A Search Engine System for Touché Argument Retrieval task to answer Controversial Questions

Edoardo Raimondi[a], Marco Alessio[a] and Nicola Levorato[a]

[a]*University of Padua, Italy*

Abstract

We present a search engine system capable of retrieving relevant arguments inside a controversial questions forum. Our focus is on processing the corpus, indexing the collection and searching for each query while also addressing the document quality problem implementing a machine learning approach. Furthermore, we provide a brief evaluation of our retrieval results based on 2020 topics for the Touché Argument Retrieval task.

Keywords

Touchè 2021, Argument Retrieval, Search Engines

1. Introduction

This report has the aim to describe the project developed for the Search Engines course 2020/21 of the University of Study of Padua, by accompanying the reader through our system in a path of increasing specificity.
The task of the homework is the Touché Argument Retrieval for Controversial Questions.
The goal of this task is to support 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.

The paper is organized as follows: Section 2 introduces related works; Section 3 describes our approach; Section 4 explains our experimental setup; Section 5 discusses our main findings and reports a statistical analysis on the 2020 edition data; finally, Section 6 draws some conclusions and outlooks for future work.

2. Related Work

The baseline that literature provides us for this specific Controversial Questions retrieval, is composed by the usage of BM25, TF-IDF, DPH.

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

edoardo.raimondi@studenti.unipd.it (E. Raimondi); marco.alessio.1@studenti.unipd.it (M. Alessio); nicola.levorato.2@studenti.unipd.it (N. Levorato)

Successful works (from 2020 edition) instead, tried to go further. They considered and managed also:

1. query expansion (WordNet synonyms/ antonyms -> GPT-2 generation);
2. document representation using transformers;
3. re-ranking on argument quality prediction;
4. re-ranking based in sentiment (neutral sentiment);
5. pseudo-relevance feedback.

Even if all of those points are essential in order to provide a noteworthy system, in this case we will focus on query expansion and re-ranking on argument quality prediction.
It will then follow a statistical and performance analysis of the chosen approaches.

3. Methodology

The general structure of our search engine system can be nimbly divided into 6 parts, which corresponds to a different package in our source code.
In particular each component has a different aim:

1. parsing corpus files and extracting documents;
2. analyze input documents to extract tokens;
3. build the index;
4. searching for each provided topic;
5. provide a linear classification of the retrieved documents;
6. provide different classes to execute the main functions of the project.

The details of each of them are separately reported in the following pages.

3.1. Parsing

The parsing part of our system is implemented in the package named "parse". This section contains the ToucheParser class used to parse every JSON corpus file inside the corpus directory of the project.
The parsing is done using the help of the Gson library that deserialize the args.me corpus in a streaming fashion. This is mandatory since the files are too large to be fitted in the main memory for the parsing.
When an object instance of the ToucheParser class is created, the constructor takes the path to the corpus file and it is deserialized using a JsonReader that is the GSON streaming JSON parser. Then the class implements some basic methods like the iterator, hasNext and next functions used to access between the parsed documents.

While parsing the files of the corpus, the ToucheParser extract each document from the JSON structure converting them into a ToucheDocument object. This class contain the most relevant

fields (like the id, the text, the conclusion) that are common in each document of the corpus and a method used to convert the object into an object instance of the ParsedDocument class that is the one used to represent a document inside all other sections.
The ParsedDocument class represents a parsed document to be indexed in fact it contains two main fields:

1. the id: is the unique identifier contained in each document of the corpus;
2. the body: is composed by appending each other relevant field of the document (like the conclusion and the text in the premise) into a single string if they are not null.

The following figure represent an example of a document in JSON format inside the corpus file "parliamentary.json". The three fields highlighted are the ones that are extracted during the parsing.

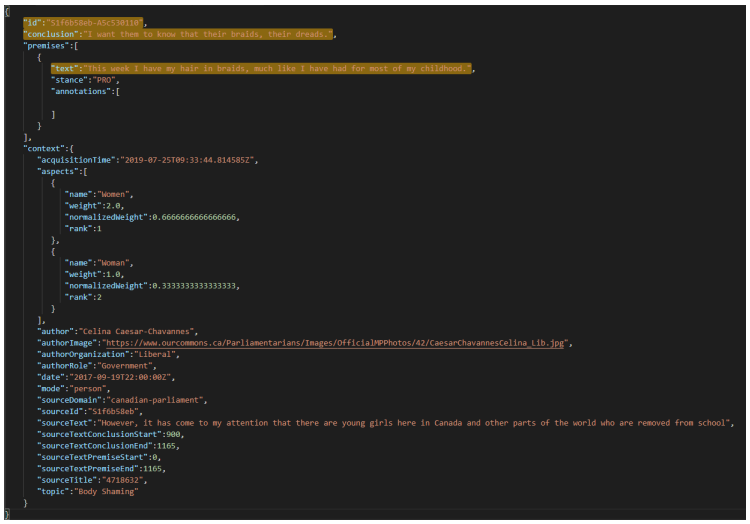

Figure 1: Example of a document in JSON format

3.2. Analyze

The analyze part of our system is implemented in the package named "analyze". This section contains the analyzer used to parse each document of the corpus and produce for each of them the corresponding stream of tokens.
The first phase for our analyzer is to try to filter out all junk data from the documents text. In the final version of the project, we found that the best compromise between quality of the results and time spent was to just discard all links through a regular expression matching filter.

Then tokenization takes place, which splits the stream of text in tokens in correspondence of white spaces and the most used punctuation characters. Then we remove all possessive forms ('s) from the end of each token and we transform them in lowercase.

The second-last phase uses a stop list in order to discard the most frequent english words from each document, that would be of no use later during indexing and searching.
At the end we apply a discard filter that aims to remove all tokens that contains non letter characters.

The processing done by our analyzer is very basic. During the development phase, a more advanced analyzer has been deployed; it was aimed to group together in a single token multiple words that together forms a unit with a specific meaning different from the one of the single words (i.e. "get" and "up" are fused together in "get up"). Because of the lack of results improvements during searching and the very tight project deadline, we have run out of time to further developing it and so we were forced to drop it.

3.3. Indexing

The indexing part of our system is implemented in the package named "index".
It does exactly what a standard Lucene indexer do.
It is based on two classes:

1. ToucheIndexer class: It builds the index inside the experiment/index folder.
   It uses the ToucheParser class described before to extract documents from the corpus files and remove any duplicate with an HashSet. Since there are some short documents in the collection that are not relevant, all documents with a number of characters in the body less than a fixed number (in our case 10 character) are discarded.
   The ramBuffer size used is 256.
   Among a few similarity functions tested, BM25 seemed to be the best one for our case;
2. BodyField class: It represents a Lucene field for containing the body of a document.
   It's also used to set or enable some option like the term vectors.

It is important to mention that the analyzer used in this section, whose details are described in the previous paragraph, is the same used for searching purpose.

3.4. Searching

The searching part of our system is implemented in the package named "search". The class ToucheSearcher performs all tasks related to search. For each topic, we use our analyzer to obtain the corresponding stream of tokens that represent it. With them, we can build the query which Lucene uses to retrieve a list of documents relative to the topic.

In order to get better results we perform query expansion: we add to the query, for each token:

- similar forms of the word (e.g. conjugation of verbs, plurals of names, ...), with a total relative weight of 50% c.a.

- synonyms of the word, with a total relative weight of 50% c.a.
- antonyms of the word, with a total relative weight of 20% c.a.

With total relative weight we mean the sum of the individual weight of the words added; note that they are all the same in our current implementation.

The dictionary used in this phase is an ad-hoc modified version of the freely-available WordNet database (available at: https://wordnet.princeton.edu/, it uses its own "WordNet 3.0" license that allows to freely use, copy, modify and distribute it without fee or royalty) obtained by discarding all synsets that are formed by more than a single word.

In order to parse the topic xml files that contains all the query to be searched we developed a specific class : XmlTopicsParser. It extract each query from a topic xml file using the DOM API that create an in-memory representation of the xml. It returns a QualityQuery array containing all the query parsed.

Once the desired number of documents has been retrieved by our search function, we then applied machine learning subroutine to all of them. (At the beginning it was considered to apply machine learning only on the top n retrieved documents, but this decision should be made in light of the assumption that the first n documents are all relevant with respect to the current query. This assumption can't be applied in our case since our system is not good enough to ensure such a strong assumption.)

Given this premise, since we are not circumscribed in a binary relevance judgment, it is necessary to consider and evaluate several aspects. Among all these possible aspects, in this specific subroutine, we are going to take care of the general writing quality. Basically we are not confident in saying that the first n documents are relevant and it becomes challenging at this point to understand how much relevant, in term of quality, a document is. In order to try to reach an acceptable bound in these terms, a linear regression model has been trained.
The training part has been object of interesting considerations that is worth describing separately. (see Section 3.5).
Suppose we now have our run with all the k documents. We let our pre-trained model classify each of them and save the corresponding predicted quality in a dictionary data structure.

The next step is to provide a final score that mix the BM25 searching score and the machine learning score. There are several way to do it but our choice (made upon measure evaluation, see section 5) has fallen in the weighted average between the two scores.
Once done it, the run is "re-write" as an output, according to the new scores (i.e. weighted average).

3.5. Linear Regression

It is implemented inside the package called "linear regression".
For this purpose we leaned on the machine learning library called "weka".
Touchè organizers provides us a dataset of arguments with relative labels including one named "Combined Quality". This feature describes the combination of rhetorical, logical and dialectical quality of a specific arguments and we are going to use it as a label for the training part.
The general pipeline is : train a linear regression model, classify all the retrieved documents and reorder them by mixing the predicted quality score by our model and the BM25 score (as described in section 3.4).
The training of the model has been performed as follow:

1. Document vectorization (both for training set and fresh data)
2. Fitting
3. Evaluation

Since the last two steps are quite standard, we will stress our explanation on the first operation. The application of the model on our system is done, and already exposed, in the searching part.

3.5.1. Document vectorization

To represent complex sentences as a vector of numbers, literature provide us a large amount of techniques. Both for lack of time and experience on this field, the chosen workflow is one of the simplest but it works properly.

First at all documents passed trough an ad-hoc customized Lucene analyzer, which applied stop word removal (with the same stop list used for the corpora to ensure more compatibility), stemming and filtering based on word lengths. We encapsulate the k most frequent words of a given domain (e.g. entire training set or set of retrieved document i.e. fresh samples to classify) in a general vector $mostFrequentWords$.
Then we need to encode each document (in this specific case a multi hot encoding is performed).
To do it, we constructed a vector v of size k+1. We set 1 or 0 the cell i if the document contains the i-th most frequent word. (e.g. $v[0] = 1$ if the document contains the most frequent word, 0 otherwise, and so on). After this step only k cell of mt vector v will be filled.
Finally in the last position we set the "Combined quality" feature in order to use it as a label. (Note: even in the classification part this k+1-th element need to be present, even if it will not obviously be consider. The reason why weka required that is still a mistery...)
All the vectors (i.e. the documents) are stored in csv files. This is done to facilities the communication between our code and weka functions, since they load data via files of this type.

3.5.2. Fitting and Evalutation

In order to fit and evaluate our model, we took our pre-vectorized training set and we split it in training and test set. Once fitted our model on the training set, we evaluate it using least mean squared error as loss function.
The evaluation helps us to tune the hyperparameters of document vectorization in the best possible way. The most important parameters were: which stop list to use, how many most frequent word to consider and the usage of the absolute frequency instead of other possible metrics.

All this work is done by different classes, as good modular programming required, which names are self-descriptive: CSVCleaner, RegressionAnalyzer, CSVWriter, Encoder, ToucheRegression.

3.6. Execution

The package "exec" has been made thinking outside the pure information retrieval purpose, but it is more a child of a software engineering point of view.
This section contains all the executable classes of the project and it is used mainly for debugging purpose. For example the Indexing class is used to build the index in the experiment directory, the Parsing class is used to get a look of the documents extracted from a chosen corpus file and the Searching class is used to build the runs.

All the corpus files should be placed inside a corpus directory, the directory experiment contains the index and the run files and all the topic xml files should be inside a topics directory.

4. Experimental Setup

Our experimental set up is made upon:

- as documents collection we used the args.me corpus (version 2020-04-01) available at https://zenodo.org/record/3734893#.YIxOLbUzaUk
- as evaluation data we used the 2020 topics available at https://webis.de/events/touche-21/shared-task-1.html#data
- as evaluation measures we massively used NCGD (because it is the same used in the 2020 touche results) but without forgetting to think about precision, recall, average precision and all the measures provided by trec eval program.
- the url of our git repository is https://bitbucket.org/upd-dei-stud-prj/seupd2021-mr/src/master/

5. Results and Discussion

5.1. Results of our system

The score provided by Lucene and the BM25 similarity judges only the relevance of a document, without considering the overall quality of it, such as the writing style and the length. On the other hand, the score provided only by Machine Learning takes care only of the general writing quality. Therefore we have decided that combining the two scores could get better results.

In order to do so, first we have to make both scores comparable, by normalizing them in the [0.0, 1.0] range using this formula:

$$score_{normalized} = \frac{score - min_{score}}{max_{score} - min_{score}} \tag{1}$$

Then we proceed by applying a linear combination to them. We used a factor *alpha* in the range [0.0, 1.0] to weight the score given by BM25 similarity and 1 – *alpha* to weight the score given by the machine learning approach. The formula we used to combine the two score is the following:

$$score_{mixed} = alpha * normScore_{BM25} + (1 - alpha) * normScore_{ML} \tag{2}$$

If the value of *alpha* is within the range [0.5, 0.8], the quality of the run increases w.r.t the two pure approaches (pure BM25 at *alpha* = 1.0, pure ML at *alpha* = 0.0), reaching the best results at about 0.6 (we choose to use NDCG@5 as metric of choice because is the one used in Touché 2020 results), as shown in Figure 2. This means that the BM25 score should be higher weighted w.r.t ML score, that is the relevance judgement is more important than the writing quality of the documents.

alpha	NDCG@5
0,00	0,0988
0,10	0,1461
0,20	0,2210
0,30	0,3311
0,40	0,4557
0,50	0,5587
0,60	0,6184
0,70	0,5969
0,80	0,5728
0,90	0,5162
1,00	0,4624

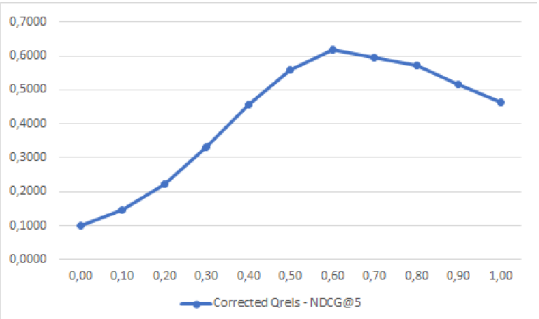

Figure 2: Scaling of NDCG@5 measure w.r.t. changes of *alpha*.

From our tests we saw that all major measure values of the run increases using this mixed approach: by keeping the top 30 of the 1000 documents initially retrieved we found the table in Figure 3.

	BM25 Only	ML Only	Mixed 0,6
#Relevant Docs Retrieved	318	110	345
#Relevant Docs	932	932	932
#Retrieved Docs	1470	1470	1470
Recall	0,3412	0,1180	0,3702
P@5	0,4653	0,1102	0,5959
P@10	0,3878	0,0918	0,4449
P@30	0,2163	0,0748	0,2347
NDCG@5	0,4624	0,0988	0,6184
NDCG@10	0,4233	0,0906	0,5211
NDCG@30	0,4234	0,1105	0,4950

Figure 3: All major performance measure for all three approaches: pure BM25 score, pure ML score, and mixed ($alpha$ = 0.6).

For all our test we have used vectors of size 250 (i.e. max number of most frequent words) to train the linear regression model. Our test shows that changes in the number of most frequent words considered has almost no impact on the NDCG@5 measure, as shown in figure 4. In the final version of our code, we decided to stick with the top 250, because the impact on runtime is still negligible and it should provide a more robust solution compared to using a lower number of words.

#Words	NDCG@5
50	0,6016
100	0,6130
150	0,6181
200	0,5974
250	0,6184
300	0,6215
400	0,6132
500	0,5963

Figure 4: Scaling of NDCG@5 measure w.r.t. the number of words used in ML phase.

5.2. Results of our system compared to Touché 2020 best system

For this comparison, we chose from Touché 2020 runs (available at https://webis.de/events/touche-21/runs-task-1-2020.zip, dataset v2) the one with the highest average NDCG@5 measure. The chosen run is "DirichletLM" by team "Swordman", against a run obtained from our system using $alpha$ = 0.6.

Since both runs retrieve 1000 documents for each topic, the first metric we decided to analyze is the recall for each topic. As we can see in Figure 5, our system performs significantly better as shown by the average recall: 0.8637 and 0.7693.

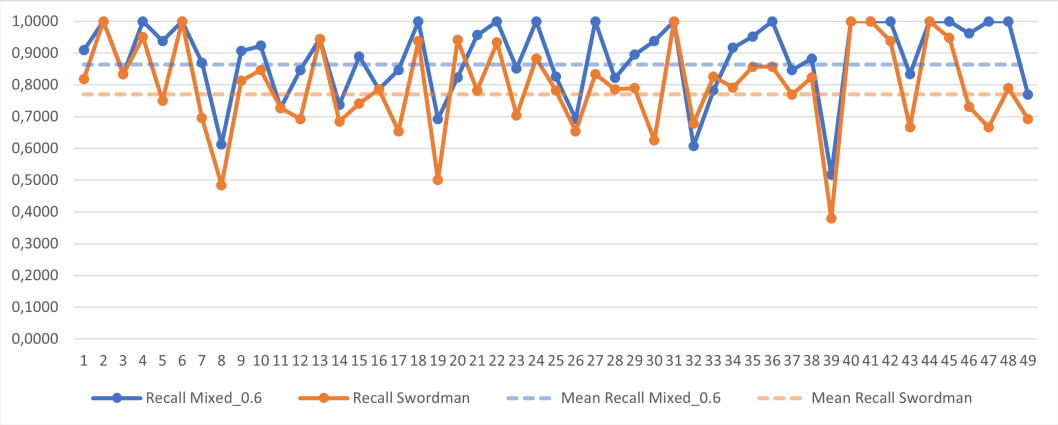

Figure 5: Recall of all 49 topics of both runs.

Although the higher recall, the Figure 6 shows that our systems performs significantly worse w.r.t "Swordman" run when we consider the NDCG@5 metric (average NDCG@5: 0.6184 and 0.8266). From this data, we can conclude that our system retrieves more significant documents but it ranks them poorly: in order to achieve higher results, efforts should be focused in this aspect.

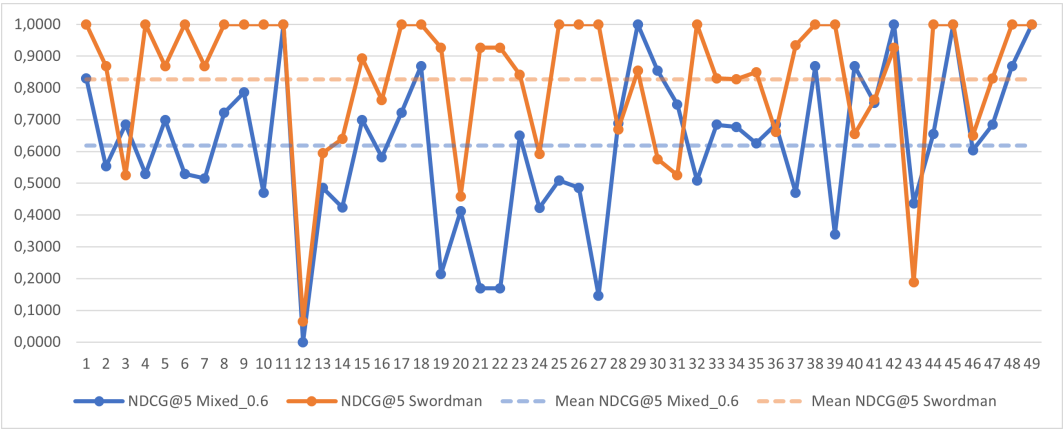

Figure 6: NDCG@5 of all 49 topics of both runs.

5.3. Statistical analysis

In order to better understand the behavior and the meaning of our results a proper statistical analysis is needed. Our main goal is to ensure, with much confidence as possible, the improvement of our approach with respect to a basic BM25 approach.
We computed 5 different runs: 1 using only BM25 and 4 adding also machine learning and changing some parameters according to what already said in section 5. Each run is represented by a 49 length array v (since we have 49 topics) and each elements of this array is a measure score for that topic. (i.e. $v[1]$ = AP of topic 1). Specifically we will consider in our analysis the average precision and NDCG@5.
Now let's have a look to our data in a more formal way. Then we will continue our investigation via hypothesis testing.

5.4. Data visualization

First at all we start with average precision.
A first overview is essential to give us a general informal taste of our data.

Baseline.BM25	Mixed0.57	Mixed0.6	Mixed0.62	Jolly
Min. :0.0353	Min. :0.0379	Min. :0.0385	Min. :0.0379	Min. :0.0316
1st Qu.:0.1477	1st Qu.:0.2106	1st Qu.:0.2071	1st Qu.:0.2063	1st Qu.:0.2200
Median :0.2258	Median :0.3060	Median :0.2830	Median :0.2824	Median :0.3184
Mean :0.2684	Mean :0.3175	Mean :0.3194	Mean :0.3202	Mean :0.3216
3rd Qu.:0.3881	3rd Qu.:0.4036	3rd Qu.:0.4213	3rd Qu.:0.4188	3rd Qu.:0.3963
Max. :0.7958	Max. :0.7146	Max. :0.7232	Max. :0.7389	Max. :0.7380

Figure 7: Statistical summary of data on average precision.

Before doing any consideration, let's clarify the notation. The 3 central columns represent runs with different weights scoring as already explained in section 5 (see Figure 2). Jolly instead is a run with slightly different searching parameters. This is done to check the variance of our machine learning model. (i.e. we expect it to be similar to the other machine learning runs). Finally Baseline BM25 is self descriptive.

Now consider a box plot representation.

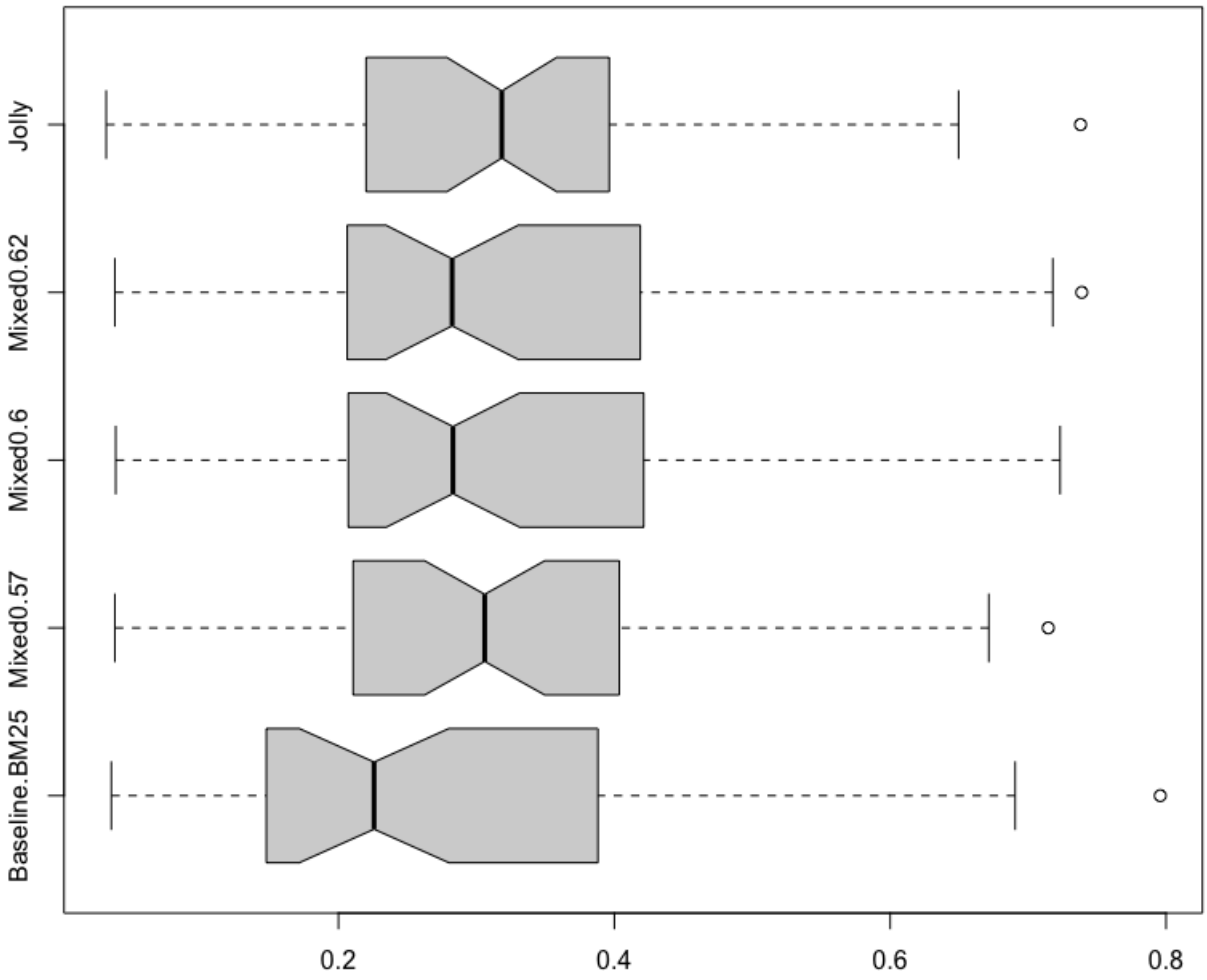

Figure 8: BoxPlot of the 5 runs on average precision.

This first plot give us a preliminary intuition about the scenario. It is indeed visible that there is a better behaviour of machine learning runs.

Another useful representation can be done using stripcharts.

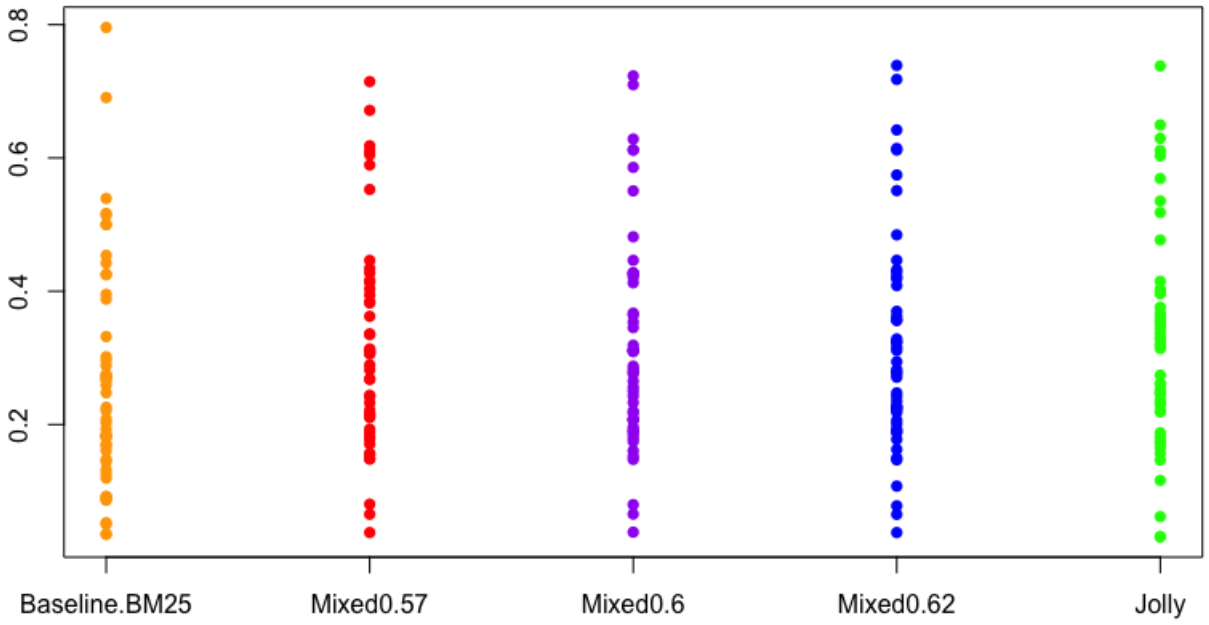

Figure 9: StripChart of the 5 runs on average precision

Here the visualization of the outliners is more evident and let us performs some considerations. Even if Baseline BM25 reaches higher AP scores, it reaches it in isolation, while, Mixed and jolly run provide a more dense higher scores. Also the points with more density for BM25 stand in a very low score area. This suggest us that the two main systems (baseline and machine learning) are different, in particular, the last one provides better results.

Exactly same considerations can be conducted looking at NDCG@5 as scoring measure, instead of average precision.

BM25	Mixed_0.5789	Mixed_0.6	Mixed_0.6190	Jolly
Min. :0.0000	Min. :0.0000	Min. :0.0000	Min. :0.0000	Min. :0.0000
1st Qu.:0.2426	1st Qu.:0.4852	1st Qu.:0.4852	1st Qu.:0.5087	1st Qu.:0.5087
Median :0.4852	Median :0.6040	Median :0.6548	Median :0.6504	Median :0.6548
Mean :0.4624	Mean :0.6086	Mean :0.6184	Mean :0.6199	Mean :0.6083
3rd Qu.:0.6844	3rd Qu.:0.7227	3rd Qu.:0.7530	3rd Qu.:0.7530	3rd Qu.:0.7883
Max. :1.0000	Max. :1.0000	Max. :1.0000	Max. :1.0000	Max. :1.0000

Figure 10: Statistical summary of data on NDCG@5.

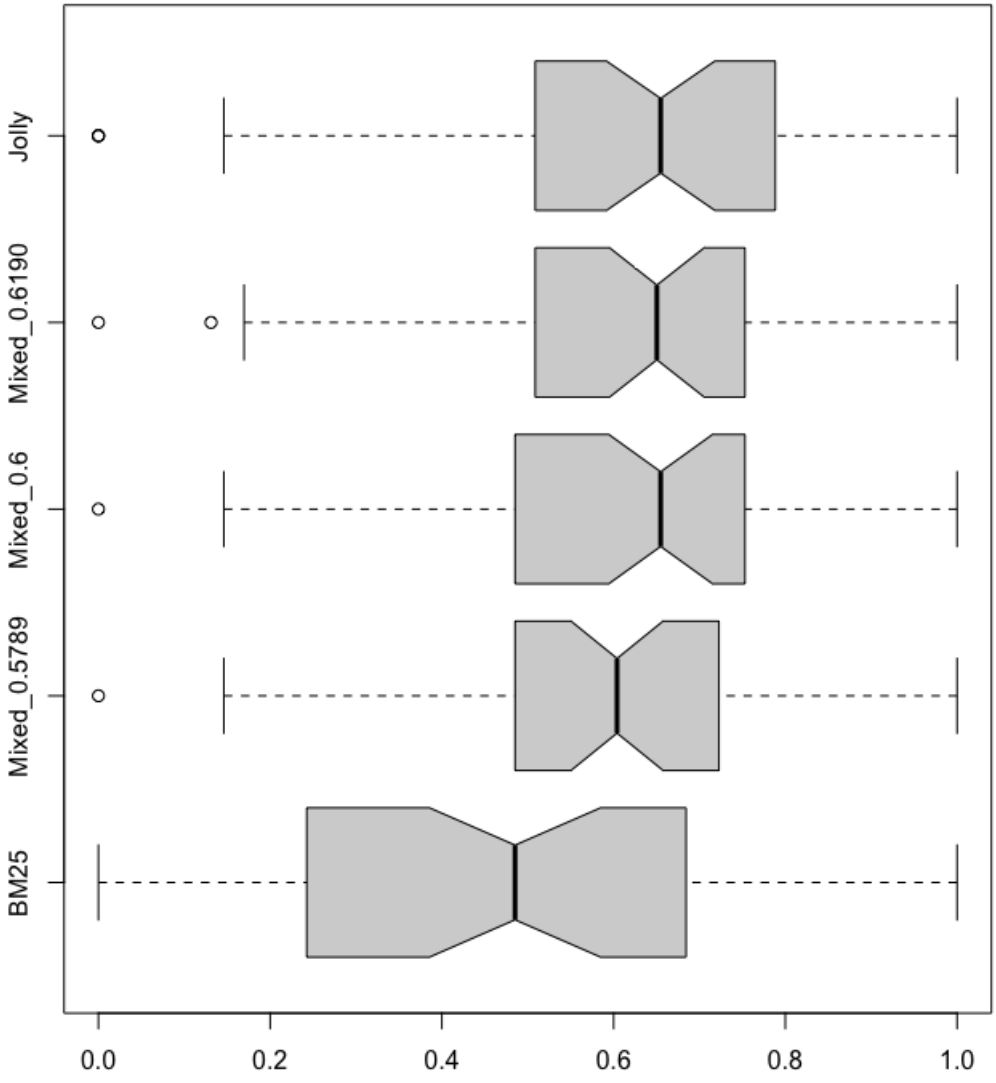

Figure 11: BoxPlot of the 5 runs on NDCG@5.

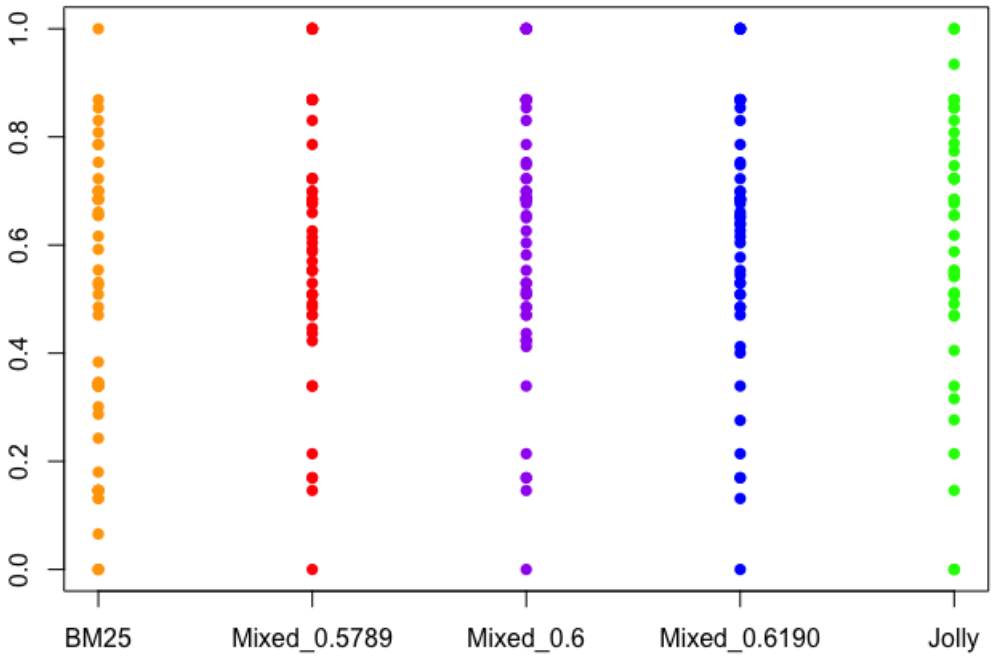

Figure 12: StripChart of the 5 runs on NDCG@5

This informally gives us more confidence about our principal intuition (i.e. there is an improvement adding our machine learning subroutine to baseline BM25).
We need to better formalize it via hypothesis testing.

5.5. Hypothesis testing

Now let's validate our assumptions using more sophisticated statistical analysis technique such as hypotesis testing and t-test.
As we stated before we are now intent on verifying the actual differences between our systems (i.e. runs).
In order to do that it is necessary to correctly formalize the null and the alternative hypothesis and define a suitable significant level α.

Hypothesis are:

1. H0 : the two runs are not different (i.e. are sampled from the same distribution);
2. H1 : the two runs are different (i.e. are sampled from different distributions).

Confidence level $\alpha = 0.05$.

A t-test has been performed between every possible combination of our 5 runs. We expect baseline BM25 to be significantly different with respect to machine learning ones (i.e. reject null hypothesis). While it is reasonable to see that the ones with machine learning are equal to each other (i.e. fail to reject null hypothesis).

Let's now take a look to the p-value results of our t-test calculated both on average precision and NDCG@5. They will be represented by a matrix t.

```
     [,1]         [,2]         [,3]         [,4]         [,5]
[1,]   NA 0.0003813653 4.423576e-05 8.366593e-06 0.0003082983
[2,]   NA           NA 4.355908e-01 4.912691e-01 0.6399939838
[3,]   NA           NA           NA 7.553086e-01 0.7969078663
[4,]   NA           NA           NA           NA 0.8637252508
[5,]   NA           NA           NA           NA           NA
```

Figure 13: p-value t-test matrix on average precision

```
     [,1]         [,2]         [,3]         [,4]         [,5]
[1,]   NA 5.81648e-05 4.544816e-06 1.820339e-06 2.495757e-05
[2,]   NA           NA 2.151513e-01 2.122997e-01 9.922500e-01
[3,]   NA           NA           NA 8.309027e-01 6.781834e-01
[4,]   NA           NA           NA           NA 6.226001e-01
[5,]   NA           NA           NA           NA           NA
```

Figure 14: p-value t-test matrix on NDCG@5

The runs are respectively baseline BM25, Mixed 0.57, Mixed 0.6, Mixed 0.62, Jolly.
Of course the matrix t is populated only on the top diagonal area since t-test(X, Y) = t-test(Y, X).
Particularly attention has to be made upon the first row. It indeed represents the test between the baseline and all the other systems. All the p-values on $t[1, :]$ are significantly below our significant level (i.e. $t[1, :] < \alpha$). This validate our preliminary assumptions and it tells us that adding a machine learning subroutine provides different (and better, looking at the data) results. It also suggest us that we could decrease a lot our confidence level without affecting the final statement. So we reject null hypothesis.
Another important confirmation arrives from the analysis of rows 2,3 and 4. They contain the p-values across system using machine learning but with different hyperparameters. In particular p-values are above to α. We can conclude that we fail to reject the null hypothesis in this case. In terms of our system this means, no matter which hyperparameters one choose, the

subroutine will provide results sampled from an unique distribution.
Finally, the fact that all these results are valid for both measures (i.e. average precision and NDCG@5) gives us even more certainty about our beliefs.

5.6. Failure analysis

A little bit of failure analysis has been performed. Both for lack of time and workforce, we considered only one topic, that is the worst case topic (i.e. it reaches 0.000 on NDCG@5). Let's have a look to it:

<topic>
<number>12</number>
<title>Should birth control pills be available over the counter?</title>
<description>
The easy access to birth control pills may have repercussions on people's everyday behavior in taking precautions, while disregarding negative side effects. A user wonders whether birth control pills should be prescription drugs, so that doctors have a chance of explaining them to their users. </description>
<narrative>
Highly relevant arguments argue for or against the availability of birth control pills without prescription. Relevant arguments argue with regard to birth control pills and their side effects only. Arguments only arguing for or against birth control are irrelevant.
</narrative>
</topic>

Faced with this topic, our system mainly returned general discussion about arguing for or against birth control, that is exactly what we do not wanted to retrieve (as specified in the narrative). This could be due to the fact that in order to totally understand the topic it is absolutely necessary to give maximal importance to the description field. Our system instead, unfairly favors the words in the title, giving them much greater weights with respect to words in the description fields.
In order to solve this situation a dynamic weights assignment could be developed.
However, looking at the runs of the previous year, it turned out this topic to be really tough for everybody. This surely does not resolve our problem but how we say in Italy: "mal comune, mezzo gaudio".

6. Conclusions and Future Work

To conclude, our works has enlighten different ways to pursue and manage in order to deal with human made documents in a forum scenario. Indeed this scenario is very challenging, due to the document corpus presenting a lot of both intra and inter documents noise. (e.g. entire completely useless documents or document bodies full of bad written argument).

Given all the technical results already provided in the previous sections, we can claim that most of the easily achievable improvements lies on developing a better documents ranking system. The things that have to be considered when trying to improve it are:

- for ML phase, instead of just using a binary model, different parameters could be used to fill our $mostFrequentWords$ vector, such as Inverse Document Frequency (IDF), relative frequency or other state of the art metrics;
- try to use other machine learning models instead of linear regression, such as Support Vector Machine (SVM) or Neural Networks (NN).

The study of documents collection is of fundamental importance to construct a good retrieval system: it's difficult to provide universal assumptions regarding them, and they must be treated case by case to get the best results. Although our system recalls a very high percentage of the relevant documents (about 86%, as shown in Section 5), there is still room for improvements regarding this aspect:

- develop a more sophisticated analyzer, in particular by using shingles and more advanced techniques of natural language processing;
- try different similarity functions;
- dynamic query weights assignment.