WSM project 2

Building IR systems based on the Lemur Project

Judy Chan

MIS 107306052

May 2022

1. Abstract

This project will implement and compare various retrieval systems using vector space models and language models, by using the toolkits of Indri-5.14 to run over 50 TREC queries against WT2G collection, and perform an evaluation of each methods by a wealth of statistics, which are about how well the uploaded file did for those queries provide by *trec_eval* program.

2. Prerequisite and Introduction

The section provides information for setting up environment for indri-5.14 and the demonstrate the basic steps, e.g., the sample command for indexing, retrieval based on the Indri parameters files, and evaluation tools.

Note: Indri can be obtained from the SourceForge Lemur Project Page.

2.1 Set up the project

The version of Indri depends on the operating system of used device, in my case, I downloaded Indri-5.14 with macOS Mojave 10.14.6. To configure successfully, make sure the gcc@6 is download and add the version index to *MakeDefns* file. After that, modify *atomic.hpp* as TA's instruction, run *make* to access Indri library and applications. Instead of install all Indri's component to my system, I included the indri-5.14 file into the project's structure like the following directory tree:

```
Wsm_Project_2
|__ indri-5.14
|__ params
|     |__ index
|     |     |__ index_nostem_stop.param
|     |     |__ index_nostem.param
|     |     |__ index_stem_stop.param
|     |     |__ index_stem.param
|     |
|     |__ retrieve
|           |__ nostem
|           |     |__ LM_fb.param
|           |     |__ LM_jm.param
|           |     |__ LM_la.param
|           |     |__ okapi.param
|           |__ nostem_stop
|           |__ stem
|           |__ stem_stop
|           (with same 4 parameter files as nostem folder.)
|__ query
|     |__ query.txt
|
|__ WT2G
|__ eval
|     |__ trec_eval-9.0.7
|     |__ qrels.401-450.txt
|     |__ trec_eval
|
|__ run.sh
|__ eval_results.py
|__ plot.py
```

- 4 different indexing methods (with/without porter stemmer * include/exclude stopwords).

2.2 Standard process to run the IR model

Step 1. Document Indexing

Create an index of the downloaded corpus. The documents are found within the WT2G folder, corpus files are in a standard format used by TREC, and in the basic way we only consider the title of each query.

Take building index without stemmer and stop words for example:

```
./indri-5.14/buildindex/IndriBuildIndex params/index/index_nostem.param
```

Step 2. Query Execution

Write a program to run the queries in *query.txt* using each of the retrieval models listed above, and output the top 1,000 results to a given path.

Take running okapi retrieval models for example:

```
./indri-5.14/runquery/IndriRunQuery params/retrieve/nostem/okapi.param
query/query/.txt > query_result/nostem/okapi.txt
```

Step 3. Evaluation and Visualization

Download *trec_eval* and use it to assess your results for each retrieval model. In order to interpret the output data more easily, I import the file *trec_eval-9.0.7*, by this standard tool used by TREC community for evaluating an ad hoc retrieval run, then visualize the summary output, such as Mean Average Precision, R-Precision, Interpolated Recall - Precision Averages at 0.10 recall, etc.

- *eval_results.py*, the python script define a object that stores the results output by obtained from *trec_eval*.

Use different command to get the different format of evaluation outcome:

a. `perl eval/trec_eval eval/qrels.401-450.txt query_result/nostem/okapi.txt`

b. `eval/trec_eval-9.0.7/trec_eval eval/qrels.401-450.txt query_result/nostem/okapi.txt`

3. Performance Comparison

3.1 The Usage of different Ranking Functions

According to the homework requirements, this project implements the following four variations of a retrieval system:

1. Vector space model, terms weighted by Okapi TF times IDF value, and inner product similarity between vectors. It can be implemented easily by specifying the baseline (non-language modeling) retrieval method in parameter file, and insert the arguments we'd like to calculate:

   ```
   <baseline> okapi, k1:2, b:0.75, k3:7 </baseline>
   ```

2. Language modeling, maximum likelihood estimates with Laplace smoothing only, query likelihood. This add-one smoothing method is not supported by Indri originally, so I create a *LaplaceMLE.hpp* in indri-5.14 by using the formula: $\rho i = \frac{mi + 1}{n+k}$.

   (m = term frequency, n=number of terms in document (doc length) , k=number of unique terms in corpus.)

   ```
   <rule> method:laplace </rule>
   ```

3. Language modeling, with Jelinek-Mercer smoothing using the corpus, 0.8 of the weight attached to the background probability, query likelihood. It can be implemented easily by specifying the smoothing rule (TermScoreFunction) retrieval method in parameter file.

   ```
   <rule> method:jm, collectionLambda:0.8 </rule>
   ```

4. Language modeling, Dirichlet smoothing using the corpus with default mu=2500, top 10 documents and 20 terms to use for feedback by adding Pseudo-Relevance Feedback Parameters, and the weight for the original query in the expanded query is set 0.5.

   ```
   <rule> method:dirichlet, mu:2500 </rule>
   <fbDocs> 10 </fbDocs>
   <fbTerms> 20 </fbTerms>
   <fbMu> 0 </fbMu>
   <fbOrigWeight> 0.5 </fbOrigWeight>
   ```

3.2 Overview of Results

Since there're two indexing methods (with or without Porter stemmer), and four ranking function as 3.1 mentioned, so we'll get 8 runs results:

To keep the report neat, in the following content use abbreviations to indicate:

- LM_la: Language model with Laplace smoothing.
- LM_jm: Language model with Jelinek-Mercer smoothing.
- LM_fb: Language model with Dirichlet smoothing and relevance Feedback.

	No Porter Stemmer			
Different Ranking Functions	**nostem_LM_fb**	**nostem_LM_jm**	**nostem_LM_la**	**nostem_Okapi**
Un-interpolated mean average precision (MAP)	**0.3004**	**0.1924**	**0.2891**	**0.217**
Precision at rank 10 documents	**0.452**	**0.302**	**0.444**	**0.398**

	With Porter Stemmer			
Different Ranking Functions	**stem_LM_fb**	**stem_LM_jm**	**stem_LM_la**	**stem_Okapi**
Un-interpolated mean average precision (MAP)	**0.326**	**0.1922**	**0.3113**	**0.2308**
Precision at rank 10 documents	**0.482**	**0.29**	**0.462**	**0.404**

Table 1: Quick Summary - Okapi / LM_la / LM_jm / LM_fb

3.2.1 The Impact of Stemming

Here are some visualized diagram with four ranking functions, generated by running:

```
$python3 plot.py $(find ./eval_results/*.txt) -f output_png_name
```

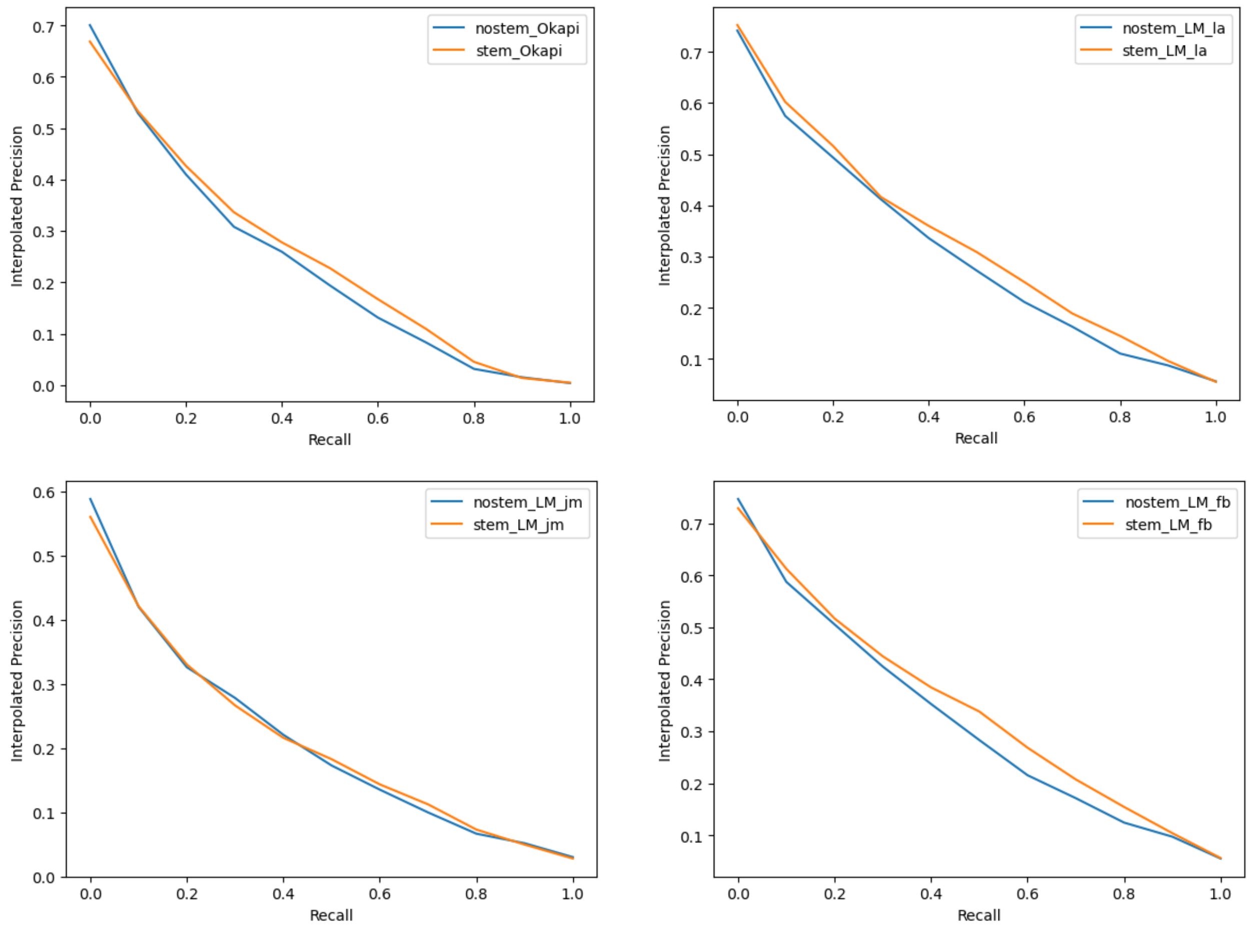

Figure 1: Interpolated Precision Curve – Comparing Stemming at Okapi / LM_la / LM_jm / LM_fb

According to the information shown in the figure, the precision without stemming(blue line) are slightly higher than those with stemming(orange line) when the value of recall are very small, but afterwards the situation will be reversed. However, only the LM with Laplace smoothing shows that indexing with Porter stemmer perform better in every level of recall.

Besides, the difference between with and without stemmer is biggest in the LM with feedback case, but it’s seeing no discrepancy with Jelinek-Mercer smoothing, and the precision results of that ranking function are worst performance.

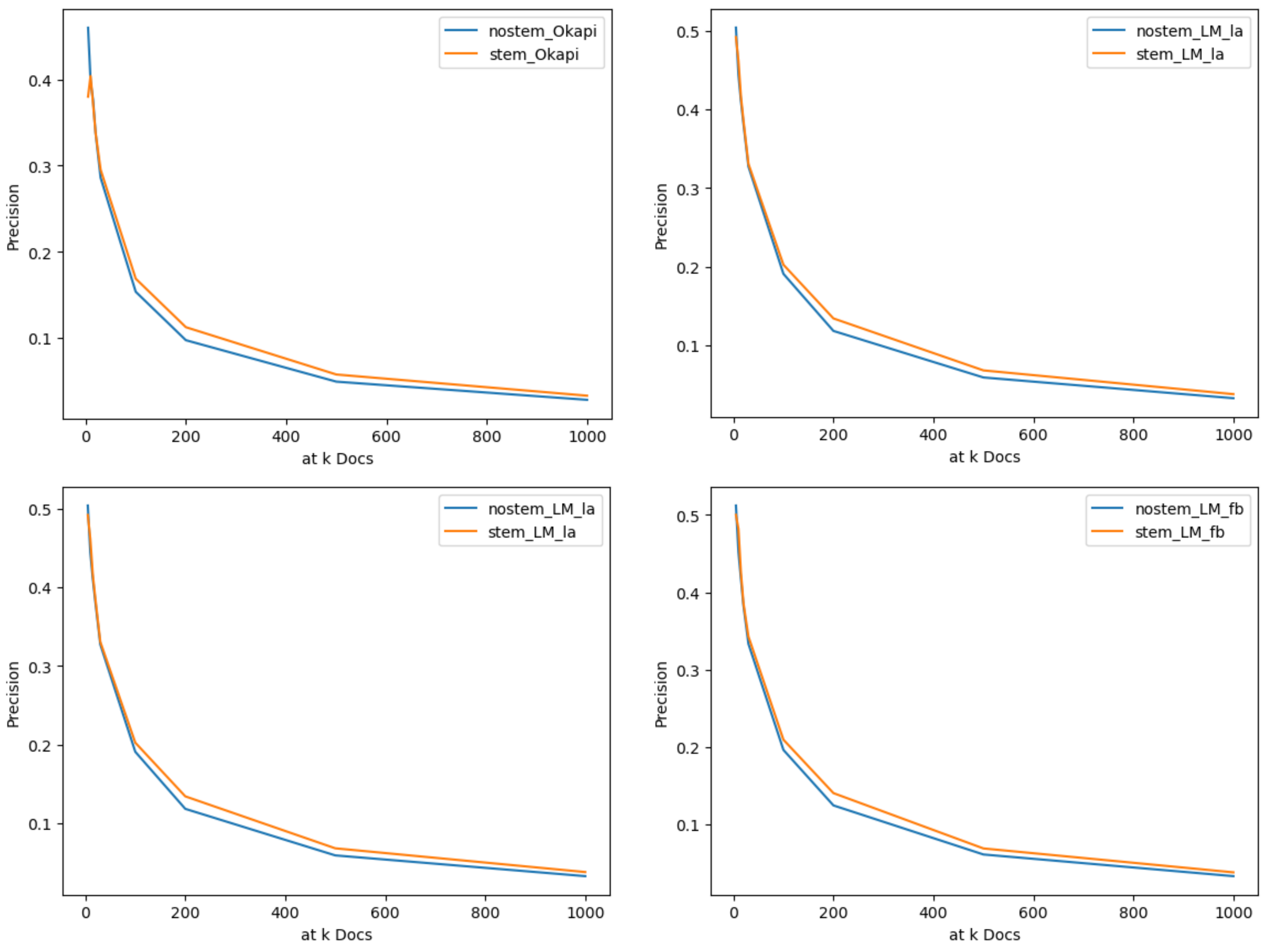

Figure 2: Precision at k Docs – Comparing Stemming at Okapi / LM_la / LM_jm / LM_fb

In Figure 2, it’s clear that indexing with stemmer have better performance in all cases, although the precision without stemmer is higher than that with stemmer in Okapi case in first few documents, but the orange line retains higher in most circumstance.

Therefore, we can conclude that the retrieval system with stemmer have better performance in most of case, but those without stemmer are also suitable for users who only need the retrieval results in previous position, but if want to consider the overall performance, then using stemmer is a reasonable choice.

3.2.2 The Impact of Ranking Functions

For this project, I apply a baseline (non-language modeling) retrieval model and there language models with Laplace smoothing, Jelinek-Mercer smoothing, and Dirichlet smoothing with relevance feedback, respectively.

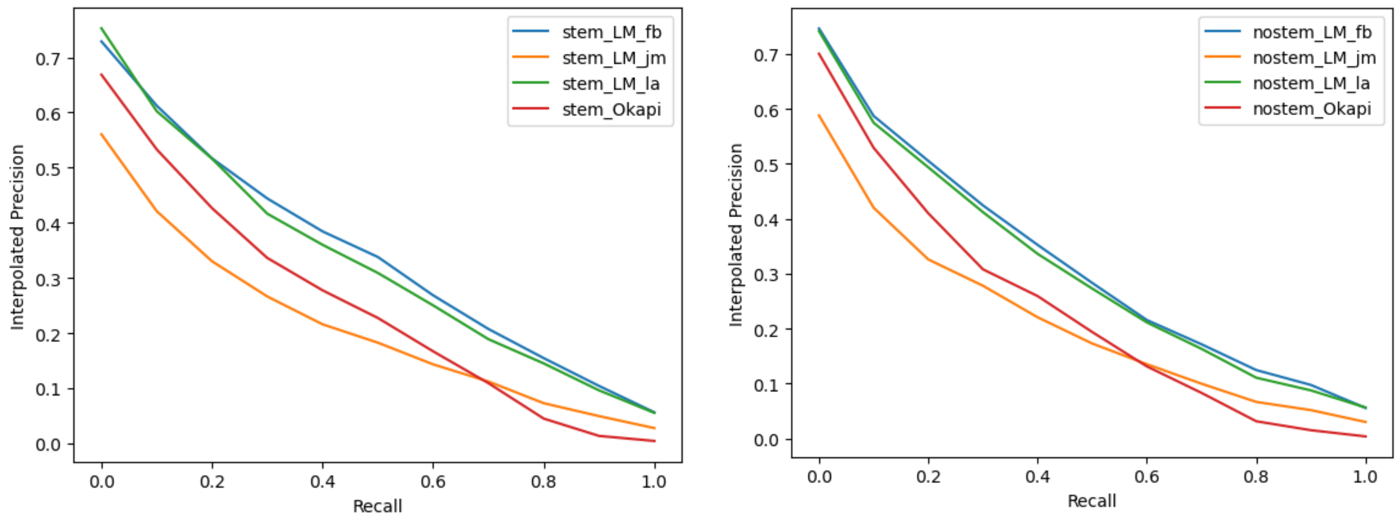

Figure 3: Interpolated Precision Curve – Compare the performance of various ranking functions

As you can see above, regardless of stemmer used in indexing, both LM_fb in the first position, and LM_la performed better than LM_jm and Okapi.

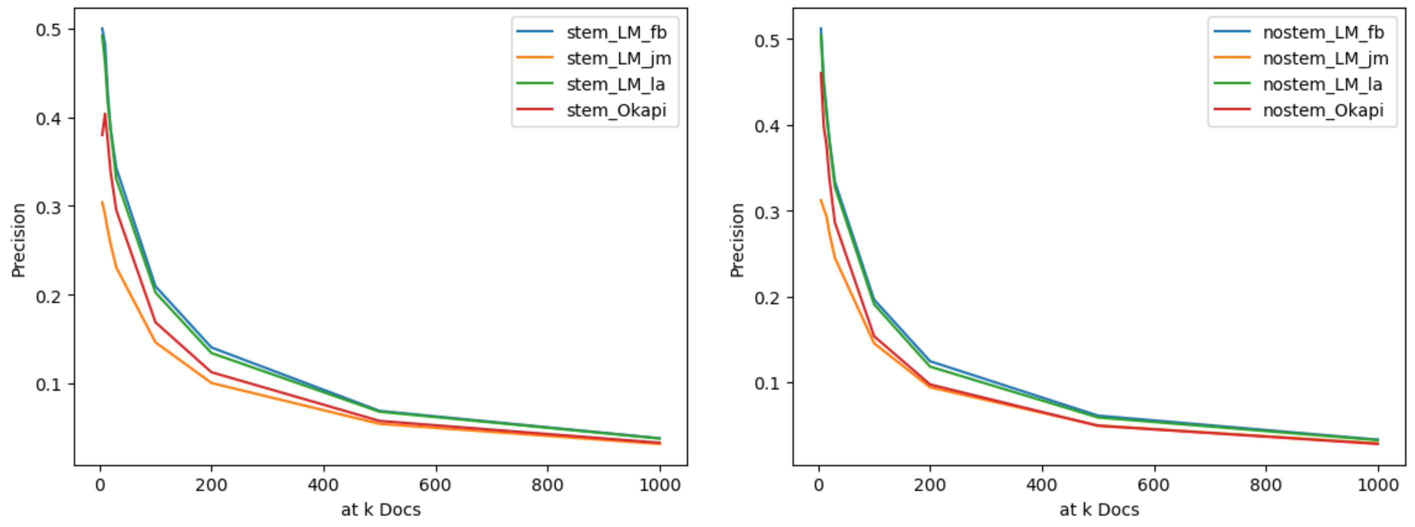

Figure 4: Precision at k Docs – Okapi / LM_la / LM_jm / LM_fb (sort by stemming or not)

The same results can be observed in figure 3 and figure 4, the blue curve (LM_fb) possess the highest precision at any given recall level, also satisfied with any amount of document retrieved in precision at k curve, which followed by the green line(LM_la). As for other two cases, the LM_jm function performs not well in both stem and nostem situations, the precision of it only surpass Okapi's after 0.7 recall level.

3.2.3 The Impact of Stop Words

Indri offers a parameter named stopper to contain many subelements named word, specifying the stopword list to filter. And there are two plots according to index used by Porter Stemmer or not.

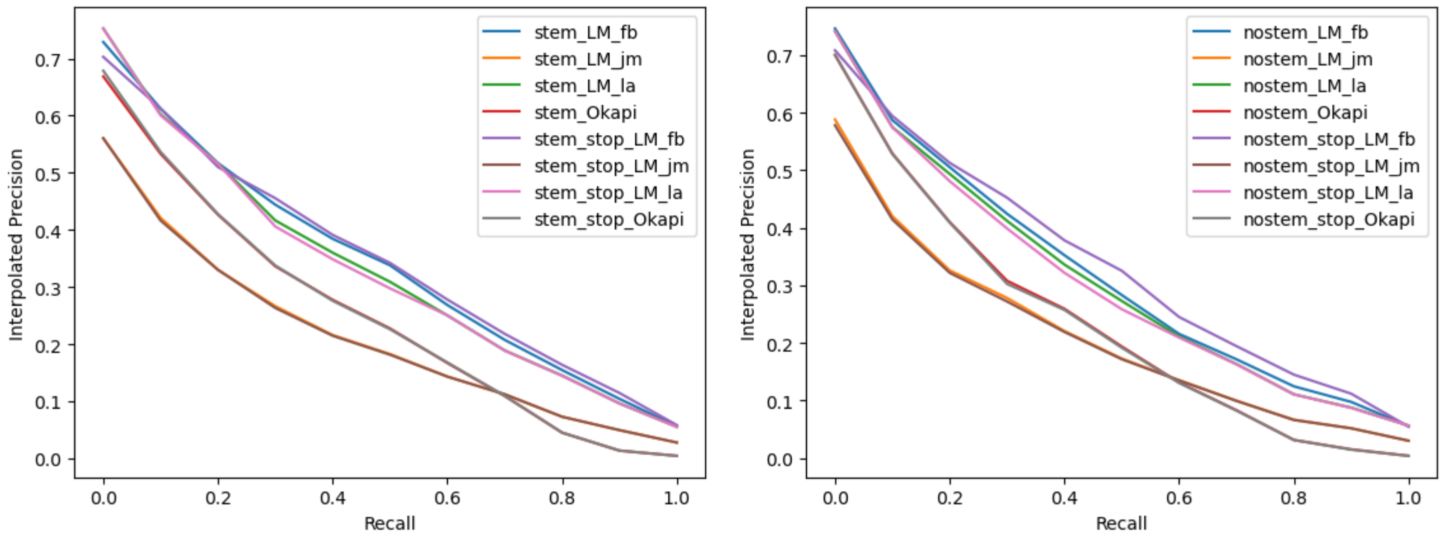

Figure 5: Interpolated Precision Curve – Compare the effect of Stopwords between each ranking function

From the line chart on the left, two ranking function with poorer performance (Okapi and LM_jm) seems to be a lack of significant differences. However, from my perspective, indexing with stopwords should help the results more precise, but in terms of LM_la shows removing stopwords is redundant and get bad results, where green line higher than pink one from recall 0.2~0.6. Then look at the picture on the right, roughly speaking, these ranking functions have a same result stated above. It's worth mentioning that the choice of whether to use stopword or not has a significant effect on the results of LM_fb without stemmer.

To sum up, the presence or absence of stopwords has little effect on the conclusion for poorer experiments. Besides, for LM_fb, the use of stopwords will have a higher precision rate especially the index without stemmer.

4. Conclusion

This experiment conduct the total 16 permutations of 4 indexing methods (stem, stem_stop, nostem, nostem_stop) and 4 ranking functions (Okapi, LM_la, LM_jm, LM_fb). In general, F-score is commonly used for evaluating information retrieval systems, so the ranking in this case based on F1 score.

Ranking	index	stopwords	ranking function	f1	MAP	P at 10 docs
1	Stem	included	LM_la	0.0739	0.3113	0.462
2	Stem	removed	LM_la	0.0738	0.3079	0.458
3	Stem	removed	LM_fb	0.0731	0.3288	0.49
4	Stem	included	LM_fb	0.0727	0.326	0.482
5	noStem	removed	LM_fb	0.0662	0.3154	0.47
.	.	.	.	.	.	.
.	.	.	.	.	.	.
13	noStem	included	LM_jm	0.0571	0.1924	0.302
14	noStem	removed	LM_jm	0.0569	0.1901	0.296
15	noStem	included	Okapi	0.056	0.398	0.217
16	noStem	removed	Okapi	0.056	0.396	0.2158

Table 2: Overview of each ranking functions (order by f1)

To sum up, using Porter Stemmer lead to a better results, and the impact of removing stopword is relatively inconsistent; the most directly related to the ranking is the choice of ranking functions.

Even if LM_la is ranked ahead of LM_fb, due to the slightly higher F1 score, however, the latter function had significantly better results in the MAP and Precision at top 10 documents (see the green label), so it’s difficult to determine which is better, but can speculated that adding pseudo feedback help the system retrieve documents more precisely.

Although Okapi function listed in the last place, it has the highest MAP value among all experiments (see the red label). To probe deeply, the f1 scores of those are not that behind.

Furthermore, there are many charts show that LM_jm is not a good method in the previous section, which can be proved by the low MAP scores (see the grey label). If exclude the LM_jm function, the performance of query generation (language model) is better than that of document generation (Okapi).

Code link: https://github.com/iamjudy/IR_experiment