Your Name: Mengyu Yang

Your Andrew ID: mengyuy

Homework 1

1. Collaboration and Originality

1. Did you receive help of any kind from anyone in developing your software for this assignment (Yes or No)? (It is not necessary to describe discussions with the instructor or TAs).

   No

2. Did you give help of any kind to anyone in developing their software for this assignment (Yes or No)?

   No

3. Are you the author of every line of source code submitted for this assignment (Yes or No)?

   No. I used the source code (QryEval.zip) provided on HW1 website as a framework and then developed other features (such as #AND #OR #NEAR operator) by myself.

4. Are you the author of every word of your report (Yes or No)?

   Yes.

2. Structured query set

2.1. Summary of query structuring strategies

All strategies I came up with aims to increase the precision while keep the process time and memory used at a reasonable range. The strategies I utilized includes:

a. Try to specify the field to search so that the query does not need to scan through the whole document. This strategy increases the precision and reduce the process cost.

b. Use #NEAR instead of only #AND and #OR, so that the matched document we get is more likely to be meaningful. A document that contains all the arguments may not the document we want if arguments are far from each other. #NEAR helps improve this problem since we can add constrains on the position of the arguments. This strategy help improve precision.

c. Reorder the arguments when necessary so that the argument with smaller result set is processed first. This may help reduce process time and reduce the memory used.

2.2. Structured queries

10:#NEAR/1(cheap.title internet.title)
This query utilizes strategy a and strategy b. The query searches documents that contain “cheap” and “internet” in title within one position. This query reduces process cost by specifying the search field so that the search is only carried on “title” field, which is much smaller than the entire document. By using #NEAR operator, we make sure that these two words are occurred as a phrase, so that we get more meaningful matched document.

12:#AND(djs.url)
This query utilizes strategy a. It specifies to query on ‘url’ field, which increases the output precision and reduces process time.

26:#NEAR/3(lower.body #NEAR/1(heart.body rate.body))
This query utilizes strategy a and b. It first searches for documents that contains the phrase “heart rate” in body and then select the documents that have “lower” before “heart rate” within a distance of 3. This structure query improves the precision a lot since it makes sure that “ heart rate” is a phrase and make sure “lower” is within a reasonable distance of “heart rate”. The output file has more probability to contain the useful information.

29:#AND(#NEAR/1(ps 2) games.inlink)
This query uses strategy a and b. It first searches for documents that contains the phrase “ps 2” in body and then select documents that also contains games in inline field. By specifying field, it improves the result precision and reduces the process cost. By using #NEAR operator, it makes sure that “ps 2” occurs as a phrase.

33:#NEAR/5(elliptical trainer)
This query uses strategy b. It searches for documents that contains elliptical and trainer within a distance of five. This improves the precision compared with using #OR and #AND because it forces the two words “elliptical” and “trainer” to be closed to each other, which produces more meaningful result than just having two words in random locations in a document.

52:#OR(avp.url)
This query uses strategy a. It specify to search “avp” on url field. This makes the search more efficient. Using “AND” or “OR” gives exactly the same precision since there is only one argument in this query.

71:#AND(india.keywords #NEAR/2(living in))
This query uses all the strategies listed above. The structured query first looks for “india” in keywords field and then searches for “living” and “in” within a distance of two. The order of “india” and “living in” is swapped because the result set of “india.keywords” is smaller than “#NEAR/2(living in)” given this index (it might be the other way around in other index though). The process cost can be reduced by swapping the query argument’s positions. This structure query improves precision a lot especially for the top results of RankedBoolean (P10 is up to 0.7).

102:#NEAR/1(fickle creek farm)
This query uses strategy a. The query finds documents that contains exactly “fickle creek farm”. This query is more reasonable than using only #OR or #AND since “fickle creek farm” is a phrase and it has meanings as a whole.

149:#OR(#NEAR/1(yellowstone national park) #AND(uplift at))
This query utilizes all the strategies a and b. First it searches “yellowstone national park” as a phrase, and then it looks for “uplift” and “at”. Since “at” is a stop word, “#AND(uplift at)”is the same as just searches for “uplift”. Finally #OR operator is applied on the results of these two parts. This query is good because it adds constrains on the relative ordered and positions on “yellowstone” “national” “park”, which is usually a general phrase. And it reduces process time by using #AND(uplift at), which has the same effect as deleting one argument.

190:#AND(#NEAR/2(brooks brothers) clearance)
Strategy b is used in this query. “brooks” and “brothers” within a distance of two is searched first and then #AND operator is applied on clearance and the result of #NEAR. Precision is improved by binding “brooks” and “brothers” together using #NEAR as “brooks brothers” has particular meaning as a whole.

3. Experimental results

3.1. Unranked Boolean

	BOW #OR	BOW #AND	Structured
P@10	0.01	0.04	0.2
P@20	0.005	0.02	0.205
P@30	0.0033	0.043	0.2133
MAP	0.001	0.0142	0.0803
Running Time	00:32	00:04	00:05

3.2. Ranked Boolean

	BOW #OR	BOW #AND	Structured
P@10	0.15	0.25	0.42
P@20	0.18	0.26	0.36
P@30	0.1667	0.2767	0.3133
MAP	0.0566	0.0980	0.1117
Running Time	00:33	00:04	00:05

4. Analysis of results

The three operators (#OR, #AND, #NEAR) can be compared in three different aspects: precision, recall and process time. #OR performs bad at precision since a document is matched once it contains any of the arguments at any locations. #AND has much higher precision as a document needs to satisfy all the arguments in a query. #NEAR/dist gives the best precision when we choose the right distance, since it not only defines the arguments needs to be searched, but also defines the relative order and relative locations of arguments. It helps a lot when we search general phrases, where #NEAR can help to make sure that the arguments are in right order and closed to each other. In this case, results given by #NEAR operator is more likely to be meaningful and useful to users.

On the other hand, #OR performs the best in terms of recall while #AND and #NEAR performs bad. The reason behind it is obvious: since #OR retrieves much more documents given the same arguments than #AND and #NEAR, it has less miss on relevant documents. In general, #NEAR behaves the worst in terms of recall since it put more constrains and requirements on retrieving documents.

As can be seen from the chart, the running time for #AND is much shorter than that of #OR. The reason involves the implementation of these two operators. In the implementation of #AND, I sort the argument list and put the shortest list as the first argument and use it to control the loop. This reduces the running time. However, in #OR operator I can't do this optimization as score list of every arguments should be scanned. This is why #OR is much slower than #AND. The running time of #NEAR is between that of #OR and #AND. In the implementation of #NEAR, it uses the same algorithm to find the document that contains all the arguments, and then it needs further check whether all the arguments are within the distance as required. Besides, the implement of #AND and #OR are based on score list while the implement of #NEAR is mainly based on inverted list.

There are two approaches to retrieve documents: ranked boolean and unranked boolean. From the chart above we can see that ranked boolean gives better precision no matter what the operator is. This is because in ranked boolean we rank documents by the term frequency. The larger the term frequency, the higher rank a document will get. By contrast, in unranked boolean, every document will get the same score 1.0 and the documents are sorted according to the document name. The rank of a document in unranked boolean tells us nothing about the relevance of the document. Thus in ranked boolean the document on the top is more likely to be relevant than in unranked boolean. When we observe P@N value, ranked boolean has more relevant document with a high ranking, thus the precision is much more than unranked boolean. The recall and running time for these two model is the same because the algorithm for retrieving documents is the same. The only difference part of implementing these two models is the way of calculating scores.

There are five fields: body, keywords, title, inline and url. Each field contains different information of the document. This gives us flexibility to search a specific field instead of the entire document. It reduces the number of matched document so that it decreases the running time and process cost. But the drawback is if we mapping the field wrongly in query we will get false negative and miss some relevant documents.

In the three experiments above, the first experiments use #OR operator on all the query. As expected, it gives bad precision, long running time, but good recall. This kind of operator is useful when the corpus is very small and users have more tolerance with false positive. The second experiment use #AND on all the arguments. The precision is better, use shorter time, but worse recall. This kind of approach to form a query is suitable for Internet users who have a bunch of documents and have less tolerance with false positive. Precision is very import in such cases. My structure queries combines three operators, use #NEAR to give more constraints on relative order and locations of arguments, and also specify fields of search. As expected, the precision improved a lot. This kind of approach is useful when we know the feature of the queries, but it’s hard to develop a general rule since every query has different situation.

During the process of construct queries, I used three strategies as mentioned in Section 2.1. It works as expected for almost all the queries. In general, #NEAR/n is very helpful to improve precision when the arguments of the query should be a phrase. However, for query 102 (fickle creek farm), I tried to use #NEAR, #OR, #AND, the combination of three operators, and specifying different field, but I didn’t find any approach that can improve the precision effectively. This may due to certain feature of this query and this particular index, for example, there are few documents that contains the phrase “fickle creak farm”, in which case no matter what operators I used, the relevant results can not be increased much. Other reasons may also possible for this query.

To summary, there is a tradeoff between precision and recall. The approach with good precision has bad recall, and the approach with great recall has poor performance on precision. #AND processes much faster than #OR. The ranked boolean model has better precision than unranked boolean, but the recall and running time is the same.