homework3 report for 11791

andrewID: xiaoxul

October 23, 2014

1 System Design

1.1 DocumentAnnotator

The DocumentAnnotator does the following things in the function *createTermFreqVector*:

1. Tokenize the text in doc, using the naive basic white-space tokenizer.
2. Use HashMap to store every token, where key is the text itself and value is its frequency.
3. Put all the tokens in the Document to current jcas's Token type, text and frequency set.

1.2 RetrivalEvaluator

The RetrivalEvaluator does the building process of different data structures for different text tpyes(query with relevance = 99, confident document with relevance = 1, non-confident document with relevance = 0) and making use of these data structures to calculate statistical performance(cosine similarity and jaccard similarity, etc.)

Main **data structures** are as follows:

1. **QIDlist** is a **LinkedHashSet** to store all the qid in a ordered and non-repeated list.
2. **QIDquery, QIDdocu1, QIDdocu0** are three **HashMap** to store all the query and documents, with qid as their key. For each qid, QIDquery is to store its query, QIDdocu1 is to store its confident document. Both of their value in HashMap are maps of tokens. QIDdocu0, on the other hand, it to store the documents with relevance equal to 0 for each qid. As there will be several of them instead of only one, the value of the HashMap is **ArrayList** of the maps of tokens.
3. **QIDscore1, QIDscore0** are two **HashMap** to store all the scores of the document, with qid as their key. The same as before, value is different. For the non-confident document with relevance equal as 0, **ArrayList** is used to maintain all the scores.
4. **rank** is HashMap to store confident document's score. Key is qid and value is rank.
5. **text** is HashMap to store confident document's text. Key is qid and value is text. These two HashMaps is simple and strait forward as in the output part, only confident document will be written to the file.

Main **processes** are as following:

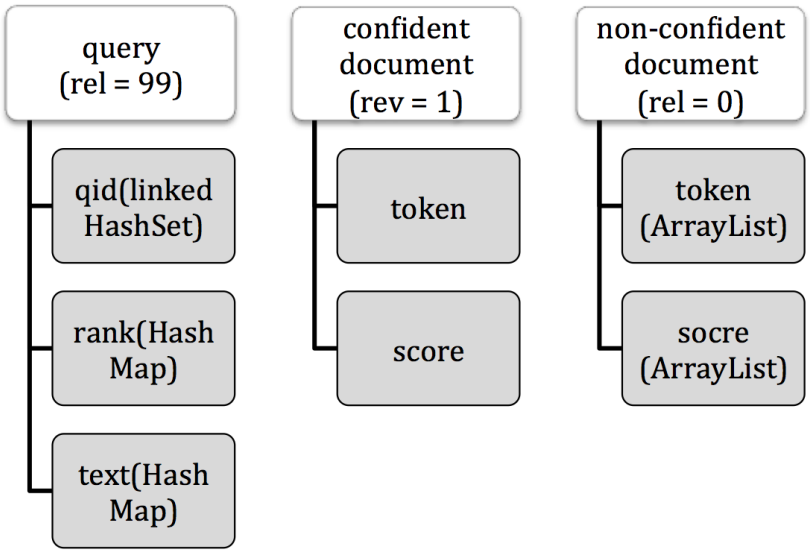
1. Build a **HashMap** which called as tokens as a **global dictionary**, putting every single token in text in it.
2. Put different doc for each qid to their fit HashMap, using relevance = 99 / 0 /1 to distinguish.
3. Compute all the statistical performance for each doc.
4. Get the rank for each confident document and compute mrr.

5. Write result into output file.

Main **statistical performance** are as following: (which will be explained in details in the next section)

1. Cosine similarity
2. Jaccard similarity
3. Dice similarity
4. Tf-Idf similarity

The whole design could be illustrated in the flowing diagram.



2 Performance Evaluation and Error Analysis

2.1 Performance Evaluation

The following is the score of each qid's confident document, based on calculation on four different similarity analysis formulations.

qid	Cosine	Jaccard	Dice	tfidf
1	0.2791	0.1111	0.0312	4.9747
2	0.2858	0.2000	0.0444	13.2095
3	0.2357	0.2222	0.1176	1.9617
4	0.2315	0.3750	0.1154	4.5122
5	0.0000	0.0000	0.0000	0.0000
6	0.5547	0.3846	0.2174	3.0650
7	0.0891	0.1429	0.0400	2.1972
8	0.1833	0.2857	0.0833	3.4692
9	0.5804	0.3333	0.1111	5.0580
10	0.5000	0.5000	0.2500	1.8800
11	0.1768	0.2500	0.0833	1.3863
12	0.3162	0.1429	0.0500	3.7436
13	0.1195	0.2000	0.0625	1.7047
14	0.4216	0.5714	0.2500	2.3511
15	0.0788	0.1429	0.0370	2.3026
16	0.2828	0.4000	0.1333	2.4079
17	0.1508	0.2500	0.0667	3.9849
18	0.2265	0.0000	0.0000	10.9634
19	0.1268	0.1000	0.0270	4.3190
20	0.3078	0.6000	0.1250	4.6744

The following is the rank of each qid's confident document, based on calculation on four different similarity analysis formulations, together with themselves' mrr.

qid	Cosine	Jaccard	Dice	tfidf
1	2	3	3	1
2	2	2	3	1
3	3	3	3	3
4	2	3	2	2
5	3	3	3	3
6	2	2	2	3
7	3	3	3	2
8	2	2	2	1
9	2	3	2	3
10	1	2	1	4
11	4	4	4	4
12	3	4	4	3
13	3	3	3	3
14	2	1	1	3
15	3	3	3	2
16	3	3	3	3
17	3	3	3	2
18	2	3	3	1
19	3	3	3	3
20	2	1	1	1
MRR	0.4375	0.4250	0.4583	0.5250

Cosine similarity, Jaccard similarity and dice similarity are a little bit similar here. The tfidf performs better yet not so beautiful. Why? Besides the fact that the data set is actually small and the tokenizer is so naive. The statistical measurement itself also reflects something. Cosine similarity only focuses on term frequency, while sometimes different words may have different weights on the result of information retrieval, and order of the terms also may have an effect. Jaccard similarity and dice similarity have drawbacks too, as both of them focus on the presence /absence of the term, without consideration about frequency. Tf-idf works better as it takes term's weight into consideration when it comes to valuing frequency.

2.2 Error Analysis

The poor statistical performance results from these following errors: Punctuations distraction, Case type mis match, Non-meaning words distraction, etc. Of course, small dataset and chosen of statistical tool have effect on this as well.

After eliminating all the punctuations and stop words(non-meaningful words), the mrr of cosine similarity get improvement from **0.4375** to **0.4458**, the mer of tfidf similarity get improvement from **0.5250** to **0.5292**. Then after lowercasing all the word left, the mrr of cosine similarity get improvement from **0.4458** to **0.4500**, the mer of tfidf similarity get improvement from **0.5292** to **0.5625**.

Though these changes are not so notable, it still shows that better tokenizer, better processing of the text would definitely improve the performance of information retrieval.

3 External Information

The similarity I used:
1. *cosine similarity*: $http://en.wikipedia.org/wiki/Cosine\_similarity$
2. *jaccard similarity*: $http://en.wikipedia.org/wiki/Jaccard\_index$
3. *dice similarity*: $http://en.wikipedia.org/wiki/S\%C3\%B8rensen\%E2\%80\%93Dice\_coefficient$
4. *tfidf similarity*: $http://en.wikipedia.org/wiki/Tf\%E2\%80\%93idf$

$http://pyevolve.sourceforge.net/wordpress/?p=1747$