Elrond-Group Touché Task1: Argument Retrieval for Controversial Questions

Vincenzo Savarino[a], Mattia Luciani[a], Giuseppe Esposito[a], Ludovico Bonan[a], Enrico Rossignolo[a] and Gianmarco Carraretto[a]

[a]*University of Padua, Italy*

Abstract

We want to solve the problem about retrieving relevant and good-quality arguments related to controversial topics. We provide 4 different approaches, from a naive Lucene *StandardTokenizer* that split words at punctuation with a stop-word list to a more complex analyzer that use natural language processing. All of them have comparable results.

Keywords

IR, CLEF, TREC

1. Introduction

Information retrieval (IR) is concerned with the structure, analysis, organization, storage, searching, and dissemination of information. An IR system is designed to make a given stored collection of information items available to a user population. At one time that information consisted of stored bibliographic items, such as online catalogs of books in a library or abstracts of scientific articles.

The widespread use of social networks, forums and blogs has provided fertile ground for discussion between numerous people on a multitude of different topics. This has generated a large amount of arguments for or against different topics, leading to the need to develop a tool that allows them to be found. This is why we decided to participate in the Touché[1]: Argument Retrieval at CLEF whose aim is to support users who search for arguments to be used in conversations (e.g. getting an overview of pros and cons). In particular, we focused on task 1, which consists of given a query on a controversial topic, retrieve relevant arguments from a focused crawl of online debate portals.

"Search Engines", course at the master degree in "Computer Engineering", Department of Information Engineering, University of Padua, Italy. Academic Year 2020/21

vincenzo.savarino@studenti.unipd.it (V. Savarino); mattia.luciani.2@studenti.unipd.it (M. Luciani); giuseppe.esposito.3@studenti.unipd.it (G. Esposito); ludovico.bonan@studenti.unipd.it (L. Bonan); enrico.rossignolo@studenti.unipd.it (E. Rossignolo); gianmarco.carraretto@studenti.unipd.it (G. Carraretto)

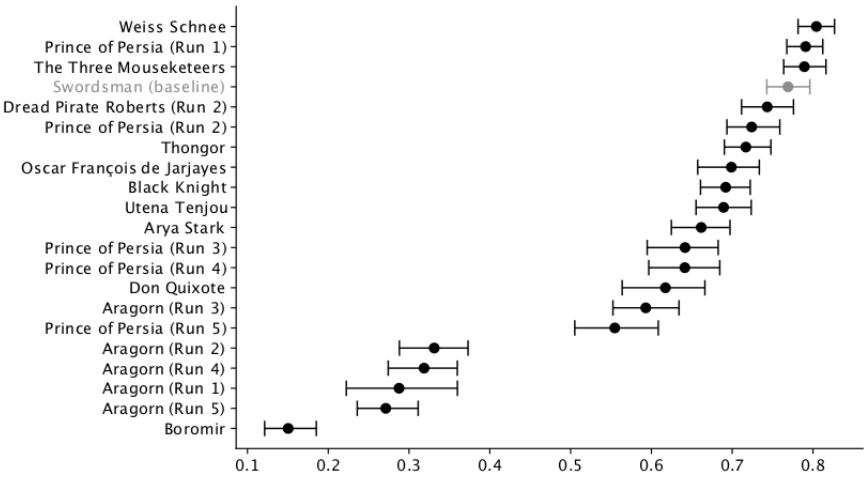

Figure 1: Mean *Normalized Discounted Cumulative Gain at cutoff of 5 (nDCG@5)* with 95% confidence intervals, 2020 edition[1]

To achieves this goal we have developed an information retrieval system using the Apache Lucene[2], a Java library providing powerful indexing and search features, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities.

The paper is organized as follows: Section 2 introduces related works; Section 3 describes our approach; Section 4 discuss some discarded ideas; Section 5 says something about the implementation; Section 6 explains our experimental setup; Section 7 discusses our main findings; finally, Section 8 draws some conclusions and outlooks for future work.

2. Related Work

This is the second year in which *Touché Task 1: Argument Retrieval for Controversial Questions* is proposed by CLEF. Last year top 3 winner teams were: Weiss Schnee, Prince of Persia and The Three Mousketeers (see figure 1).

Weiss Schnee team used DPH parameters-free weighting model with *Query Expansion* using embeddings words. After it, they re-ranked the result based on quality.

Prince of Persia used multiple models but always with augmentation through synonyms (WordNet[3]) and re-ranking using sentiment analysis based on neutral sentiment.

The third one used simply *DirichletLM* similarity and they still obtained very good results.

Thus we know that model based on *Term Frequency (TF)* and *Inverse Document Frequency (IDF)* are a good starting point but statistical models are better and argument quality matters.

3. Methodology

We approached the problem a step at time:

1. We started with reading and understanding the content of the dataset an how it could be parsed. We decided to select two field for every document (as body and title) and discard the rest.
2. We wrote a parser with the ability of filtering out duplicated documents.
3. We continued making experiments and edits to the index and search algorithms doing at the same time some evaluation with the *trec_eval*[4] utility. In particular in this step we focused on maximizing some descriptive measure like the *Mean Average Precision (MAP)*.
4. At the end we evaluated all of them and selected the best and most interesting ones.

4. Discarded ideas

Before describing our presented runs, we want to talk about those ideas that we thought were good, but failed to provide good result in practice.

4.1. Synonyms Expansion

We implemented *Query Expansion* using the WordNet[3] database. We thought that adding related words as synonyms could improve the score. Unfortunately the result showed the opposite and this approach was a lot worse than the other, even the simplest ones. The reason might be that this expansion added words that were too different from the original ones[1], leading to unrelated documents. A little variation of this approach was to extract categories related to words from the WordNet database, but not even this gave better result. So, even if the relative filters are still in our code, we don't use them anymore.

4.2. Word-Ngrams

Another approach that we tried was to search for a sequence of consecutive words and using this for refining the documents score. The idea was that, if a document contains the same words in the same order then is should be more relevant. However, even if this approach didn't produce result as bad as the synonyms expansion, it was still worse than without.

5. Implementation

As base for our implementation we developed a parser (`Task1Parser.java`) with the ability to build an object representation of a document. To do this we used the Jackson Json Library[5] to obtain a java tree-like representation and extracting different fields:

[1] For example 'cat' retrived as synonyms words like *hombre, guy, barf, caterpillar, ...*

- *premises* becomes the **body** of our document as it held the content of the post[2]
- *discussionTitle* becomes the **title** of our document, as it could be used for improve the search
- we also included *sourceDomain* for experimenting as it's not used for searching

With a similar approach we wrote a parser for the topics[3]: in this case they are an XML document with an identification number and a title[4] for each query. This (short) parser are defined inside classes `Topic.java` and `Topics.java` were the latter is the one that contains the parsing method.

Another base class we used is `DirectoryIndexer.java` that provide a method for indexing a collection of documents using Lucene[2] as indexing library. It initialize a new index and populate it with the parsed documents we provide. This class was inherited from an example provided to us by our professor Nicola Ferro [5]. However, we also wrote a variant of this class called `DirectoryIndexerMT.java` that do the indexing using multiple thread as some run required a long time on a single one.

Every run we present is composed by two class: an Analyzer and a Searcher. The first serves the purpose of converting both the document's and the query's text in a stream of tokens eventually applying a series of filters or transformations. The second provides a method that combine one or more query implementation and use them for doing the actual search. On top of that it's necessary to provide a Similarity class that defines the mathematical function used for calculating the score. Generally speaking, for making the search compliant with the Y information retrieval scheme it's required to provide the same similarity and the same analyzer[6].

In the following subsections we present the description of our runs.

5.1. Simple approach

We started with a simple approach: we used the *StandardTokenizer* from Lucene and, after converting to lowercase, a stop-list named *99webtools.txt*[6]. The standard tokenizer splits documents at punctuation and space and the stop-list remove the most used English words that are also the most frequent on our dataset. Finally we build a boolean query with *SHOULD* clause and use the *BM25* similarity to compute the scores.

5.2. KRun

The `KAnalyzer.java` used for this run provide the following operation:

[2]Documents are post from online discussions

[3]The topics contain the information we have to search

[4]The 2020-version of the topics included also a description and a narrative fields that are absent inside the 2021 one

[5]http://www.dei.unipd.it/~ferro/

[6]In our work there is an exception to this rule when we provide different analyzers during the search phase but only for filtering portion of the stream that we have merged during the indexing

1. Separate different words using Lucene's `StandardTokenizer`
2. Transform every token to lowercase
3. Remove stop words from *99webtools*[6]
4. Apply the `KStemFilter`[7]

The searcher simply search for matching words inside the document body and title using the provided analyzer. The similarity that gives best result with this approach is the LM Dirichlet Similarity.

5.3. TaskBody

The `TaskAnalyzer.java` used for this run provide the following operation:

1. Separate different words using Lucene's `StandardTokenizer`
2. Transform every token to lower case
3. Remove english possessives for making the stream more uniform
4. Apply the `EnglishMinimalStemFilter` for making the stream more uniform
5. Remove stop words from *99webtools*[6]
6. Apply the Lovins Stemmer[8]

The searcher simply search for matching words inside the document body using the provided analyzer. Although it seems strange, the use of two stemmers is justified by the fact that the first stem only apply minimal stemmer rules and it's used to simplify the token and increasing the possibility for the stop word to be removed. The similarity that gives best result with this approach is the DFI Similarity with `IndependenceStandardize` as independent measure.

5.4. OpenNlp

This run use a custom analyzer (`OpenNlpAnalyzer.java`) based on Apache OpenNLP[9]. It provide the following operations:

1. Create the token stream using the `OpenNLPTokenizer`
2. Apply the `OpenNLPPOSFilter` that attach part-of-the-speech types to the tokens
3. Remove token marked with a custom list of types that we found non very informative for this task
4. Breaks the hyphens in multiple tokens[7]
5. Transform every token to lower case
6. Replace some words, specifically *'s*, *'re* and *'m*, with the expanded form
7. Remove stop words from *99webtools*[6]
8. Apply the Porter Stemmer[10]
9. Add a synonym for every token composed with the concatenation of the type associated with the token, between angle brackets, with the original token

[7] Although this is normally not recommended we found it give better result

10. Optionally filter the stream to only return the original or the type-concatenated tokens

The filtering on the last point is not applied during the indexing. During the search two different analyzer are used, one that keep only the original tokens and the other that keep only the typed ones. The searcher `OpenNlpTaskSearcher.java` generate a query in this way:

1. Generate a query called `normalQuery` as the combination of
   a) a query constructed with the original token that search on the body of the document
   b) a query constructed with the original token that search on the title of the document
2. Generate a query called `typedQuery` as the combination of
   a) a query constructed with the typed token that search on the body of the document
   b) a query constructed with the typed token that search on the title of the document
3. The final query is the combination of `normalQuery` and `typedQuery`

This searcher try to exploit the types detected by OpenNLP to improve search. It's also worth to notice that this use a tokenizer algorithm different from other approach and removes stopwords also by their part inside a phrase and not also from a predefined word list. The similarity that give best result with this approach is the LM Dirichlet Similarity. Given the big amount of computation (with respect to other tested analyzers) both the indexing and the searching are done with a multi-threaded approach.

6. Experimental Setup

6.1. Collection

The collection of documents contains 5 *JSON* files without break-lines with different fields.

The document collections used is args.me corpus [11] which includes 387 740 arguments. They are crawled from the debate portals Debatewise (14 353 arguments), IDebate.org (13 522 arguments), Debatepedia (21 197 arguments), and Debate.org (338 620 arguments). Moreover, the corpus contains 48 arguments from Canadian Parliament discussions. The arguments are extracted using heuristics that are designed for each debate portal. While working with the collection, we realized that there were a significant number of duplicate documents, making it impossible to use the trec_eval [4] tool during the performance measurement phase of our system. Therefore, we skipped them by using a Hash Table.

6.2. Topics

Topics are provided by CLEF as one XML file and have only one field (title). This file contains exactly 50 queries.

Example of topic 2021:

```
<topics>
    ...
    <topic>

        <number>1</number>

        <title>Should nuclear weapons be abolished?</title>

    </topic>
    ...
</topics>
```

For the evaluations we have used the topics of 2020 as it is supplied also the *qrels* file which allows to carry out the evaluations of the performances. Although this topics also have additional fields we ignored them as we needed to estimate the performance with the newer title-only topics.

6.3. Evaluation measures

We will use the following measures:

- nDCG@5: we used it to compare our results to the Touché results of last year.
- *Interpolated precision-recall curve*: it indicates the trade-off between precision and recall
- MAP: it's the mean of average precision over all the topics
- the number of *retrieved relevant documents*
- *Time*: time taken for the run

6.4. Git repository and its organization

Our repository is on BitBucket at the following link https://bitbucket.org/upd-dei-stud-prj/seupd2021-rs/.

The `Main` class of the application work as a starting point: it receive a string as a command-line argument and execute the relative approach (defined inside `PreparedRuns.java`). All the other class are organized in different sub-packages in this way:

- `analyzer`: contains analyzers and analyzing-related class
    - `analyzer.filters`: contains filters used within ours analyzers
- `index`: contains classes related to the indexing process
- `parse`: contains classes related to the parsing process

- `search`: contains searcher and search-related class
    - `search.queries`: contains helper classes for generating custom queries
- `topics`: contains classes for parsing and representing topics
- `utils`: contains extra utility classes

In addition the repository contains the following resources:

- `opennlp`: folder containing the binary models for Apache OpenNLP[9]
- `words.db`: folder containing WordNet[3] database files
- `99webtools.txt`: stop-word list from 99webtools[6]
- `example.properties`: example property file, for executing the program from source it's needed to create a copy of this and filling it appropriately
- `wn_s.pl`: WordNet[3] database in Prolog format

6.5. Hardware & Software

As hardware for the runs we have used a laptop with Intel i7-8750H (4.10GHz, 6 core/12 threads) processor, 16GB of ram and 500GB of Solid State Drive Samsung SSD 970 EVO Plus 500GB (2B2QEXM7).

The system was running ArchLinux with kernel 5.11.16-arch1-1 and used OpenJDK Runtime Environment AdoptOpenJDK-16.0.1+9 for running the application.

7. Results and Discussion

To evaluate the performance of our approaches, we used files (qrles and topics) from the previous year (2020) provided by CLEF[1].

After several tests, we decided to present 4 results.

The Simple approach was the first one we developed and we used it as a starting base for improving our system. It has strongly lower performance than the other three presented approaches: as it's easy to understand looking at the Table 1, despite it's fast execution time, it has the worst nDCG@5 and MAP values. The only good measure we obtain with that approach is the number of relevant retrieved documents that is, in fact, quite better than the others. Clearly, there are more relevant documents but they are badly ranked. The other 3 approaches result similar as it is easy to verify in the table even if OpenNlp employs more time.

Looking at the results of the other 3 runs, that are KRun, TaskBody and OpenNlp, we noticed that is difficult to evaluate which method is the better. In general the scores are very similar between each other but, depending on which score we choose, we obtain a different ranking. Among these three runs, the most disappointing one is OpenNlp because, in respect with the other two, it has similar value of nDCG@5 and MAP but it is very slow to run. However we

Method	nDCG@5	MAP	Time (s)	Retrived relevant docs
Simple approach	0.3337	0.1271	79,55	1165
KRun	0.5422	0.1654	79,27	963
TaskBody	0.5243	0.1698	90,03	997
OpenNlp	0.5334	0.1653	498	979

Table 1
Results of the runs

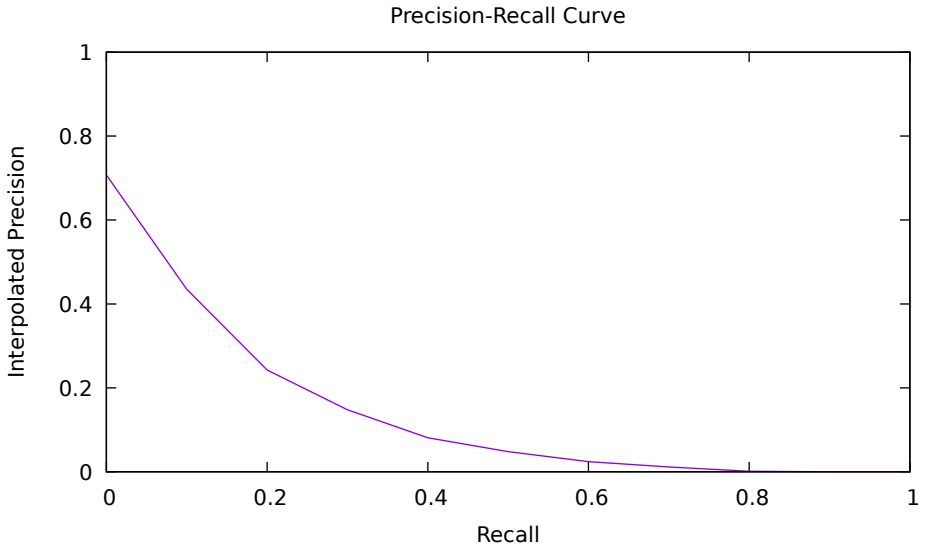

Figure 2: Precision-Recall curve of our Simplest Approach.

cannot be sure which approach will score better on the newer query, so we decided to include it anyway.

In addition to the measures previously discussed we also considered the Precision-Recall trade-off for all implemented systems.

This measure is used to observe the number of the false positive/false negative, which are the retrieved documents that are not relevant and the not retrieved documents that are relevant. In this Trade-off we can't have both precision and recall high, but if we increase precision, it will reduce recall, and vice versa. So an high value of Precision allows to reduce the number of false positive, while an high value of Recall allows to reduce the number of false negative. From these graphs we noticed that the Simple approach was the best one for higher values of recall with respect to the other ones maybe this is due from the fact that retrieved more relevant document.

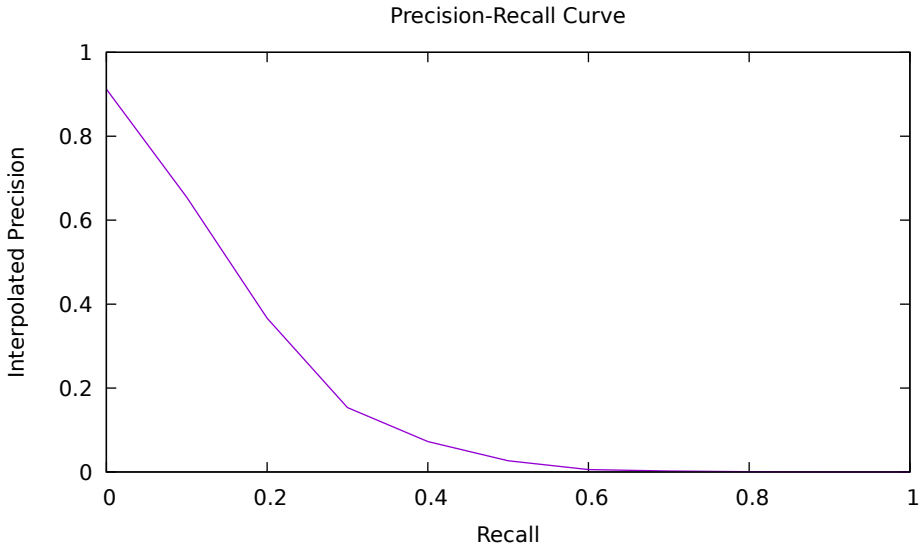

Figure 3: Precision-Recall curve of KRun.

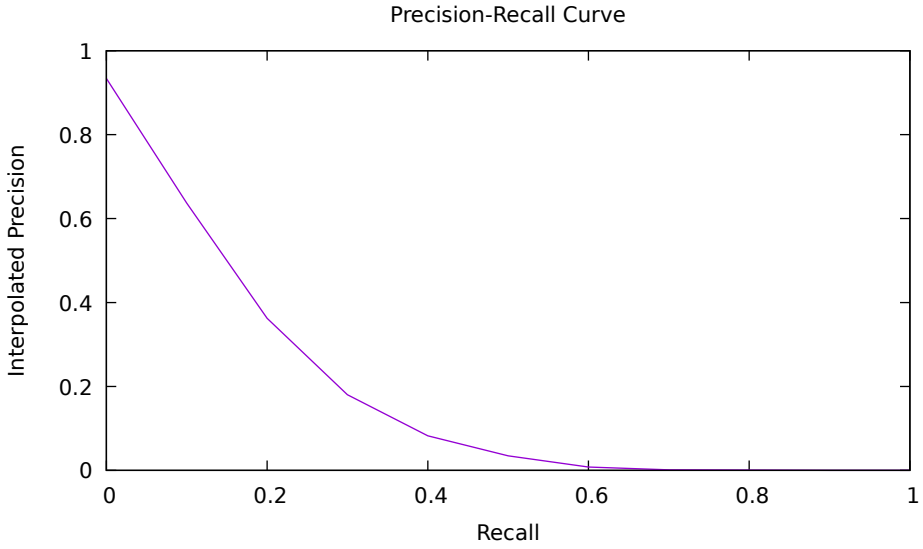

Figure 4: Precision-Recall curve of our TaskBody.

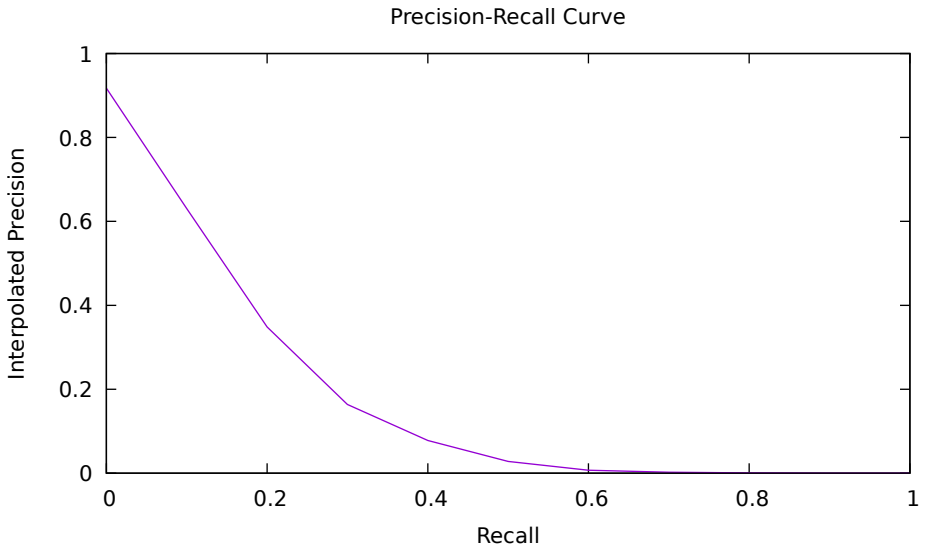

Figure 5: Precision-Recall curve of our OpenNlp.

8. Conclusions and Future Work

In conclusion, we have presented four different approaches for retrieving information, they all shown (relatively) good performance with the last year's queries and we are impatient to see how they will score with the newer ones.

A future improvement may consist on training a neural network on the qrels file. We though to use a vector representation of the analysed documents (may be provided by Word2Vec[12]) and combined with the topic's one to predict the score. In this way it is possible to re-rank the result obtained with our previous methods.

Another possibility is to train a neural network to understand the topic overall quality and re-rank based on the quality score maybe using the dataset provided by Gienapp et al[13]. In this case we need to store on the index a new field to keep the verbatim documents.

References

[1] Touché @ clef, 2021. URL: https://touche.webis.de.

[2] Apache Lucene Core, 2021. URL: https://lucene.apache.org/core/.

[3] P. University, About wordnet, 2010. URL: https://wordnet.princeton.edu/.

[4] Evaluation software used in the text retrieval conference, 2020. URL: https://github.com/usnistgov/trec_eval.

[5] Jackson project, 2021. URL: https://github.com/FasterXML/jackson.

[6] List of english stop words, 2013. URL: https://99webtools.com/blog/list-of-english-stop-words/.

[7] R. Krovetz, Viewing Morphology as an Inference Process, in: R. Korfhage, E. Rasmussen, P. Willett (Eds.), Proc. 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1993), ACM Press, New York, USA, 1993, pp. 191–202.

[8] J. B. Lovins, Development of a Stemming Algorithm, Mechanical Translation and Computational Linguistics 11 (1968) 22–31.

[9] Apache opennlp, 2021. URL: https://opennlp.apache.org/.

[10] M. F. Porter, An algorithm for suffix stripping, Program 14 (1980) 130–137.

[11] Y. Ajjour, H. Wachsmuth, J. Kiesel, M. Potthast, M. Hagen, B. Stein, args.me corpus, 2020. URL: https://doi.org/10.5281/zenodo.3734893. doi:10.5281/zenodo.3734893.

[12] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, 2013. arXiv:1301.3781.

[13] L. Gienapp, B. Stein, M. Hagen, M. Potthast, Webis Argument Quality Corpus 2020 (Webis- ArgQuality-20), 2020. URL: https://doi.org/10.5281/zenodo.3780049. doi:10.5281/zenodo.3780049.