Touché Task 1: Argument Retrieval for Controversial Questions. Resolution provided by Team Goemon.

Leonardo Carlassare[a], Matteo Carnelos[a], Laura Menotti[a], Thomas Porro[a] and Gianmarco Prando[a]

[a]*University of Padua, Italy*

Abstract

The *Information Retrieval (IR)* system built by Team Goemon provides a solution for the Touché Task 1: Argument Retrieval for Controversial Questions. The goal is to support users who search for arguments used in conversations. Given a question on a controversial topic, the task is to retrieve arguments from a focused crawl of online portals.

Keywords

Argument Retireval, Touché Task, Team Goemon

1. Introduction

This report provides a brief explanation of the IR system built by Team Goemon Ishikawa during the Search Engines course at the master degree in Computer Engineering of the University of Padua. The goal of the above system is to provide a solution for the Touché Task 1: Argument Retrieval for Controversial Questions.

Touché Task 1 aims at supporting users who search for arguments to be used in conversations (e.g., getting an overview of pros and cons or just looking for arguments in line with a user's stance). Given a question on a controversial topic, the task is to retrieve relevant arguments from a focused crawl of online debate portals[1]. More specifically, the corpus comprises 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.[2] The arguments for each debate portals are stored in a JSON file, therefore the corpus comprises 4 JSON files.

The required IR system will parse the JSON files, index the retrieved document and produce a run on 50 different provided topics. For each topic, the system will retrieve the 1000 most relevant documents. As a future work the team will assess whether the system is effective by means of provided relevance judgments and the trec_eval evaluation measures.

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

leonardo.carlassare@studenti.unipd.it (L. Carlassare); matteo.carnelos@studenti.unipd.it (M. Carnelos); laura.menotti@studenti.unipd.it (L. Menotti); thomas.porro@studenti.unipd.it (T. Porro); gianmarco.prando@studenti.unipd.it (G. Prando)

2. Related Work

All our tests are available in our git repository in the directory `src/test/java`. These results come from the trec_eval software and they are based on last year topics and relevance judgements. We based the efficiency of our system on two measures: as far as correctness is concerned we used the *Normalized Discounted Cumulated Gain (nDCG)* measure, more specifically nDCG@5 both on average and for each topic but we also checked the computational time each system required.

2.1. Our baseline

We started our project by building a basic IR system where we used as similarity function BM25 and we indexed the document using the stop list provided by Lucene and no stemming. This approach scored on 2020 topics an overall nDCG@5 of 0.4057.

2.2. Improved Tests

From the above result, we decided to change the similarity function from BM25 to Dirichlet, which improved the overall score. The result for this system are available at section 5. Among all our tests, we decide to analyze in detail the three most significant ones.

- DirichletLM Similarity, OpenNLP Tokenizer, Terrier Stop, No Stem
- DirichletLM Similarity, OpenNLP Tokenizer, Atire Stop, OpenNLP Lemmatizer
- DirichletLM Similarity, Lucene Tokenizer, Lucene Stop, No Stem, Query Expansion

In the first system we changed the stop list with an intermediate one that is bigger than the Lucene one. We also changed the tokenizer with the `OpenNLPTokenizer`, which is more complex and provides a deeper analysis of the documents. In this version, we do not use any stemmer. The overall nDCG@5 for 2020 topics is 0.658. We report a comparison from this system and the one we chose. The graph shows the nDCG@5 variation for each topic.

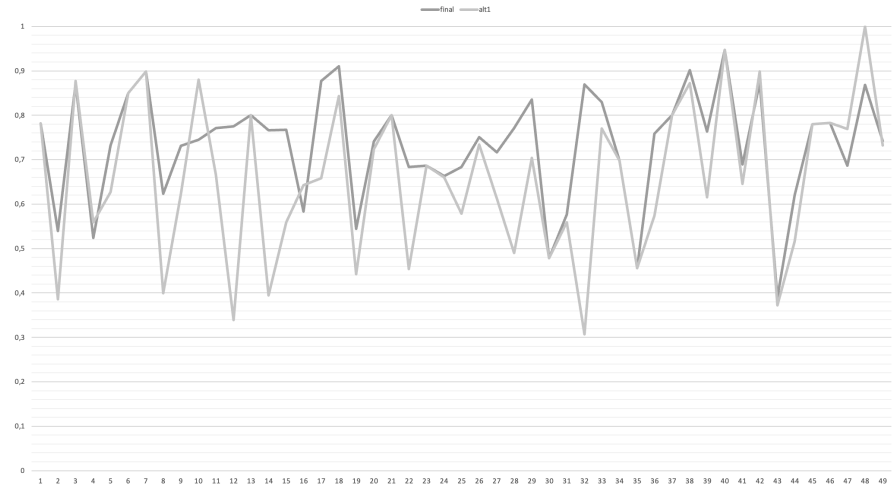

Figure 1: comparison from our system (final) and the test system (alt1).

From the graph 1 we can clearly see that we achieved an improvement from our starting baseline however our system scores better in most topics. Most importantly, this approach has a swinging trend, in fact some topics score the same as our system but some other ones score very poorly. We conclude that this system is unstable and its efficiency is based on which query it parses.

In the second system we changed the stop list more complete and sophisticated. We kept the same `OpenNLPTokenizer` as before and we added an `OpenNLP Lemmatizer` for the stemming. The overall nDCG@5 for 2020 topics is 0.614. We report a comparison from this system and the one we chose. The graph shows the nDCG@5 variation for each topic.

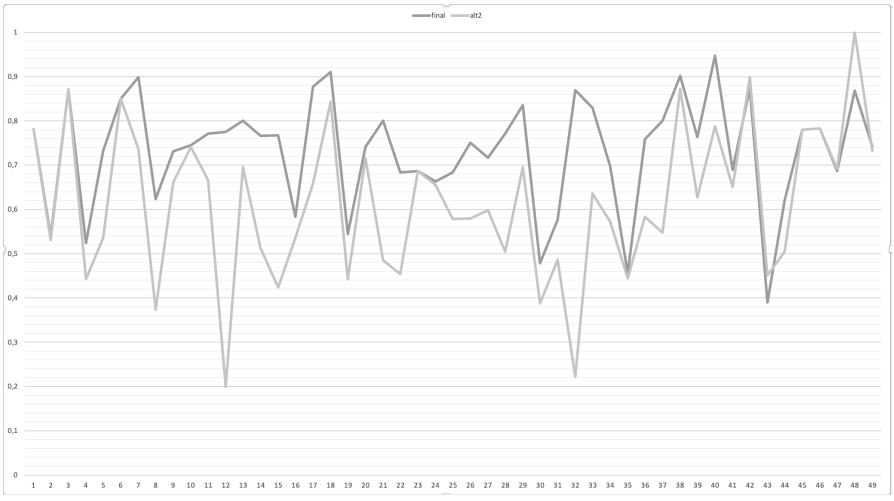

Figure 2: comparison from our system (final) and the test system (alt2).

From the graph in 2 we can clearly see that we achieved an improvement from our starting baseline however our system scores better in most topics. In addition, his system scores worse than the one tested above and like the first one this approach has a swinging trend. We conclude that this system is unstable and its efficiency is based on which query it parses. Another important thing to notice is that this approach requires a huge computational time with respect to the other tested systems. In fact, the other systems required at most five minutes to index the documents while this one required more than forty minutes. For these reasons, we discard this approach since it was computational heavy and it did not bring any improvements. However, we thought it was an interesting system to analyze in the report.

In the third system we kept everything like our system and we added the query expansion, which is described at section 3. The overall nDCG@5 for 2020 topics is 0.722. We report a comparison from this system and the one we chose. The graph shows the nDCG@5 variation for each topic.

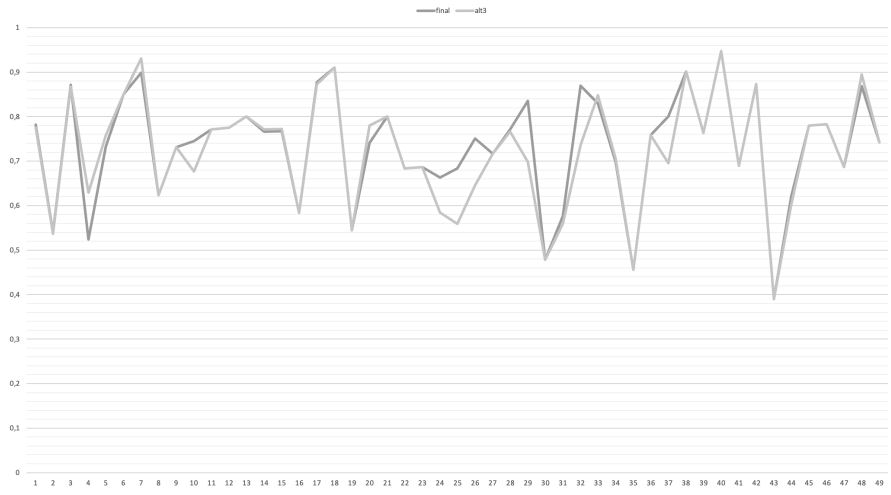

Figure 3: comparison from our system (final) and the test system (alt3).

From the graph in figure 3 we can see how the two systems are very similar. In fact, the query expansion did not improved much our system. We think this is because of the topics' titles syntax. Although we see a slight improvement, we decided to discard this approach in our final system because of the heaviness of the computation. Moreover, the use of query expansion required more indexing time and space therefore the costs outweigh the benefits.

3. Methodology

This section will address how our system was built and also how we organized the work. As a matter of facts we would like to briefly explain the workflow our team decided to adopt.

The main methodology we adopted comprises two main phases, which we confront in two different work modalities. Firstly, we built a basic IR system with some standard document

processing, indexing and searching in order to have a robust project structure that actually works. As far as the workflow is concerned, we decided to divide the workload in some areas and each member. contributed on a different task. Moreover, the main areas of work were the document parser, the indexing and the searcher. Subsequently, we focused on improving our system's efficiency with some changes in the analyzer and in the similarity function. In this phase we all worked together in several team meetings and we actually decided to have a dedicated repository maintainer for this part of the project in order to avoid conflicts between local versions.

3.1. Building the baseline

As stated before, in this phase we built a basic IR system in order to create a robust baseline for our project. The system is divided in different packages devoted to different tasks. The `parse` package handles the document parsing, `index` operates on the indexing of the documents while `search` takes care of topics parsing and the search for relevant documents. In the end, there is a main class called `Ryusei` (Ryusei is the name of Goemon's sword) that indexes Touché corpus and produces the run for provided topics.

The first thing we did was analysing the Touché corpus and observe the most useful fields to assess the relevance of the documents in order to build the `parse` package. As a result, we identified the following fields: conclusion, text and stance. The above-mentioned fields together with the document id are mapped into constants in the Java class `DebateFields`, which is used during the parsing of the JSON files that compose the Touché corpus. The core of the package is the Java class `DebateParser`, which extends the `DocumentParser` abstract class. The actual parsing of each document is done by means of the `parse` method which extracts the id and the relevant fields for the indexing and produces a `ParsedDocument` object, whose specifics are inside the homonym class. Since the Touché corpus is composed of JSON files, the parsing uses the Jackson library. All the classes concerned with the topic parsing are contained in this package as well. In particular, the `ToucheField` class maps the relevant field names contained in the topics' XML files into constants while `TopicParser` is an abstract class that defines an abstract topic parser. The real parsing of the topics is done by the `ToucheParser` class, which parse the XML file containing the topics and represents each topic as a `QualityQuery` object with a field that corresponds to the topic title. In fact, after some testing we saw that our system works best if we only parse the title without description and narrative.

Once the parsing was completed, we focused on producing the indexing of the entire corpus, whose implementation is inside the `index` package. Each field of the `ParsedDocument` object (apart from the id) is represented in its own class that extends the `Lucene` class textitField and specifies the index options for the field. All the fields are tokenized, not stored and the index keeps only document ids and term frequencies in order to minimize the space occupation. The Java class `DebateIndexer` indexes the debates documents processing a whole directory tree, since there are different files. The *index* method parse each document one file at a time that is inside the document directory path provided during the construction of the `DebateIndexer` object and for each `ParsedDocument` object, it creates a `Lucene Document` object with the above-mentioned fields plus the document id, which is stored as a `StringField`. Each indexed

document's id is put inside an `HashMap` in order to eliminate duplicates. This method has a boolean parameter `verbose` that, if set to true, allows the method to print the process status.

In the end we developed the `search` package which actually contains only one class, which is called `DebateSearcher`. In this class, after the topics have been parsed and represented as `QualityQuery` objects, queries are built and the most relevant document for each query are searched by means of the Lucene `search` method. After the search, the score of each document for each topic is stored in the run file following the trec convention.

3.2. Upgrading the system

We initially used an analyzer with off-the-shelf tokenizer and stop list provided by the *Lucene* library and the lowercase filter while we adopted the BM25 similarity. At this point the system was tested on last year topics and relevance judgments and we upgraded it after some different experimental testing on different stop lists, stemmer and similarity functions. All these tests were performed by means of a Java Testing class which are available in our git repository in the directory `src/test/java`. In particular, in the class `RyuseiTest` we tested several combinations of Lucene Tokenizer, OpenNLP Tokenizer, Lucene stop, Atire stop, Terrier stop, Smart stop, no stem and OpenNLP Lemmatizer. In the end, we also changed the similarity function from BM25 to DirichletLM.

We also used the query expansion method provided by `Lucene`. In order to use this method we implemented it through the predefined filters already defined in `Lucene`. The filters we use are `SynonymMapFilter` followed by `FlattenGraphFilter` as specified in the documentation. The arguments of the first methods are the `TokenStream` (the same that is used in the "classic filters"), the third one is the ignore case property that we have set as true since we do not differentiate the uppercase from the lowercase. The second argument is the most important since it defines the `SynonymMap` to use when we're building the index. In order to create it we have used the `SynonymMapBuilder` that allows us to create a list where a term is linked to all his synonyms. For the construction of the list of synonyms we have used the `WordnetDatabase` (available at the following link[3]). In particular, we used the file `wd_s.pl` which comprises the most popular synonyms of the English Language (the documentation of the format of the file is described here). Once we have parsed the file of `Wordnet` we just build the `SynonymMap` through its builder and the Object is passed to the `SynonymMapFilter` method.

The results of some of our tests are available in section 2. After analysing the results of the different tests we decided to change the similarity of our system from BM25 and Dirichlet because we saw a great improvement. In particular, nDCG@5 for all topics increased from 0.405 to 0.733. As far as the other tests are concerned, we decided to use the Lucene stop list and no stemming, because this combination was the best time-efficient and stable one. In fact, the other runs have greater results for some topics but at the same time some topics scored very poorly, therefore we would have had some very satisfied users and some disappointed ones. In conclusion, we have privileged more stable runs over more swinging ones that gave better results for some queries, so to have more probability to see similar results also for this year topics. The results on the last year topics for our current system are available in section 5.

4. Experimental Setup

This section describes the experimental setup we worked with. As stated before, the IR system we developed aims at providing a solution for the Touché Task 1, which is supporting users who search for arguments on a controversial topic, retrieving relevant documents from a crawl of online debate portals.

The corpus used for this task comprises a 4 JSON files which have in total 387740 documents retrieved from the debate protals Debatewise (14 353 arguments), IDebate.org (13 522 arguments), Debatepedia (21 197 arguments) and Debate.org (338 620 arguments). The arg.me corpus[2] can be downloaded at dataset.

As far as evaluation is concerned, we used the trec_eval software to assess the efficiency of the system. In fact, we were provided last year's topicswith the relative relevance judgement and the evaluation measure used by the Touché committee is nDCG, more specifically nDCG@5.

Our system is available at the following BitBucket repository url. Moreover, the directory `src` contains all the source code and some code for testing while `experiment` contains some runs, the qrels for 2020 (inside the `qrels` directory) and the topics both of 2020 and 2021 (inside the `topics` directory).

As far as hardware is concerned, we used the Tira software for the submission to the Touché Committee and we used our local machines for testing, therefore the system was tested both on Windows and MacOS environments.

5. Results and Discussion

Overall, we are satisfied with the performance of our system both in terms of effectiveness and stability. The overall nDCG@5 for 2020 topics is 0.7333. The nDCG@5 for each topic is provided in the figure 4 which compares our system's performance with the one of our starting baseline.

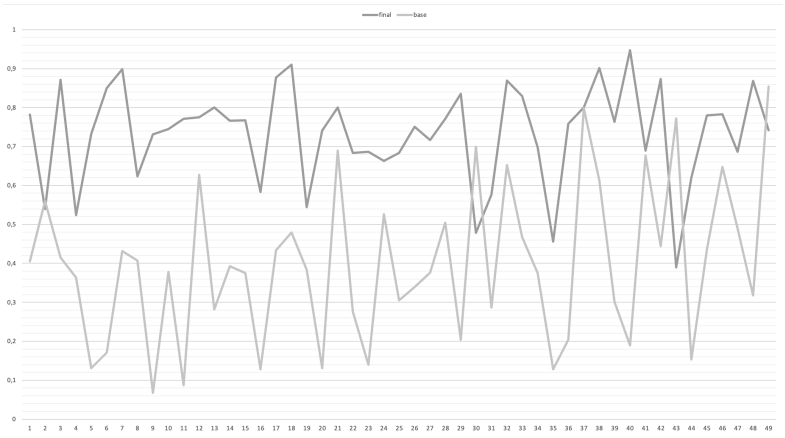

Figure 4: comparison from our system (final) and the starting baseline (base).

We gave a lot of importance to the advice of the Touché Committee therefore we used statistical models, like DirichletLM, and privileged argument quality over quantity. In fact, while parsing we extract only the core premise of the document in order to produce a quality argument when searching for topics. As far as query expansion is concerned, we conclude that, given the specificity of the topics, query expansion slightly helped in achieving our goal however costs outweigh the benefits.

6. Conclusions and Future Work

We confirm that statistical models works good for this type of documents. After testing OpenNLP models we arrive at the conclusion that these kind of models are computationally heavy and provide very light improvements if none with this corpus.

As further improvements, we think that using some Machine Learning techniques, e.g. Sentiment Analysis, could improve our system's performance. For example, knowing a-priori if a query expects pro or con arguments we can filter the results in order to better match user sentiment. In addition, a more precise query expansion could be done by using a bigger and more specific synonyms database.

References

[1] Toché Task Commitee, Touché task 1: Argument retrieval for controversial questions, 2021. URL: https://webis.de/events/touche-21/shared-task-1.html.

[2] Yamen Ajjour, Henning Wachsmuth, Johannes Kiesel, Martin Potthast, Matthias Hagen & Benno Stein, args.me corpus (Version 2020-04-01) [Data set]. Presented at the 42nd German Conference on Artificial Intelligence (KI 2019), Kassel, Germany: Zenodo. (2020). doi:10.5281/zenodo.3734893.

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