Experimaestro and Datamaestro

Benjamin Piwowarski

Experimaestro and Datamaestro: Experiment and Dataset Managers (for IR)

Benjamin Piwowarski
b@piwowarski.fr
CNRS, LIP6/Sorbonne Université

ABSTRACT

Ensuring reproducibility is key to all scientific domains. As Information Retrieval (IR) experiments are often composed of several steps that can be shared between tested models, and rely on various resources, it is difficult to keep track of all the experimental settings and to ensure experiments can be reproduced easily. In this demo paper, we present two managers, Experimaestro and Datamaestro, and their add-ons for IR, designed to help to define and run experimental plans.

KEYWORDS

Experiment manager, dataset manager

ACM Reference Format:
Benjamin Piwowarski. 2020. Experimaestro and Datamaestro: Experiment and Dataset Managers (for IR). In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20), July 25–30, 2020, Virtual Event, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3397271.3401410

1 INTRODUCTION

Ensuring reproducibility is key to all scientific domains. In computer science, this implies preserving the processing code and software environment on the one hand, the dataset and experimental procedure on the other. Code and environment can nowadays be somehow handled by virtualization solutions, but the dataset and experimental procedure are often dealt with handcrafted scripts, experimental parameters being buried in scripts or directly in the code. This makes it difficult, or even practically impossible, to ensure that experiments can be reproduced easily and exposed publicly. More importantly, changing parts of the experiments is made harder by the fact that experimental components are not independent. This problem has been recognized as one of the key goals in the 2018 Dagstuhl reproducibility workshop [6].

In Information Retrieval, the situation is exacerbated by the fact that experiments are composed of several steps (e.g. pre-processing, learning, evaluation), each step being potentially shared between models, and relying on various resources (e.g. word embeddings, document datasets, queries). It is hence difficult to keep track of all the experimental settings.

SIGIR '20, July 25–30, 2020, Virtual Event, China

ACM ISBN 978-1-4503-8016-4/20/07…$15.00
https://doi.org/10.1145/3397271.3401410

To achieve this reproducibility goal, and with information access experiments in mind, we designed Experimaestro[1] and Datamaestro[2], which aim at fulfilling the following requirements: (i) Clearly define experimental parameters for each task, and among those, which are of interest for a specific experiment; (ii) Avoid running two times the same task with the same *experimental* parameters ; (iii) Allows easy experimental management by generating folder paths that are *unique for a given set of experimental parameters*; (iv) Allows to monitor jobs and their results through command line and a web interface; (v) Automates the process of downloading and organizing datasets. In this demo paper, we present both frameworks, and focus on examples from two add-ons specifically designed for IR (and more broadly text-related tasks), namely Experimaestro-IR[3] and Datamaestro-text[4].

2 DATASET MANAGER

2.1 Related works

The closest related software is Kaggle[5], which hosts datasets and provides an API to search and download datasets – as well as handling competition uploads. However, Kaggle is proprietary and limited to datasets that are uploaded to the platform. There are also numerous datasets listings on the Web – the most known in IR being the TREC website. However, there is a lack of unified access to those resources – i.e., even if all data were freely downloadable from the Internet, there is no machine readable description of a resource. For instance, TREC topics have a wide variety of formats and data paths, and many research codes duplicate the efforts in accessing those resources.

2.2 Datamaestro

The purpose of Datamaestro is to provide a simple and distributed tool to catalog datasets (even if their access is partially restricted, e.g. TREC collections), and to provide basic pre-processing functionality. The main Datamaestro module is written in Python, and does not define any dataset directly. The task of defining datasets is handled by Datamaestro *repositories*, such as the *text* repository[6] that contains IR-related repositories – examples are presented below. Datamaestro focus on simple but common goals of a generic dataset manager:

- Providing a way to describe datasets through Python annotations. Figure 1 gives an example for the TREC-5 dataset: four datasets are defined (documents, topics, assessments, as well

[1] http://experimaestro.github.io/experimaestro/
[2] http://experimaestro.github.io/datamaestro/
[3] https://experimaestro-ir.readthedocs.io/
[4] https://experimaestro.github.io/datamaestro_text/
[5] https://www.kaggle.com/
[6] https://experimaestro.github.io/datamaestro_text/

```
1  @linkfolder("documents",
2      [DatafolderPath("gov.nist.trec.tipster", "Disk2/AP")])
3  def ap88(documents):
4      """Associated Press document collection (1988)"""
5      return {"path": documents}
6
7  ...
8
9  @links("documents", ap88=ap88.path, ...)
10 @dataset(TipsterCollection, id="5.documents")
11 def trec5_documents(documents):
12     """TREC-5 documents"""
13     return {"path": documents}
14
15 @filedownloader("topics.sgml",
16   "http://trec.nist.gov/data/topics_eng/topics.251-300.gz")
17 @dataset(TrecAdhocTopics, id="5.topics")
18 def trec5_topics(topics):
19     return {"path": topics, "parts": ["title", "desc"]}
20
21 @concatdownload("assessments.qrels",
22  url="http://trec.nist.gov/.../...parts1-5.tar.gz")
23 @dataset(TrecAdhocAssessments, id="5.qrels")
24 def trec5_assessments(assessments):
25     return {"path": assessments}
26
27 @reference("documents", trec5_documents)
28 @reference("topics", trec5_topics)
29 @reference("assessments", trec5_assessments)
30 @dataset(Adhoc, id="5")
31 def trec5(documents, topics, assessments):
32     "Ad-hoc task of TREC 5 (1996)"
33     return {"documents": documents, "topics": topics,
34             "assessments": assessments}
```

Figure 1: Definition of the TREC-5 collection

as the full TREC-5 dataset comprising the three previously cited datasets). Each definition is composed of two parts: (1) the definition of resources to download or prepare (e.g. lines 19-23); and (2) the definition of the object that describes the dataset (e.g. lines 25-26), and which is returned when asking for a given dataset through the API or using the command line.

- Providing standard tools to download data (e.g. uncompress a ZIP archive and concatenate the files with the `@concatdownload` annotation).

Each dataset is defined by a unique ID (at least within the repository), such as `gov.nist.trec.adhoc.5` for TREC-5. Once datasets are defined, they can be downloaded using their ID with the `prepare` command as follows:

```
$ datamaestro prepare gov.nist.trec.adhoc.5
```

The above command automatically downloads the topics and assessments (`@filedownloader` and `@concatdownload` instructions), and locate the different TREC collections through the definition of *data folders*, i.e. paths to data which has to be manually handled (`@linkfolder` and `DatafolderPath` instructions). The command also outputs a JSON that gives the ID of the dataset and the path to its data (and which is directly a JSON representation of the return statement of lines 25-26 in Figure 1):

```
{
  "documents": {
    "path": ".../5/documents",
    "id": "gov.nist.trec.adhoc.5.documents"
  },
  "topics": {
    "path": ".../5/topics/topics.sgml",
    "parts": [ "title", "desc" ],
    "id": "gov.nist.trec.adhoc.5.topics"
  },
  "assessments": {
    "path": ".../5/qrels/assessments.qrels",
    "id": "gov.nist.trec.adhoc.5.qrels"
  },
  "id": "gov.nist.trec.adhoc.5"
}
```

The above output shows that paths to documents, assessments and topics are given, as well as identifiers for each (sub)collection. The Python interface often allows for more interaction with data, such as transforming word embeddings into a Numpy matrix, and more generally by defining standard ways to interact with specific data types. The data types are defined within a hierarchy (following the Python class hierarchy); for instance, in line 22 of Figure 1, the TREC-5 dataset is defined as an `Adhoc` dataset, which is composed of three mandatory fields (documents, topics and assessments). This allows in turn, using Experimaestro, to define tasks that are defined for Adhoc datasets.

Beside the command line, it is also possible to browse the datasets via a web interface (Figure 2). This web interface organizes datasets by tags and tasks, and is planned to allow a quick access to prepared and available datasets.

3 EXPERIMENT MANAGER

3.1 Related works

Experiment managers are conceptually linked to job scheduling software such as cluster-based OAR [3] or Slurm [4]. Those tools however do not target experiment management, and are thus orthogonal to our purpose. There are projects closer to our work, namely Comet [1], Sacred [8], FGLab [2], Sumatra [5] that all track down experimental parameters. Comet has a strong focus on collaboration and note taking, but targets machine learning single shot experiments and is not open source. Sumatra and FGLab are based on parameter files and are less flexible. The closest to our work, Sacred, is an open-source project that allows to have pre-processing steps (the *ingredients*), but there is no way to build complex experimental plans as in Experimaestro. More precisely, Sacred and all other experiment managers (as far as we know) targets a single run of an experimental pipeline rather than managing a set of related experimental tasks. They all consider that an experimental plan is declarative – typically defined as a set of parameter files, but this turns out to make things complicated when building complex experimental plans.

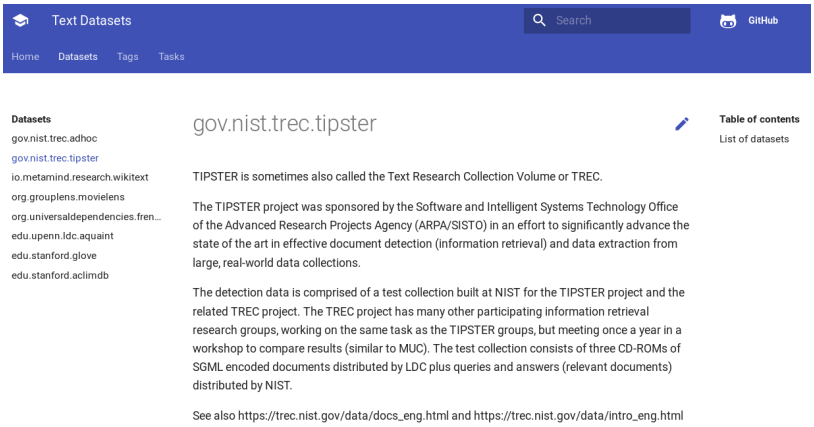

Figure 2: Datamaestro text-related repository

Compared to those projects, Experimaestro has three distinctive features. More precisely, it (1) defines types and tasks that can be *composed* within an experimental plan, (2) has a clear way to indicate which experimental parameters are monitored through the use of *tags*, (3) automatically organizes *tasks outputs* within the file system, removing the burden of choosing where to store a task result, and (4) most importantly, it defines experiments *imperatively* and not *declaratively*.

In IR, there is a trend towards reproducibility, such as OpenNIR [9] and Capreolus [11] for Neural IR, or Anserini [10] for standard IR models. OpenNIR is based on its own configuration system (parameter files), Capreolus is based on Sacred, and Anserini is controlled by command line arguments. In all cases, it is possible to reproduce experimental results and manipulate some experimental parameters through a command line interface. However, modifying the pipeline is not easy since these projects are not modular. With Experimaestro-IR, we show that experiments can be defined by re-using experimental components, leading the way to an easier and standard way to structure IR-related projects – in particular, we leverage wrapped Anserini [10] for IR[7] and adapted OpenNIR [9] for neural IR [8].

3.2 Types and tasks

Types and tasks are the main components of an experimental plan. Defining types and tasks is akin to defining structures in any strongly-typed programming language. Data types can be either simple (real, integer, boolean, string or path) or complex (arrays or dictionaries).

*Config*urations and *tasks* are defined as dictionaries (keys associated with types), and some properties such as a default value or an `ignored` flag which are useful when computing the signature of an experiment (see section 3.4). The example in Figure 3 a is a simple configuration for the BM25 model, defining two parameters (`k1` and `b`) with their default values. It has a type identified a unique ID (e.g. `ir.model.bm25`). *Configurations* are used as configuration

```
@argument("k1", default=0.9)
@argument("b", default=0.4)
@config("ir.model.bm25")
class BM25(Model): pass
```

```
@argument("storePositions", default=False)
@argument("storeDocvectors", default=False)
@argument("storeRawDocs", default=False)
@argument("storeTransformedDocs", default=False)
@argument("documents", type=AdhocDocuments)
@argument("threads", default=8, ignored=True)
@pathargument("index_path", "index")
@task(ANSERINI_NS.index, description="Index documents")
class IndexCollection:
    def execute(self):
        # Calls java command and report progress
        pass
```

Figure 3: Definition of BM-25 model and of the indexation task

units (e.g. a dataset from Datamaestro, a stemmer configuration, or the optimizer to use for a gradient descent).

When it comes to actually running code, Experimaestro allows to define *tasks* that are special kinds of *configurations*. The task in Figure 3 allows to index a data collection retrieved from Datamaestro, based on various experimental parameters (`storePositions`, `storeDocvectors`, `storeRawDocs`, `storeTransformedDocs` and the collection `documents`). It also defines parameters which do not change the outcome but rather (1) the processing (e.g. `threads`) and are marked with an `ignored` flag, (2) the output location on disk (e.g. `index_path`). In both cases, the parameter value should be ignored when computing the signature of the experiment. The method `execute` is called when the task is effectively run, with the different parameters accessible through `self` in the execute method.

[7] https://github.com/bpiwowar/experimaestro-ir/
[8] https://github.com/bpiwowar/OpenNIR-xpm

3.3 Running experiments

When *configurations* and *tasks* are defined, it is possible to assemble them by defining an experimental plan. *Contrarily* to all the other frameworks, Experimaestro has adopted an imperative style to define an experiment. This makes it particularly easy to define complex experimental plans. The code in Figure 4 shows a simple but full experimental plan. The different tasks (whose definition is not shown here) are used to perform various parts of the experiment: (i) Word embeddings are downloaded and used to defined a vocabulary (line 3-4); (ii) The robust collection index downloaded and pre-processed for OpenNir (line 8); (iii) The DRMM is defined (l. 8) and learned (l. 9-11) (iv) The learned model is evaluated on a held out set (l. 15) Each task is submitted with `.submit()` (lines 5, 12 and 15), and handled to a job scheduler that monitors and runs the tasks (on the local machine, or in future versions through SSH or schedulers like OAR) by running the `execute()` method (see Figure 3).

Finally, while many parameters can have an effect on the process outcome, only a subset of those are monitored during a typical experiment. These are specially marked using *tagging*. In the code above, one tag is used (line 8). These tags can be easily retrieved (e.g. when generating the final report), and are also easily accessible when interacting with the command line and web interfaces through a local server which can be launched for any experiment.

3.4 Unique task ID

Notice that there is no indication of the folder where tasks are run and store results is given in the experimental plan, beside the location of the main experiment directory (not shown here). This is one of the strength of Experimaestro, i.e. the exact location is determined when a task is submitted, and is *unique* for a given set of experimental parameters – this allows to avoid running twice the same task and the painful creation of *unique* folder names for each experiment (such as in e.g. Capreolus or OpenNIR), which are error-prone and time-consuming.

```
1  # Prepare the collection
2  random = Random()
3  wordembs = prepare_dataset("edu.stanford.glove.6b.50")
4  vocab = WordvecUnkVocab(data=wordembs, random=random)
5  robust = RobustDataset.prepare().submit()
6
7  # Train with OpenNIR DRMM model
8  ranker = Drmm(vocab=vocab).tag("ranker", "drmm")
9  predictor = Reranker()
10 trainer = PointwiseTrainer()
11 learner = Learner(trainer=trainer, random=random,
   ↪  ranker=ranker, valid_pred=predictor,
12     train_dataset=robust.subset('trf1'),
       ↪  val_dataset=robust.subset('vaf1'),
       ↪  max_epoch=max_epoch)
13 model = learner.submit()
14
15 # Evaluate
16 Evaluate(dataset=robust.subset('f1'), model=model,
   ↪  predictor=predictor).submit()
```

Figure 4: Experimental plan for training a neural IR model (using a special version of OpenNIR)

Experimaestro has a automated process that generates a *unique* signature for each task *depending on experimental parameters* – this idea is used for instance in PlanOut [7] to uniquely identify the system parameters in A/B testing. First, any value can be associated with a unique byte string: the byte string is obtained by outputting the type of the value (e.g. `string`, `ir.adhoc.dataset`) and the value itself as a binary string. A special handling of dictionaries (i.e. configurations and tasks) is performed by sorting keys in ascending lexicographic order, thus ensuring the uniqueness of the representation. Moreover,

- Default values are removed (e.g. `k1` when set to 0.9). This allows to handle the situation where one adds a new experimental parameter (e.g. a new loss component). In that case, using a default parameter allows to add this parameter *without* invalidating all the previously ran experiments.
- Ignored values are removed (e.g. the number of threads when indexing, the path where the index is stored)

When `.submit()` is called, Experimaestro automatically computes the task byte string, and its signature. The identifier will be composed of the task ID and of the identifier, e.g. `ir.model.bm25/133778acb...`. All the artifacts generated by this task are contained within this folder (e.g. the argument `index_path`), allowing easy task management (e.g. lookup results, cleaning up old experiments, etc.).

4 CONCLUSION

In this paper, we have presented Experimaestro and Datamaestro, and their extensions to IR (Experimaestro-IR, OpenNIR-XPM and Datamaestro-text), which aim at helping reproducibility, as well as easing the management of complex experiments in Information Retrieval as well as other computer science fields. This demo paper also seeks to publicize the projects so that new functionalities and datasets are added to them – it is thus an open call for using such frameworks as a basis for ongoing IR experimental software projects.

REFERENCES

[1] 2019. Comet. https://www.comet.ml/. (2019).
[2] 2019. FGLab. https://kaixhin.github.io/FGLab/. (2019).
[3] 2019. OAR. http://oar.imag.fr/. (2019).
[4] 2019. Slurm. https://slurm.schedmd.com/. (2019).
[5] 2019. Sumatra. https://pythonhosted.org/Sumatra/. (2019).
[6] Vaibhav Bajpai, Olivier Bonaventure, Kimberly Claffy, and Daniel Karrenberg. 2019. Encouraging Reproducibility in Scientific Research of the Internet (Dagstuhl Seminar 18412). *Dagstuhl Reports* 8, 10 (2019), 41–62. https://doi.org/10.4230/DagRep.8.10.41
[7] Eytan Bakshy, Dean Eckles, and Michael S. Bernstein. 2014. Designing and Deploying Online Field Experiments. *arXiv:1409.3174 [cs, stat]* (Sept. 2014). arXiv:cs, stat/1409.3174 http://arxiv.org/abs/1409.3174
[8] Klaus Greff. 2015. Sacred. (March 2015). https://doi.org/10.5281/zenodo.16386
[9] Sean MacAvaney. OpenNIR: A Complete Neural Ad-Hoc Ranking Pipeline *(WSDM '20)*. 845–848. https://doi.org/10/ggk85j
[10] Peilin Yang, Hui Fang, and Jimmy Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. 10, 4 (2018). https://doi.org/10/ggmdws
[11] Andrew Yates, Siddhant Arora, Xinyu Zhang, Wei Yang, Kevin Martin Jose, and Jimmy Lin. Capreolus: A Toolkit for End-to-End Neural Ad Hoc Retrieval *(WSDM '20)*. 861–864. https://doi.org/10/ggjnkm