An Information Retrieval Experiment Framework for Domain Specific Applications

Harrisen Scells
Queensland University of Technology
Brisbane, Queensland
harrisen.scells@hdr.qut.edu.au

Daniel Locke
Queensland University of Technology
Brisbane, Queensland
daniel.locke@hdr.qut.edu.au

Guido Zuccon
Queensland University of Technology
Brisbane, Queensland
g.zuccon@qut.edu.au

ABSTRACT

We present a framework for constructing and executing information retrieval experiment pipelines. The framework as a whole is built primarily for domain specific applications such as medical literature search for systematic reviews, or finding factually or legally applicable case law in the legal domain; however it can also be used for more general tasks. There are a number of pre-implemented components that enable common information retrieval experiments such as ad-hoc retrieval or query analysis through query performance predictors. In addition, this collection of tools seeks to be user friendly, well documented, and easily extendible. Finally, the entire pipeline can be distributed as a single binary with no dependencies, ready to use with a simple domain specific language (DSL) for constructing pipelines.

ACM Reference Format:
Harrisen Scells, Daniel Locke, and Guido Zuccon. 2018. An Information Retrieval Experiment Framework for Domain Specific Applications. In *SIGIR '18: The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, July 8–12, 2018, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3209978.3210167

1 PROBLEM AND TARGET USERS

Information retrieval (IR) is a largely empirical field, where theoretical advances are empirically demonstrated and validated, often across a large set of collections, tasks and domains. Experiments often consist of glueing together a number of tools for indexing, query parsing, retrieving, and evaluating, among other tasks. Often, these tools involve the use of multiple programming languages and their combination (or glueing) is often referred to as a pipeline.

As an example, take an experiment of evaluating the effectiveness of query performance predictors [2]. To carry out such an experiment, one must first transform the queries from their original format (e.g. PubMed) into a format suitable for the IR system they use (e.g. Elasticsearch, Terrier, Indri, Galago). Next, one must write code that either extends that system to implement each query performance predictor[1], or implement each query performance predictor separately to the search system by reading in result files produced by it. Next they must produce configuration files for the system and possibly for the extensions they produced. Then they must execute the system and collect the results. In turn, they will need to evaluate the produced result files using a separate utility (e.g. `trec_eval`) and possibly within a separate script (e.g. in Bash). Finally, if one was inclined to compare the results to a different system, they must redo the previous steps and possibly perform statistical analysis (e.g., in R, Python, or MS Excel); rewriting code, potentially introducing bugs. Returning to the pipeline to update it or attempting to extend it often results in rewriting part of the code or starting again with a different approach. Not to mention that often the scripts or programs behind the pipelines require dependencies that become outdated and incompatible with past and future versions.

Our proposed framework aims to make the creation and maintaince of such pipelines easier. The framework targets both inexperienced researchers that would use it for basic experiments and experienced users that are able to further extend the framework. The result of both use cases is a reproducible pipeline that can easily be redistributed.

The pipeline framework proposed and detailed in this paper abstracts many of these pain points. For example: if one wanted to perform the same experiment using this framework (and if the interfaces for a search system are already implemented, e.g., Elasticsearch), then one only needs to implement the interface for a new query performance predictor. They are then able to perform the entire experiment by specifying a pipeline such as the one in Figure 1. Each component of this pipeline (i.e. `cqr`, `transmute`, `groove`, `boogie`), as well as the domain specific language (DSL) used to construct experimental pipelines are detailed in Section 2.

2 EXPERIMENTAL FRAMEWORK

The pipeline that combines the experimental framework is composed of four components: a query representation that is common among the three other components; a query parser and compiler for transforming other query languages into the common query representation, a pipeline for executing IR experiments, and finally a domain-specific language (DSL) to construct a pipeline. The four components are available as open source libraries on GitHub. All of the code for the framework is written in Go [5] — a highly stable and backwards compatible language that puts an emphasis on ensuring programs written in the past will work in the future, and that produces packaged binary files with zero dependencies (two highly valuable features to the problem this framework seeks to address).

[1]Typically this is a difficult task if one does not have knowledge of the system and all of the nuances, and deals with potentially limited documentation.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGIR '18, July 8–12, 2018, Ann Arbor, MI, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5657-2/18/07.
https://doi.org/10.1145/3209978.3210167

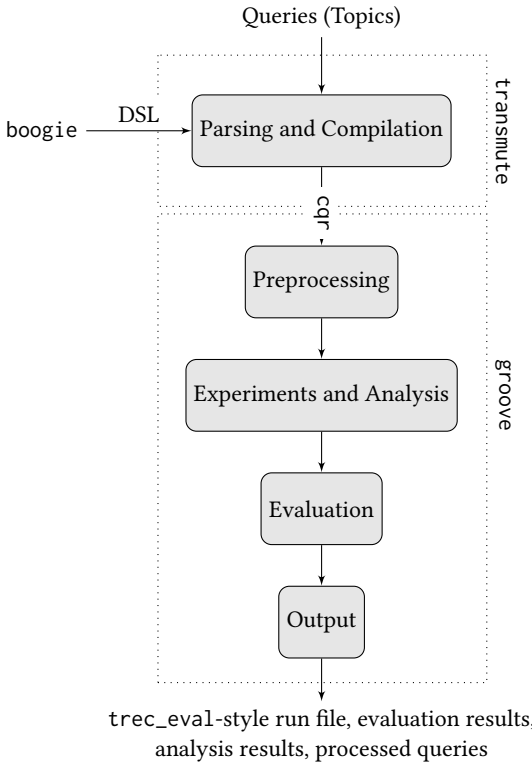

Figure 1: Overview of the steps in the pipeline. Each grey box represents a step in the pipeline.

As of publication the framework has the following functionalities:

(1) The ability to load four types of query formats:
   - MEDLINE: a popular query language used for medical literature queries and in systematic reviews;
   - PubMed: another popular query language used for medical literature queries and in systematic reviews;
   - LexesNexis: a popular query language for legal search[2] and;
   - Keyword: typical queries comprising a string of characters.
(2) The ability to use two search systems: Elasticsearch [1], and Terrier [4]; with the Galago system currently being implemented
(3) Query preprocessing methods to modify queries such as removing alpha-numeric characters or numbers (extendible to other character sets), and Boolean query transformation methods to, for example, modify logical operators, or simplify Boolean expressions (extendible),
(4) Perform measurements on queries, such as counting the number of terms, clauses, or logical operators in Boolean queries and computing query performance predictor scores. In particular, the following have been implemented:
   - Pre-retrieval predictors: Average Inverse Collection Term Frequency, Average; Sum; Max; and Standard Deviation of Inverse Document Frequency, Sum and Max Collection Query Similarity, Simplified Clarity Score and Query Scope.
   - Post-retrieval predictors: Weighted Information Gain, Weighted Entropy Gain, Normalised Query Commitment, and Clarity Score.

[2]See implementation at https://github.com/dan-locke/lexes

```
1. MMSE*.ti,ab.
2. sMMSE.ti,ab.
3. Folstein*.ti,ab.
4. MiniMental.ti,ab.
5. "mini mental stat*".ti,ab.
6. or/1-5
```

Figure 2: A typical Medline query. Note the operator on line 6 refers to the keywords between lines 1 through 5, the field restrictions (.ti,ab.), and the explicit stemming (*).

   This framework can be extended to perform other experiments, for example implementing retrieval functions.
(5) Retrieval results can be evaluated by specifying any combination of the number of documents retrieved, the number of relevant documents, the total number of relevant documents retrieved, precision and recall (in addition to other standard evaluation measures that can be implemented with ease or by integrating a tool such as `trec_eval`). The result of the measurements and evaluation can also be output in either comma separated file format (`csv`) or `JSON` notation.

Each of these components: query formats, search systems, preprocessing, Boolean transformations, measurements, evaluation measures, and output options can be extended.

2.1 Common Query Representation (cqr)

The common query representation module is used to simplify experiments in the rest of the pipeline. It forms the basis for how queries are represented and specifies how they can be transformed. The representation is similar to that of the Elasticsearch DSL. There are two possible representations of a query: the first is a keyword which is similar to what is seen in web search — a string of characters restricted to some fields; the second is a Boolean query — which combines keyword queries with logical operators. A CQR query in human-readable notation takes the form of `JSON`. An example of a common query representation of a typical MEDLINE query (Figure 2) is visualised in Figure 3.

The code for the CQR component is made available at https://github.com/hscells/cqr

2.2 Query Parser/Compiler (transmute)

When replicating a domain-specific IR task, it is often required to use a specific query language [3, 6]. These queries can either be manually translated by hand or automatically by a parser/compiler into a target query language for use in an IR system. The `transmute` library and command-line tool is a parser/compiler for queries from one query language to another. Currently, `transmute` can transform Medline and PubMed queries, and CQR queries into Elasticsearch queries (the Terrier query language is currently being implemented).

The code for the query parser/compiler component is made available at https://github.com/hscells/transmute

2.3 Experiment Pipeline (groove)

The `groove` library provides abstractions and implementations for performing IR experiments. A `groove` pipeline comprises a query source (the format of the queries), a statistic source (a source for computing IR statistics; i.e. a search engine such as Elasticsearch

```
{
    "operator": "or",
    "children": [
        {
            "query": "MMSE*",
            "fields": ["title", "text"],
            "options": {
                "truncated": true
            }
        },
        {
            "query": "sMMSE",
            "fields": ["title", "text"],
        },
        {
            "query": "Folstein*",
            "fields": ["title", "text"],
            "options": {
                "truncated": true
            }
        },
        {
            "query": "MiniMental",
            "fields": ["title", "text"]
        },
        {
            "query": "\"mini mental stat*\"",
            "fields": ["title", "text"],
            "options": {
                "truncated": true
            }
        }
    ]
}
```

Figure 3: A CQR query in JSON notation. The original query (Figure 2) has been parsed and compiled into the one above.

or Terrier), preprocessing steps (e.g., lowercase, stemming, stopword removal), any measurements to make (e.g., query performance predictors, retrieval results, evaluation criteria), and any output formats (e.g., JSON, csv). Each component in the pipeline is extendible and well documented.

Currently, groove can load Medline and PubMed queries (query languages commonly used in systematic reviews) via transmute, and LexesNexis queries (a query language commonly used in legal IR) via an implementation integrated into groove.

The code for the experiment pipeline component is made available at https://github.com/hscells/groove

2.4 DSL Front-end (boogie)

The boogie command-line utility provides higher level access to groove. It allows for the specification of a pipeline to be written in a domain specific language (DSL) that groove can then execute. An example specification pipeline file can be seen in Figure 4. At a high level, the purpose of this pipeline is to compute measurements for MEDLINE queries between a number of query performance predictors and the performance of the actual effectiveness of the queries in terms of precision and recall. At a deeper level, reading the pipeline from top to bottom: the type of queries are specified (medline), the source of statistical information (i.e. search engine) is configured to point to an Elasticsearch instance, the list of query performance predictors measurements are listed followed by the evaluation measures to record, and, finally, the last item specifies how the results of the pipeline should be output and in which formats (the trec_eval-style results file is also output so as to record the retrieval results.

This pipeline file can be distributed or shared to allow others to see exactly which steps occurred and to provide a high-level view of the experiment. In addition to this, rapid configuration and modification can be performed, allowing, for example, different parameters of retrieval functions to be changed. Finally, adding language features to the DSL (e.g., adding new output formats or measurements) has been made more convenient by using a dependency-injection style system for registering components.

The code for the domain specific language component is made available at https://github.com/hscells/boogie

3 IMPACT

To the best of the authors knowledge, such a tightly integrated collection of tools to perform domain specific IR experiments does not exist. Some tools such as Galago incorporate evaluation into a basic pipeline, the documentation of such tools is however limited and extending it requires expertise with the system.

While the focus of the framework is domain specific applications (i.e. the tooling and libraries are built for this purpose), this is not a limitation of the framework. For example, broader IR experiments such as ad-hoc web retrieval experiments can also be performed. The key factor in the framework is the separation of the IR system from the experiment specification. In this way, the system is configured in a typical way and the code that performs experiments is independent of that. This separation of the search system codebase and the experiment codebase enables researchers to more easily compare two systems, implement new features for existing systems, and rapidly prototype IR experiments.

4 USAGE

This experiment pipeline framework can be used in two ways. The first is a command-line utility whereby a pipeline is specified as in Figure 4. The second is programmatic access, where a pipeline is constructed directly in groove using Go. An example of the library usage can be seen in Figure 5. A web version that enables experiments to be designed through an intuitive interface is also in development for users with limited programming or command-line knowledge.

Using the boogie DSL, one can perform experiments that would otherwise be non-trivial. For example, to compare two systems one can switch the configuration item from one search system to another with no additional code to write. Another benefit of the boogie DSL is the ability to share and record experiments in a human-readable format. We hope that this tool can lower the barrier to entry and reduce redundant code written for IR experiments.

All four components of this framework are available as a downloadable interface at http://ielab.io/querylab.

```
{
  "query": {
    "format": "medline"
  },
  "statistic": {
    "source": "elasticsearch",
    "options": {
      "hosts": ["http://localhost:9200"],
      "index": "medline",
      "field": "abstract",
      "scroll": true,
      "search": { "size": 10000, "run_name": "qpp" }
    }
  },
  "measurements": ["avg_ictf", "sum_idf", "avg_idf",
                "max_idf", "std_idf", "clarity_score"],
  "evaluation": [
    { "evaluate": "precision" },
    { "evaluate": "recall" }
  ],
  "output": {
    "evaluations": {
      "qrels": "medline.qrels",
      "formats": [
        {
          "format": "json",
          "filename": "medline_qpp_eval.json"
        }
      ]
    },
    "measurements": {
      "formats": [
        {
          "format": "json",
          "filename": "medline_qpp.json"
        }
      ]
    },
    "trec_results": {
      "output": "medline_qpp.results"
    }
  }
}
```

Figure 4: An example `boogie` pipeline specification.

```
// Construct the pipeline.
pipelineChannel := make(chan groove.PipelineResult)
p := pipeline.NewGroovePipeline(
    query.NewTransmuteQuerySource(
    query.MedlineTransmutePipeline),
    stats.NewElasticsearchStatisticsSource(
            stats.ElasticsearchHosts(
                "http://localhost:9200"),
        stats.ElasticsearchIndex("medline"),
        stats.ElasticsearchField("abstract"),
        stats.ElasticsearchScroll(true),
        stats.ElasticsearchSearchOptions(
            stats.SearchOptions{
                Size:    10000,
                RunName: "qpp",
            })),
    pipeline.Measurement(preqpp.AvgICTF, preqpp.SumIDF,
                        preqpp.AvgIDF, preqpp.MaxIDF,
                        preqpp.StdDevIDF,
                        postqpp.ClarityScore),
    pipeline.Evaluation(eval.PrecisionEvaluator,
                         eval.RecallEvaluator),
    pipeline.MeasurementOutput(output.
                               JsonMeasurementFormatter),
    pipeline.EvaluationOutput("medline.qrels",
                          output.JsonEvaluationFormatter),
    pipeline.TrecOutput("medline_qpp.results"))

// Execute it on a directory of queries. A pipeline
   executes queries in parallel.
go p.Execute("./medline", pipelineChannel)

for {
    // Continue until completed.
    result := <-pipelineChannel
    if result.Type == groove.Done {
        break
    }
    switch result.Type {
    case groove.Measurement:
        // Process the measurement outputs.
        err := ioutil.WriteFile("medline_qpp.json",
        bytes.NewBufferString(result.Measurements[0])
            .Bytes(), 0644)
        if err != nil {
            log.Fatal(err)
        }
    case groove.Evaluation:
        // Process the evaluation outputs.
        err := ioutil.WriteFile("medline_qpp_eval.json",
            bytes.NewBufferString(result.Evaluations[0])
            .Bytes(), 0644)
        if err != nil {
            log.Fatal(err)
        }
    }
}
```

Figure 5: Example of using the programmatic interface with the groove library. The result of this code has the same effect as that of the pipeline in Figure 4.

5 REQUIREMENTS FOR PRESENTATION OF THE DEMONSTRATION

Along with the wireless network access and poster mount provided, we kindly request two large monitors with two HDMI cables. We plan to demonstrate the software interactively, allowing attendees on one monitor to formulate, edit, and run pipelines and see the results through a custom interface. On the second monitor, we plan to display aspects of the code attendees would like to view in more detail.

REFERENCES

[1] S. Banon. 2011. Elasticsearch: An open source, distributed, RESTful search engine.
[2] S. Cronen-Townsend, Y. Zhou, and W. B. Croft. 2002. Predicting query performance. In *SIGIR*.
[3] D. Locke, G. Zuccon, and H. Scells. 2017. Automatic Query Generation from Legal Texts for Case Law Retrieval. In *Information Retrieval Technology*.
[4] I. Ounis, G. Amati, Plachouras V., B. He, C. Macdonald, and Johnson. 2005. Terrier Information Retrieval Platform. In *ECIR*.
[5] R. Pike. 2009. The Go Programming Language. (2009).
[6] H. Scells, G. Zuccon, B. Koopman, A. Deacon, L. Azzopardi, and S. Geva. 2017. Integrating the framing of clinical questions via PICO into the retrieval of medical literature for systematic reviews. In *CIKM'17*.