Pytrec_eval: An Extremely Fast Python Interface to trec_eval[*]

Christophe Van Gysel
University of Amsterdam
Amsterdam, The Netherlands
chris@stophr.be

Maarten de Rijke
University of Amsterdam
Amsterdam, The Netherlands
derijke@uva.nl

ABSTRACT

We introduce pytrec_eval, a Python interface to the trec_eval information retrieval evaluation toolkit. pytrec_eval exposes the reference implementations of trec_eval within Python as a native extension. We show that pytrec_eval is around one order of magnitude faster than invoking trec_eval as a sub process from within Python. Compared to a native Python implementation of NDCG, pytrec_eval is twice as fast for practically-sized rankings. Finally, we demonstrate its effectiveness in an application where pytrec_eval is combined with Pyndri and the OpenAI Gym where query expansion is learned using Q-learning.

CCS CONCEPTS

• **Information systems → Evaluation of retrieval results**;

KEYWORDS

IR evaluation, toolkits

ACM Reference Format:

Christophe Van Gysel and Maarten de Rijke. 2018. Pytrec_eval: An Extremely Fast Python Interface to trec_eval. 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/10.1145/3209978.3210065

1 INTRODUCTION

Evaluation is a crucial component of any information retrieval (IR) system [2]. Reusable test collections and off-line evaluation measures [7] have been the dominating paradigm for experimentally validating IR research for the last 30 years. The popularity and ubiquity of off-line IR evaluation measures is partly due to the Text REtrieval Conference (TREC) [5]. TREC led to the development of the trec_eval[1] software package that is the standard tool for evaluating a collection of rankings. The trec_eval tool allows IR researchers to easily compute a large number of evaluation measures using standardized input and output formats. For a document collection, a test collection of queries with query/document relevance information (i.e., qrel) and a set of rankings generated by a particular IR system (i.e., a system run) for the test collection queries, trec_eval outputs a standardized output format containing evaluation measure values. The adoption of trec_eval as an integral part of IR research has led to the following benefits: (a) *standardized formats* for system rankings and query relevance information such that different research groups can exchange experimental results with minimal communication, and (b) *open-source reference implementations of evaluation measures*—provided by a third party (i.e., NIST)—that promotes transparent and consistent evaluation.

While the availability of trec_eval has brought many benefits to the IR community, it has the downside that it is available only as a standalone executable that is interfaced by passing files with rankings and ground truth information. In recent years, the Python programming language has risen in popularity due to its feature richness (i.e., scientific libraries and data structures) and holistic language design [3]. Research progresses at a rate proportional to the time it takes to implement an idea, and consequently, scripting languages (e.g., Python) are preferred over conventional programming languages [6]. Within IR research, retrieval systems are often implemented and optimized using Python (e.g., [4, 9]) and for their evaluation trec_eval is used. However, invoking trec_eval from Python is expensive as it involves (1) serializing the internal ranking structures to disk files, (2) invoking trec_eval through the operating system, and (3) parsing the trec_eval evaluation output from the standard output stream. This workflow is unnecessarily inefficient as it incurs (a) a double I/O cost when the ranking is first serialized by the Python script and subsequently parsed by trec_eval, and (b) a context-switching overhead as the invocation of trec_eval needs to be processed by the operating system.

We introduce pytrec_eval to counter these excessive efficiency costs and avoid a wild growth of ad-hoc Python-based evaluation measure implementations. pytrec_eval builds upon the trec_eval source code and exposes a Python-first interface to the trec_eval evaluation toolkit as a native Python extension. Rankings constructed in Python can directly be passed to the evaluation procedure, without incurring disk I/O costs; evaluation is performed using the original trec_eval implementation. Due to pytrec_eval's implementation as a native Python extension, context-switching overheads are avoided as the evaluation procedure and its invocation reside within the same process. Next to improved efficiency, pytrec_eval brings the following benefits: (a) current and future reference trec_eval implementations of IR evaluation measures are available within Python, and (b) as the evaluation measures are implemented in C, their execution are typically faster than native Python-based alternatives. The main purpose of this paper is to describe pytrec_eval, provide empirical evidence of the speedup that pytrec_eval delivers, and showcase the use of pytrec_eval in a reinforcement learning application. We ask the following questions: **(RQ1)** What speedup do we obtain when using pytrec_eval over trec_eval (serialize-invoke-parse workflow)? **(RQ2)** How fast is pytrec_eval compared to native Python implementations of IR evaluation measures? We also present a demo application that combines Pyndri [9] and pytrec_eval in a query formulation

[*] Open-source implementation is available at https://github.com/cvangysel/pytrec_eval.
[1] https://github.com/usnistgov/trec_eval

Permission to make digital or hard copies of all or part 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 components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGIR '18, July 8–12, 2018, Ann Arbor, MI, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5657-2/18/07…$15.00
https://doi.org/10.1145/3209978.3210065

```python
qrel = {'q1': {'d1': 1, 'd2': 0},
        'q2': {'d2': 1}}
run = {'q1': {'d1': 0.5, 'd2': 2.0},
       'q2': {'d1': 0.5, 'd2': 0.6}}

evaluator = pytrec_eval.RelevanceEvaluator(
        qrel, {'map', 'ndcg'})
result = evaluator.evaluate(run)

# result equals
# {'q1': {'map': 0.5, 'ndcg': 0.6309297535714575},
#  'q2': {'map': 1.0, 'ndcg': 1.0}}
```

Code snippet 1: Minimal example of how `pytrec_eval` can be used to compute IR evaluation measures. Evaluation measures (NDCG, MAP) are computed for two queries—`q1` and `q2`—and two documents—`d1` and `d2`—where for `q2` we only have partial relevance (`d1` is assumed to be non-relevant).

reinforcement learning setting and provide the environment and the reward signal, integrated within the OpenAI Gym [1].

2 EVALUATING USING PYTREC_EVAL

The `pytrec_eval` library has a minimalistic design. Its main interface is the `RelevanceEvaluator` class. The `RelevanceEvaluator` class takes as arguments (1) query relevance ground truth, a dictionary of query identifiers to a dictionary of document identifiers and their integral relevance level, and (2) a set of evaluation measures to compute (e.g., `ndcg`, `map`). Code snippet 1 shows a minimal example on how `pytrec_eval` can be used to evaluate a ranking. Rankings are encoded by a mapping from document identifiers to their retrieval scores. Internally, `pytrec_eval` sorts the documents in decreasing order of retrieval score. This behavior mimics the implementation of `trec_eval`, which ignores the order of documents within the user-provided file, and only considers the document scores. Similar to `trec_eval`, document ties, which occur when two documents are assigned the same score, are broken by secondarily sorting on document identifier. Query relevance ground truth is passed to `pytrec_eval` in a similar way to document scores, where relevance is encoded as an integer rather than a floating point value.

Beyond measures computed over the full ranking of documents, `pytrec_eval` also supports measures computed up to a particular rank k. The values of k are the same as the ones used by `trec_eval`. For example, measures `ndcg_cut` and `P` correspond to NDCG@k and precision@k, respectively, with $k = 5, 10, 15, 20, 30, 100, 200, 500, 1000$. The set of supported evaluation measures is stored in the `pytrec_eval.supported_measures` property and the identifiers are the same as used by `trec_eval` (i.e., running `trec_eval` with arguments `-m ndcg_cut --help` will show documentation for the NDCG@k measure). To mimic the behavior of `trec_eval` to compute all known evaluation measures (i.e., passing argument `-m all_trec` to `trec_eval`), just instantiate `RelevanceEvaluator` with `pytrec_eval.supported_measures` as the second argument.

3 BENCHMARK RESULTS

As demonstrated above, `pytrec_eval` conveniently exposes popular IR evaluation measures within Python. However, the same functionality could be exposed by invoking `trec_eval` in a serialize-invoke-parse workflow—or—by implementing the evaluation measure natively in Python. In this section we provide empirical benchmark results that show that `pytrec_eval`, beyond its convenience, is also faster at computing evaluation measures than these two alternatives (i.e., invoking `trec_eval` or native Python).

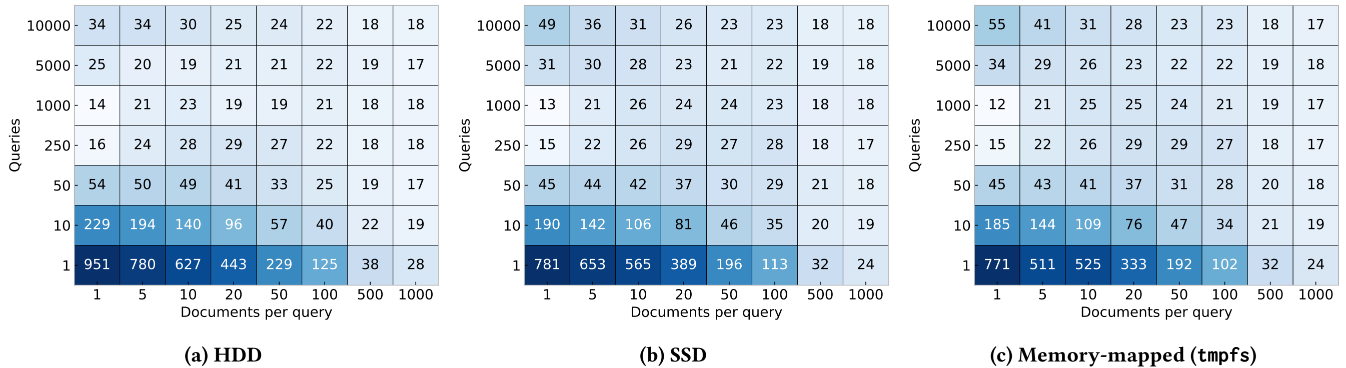
Experimental setup. For every hyperparameter configuration, the runtime measurement was repeated 20 times and the average runtime is reported. Speedup denotes the ratio of the runtime of the alternative method (i.e., `trec_eval` or native Python) over the runtime of `pytrec_eval` and consequently, a speedup of 1.0 means that both methods are equally fast. When invoking `trec_eval` using the serialize-invoke-parse workflow, rankings are written from Python to storage without sorting, as `trec_eval` itself sorts the rankings internally. The resulting evaluation output is read from `stdout` to a Python string and we do not extract the measure values, as different parsing strategies can lead to large variance in runtime. For the native Python implementation, we experimented with different open-source implementations of the NDCG measure and adapted the fastest implementation as our baseline. The implementation does not make use of NumPy or other scientific Python libraries as (a) we wish to compare to native Python directly and (b) the NumPy-based implementations we experimented with were less efficient than the native implementation we settled with, as NumPy-based implementations require that the rankings are encoded in dense arrays before computing evaluation measures. The evaluated rankings and ground-truth were synthesized by assigning every document a distinct ranking score in $\mathbb{N}$ and a relevance level of 1. This allows us to evaluate different evaluation measure implementations with rankings and query sets of different sizes. Experiments were run using a single Intel Xeon CPU (E5-2630 v3) clocked at 2.4GHz, DDR4 RAM clocked at 2.4GHz, an Intel SSD (DC S3610) with sequential read/write speeds of 550MB/s and 450MB/s, respectively, and a hard disk drive (Seagate ST2000NX0253) with a rotational speed of 7200 rpm. All code used to run our experiments is available under the MIT open-source license.[2]

Results. We now answer our research questions by comparing the runtime performance of `pytrec_eval` to `trec_eval` (**RQ1**) and a native Python implementation (**RQ2**).

RQ1 What speedup do we obtain when using `pytrec_eval` over `trec_eval` (serialize-invoke-parse workflow)?

Fig. 1 shows matrices of speedups of `pytrec_eval` over `trec_eval` obtained using different storage types (increasing order of throughput capacity): a regular hard disk drive (HDD), a solid state drive (SSD) and a memory-mapped file system (`tmpfs`). For the degenerate case where we have a single query and a single returned document, we observe that there is a clear difference between the different storages. In particular, we can see that `tmpfs` is faster than SSD, and in turn, SSD is faster than the HDD. However, for larger configurations (upper right box in every grid; 10,000 queries with 1,000 documents) we see that the difference between the storage types fades away and that `pytrec_eval` always achieves a speedup of at least 17 over `trec_eval`. This is because (a) starting the serialization (e.g., disk seek time) is expensive (as can be seen in the left-lower box of every grid), but that cost is quickly overshadowed by (b) the cost of context switching between processes. In the case of `pytrec_eval`, however, context switching is avoided as all logic runs as part of the same process. Consequently, we can conclude that `pytrec_eval` is at least one order of magnitude faster than invoking `trec_eval` using a serialize-invoke-parse workflow.

[2]The benchmark code can be found in the `benchmarks` sub-directory of the `pytrec_eval` repository; see the footnote on the first page.



(a) HDD (b) SSD (c) Memory-mapped (`tmpfs`)

Figure 1: Speedup of `pytrec_eval` (down-rounded speedup in each box; runtime measured as average over 20 repetitions) compared to invoking `trec_eval` using a serialize-invoke-parse workflow (§1) for different numbers of queries, different numbers of ranked documents per query, and using different types of storage (hard disk drive, solid state drive and random access memory) for serializing the rankings and query relevance ground truth.

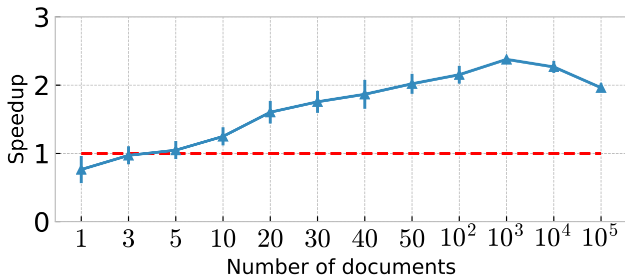


Figure 2: Speedup of `pytrec_eval` over a native Python implementation of the NDCG evaluation measure (we report average speedup and its standard deviation over 20 repetitions). For practically-sized rankings, `pytrec_eval` is consistently faster than the native Python implementation.

RQ2 How fast is `pytrec_eval` compared to native Python implementations of IR evaluation measures?

Fig. 2 shows the speedup of `pytrec_eval` over a Python-native implementation of NDCG for a single query and a varying number of documents. Here we see that for extremely short rankings (1–3 documents), the native implementation outperforms `pytrec_eval`. However, for rankings consisting of 5 documents or more, we can see that `pytrec_eval` provides a consistent performance boost over the native implementation. The reason for the sub-native performance of `pytrec_eval` for very short rankings is because—before `pytrec_eval` computes evaluation measures—rankings need to be converted into the internal C format used by `trec_eval`. The Python-native implementation does not require this transformation, and consequently, can thus be slightly faster when rankings are very short. However, it is important to note that short rankings are uncommon in IR and that the average ranking consists of around 100 to 1,000 documents. We conclude that `pytrec_eval` is faster than native Python implementations for practically-size rankings.

4 EXAMPLE: Q-LEARNING

We showcase the integration of the Pyndri indexing library [9] and `pytrec_eval` within the OpenAI Gym [1], a reinforcement learning library, for the task of query expansion. In particular, we use Pyndri to rank documents according to a textual query and subsequently evaluate the obtained ranking using `pytrec_eval`. The reinforcement learning agent navigates an environment where actions correspond to adding a term to the query. Rewards are given by an increase or decrease in evaluation measure (i.e., NDCG). The goal is for the agent to learn a policy π^* that optimizes the expected value of the total reward. For the purpose of this demonstration of software interoperability, we synthesize a test collection in order to (1) limit the computational complexity that arises from real-world collections, and (2) to give us the ability to create an unlimited number of training queries and relevance judgments.

Document collection. We construct a synthetic document collection D, of a given size $|D| = 100$, following the principles laid out by Tague et al. [8]. For a given vocabulary size $|V| = 10{,}000$, we construct vocabulary V consisting of symbolic tokens. We sample collection-wide unigram ($|V|$ parameters) and bigram ($|V|^2$ parameters) pseudo counts from an exponential distribution ($\lambda = 1.0$). This incorporates term specificity within our synthetic collection, as only few term uni- and bigrams will be frequent and most will be infrequent. These pseudo counts will then serve as the concentration parameters of Dirichlet distributions from which we will sample a uni- and bigram language model for every document. We create $|D|$ documents as follows. For every document d, given the average document length $\mu_d = 200$, we sample its document size, $|d|$, from a Poisson with mean μ_d. We then sample two language models—one for unigrams $P(w \mid d)$ and another for bigrams $P((x, y) \mid d)$—from a Dirichlet distribution where the concentration parameters we defined earlier for the whole collection. The document is then constructed as follows. Until we have reached $|d|$ tokens, we repeat the following: (a) sample an n-gram size from a predefined probability distribution ($P(n = 1) = 0.9$, $P(n = 2) = 0.1$), and subsequently, (b) sample an n-gram from the corresponding language model. We truncate a document if it exceeds its pre-defined length $|d|$.

Query collection. Once we obtained our synthetic document collection D, we proceed by constructing our query set Q, of a given size $|Q| = 100{,}000$, as follows. For every query q to be constructed, we select $r = 5$ documents uniformly at random from D and denote these as the set of relevant documents $R_q \subset D$ for query q. Given the average query length $\mu_q = 3$, the length of query q, $|q|$, is then sampled from a Poisson distribution with mean μ_q. We write $P(w \mid R_q)$ and $P(w \mid D)$ to denote the empirical language models

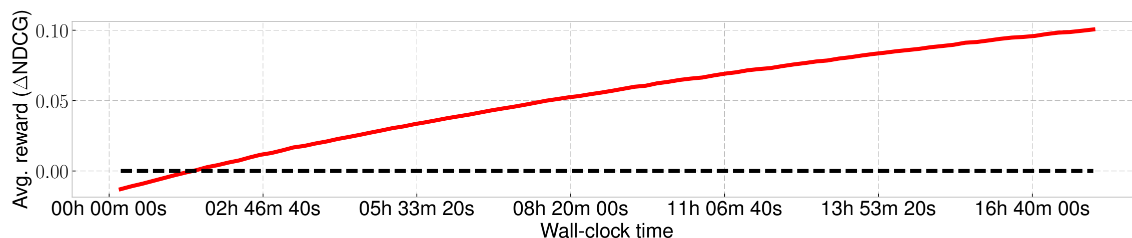


Figure 3: Average reward (ΔNDCG) obtained by the Q-learning algorithm over time while training the reinforcement learning agent. The agent learns to select vocabulary terms that improve retrieval effectiveness for the set of 100k training queries.

estimated from concatenating the relevant documents for query q and from concatenating all documents in the collection D (i.e., the collection language model), respectively. The $|q|$ terms of query q are sampled with replacement from $P\left(w \mid R_q\right)(1.0 - P(w \mid D))$, such that terms specific to R_q and uncommon in D are selected.

Environment. For each query q, the environment is initialized to the state where only the query terms are present. At any given state, the agent can then choose to expand the query terms with any unigram term from the vocabulary V in addition to a null operation action. Rankings are obtained by querying the Indri search engine using Pyndri, using a Dirichlet language model (μ = 2,500), and obtaining a ranking of the top-10 documents. The reward of choosing an action is the ΔNDCG that is obtained by expanding the query with the chosen term. As observation, the agent receives a binary vector indicating which terms of the vocabulary V occur at least once in the current expanded query. After 5 actions—or a perfect NDCG (i.e., 1.0) is achieved—the episode terminates.

Reinforcement learning agent. We learn an optimal policy tabular π^* using Q-learning where the initial values of the $Q(\cdot)$ are initialized to zero. We set the learning rate $\alpha = 0.1$ and the discount factor $\gamma = 0.95$. During learning, we maintain an ϵ-greedy strategy with $\epsilon = 0.05$. Fig. 3 shows the average reward obtained while training an agent on the reinforcement learning problem defined above. The average reward obtained by the agent increases over time. In particular, this example showcases that different IR libraries (Pyndri, `pytrec_eval`) can easily be integrated with machine learning libraries (OpenAI Gym) to quickly prototype ideas. An essential part here is that expensive operations (i.e., ranking and evaluation) are performed in efficient low-level languages, whereas prototyping occurs in the high-level Python scripting language. All code used in this example is available under the MIT open-source license.[3]

5 CONCLUSIONS

In this paper we introduced `pytrec_eval`, a Python interface to `trec_eval`. `pytrec_eval` builds upon the `trec_eval` source code and exposes a Python-first interface to the `trec_eval` evaluation toolkit as a native Python extension. This allows for convenient and fast invocation of IR evaluation measures directly from Python. We showed that `pytrec_eval` is around one order of magnitude faster than invoking `trec_eval` in a serialize-invoke-parse workflow as it avoids the costs associated with (1) the serialization of the rankings to storage, and (2) operation system context switching. Compared to a native Python implementation of NDCG, `pytrec_eval` is approximately twice as fast for practically-sized rankings (100 to 1,000 documents). In addition, we showcased the integration of Pyndri [9] and `pytrec_eval` within the OpenAI Gym [1] and showed that all three modules can be combined to quickly prototype ideas.

In this paper, we used a tabular function during Q-learning; other functional forms—such as a deep neural network—can also be used. Pyndri and `pytrec_eval` expose common IR operations through a convenient Python interface. Beyond the convenience that both modules provide, an important design principle is that expensive operations (e.g., indexing, ranking) are performed using efficient low-level languages (e.g., C), while Python takes on the role of an instructor that links the expensive operations. Future work consists of exposing more IR operations as Python libraries and allowing more interoperability amongst modules. For example, currently Pyndri converts its internal Indri structures to Python structures, which are then again converted back to internal `trec_eval` structures by `pytrec_eval`. A closer integration of Pyndri and `pytrec_eval` could result in even faster execution times as both can communicate directly—in cases where one is only interested in the evaluation measures and not the rankings—rather than through Python.

Acknowledgements. This research was supported by Ahold Delhaize, Amsterdam Data Science, the Bloomberg Research Grant program, the China Scholarship Council, the Criteo Faculty Research Award program, Elsevier, the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement nr 312827 (VOX-Pol), the Google Faculty Research Awards program, the Microsoft Research Ph.D. program, the Netherlands Institute for Sound and Vision, the Netherlands Organisation for Scientific Research (NWO) under project nrs CI-14-25, 652.002.001, 612.001.551, 652.001.003, and Yandex. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

REFERENCES

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI gym, 2016.
[2] D. Harman. Information retrieval evaluation. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 3(2):1–119, 2011.
[3] H. Koepke. Why python rocks for research. https://www.stat.washington.edu/~hoytak/_static/papers/why-python.pdf, 2010. Accessed February 12, 2018.
[4] D. Li and E. Kanoulas. Bayesian optimization for optimizing retrieval systems. In *WSDM*. ACM, February 2018.
[5] NIST. Text retrieval conference, 1992–2017.
[6] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct. 2000.
[7] M. Sanderson. Test collection based evaluation of information retrieval systems. *Foundations and Trends in Information Retrieval*, 4(4):247–375, 2010.
[8] J. Tague, M. Nelson, and H. Wu. Problems in the simulation of bibliographic retrieval systems. In *SIGIR*, pages 236–255. ACM, June 1980.
[9] C. Van Gysel, E. Kanoulas, and M. de Rijke. Pyndri: a python interface to the indri search engine. In *ECIR*, pages 744–748. Springer, April 2017.

[3] The reinforcement learning code can be found in the `examples` sub-directory of the `pytrec_eval` repository; see the footnote on the first page.