ABSTRACT
Axiomatic approaches to information retrieval have played a key role in determining basic constraints that characterize good retrieval models. Beyond their importance in retrieval theory, axioms have been operationalized to improve an initial ranking, to "guide" retrieval, or to explain some model’s rankings. However, recent open-source retrieval frameworks like PyTerrier and Pyserini, which made it easy to experiment with sparse and dense retrieval models, have not included any retrieval axiom support so far.

To fill this gap, we propose ir_axioms, an open-source Python framework that integrates retrieval axioms with common retrieval frameworks. We include reference implementations for 25 retrieval axioms, as well as components for preference aggregation, re-ranking, and evaluation. New axioms can easily be defined by implementing an abstract data type or by intuitively combining existing axioms with Python operators or regression. Integration with PyTerrier and ir_datasets makes standard retrieval models, corpora, topics, and relevance judgments—including those used at TREC—immediately accessible for axiomatic experimentation. Our experiments on the TREC Deep Learning tracks showcase some potential research questions that ir_axioms can help to address.

CCS CONCEPTS
• Information systems → Retrieval models and ranking.

KEYWORDS
Axiomatic Thinking for Information Retrieval; Software Framework; Software Toolkit; Evaluation

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.

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8732-3/22/07... $15.00
https://doi.org/10.1145/3477495.3531743

1 INTRODUCTION AND BACKGROUND
In the field of information retrieval, many open-source frameworks and toolkits are available that provide a convenient way to experiment with all kinds of retrieval models on standard corpora. One of the early examples is the Terrier platform [29]—open-sourced in 2004—that supports retrieval and learning-to-rank experiments on most of the document collections used at TREC; the recent PyTerrier framework [30] builds on it. Also the Lucene-based Java toolkit Anserini [45, 46] and its Python interface Pyserini [23] were developed to enable experiments with sparse retrieval models and easy access to the majority of the data used at TREC. Tightly integrated with Pyserini, PyGaggle [34] is a toolkit for neural ranking and question answering (e.g., using BERT-based [12] or T5-based [37] pairwise and pointwise rankers). Other toolkits and libraries that implement sparse and dense retrieval include MatchZoo [19], OpenNIR [26], Capreolus [47], OpenMatch [24], and Tevatron [17].

However, besides the many implemented retrieval models and evaluation metrics, so far, none of these frameworks, libraries, and toolkits include components for retrieval axioms. Proposed in the more theoretical field of axiomatic thinking for information retrieval, axioms are constraints that a “good” retrieval model should fulfill—often in the form of ranking preferences between documents. Recently, axioms were also practically applied in re-ranking experiments [20], to meta-learn how to combine the scores of different retrieval models [2], to regularize neural retrieval models [40], or to analyze and explain rankers [8, 16, 27, 38, 42]. To close the gap of axiom support in major retrieval toolkits, we develop ir_axioms: a Python framework to conduct retrieval experiments with axioms.1

Following the best practices of existing retrieval frameworks and toolkits, ir_axioms is a modular software that allows to effortlessly build pipelines and combine axioms using a set of special operators. Tightly integrated with PyTerrier and ir_datasets [28], ir_axioms can be used with a wide range of retrieval models, indexers, collections, and evaluation functions. All axioms are implemented as class objects that can be parameterized (e.g., to change an axiom’s preconditions). Further axioms can easily be defined by extending the Axiom class. A KwikSortReRanker module helps to implement axiomatic re-ranking experiments, and the AxiomaticExperiment module provides functionality for axiomatic analyses. An interesting feature of ir_axioms is to localize document pairs in some ranking that are incorrectly ordered according to manual relevance judgments along with the respective axiom preferences. This allows for a deeper analysis and better understanding of a retrieval model’s incorrect ranking decisions. We demonstrate the functionality of ir_axioms on such example use cases in the scenario of the TREC 2019 and 2020 Deep Learning tracks [10, 11].
2 AXIOMS FOR INFORMATION RETRIEVAL

Retrieval axioms are basic constraints that a good ranking function should fulfill. For instance, the axiom TFC1 states that documents with more query term occurrences should be ranked higher [13]. This constraint obviously can be "wrong" (e.g., if a document with more occurrences is a near-duplicate of another document already high in the ranking, a document with fewer occurrences might be preferable). The term 'retrieval axiom' thus needs to be interpreted a bit different than, for instance, axioms in geometry or probability theory. Still analyzing how well ranking functions satisfy specific retrieval axioms has led to improved ranking functions [25]—but also to the identification of incompatibility axiom sets [18].

Table 1 shows a selection of retrieval axioms from the literature. Making use of term frequency, term discrimination power (e.g., idf, or similarity (e.g., WordNet-based), many of these axioms induce a partial order on documents: a preference $d_1 >_A d_2$ implies that $d_1$ should be ranked higher than $d_2$ according to axiom $A$.

More recent studies have tried to integrate axiomatic ideas directly into the retrieval process: (1) improving an initial retrieval result via re-ranking according to weighted axiom preferences [6, 20], (2) using axioms as regularization loss in neural models [40], and (3) explaining neural rankings [8, 38, 42]. For an easier setup of such studies, ir_axioms implements 25 axioms and makes them available for experiments with standard retrieval toolkits.

Table 2 shows the axioms currently implemented in ir_axioms. Following Hagen et al. [20] and Völiske et al. [42], our implementations try to incorporate an axiom’s formalized idea while also making it applicable in practical settings. We thus reformulate the axioms to work for arbitrary queries (e.g., the original TFC1 requires single-term queries), to express pairwise preferences (e.g., DIV), etc. Also, we include parameters to relax or strengthen some pre- and filter conditions (e.g., allowing for some small relative document length difference in TFC1). For axioms employing term similarity (REG and STMC1/2), we provide two variants based on WordNet synsets [32] or on fastText embeddings [4]. Axioms that cannot be easily reformulated to yield practically usable pairwise ranking preferences are not included (e.g., TFC2 or LB2).

Most recent among the currently implemented axioms are the argumentativeness axioms [5, 6] that aim at queries that probably require more argumentative results (e.g., mercy killing). The axiom ArgUC favors documents with more argument units (premises and claims), the axioms QTAArg and QTPArg favor documents where the query terms appear closer to the argument units and closer to the beginning of the document, while the axiom aSLDoc favors more "readable" documents (average sentence length: 12–20 words).

Finally, we also include the axioms ORIG [20] and ORACLE. The preferences of ORIG simply follow some original ranking (e.g., a ranking returned by BM25 [39] or any other retrieval model). This enables weighting comparisons of an initial ranking to some weighted combinations of other axioms. The preferences of ORACLE are intended to follow human judgments if available (e.g., preferences between any documents from different relevance groups in TREC qrel files). This allows to incorporate "ground-truth" relevance information into experimental axiomatic settings.

Table 1: Selected axioms from previous studies (intuition, pre- (in gray) and filter conditions, concluded ranking preference).

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
<th>Example Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFC1</td>
<td>Prefer documents with more query term occurrences.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>TFC2</td>
<td>Additional query term occurrences yield smaller retrieval score improvements.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>TFC3</td>
<td>Prefer documents with occurrences of more query terms.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>LNC1</td>
<td>Penalize longer documents for non-relevant terms.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>LNC2</td>
<td>Do not prefer shorter documents when matched query term ratio is the same.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>TF-LNC</td>
<td>Revised additional query terms more than document length is penalized.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>LB1</td>
<td>Do not override the term presence–absence gap with length normalization.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>LB2</td>
<td>Repeated query term occurrence is less important than first occurrence.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>REG</td>
<td>Prefer documents covering more different query aspects.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>AND</td>
<td>Prefer documents containing all query terms.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>STMC1</td>
<td>Prefer documents with terms more similar to query terms.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>STMC2</td>
<td>Do not reward similar terms more than exact matches.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>PROX1</td>
<td>Prefer documents with shorter distance between query term pairs.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>PROX2</td>
<td>Prefer documents with earlier query term occurrences.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>PROX3</td>
<td>Prefer documents where the query occurs earlier as a phrase.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>PROX4</td>
<td>Prefer documents that contain all query terms in a shorter substring.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
<tr>
<td>PROX5</td>
<td>Prefer documents where the query terms are closer together on average.</td>
<td>$q_1$, $d_1$, $d_2$, $</td>
</tr>
</tbody>
</table>

6Complete list of changes to the original axiom formulations: https://github.com/webis-de/ir_axioms/blob/main/documentation/axioms.md
When an axiom requires access to a document’s content (i.e., text), with Pyserini [23] and PyTerrier [30]. Both Pyserini’s underlying Table 2: The axioms in ir_axioms (e.g., BM25) to learning to rank (e.g., LambdaMART [7]) and to dense retrieval (e.g., ColBERT [22]) and neural re-ranking (e.g., monoT5 and duoT5 [36]). Through PyTerrier, ir_axioms also has easy access to a variety of benchmark datasets and pre-built indices from ir_datasets [28]. PyTerrier’s data model uses a pandas DataFrame [31] to represent a set of documents, or a set of queries, or the results retrieved for each query, etc. Functions (called transformers in PyTerrier) can be implemented to transform a DataFrame (e.g., from queries to ranked documents). Special operators can be used to combine transformers in a pipeline, passing the output from one transformer to the next.

Since PyTerrier allows to extend transformers by user-defined transformer classes, in ir_axioms we include custom axiom-specific transformers that can directly be used in retrieval pipelines (cf. Table 3). For instance, a transformation \( R \rightarrow R' \) means to re-rank a ranking \( R \) by applying some modification, whereas \( R \rightarrow R_y \) denotes some feature extraction from \( R \). This way, ir_axioms deeply integrates with PyTerrier’s declarative definitions of retrieval pipelines and its transparent data model. As a result, axioms can be conveniently applied to the rankings produced by any of the many retrieval models implemented in PyTerrier.

### 3.2 Schemes to Combine and Weight Axioms

Each retrieval axiom serves as a proxy for a single important constraint that a good ranking function should fulfill. Combining different axioms (i.e., their preference decisions) into ensembles then can be effective [20]. In ir_axioms, configurable axiom groupings enable such ensembles and can act as axioms themselves, which can in turn be combined with other axioms. Combination or manipulation of axioms is governed by arithmetic and logic Python operators that are overloaded in ir_axioms to allow for declarative axiom expressions and intuitive experimentation (cf. Table 4). The examples in Listing 1 show how axioms can be combined. One possibility is a weighted linear combination of the preference values using point-wise addition and scalar multiplication:

\[
\text{pref}_{A+B}(q, d_i, d_j) = \text{pref}_A(q, d_i, d_j) + \text{pref}_B(q, d_i, d_j),
\]

\[
\text{pref}_{A \cdot A}(q, d_i, d_j) = x \cdot \text{pref}_A(q, d_i, d_j).
\]

Besides linear combinations, axioms can also be combined "conjunctively" in various flavors. By using the \&-operator or the aggregation class AndAxiom, a non-zero preference is returned iff all axioms return the same non-zero preference (all axioms agree). By using the VoteAxiom class, the conjunctive agreement can be relaxed so that each axiom "votes" for the preference individually. The majority vote is returned iff it reaches a specified threshold (passed as an argument to the VoteAxiom class; e.g., 50% for the absolute majority as in the second example in Table 5). Hence, AndAxiom is equivalent...
Table 3: Axiom-specific transformer classes, their types, and description. Following the PyTerrier transformer notations, \( R \) denotes ranking and \( f \) denotes feature extraction.

<table>
<thead>
<tr>
<th>Transformer Class</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AggregatedPreferences</td>
<td>( R \to R_f )</td>
<td>Aggregate axiom preferences for each document (cf. Section 4.3.2).</td>
</tr>
<tr>
<td>EstimatorKwikSortReranker</td>
<td>( R \to R' )</td>
<td>Train estimator for ORACLE, use it to re-rank with KwikSort (cf. Section 4.3.3).</td>
</tr>
<tr>
<td>KwikSortReranker</td>
<td>( R \to R' )</td>
<td>Re-rank using axiom preferences aggregated by KwikSort (cf. Section 4.3.1).</td>
</tr>
<tr>
<td>PreferenceMatrix</td>
<td>( R \to (R \times R)_f )</td>
<td>Compute an axiom’s preference matrix (cf. Section 4.1).</td>
</tr>
</tbody>
</table>

Table 4: Operators in \( \text{ir}_\text{axioms} \) that allow to combine, modify, and cache preferences of retrieval axioms.

- **Binary**
  - \( + \) \textit{add}: Add axiom preferences or constants.
  - \( - \) \textit{subtract}: Subtract axiom preferences or constants.
  - \( \times \) \textit{multiply}: Multiply axiom preferences by a weight.
  - \( / \) \textit{divide}: Divide axiom preferences by a weight.
  - \( | \) \textit{cascade}: Fallback if an axiom preference is 0.
  - \& \textit{conjunction}: Return preference if all axioms agree.
  - \% \textit{majority}: Absolute majority vote of involved axioms.

- **Unary**
  - \( \sim \) \textit{cache}: Cache axiom preferences on disk.
  - \( -\) \textit{nagate}: Negate axiom preferences.
  - \( +\) \textit{normalize}: Normalize axiom preferences to \((-1, 0, 1)\).

Listing 2: Normalize and cache axiom preferences.

```python
# Normalizing the combined STMC-preferences.
normalized_stmc = + (STMC1() + STMC2())

# Caching the preferences of ArgUC.
cached_arguc = ~ ArgUC()
```

To \texttt{VoteAxiom} with a required majority of 100%. When an axiom returns no preference, a cascade axiom can define a fallback:

\[
\text{pref}_{A|B}(q, d_i, d_j) = \begin{cases} 
\text{pref}_A(q, d_i, d_j) & \text{if } \text{pref}_A(q, d_i, d_j) \neq 0, \\
\text{pref}_B(q, d_i, d_j) & \text{otherwise}.
\end{cases}
\]

The last example in Listing 1 uses the ‘cascade’ operator to return the ORIG preference in case that STMC1 returns a ‘zero’ preference.

Bondarenko et al. [5] proposed three weighting schemes by which some axiom set may “overrule” an ORIG preference (i.e., the initial ranking): equal weights, majority voting, and total agreement. Table 5 shows formulations of these schemes with the combination operators and aggregation axiom classes of \( \text{ir}_\text{axioms} \).

The aggregation of axioms in \( \text{ir}_\text{axioms} \) is optimized for efficiency by early stopping the preference computation in a cascade (\texttt{CascadeAxiom}) or a logical conjunction (\texttt{AndAxiom}). For example, if in a conjunction \( A_1 \& \ldots \& A_n \) the \( A_1 \)-preference is 0, then the conjunctive preference \( \text{pref}_{A_1 \& \ldots \& A_n}(q, d_i, d_j) \) must also be 0, and the preference computation can be skipped for all other axioms.

### 3.3 Preference Normalization and Caching

Besides the above binary operators, \( \text{ir}_\text{axioms} \) also includes unary operators: negation, normalization, and caching (cf. Table 4). Negating an axiom (unary ‘\( \sim \)’) simply inverts its preferences:

\[
\text{pref}_{A}(q, d_i, d_j) = -\text{pref}_A(q, d_i, d_j).
\]

Since preference values are not restricted (e.g., \( \text{pref}_A(q, d_i, d_j) = 42 \) and \( \text{pref}_A(q, d_i, d_j) = 0.815 \) both express \( d_i > A d_j \)), it can be advisable to sometimes re-calibrate them (e.g., after linear combinations). In \( \text{ir}_\text{axioms} \), the preference values of any axiom or combination can be normalized to \((1, 0, -1)\) using the unary ‘\( +\)’:

\[
\text{pref}_{A}(q, d_i, d_j) = \begin{cases} 
1 & \text{if } \text{pref}_A(q, d_i, d_j) > 0, \\
-1 & \text{if } \text{pref}_A(q, d_i, d_j) < 0, \\
0 & \text{if } \text{pref}_A(q, d_i, d_j) = 0.
\end{cases}
\]

The first example in Listing 2 normalizes an STMC-combination.

A final unary operator ‘\( \sim \)’ offers some convenience functionality for experiments in which the same axioms are applied more than once on similar document sets (e.g., rankings of different retrieval models for the same query with many overlapping documents). To avoid re-computing the same axiom preferences over and over in such scenarios, \( \text{ir}_\text{axioms} \) provides a caching mechanism using the \texttt{diskcache} library\(^3\) to store preferences in an SQLite database. The second example in Listing 2 caches ArgUC preferences—quite a costly axiom due to calls of the external TARGET API [9]. But also other axioms benefit from caching. For example, in our experiments, enabling caching speeds up the computation of STMC1 preferences for the top-20 results of runs from the TREC 2020 Deep Learning track by 52% for the second run file.

---

\(^3\)https://pypi.org/project/diskcache/
We demonstrate the functionality of `ir_axioms` with the `@dataclass` decorator. A new axiom can also be parameterized—similar to, for instance, `ir_axioms` is a straightforward task: the abstract base class `Axiom` can be extended as shown in Listing 3. Assigning a name (optional) registers the new axiom in the `ir_axioms` registry and overriding the `preference()` method ensures that the new axiom’s preference values `pref_DfV(q, d, d)` are returned. Python’s built-in `@dataclass` method can be implemented to support caching. A new axiom can also be parameterized—similar to, for instance, `STMC1` that allows to choose the similarity function or the argumentative axioms that allow to choose the pre-trained TARGER model. For newly defined axioms, customizability can be achieved by adding all parameters as class properties and decorating the class with the `@dataclass` decorator. This automatically adds a constructor with the required parameters and implements the `__repr__()` method for caching preference values. All axioms in `ir_axioms` are implemented as data classes.

### 3.4 Defining New Axioms

Adding new axioms to `ir_axioms` is a straightforward task: the abstract base class `Axiom` can be extended as shown in Listing 3. Assigning a name (optional) registers the new axiom in the `ir_axioms` registry and overriding the `preference()` method ensures that the new axiom’s preference values `pref_DfV(q, d, d)` are returned. Python’s built-in `@dataclass` method can be implemented to support caching.

A new axiom can also be parameterized—similar to, for instance, `STMC1` that allows to choose the similarity function or the argumentative axioms that allow to choose the pre-trained TARGER model. For newly defined axioms, customizability can be achieved by adding all parameters as class properties and decorating the class with the `@dataclass` decorator. This automatically adds a constructor with the required parameters and implements the `__repr__()` method for caching preference values. All axioms in `ir_axioms` are implemented as data classes.

### 4 EXPERIMENTS AND USE CASES

We demonstrate the functionality of `ir_axioms` on three use cases: (1) post-hoc axiomatic analyses of rankings and judgments from shared retrieval tasks, (2) example-based analyses of rankings with respect to axiom violations, and (3) axiomatic result re-ranking in several variants. For our experiments, we use the setup of the TREC 2019 and 2020 Deep Learning tracks [10, 11]. All (in-)equalities in the axioms’ pre- and filter conditions are parameterized with a 10% margin so that equality conditions are “softened” to allow for some slight differences and inequality conditions are “strengthened” to require differences of at least 10% to express a preference.

#### 4.1 Post-Hoc Axiomatic Analyses

Post-hoc axiomatic analyses of a shared retrieval task could ask questions like how consistent the relevance judgments and the submitted rankings are with axiom preferences. Such analyses are supported by a range of utility functions in `ir_axioms`. The `AxiomaticExperiment` class provides the entry-point to a post-hoc analysis. Listing 4 shows how an `AxiomaticExperiment` is instantiated by passing as parameters the rankings (from run files), the topics, the relevance judgments, an index location, and the axioms to use; similar to how an `Experiment` is instantiated in PyTerrier [30]. The preferences member of an `AxiomaticExperiment` then provides access to all axiom preferences in a `pandas.DataFrame`.

#### 4.1.1 Axiom Preference Distributions

To calculate the distribution of axiom preferences for all rankings that were submitted in the Deep Learning tracks, we use the `preference_distribution` method from `AxiomaticExperiment` that returns the distribution of all preferences in a `pandas.DataFrame`. Analyzing such distributions provides an overview of which axioms might be interesting in the context of a shared task. Table 6 shows the absolute numbers of pairs in the top-10 results of the submitted runs for which an axiom had no preference or matched / did not match the preference in the ranking (i.e., ORIG). Interestingly, most axioms express relatively few preferences (e.g., TFC1 only for 4–13% of the pairs; reason: precondition ensures that the axiom is applied only to documents of about the same length). Still, when an axiom expresses a preference, this preference tends to agree with a ranking more often than not (exception: DIV). In the extreme cases of TFC3 and M-TDC with hardly any preferences, their restrictive pre- and filter conditions are the cause (e.g., hardly any query has terms of approximately equal inverse document frequency). On the other end of the spectrum are the axioms REG, DIV, STMC1, PROX1, and PROX2 that all rather more often express a preference than not.

#### 4.1.2 Consistency of Rankings or Judgments with Axioms

Contradictions between axiom preferences and rankings or relevance judgments can be identified by calling the `preference_consistency` function of the `AxiomaticExperiment` class. Table 7 shows the consistency of some selected axioms’ positive preferences with the preferences of the nDCG@10-wise best Deep Learning track runs from different categories (columns ‘NNLM’ (neural networks with language models), ‘NN’ (neural networks), and ‘Trad’ (traditional); categories from the Deep Learning track organizers) and with the relevance judgments of a task (column ‘Qrel’). From the axiom groups with the same objectives, we selected the axioms with the most non-zero preferences in Table 6. In all tasks of the Deep Learning tracks, the best language model-based neural runs (NNLM) were more effective than the best neural runs (NN) and the best neural runs were more effective than the best traditional runs (Trad).

The positive axiom preferences agree with the relevance judgments in 66–81% of the cases on all tasks, while the agreement with the best runs usually is lower. This hints at some potential for possible ranking improvements by deeper axiomatic analyses of a run’s decisions (cf. Section 4.2). For instance, the REG, DIV, STMC1, PROX1, and PROX2 axioms have substantially higher agreement with the judgments than with the best runs and they are the axioms that most often express preferences (cf. Table 6). It thus seems that further diversifying a run’s top results and matching the query...
terms or some semantically similar terms in result documents that contain the query terms closer to each other might be good ideas to further improve the best runs’ top ranks.

Among the individual axioms, AND has the highest agreement rates with the Deep Learning track’s best runs and AND, PROX1, and Hauff [8], and Völske et al. [42] in their studies on axiomatically improving, diagnosing, or explaining rankings. Nowadays, retrieval pipelines often re-rank the top-\(k\) results of a baseline model (e.g., BM25) by using learning-to-rank or neural methods. Also axiom combinations have been applied as re-rankers to improve the retrieval effectiveness [20].

### 4.3 Axiomatic Result Re-Ranking

Nowadays, retrieval pipelines often re-rank the top-\(k\) results of a baseline model (e.g., BM25) by using learning-to-rank or neural methods. Also axiom combinations have been applied as re-rankers to improve the retrieval effectiveness [20].

#### 4.3.1 KwikSort to Aggregate Rankings from Axiom Preferences

In their axiomatic re-ranking experiments, Hagen et al. [20] used the
A highly relevant document is returned at rank 5, lower than a less relevant document at rank 3. This pair, for instance, violates TFC1, REG, and STMC1 but is consistent with PROX1, PROX2, and ArgUC; fastText embeddings used for REG and STMC1.

Table 8: A pair from a ranking of the most effective run that participated in the TREC 2019 Deep Learning passage retrieval task.

The best runs from the respective tasks are: Passage'19: idst_bert_p1 (NNLM), TUW19-p3-f (NN), srchvrs_ps_run3 (Trad); Document'19: idst_bert_v3 (NNLM), TUW19-d3-re (NN), srchvrs_run1 (Trad); Passage'20: pash_r3 (NNLM), TUW-TK-Sparse (NN), bl_bcai_md11_vt (Trad); Document'20: d_d2q_duo (NNLM), ndrm3-orc-full (NN), bl_bcai_multfld (Trad).

Table 7: Consistency (in percent) of some selected axioms’ positive preferences with the preferences in the top-10 results of the best submitted run based on neural networks (NN) or on neural networks and large language models (NNLM), the best traditional run (Trad), and the respective non-negative relevance judgments from the TREC 2019 and 2020 Deep Learning tracks (Qrel); fastText embeddings used for REG and STMC1, each top-10 ranking contributes 45 preference pairs. A non-negative judgment preference counts as a match for a positive axiom preference (i.e., axiom preferences within a relevance level are acceptable).

The best runs from the respective tasks are: Passage’19: idst_bert_p1 (NNLM), TUW19-p3-f (NN), srchvrs_ps_run3 (Trad); Document’19: idst_bert_v3 (NNLM), TUW19-d3-re (NN), srchvrs_run1 (Trad); Passage’20: pash_r3 (NNLM), TUW-TK-Sparse (NN), bl_bcai_md11_vt (Trad); Document’20: d_d2q_duo (NNLM), ndrm3-orc-full (NN), bl_bcai_multfld (Trad).

KwikSort method [1] from the field of computational social choice to derive a final ranking from axiom preferences. In ir_axioms, KwikSort is implemented as a PyTerrier transformer operation. This way, any KwikSort re-ranking can be directly evaluated on test collections using the PyTerrier Experiment class [30].

Note, however, that a KwikSort-aggregated ranking can contradict individual axiom preferences (e.g., when combining several axioms’ preferences, they might differ on some document pair).

The axiom combination and weighting schemes implemented in ir_axioms (cf. Section 3.2) can help to avoid some such situations. For example, in Listing 5, three axioms are conjunctively combined to re-rank the top-20 results of BM25 with the ORIG axiom as a fallback when the three axioms do not agree. However, some circular aggregation issues are still possible. For instance, in the example of Listing 5, the three axioms could favor \( d_i \) over \( d_j \) and \( d_j \) over \( d_k \) but may not agree on a preference for \( d_k \) and \( d_i \) for which the fallback ORIG might then favor \( d_k \) over \( d_i \). In such cases, a KwikSort aggregation will still contradict at least one of the preferences.

4.3.2 Axiom Preferences as Features for Learning to Rank. Besides directly aggregating rankings from (weighted) pairwise axiom preferences via KwikSort, the preferences could also be interesting features for arbitrary learning-to-rank approaches like LambdaMART [44]. An axiom \( A \)'s preferences for the top-\( k \) results of some baseline ranker form a \( k \times k \) preference matrix \( P_A = [p_{ij}] \in \mathbb{R}^{k \times k} \) with \( p_{ij} = pref_A(q, d_i, d_j) \). To create a characteristic preference feature for each document \( d_i \) in a to-be-re-ranked top-\( k \) result list, ir_axioms allows to combine the preferences for \( d_i \) (i.e., the entries from the \( i \)-th matrix row) using simple functions like the

### Listing 5: Re-ranking BM25 based on axiomatic preferences.

```python
bm25 = BatchRetrieve(index, "BM25")

# If ArgUC, QTArg, and QTPArg don't agree, use ORIG.
axiom = (ArgUC() & QTArg() & QTPArg()) | ORIG()

# Re-rank the BM25 top-20 using KwikSort.
kwiksort = bm25 % 20 >> \\
    KwikSortReranker(axiom, index)

pipeline = kwiksort ^ bm25
```
with better relevance judgments should be preferred. Re-ranking median or the arithmetic mean:

$\text{f}_{\text{mean}}(d_i) = \frac{1}{k} \sum_{j=1}^{k} p_{ij}$.

In a feature vector, aggregated preferences from multiple axioms and/or multiple aggregation functions can be combined.

In \text{ir\_axioms}, the aggregation of preferences to learning-to-rank features is implemented as a PyTerrier transformer operator. Listing 6 shows an example in which a LambdaMART re-ranker is trained on mean- and median-aggregated axiom preferences as features to re-rank the top-10 results of BM25.

### 4.3.3 Re-Ranking by Estimating the ORACLE Axiom

The ORACLE axiom represents the ranking preferences implicitly stated in the human relevance judgments of a shared retrieval task. Formally, if \( \text{rel}(q,d_i) > \text{rel}(q,d_j) \) then \( d_i >_{\text{ORACLE}} d_j \); informally, documents with better relevance judgments should be preferred. Re-ranking a baseline’s top-\( k \) results using KwikSort on the ORACLE preferences would yield a perfect ranking of the \( k \) documents (but not necessarily an overall perfect ranking since the top-\( k \) might miss some highly relevant documents). However, in practical scenarios without relevance judgments, ORACLE preferences are not available. Still, the ORACLE preferences from shared tasks and test collections could be used to train an estimator for unseen document pairs by combining the preferences of other axioms [20].

In \text{ir\_axioms}, a special class \text{EstimatorAxisom} is implemented that can be used to predict an arbitrary target axiom’s preferences via a classification or regression method from scikit-learn [35] with the preferences of a pre-defined set of other axioms as features. During training, the target axiom preferences need to be available (e.g., judgments for some test collections in case of the ORACLE axiom) but in the later estimation phase for unseen pairs, only the preferences of the other axioms are needed as input to the estimator. On the estimated preferences, KwikSort can be run to generate a final ranking. Listing 7 shows an example using the \text{EstimatorKwikSortReranker} that combines ORACLE estimation and KwikSort re-ranking in a single PyTerrier module.

### 4.3.4 Axiomatic Re-Ranking Experiments on MS MARCO

To evaluate axiomatic re-ranking with \text{ir\_axioms}, we conduct experiments on the MS MARCO passage retrieval dataset [33] using the relevance judgments from the TREC 2019 Deep Learning track [11] for training and the ones from the TREC 2020 Deep Learning track [10] for testing. In the experiments, we re-rank the top-20 BM25 results using three different strategies: (1) KwikSort with all axioms, (2) KwikSort on an estimated ORACLE axiom, and (3) LambdaMART with preferences from all axioms as features.

For the first KwikSort re-ranker, all 25 axioms (cf. Table 2) are combined in an absolute majority voting scheme with a fallback option to the ORIG axiom (like in the second example in Table 5). In the second KwikSort re-ranker, the ORACLE axiom’s preferences are estimated by a random forest classifier that uses all 25 axioms as features (training on the TREC 2019 Deep Learning track for 100 iterations, tree depth set to 3).

For the LambdaMART re-ranker, we aggregate the preferences of each axiom \( A \) for each document using three aggregation functions:

\[
\begin{align*}
  f_{\text{hk}}(d_i) &= \left\lfloor \frac{\{d_j : d_i \geq_A d_j, \text{ } j = 1, \ldots, k\}}{k} \right\rfloor, \\
  f_{\text{lo}}(d_i) &= \left\lfloor \frac{\{d_j : d_i \leq_A d_j, \text{ } j = 1, \ldots, k\}}{k} \right\rfloor, \\
  f_{\text{eq}}(d_i) &= \left\lfloor \frac{\{d_j : d_i =_A d_j, \text{ } j = 1, \ldots, k\}}{k} \right\rfloor.
\end{align*}
\]

These features represent the number of documents above or below which \( d_i \) could be ranked according to \( A \), and how often \( d_i \) does not express a preference. The LambdaMART re-ranker is trained on the topics of the TREC 2019 Deep Learning track for 1000 iterations (no tuning of the LambdaMART parameters, though).

Table 9 shows the retrieval effectiveness of the baseline and the re-rankers as measured by normalized discounted cumulative gain (nDCG) [21] that was also used at the TREC 2020 Deep Learning track. Since KwikSort and LambdaMART are not deterministic, we let the re-rankers each create 10 rankings per query and apply 5-fold cross-validation to obtain average evaluation results.

Our basic experiments are meant to demonstrate the functionality of \text{ir\_axioms} (i.e., no parameter tuning, etc.). Still, the results of the KwikSort-RF and LambdaMART re-rankers show the potential of axiom preferences for retrieval effectiveness improvements—none of the differences are statistically significant, though. Using aggregated axiom preferences as learning-to-rank features and combining them with other common features could be an interesting

Listing 6: Re-ranking BM25 with LambdaMART using mean- and median-aggregated axiom preferences as features.

```python
bm25 = BatchRetrieve(index, "BM25")

# Aggregate ArgUC, QTArg, etc. by mean/median.
axioms = [ArgUC(), QTArg(), ...]
aggs = [mean, median]

# Compute features for top-10 results.
features = bm25 % 10 >> \
  AggregatedAxiomaticPreferences(axioms, index, aggs)

# Train LambdaMART re-ranker.
mlart = LGBMRanker(objective="lambdarank")
ltr = features >> apply_learned_model(mlart, "ltr")
ltr.fit(train_topics, train_qrels, \
  dev_topics, dev_qrels)
pipeline = ltr ^ bm25
```

Listing 7: Re-ranking BM25 using KwikSort aggregation on a trained estimator for the ORACLE preferences.

```python
bm25 = BatchRetrieve(index, "BM25")

# Estimate ORACLE based on ArgUC, QTArg, etc.
axioms = [ArgUC(), QTArg(), ...]
rf = RandomForestClassifier()

# Train Random Forest classifier on top-20 results.
kwiksort_rf = bm25 % 20 >> \
  EstimatorKwikSortReranker(axioms, rf, index)
kwiksort_rf.fit(train_topics, train_qrels)
pipeline = kwiksort_rf ^ bm25
```
direction for further axiomatic experiments. Additionally, formulating new axioms that capture different angles of relevance also seems to be a very promising direction. With ir_axioms at their fingertips, researchers working on any of these topics can now simply focus on expressing their respective axiomatic ideas and then rely on the framework for conducting their experiments.

5 CONCLUSIONS AND FUTURE WORK

With the open-source framework ir_axioms, we provide tools to experiment with retrieval axioms—basic constraints that are meant to characterize good ranking functions. Implementations of 25 commonly used and well-understood retrieval axioms provide a good starting point to experiment with axiomatic re-ranking and evaluation. Since ir_axioms is tightly integrated with PyTerrier, it can be combined with a wide range of retrieval models, test collections, and evaluation functions. This makes it easy to incorporate axiomatic approaches into state-of-the-art retrieval pipelines.

Our use cases show the potential of conducting axiomatic retrieval experiments with ir_axioms in a declarative, easy-to-understand way. Further use cases could be to actually reproduce some of the results of recent axiomatic studies like the re-ranking experiments of Hagen et al. [20] and Bondarenko et al. [6], the meta-learning experiments of Arora and Yates [2], the regularization experiments of Rosset et al. [40], or the diagnosis and explanation experiments of Rennings et al. [38], Câmara and Hauff [8], Formal et al. [16], Vůlka et al. [42], and MacAvaney et al. [27]. However, so far, not all of these experiments are supported in ir_axioms (exception: re-ranking). Still, other types of experiments can further be added to ir_axioms, and we gratefully accept contributions.\(^4\)

Previous studies and also our experiments indicate that the current axioms are somewhat limited in their expressiveness. Axiomatically capturing further angles of relevance is an interesting direction for future work. Similar to new experiments, also new axioms can be directly added to ir_axioms. On the technical side, we plan to add parallelization to speed up axiomatic analyses (e.g., for computing axiom preference matrices), and we plan to simplify axiomatic consistency checks at various points in multi-stage retrieval pipelines.

\(^4\)GitHub: https://github.com/websis-de/ir_axioms/
Python package: https://pypi.org/project/ir_axioms/

ACKNOWLEDGMENTS

This work has been partially supported by the DFG through the project “ACQua 2.0: Answering Comparative Questions with Arguments” (grant HA 5851/2-2) as part of the priority program “RATION: Robust Argumentation Machines” (SPP 1999).

REFERENCES


Table 9: Retrieval effectiveness on the TREC 2020 Deep Learning track (passage retrieval task) when re-ranking the top-20 passages of BM25 with the three axiomatic re-ranking strategies: (1) KwikSort with majority voting (KwikSort-MV), (2) KwikSort with a random forest estimator for the ORACLE axiom (KwikSort-RF), and (3) LambdaMART with axiom preference features.

<table>
<thead>
<tr>
<th>(Re-)Ranker</th>
<th>nDCG@5</th>
<th>nDCG@10</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM25 (init. rank.)</td>
<td>0.497</td>
<td>0.494</td>
</tr>
<tr>
<td>KwikSort-MV</td>
<td>0.496</td>
<td>0.492</td>
</tr>
<tr>
<td>KwikSort-RF</td>
<td>0.516</td>
<td>0.498</td>
</tr>
<tr>
<td>LambdaMART</td>
<td>0.517</td>
<td>0.498</td>
</tr>
</tbody>
</table>