fulltext	True		
irds:cord19/fulltext/trec-covid	True		[title, description, narrative]
irds:cord19/trec-covid	True		[title, description, narrative]

dataset	corpus	index	topics
irds:cord19/trec-covid/round1	True		[title, description, narrative]
irds:cord19/trec-covid/round2	True		[title, description, narrative]
irds:cord19/trec-covid/round3	True		[title, description, narrative]
irds:cord19/trec-covid/round4	True		[title, description, narrative]
irds:cord19/trec-covid/round5	True		[title, description, narrative]
irds:cranfield	True		True
irds:dpr-w100	True		
irds:dpr-w100/natural-questions/dev	True		[text, answers]
irds:dpr-w100/natural-questions/train	True		[text, answers]
irds:dpr-w100/trivia-qa/dev	True		[text, answers]
irds:dpr-w100/trivia-qa/train	True		[text, answers]
irds:gov	True		
irds:gov/trec-web-2002	True		[title, description, narrative]
irds:gov/trec-web-2002/named-page	True		True
irds:gov/trec-web-2003	True		[title, description]
irds:gov/trec-web-2003/named-page	True		True
irds:gov/trec-web-2004	True		True
irds:gov2	True		
irds:gov2/trec-mq-2008	True		True
irds:gov2/trec-tb-2004	True		[title, description, narrative]
irds:gov2/trec-tb-2005	True		[title, description, narrative]
irds:gov2/trec-tb-2005/efficiency	True		True
irds:gov2/trec-tb-2005/named-page	True		True
irds:gov2/trec-tb-2006	True		[title, description, narrative]
irds:gov2/trec-tb-2006/efficiency	True		True
irds:gov2/trec-tb-2006/efficiency/10k	True		True
irds:gov2/trec-tb-2006/efficiency/stream1	True		True
irds:gov2/trec-tb-2006/efficiency/stream2	True		True
irds:gov2/trec-tb-2006/efficiency/stream3	True		True
irds:gov2/trec-tb-2006/efficiency/stream4	True		True
irds:gov2/trec-tb-2006/named-page	True		True
irds:msmarco-passage	True		
irds:msmarco-passage/dev	True		True
irds:msmarco-passage/dev/small	True		True
irds:msmarco-passage/eval	True		True
irds:msmarco-passage/eval/small	True		True
irds:msmarco-passage/train	True		True
irds:msmarco-passage/trec-dl-2019	True		True
irds:msmarco-passage/trec-dl-2020	True		True
irds:mmarco			
irds:mr-tydi			
irds:mr-tydi/en	True		True
irds:mr-tydi/en/dev	True		True
irds:mr-tydi/en/test	True		True
irds:mr-tydi/en/train	True		True
irds:msmarco-document	True		
irds:msmarco-document/dev	True		True
irds:msmarco-document/eval	True		True
irds:msmarco-document/orcas	True		True

dataset	corpus	index	topics
irds:msmarco-document/train	True		True
irds:msmarco-document/trec-dl-2019	True		True
irds:msmarco-document/trec-dl-2020	True		True
irds:msmarco-document-v2	True		
irds:msmarco-document-v2/dev1	True		True
irds:msmarco-document-v2/dev2	True		True
irds:msmarco-document-v2/train	True		True
irds:msmarco-document-v2/trec-dl-2019	True		True
irds:msmarco-document-v2/trec-dl-2020	True		True
irds:msmarco-document-v2/trec-dl-2021	True		True
irds:msmarco-passage-v2	True		
irds:msmarco-passage-v2/dev1	True		True
irds:msmarco-passage-v2/dev2	True		True
irds:msmarco-passage-v2/train	True		True
irds:msmarco-passage-v2/trec-dl-2021	True		True
irds:msmarco-qna	True		
irds:msmarco-qna/dev	True		[text, type, answers]
irds:msmarco-qna/eval	True		[text, type]
irds:msmarco-qna/train	True		[text, type, answers]
irds:nfcorpus	True		
irds:nfcorpus/dev	True		[title, all]
irds:nfcorpus/dev/nontopic	True		True
irds:nfcorpus/dev/video	True		[title, desc]
irds:nfcorpus/test	True		[title, all]
irds:nfcorpus/test/nontopic	True		True
irds:nfcorpus/test/video	True		[title, desc]
irds:nfcorpus/train	True		[title, all]
irds:nfcorpus/train/nontopic	True		True
irds:nfcorpus/train/video	True		[title, desc]
irds:natural-questions	True		
irds:natural-questions/dev	True		True
irds:natural-questions/train	True		True
irds:nyt	True		
irds:nyt/trec-core-2017	True		[title, description, narrative]
irds:nyt/wksup	True		True
irds:pmc			
irds:pmc/v1	True		
irds:pmc/v1/trec-cds-2014	True		[type, description, summary]
irds:pmc/v1/trec-cds-2015	True		[type, description, summary]
irds:pmc/v2	True		
irds:pmc/v2/trec-cds-2016	True		[type, note, description, summary]
irds:argsme/2020-04-01/touche-2020-task-1	True		[title, description, narrative]
irds:clueweb12/touche-2020-task-2	True		[title, description, narrative]
irds:argsme/2020-04-01/touche-2021-task-1	True		True
irds:clueweb12/touche-2021-task-2	True		[title, description, narrative]
irds:argsme/1.0/touche-2020-task-1/uncorrected	True		[title, description, narrative]
irds:argsme/2020-04-01/touche-2020-task-1/uncorrected	True		[title, description, narrative]
irds:trec-robust04	True		[title, description, narrative]
irds:tripclick	True		

dataset	corpus	index	topics
irds:tripclick/logs	True		
irds:tripclick/test	True		True
irds:tripclick/test/head	True		True
irds:tripclick/test/tail	True		True
irds:tripclick/test/torso	True		True
irds:tripclick/train	True		True
irds:tripclick/train/head	True		True
irds:tripclick/train/head/dctr	True		True
irds:tripclick/train/tail	True		True
irds:tripclick/train/torso	True		True
irds:tripclick/val	True		True
irds:tripclick/val/head	True		True
irds:tripclick/val/head/dctr	True		True
irds:tripclick/val/tail	True		True
irds:tripclick/val/torso	True		True
irds:vaswani	True		True
irds:wapo			
irds:wapo/v2	True		
irds:wapo/v2/trec-core-2018	True		[title, description, narrative]
irds:wapo/v2/trec-news-2018	True		[doc_id, url]
irds:wapo/v2/trec-news-2019	True		[doc_id, url]
irds:wapo/v3/trec-news-2020			[doc_id, url]
irds:wikir			
irds:wikir/en1k	True		
irds:wikir/en1k/test	True		True
irds:wikir/en1k/training	True		True
irds:wikir/en1k/validation	True		True
irds:wikir/en59k	True		
irds:wikir/en59k/test	True		True
irds:wikir/en59k/training	True		True
irds:wikir/en59k/validation	True		True
irds:wikir/en78k	True		
irds:wikir/en78k/test	True		True
irds:wikir/en78k/training	True		True
irds:wikir/en78k/validation	True		True
irds:wikir/ens78k	True		
irds:wikir/ens78k/test	True		True
irds:wikir/ens78k/training	True		True
irds:wikir/ens78k/validation	True		True
irds:trec-fair-2021	True		
irds:trec-fair-2021/train	True		[text, keywords, scope, homepage]
irds:trec-fair-2021/eval	True		[text, keywords, scope]
trec-deep-learning-docs	True	True	[train, dev, test, test-2020, leaderboard
trec-deep-learning-passages	True	True	[train, dev, dev.small, eval, eval.small,

CHAPTER

THREE

TERRIER INDEXING

PyTerrier has a number of useful classes for creating Terrier indices, which can be used for retrieval, query expansion, etc. There are four indexer classes:

- You can create an index from TREC-formatted files, from a TREC test collection, using TRECCollectionIndexer.
- For indexing TXT, PDF, Microsoft Word files, etc files you can use FilesIndexer.
- For indexing Pandas Dataframe you can use DFIndexer.
- For any abitrary iterable dictionaries, you can use IterDictIndexer.

There are also different types of indexing supported in Terrier that are exposed in PyTerrier.

We explain both the indexing types and the indexer classes below, with examples. Further worked examples of indexing are provided in the example indexing notebook.

3.1 Index Types

`class pyterrier.index.`**`IndexingType`**`(`*value*`)`

This enum is used to determine the type of index built by Terrier. The default is CLASSIC.

`CLASSIC = 1`

A classical indexing regime, which also creates a direct index structure, useful for query expansion

`SINGLEPASS = 2`

A single-pass indexing regime, which builds an inverted index directly. No direct index structure is created. Typically is faster than classical indexing.

`MEMORY = 3`

An in-memory index. No direct index is created.

3.2 Indexer Classes

3.2.1 TRECCollectionIndexer

`class pyterrier.`**`TRECCollectionIndexer`**`(`*index_path*, *blocks=False*, *overwrite=False*, *type=IndexingType.CLASSIC*, *collection='trec'*, *verbose=False*, *meta={'docno': 20}*, *meta_reverse=['docno']*, *meta_tags={}*`)`

Use this Indexer if you wish to index a TREC formatted collection

Init method

Parameters

- **index_path** (*str*) – Directory to store index. Ignored for IndexingType.MEMORY.
- **blocks** (*bool*) – Create indexer with blocks if true, else without blocks. Default is False.
- **overwrite** (*bool*) – If index already present at *index_path*, True would overwrite it, False throws an Exception. Default is False.
- **type** (IndexingType) – the specific indexing procedure to use. Default is IndexingType.CLASSIC.
- **collection** (*Class name, or Class instance, or one of "trec", "trecweb", "warc"*) –
- **meta** (*Dict[str,int]*) – What metadata for each document to record in the index, and what length to reserve. Defaults to *{"docno" : 20}*.
- **meta_reverse** (*List[str]*) – What metadata shoudl we be able to resolve back to a docid. Defaults to *["docno"]*.
- **meta_tags** (*Dict[str,str]*) – For collections formed using tagged data (e.g. HTML), which tags correspond to which metadata. This is useful for recording the text of documents for use in neural rankers - see *Working with Document Texts*.

index(*files_path*)
Index the specified TREC formatted files

Parameters **files_path** – can be a String of the path or a list of Strings of the paths for multiple files

Example indexing the TREC WT2G corpus:

```
import pyterrier as pt
pt.init()
# list of filenames to index
files = pt.io.find_files("/path/to/WT2G/wt2g-corpus/")

# build the index
indexer = pt.TRECCollectionIndexer("./wt2g_index", verbose=True, blocks=False)
indexref = indexer.index(files)

# load the index, print the statistics
index = pt.IndexFactory.of(indexref)
print(index.getCollectionStatistics().toString())
```

3.2.2 FilesIndexer

class pyterrier.**FilesIndexer**(*index_path*, **args*, *meta={'docno': 20, 'filename': 512}*, *meta_reverse=['docno']*, *meta_tags={}*, ***kwargs*)
Use this Indexer if you wish to index a pdf, docx, txt etc files

Parameters

- **index_path** (*str*) – Directory to store index. Ignored for IndexingType.MEMORY.
- **blocks** (*bool*) – Create indexer with blocks if true, else without blocks. Default is False.
- **type** (IndexingType) – the specific indexing procedure to use. Default is IndexingType.CLASSIC.

- **meta** (*Dict[str,int]*) – What metadata for each document to record in the index, and what length to reserve. Defaults to *{"docno" : 20, "filename" : 512}*.
- **meta_reverse** (*List[str]*) – What metadata shoudl we be able to resolve back to a docid. Defaults to *["docno"]*,
- **meta_tags** (*Dict[str,str]*) – For collections formed using tagged data (e.g. HTML), which tags correspond to which metadata. Defaults to empty. This is useful for recording the text of documents for use in neural rankers - see *Working with Document Texts*.

Init method

Parameters

- **index_path** (*str*) – Directory to store index. Ignored for IndexingType.MEMORY.
- **blocks** (*bool*) – Create indexer with blocks if true, else without blocks. Default is False.
- **overwrite** (*bool*) – If index already present at *index_path*, True would overwrite it, False throws an Exception. Default is False.
- **verbose** (*bool*) – Provide progess bars if possible. Default is False.
- **type** (IndexingType) – the specific indexing procedure to use. Default is IndexingType.CLASSIC.

index(*files_path*)
Index the specified files.

Parameters **files_path** – can be a String of the path or a list of Strings of the paths for multiple files

3.2.3 DFIndexer

class pyterrier.**DFIndexer**(*index_path*, **args*, *blocks=False*, *overwrite=False*, *verbose=False*, *meta_reverse=['docno']*, *type=IndexingType.CLASSIC*, ***kwargs*)
Use this Indexer if you wish to index a pandas.Dataframe

Init method

Parameters

- **index_path** (*str*) – Directory to store index. Ignored for IndexingType.MEMORY.
- **blocks** (*bool*) – Create indexer with blocks if true, else without blocks. Default is False.
- **overwrite** (*bool*) – If index already present at *index_path*, True would overwrite it, False throws an Exception. Default is False.
- **verbose** (*bool*) – Provide progess bars if possible. Default is False.
- **type** (IndexingType) – the specific indexing procedure to use. Default is IndexingType.CLASSIC.

index(*text*, **args*, ***kwargs*)
Index the specified

Parameters

- **text** (*pd.Series*) – A pandas.Series(a column) where each row is the body of text for each document

- ***args** – Either a pandas.Dataframe or pandas.Series. If a Dataframe: All columns(including text) will be passed as metadata If a Series: The Series name will be the name of the metadata field and the body will be the metadata content
- ****kwargs** – Either a list, a tuple or a pandas.Series The name of the keyword argument will be the name of the metadata field and the keyword argument contents will be the metadata content

Example indexing a dataframe:

```
# define an example dataframe of documents
import pandas as pd
df = pd.DataFrame({
    'docno':
    ['1', '2', '3'],
    'url':
    ['url1', 'url2', 'url3'],
    'text':
    ['He ran out of money, so he had to stop playing',
    'The waves were crashing on the shore; it was a',
    'The body may perhaps compensates for the loss']
})

# index the text, record the docnos as metadata
pd_indexer = pt.DFIndexer("./pd_index")
indexref = pd_indexer.index(df["text"], df["docno"])
```

3.2.4 IterDictIndexer

`class pyterrier.IterDictIndexer`(*index_path*, **args*, *meta={'docno': 20}*, *meta_reverse=['docno']*, *threads=1*, ***kwargs*)

Use this Indexer if you wish to index an iter of dicts (possibly with multiple fields). This version is optimized by using multiple threads and POSIX fifos to tranfer data, which ends up being much faster.

Parameters

- **index_path** (`str`) – Directory to store index. Ignored for IndexingType.MEMORY.
- **meta** (`Dict[str,int]`) – What metadata for each document to record in the index, and what length to reserve. Defaults to *{"docno" : 20}*.
- **meta_reverse** (`List[str]`) – What metadata shoudl we be able to resolve back to a docid. Defaults to *["docno"]*,

index(*it*, *fields=('text',)*, *meta=None*, *meta_lengths=None*)

Index the specified iter of dicts with the (optional) specified fields

Parameters

- **it** (`iter[dict]`) – an iter of document dict to be indexed
- **fields** (`list[str]`) – keys to be indexed as fields
- **meta** (`list[str]`) – keys to be considered as metdata
- **meta_lengths** (`list[int]`) – length of metadata, defaults to 512 characters

Examples using IterDictIndexer

An iterdict can just be a list of dictionaries:

```
docs = [ { 'docno' : 'doc1', 'text' : 'a b c' }  ]
iter_indexer = pt.IterDictIndexer("./index")
indexref1 = iter_indexer.index(docs, meta=['docno', 'text'], meta_lengths=[20, 4096])
```

A dataframe can also be used, virtue of its `.to_dict()` method:

```
df = pd.DataFrame([['doc1', 'a b c']], columns=['docno', 'text'])
iter_indexer = pt.IterDictIndexer("./index")
indexref2 = indexer.index(df.to_dict(orient="records"))
```

However, the main power of using IterDictIndexer is for processing indefinite iterables, such as those returned by generator functions. For example, the tsv file of the MSMARCO Passage Ranking corpus can be indexed as follows:

```
dataset = pt.get_dataset("trec-deep-learning-passages")
def msmarco_generate():
    with pt.io.autoopen(dataset.get_corpus()[0], 'rt') as corpusfile:
        for l in corpusfile:
            docno, passage = l.split("\t")
            yield {'docno' : docno, 'text' : passage}

iter_indexer = pt.IterDictIndexer("./passage_index")
indexref3 = iter_indexer.index(msmarco_generate(), meta=['docno', 'text'], meta_
↪lengths=[20, 4096])
```

IterDictIndexer can be used in connection with *Indexing Pipelines*.

Similarly, indexing of JSONL files is similarly a few lines of Python:

```
def iter_file(filename):
  import json
  with open(filename, 'rt') as file:
    for l in file:
      # assumes that each line contains 'docno', 'text' attributes
      # yields a dictionary for each json line
      yield json.loads(l)

indexref4 = pt.IterDictIndexer("./index").index(iter_file("/path/to/file.jsonl"), meta=[
↪'docno', 'text'], meta_lengths=[20, 4096])
```

NB: Use `pt.io.autoopen()` as a drop-in replacement for `open()` that supports files compressed by gzip etc.

Indexing TREC-formatted files using IterDictIndexer

If you have TREC-formatted files that you wish to use with an IterDictIndexer-like indexer, `pt.index.treccollection2textgen()` can be used as a helper function to aid in parsing such files.

`pyterrier.index.`**`treccollection2textgen`**(*files*, *meta=['docno']*, *meta_tags={'text': 'ELSE'}*, *verbose=False*, *num_docs=None*, *tag_text_length=4096*)

Creates a generator of dictionaries on parsing TREC formatted files. This is useful for parsing TREC-formatted corpora in indexers like IterDictIndexer, or similar indexers in other plugins (e.g. ColBERTIndexer).

Parameters

- **`files`** (-) – list of files to parse in TREC format.
- **`meta`** (-) – list of attributes to expose in the dictionaries as metadata.
- **`meta_tags`** (-) – mapping of TREC tags as metadata.

- **tag_text_length** (-) – maximium length of metadata. Defaults to 4096.
- **verbose** (-) – set to true to show a TQDM progress bar. Defaults to True.
- **num_docs** (-) – a hint for TQDM to size the progress bar based on document counts rather than file count.

Example:

```
files = pt.io.find_files("/path/to/Disk45")
gen = pt.index.treccollection2textgen(files)
index = pt.IterDictIndexer("./index45").index(gen)
```

Example using Indexing Pipelines:

```
files = pt.io.find_files("/path/to/Disk45")
gen = pt.index.treccollection2textgen(files)
indexer = pt.text.sliding() >> pt.IterDictIndexer("./index45")
index = indexer.index(gen)
```

Threading

On UNIX-based systems, IterDictIndexer can also perform multi-threaded indexing:

```
iter_indexer = pt.IterDictIndexer("./passage_index_8", threads=8)
indexref6 = iter_indexer.index(msmarco_generate(), meta=['docno', 'text'], meta_
↪lengths=[20, 4096])
```

Note that the resulting index ordering with multiple threads is non-deterministic; if you need deterministic behavior you must index in single-threaded mode. Furthermore, indexing can only go as quickly as the document iterator, so to take full advantage of multi-threaded indexing, you will need to keep the iterator function light-weight. Many datasets provide a fast corpus iteration function (`get_corpus_iter()`), see more information in the *Importing Datasets*.

3.3 Indexing Configuration

Our aim is to expose all conventional Terrier indexing configuration through PyTerrier, for instance as constructor arguments to the Indexer classes. However, as Terrier is a legacy platform, some changes will take time to integrate into Terrier. Moreover, the manner of the configuration needed varies a little between the Indexer classes. In the following, we list common indexing configurations, and how to apply them when indexing using PyTerrier, noting any differences betweeen the Indexer classes.

- *stemming configuation or stopwords*: the default Terrier indexing configuration is to use a term pipeline of *Stopwords,PorterStemmer*. You can change the term pipeline configuration using the *"termpipeline"* property:

  ```
  indexer.setProperty("termpipelines", "")
  ```

  Note that any subsequent indexing or retrieval operation would also require the *"termpipeline"* property to be suitably updated. See the org.terrier.terms package for a list of the available term pipeline objects provided by Terrier.

- *languages and tokenisation*: Similarly, the choice of tokeniser is controlled by the *"tokeniser"* property. EnglishTokeniser is the default tokeniser. Other tokenisers are listed in org.terrier.indexing.tokenisation package. For instance, use *indexer.setProperty("tokeniser", "UTFTokeniser")* when indexing non-English text.

- *positions (aka blocks)* - all indexers expose a *blocks* boolean constructor argument to allow position information to be recoreded in the index. Defaults to False, i.e. positions are not recorded.

- *fields* - fields refers to storing the frequency of a terms occurrence in different parts of a document, e.g. title vs body vs anchor text. In the IterDictIndexer, fields are set in the *index()* method; otherwise the *"FieldTags.process"* property must be set. See the Terrier indexing documentation on fields for more information.
- *changing the tags parsed by TREC Collection* - use the relevant properties listed in the Terrier indexing documentation.
- *metaindex configuration*: metadata refers to the arbitrary strings associated to each document recorded in a Terrier index. These can range from the *"docno"* attribute of each document, as used to support experimentation, to other attributes such as the URL of the documents, or even the raw text of the document. Indeed, storing the raw text of each document is a trick often used when applying additional re-rankers such as BERT (see pyterrier_bert for more information on integrating PyTerrier with BERT-based re-rankers). Indexers now expose *meta* and *meta_tags* constructor kwarg to make this easier.
- *reverse metaindex configuration*: on occasion, there is a need to lookup up documents in a Terrier index based on their metadata, e.g. "docno". The *meta_reverse* constructor kwarg allows meta keys that support reverse lookup to be specified.

CHAPTER

FOUR

TERRIER RETRIEVAL

BatchRetrieve is one of the most commonly used PyTerrier objects. It represents a retrieval transformation, in which queries are mapped to retrieved documents. BatchRetrieve uses a pre-existing Terrier index data structure, typically saved on disk.

Typical usage:

```
index = pt.IndexFactory.of("/path/to/data.properties")
tf_idf = pt.BatchRetrieve(index, wmodel="TF_IDF")
bm25 = pt.BatchRetrieve(index, wmodel="BM25")
pl2 = pt.BatchRetrieve(index, wmodel="PL2")

pt.Experiment([tf_idf, bm25, pl2], topic, qrels, eval_metrics=["map"])
```

As BatchRetrieve is a retrieval transformation, it takes as input dataframes with columns *[“qid”, “query”]*, and returns dataframes with columns *[“qid”, “query”, “docno”, “score”, “rank”]*.

However, BatchRetrieve can also act as a re-ranker. In this scenario, it takes as input dataframes with columns *[“qid”, “query”, “docno”]*, and returns dataframes with columns *[“qid”, “query”, “docno”, “score”, “rank”]*.

For instance, to create a re-ranking pipeline that re-scores the top 100 BM25 documents using PL2:

```
bm25 = pt.BatchRetrieve(index, wmodel="BM25")
pl2 = pt.BatchRetrieve(index, wmodel="PL2")
pipeline = (bm25 % 100) >> pl2
```

4.1 BatchRetrieve

class pyterrier.**BatchRetrieve**(*index_location*, *controls=None*, *properties=None*, *metadata=['docno']*, *num_results=None*, *wmodel=None*, *threads=1*, ***kwargs*)

Use this class for retrieval by Terrier

Init method

Parameters

- **index_location** – An index-like object - An Index, an IndexRef, or a String that can be resolved to an IndexRef
- **controls** (*dict*) – A dictionary with the control names and values
- **properties** (*dict*) – A dictionary with the property keys and values
- **verbose** (*bool*) – If True transform method will display progress

- **num_results** (*int*) – Number of results to retrieve.
- **metadata** (*list*) – What metadata to retrieve

static from_dataset(*dataset*, *variant=None*, *version='latest'*, ***kwargs*)

Instantiates a BatchRetrieve object from a pre-built index access via a dataset. Pre-built indices are ofen provided via the Terrier Data Repository.

Examples:

```
dataset = pt.get_dataset("vaswani")
bm25 = pt.BatchRetrieve.from_dataset(dataset, "terrier_stemmed", wmodel="BM25")
#or
bm25 = pt.BatchRetrieve.from_dataset("vaswani", "terrier_stemmed", wmodel="BM25
↪")
```

Index Variants:

There are a number of standard index names.

- *terrier_stemmed* - a classical index, removing Terrier's standard stopwords, and applying Porter's English stemmer
- *terrier_stemmed_positions* - as per *terrier_stemmed*, but also containing position information
- *terrier_unstemmed* - a classical index, without applying stopword removal or stemming
- *terrier_stemmed_text* - as per *terrier_stemmed*, but also containing the raw text of the documents
- *terrier_unstemmed_text* - as per *terrier_stemmed*, but also containing the raw text of the documents

transform(*queries*)

Performs the retrieval

Parameters **queries** – String for a single query, list of queries, or a pandas.Dataframe with columns=['qid', 'query']. For re-ranking, the DataFrame may also have a 'docid' and or 'docno' column.

Returns pandas.Dataframe with columns=['qid', 'docno', 'rank', 'score']

4.2 Terrier Configuration

When using PyTerrier, we have to be aware of the underlying Terrier configuration, namely *properties* and *controls*. Properties are global configuration and were traditionally configured by editing a *terrier.properties* file; In contrast, controls are per-query configuration. In PyTerrier, we specify both when we construct the BatchRetrieve object:

Common controls:

- *"wmodel"* - the name of the weighting model. (This can also be specified using the wmodel kwarg). Valid values are the Java class name of any Terrier weighting model. Terrier provides many, such as *"BM25"*, *"PL2"*. A list can be found in the Terrier weighting models javadoc.
- *"qe"* - whether to run the Divergence from Randomness query expansion.
- *"qemodel"* - which Divergence from Randomness query expansion model. Default is *"Bo1"*. A list can be found the Terrier query expansion models javadoc.

Common properties:

- *"termpipelines"* - the default Terrier term pipeline configuration is *"Stopwords,PorterStemmer"*. If you have created an index with a different configuration, you will need to set the *"termpipelines"* property for *each* BatchRetrieve constructed.

Examples:

```
# these two BatchRetrieve instances are identical, using the same weighting model
bm25a = pt.BatchRetrieve(index, wmodel="BM25")
bm25b = pt.BatchRetrieve(index, controls={"wmodel":"BM25"})

# this one also applies query expansion inside Terrier
bm25_qe = pt.BatchRetrieve(index, wmodel="BM25" controls={"qe":"on", "qemodel" : "Bo1"})

# when we introduce an unstemmed BatchRetrieve, we ensure to explicitly set the␣
↪termpipelines
# for the other BatchRetrieve as well
bm25s_unstemmed = pt.BatchRetrieve(indexUS, wmodel="BM25", properties={"termpipelines" :
↪""})
bm25s_stemmed = pt.BatchRetrieve(indexSS, wmodel="BM25", properties={"termpipelines" :
↪"Stopwords,PorterStemmer"})
```

4.3 Index-Like Objects

When working with Terrier indices, BatchRetrieve allows can make use of:

- a string representing an index, such as "/path/to/data.properties"
- a Terrier IndexRef object, constructed from a string, but which may also hold a reference to the existing index.
- a Terrier Index object - the actual loaded index.

In general, there is a significant cost to loading an Index, as data structures may have to be loaded from disk. Where possible, for faster reuse, load the actual Index.

Bad Practice:

```
bm25 = pt.BatchRetrieve("/path/to/data.properties", wmodel="BM25")
pl2 = pt.BatchRetrieve("/path/to/data.properties", wmodel="PL2")
# here, the same index must be loaded twice
```

Good Practice:

```
index = pt.IndexFactory.of("/path/to/data.properties")
bm25 = pt.BatchRetrieve(index, wmodel="BM25")
pl2 = pt.BatchRetrieve(index, wmodel="PL2")
# here, we share the index between two instances of BatchRetrieve
```

4.4 TextScorer

Sometimes we want to apply Terrier to compute the score of document for a given query when we do not yet have the documents indexed. TextScorer allows a neat-workaround, in that an index is created on-the-fly for the documents, and these are then scored.

Optionally, an index-like object can be specified as the *background_index* kwarg, which will be used for the collection statistics (e.g. term frequencies, document lengths etc.

class pyterrier.batchretrieve.**TextScorer**(*takes='docs'*, ***kwargs*)

A re-ranker class, which takes the queries and the contents of documents, indexes the contents of the documents using a MemoryIndex, and performs ranking of those documents with respect to the queries. Unknown kwargs are passed to BatchRetrieve.

Parameters

- **takes** (*str*) – configuration - what is needed as input: *"queries"*, or *"docs"*. Default is *"docs"* since v0.8.
- **returns** (*str*) – configuration - what is needed as output: *"queries"*, or *"docs"*. Default is *"docs"*.
- **body_attr** (*str*) – what dataframe input column contains the text of the document. Default is *"body"*.
- **wmodel** (*str*) – example of configuration passed to BatchRetrieve.

Example:

```
df = pd.DataFrame(
    [
        ["q1", "chemical reactions", "d1", "professor protor poured the chemicals"],
        ["q1", "chemical reactions", "d2", "chemical brothers turned up the beats"],
    ], columns=["qid", "query", "text"])
textscorer = pt.TextScorer(takes="docs", body_attr="text", wmodel="TF_IDF")
rtr = textscorer.transform(df)
#rtr will score each document for the query "chemical reactions" based on the␣
↪provided document contents
```

4.5 Non-English Retrieval

By default, PyTerrier is configured for indexing and retrieval in English. See our notebook (colab) for details on how to configure PyTerrier in other languages.

4.6 Custom Weighting Models

Normally, weighting models are specified as a string class names. Terrier then loads the Java class of that name (it will search the org.terrier.matching.models package unless the class name is fully qualified (e.g. *"com.example.MyTF"*).

If you have your own Java weighting model instance (which extends the WeightingModel abstract class, you can load it and pass it directly to BatchRetrieve:

```
mymodel = pt.autoclass("com.example.MyTF")()
retr = pt.BatchRetrieve(indexref, wmodel=mymodel)
```

More usefully, it is possible to express a weighting model entirely in Python, as a function or a lambda expression, that can be used by Terrier for scoring. In this example, we create a Terrier BatchRetrieve instance that scores based solely on term frequency:

```
Tf = lambda keyFreq, posting, entryStats, collStats: posting.getFrequency()
retr = pt.BatchRetrieve(indexref, wmodel=Tf)
```

All functions passed must accept 4 arguments, as follows:

- keyFrequency(float): the weight of the term in the query, usually 1 except during PRF.
- posting(Posting): access to the information about the occurrence of the term in the current document (frequency, document length etc).
- entryStats(EntryStatistics): access to the information about the occurrence of the term in the whole index (document frequency, etc.).
- collStats(CollectionStatistics): access to the information about the index as a whole (number of documents, etc).

Note that due to the overheads of continually traversing the JNI boundary, using a Python function for scoring has a marked efficiency overhead. This is probably too slow for retrieval using most indices of any significant size, but allows simple explanation of weighting models and exploratory weighting model development.

CHAPTER

FIVE

RUNNING EXPERIMENTS

PyTerrier aims to make it easy to conduct an information retrieval experiment, namely, to run a transformer pipeline over a set of queries, and evaluating the outcome using standard information retrieval evaluation metrics based on known relevant documents (obtained from a set relevance assessments, also known as *qrels*). The evaluation metrics are calculated by the pytrec_eval library, a Python wrapper around the widely-used trec_eval evaluation tool.

The main way to achieve this is using *pt.Experiment()*.

5.1 API

`pyterrier.`**`Experiment`**(*retr_systems*, *topics*, *qrels*, *eval_metrics*, *names=None*, *perquery=False*, *dataframe=True*, *batch_size=None*, *filter_by_qrels=False*, *filter_by_topics=True*, *baseline=None*, *test='t'*, *correction=None*, *correction_alpha=0.05*, *highlight=None*, *round=None*, *verbose=False*, *save_dir=None*, *save_mode='reuse'*, ***kwargs*)

Allows easy comparison of multiple retrieval transformer pipelines using a common set of topics, and identical evaluation measures computed using the same qrels. In essence, each transformer is applied on the provided set of topics. Then the named trec_eval evaluation measures are computed (using *pt.Utils.evaluate()*) for each system.

Parameters

- **`retr_systems`** (*`list`*) – A list of transformers to evaluate. If you already have the results for one (or more) of your systems, a results dataframe can also be used here. Results produced by the transformers must have "qid", "docno", "score", "rank" columns.
- **`topics`** – Either a path to a topics file or a pandas.Dataframe with columns=['qid', 'query']
- **`qrels`** – Either a path to a qrels file or a pandas.Dataframe with columns=['qid','docno', 'label']
- **`eval_metrics`** (*`list`*) – Which evaluation metrics to use. E.g. ['map']
- **`names`** (*`list`*) – List of names for each retrieval system when presenting the results. Default=None. If None: Obtains the *str()* representation of each transformer as its name.
- **`batch_size`** (*`int`*) – If not None, evaluation is conducted in batches of batch_size topics. Default=None, which evaluates all topics at once. Applying a batch_size is useful if you have large numbers of topics, and/or if your pipeline requires large amounts of temporary memory during a run.
- **`filter_by_qrels`** (*`bool`*) – If True, will drop topics from the topics dataframe that have qids not appearing in the qrels dataframe.
- **`filter_by_topics`** (*`bool`*) – If True, will drop topics from the qrels dataframe that have qids not appearing in the topics dataframe.

- **perquery** (*bool*) – If True return each metric for each query, else return mean metrics across all queries. Default=False.
- **save_dir** (*str*) – If set to the name of a directory, the results of each transformer will be saved in TREC-formatted results file, whose filename is based on the systems names (as specified by `names` kwarg). If the file exists and `save_mode` is set to "reuse", then the file will be used for evaluation rather than the transformer. Default is None, such that saving and loading from files is disabled.
- **save_mode** (*str*) – Defines how existing files are used when `save_dir` is set. If set to "reuse", then files will be preferred over transformers for evaluation. If set to "overwrite", existing files will be replaced. Default is "reuse".
- **dataframe** (*bool*) – If True return results as a dataframe, else as a dictionary of dictionaries. Default=True.
- **baseline** (*int*) – If set to the index of an item of the retr_system list, will calculate the number of queries improved, degraded and the statistical significance (paired t-test p value) for each measure. Default=None: If None, no additional columns will be added for each measure.
- **test** (*string*) – Which significance testing approach to apply. Defaults to "t". Alternatives are "wilcoxon" - not typically used for IR experiments. A Callable can also be passed - it should follow the specification of scipy.stats.ttest_rel(), i.e. it expect two arrays of numbers, and return an array or tuple, of which the second value will be placed in the p-value column.
- **correction** (*string*) – Whether any multiple testing correction should be applied. E.g. 'bonferroni', 'holm', 'hs' aka 'holm-sidak'. Default is None. Additional columns are added denoting whether the null hypothesis can be rejected, and the corrected p value. See statsmodels.stats.multitest.multipletests() for more information about available testing correction.
- **correction_alpha** (*float*) – What alpha value for multiple testing correction. Default is 0.05.
- **highlight** (*str*) – If *highlight="bold"*, highlights in bold the best measure value in each column; if *highlight="color"* or *"colour"*, then the cell with the highest metric value will have a green background.
- **round** (*int*) – How many decimal places to round each measure value to. This can also be a dictionary mapping measure name to number of decimal places. Default is None, which is no rounding.
- **verbose** (*bool*) – If True, a tqdm progress bar is shown as systems (or systems*batches if batch_size is set) are executed. Default=False.

Returns A Dataframe with each retrieval system with each metric evaluated.

5.2 Examples

5.2.1 Average Effectiveness

Getting average effectiveness over a set of topics:

```
dataset = pt.get_dataset("vaswani")
# vaswani dataset provides an index, topics and qrels

```

(continues on next page)

(continued from previous page)

```
# lets generate two BRs to compare
tfidf = pt.BatchRetrieve(dataset.get_index(), wmodel="TF_IDF")
bm25 = pt.BatchRetrieve(dataset.get_index(), wmodel="BM25")

pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map", "recip_rank"]
)
```

The returned dataframe is as follows:

	name	map	recip_rank
0	BR(TF_IDF)	0.290905	0.699168
1	BR(BM25)	0.296517	0.725665

Each row represents one system. We can manually set the names of the systems, using the *names=* kwarg, as follows:

```
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map", "recip_rank"],
    names=["TF_IDF", "BM25"]
)
```

This produces dataframes that are more easily interpretable.

	name	map	recip_rank
0	TF_IDF	0.290905	0.699168
1	BM25	0.296517	0.725665

We can also reduce the number of decimal places reported using the *round=* kwarg, as follows:

```
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map", "recip_rank"],
    round={"map" : 4, "recip_rank" : 3},
    names=["TF_IDF", "BM25"]
)
```

The result is as follows:

	name	map	recip_rank
0	TF_IDF	0.2909	0.699
1	BM25	0.2965	0.726

Passing an integer value to *round=* (e.g. *round=3*) applies rounding to all evaluation measures.

5.2.2 Significance Testing

We can perform significance testing by specifying the index of which transformer we consider to be our baseline, e.g. *baseline=0*:

```
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map", "recip_rank"],
    names=["TF_IDF", "BM25"],
    baseline=0
)
```

In this case, additional columns are returned for each measure, indicating the number of queries improved compared to the baseline, the number of queries degraded, as well as the paired t-test p-value in the difference between each row and the baseline row. NB: For the baseline, these values are NaN (not applicable).

	name	map	recip_rank	map +	map -	map p-value	recip_rank +	recip_rank -	recip_rank p-value
0	TF_IDF	0.290905	0.699168	nan	nan	nan	nan	nan	nan
1	BM25	0.296517	0.725665	46	45	0.237317	16	3	0.0258549

For this test collection, between the TF_IDF and BM25 weighting models, there is no significant difference observed in terms of MAP, but there is a significant different in terms of mean reciprocal rank (*p<0.05*). Indeed, while BM25 improves average precision for 46 queries over TF_IDF, it degrades it for 45; on the other hand, the rank of the first relevant document is improved for 16 queries by BM25 over TD_IDF.

Further more, modern experimental convention suggests that it is important to correct for multiple testing in the comparative evaluation of many IR systems. Experiments provides supported for the multiple testing correction methods supported by the statsmodels package, such as *Bonferroni*:

```
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map"],
    names=["TF_IDF", "BM25"],
    baseline=0,
    correction='bonferroni'
)
```

This adds two further columns for each measure, denoting if the null hypothesis can be rejected (e.g. *"map reject"*), as well as the corrected p value (*"map p-value corrected"*), as shown below:

	name	map	map +	map -	map p-value	map reject	map p-value corrected
0	TF_IDF	0.290905	nan	nan	nan	False	nan
1	BM25	0.296517	46	45	0.237317	False	0.474634

The table below summarises the multiple testing correction methods supported:

	Aliases	Correction Method
0	[‘b’, ‘bonf’, ‘bonferroni’]	Bonferroni
1	[‘s’, ‘sidak’]	Sidak
2	[‘h’, ‘holm’]	Holm
3	[‘hs’, ‘holm-sidak’]	Holm-Sidak
4	[‘sh’, ‘simes-hochberg’]	Simes-Hochberg
5	[‘ho’, ‘hommel’]	Hommel
6	[‘fdr_bh’, ‘fdr_i’, ‘fdr_p’, ‘fdri’, ‘fdrp’]	FDR Benjamini-Hochberg
7	[‘fdr_by’, ‘fdr_n’, ‘fdr_c’, ‘fdrn’, ‘fdrcorr’]	FDR Benjamini-Yekutieli
8	[‘fdr_tsbh’, ‘fdr_2sbh’]	FDR 2-stage Benjamini-Hochberg
9	[‘fdr_tsbky’, ‘fdr_2sbky’, ‘fdr_twostage’]	FDR 2-stage Benjamini-Krieger-Yekutieli
10	[‘fdr_gbs’]	FDR adaptive Gavrilov-Benjamini-Sarkar

Any value in the Aliases column can be passed to Experiment’s *correction=* kwarg.

5.2.3 Per-query Effectiveness

Finally, if necessary, we can request per-query performances using the *perquery=True* kwarg:

```
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map", "recip_rank"],
    names=["TF_IDF", "BM25"],
    perquery=True
)
```

This provides a dataframe where each row is the performance of a given system for a give query on a particular evaluation measure.

	name	qid	measure	value
186	BM25	1	map	0.26794
187	BM25	1	recip_rank	1
204	BM25	10	map	0.115631
205	BM25	10	recip_rank	0.5
206	BM25	11	map	0.0776046

NB: For brevity, we only show the top 5 rows of the returned table.

5.2.4 Saving and Reusing Results

For some research tasks, it is considered good practice to save your results files when conducting experiments. This allows several advantages:

- It permits additional evaluation (e.g. more measures, more signifiance tests) without re-applying potentially slow transformer pipelines.
- It allows transformer results to be made available for other experiments, perhaps as a virtual data appendix in a paper.

Saving can be enabled by adding the `save_dir` as a kwarg to pt.Experiment:

```
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map", "recip_rank"],
    names=["TF_IDF", "BM25"],
    save_dir="./",
)
```

This will save two files, namely, TF_IDF.res.gz and BM25.res.gz to the current directory. If these files already exist, they will be "reused", i.e. loaded and evaluated in preference to application of the tfidf and/or bm25 transformers. If experiments are being conducted on multiple different topic sets, care should be taken to ensure that previous results for a different topic set are not reused for evaluation.

If a transformer has been updated, outdated results files can be mistakenly used. To prevent this, set the `save_mode` kwarg to `"overwrite"`:

```
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=["map", "recip_rank"],
    names=["TF_IDF", "BM25"],
    save_dir="./",
    save_mode="overwrite"
)
```

5.2.5 Missing Topics and/or Qrels

There is not always a one-to-one correspondance between the topic/query IDs (qids) that appear in the provided `topics` and `qrels`. Qids that appear in topics but not qrels can be due to incomplete judgments, such as in sparsely labeled datasets or shared tasks that choose to omit some topics (e.g., due to cost). Qids that appear in qrels but no in topics can happen when running a subset of topics for testing purposes (e.g., `topics.head(5)`).

The `filter_by_qrels` and `filter_by_topics` parameters control the behaviour of an experiment when topics and qrels do not perfectly overlap. When `filter_by_qrels=True`, topics are filtered down to only the ones that have qids in the qrels. Similarly, when `filter_by_topics=True`, qrels are filtered down to only the ones that have qids in the topics.

For example, consier topics that include qids `A` and `B` and qrels that include `B` and `C`. The results with each combination of settings are:

`filter_by_topics`	`filter_by_qrels`	Results consider	Notes
`True` (default)	`False` (default)	`A,B`	`C` is removed because it does not appear in the topics.
`True` (default)	`True`	`B`	Acts as an intersection of the qids found in the qrels and topics.
`False`	`False` (default)	`A,B,C`	Acts as a union of the qids found in qrels and topics.
`False`	`True`	`B,C`	`A` is removed because it does not appear in the qrels.

Note that, following IR evaluation conventions, topics that have no relevance judgments (`A` in the above example) do not contribute to relevance-based measures (e.g., `map`), but still contribute to efficiency measures (e.g., `mrt`). As such, aggregate relevance-based measures will not change based on the value of `filter_by_qrels`. When `perquery=True`, topics that have no relevance judgments (`A`) will give a value of `NaN`, indicating that they are not defined and should not contribute to the average.

The defaults (`filter_by_topics=True` and `filter_by_qrels=False`) were chosen because they likely reflect the intent of the user in most cases. In particular, it runs all topics requested and evaluates on only those topics. However, you may want to change these settings in some circumstnaces. E.g.:

- If you want to save time and avoid running topics that will not be evaluated, set `filter_by_qrels=True`. This can be particularly helpful for large collections with many missing judgments, such as MS MARCO.
- If you want to evaluate across all topics from the qrels set `filter_by_topics=False`.

Note that in all cases, if a requested topic that appears in the qrels returns no results, it will properly contribute a score of 0 for evaluation.

5.3 Available Evaluation Measures

All trec_eval evaluation measure are available. Often used measures, including the name that must be used, are:

- Mean Average Precision (*map*).
- Mean Reciprocal Rank (*recip_rank*).
- Normalized Discounted Cumulative Gain (*ndcg*), or calculated at a given rank cutoff (e.g. *ndcg_cut_5*).
- Number of queries (*num_q*) - not averaged.
- Number of retrieved documents (*num_ret*) - not averaged.
- Number of relevant documents (*num_rel*) - not averaged.
- Number of relevant documents retrieved (*num_rel_ret*) - not averaged.
- Interpolated recall precision curves (*iprec_at_recall*). This is family of measures, so requesting *iprec_at_recall* will output measurements for *IPrec@0.00*, *IPrec@0.10*, etc.
- Precision at rank cutoff (e.g. *P_5*).
- Recall (*recall*) will generate recall at different cutoffs, such as *recall_5*, etc.).
- Mean response time (*mrt*) will report the average number of milliseconds to conduct a query (this is calculated by *pt.Experiment()* directly, not pytrec_eval).
- trec_eval measure *families* such as *official*, *set* and *all_trec* will be expanded. These result in many measures being returned. For instance, asking for *official* results in the following (very wide) output reporting the usual default metrics of trec_eval:

	name	P@5	P@10	P@15	P@20	P@30	P@100	P@200	P@500	P@1000	Rprec	Bpref	IPrec@0.0	IPrec@0.1	IPrec@0.2	IPrec@0.3	IPrec@0.4	IPrec@0.5	IPrec@0.6	IPrec@0.7	IPrec@0.8	IPrec@0.9	IPrec@1.0	AP	NumQ	NumRel	NumRet	NumRet(rel=1)	RR
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

See also a list of common TREC eval measures.

5.4 Evaluation Measures Objects

Using the ir_measures Python package, PyTerrier supports evaluation measure objects. These make it easier to express measure configurations such as rank cutoffs:

```
from pyterrier.measures import *
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics(),
    dataset.get_qrels(),
    eval_metrics=[AP, RR, nDCG@5],
)
```

NB: We have to use *from pyterrier.measures import* *, as *from pt.measures import* * wont work.

More specifically, lets consider the TREC Deep Learning track passage ranking task, which requires NDCG@10, NDCG@100 (using graded labels), as well as MRR@10 and MAP using binary labels (where relevant is grade 2 and above). The necessary incantation of *pt.Experiment()* looks like:

```
from pyterrier.measures import *
dataset = pt.get_dataset("trec-deep-learning-passages")
pt.Experiment(
    [tfidf, bm25],
    dataset.get_topics("test-2019"),
    dataset.get_qrels("test-2019"),
    eval_metrics=[RR(rel=2), nDCG@10, nDCG@100, AP(rel=2)],
)
```

The available evaluation measure objects are listed below.

`pyterrier.measures.P`(***kwargs*)

Basic measure for that computes the percentage of documents in the top cutoff results that are labeled as relevant. cutoff is a required parameter, and can be provided as P@cutoff.

```
@misc{rijsbergen:1979:ir,
  title={Information Retrieval.},
  author={Van Rijsbergen, Cornelis J},
  year={1979},
  publisher={USA: Butterworth-Heinemann}
}
```

`pyterrier.measures.R`(***kwargs*)

Recall@k (R@k). The fraction of relevant documents for a query that have been retrieved by rank k.

NOTE: Some tasks define Recall@k as whether any relevant documents are found in the top k results. This software follows the TREC convention and refers to that measure as Success@k.

`pyterrier.measures.AP`(***kwargs*)

The [Mean] Average Precision ([M]AP). The average precision of a single query is the mean of the precision scores at each relevant item returned in a search results list.

AP is typically used for adhoc ranking tasks where getting as many relevant items as possible is. It is commonly referred to as MAP, by taking the mean of AP over the query set.

```
@article{Harman:1992:ESIR,
  author = {Donna Harman},
  title = {Evaluation Issues in Information Retrieval},
  journal = {Information Processing and Management},
  volume = {28},
  number = {4},
  pages = {439 - -440},
  year = {1992},
}
```

`pyterrier.measures.RR(**kwargs)`

The [Mean] Reciprocal Rank ([M]RR) is a precision-focused measure that scores based on the reciprocal of the rank of the highest-scoring relevance document. An optional cutoff can be provided to limit the depth explored. rel (default 1) controls which relevance level is considered relevant.

```
@article{kantor2000trec,
  title={The TREC-5 Confusion Track},
  author={Kantor, Paul and Voorhees, Ellen},
  journal={Information Retrieval},
  volume={2},
  number={2-3},
  pages={165--176},
  year={2000}
}
```

`pyterrier.measures.nDCG(**kwargs)`

The normalized Discounted Cumulative Gain (nDCG). Uses graded labels - systems that put the highest graded documents at the top of the ranking. It is normalized wrt. the Ideal NDCG, i.e. documents ranked in descending order of graded label.

```
@article{Jarvelin:2002:CGE:582415.582418,
  author = {J"{a}rvelin, Kalervo and Kek"{a}l"{a}inen, Jaana},
  title = {Cumulated Gain-based Evaluation of IR Techniques},
  journal = {ACM Trans. Inf. Syst.},
  volume = {20},
  number = {4},
  year = {2002},
  pages = {422--446},
  numpages = {25},
  url = {http://doi.acm.org/10.1145/582415.582418},
}
```

`pyterrier.measures.ERR(**kwargs)`

The Expected Reciprocal Rank (ERR) is a precision-focused measure. In essence, an extension of reciprocal rank that encapsulates both graded relevance and a more realistic cascade-based user model of how users brwose a ranking.

`pyterrier.measures.Success(**kwargs)`

1 if a document with at least rel relevance is found in the first cutoff documents, else 0.

NOTE: Some refer to this measure as Recall@k. This software follows the TREC convention, where Recall@k is defined as the proportion of known relevant documents retrieved in the top k results.

`pyterrier.measures.Judged(**kwargs)`

Percentage of results in the top k (cutoff) results that have relevance judgments. Equivalent to P@k with a rel lower than any judgment.

`pyterrier.measures.NumQ(**kwargs)`

The total number of queries.

`pyterrier.measures.NumRet(**kwargs)`

The number of results returned. When rel is provided, counts the number of documents returned with at least that relevance score (inclusive).

`pyterrier.measures.NumRelRet(**kwargs)`

The number of results returned. When rel is provided, counts the number of documents returned with at least that relevance score (inclusive).

`pyterrier.measures.NumRel(**kwargs)`

The number of relevant documents the query has (independent of what the system retrieved).

`pyterrier.measures.Rprec(**kwargs)`

The precision of at R, where R is the number of relevant documents for a given query. Has the cute property that it is also the recall at R.

```
@misc{Buckley2005RetrievalSE,
  title={Retrieval System Evaluation},
  author={Chris Buckley and Ellen M. Voorhees},
  annote={Chapter 3 in TREC: Experiment and Evaluation in Information Retrieval},
  howpublished={MIT Press},
  year={2005}
}
```

`pyterrier.measures.Bpref(**kwargs)`

Binary Preference (Bpref). This measure examines the relative ranks of judged relevant and non-relevant documents. Non-judged documents are not considered.

```
@inproceedings{Buckley2004RetrievalEW,
  title={Retrieval evaluation with incomplete information},
  author={Chris Buckley and Ellen M. Voorhees},
  booktitle={SIGIR},
  year={2004}
}
```

`pyterrier.measures.infAP(**kwargs)`

Inferred AP. AP implementation that accounts for pooled-but-unjudged documents by assuming that they are relevant at the same proportion as other judged documents. Essentially, skips documents that were pooled-but-not-judged, and assumes unjudged are non-relevant.

Pooled-but-unjudged indicated by a score of -1, by convention. Note that not all qrels use this convention.

CHAPTER

SIX

QUERY REWRITING & EXPANSION

Query rewriting refers to changing the formulation of the query in order to improve the effectiveness of the search ranking. PyTerrier supplies a number of query rewriting transformers designed to work with BatchRetrieve.

Firstly, we differentiate between two forms of query rewriting:

- *Q* -> *Q*: this rewrites the query, for instance by adding/removing extra query terms. Examples might be a WordNet- or Word2Vec-based QE; The input dataframes contain only *["qid", "docno"]* columns. The output dataframes contain *["qid", "query", "query_0"]* columns, where *"query"* contains the reformulated query, and *"query_0"* contains the previous formulation of the query.
- *R* -> *Q*: these class of transformers rewrite a query by making use of an associated set of documents. This is typically exemplifed by pseudo-relevance feedback. Similarly the output dataframes contain *["qid", "query", "query_0"]* columns.

The previous formulation of the query can be restored using *pt.rewrite.reset()*, discussed below.

6.1 SequentialDependence

This class implements Metzler and Croft's sequential dependence model, designed to boost the scores of documents where the query terms occur in close proximity. Application of this transformer rewrites each input query such that:

- pairs of adjacent query terms are added as *#1* and *#uw8* complex query terms, with a low weight.
- the full query is added as *#uw12* complex query term, with a low weight.
- all terms are weighted by a proximity model, either Dirichlet LM or pBiL2.

For example, the query *pyterrier IR platform* would become *pyterrier IR platform #1(pyterrier IR) #1(IR platform) #uw8(pyterrier IR) #uw8(IR platform) #uw12(pyterrier IR platform)*. NB: Acutally, we have simplified the rewritten query - in practice, we also (a) set the weight of the proximity terms to be low using a *#combine()* operator and (b) set a proximity term weighting model.

This transfomer is only compatible with BatchRetrieve, as Terrier supports the *#1* and *#uwN* complex query terms operators. The Terrier index must have blocks (positional information) recorded in the index.

`class pyterrier.rewrite.`**`SequentialDependence`**(*verbose=0*, *remove_stopwords=True*, *prox_model=None*, ***kwargs*)

Implements the sequential dependence model, which Terrier supports using its Indri/Galagoo compatible matchop query language. The rewritten query is derived using the Terrier class DependenceModelPreProcess.

This transformer changes the query. It must be followed by a Terrier Retrieve() transformer. The original query is saved in the *"query_0"* column, which can be restored using *pt.rewrite.reset()*.

`transform`(*topics_and_res*)

Abstract method for all transformations. Typically takes as input a Pandas DataFrame, and also returns one.

Example:

```
sdm = pt.rewrite.SequentialDependence()
dph = pt.BatchRetrieve(index, wmodel="DPH")
pipeline = sdm >> dph
```

References:

- A Markov Random Field Model for Term Dependencies. Donald Metzler and W. Bruce Croft. In Proceedings of SIGIR 2005.
- Incorporating Term Dependency in the DFR Framework. Jie Peng, Craig Macdonald, Ben He, Vassilis Plachouras, Iadh Ounis. In Proceedings of SIGIR 2007. July 2007. Amsterdam, the Netherlands. 2007.

6.2 Bo1QueryExpansion

This class applies the Bo1 Divergence from Randomess query expansion model to rewrite the query based on the occurences of terms in the feedback documents provided for each query. In this way, it takes in a dataframe with columns *["qid", "query", "docno", "score", "rank"]* and returns a dataframe with *["qid", "query"]*.

`class pyterrier.rewrite.`**`Bo1QueryExpansion`**`(`*`*args`*`, `*`**kwargs`*`)`

Applies the Bo1 query expansion model from the Divergence from Randomness Framework, as provided by Terrier. It must be followed by a Terrier Retrieve() transformer. The original query is saved in the *"query_0"* column, which can be restored using *pt.rewrite.reset()*.

Instance Attributes:

- fb_terms(int): number of feedback terms. Defaults to 10
- fb_docs(int): number of feedback documents. Defaults to 3

Parameters

- **`index_like`** – the Terrier index to use.
- **`fb_terms`** (*`int`*) – number of terms to add to the query. Terrier's default setting is 10 expansion terms.
- **`fb_docs`** (*`int`*) – number of feedback documents to consider. Terrier's default setting is 3 feedback documents.

`transform``(`*`topics_and_res`*`)`

Abstract method for all transformations. Typically takes as input a Pandas DataFrame, and also returns one.

Example:

```
bo1 = pt.rewrite.Bo1QueryExpansion(index)
dph = pt.BatchRetrieve(index, wmodel="DPH")
pipelineQE = dph >> bo1 >> dph
```

Alternative Formulations

Note that it is also possible to configure BatchRetrieve to perform QE directly using controls, which will result in identical retrieval effectiveness:

```
pipelineQE = pt.BatchRetrieve(index, wmodel="DPH", controls={"qemodel" : "Bo1", "qe" :
↪"on"})
```

However, using *pt.rewrite.Bo1QueryExpansion* is preferable as:

- the semantics of retrieve >> rewrite >> retrieve are clearly visible.
- the complex control configuration of Terrier need not be learned.
- the rewritten query is visible outside, and not hidden inside Terrier.

References:

- Amati, Giambattista (2003) Probability models for information retrieval based on divergence from randomness. PhD thesis, University of Glasgow.

6.3 KLQueryExpansion

Similar to Bo1, this class deploys a Divergence from Randomess query expansion model based on Kullback Leibler divergence.

`class pyterrier.rewrite.`**`KLQueryExpansion`**`(`**args*, ***kwargs*`)`

Applies the KL query expansion model from the Divergence from Randomness Framework, as provided by Terrier. This transformer must be followed by a Terrier Retrieve() transformer. The original query is saved in the *"query_0"* column, which can be restored using *pt.rewrite.reset()*.

Instance Attributes:

- fb_terms(int): number of feedback terms. Defaults to 10
- fb_docs(int): number of feedback documents. Defaults to 3

Parameters

- **`index_like`** – the Terrier index to use
- **`fb_terms`** (*`int`*) – number of terms to add to the query. Terrier's default setting is 10 expansion terms.
- **`fb_docs`** (*`int`*) – number of feedback documents to consider. Terrier's default setting is 3 feedback documents.

`transform``(`*topics_and_res*`)`

Abstract method for all transformations. Typically takes as input a Pandas DataFrame, and also returns one.

References:

- Amati, Giambattista (2003) Probability models for information retrieval based on divergence from randomness. PhD thesis, University of Glasgow.

6.4 RM3

`class pyterrier.rewrite.`**`RM3`**`(`**args*, *fb_terms=10*, *fb_docs=3*, *fb_lambda=0.6*, ***kwargs*`)`

Performs query expansion using RM3 relevance models. RM3 relies on an external Terrier plugin, terrier-prf. You should start PyTerrier with *pt.init(boot_packages=["com.github.terrierteam:terrier-prf:-SNAPSHOT"])*.

This transformer must be followed by a Terrier Retrieve() transformer. The original query is saved in the *"query_0"* column, which can be restored using *pt.rewrite.reset()*.

Instance Attributes:

- fb_terms(int): number of feedback terms. Defaults to 10

- fb_docs(int): number of feedback documents. Defaults to 3
- fb_lambda(float): lambda in RM3, i.e. importance of relevance model viz feedback model. Defaults to 0.6.

Example:

```
bm25 = pt.BatchRetrieve(index, wmodel="BM25")
rm3_pipe = bm25 >> pt.rewrite.RM3(index) >> bm25
pt.Experiment([bm25, rm3_pipe],
            dataset.get_topics(),
            dataset.get_qrels(),
            ["map"]
            )
```

Parameters

- **index_like** – the Terrier index to use
- **fb_terms** (*int*) – number of terms to add to the query. Terrier's default setting is 10 expansion terms.
- **fb_docs** (*int*) – number of feedback documents to consider. Terrier's default setting is 3 feedback documents.
- **fb_lambda** (*float*) – lambda in RM3, i.e. importance of relevance model viz feedback model. Defaults to 0.6.

transform(*queries_and_docs*)
Abstract method for all transformations. Typically takes as input a Pandas DataFrame, and also returns one.

References:

- Nasreen Abdul-Jaleel, James Allan, W Bruce Croft, Fernando Diaz, Leah Larkey, Xiaoyan Li, Mark D Smucker, and Courtney Wade. UMass at TREC 2004: Novelty and HARD. In Proceedings of TREC 2004.

6.5 AxiomaticQE

class pyterrier.rewrite.**AxiomaticQE**(**args*, *fb_terms=10*, *fb_docs=3*, ***kwargs*)
Performs query expansion using axiomatic query expansion. This class relies on an external Terrier plugin, terrier-prf. You should start PyTerrier with *pt.init(boot_packages=["com.github.terrierteam:terrier-prf:-SNAPSHOT"])*.

This transformer must be followed by a Terrier Retrieve() transformer. The original query is saved in the *"query_0"* column, which can be restored using *pt.rewrite.reset()*.

Instance Attributes:

- fb_terms(int): number of feedback terms. Defaults to 10
- fb_docs(int): number of feedback documents. Defaults to 3

Parameters

- **index_like** – the Terrier index to use
- **fb_terms** (*int*) – number of terms to add to the query. Terrier's default setting is 10 expansion terms.

- **fb_docs** (*int*) – number of feedback documents to consider. Terrier's default setting is 3 feedback documents.

transform(*queries_and_docs*)
Abstract method for all transformations. Typically takes as input a Pandas DataFrame, and also returns one.

References:

- Hui Fang, Chang Zhai.: Semantic term matching in axiomatic approaches to information retrieval. In: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 115–122. SIGIR 2006. ACM, New York (2006).
- Peilin Yang and Jimmy Lin, Reproducing and Generalizing Semantic Term Matching in Axiomatic Information Retrieval. In Proceedings of ECIR 2019.

6.6 Combining Query Formulations

pyterrier.rewrite.**linear**(*weightCurrent*, *weightPrevious*, *format='terrierql'*, ***kwargs*)
Applied to make a linear combination of the current and previous query formulation. The implementation is tied to the underlying query language used by the retrieval/re-ranker transformers. Two of Terrier's query language formats are supported by the *format* kwarg, namely *"terrierql"* and *"matchoptql"*. Their exact respective formats are detailed in the Terrier documentation.

Parameters

- **weightCurrent** (*float*) – weight to apply to the current query formulation.
- **weightPrevious** (*float*) – weight to apply to the previous query formulation.
- **format** (*str*) – which query language to use to rewrite the queries, one of "terrierql" or "matchopql".

Example:

```
pipeTQL = pt.apply.query(lambda row: "az") >> pt.rewrite.linear(0.75, 0.25, format=
↪"terrierql")
pipeMQL = pt.apply.query(lambda row: "az") >> pt.rewrite.linear(0.75, 0.25, format=
↪"matchopql")
pipeT.search("a")
pipeM.search("a")
```

Example outputs of *pipeTQL* and *pipeMQL* corresponding to the query "a" above:

- Terrier QL output: *"(az)^0.750000 (a)^0.250000"*
- MatchOp QL output: *"#combine:0:0.750000:1:0.250000(#combine(az) #combine(a))"*

Return type TransformerBase

6.7 Resetting the Query Formulation

The application of any query rewriting operation, including the apply transformer, *pt.apply.query()*, will return a dataframe that includes the *input* formulation of the query in the *query_0* column, and the new reformulation in the *query* column. The previous query reformulation can be obtained by inclusion of a reset transformer in the pipeline.

`pyterrier.rewrite.`**`reset`**`()`

Undoes a previous query rewriting operation. This results in the query formulation stored in the *"query_0"* attribute being moved to the *"query"* attribute, and, if present, the *"query_1"* being moved to *"query_0"* and so on. This transformation is useful if you have rewritten the query for the purposes of one retrieval stage, but wish a subquent transformer to be applies on the original formulation.

Internally, this function applies *pt.model.pop_queries()*.

Example:

```
firststage = pt.rewrite.SDM() >> pt.BatchRetrieve(index, wmodel="DPH")
secondstage = pyterrier_bert.cedr.CEDRPipeline()
fullranker = firststage >> pt.rewrite.reset() >> secondstage
```

Return type `TransformerBase`

6.8 Stashing the Documents

Sometimes you want to apply a query rewriting function as a re-ranker, but your rewriting function uses a different document ranking. In this case, you can use *pt.rewrite.stash_results()* to stash the retrieved documents for each query, so they can be recovered and re-ranked later using your rewritten query formulation.

`pyterrier.rewrite.`**`stash_results`**`(`*clear=True*`)`

Stashes (saves) the current retrieved documents for each query into the column *"stashed_results_0"*. This means that they can be restored later by using *pt.rewrite.reset_results()*. thereby converting a retrieved documents dataframe into one of queries.

Args: clear(bool): whether to drop the document and retrieved document related columns. Defaults to True.

Return type `TransformerBase`

`pyterrier.rewrite.`**`reset_results`**`()`

Applies a transformer that undoes a *pt.rewrite.stash_results()* transformer, thereby restoring the ranked documents.

Return type `TransformerBase`

Example: Query Expansion as a re-ranker

Some papers advocate for the use of query expansion (PRF) as a re-ranker. This can be attained in PyTerrier through use of *stash_results()* and *reset_results()*:

```
# index: the corpus you are ranking

dph = pt.BatchRetrieve(index)
Pipe = dph
    >> pt.rewrite.stash_results(clear=False)
    >> pt.rewrite.RM3(index)
    >> pt.rewrite.reset_results()
    >> dph
```

Summary of dataframe types:

output of	dataframe contents	actual columns
dph	R	qid, query, docno, score
stash_results	R + “stashed_results_0”	qid, query, docno, score, stashed_results_0
RM3	Q + “stashed_results_0”	qid, query, query_0, stashed_results_0
reset_results	R	qid, query, docno, score, query_0
dph	R	qid, query, docno, score, query_0

Indeed, as we need RM3 to have the initial ranking of documents as input, we use *clear=False* as the kwarg to stash_results().

Example: Collection Enrichment as a re-ranker:

```
# index: the corpus you are ranking
# wiki_index: index of Wikipedia, used for enrichment

dph = pt.BatchRetrieve(index)
Pipe = dph
    >> pt.rewrite.stash_results()
    >> pt.BatchRetrieve(wiki_index)
    >> pt.rewrite.RM3(wiki_index)
    >> pt.rewrite.reset_results()
    >> dph
```

In general, collection enrichment describes conducting a PRF query expansion process on an external corpus (often Wikipedia), before applying the reformulated query to the main corpus. Collection enrichment can be used for improving a first pass retrieval (*pt.BatchRetrieve(wiki_index) >> pt.rewrite.RM3(wiki_index) >> pt.BatchRetrieve(main_index)*). Instead, the particular example shown above applies collection enrichment as a re-ranker.

Summary of dataframe types:

output of	dataframe contents	actual columns
dph	R	qid, query, docno, score
stash_results	Q + “stashed_results_0”	qid, query, saved_docs_0
BatchRetrieve	R + “stashed_results_0”	qid, query, docno, score, stashed_results_0
RM3	Q + “stashed_results_0”	qid, query, query_0, stashed_results_0
reset_results	R	qid, query, docno, score, query_0
dph	R	qid, query, docno, score, query_0

In this example, we have a BatchRetrieve instance executed on the wiki_index before RM3, so we clear the document ranking columns when using *stash_results()*.

CHAPTER

SEVEN

LEARNING TO RANK

7.1 Introduction

PyTerrier makes it easy to formulate learning to rank pipelines. Conceptually, learning to rank consists of three phases:

1. identifying a candidate set of documents for each query
2. computing extra features on these documents
3. using a learned model to re-rank the candidate documents to obtain a more effective ranking

PyTerrier allows each of these phases to be expressed as transformers, and for them to be composed into a full pipeline.

In particular, conventional retrieval transformers (such as *pt.BatchRetrieve*) can be used for the first phase. To permit the second phase, PyTerrier data model allows for a *"features"* column to be associated to each retrieved document. Such features can be generated using specialised transformers, or by combining other re-ranking transformers using the ** feature-union operator; Lastly, to facilitate the final phase, we provide easy ways to integrate PyTerrier pipelines with standard learning libraries such as sklearn, XGBoost and LightGBM.

In the following, we focus on the second and third phases, as well as describe ways to assist in conducting learning to rank experiments.

7.2 Calculating Features

7.2.1 Feature Union (**)

PyTerrier's main way to faciliate calculating extra features is through the ** operator. Consider an example where the candidate set should be identified using the BM25 weighting model, and then additional features computed using the Tf and PL2 models:

```
bm25 = pt.BatchRetrieve(index, wmodel="BM25")
tf = pt.BatchRetrieve(index, wmodel="Tf")
pl2 = pt.BatchRetrieve(index, wmodel="PL2")
pipeline = bm25 >> (tf ** pl2)
```

The output of the bm25 ranker would look like:

	qid	docno	score
1	q1	d5	(bm25 score)

Application of the feature-union operator (**) ensures that *tf* and *pl2* operate as *re-rankers*, i.e. they are applied only on the documents retrieved by *bm25*. For each document, the score calculate by *tf* and *pl2* are combined into the *"features"* column, as follows:

	qid	docno	score	features
1	q1	d5	(bm25 score)	[tf score, pl2 score]

7.2.2 FeaturesBatchRetrieve

When executing the pipeline above, the re-ranking of the documents again can be slow, as each separate BatchRetrieve object has to re-access the inverted index. For this reason, PyTerrier provides a class called FeaturesBatchRetrieve, which allows multiple query dependent features to be calculated at once, by virtue of Terrier's Fat framework.

`class pyterrier.`**`FeaturesBatchRetrieve`**(*index_location*, *features*, *controls=None*, *properties=None*, *threads=1*, ***kwargs*)

Use this class for retrieval with multiple features

Init method

Parameters

- **`index_location`** – An index-like object - An Index, an IndexRef, or a String that can be resolved to an IndexRef
- **`features`** (*`list`*) – List of features to use
- **`controls`** (*`dict`*) – A dictionary with the control names and values
- **`properties`** (*`dict`*) – A dictionary with the property keys and values
- **`verbose`** (*`bool`*) – If True transform method will display progress
- **`num_results`** (*`int`*) – Number of results to retrieve.

`transform`(*queries*)

Performs the retrieval with multiple features

Parameters **`queries`** – String for a single query, list of queries, or a pandas.Dataframe with columns=['qid', 'query']. For re-ranking, the DataFrame may also have a 'docid' and or 'docno' column.

Returns pandas.DataFrame with columns=['qid', 'docno', 'score', 'features']

An equivalent pipeline to the example above would be:

```
#pipeline = bm25 >> (tf ** pl2)
pipeline = pt.FeaturesBatchRetrieve(index, wmodel="BM25", features=["WMODEL:Tf",
↪"WMODEL:PL2"]
```

7.2.3 Apply Functions

If you have a way to calculate one or multiple ranking features at once, you can use pt.apply functions to create your feature sets. See the *pyterrier.apply - Custom Transformers* for examples. Functions created by pt.apply can be combined using the ** operator.

7.3 Learning

`pyterrier.ltr.`**`apply_learned_model`**`(`*learner*, *form='regression'*, ***kwargs*`)`

Results in a transformer that can take in documents that have a "features" column, and pass that to the specified learner via its transform() function, to obtain the documents' "score" column. Learners should follow the sklearn's general pattern with a fit() method (c.f. an sklearn Estimator) and a predict() method.

xgBoost and LightGBM are also supported through the use of *type='ltr'* kwarg.

Parameters

- **`learner`** – an sklearn-compatible estimator
- **`form`** (*`str`*) – either 'regression' or 'ltr'

Return type `TransformerBase`

The resulting transformer implements EstimatorBase, in other words it has a *fit()* method, that can be trained using training topics and qrels, as well as (optionally) validation topics and qrels. See also *EstimatorBase*.

7.3.1 SKLearn

A sklearn regressor can be passed directly to *pt.ltr.apply_learned_model()*:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=400)
rf_pipe = pipeline >> pt.ltr.apply_learned_model(rf)
rf_pipe.fit(train_topics, qrels)
pt.Experiment([bm25, rf_pipe], test_topics, qrels, ["map"], names=["BM25 Baseline", "LTR
↪"])
```

Note that if the feature definitions in the pipeline change, you will need to create a new instance of *rf*.

For analysis purposes, the feature importances identified by RandomForestRegressor can be accessed through *rf.features_importances_* - see the relevant sklearn documentation for more information.

7.3.2 Gradient Boosted Trees & LambdaMART

Both XGBoost and LightGBM provide gradient boosted regression tree and LambdaMART implementations. These support a sklearn-like interface that is supported by PyTerrier by supplying *form="ltr"* kwarg to *pt.ltr.apply_learned_model()*:

```
import xgboost as xgb
# this configures XGBoost as LambdaMART
lmart_x = xgb.sklearn.XGBRanker(objective='rank:ndcg',
      learning_rate=0.1,
      gamma=1.0,
```

(continues on next page)

(continued from previous page)

```
      min_child_weight=0.1,
      max_depth=6,
      verbose=2,
      random_state=42)

lmart_x_pipe = pipeline >> pt.ltr.apply_learned_model(lmart_x, form="ltr")
lmart_x_pipe.fit(train_topics, train_qrels, validation_topics, validation_qrels)

import lightgbm as lgb
# this configures LightGBM as LambdaMART
lmart_l = lgb.LGBMRanker(task="train",
    min_data_in_leaf=1,
    min_sum_hessian_in_leaf=100,
    max_bin=255,
    num_leaves=7,
    objective="lambdarank",
    metric="ndcg",
    ndcg_eval_at=[1, 3, 5, 10],
    learning_rate= .1,
    importance_type="gain",
    num_iterations=10)
lmart_l_pipe = pipeline >> pt.ltr.apply_learned_model(lmart_l, form="ltr")
lmart_l_pipe.fit(train_topics, train_qrels, validation_topics, validation_qrels)

pt.Experiment(
    [bm25, lmart_x_pipe, lmart_l_pipe],
    test_topics,
    test_qrels,
    ["map"],
    names=["BM25 Baseline", "LambdaMART (xgBoost)", "LambdaMART (LightGBM)" ]
)
```

Note that if the feature definitions in the pipeline change, you will need to create a new instance of XGBRanker (or LGBMRanker, as appropriate).

In our experience, LightGBM *tends* to be more effective than xgBoost.

Similar to sklearn, both XGBoost and LightGBM provide feature importances via *lmart_x.features_importances_* and *lmart_l.features_importances_*.

7.3.3 FastRank: Coordinate Ascent

We now support FastRank for learning models:

```
!pip install fastrank
import fastrank
train_request = fastrank.TrainRequest.coordinate_ascent()
params = train_request.params
params.init_random = True
params.normalize = True
params.seed = 1234567
```

(continues on next page)

(continued from previous page)

```
ca_pipe = pipeline >> pt.ltr.apply_learned_model(train_request, form="fastrank")
ca_pipe.fit(train_topics, train_qrels)
```

FastRank provides two learners: a random forest implementation (*fastrank.TrainRequest.random_forest()*) and coordinate ascent (*fastrank.TrainRequest.coordinate_ascent()*), a linear model.

7.4 Working with Features

We provide additional transformations functions to aid the analysis of learned model, for instance, removing (ablating) features from a complex ranking pipeline.

`pyterrier.ltr.`**`ablate_features`**`(`*fids*`)`

Ablates features (sets feature value to 0) from a pipeline. This is useful for performing feature ablation studies, whereby a feature is removed from the pipeline before learning.

Parameters **`fids`** – one or a list of integers corresponding to features indices to be removed

Return type `TransformerBase`

Example:

```
# assume pipeline is a retrieval pipeline that produces four ranking features
numf=4
rankers = []
names = []
# learn a model for all four features
full = pipeline >> pt.ltr.apply_learned_model(RandomForestRegressor(n_estimators=400))
full.fit(trainTopics, trainQrels, validTopics, validQrels)
rankers.append(full)

# learn a model for 3 features, removing one each time
for fid in range(numf):
    ablated = pipeline >> pt.ltr.ablate_features(fid) >> pt.ltr.apply_learned_
↪model(RandomForestRegressor(n_estimators=400))
    ablated.fit(trainTopics, trainQrels, validTopics, validQrels)
    rankers.append(ablated)

# evaluate the full (4 features) model, as well as the each model containing only 3␣
↪features)
pt.Experiment(
    rankers,
    test_topics,
    test_qrels,
    ["map"],
    names=["Full Model"]  + ["Full Minus %d" % fid for fid in range(numf)
)
```

`pyterrier.ltr.`**`keep_features`**`(`*fids*`)`

Reduces the features in a pipeline to only those mentioned. This is useful for performing feature ablation studies, whereby only some features are kept (and other removed) from a pipeline before learning occurs.

Parameters **`fids`** – one or a list of integers corresponding to the features indice to be kept

Return type `TransformerBase`

`pyterrier.ltr.feature_to_score(fid)`

Applies a specified feature for ranking. Useful for evaluating which of a number of pre-computed features are useful for ranking.

Parameters **fid** – a single feature id that should be kept

Return type `TransformerBase`

`pyterrier.ltr.score_to_feature()`

Takes the document's "score" from the score attribute, and uses it as a single feature. In particular, a feature union operator does not use any score of the documents in the candidate set as a ranking feaure. Using the resulting transformer within a feature-union means that an additional ranking feature is added to the "feature" column.

Example:

```
cands = pt.BatchRetrieve(index, wmodel="BM25")
bm25f = pt.BatchRetrieve(index, wmodel="BM25F")
pl2f = pt.BatchRetrieve(index, wmodel="PL2F")

two_features = cands >> (bm25f  **  pl2f)
three_features = cands >> (bm25f  **  pl2f ** pt.ltr.score_to_feature())
```

Return type `TransformerBase`

CHAPTER

EIGHT

PYTERRIER DATA MODEL

Pyterrier allows the chaining of different transformers in different manners. Each transformer has a `transform()` method, which takes as input a Pandas dataframe, and returns a dataframe also.

Different transformers change the dataframes in different ways - for instance, retrieving documents, or rewriting queries.

We define different dataframe "types" of data frame - the "primary key" is emphasised.

Data Type	Required Columns	Description
Q	`["qid", "query"]`	A set of queries
D	`["docno"]`	A set of documents
R	`["qid", "docno", "score", "rank"]`	A ranking of documents for each query
R w/ features	`["qid", "docno", "score", "rank", "features"]`	A numpy array of features for each document

8.1 1. Queries (Q)

A dataframe with two columns, uniquely identified by qid:

- *qid*: A unique identified for each query (primary key)
- query: The textual form of each query

Different transformers may have additional columns, but none are currently implemented.

An example dataframe with one query might be constructed as:

```
pd.DataFrame([["q1", "a query"]], columns=["qid", "query"])
```

or `pt.new.ranked_documents()`

```
pt.new.queries(["a query"], qid=["q1"])
```

When a query has been rewritten, for instance by applying the sequential dependence model or query expansion, the previous formulation of the query is available under the "query_0" attribute.

8.2 2. Set of documents (D)

A dataframe with columns:

- *docno*: The unique idenitifier for each document (primary key)

There might be other attributes such as text

8.3 3. Ranked Documents (R)

A dataframe representing which documents are retrieved and scored for a given query. This dataframe type has various columns that clearly inspired by the TREC results format:

- *qid*: A unique identifier for each query (primary key)
- query: The textual form of each query
- *docno*: The unique idenitifier for each document (primary key)
- score: The score for each document for that query
- rank: The rank of the document for that query

Note that rank is computed by sorting by qid ascending, then score descending. The first rank for each query is 0. The `pyterrier.model.add_rank()` function is used for adding the rank column.

Optional columns might support additional transformers, such as text (for the contents of the documents), url or title columns. Their presence can facilitate more advanced transformers, such as BERT-based transformers which operate on the raw text of the documents. For instance, if the Terrier index has additional metadata attributes, these can be included by BatchRetrieve using the `metadata` kwarg, i.e. `pt.BatchRetrieve(index, metadata=["docno", "title", "body"])`.

Note that the retrieved documents is a subset of the cartesian product of documents and queries; it is important that the query (text) attribute is present for at least ONE document rather than all documents for a given query.

An example dataframe with two documents might be constructed as:

```
pd.DataFrame([["q1", "a query", "d5", 5.2, 9], ["q1", None, "d10", 4.9, 1]], columns=[
↪"qid", "query", "docno", "score", "rank")
```

or using `pt.new.ranked_documents()`:

```
pt.new.ranked_documents([[5.2, 4.9]], qid=["q1"], docno=[["d5", "d10"]])
```

8.4 4. Set of documents with features

A dataframe with more columns, clearly inspired by the TREC results format:

- *qid*: A unique identifier for each query (primary key)
- query: The textual form of each query
- *docno*: The unique idenitifier for each document (primary key)
- features: A Numpy array of feature scores for each document

These may optionally have been ranked by a score attribute.

CHAPTER

NINE

PYTERRIER TRANSFORMERS

PyTerrier's retrieval architecture is based on three concepts:

- dataframes with pre-defined types (each with a minimum set of known attributes), as detailed in the data model.
- the *transformation* of those dataframes by standard information retrieval operations, defined as transformers.
- the compsition of transformers, supported by the operators defined on transformers.

In essence, a PyTerrier transformer is a class with a `transform()` method, which takes as input a dataframe, and changes it, before returning it.

Input	Output	Cardinality	Example	Concrete Transformer Example
Q	Q	1 to 1	Query rewriting	*pt.rewrite.SDM()*
Q	Q x D	1 to N	Retrieval	*pt.BatchRetrieve()*
Q x D	Q	N to 1	Query expansion	*pt.rewrite.RM3()*
Q x D	Q x D	1 to 1	Re-ranking	*pt.apply.doc_score()*
Q x D	Q x Df	1 to 1	Feature scoring	*pt.FeaturesBatchRetrieve()*

9.1 Optimisation

Some operators applied to transformer can be optimised by the underlying search engine - for instance, cutting a ranking earlier. So while the following two pipelines are semantically equivalent, the latter might be more efficient:

```
pipe1 = BatchRetrieve(index, "BM25") % 10
pipe2 = pipe1.compile()
```

9.2 Fitting

When *fit()* is called on a pipeline, all estimators (transformers that also have a `fit()` method, as specified by *EstimatorBase*) within the pipeline are fitted, in turn. This allows one (or more) stages of learning to be integrated into a retrieval pipeline. See *Learning to Rank* for examples.

When calling fit on a composed pipeline (i.e. one created using the >> operator), this will will call `fit()` on any estimators within that pipeline.

9.3 Transformer base classes

9.3.1 Transformer

This class is the base class for all transformers.

class pyterrier.**Transformer**

name = 'Transformer'

Base class for all transformers. Implements the various operators >> + * | & as well as `search()` for executing a single query and `compile()` for rewriting complex pipelines into more simples ones.

static from_df(*input*, *uniform=False*)

Instantiates a transformer from an input dataframe. Some rows from the input dataframe are returned in response to a query on the `transform()` method. Depending on the value *uniform*, the dataframe passed as an argument to `transform()` can affect this selection.

If *uniform* is True, input will be returned in its entirety each time. If *uniform* is False, rows from input that match the qid values from the argument dataframe.

Return type *Transformer*

transform(*topics_or_res*)

Abstract method for all transformations. Typically takes as input a Pandas DataFrame, and also returns one.

Return type `DataFrame`

transform_iter(*input*)

Method that proesses an iter-dict by instantiating it as a dataframe and calling transform(). Returns the DataFrame returned by transform(). Used in the implementation of index() on a composed pipeline.

Return type `DataFrame`

transform_gen(*input*, *batch_size=1*, *output_topics=False*)

Method for executing a transformer pipeline on smaller batches of queries. The input dataframe is grouped into batches of batch_size queries, and a generator returned, such that transform() is only executed for a smaller batch at a time.

Parameters

- **input** (*DataFrame*) – a dataframe to process
- **batch_size** (*int*) – how many input instances to execute in each batch. Defaults to 1.

Return type `Iterator[DataFrame]`

search(*query*, *qid='1'*, *sort=True*)

Method for executing a transformer (pipeline) for a single query. Returns a dataframe with the results for the specified query. This is a utility method, and most uses are expected to use the transform() method passing a dataframe.

Parameters

- **query** (*str*) – String form of the query to run
- **qid** (*str*) – the query id to associate to this request. defaults to 1.
- **sort** (*bool*) – ensures the results are sorted by descending rank (defaults to True)

Example:

```
bm25 = pt.BatchRetrieve(index, wmodel="BM25")
res = bm25.search("example query")

# is equivalent to
queryDf = pd.DataFrame([["1", "example query"]], columns=["qid", "query"])
res = bm25.transform(queryDf)
```

Return type `DataFrame`

`compile()`
Rewrites this pipeline by applying of the Matchpy rules in rewrite_rules. Pipeline optimisation is discussed in the ICTIR 2020 paper on PyTerrier.

Return type *`Transformer`*

`parallel`(*N*, *backend='joblib'*)
Returns a parallelised version of this transformer. The underlying transformer must be "picklable".

Parameters

- **`N`** (*`int`*) – how many processes/machines to parallelise this transformer over.
- **`backend`** (*`str`*) – which multiprocessing backend to use. Only two backends are supported, 'joblib' and 'ray'. Defaults to 'joblib'.

Return type *`Transformer`*

`get_parameter`(*name*)
Gets the current value of a particular key of the transformer's configuration state. By default, this examines the attributes of the transformer object, using hasattr() and setattr().

`set_parameter`(*name*, *value*)
Adjusts this transformer's configuration state, by setting the value for specific parameter. By default, this examines the attributes of the transformer object, using hasattr() and setattr().

Moreover, by extending Transformer, all transformer implementations gain the necessary "dunder" methods (e.g. `__rshift__()`) to support the transformer operators (>>, + etc). NB: This class used to be called `pyterrier.transformer.TransformerBase`

9.3.2 EstimatorBase

This class exposes a `fit()` method that can be used for transformers that can be trained.

class `pyterrier.transformer.`**`EstimatorBase`**
This is a base class for things that can be fitted.

`fit`(*topics_or_res_tr*, *qrels_tr*, *topics_or_res_va*, *qrels_va*)
Method for training the transformer.

Parameters

- **`topics_or_res_tr`** (*`DataFrame`*) – training topics (usually with documents)
- **`qrels_tr`** (*`DataFrame`*) – training qrels
- **`topics_or_res_va`** (*`DataFrame`*) – validation topics (usually with documents)
- **`qrels_va`** (*`DataFrame`*) – validation qrels

The ComposedPipeline implements `fit()`, which applies the interimediate transformers on the specified training (and validation) topics, and places the output into the `fit()` method of the final transformer.

9.3.3 Internal transformers

A significant number of transformers are defined in pyterrier.transformer to implement operators etc. Its is not expected to use these directly but they are documented for completeness.

Symbol	Name	Implementing transformer
>>	compose/then	ComposedPipeline
|	set-union	SetUnionTransformer
&	set-intersection	SetIntersectionTransformer
+	linear	CombSumTransformer
+	scalar-product	ScalarProductTransformer
%	rank-cutoff	RankCutoffTransformer
**	feature-union	FeatureUnionPipeline
^	concatenate	ConcatenateTransformer
~	cache	ChestCacheTransformer

9.3.4 Indexing Pipelines

Transformers can be chained to create indexing pipelines. The last element in the chain is assumed to be an indexer like IterDictIndexer - it should implement an `index()` method like IterDictIndexerBase. For instance:

```
docs = [ {"docno" : "1", "text" : "a" } ]
indexer = pt.text.sliding() >> pt.IterDictIndexer()
indexer.index(docs)
```

This is implemented by several methods:

- The last stage of the pipeline should have an `index()` method that accepts an iterable of dictionaries
- ComposedPipeline has a special `index()` method that breaks the input iterable into chunks (the size of chunks can be altered by a batch_size kwarg) and passes those through the intermediate pipeline stages (i.e. all but the last).
- In the intermediate pipeline stages, the `transform_iter()` method is called - by default this instantiates a DataFrame on batch_size records, which is passed to `transform()`.
- These are passed to `index()` of the last pipeline stage.

9.4 Writing your own transformer

The first step to writing your own transformer for your own code is to consider the type of change being applied. Several common transformations are supported through the functions in the *pyterrier.apply - Custom Transformers* module. See the *pyterrier.apply - Custom Transformers* documentation.

However, if your transformer has state, such as an expensive model to be loaded at startup time, you may want to extend `pt.Transformer` directly.

Here are some hints for writing Transformers:

- Except for an indexer, you should implement a `transform()` method.
- If your approach ranks results, use `pt.model.add_ranks()` to add the rank column.
- If your approach can be trained, your transformer should extend EstimatorBase, and implement the `fit()` method.

- If your approach is an indexer, your transformer should extend IterDictIndexerBase and implement `index()` method.

9.5 Mocking Transformers from DataFrames

You can make a Transformer object from dataframes. For instance, a unifom transformer will always return the input dataframe any time `transform()` is called:

```
df = pt.new.ranked_documents([[1,2]])
uniformT = pt.Transformer.from_df(df, uniform=True)
# uniformT.transform() always returns df, regardless of arguments
```

You can also create a Transformer object from existing results, e.g. saved on disk using `pt.io.write_results()` etc. The resulting "source transformer" will return all results by matching on the qid of the input:

```
res = pt.io.read_results("/path/to/baseline.res.gz")
baselineT = pt.Transformer.from_df(df, uniform=True)

Q1 = pt.new.queries("test query", qid="Q1")
resQ1 = baselineT.transform(Q1)
```

CHAPTER

TEN

OPERATORS ON TRANSFORMERS

Part of the power of PyTerrier comes from the ease in which researchers can formulate complex retrieval pipelines. This is made possible by the operators available on Pyterrier's transformer objects. The following table summarises the available operators:

Operator	Meaning
>>	Then - chaining pipes
+	Linear combination of scores
*	Scalar factoring of scores
&	Document Set Intersection
|	Document Set Union
%	Apply rank cutoff
^	Concatenate run with another
**	Feature Union
~	Cache transformer result

NB: These operators retain their default Python operator precedence - that may not be aligned with your expectations in a PyTerrier context (e.g. & is higher than >>).

10.1 Then (>>)

Apply one transformation followed by another:

```
#rewrites topics to include #1 etc
sdm = pt.rewrite.SDM()
br = BatchRetrieve(index, "DPH")

res = br.transform( sdm.transform(topics))
```

We use >> as a shorthand for then (also called compose):

```
res = (sdm >> br).transform(topics)
```

Example:

Consider a topics dataframe as follows:

qid	query
q1	test query

Then the application of SDM() would produce:

qid	query
q1	test query #1(test query) #uw8(test query)

NB: In practice the query reformulation generated by SDM() is more complex, due to the presence of weights etc in the resulting query.

Then the final res dataframe would contain the results of applying BatchRetrieve on the rewritten queries, as follows:

qid	query	docno	score	rank
q1	test query #1(test query) #uw8(test query)	d10	4	0
q1	test query #1(test query) #uw8(test query)	d04	3.8	1

NB: Then can also be used for retrieval and re-ranking pipelines, such as:

```
pipeline = BatchRetrieve(index, "DPH") >> BatchRetrieve(index, "BM25")
```

10.2 Linear Combine and Scalar Factor (+, *)

The linear combine (+) and scalar factor (*) operators allow the scores of different retrieval systems to be linearly combined (with weights).

Instead of the following Python:

```
br_DPH = BatchRetrieve(index, "DPH")
br_BM25 = BatchRetrieve(index, "BM25")

res1 = br_DPH.transform(topics)
res2 = br_BM25.transform(topics)
res = res1.merge(res2, on=["qid", "docno"])
res["score"] = 2 * res["score_x"] + res["score_y"]
```

We use binary + and * operators. This is natural, as it is intuitive to combine weighted retrieval functions using + and *

```
br_DPH = BatchRetrieve(index, "DPH")
br_BM25 = BatchRetrieve(index, "BM25")
res = (2* br_DPH + br_BM25).transform(topics)
```

If the DPH and BM25 transformers would respectively return:

qid	docno	score	rank
q1	d10	2	0
q1	d12	1	1

qid	docno	score	rank
q1	d10	4	0
q1	d01	3	1

then the application of the transformer represented by the expression *(2* br_DPH + br_BM25)* would be:

qid	docno	score	rank
q1	d10	8	0
q1	d01	3	1
q1	d12	2	2

NB: Any documents not present in one of the constituent rankings will contribute a score of 0 to the final score of that document.

Precedence and Associativity

The + and * operators retain their classical precendence among Pythons operators. This means that the intended semantics of an expression of linear combinations and scalar factors are clear - indeed, * binds higher than +, so *2* br_DPH* + *br_BM25* is interpreted as *(2* br_DPH)* + *br_BM25*.

10.3 Set Intersection and Union (&, |)

The set that only includes documents that occur in the intersection (&) and union (|) of both retrieval sets. Scores and ranks are not returned - hence, the rankings documents would normally be re-scored:

```
BM25_br = BatchRetrieve(index, "BM25")
PL2_br = BatchRetrieve(index, "PL2")

res_intersection = (BM25_br & PL2_br).transform(topics)
res_union = (BM25_br | PL2_br).transform(topics)
```

Examples:

If the BM25 and PL2 pipelines would respectively return:

qid	docno	score	rank
q1	d10	4.3	0
q1	d12	4.1	1

qid	docno	score	rank
q1	d10	4.3	0
q1	d01	3.9	1

then the application of the set intersection operator (&) would result in a ranking only containing documents appear in both transformers:

qid	docno
q1	d10

and the application of the set union operator (|) would return documents retrieved by either transformer:

qid	docno
q1	d10
q1	d12
q1	d01

Note that, as these are set operators, there are no ranks and scores returned in the output.

10.4 Rank Cutoff (%)

The % operator is called rank cutoff, and limits the number of results for each query:

```
pipe1 = pt.BatchRetrieve(index, "BM25") % 2
```

Example:

If a retrieval pipeline returns:

qid	docno	score	rank
q1	d10	4.3	0
q1	d12	4.1	1
q1	d05	3.9	2
q1	d03	3.5	3
q1	d01	2.5	4

then the application of the rank cutoff operator would be:

qid	docno	score	rank
q1	d10	4.3	0
q1	d12	4.1	1

10.5 Concatenate (^)

Sometimes, we may only want to apply an expensive retrieval process on a few top-ranked documents, and fill up the rest of the ranking with the rest of the documents (removing duplicates). We can do that using the concatenate operator. Concretely, in the example below, *alldocs* is our candidate set, of say 1000 documents per query. We re-rank the top 3 documents for each query using *ExpensiveReranker()*, in a pipeline called *topdocs*. We then use the concatenate operator (^) to append the remaining documents from alldocs, such that they have scores and ranks adjusted to appear just after the documents obtained from the *topdocs* pipeline:

```
alldocs = BatchRetrieve(index, "BM25")
topdocs = alldocs % 3 >> ExpensiveReranker()
finaldocs = topdocs ^ alldocs
```

Example:

If *alldocs* returns:

qid	docno	score	rank
q1	d10	4.3	0
q1	d12	4.1	1
q1	d05	3.9	2
q1	d03	3.5	3
q1	d01	2.5	4

Then alldocs would compute scores on the top 3 ranked documents (d10, d12, d05). After applying ExpensiveReranker() to score and re-ranked these 3 documents, topdocs could be as follows:

qid	docno	score	rank
q1	d05	1.0	0
q1	d10	0.9	1
q1	d12	0.8	2

Then finaldocs would be:

qid	docno	score	rank
q1	d05	1.0	0
q1	d10	0.9	1
q1	d12	0.8	2
q1	d03	0.7999	3
q1	d01	-0.2001	4

Note that score of d03 is adjusted to appear just under the last ranked document from topdocs (we use a small value of epsilon=0.0001) as the minimum difference between the least ranked document from topdocs and the highest remaining document from alldocs. The relative ordering of documents from alldocs is unchanged, but the gaps between their scores are maintained, so the difference between d03 and d01 is a score delta of -1 in both alldocs and finaldocs.

10.6 Feature Union (**)

Here we take one system, e.g. DPH, to get an initial candidate set, then add more systems as features.

The Python would have looked like:

```
sample_br = BatchRetrieve(index, "DPH")
BM25F_br = BatchRetrieve(index, "BM25F")
PL2F_br = BatchRetrieve(index, "PL2F")

sampleRes = sample_br.transform(topics)
# assumes sampleRes contains the queries
BM25F_res = BM25F_br.transform(sampleRes)
PL2F_res = PL2F_br.transform(sampleRes)

final_res = BM25F_res.join(PL2F_res, on=["qid", "docno"])
final_res["features"] = np.stack(final_res["features_x"], final_res["features_y"])
```

Instead, we use ** to denote feature union:

```
sample_br = BatchRetrieve(index, "DPH")
BM25F_br = BatchRetrieve(index, "BM25F")
PL2F_br = BatchRetrieve(index, "PL2F")

# ** is the feature union operator. It requires a candidate document set as input
(BM25F_br ** PL2F_br)).transform(sample_br.transform(topics))
# or combined with the then operator, >>
(sample_br >> (BM25F_br ** PL2F_br)).transform(topics)
```

NB: Feature union expects the documents being returned by each side of the union to be identical. It will produce a warning if they are not identical. Documents not returned will obtain a score of 0 for that feature.

Example:

For example, consider that sample_br returns a ranking as follows:

qid	docno	score	rank
q1	d10	4.3	0

Further, for document d10, BM25F and PL2F return scores respectively of 4.9 and 13.0. The application of the feature union operator above would be a ranking with features as follows:

qid	docno	score	rank	features
q1	d10	4.3	0	[4.9, 13.0]

More examples of feature union can be found in the learning-to-rank documentation (*Learning to Rank*).

Precedence and Associativity

Feature union is associative, so in the following examples, *x1*, *x2* and *x3* have identical semantics:

```
x1 = sample_br >> ( BM25F_br ** PL2F_br ** urllen)
x2 =  sample_br >> ( (BM25F_br ** PL2F_br) ** urllen)
x3 =  sample_br >> ( BM25F_br ** (PL2F_br ** urllen))
```

Pipelines *x1*, *x2* and *x3* are all pipelines that create **identical** document rankings with three features, in the precise order BM25F, PL2F and urllength.

Note that >> has higher operator precendence in Python than **. For this reason, feature unions usually need to be expressed in parentheses. In this way the semantics of pipelines *a*, *b* and *c* in the example below are not identical, and indeed, *a* is parsed like *b*, while *c* is almost always the desired outcome:

```
# a is parsed in the same way as b, when the likely desired parse was c
a = sample_br >> BM25F_br ** PL2F_br
b = (sample_br >> BM25F_br) ** PL2F_br)
c = sample_br >> ( BM25F_br ** PL2F_br)
```

10.7 Caching (~)

Some transformers are expensive to apply. For instance, we might find ourselves repeatedly running our BM25 baseline. We can request Pyterrier to _cache_ the outcome of a transformer for a given qid by using the unary ~ operator.

Consider the following example:

```
from pyterrier import BatchRetrieve, Experiment
firstpass = BatchRetrieve(index, "BM25")
reranker = ~firstpass >> BatchRetrieve(index, "BM25F")
Experiment([~firstpass, ~reranker], topics, qrels)
```

In this example, *firstpass* is cached when it is used in the Experiment evaluation, as well as when it is used in the reranker. We also cache the outcome of the Experiment, so that another evaluation will be faster.

By default, Pyterrier caches results to *~/.pyterrier/transformer_cache/*.

CHAPTER

ELEVEN

EXAMPLES OF RETRIEVAL PIPELINES

11.1 Query Rewriting

11.1.1 Sequential Dependence Model

```
pipe = pt.rewrite.SDM() >> pt.BatchRetrieve(indexref, wmodel="BM25")
```

Note that the SDM() rewriter has a number of constructor parameters:

- `remove_stopwords` - defines if stopwords should be removed from the query
- `prox_model` - change the proximity model. For true language modelling, you should set `prox_model` to “org.terrier.matching.models.Dirichlet_LM”

11.1.2 Divergence from Randomness Query Expansion

A simple QE transformer can be achieved using

```
qe = pt.BatchRetrieve(indexref, wmodel="BM25", controls={"qe" : "on"})
```

As this is pseudo-relevance feedback in nature, it identifies a set of documents, extracts informative term in the top-ranked documents, and re-exectutes the query.

However, more control can be achieved by using the QueryExpansion transformer separately, as thus:

```
qe = (pt.BatchRetrieve(indexref, wmodel="BM25") >>
    pt.rewrite.QueryExpansion(indexref) >>
    pt.BatchRetrieve(indexref, wmodel="BM25")
)
```

The QueryExpansion() object has the following constructor parameters:

- `index_like` - which index you are using to obtain the contents of the documents. This should match the preceeding BatchRetrieve.
- `fb_docs` - number of feedback documents to examine
- `fb_terms` - number of feedback terms to add to the query

Note that different indexes can be used to achieve query expansion using an external collection (sometimes called collection enrichment or external feedback). For example, to expand queries using Wikipedia as an external resource, in order to get higher quality query re-weighted queries, would look like this:

```
pipe = (pt.BatchRetrieve(wikipedia_index, wmodel="BM25") >>
    pt.rewrite.QueryExpansion(wikipedia_index) >>
    pt.BatchRetrieve(local_index, wmodel="BM25")
)
```

11.1.3 RM3 Query Expansion

We also provide RM3 query expansion, by virtue of an external plugin to Terrier called terrier-prf. This needs to be load at initialisation time.

```
pt.init(boot_packages=["com.github.terrierteam:terrier-prf:-SNAPSHOT"])
pipe = (pt.BatchRetrieve(indexref, wmodel="BM25") >>
    pt.rewrite.RM3(indexref) >>
    pt.BatchRetrieve(indexref, wmodel="BM25")
)
```

11.2 Combining Rankings

Sometimes we have good retrieval approaches and we wish to combine these in a unsupervised manner. We can do that using the linear combination operator:

```
bm25 = pt.BatchRetrieve(indexref, wmodel="BM25")
dph = pt.BatchRetrieve(indexref, wmodel="DPH")
linear = bm25_cands + dph_cands
```

Of course, some weighting can help:

```
bm25 = pt.BatchRetrieve(indexref, wmodel="BM25")
dph = pt.BatchRetrieve(indexref, wmodel="DPH")
linear = bm25_cands + 2* dph_cands
```

However, if the score distributions are not similar, finding a good weight can be tricky. Normalisation of retrieval scores can be advantagous in this case. We provide PerQueryMaxMinScoreTransformer() to make easy normalisation.

```
bm25 = pt.BatchRetrieve(indexref, wmodel="BM25") >> pt.pipelines.
↪PerQueryMaxMinScoreTransformer()
dph = pt.BatchRetrieve(indexref, wmodel="DPH" >> pt.pipelines.
↪PerQueryMaxMinScoreTransformer()
linear = 0.75 * bm25_cands + 0.25 * dph_cands
```

11.3 Learning to Rank

Having shown some of the main formulations, lets show how to build different formulations into a LTR model.

- Some authors report that it is useful to take a union of different retrieval mechanisms in order to build a good candidate set. We use the set-union operator here to combine the rankings of BM25 and DPH weighting models.
- We then score each of the retrieved documents

```
bm25_cands = pt.BatchRetrieve(indexref, wmodel="BM25")
dph_cands = pt.BatchRetrieve(indexref, wmodel="DPH")
all_cands = bm25_cands | dph_cands

all_features = all_cands >> (
    pt.BatchRetrieve(indexref, wmodel="BM25F") **
    pt.rewrite.SDM() >> pt.BatchRetrieve(indexref, wmodel="BM25")
    )

import xgboost as xgb
params = {'objective': 'rank:ndcg',
          'learning_rate': 0.1,
          'gamma': 1.0, 'min_child_weight': 0.1,
          'max_depth': 6,
          'verbose': 2,
          'random_state': 42
         }
lambdamart = pt.ltr.apply_learned_model(xgb.sklearn.XGBRanker(**params), form='ltr')
final_pipe = all_features >> lambdamart
final_pipe.fit(tr_topics, tr_qrels, va_topics, va_qrels)
```

CHAPTER

TWELVE

WORKING WITH DOCUMENT TEXTS

Many modern retrieval techniques are concerned with operating directly on the text of documents. PyTerrier supports these forms of interactions.

12.1 Indexing and Retrieval of Text in Terrier indices

If you are using a Terrier index for your first-stage ranking, you will want to record the text of the documents in the MetaIndex. The following configuration demonstrates saving the title and remainder of the documents separately in the Terrier index MetaIndex when indexing a TREC-formatted corpus:

```
files = []  # list of filenames to be indexed
indexer = pt.TRECCollectionIndexer(INDEX_DIR,
    # record that we save additional document metadata called 'text'
    meta= {'docno' : 26, 'text' : 2048},
    # The tags from which to save the text. ELSE is special tag name, which means␣
↪anything not consumed by other tags.
    meta_tags = {'text' : 'ELSE'}
    verbose=True)
indexref = indexer.index(files)
index = pt.IndexFactory.of(indexref)
```

On the other-hand, for a TSV-formatted corpus such as MSMARCO passages, indexing is easier using IterDictIndexer:

```
def msmarco_generate():
    dataset = pt.get_dataset("trec-deep-learning-passages")
    with pt.io.autoopen(dataset.get_corpus()[0], 'rt') as corpusfile:
        for l in corpusfile:
            docno, passage = l.split("\t")
            yield {'docno' : docno, 'text' : passage}

iter_indexer = pt.IterDictIndexer("./passage_index")
indexref = iter_indexer.index(msmarco_generate(), meta={'docno' : 20, 'text': 4096})
```

During retrieval you will need to have the text stored as an attribute in your dataframes.

This can be achieved in one of several ways:

- requesting document metadata when using *BatchRetrieve*
- adding document metadata later using *get_text()*

BatchRetrieve accepts a *metadata* keyword-argument which allows for additional metadata attributes to be retrieved.

Alternatively, the *pt.text.get_text()* transformer can be used, which can extract metadata from a Terrier index or IRDSDataset for documents already retrieved. The main advantage of using IRDSDataset is that it supports all document fields, not just those that were included as meta fields when indexing.

Examples:

```
# the following pipelines are equivalent
pipe1 = pt.BatchRetrieve(index, metadata=["docno", "body"])

pipe2 = pt.BatchRetrieve(index) >> pt.text.get_text(index, "body")

dataset = pt.get_dataset('irds:vaswani')
pipe3 = pt.BatchRetrieve(index) >> pt.text.get_text(dataset, "text")
```

`pyterrier.text.`**`get_text`**(*indexlike*, *metadata='body'*, *by_query=False*, *verbose=False*)

A utility transformer for obtaining the text from the text of documents (or other document metadata) from Terrier's MetaIndex or an IRDSDataset docstore.

Parameters

- **indexlike** – a Terrier index or IRDSDataset to retrieve the metadata from
- **metadata** (*list(str) or str*) – a list of strings of the metadata keys to retrieve from the index. Defaults to ["body"]
- **by_query** (*bool*) – whether the entire dataframe should be progressed at once, rather than one query at a time. Defaults to false, which means that all document metadata will be fetched at once.
- **verbose** (*bool*) – whether to print a tqdm progress bar. Defaults to false. Has no effect when by_query=False

Example:

```
pipe = ( pt.BatchRetrieve(index, wmodel="DPH")
    >> pt.text.get_text(index)
    >> pt.text.scorer(wmodel="DPH") )
```

Return type `TransformerBase`

12.2 Scoring query/text similarity

`pyterrier.text.`**`scorer`**(**args*, ***kwargs*)

This allows scoring of the documents with respect to a query, without creating an index first. This is an alias to pt.TextScorer(). Internally, a Terrier memory index is created, before being used for scoring.

Example:

```
df = pd.DataFrame(
    [
        ["q1", "chemical reactions", "d1", "professor protor poured the chemicals"],
        ["q1", "chemical reactions", "d2", "chemical brothers turned up the beats"],
    ], columns=["qid", "query", "docno", "text"])
textscorerTf = pt.text.scorer(body_attr="text", wmodel="Tf")
rtr = textscorerTf.transform(df)
```

(continues on next page)

(continued from previous page)

```
# rtr will have a score for each document for the query "chemical reactions" based␣
↪on the provided document contents
# both attain score 1, as, after stemming, they both contain one occurrence of the␣
↪query term 'chemical'
# ["q1", "chemical reactions", "d1", "professor protor poured the chemicals", 0, 1]
# ["q1", "chemical reactions", "d2", "chemical brothers turned up the beats", 0, 1]
```

For calculating the scores of documents using any weighting model with the concept of IDF, it may be useful to make use of an existing Terrier index for background statistics:

```
textscorerTfIdf = pt.text.scorer(body_attr="text", wmodel="TF_IDF", background_
↪index=index)
```

Return type `TransformerBase`

Other text scorers are available in the form of neural re-rankers - separate to PyTerrier, see *Neural Rankers and Rerankers*.

12.3 Working with Passages rather than Documents

As documents are long, relevant content may only be found in a small portion of the document. Moreover, some models are more suited to operating on small parts of the document. For this reason, passage-based retrieval techniques have been conceived. PyTerrier supports the creation of passages from longer documents, and for the aggregation of scores from these passages.

`pyterrier.text.`**`sliding`**(*text_attr='body'*, *length=150*, *stride=75*, *join=' '*, *prepend_attr='title'*, ***kwargs*)

A useful transformer for splitting long documents into smaller passages within a pipeline. This applies a *sliding* window over the text, where each passage is the give number of tokens long. Passages can overlap, if the stride is set smaller than the length. In applying this transformer, docnos are altered by adding '%p' and a passage number. The original scores for each document can be recovered by aggregation functions, such as *max_passage()*.

For the puposes of obtaining passages of a given length, tokenisation is perfomed simply by splitting on one-or-more spaces, i.e. based on the Python regular expression `re.compile(r'\s+')`.

Parameters

- **`text_attr`** (*str*) – what is the name of the dataframe attribute containing the main text of the document to be split into passages. Default is 'body'.
- **`length`** (*int*) – how many tokens in each passage. Default is 150.
- **`stride`** (*int*) – how many tokens to advance each passage by. Default is 75.
- **`prepend_attr`** (*str*) – whether another document attribute, such as the title of the document, to each passage, following [Dai2019]. Defaults to 'title'.
- **`title_attr`** (*str*) – what is the name of the dataframe attribute containing the title the document to be split into passages. Default is 'title'. Only used if prepend_title is set to True.

Example:

```
pipe = ( pt.BatchRetrieve(index, wmodel="DPH", metadata=["docno", "body"])
    >> pt.text.sliding(length=128, stride=64, prepend_attr=None)
```

(continues on next page)

(continued from previous page)

```
    >> pt.text.scorer(wmodel="DPH")
    >> pt.text.max_passage() )
```

Return type `TransformerBase`

Example Inputs and Outputs:

Consider the following dataframe with one or more documents:

qid	docno	text
q1	d1	a b c d

The result of applying *pyterrier.text.sliding(length=2, stride=1, prepend_title=False)* would be:

qid	docno	text
q1	d1%p1	a b
q1	d1%p2	b c
q1	d1%p3	c d

`pyterrier.text.`**`max_passage`**`()`

Scores each document based on the maximum score of any constituent passage. Applied after a sliding window transformation has been scored.

Return type `TransformerBase`

`pyterrier.text.`**`first_passage`**`()`

Scores each document based on score of the first passage of that document. Note that this transformer is rarely used in conjunction with the sliding window transformer, as all passages would required to be scored, only for the first one to be used.

Return type `TransformerBase`

`pyterrier.text.`**`mean_passage`**`()`

Scores each document based on the mean score of all constituent passages. Applied after a sliding window transformation has been scored.

Return type `TransformerBase`

`pyterrier.text.`**`kmaxavg_passage`**`(`*`k`*`)`

Scores each document based on the average score of the top scoring k passages. Generalises combination of mean_passage() and max_passage(). Proposed in [Chen2020].

Parameters **k** (*int*) – The number of top-scored passages for each document to use when scoring

Return type `TransformerBase`

12.3.1 Examples

Assuming that a retrieval pipeline such as *sliding()* followed by *scorer()* could return a dataframe that looks like this:

qid	docno	rank	score
q1	d1%p5	0	5.0
q1	d2%p4	1	4.0
q1	d1%p3	2	3.0
q1	d1%p1	3	1.0

The output of the *max_passage()* transformer would be:

qid	docno	rank	score
q1	d1	0	5.0
q1	d2	1	4.0

The output of the *mean_passage()* transformer would be:

qid	docno	rank	score
q1	d1	0	4.5
q1	d2	1	4.0

The output of the *first_passage()* transformer would be:

qid	docno	rank	score
q1	d2	0	4.0
q1	d1	1	1.0

Finally, the output of the *kmaxavg_passage(2)* transformer would be:

qid	docno	rank	score
q1	d2	1	4.0
q1	d1	0	1.0

12.4 Query-biased Summarisation (Snippets)

`pyterrier.text.snippets`(*text_scorer_pipe*, *text_attr='text'*, *summary_attr='summary'*, *num_psgs=5*, *joinstr='...'*)

Applies query-biased summarisation (snippet), by applying the specified text scoring pipeline.

Parameters

- **text_scorer_pipe** (`Transformer`) – the pipeline for scoring passages in response to the query. Normally this applies passaging.
- **text_attr** (*str*) – what is the name of the attribute that contains the text of the document
- **summary_attr** (*str*) – what is the name of the attribute that should contain the query-biased summary for that document
- **num_psgs** (*int*) – how many passages to select for the summary of each document
- **joinstr** (*str*) – how to join passages for a given document together

Example:

```
# retrieve documents with text
br = pt.BatchRetrieve(index, metadata=['docno', 'text'])

# use Tf as a passage scorer on sliding window passages
psg_scorer = (
    pt.text.sliding(text_attr='text', length=15, prepend_attr=None)
    >> pt.text.scorer(body_attr="text", wmodel='Tf', takes='docs')
)

# use psg_scorer for performing query-biased summarisation on docs retrieved by br
retr_pipe = br >> pt.text.snippets(psg_scorer)
```

Return type `None`

12.5 References

- [Chen2020] ICIP at TREC-2020 Deep Learning Track, X. Chen et al. Procedings of TREC 2020.
- [Dai2019] Deeper Text Understanding for IR with Contextual Neural Language Modeling. Z. Dai & J. Callan. Proceedings of SIGIR 2019.

CHAPTER

THIRTEEN

NEURAL RANKERS AND RERANKERS

PyTerrier is designed with for ease of integration with neural ranking models, such as BERT. In short, neural re-rankers that can take the text of the query and the text of a document can be easily expressed using an *pyterrier.apply - Custom Transformers* transformer.

More complex rankers (for instance, that can be trained within PyTerrier, or that can take advantage of batching to speed up GPU operations) typically require more complex integrations. We have separate repositories with integrations of well-known neural re-ranking plaforms (CEDR, ColBERT).

13.1 Indexing, Retrieval and Scoring of Text using Terrier

If you are using a Terrier index for your first-stage ranking, you will want to record the text of the documents in the MetaIndex. More of PyTerrier's support for operating on text is documented in *Working with Document Texts*.

13.2 Available Neural Dense Retrieval and Re-ranking Integrations

- OpenNIR has integration with PyTerrier - see its notebook examples.
- PyTerrier_ColBERT contains a ColBERT integration, including both a text-scorer and a end-to-end dense retrieval.
- PyTerrier_ANCE contains an ANCE integration for end-to-end dense retrieval.
- PyTerrier_T5 contains a monoT5 integration.
- PyTerrier_doc2query contains a docT5query integration.
- PyTerrier_DeepCT contains a DeepCT integration.
- The separate PyTerrier_BERT repository includes CEDR integration (including "vanilla" BERT models), as well as an earlier ColBERTPipeline integration.
- An initial BERT-QE integration is available.

The following gives an example ranking pipeline using ColBERT for re-ranking documents in PyTerrier. Long documents are broken up into passages using a sliding-window operation. The final score for each document is the maximum of any consitutent passages:

```
from pyterrier_bert.colbert import ColBERTPipeline

pipeline = DPH_br_body >> \
    pt.text.sliding() >> \
```

(continues on next page)

(continued from previous page)

```
    ColBERTPipeline("/path/to/checkpoint") >> \
    pt.text.max_passage()
```

13.3 Outlook

We continue to work on improving the integration of neural rankers and re-rankers within PyTerrier.

CHAPTER

FOURTEEN

ACCESSING TERRIER'S INDEX API

Once a Terrier index has been built, PyTerrier provides a number of ways to access it. In doing so, we access the standard Terrier index API, however, some types are patched by PyTerrier to make them easier to use.

NB: Examples in this document are also available as a Jupyter notebook:

- GitHub: https://github.com/terrier-org/pyterrier/blob/master/examples/notebooks/index_api.ipynb
- Google Colab: https://colab.research.google.com/github/terrier-org/pyterrier/blob/master/examples/notebooks/index_api.ipynb

14.1 Loading an Index

Terrier has IndexRef and Index objects, along with an IndexFactory class that allows an Index to be obtained from the IndexRef.

IndexRef is essentially a String that tells Terrier where the index is located. Tyically it is a file location, pointing to a data.properties file:

```
indexref = pt.IndexRef.of("/path/to/data.properties")
```

IndexRefs can also be obtained from a PyTerrier dataset:

```
indexref = dataset.get_index()
```

IndexRef objects can be directly passed to BatchRetrieve:

```
pt.BatchRetrieve(indexref).search("chemical reactions")
```

If you want to access the underlying data structures, you need to use IndexFactory, using the indexref, or the string location:

```
index = pt.IndexFactory.of(indexref)
#or
index = pt.IndexFactory.of("/path/to/data.properties")
```

NB: BatchRetrieve will accept anything "index-like", i.e. a string location of an index, an IndexRef or an Index.

14.2 Whats in an Index

An index has several data structures:

- the CollectionStatistics - the salient global statistics of the index (number of documents, etc).
- the Lexicon - consists of an entry for each unique term in the index, which contains the corresponding statistics of each term (frequency etc), and a pointer to the inverted index posting list for that term.
- the inverted index (a PostingIndex) - contains the posting list for each term, which records the documents that a given term appears in, and with what frequency for each document.
- the DocumentIndex - contains the length of the document (and other field lengths).
- the MetaIndex - contains document metadata, such as the docno, and optionally the raw text and the URL of each document.
- the direct index (also a PostingIndex) - contains a posting list for each document, detailing which terms occur in that document and with which frequency. The presence of the direct index depends on the IndexingType that has been applied - single-pass and some memory indices do not provide a direct index.

Each of these objects is available from the Index using a get method, e.g. *index.getCollectionStatistics()*. For instance, we can easily view the CollectionStatistics:

```
print(index.getCollectionStatistics())
Number of documents: 11429
Number of terms: 7756
Number of postings: 224573
Number of fields: 0
Number of tokens: 271581
Field names: []
Positions:   false
```

In this example, the indexed collection had 11429 documents, which contained 271581 word occurrences. 7756 unique words were identified. The total number of postings in the inverted index is 224573. This index did not record fields during indexing (which can be useful for models such as BM25F). Similarly, positions, which are used for phrasal queries or proximity models were not recorded.

We can check what metadata is recorded:

```
print(index.getMetaIndex().getKeys())
```

Usually, this will respond with *['docno']* - indeed docno is by convention the unique identifier for each document.

NB: Terrier's Index API is just that, an API of interfaces and abstract classes - depending on the indexing configuration, the exact implementation you will receive will differ.

14.3 Using a Terrier index in your own code

14.3.1 How many documents does term X occur in?

We use the Lexicon object, particularly the getLexiconEntry(String) method. However, PyTerrier aliases this, so lookup can be done like accessing a dictionary:

```
index.getLexicon()["chemic"].getDocumentFrequency()
```

As our index is stemmed, we used the stemmed form of the word 'chemical' which is 'chemic'

14.3.2 What is the un-smoothed probability of term Y occurring in the collection?

Here, we again use the Lexicon of the underlying Terrier index. We check that the term occurs in the lexicon (to prevent a KeyError). The Lexicon returns a LexiconEntry, which allows us access to the number of occurrences of the term in the index.

Finally, we use the CollectionStatistics object to determine the total number of occurrences of all terms in the index:

```
index.getLexicon()["chemic"].getFrequency() / index.getCollectionStatistics().
↪getNumberOfTokens() if "chemic" in index.getLexicon() else 0
```

14.3.3 What terms occur in the 11th document?

Here we use the direct index. We need a Pointer into the direct index, which we obtain from the DocumentIndex. PostingIndex.getPostings() is our method to get a posting list. Indeed, it returns an IterablePosting. Note that IterablePosting can be used in Python for loops:

```
di = index.getDirectIndex()
doi = index.getDocumentIndex()
lex = index.getLexicon()
docid = 10 #docids are 0-based
#NB: postings will be null if the document is empty
for posting in di.getPostings(doi.getDocumentEntry(docid)):
    termid = posting.getId()
    lee = lex.getLexiconEntry(termid)
    print("%s with frequency %d" % (lee.getKey(),posting.getFrequency()))
```

14.3.4 What documents does term "Z" occur in?

Here we use the inverted index (also a PostingIndex). The Pointer this time comes from the Lexicion, in that the LexiconEntry implements Pointer. Finally, we use the MetaIndex to lookup the docno corresponding to the docid:

```
meta = index.getMetaIndex()
inv = index.getInvertedIndex()

le = lex.getLexiconEntry( "chemic" )
# the lexicon entry is also our pointer to access the inverted index posting list
for posting in inv.getPostings( le ):
    docno = meta.getItem("docno", posting.getId())
    print("%s with frequency %d " % (docno, posting.getFrequency()))
```

14.3.5 What are the PL2 weighting model scores of documents that "Y" occurs in?

Use of a WeightingModel class needs some setup, namely the EntryStatistics of the term (obtained from the Lexicon, in the form of the LexiconEntry), as well as the CollectionStatistics (obtained from the index):

```
inv = index.getInvertedIndex()
meta = index.getMetaIndex()
lex = index.getLexicon()
le = lex.getLexiconEntry( "chemic" )
```

(continues on next page)

(continued from previous page)

```
wmodel = pt.autoclass("org.terrier.matching.models.PL2")()
wmodel.setCollectionStatistics(index.getCollectionStatistics())
wmodel.setEntryStatistics(le);
wmodel.setKeyFrequency(1)
wmodel.prepare()
for posting in inv.getPostings(le):
    docno = meta.getItem("docno", posting.getId())
    score = wmodel.score(posting)
    print("%s with score %0.4f"  % (docno, score))
```

Note that using BatchRetrieve or similar is probably an easier prospect for such a use case.

CHAPTER

FIFTEEN

PARALLELISATION

With large datasets, retrieval can sometimes take some time. To address this, PyTerrier Transformers can parallelised.

Each Transformer has a *.parallel()* method, which parallelises the transformer. Two backends are supported:

- *'joblib'* - uses multiple processes on your current machine. Resources such as indices will be opened multiple times on your machine. Joblib is the default backend for parallelisation in PyTerrier.
- *'ray'* - uses multiple processes on your machine or on other machines in the same cluster, orchestrated in a Ray cluster. Large indices will be reopened on each machine.

Parallelisation occurs by partitioning dataframes and separating them across different processes. Partitioning depends on the type of the input dataframe:

- queries: partitioned by qid
- documents: partitioned by docno
- ranked documents: partitioned by qid

NB: Parallelisation is an experimental features. Please let us know what works or what doesnt work using the PyTerrier issue tracker.

NBB: Parallelisation is known not to work on Windows, and hence is disabled.

15.1 Parallelisation using Joblib

A transformer pipeline can be parallelised by using the .parallel() transformer method:

```
dph = pt.BatchRetrieve(index, wmodel='DPH')
dph_fast = dph.parallel(2)
```

In this way, any set of queries passed to dph_fast will be separated into two partitions, based on qid, and executed on dph.

15.2 Parallelisation using Ray

Ray is a framework for distributing Python tasks across multiple machines. For using it in PyTerrier, setup your Ray cluster by following the Ray documentation. Thereafter parallelisation over Ray can be used in PyTerrier in a similar way as for joblib:

```
import ray
ray.init() #configure Ray as per your cluster setup
dph = pt.BatchRetrieve(index, wmodel='DPH')
dph_fast = dph.parallel(2, backend='ray')
```

In particular, *ray.init()* must have been called before calling *.parallel()*.

15.3 What to Parallelise

Only transformers that can be pickled. Transformers that use native code may not be possible to pickle. Some standard PyTerrier transformers have additional support for parallelisation:

- Terrier retrieval: pt.BatchRetrieve(), pt.FeaturesBatchRetrieve()
- Anserini retrieval: pt.AnseriniBatchRetrieve()

Pure python transformers, such as *pt.text.sliding()* are picklable. However, parallelising only *pt.text.sliding()* may not produce efficiency gains, due to the overheads of shuffling data back and forward.

Entire transformer pipelines (i.e. combined using operators) can be pickled and parallelised. In general, you should parallelise the most inefficient component of your process, while also minimising the amount of data being transferred between processes. For instance, consider the following pipeline:

```
pipe = pt.BatchRetrieve(index, metadata=["docno", "text"] >> pt.text.sliding() >> pt.
↪text.scorer() >> pt.text.max_passage()
```

While BatchRetrieve might represent the slowest component of the pipeline, it might make sense to parallelise pipe as a whole, rather than just BatchRetrieve, as then only the queries and final results need to be passed betwene processes. Indeed among the following semantically equivalent pipelines, we expect *parallel_pipe0* and *parallel_pipe2* to be faster than *parallel_pipe1*:

```
parallel_pipe0 = pt.BatchRetrieve(index, metadata=["docno", "text"]).parallel() >> pt.
↪text.sliding() >> pt.text.scorer() >> pt.text.max_passage()
parallel_pipe1 = ( pt.BatchRetrieve(index, metadata=["docno", "text"]).parallel() >> pt.
↪text.sliding() ).parallel(2)  >> pt.text.max_passage()
parallel_pipe2 = pipe.parallel(2)
```

There are of course overheads on paralllelisation - for instance, the Terrier index has to be loaded for *each* separate process, so your machine(s) require enough memory. Shared resources such as GPU cards will need careful consideration - adding multiple processes accesssing the same resources will not increase speed and may add problems instead.

Finally, we do not recommend parallelisation in resource-constrained containerised environments such as Google Colab.

If you find PyTerrier transformers that do not parallelise and you think it should, please raise an issue on the PyTerrier github repository.

15.4 Outlook

We expect to integate parallelisation at different parts of the PyTerrier platform, such as for conducting a gridsearch. Moreover, we hope that proper integration of multi-threaded retrieval in pt.BatchRetrieve() (while requires upstream improvements in the underlying Terrier platform) will reduce the need for this form of parallelisation.

CHAPTER

SIXTEEN

TUNING TRANSFORMER PIPELINES

Many approaches will have parameters that require tuning. PyTerrier helps to achieve this by proving a grid evaluation functionality that can tune one or more parameters using a particular evaluation metric. There are two functions which helps to achieve this:

- *pt.GridScan()* exhaustively evaluates all possibile parameters settings and computes evaluation measures.
- *pt.GridSearch()* applies GridScan, and determines the most effective parameter setting for a given evaluation measure.
- *pt.KFoldGridSearch()* applies GridSearch on different folds, in order to determine the most effective parameter setting for a given evaluation measure on the training topics for each fold. The results on the test topics are returned.

All of these functions are designed to have an API very similar to pt.Experiment().

16.1 Pre-requisites

GridScan makes several assumptions:

- the parameters that you wish to tune are available as instance attributes within the transformers, or that the transformer responds suitably to *set_parameter()*.
- changing the relevant parameters has an impact upon subsequent calls to *transform()*.

Note that transformers implemented using pt.apply functions cannot address the second requirement, as any parameters are captured naturally within the closure, and not as instances attributes of the transformer.

16.2 Parameter Scanning and Searching API

`pyterrier.`**`GridScan`**(*pipeline*, *params*, *topics*, *qrels*, *metrics=['map']*, *jobs=1*, *backend='joblib'*, *verbose=False*, *batch_size=None*, *dataframe=True*)

GridScan applies a set of named parameters on a given pipeline and evaluates the outcome. The topics and qrels must be specified. The trec_eval measure names can be optionally specified. The transformers being tuned, and their respective parameters are named in the param_dict. The parameter being varied must be changable using the `set_parameter()` method. This means instance variables, as well as controls in the case of BatchRetrieve.

Parameters

- **`pipeline`** (*`TransformerBase`*) – a transformer or pipeline
- **`params`** (*`dict`*) – a two-level dictionary, mapping transformer to param name to a list of values

- **topics** (*DataFrame*) – topics to tune upon
- **qrels** (*DataFrame*) – qrels to tune upon
- **metrics** (*List[str]*) – name of the metrics to report for each setting. Defaults to ["map"].
- **batch_size** (*int*) – If not None, evaluation is conducted in batches of batch_size topics. Default=None, which evaluates all topics at once. Applying a batch_size is useful if you have large numbers of topics, and/or if your pipeline requires large amounts of temporary memory during a run. Default is None.
- **jobs** (*int*) – Number of parallel jobs to run. Default is 1, which means sequentially.
- **backend** (*str*) – Parallelisation backend to use. Defaults to "joblib".
- **verbose** (*bool*) – whether to display progress bars or not
- **dataframe** (*bool*) – return a dataframe or a list

Returns A dataframe showing the effectiveness of all evaluated settings, if dataframe=True A list of settings and resulting evaluation measures, if dataframe=False

Raises **ValueError** – if a specified transformer does not have such a parameter

Example:

```
# graph how PL2's c parameter affects MAP
pl2 = pt.BatchRetrieve(index, wmodel="PL2", controls={'c' : 1})
rtr = pt.GridScan(
    pl2,
    {pl2 : {'c' : [0.1, 1, 5, 10, 20, 100]}},
    topics,
    qrels,
    ["map"]
)
import matplotlib.pyplot as plt
plt.plot(rtr["tran_0_c"], rtr["map"])
plt.xlabel("PL2's c value")
plt.ylabel("MAP")
plt.show()
```

Return type Union[DataFrame, List[Tuple[List[Tuple[TransformerBase, str, Union[str, float, int]]], Dict[str, float]]]]

pyterrier.**GridSearch**(*pipeline*, *params*, *topics*, *qrels*, *metric='map'*, *jobs=1*, *backend='joblib'*, *verbose=False*, *batch_size=None*, *return_type='opt_pipeline'*)

GridSearch is essentially, an argmax GridScan(), i.e. it returns an instance of the pipeline to tune with the best parameter settings among params, that were found that were obtained using the specified topics and qrels, and for the specified measure.

Parameters

- **pipeline** (*TransformerBase*) – a transformer or pipeline to tune
- **params** (*dict*) – a two-level dictionary, mapping transformer to param name to a list of values
- **topics** (*DataFrame*) – topics to tune upon
- **qrels** (*DataFrame*) – qrels to tune upon

- **metric** (*str*) – name of the metric on which to determine the most effective setting. Defaults to "map".
- **batch_size** (*int*) – If not None, evaluation is conducted in batches of batch_size topics. Default=None, which evaluates all topics at once. Applying a batch_size is useful if you have large numbers of topics, and/or if your pipeline requires large amounts of temporary memory during a run. Default is None.
- **jobs** (*int*) – Number of parallel jobs to run. Default is 1, which means sequentially.
- **backend** (*str*) – Parallelisation backend to use. Defaults to "joblib".
- **verbose** (*bool*) – whether to display progress bars or not
- **return_type** (*str*) – whether to return the same transformer with optimal pipeline setting, and/or a setting of the higher metric value, and the resulting transformers and settings.

Return type `Union[TransformerBase, Tuple[float, List[Tuple[TransformerBase, str, Union[str, float, int]]]]]`

16.3 Examples

16.3.1 Tuning BM25

When using BatchRetrieve, the *b* parameter of the BM25 weighting model can be controled using the "c" control. We must give this control an initial value when contructing the BatchRetrieve instance. Thereafter, the GridSearch parameter dictionary can be constructed by refering to the instance of transformer that has that parameter:

```
BM25 = pt.BatchRetrieve(index, wmodel="BM25", controls={"c" : 0.75})
pt.GridSearch(
    BM25,
    {BM25 : {"c" : [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1 ]}}
    train_topics,
    train_qrels,
    "map")
```

Terrier's BM25 also responds to controls named *"bm25.k_1"* and *"bm25.k_3"*, such that all three controls can be tuned concurrently:

```
BM25 = pt.BatchRetrieve(index, wmodel="BM25", controls={"c" : 0.75, "bm25.k_1": 0.75,
↪"bm25.k_3": 0.75})
pt.GridSearch(
    BM25,
    {BM25: {"c" : [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1 ],
            "bm25.k_1": [0.3, 0.6, 0.9, 1.2, 1.4, 1.6, 2],
            "bm25.k_3": [0.5, 2, 4, 6, 8, 10, 12, 14, 20]
    }}
    train_topics,
    train_qrels,
    "map")
```

16.3.2 Tuning BM25 and RM3

The query expansion transformer in pt.rewrite have parameters controlling the number of feedback documents and expansion t

- fb_terms – the number of terms to add to the query.
- fb_docs – the size of the pseudo-relevant set.

A full tuning of BM25 and RM3 can be achieved as thus:

```
bm25_for_qe = pt.BatchRetrieve(index, wmodel="BM25", controls={"c" : 0.75})
rm3 = pt.rewrite.RM3(index, fb_terms=10, fb_docs=3)
pipe_qe = bm25_for_qe >> rm3 >> bm25_for_qe

param_map = {
        bm25_for_qe : { "c" : [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1 ]},
        rm3 : {
            "fb_terms" : list(range(1, 12, 3)),
            "fb_docs" : list(range(2, 30, 6))
        }
}
pipe_qe = pt.GridSearch(pipe_qe, param_map, train_topics, train_qrels)
pt.Experiment([pipe_qe], test_topics, test_qrels, ["map"])
```

16.4 Using Multiple Folds

`pyterrier.`**`KFoldGridSearch`**(*pipeline*, *params*, *topics_list*, *qrels*, *metric='map'*, *jobs=1*, *backend='joblib'*, *verbose=False*, *batch_size=None*)

Applies a GridSearch using different folds. It returns the *results* of the tuned transformer pipeline on the test topics. The number of topics dataframes passed to topics_list defines the number of folds. For each fold, all but one of the dataframes is used as training, and the remainder used for testing.

The state of the transformers in the pipeline is restored after the KFoldGridSearch has been executed.

Parameters

- **pipeline** (*TransformerBase*) – a transformer or pipeline to tune
- **params** (*dict*) – a two-level dictionary, mapping transformer to param name to a list of values
- **topics_list** (*List[DataFrame]*) – a *list* of topics dataframes to tune upon
- **qrels** (*DataFrame or List[DataFrame]*) – qrels to tune upon. A single dataframe, or a list for each fold.
- **metric** (*str*) – name of the metric on which to determine the most effective setting. Defaults to "map".
- **batch_size** (*int*) – If not None, evaluation is conducted in batches of batch_size topics. Default=None, which evaluates all topics at once. Applying a batch_size is useful if you have large numbers of topics, and/or if your pipeline requires large amounts of temporary memory during a run. Default is None.
- **jobs** (*int*) – Number of parallel jobs to run. Default is 1, which means sequentially.
- **backend** (*str*) – Parallelisation backend to use. Defaults to "joblib".

- **verbose** (*bool*) – whether to display progress bars or not

Returns: A tuple containing, firstly, the results of pipeline on the test topics after tuning, and secondly, a list of the best parameter settings for each fold.

Consider tuning PL2 where folds of queries are pre-determined:

```
pl2 = pt.BatchRetrieve(index, wmodel="PL2", controls={'c' : 1})
tuned_pl2, _ = pt.KFoldGridSearch(
    pl2,
    {pl2 : {'c' : [0.1, 1, 5, 10, 20, 100]}},
    [topicsf1, topicsf2],
    qrels,
    ["map"]
)
pt.Experiment([pl2, tuned_pl2], all_topics, qrels, ["map"])
```

As 2 splits are defined, PL2 is first tuned on topicsf1 and tested on topicsf2, then trained on topicsf2 and tested on topicsf1. The results dataframe of PL2 after tuning of the c parameter are returned by the KFoldGridSearch, and can be used directly in a pt.Experiment().

Return type `Tuple[DataFrame, Tuple[float, List[Tuple[TransformerBase, str, Union[str, float, int]]]]]`

16.5 Parallelisation

GridScan, GridSearch and KFoldGridSearch can all be accelerated using parallelisation to conduct evalutions of different parameter settings in parallel. Both accept *jobs* and *backend* kwargs, which define the number of backend processes to conduct, and the parallelisation backend. For instance:

```
pt.GridSearch(pipe_qe, param_map, train_topics, train_qrels, jobs=10)
```

This incantation will fork 10 Python processes to run the different settings in parallel. Each process will load a new instance of any large data structures, such as Terrier indices, so your machines must have sufficient memory to load 10 instances of the index.

The Ray backend offers parallelisation across multiple machines. For more information, see *Parallelisation*.

CHAPTER

SEVENTEEN

EXPERIMENTS ON TREC ROBUST 2004

This document gives a flavour of indexing and obtaining retrieval baselines on the TREC Robust04 test collections. You can run these experiments for yourself by using the associated provided notebook.

You need to have obtain the TREC Disks 4 & 5 corpora from NIST.

Topics and Qrels are provided through the `"trec-robust-2004"` PyTerrier dataset.

17.1 Indexing

Indexing is fairly simply. We apply a filter to remove files that shouldn't be indexed, including the Congressional Record. Indexing on a reasonable machine using a single-thread takes around 7 minutes.

```
DISK45_PATH="/path/to/disk45"
INDEX_DIR="/path/to/create/the/index"

files = pt.io.find_files(DISK45_PATH)
# no-one indexes the congressional record in directory /CR/
# indeed, recent copies from NIST dont contain it
# we also remove some of the other unneeded files
bad = ['/CR/', '/AUX/', 'READCHG', 'READFRCG']
for b in bad:
    files = list(filter(lambda f: b not in f, files))
indexer = pt.TRECCollectionIndexer(INDEX_DIR, verbose=True)
indexref = indexer.index(files)
```

17.2 Retrieval - Simple Weighting Models

Here we define and evaluate standard weighting models.

```

BM25 = pt.BatchRetrieve(index, wmodel="BM25")
DPH  = pt.BatchRetrieve(index, wmodel="DPH")
PL2  = pt.BatchRetrieve(index, wmodel="PL2")
DLM  = pt.BatchRetrieve(index, wmodel="DirichletLM")

pt.Experiment(
    [BM25, DPH, PL2, DLM],
    pt.get_dataset("trec-robust-2004").get_topics(),
```

(continues on next page)

(continued from previous page)

```
    pt.get_dataset("trec-robust-2004").get_qrels(),
    eval_metrics=["map", "P_10", "P_20", "ndcg_cut_20"],
    names=["BM25", "DPH", "PL2", "Dirichlet QL"]
)
```

Results are as follows:

	name	map	P_10	P_20	ndcg_cut_20
0	BM25	0.241763	0.426104	0.349398	0.408061
1	DPH	0.251307	0.44739	0.361446	0.422524
2	PL2	0.229386	0.420884	0.343775	0.402179
3	Dirichlet QL	0.236826	0.407631	0.337952	0.39687

17.3 Retrieval - Query Expansion

Here we define and evaluate standard weighting models on top of DPH and BM25, respectively. We use the default Terrier parameters for query expansion, namely:

- 10 expansion terms
- 3 documents
- For RM3, a lambda value of 0.5

```
Bo1 = pt.rewrite.Bo1QueryExpansion(index)
KL = pt.rewrite.KLQueryExpansion(index)
RM3 = pt.rewrite.RM3(index)
pt.Experiment(
    [
            BM25,
            BM25 >> Bo1 >> BM25,
            BM25 >> KL >> BM25,
            BM25 >> RM3 >> BM25,
    ],
    pt.get_dataset("trec-robust-2004").get_topics(),
    pt.get_dataset("trec-robust-2004").get_qrels(),
    eval_metrics=["map", "P_10", "P_20", "ndcg_cut_20"],
    names=["BM25", "+Bo1", "+KL", "+RM3"]
    )

pt.Experiment(
    [
            DPH,
            DPH >> Bo1 >> DPH,
            DPH >> KL >> DPH,
            DPH >> RM3 >> DPH,
    ],
    pt.get_dataset("trec-robust-2004").get_topics(),
    pt.get_dataset("trec-robust-2004").get_qrels(),
```

(continues on next page)

(continued from previous page)

```
    eval_metrics=["map", "P_10", "P_20", "ndcg_cut_20"],
    names=["DPH", "+Bo1", "+KL", "+RM3"]
    )
```

Results are as follows:

	name	map	P_10	P_20	ndcg_cut_20
0	BM25	0.241763	0.426104	0.349398	0.408061
1	+Bo1	*0.279458*	0.448996	0.378916	*0.436533*
2	+KL	0.279401	0.444177	0.378313	0.435196
3	+RM3	0.276544	*0.453815*	*0.379518*	0.430367
—-	———	———-	———-	———-	—————
0	DPH	0.251307	0.447390	0.361446	0.422524
1	+Bo1	0.285334	0.458635	0.387952	*0.444528*
2	+KL	*0.285720*	0.458635	0.386948	0.442636
3	+RM3	0.281796	*0.461044*	*0.389960*	0.441863

CHAPTER

EIGHTEEN

PYTERRIER.IO - READING/WRITING FILES

This module provides useful utility methods for reading and writing files. In particular, it also provides support for reading and writing standard formats, such as TREC-formatted topics files or *run* files.

`pyterrier.io.`**`autoopen`**(*filename*, *mode='rb'*)
A drop-in for open() that applies automatic compression for .gz and .bz2 file extensions

`pyterrier.io.`**`find_files`**(*dir*)
Returns all the files present in a directory and its subdirectories

Parameters **`dir`** (*`str`*) – The directory containing the files

Returns A list of the paths to the files

Return type paths(list)

`pyterrier.io.`**`finalized_open`**(*path*, *mode*)
Opens a file for writing, but reverts it if there was an error in the process.

Parameters

- **`path`** (*`str`*) – Path of file to open
- **`mode`** (*`str`*) – Either t or b, for text or binary mode

Example

Returns a contextmanager that provides a file object, so should be used in a "with" statement. E.g.:

```
with pt.io.finalized_open("file.txt", "t") as f:
    f.write("some text")
# file.txt exists with contents "some text"
```

If there is an error when writing, the file is reverted:

```
with pt.io.finalized_open("file.txt", "t") as f:
    f.write("some other text")
    raise Exception("an error")
# file.txt remains unchanged (if existed, contents unchanged; if didn't exist, still␣
↪doesn't)
```

`pyterrier.io.`**`finalized_autoopen`**(*path*, *mode*)
Opens a file for writing with `autoopen`, but reverts it if there was an error in the process.

Parameters

- **`path`** (*`str`*) – Path of file to open

- **mode** (*str*) – Either t or b, for text or binary mode

Example

Returns a contextmanager that provides a file object, so should be used in a "with" statement. E.g.:

```
with pt.io.finalized_autoopen("file.gz", "t") as f:
    f.write("some text")
# file.gz exists with contents "some text"
```

If there is an error when writing, the file is reverted:

```
with pt.io.finalized_autoopen("file.gz", "t") as f:
    f.write("some other text")
    raise Exception("an error")
# file.gz remains unchanged (if existed, contents unchanged; if didn't exist, still␣
↪doesn't)
```

pyterrier.io.**ok_filename**(*fname*)

Checks to see if a filename is valid.

Return type bool

pyterrier.io.**touch**(*fname*, *mode=438*, *dir_fd=None*, ***kwargs*)

Eqiuvalent to touch command on linux. Implementation from https://stackoverflow.com/a/1160227

pyterrier.io.**read_results**(*filename*, *format='trec'*, *topics=None*, *dataset=None*, ***kwargs*)

Reads a file into a results dataframe.

Parameters

- **filename** (*str*) – The filename of the file to be read. Compressed files are handled automatically. A URL is also supported for the "trec" format.
- **format** (*str*) – The format of the results file: one of "trec", "letor". Default is "trec".
- **topics** (*None or pandas.DataFrame*) – If provided, will merge the topics to merge into the results. This is helpful for providing query text. Cannot be used in conjunction with dataset argument.
- **dataset** (*None, str or* pyterrier.datasets.Dataset) – If provided, loads topics from the dataset (or dataset ID) and merges them into the results. This is helpful for providing query text. Cannot be used in conjunction with dataset topics.
- ****kwargs** (*dict*) – Other arguments for the internal method

Returns dataframe with usual qid, docno, score columns etc

Examples:

```
# a dataframe of results can be used directly in a pt.Experiment
pt.Experiment(
    [ pt.io.read_results("/path/to/baselines-results.res.gz") ],
    topics,
    qrels,
    ["map"]
)
```

(continues on next page)

(continued from previous page)

```
# make a transformer from a results dataframe, include the query text
first_pass = pt.Transformer.from_df( pt.io.read_results("/path/to/results.gz",␣
↪topics=topics) )
# make a max_passage retriever based on a previously saved results
max_passage = (first_pass
    >> pt.text.get_text(dataset)
    >> pt.text.sliding()
    >> pt.text.scorer()
    >> pt.text.max_passage()
)
```

`pyterrier.io.`**`write_results`**(*res*, *filename*, *format='trec'*, *append=False*, ***kwargs*)

Write a results dataframe to a file.

Parameters

- **res** (*DataFrame*) – A results dataframe, with usual columns of qid, docno etc
- **filename** (*str*) – The filename of the file to be written. Compressed files are handled automatically.
- **format** (*str*) – The format of the results file: one of "trec", "letor", "minimal"
- **append** (*bool*) – Append to an existing file. Defaults to False.
- ****kwargs** (*dict*) – Other arguments for the internal method

Supported Formats:

- "trec" – output columns are $qid Q0 $docno $rank $score $runname, space separated
- "letor" – This follows the LETOR and MSLR datasets, in that output columns are $label qid:$qid [$fid:$value]+ # docno=$docno
- "minimal": output columns are $qid $docno $rank, tab-separated. This is used for submissions to the MSMARCO leaderboard.

`pyterrier.io.`**`read_topics`**(*filename*, *format='trec'*, ***kwargs*)

Reads a file containing topics.

Parameters

- **filename** (*str*) – The filename of the topics file. A URL is supported for the "trec" and "singleline" formats.
- **format** (*str*) – One of "trec", "trecxml" or "singleline". Default is "trec"

Returns pandas.Dataframe with columns=['qid','query'] both columns have type string

Supported Formats:

- "trec" – an SGML-formatted TREC topics file. Delimited by TOP tags, each having NUM and TITLE tags; DESC and NARR tags are skipped by default. Control using whitelist and blacklist kwargs
- "trecxml" – a more modern XML formatted topics file. Delimited by topic tags, each having nunber tags. query, question and narrative tags are parsed by default. Control using tags kwarg.
- "singeline" – one query per line, preceeded by a space or colon. Tokenised by default, use tokenise=False kwargs to prevent tokenisation.

`pyterrier.io.`**`read_qrels`**(*file_path*)

Reads a file containing qrels (relevance assessments)

Parameters **`file_path`** (*`str`*) – The path to the qrels file. A URL is also supported.

Returns pandas.Dataframe with columns=['qid','docno', 'label'] with column types string, string, and int

CHAPTER

NINETEEN

PYTERRIER.APPLY - CUSTOM TRANSFORMERS

PyTerrier pipelines are easily extensible through the use of apply functions. These are inspired by the Pandas apply() method, which allow to apply a function to each row of a dataframe. Instead, in PyTerrier, the apply methods allow to construct pipeline transformers to address common use cases by using custom functions (including Python lambda functions) to easily transform inputs.

The table below lists the main classes of transformation in the PyTerrier data model, as well as the appropriate apply method to use in each case. In general, if there is a one-to-one mapping between the input and the output, then the specific pt.apply methods should be used (i.e. `query()`, `doc_score()`, `.doc_features()`). If the cardinality of the dataframe changes through applying the transformer, then `generic()` or `by_query()` must be applied.

In particular, through the use of `pt.apply.doc_score()`, any reranking method that can be expressed as a function of the text of the query and the text of the doucment can used as a reranker in a PyTerrier pipeline.

Each apply method takes as input a function (e.g. a function name, or a lambda expression). Objects that are passed to the function vary in terms of the type of the input dataframe (queries or ranked documents), and also vary in terms of what should be returned by the function.

Input	Output	Cardinality	Example	Example apply	Function Input type	Function Return type
Q	Q	1 to 1	Query rewriting	*pt.apply.query()*	row of one query	str
Q x D	Q x D	1 to 1	Re-ranking	*pt.apply.doc_score()*	row of one document	float
Q x D	Q x Df	1 to 1	Feature scoring	*pt.apply.doc_features()*	row of one document	numpy array
Q x D	Q	N to 1	Query expansion	*pt.apply.generic()*	entire dataframe	entire dataframe
				pt.apply.by_query()	dataframe for 1 query	dataframe for 1 query
Q	Q x D	1 to N	Retrieval	*pt.apply.generic()*	entire dataframe	entire dataframe
				pt.apply.by_query()	dataframe for 1 query	dataframe for 1 query

In each case, the result from calling a pyterrier.apply method is another PyTerrier transformer (i.e. extends `pt.Transformer`), which can be used for experimentation or combined with other PyTerrier transformers through the standard PyTerrier operators.

If *verbose=True* is passed to any pyterrier apply method (except *generic()*), then a TQDM progress bar will be shown as the transformer is applied.

19.1 Example

In the following, we create a document re-ranking transformer that increases the score of documents by 10% if their url attribute contains *"https:"*

```
>>> df = pd.DataFrame([["q1", "d1", "https://www.example.com", 1.0, 1]], columns=["qid",
↪"docno", "url", "score", "rank"])
>>> df
qid docno                      url  score  rank
0  q1    d1  https://www.example.com    1.0     1
>>>
>>> http_boost = pt.apply.doc_score(lambda row: row["score"] * 1.1 if "https:" in row[
↪"url"] else row["score"])
>>> http_boost(df)
qid docno                      url  score  rank
0  q1    d1  https://www.example.com    1.1     0
```

Further examples are shown for each apply method below.

19.2 Apply Methods

`pyterrier.apply.`**`query`**(*fn*, **args*, ***kwargs*)

Create a transformer that takes as input a query, and applies a supplied function to compute a new query formulation.

The supplied function is called once for each query, and must return a string containing the new query formulation. Each time it is called, the function is supplied with a Panda Series representing the attributes of the query.

The previous query formulation is saved in the "query_0" column. If a later pipeline stage is intended to resort to be executed on the previous query formulation, a *pt.rewrite.reset()* transformer can be applied.

Parameters

- **fn** (*Callable*) – the function to apply to each row. It must return a string containing the new query formulation.
- **verbose** (*bool*) – if set to True, a TQDM progress bar will be displayed

Examples:

```
# this will remove pre-defined stopwords from the query
stops=set(["and", "the"])

# a naieve function to remove stopwords - takes as input a Pandas Series, and␣
↪returns a string
def _remove_stops(q):
    terms = q["query"].split(" ")
    terms = [t for t in terms if not t in stops ]
    return " ".join(terms)

# a query rewriting transformer that applies the _remove_stops to each row of an␣
↪input dataframe
p1 = pt.apply.query(_remove_stops) >> pt.BatchRetrieve(index, wmodel="DPH")
```

(continues on next page)

(continued from previous page)

```
# an equivalent query rewriting transformer using an anonymous lambda function
p2 = pt.apply.query(
        lambda q :  " ".join([t for t in q["query"].split(" ") if t not in stops ])
    ) >> pt.BatchRetrieve(index, wmodel="DPH")
```

In both of the example pipelines above (*p1* and *p2*), the exact topics are not known until the pipeline is invoked, e.g. by using *p1.transform(topics)* on a topics dataframe, or within a *pt.Experiment()*. When the pipeline is invoked, the specified function (*_remove_stops* in the case of *p1*) is called for **each** row of the input datatrame (becoming the *q* function argument).

Return type *Transformer*

`pyterrier.apply.`**`doc_score`**(*fn*, **args*, ***kwargs*)

Create a transformer that takes as input a ranked documents dataframe, and applies a supplied function to compute a new score. Ranks are automatically computed.

The supplied function is called once for each document, and must return a float containing the new score for that document. Each time it is called, the function is supplied with a Panda Series representing the attributes of the query and document.

Parameters

- **fn** (*Callable*) – the function to apply to each row
- **verbose** (*bool*) – if set to True, a TQDM progress bar will be displayed

Example:

```
# this transformer will subtract 5 from the score of each document
p = pt.BatchRetrieve(index, wmodel="DPH") >>
    pt.apply.doc_score(lambda doc : doc["score"] -5)
```

Return type *Transformer*

`pyterrier.apply.`**`doc_features`**(*fn*, **args*, ***kwargs*)

Create a transformer that takes as input a ranked documents dataframe, and applies the supplied function to each document to compute feature scores.

The supplied function is called once for each document, must each time return a 1D numpy array. Each time it is called, the function is supplied with a Panda Series representing the attributes of the query and document.

Parameters

- **fn** (*Callable*) – the function to apply to each row
- **verbose** (*bool*) – if set to True, a TQDM progress bar will be displayed

Example:

```
# this transformer will compute the character and number of word in each document␣
↪retrieved
# using the contents of the document obtained from the MetaIndex

def _features(row):
    docid = row["docid"]
    content = index.getMetaIndex.getItem("text", docid)
    f1 = len(content)
```

(continues on next page)

(continued from previous page)

```
    f2 = len(content.split(" "))
    return np.array([f1, f2])

p = pt.BatchRetrieve(index, wmodel="BM25") >>
    pt.apply.doc_features(_features )
```

Return type *Transformer*

pyterrier.apply.**rename**(*columns*, **args*, ***kwargs*)
Creates a transformer that renames columns in a dataframe.

Parameters **columns** (*dict*) – A dictionary mapping from old column name to new column name

Example:

```
pipe = pt.BatchRetrieve(index, metadata=["docno", "body"]) >> pt.apply.rename({'body
↪':'text'})
```

Return type *Transformer*

pyterrier.apply.**generic**(*fn*, **args*, ***kwargs*)
Create a transformer that changes the input dataframe to another dataframe in an unspecified way.

The supplied function is called once for an entire result set as a dataframe (which may contain one of more queries). Each time it should return a new dataframe. The returned dataframe should abide by the general PyTerrier Data Model, for instance updating the rank column if the scores are amended.

Parameters **fn** (*Callable*) – the function to apply to each row

Example:

```
# this transformer will remove all documents at rank greater than 2.

# this pipeline would remove all but the first two documents from a result set
pipe = pt.BatchRetrieve(index) >> pt.apply.generic(lambda res : res[res["rank"] <␣
↪2])
```

Return type *Transformer*

pyterrier.apply.**by_query**(*fn*, **args*, ***kwargs*)
As *pt.apply.generic()* except that fn receives a dataframe for one query at at time, rather than all results at once.

Return type *Transformer*

19.3 Making New Columns and Dropping Columns

Its also possible to construct a transformer that makes a new column on a row-wise basis by directly naming the new column in pt.apply.

For instance, if the column you are creating is called rank_2, it might be created as follows:

```
pipe = pt.BatchRetrieve(index) >> pt.apply.rank_2(lambda row: row["rank"] * 2)
```

To create a transformer that drops a column, you can instead pass *drop=True* as a kwarg:

```
pipe = pt.BatchRetrieve(index, metadata=["docno", "text"] >> pt.text.scorer() >> pt.
↪apply.text(drop=True)
```

CHAPTER TWENTY

PYTERRIER.ANSERINI - ANSERINI/LUCENE SUPPORT

Through an integration of pyserini, PyTerrier can integrate results from the Lucene-based Anserini platform into retrieval pipelines.

class pyterrier.anserini.**AnseriniBatchRetrieve**(*index_location*, *k=1000*, *wmodel='BM25'*, ***kwargs*)

Allows retrieval from an Anserini index. To use this class, PyTerrier should have been started using *pt.init(boot_packages=["io.anserini:anserini:0.9.2:fatjar"])*.

Construct an AnseriniBatchRetrieve retrieve.

Parameters

- **index_location** (*str*) – The location of the Anserini index.
- **wmodel** (*str*) – Weighting models supported by Anserini. There are three options:
  - *"BM25"* - the BM25 weighting model
  - *"QLD"* - Dirichlet language modelling
  - *"TFIDF"* - Lucene's ClassicSimilarity.
- **k** (*int*) – number of results to return. Default is 1000.

transform(*queries*)

Performs the retrieval

Parameters queries – String for a single query, list of queries, or a pandas.Dataframe with columns=['qid', 'query']

Returns pandas.DataFrame with columns=['qid', 'docno', 'rank', 'score']

20.1 Examples

Comparative retrieval from Anserini and Terrier:

```
trIndex = "/path/to/data.properties"
luceneIndex "/path/to/lucene-index-dir"

BM25_tr = pt.BatchRetrieve(trIndex, wmodel="BM25")
BM25_ai = pt.anserini.AnseriniBatchRetrieve(luceneIndex, wmodel="BM25")

pt.Experiment([BM25_tr, BM25_ai], topics, qrels, eval_metrics=["map"])
```

AnseriniBatchRetrieve can also be used as a re-ranker:

```
BM25_tr = pt.BatchRetrieve(trIndex, wmodel="BM25")
QLD_ai = pt.anserini.AnseriniBatchRetrieve(luceneIndex, wmodel="QLD")

pipe = BM25_tr >> QLD_ai
```

CHAPTER

TWENTYONE

PYTERRIER.NEW - CREATING NEW DATAFRAMES

This module provides useful utility methods for creating example dataframes for queries and ranked documents.

`pyterrier.new.`**`empty_Q`**`()`

Returns an empty dataframe with columns *["qid", "query"]*.

Return type `DataFrame`

`pyterrier.new.`**`queries`**`(`*queries*, *qid=None*, ***others*`)`

Creates a new queries dataframe. Will return a dataframe with the columns *["qid", "query"]*. Any further lists in others will also be added.

Parameters

- **`queries`** – The search queries. Either a string, for a single query, or a sequence (e.g. list of strings)
- **`qids`** – Corresponding query ids. Either a string, for a single query, or a sequence (e.g. list of strings). Must have same length as queries.
- **`others`** – A dictionary of other attributes to add to the query dataframe

Examples:

```
# create a dataframe with one query, qid "1"
one_query = pt.new.queries("what the noise was was the question")

# create a dataframe with one query, qid "5"
one_query = pt.new.queries("what the noise was was the question", 5)

# create a dataframe with two queries
one_query = pt.new.queries(["query text A", "query text B"], ["1", "2"])

# create a dataframe with two queries
one_query = pt.new.queries(["query text A", "query text B"], ["1", "2"],␣
↪categories=["catA", "catB"])
```

Return type `DataFrame`

`pyterrier.new.`**`empty_R`**`()`

Returns an empty dataframe with columns *["qid", "query", "docno", "rank", "score"]*.

Return type `DataFrame`

`pyterrier.new.`**`ranked_documents`**`(`*scores*, *qid=None*, *docno=None*, ***others*`)`

Creates a new ranked documents dataframe. Will return a dataframe with the columns *["qid", "docno", "score", "rank"]*. Any further lists in others will also be added.

Parameters

- **scores** – The scores of the retrieved documents. Must be a list of lists.
- **qid** – Corresponding query ids. Must have same length as the first dimension of scores. If omitted, documents, qids are computed as strings starting from "1"
- **docno** – Corresponding docnos. Must have same length as the first dimension of scores and each 2nd dimension must be the same as the number of documents retrieved. If omitted, docnos are computed as strings starting from "d1" for each query.
- **others** – A dictionary of other attributes to add to the query dataframe.

Examples:

```
# one query, one document
R1 = pt.new.ranked_documents([[1]])

# one query, two documents
R2 = pt.new.ranked_documents([[1, 2]])

# two queries, one documents each
R3 = pt.new.ranked_documents([[1], [2]])

# one query, one document, qid specified
R4 = pt.new.ranked_documents([[1]], qid=["q100"])

# one query, one document, qid and docno specified
R5 = pt.new.ranked_documents([[1]], qid=["q100"], docno=[["d20"]])
```

Return type `DataFrame`

CHAPTER

TWENTYTWO

PYTERRIER.DEBUG - TRANSFORMERS FOR DEBUGGING

Its very easy to write complex pipelines with PyTerrier. Sometimes you need to inspect dataframes in the middle of a pipeline. The pt.debug transformers display the columns or the data, and can be inserted into pipelines during development.

22.1 Debug Methods

`pyterrier.debug.print_columns(`*by_query=False*, *message=None*`)`

Returns a transformer that can be inserted into pipelines that can print the column names of the dataframe at this stage in the pipeline:

Parameters

- **by_query** (-) – whether to display for each query. Defaults to False.
- **message** (-) – whether to display a message before printing. Defaults to None, which means no message. This is useful when `print_columns()` is being used multiple times within a pipeline

Example:

```
pipe = (
    bm25
    >> pt.debug.print_columns()
    >> pt.rewrite.RM3()
    >> pt.debug.print_columns()
    bm25
```

When the above pipeline is executed, two sets of columns will be displayed

- *[“qid”, “query”, “docno”, “rank”, “score”]* - the output of BM25, a ranking of documents
- *[“qid”, “query”, “query_0”]* - the output of RM3, a reformulated query

Return type `TransformerBase`

`pyterrier.debug.print_num_rows(`*by_query=True*, *msg='num_rows'*`)`

Returns a transformer that can be inserted into pipelines that can print the number of rows names of the dataframe at this stage in the pipeline:

Parameters

- **by_query** (-) – whether to display for each query. Defaults to True.

- **message** (-) – whether to display a message before printing. Defaults to "num_rows". This is useful when `print_columns()` is being used multiple times within a pipeline

Example:

```
pipe = (
    bm25
    >> pt.debug.print_num_rows()
    >> pt.rewrite.RM3()
    >> pt.debug.print_num_rows()
    bm25
```

When the above pipeline is executed, the following output will be displayed

- *num_rows 1: 1000* - the output of BM25, a ranking of documents
- *num_rows 1: 1* - the output of RM3, the reformulated query

Return type `TransformerBase`

`pyterrier.debug.`**`print_rows`**(*by_query=True*, *jupyter=True*, *head=2*, *message=None*, *columns=None*)

Returns a transformer that can be inserted into pipelines that can print some of the dataframe at this stage in the pipeline:

Parameters

- **by_query** (-) – whether to display for each query. Defaults to True.
- **jupyter** (-) – Whether to use IPython's display function to display the dataframe. Defaults to True.
- **head** (-) – The number of rows to display. None means all rows.
- **columns** (-) – Limit the columns for which data is displayed. Default of None displays all columns.
- **message** (-) – whether to display a message before printing. Defaults to None, which means no message. This is useful when `print_rows()` is being used multiple times within a pipeline

Example:

```
pipe = (
    bm25
    >> pt.debug.print_rows()
    >> pt.rewrite.RM3()
    >> pt.debug.print_rows()
    bm25
```

Return type `TransformerBase`

CHAPTER

TWENTYTHREE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX