Information retrieval course project

Team #16 Member:

Hang Shi

Xianlong Zhang

Xiaofei Huang

Course: Information Retrieval, String 2016 Semester

Instructor: **Nada Naji**

1. Instruction

For task1, we implement BM25, Lucene and tf-idf get the top 100 ranked lists. For task2, we use (1)Thesauri, (2)Dice coefficient for query expansion, for task3 we generate Precision & Recall tables for each query in every search engine, AvP & RR tables for each search engine and an MAP & MRR table for the entire project.

Hang Shi writes the Dice.py, which implements dice coefficient for query expansion; Snippet.py which generates snippet for every rank list of each search engine; and the Pesudo_RF_1half.py, which tried Rocchio's Algorithm for query expansion and returns a weighted vector for each modified query. He also implements Lucene for the 64 queries and get the files in '/Lucene_SE'.

In phase 1, the part of base BM25 search engine is designed by Xianlong Zhang, and the task 3 are finished by him as well. In phase 2, the code of evaluation are programmed by him, and results of evaluation will be automatically written into excels. Besides, he also process the raw documents and queries for both tasks.

The rest of works are done by Xiaofei Huang, including all the evaluation files, tf-idf as a retrieval model and Thesauri in query expansion.

2. literature and resources

In the process of building search engine, we imported two external libraries, which are nltk[1] and json. Nltk is convenient tool for natural language processing, it has build-in functions such as tokenization, frequency calculation, dictionary of thesaurus and so on. And in our project, json is used to export index and some other test data without damage their dictionary structure.

We look into CLAUDIO CARPINETO and GIOVANNI ROMANO's article: 'A Survey of Automatic Query Expansion in Information Retrieval' for the dice coefficient algorithm.

3. Implementation and Discussion

3.1. Tf_idf Search Engine:

In the project, we use the simplest way to apply the tf_idf model [2] to calculate score: summing the products of tf and idf for each query term. In the case of tf, we choose the logarithmically scaled frequency weighting scheme, which define that $tf(t,d) = 1 + \log f_{t,d}$, or zero if $f_{t,d}$ is zero. The idf component also use the logarithmically scaled: $idf(t,D) = \log(N/n_t)$. However, the evaluation result of this search engine is worse than the other two basic search engines. Tf_idf approach is based on the assumption that the terms, which occur frequently in

some documents, but exist seldom in else, are significant to distinguish documents. That means the weight of not only common words but also some specific field words about professional collection (like “technique” in “CACM” collection) will be reduced. So the simple tf_idf measure obviously cannot perform very well since it fails in reflecting the real degrees of importance and distributions of terms.

For tf_idf_stop and BM25_stop search engines, based on the tf_idf and BM25 basic search engines respectively, we filter the common words in both queries and index before searching query. Comparing with the basic SEs, the performance of these two SEs have improvement but not so much because “CACM” collection just contains not very long documents so that common words appear not so frequently.

To improve the effectiveness of our tf_idf SE, we can smooth both tf and idf, and do normalization to eliminate the effects of certain gross influences.

3.2. Query Expansion

3.2.1. BM25_QE_Thesauri SE:

We do query expansion based on the synonyms in a semantically oriented dictionary of English, WordNet[3]. WordNet goups English words into 117,659 sets of synonyms called synsets, so it is huge enough for “CACM” to expand query terms. A natural language processing tool for Python is NLTK, which provides some function to visit WordNet. For this SE, we initially filter the common words for the terms of query. Then by searching the WordNet we find out the most common semantic of term, which is represented by “sense”, and get the set of synonyms under this “sense”, which is represented by “lemma”. After that, calculating the BM25 scores of these “lemmas” separately instead of the original term by these “lemmas”, we compare the original term’s score with these “lemmas” scores multiplying by weight. Finally we regard the highest score as the ranking score.

There are two parameters affecting the performance of this SE. One is α, the number of lemmas of each term that are added into the expansion list. The other is β, the weight of each lemma (the degree of importance of the original term must be higher than that of lemma). After adjusting the values of α and β, we figure out that when α=3 and β=0.7, this search engine works much better than basic SEs, but the efficiency of the SE decreases.

Not in “CACM” case, but in most common case, query expansion based on thesaurus[4] sometimes may increase the precision by expanding noun, adj., adv., but decrease precision by expanding verb.. In further study, we can expand terms except for verb. words. We can classify the terms’ parts of speech before expanding. On the other hand, we can compute the semantic similarity of the original term and the lemma and choose the most similar one or two as the expansion words. In this way, the precision of SE will improve apparently.

3.2.2. Dice coefficient for query expansion

When using dice[4]coefficient for query expansion, we take each term in the query and

calculate the number of documents, named n_a, for where the query term A appears. And for each term in the entire corpus, we calculate the n_b for their occurrence. And n_{ab} which shows in how many documents term A and B both appears. And we can get a rank:

$$\text{Dice coefficient} = 2 * \text{nab}/(\text{na} + \text{nb})$$

We take the top K results of each query term for their expansion and add them to the expanded query. The performance of dice algorithm depends on the value of K, we have tried some values of K and get such result:

The value of K	MAP	MRR
2	0.106463	0.40625
3	0.109413	0.42188
4	0.105056	0.43750
6	0.103687	0.421875

So generally speaking, when K = 3, i.e. for each query term, add another two nearest terms to the query, we can get the best performance.

But we fund that dice coefficient in this project do not work as well as the non-expansion one, just slightly worse. This expansion doesn't work well. It's maybe because topic drift. Because dice take the co-occurrence as the only measurement of expansion, some terms are off-topic but happened to occur frequently, will be selected. So the topic may sometimes drift a little and make the result bad. This condition does not always happen, so the overall result just been worse a little. And if we add more terms, the drift becomes larger, making query results even worse.

3.2.3 (Not used but tried) Rocchio's Algoritm

This pseudo relevance feedback method is just tried but not chosen. Because I found not suitable for large corpus, run in small devices (like my poorly little Dell laptop). We need to bulid a vector for the whole vocabulary in the '/cacm', then dimension is pretty huge. And weight for each term need to be stored in a dictionary in our code, the combination of dictionary and list make things worse. Rocchio takes too long so we give it up.

3.3. Task 3

We adopted BM25 as the search engine since this algorithm would perform better than purely tf-idf implementation.

For (A), we used the common_words (list of stop words) to get rid of influence of stop words. In get_query() and build_index_stopping() blocks, any word appearing in common_words would be removed so that we neither calculate scores of these words in queries nor in corpus.

Query by query analysis:
We take query 1 for 7 different runs to analyze the different effectiveness, top 5 relevant documents for each search engine are as Table 1:

Table 1. Query by query analysis among different search engines

BM25	BM25_stop	BM25_Thesa	BM25_Dice	Tf_idf	Tf_idf_stop	Lucene
1.CACM-2319	1. CACM-2319	1. CACM-2319	1. CACM-1410	1.CACM-2319	1. CACM-2319	1. CACM-2629
2.CACM-1410	2. CACM-1410	2. CACM-1506	2. CACM-2317	2.CACM-1605	2. CACM-2380	2. CACM-2319
3.CACM-2379	3. CACM-1519	3. CACM-1410	3. CACM-2319	3.CACM-1591	3. CACM-1410	3. CACM-1410
4.CACM-1519	4. CACM-2380	4. CACM-2379	4. CACM-1844	4.CACM-1698	4. CACM-1680	4. CACM-2151
5.CACM-1605	5. CACM-1523	5. CACM-1519	5. CACM-2151	5.CACM-1410	5. CACM-2358	5. CACM-1236

We set BM25 search engine as standard to compare with other search engines. From the results we can find some differences:

a) CACM-2379 is not present in bm25_stop, tf_idf_stop and lucene for BM25 didn't deal with stop words, which may result into a higher score.
b) CACM-1519 is not in BM25_Dice, Tf_idf_stop and lucene but appears in BM25_stop and BM25_Thesa, which infers that for these three search engine, the effects are not performed very well, because BM25_stop and BM25_Thesauri should work best among these seven search engines.
c) CACM-1605 is not in BM25_stop, BM25_Thesa, BM25_Dice, Tf_idf_stop, Lucene. Actually, CACM-1605 still ranks high(6-10) in these search engines, but not among top 5, it's even more relevant than some documents rank higher than it, but the paragraph in this document is longer than others, especially after removing stop words in corpus, which can result into a less score.

For (B), according to the rank results, we found three queries interesting, which are "parallel algorithm", "parallel processor in inform retriev" and "perform evalu and model of comput system". In the first one, I checked the top 5 ranked documents, and I found three documents (2714, 2664, 2973) occurring in the second query as well, that's to say, these two queries are somewhat overlapping, or say, the topics are correlated. As for the third query above, the score of documents for it are rather high, because there are two stop words in this query, and without using list of stop words, there exists deviation among results. For example, the document 3070 ranks higher than 2984, however, the content in 2984 apparently more relevant to this query. The reason why document 3070 get a higher score is the frequent application of stop words "of" and "and".

In Phase 2, the evaluation program is mainly divided into three parts: getting documents we need, calculating precision and recall, and calculating precision @k, MAP, MRR. We evaluate our results by comparing this values, and then perfect our further study.

3.4. (extra credit) Snippet

The algorithm is written from scratch. We read the results of each search engine and extract the content of corresponding documents. Then each word in document is scanned, if it is in the query terms, we add and to the two sides of the term to let it bolded, which highlights the query terms in query doucments. And is also add to mark the whole

content.

4. Results

The results are in folders ‘XXX_SE_Eval’ and ‘Evaluation’, please consult to Readme.pdf for more details.

5. Conclusion and Outlook

The BM25 search engine using list of stop words should work best, since BM25 itself is an excellent algorithm to calculate score, and with removing stop words, most noises are eliminated. However, in this task(BM25 removing stop words), we were required to calculate the corpus without stemming, so if we used stemmed queries and corpus after removing stop words, the search engine will perform better.

For the query expansion, thesauri works best. We use the bulid-in library of thesauri in the package nltk, which is a maturing and well fixed. So the result is better than the non-expended queries. When use dice coefficient, sometimes topic drift happens, make things worse. And when trying Rocchio’s algorithm, we find the vocabulary size is very large. It needs a lot of CPU processing calculating weights for each term in the entire corpus, as well as the BM25 scores. So it’s not suitable for very large dataset, unless you have power processors.

An excellent search engine not only calculates the rank scores and present them to users, but gives an advanced user experience such as typos-detection. Detecting typos can somewhat avoid the presence of non-relevant documents, and gives a more accurate results, and functions like this will be incorporated in our future works.

Reference

[1] Accessing Text Corpora and Lexical Resources: http://www.nltk.org/book/ch02.html
[2] tf–idf: https://en.wikipedia.org/wiki/Tf%E2%80%93idf
[3] WordNet: https://en.wikipedia.org/wiki/WordNet
[4] CLAUDIO CARPINETO and GIOVANNI ROMANO.: A Survey of Automatic Query Expansion in Information Retrieval, Fondazione Ugo Bordoni, ACM Computing Surveys, Vol. 44, No. 1, Article 1 (2012)