Tanay Gavankar
tgavanka
11-441 HW1 Report

Query Parsing Choices

The query parsing choices were relatively straightforward. All hyphenated words (i.e. “word-word”) were replaced with a structured query of #NEAR/1(word word). This seemed to be the logical choice as hyphenations usually indicate that the words should be next to one another, and so this was accomplished via the NEAR operator. All other punctuation was converted to spaces, and the query was space-delineated.

Structured Queries

The general strategy for generating structured categories was to try to understand what information need the original query was trying to meet, and update it accordingly. One word queries were more difficult to categorize, and so they were converted to a search of both the body and the title for the search term.

1:#AND(obama #NEAR/1(family tree))
This query seemed to be looking for Obama’s family tree, and so the user probably wanted the phrase “family tree” in their results.

2:#AND(french lick #OR(resort and casino))
French Lick is a popular resort/casino, but rather than matching all of the terms, we also want to return results that only include “French Lick Resort” or “French Lick Casino”.

6:#OR(kcs.body kcs.title)
Since this is a one-word query, we search both the title and the body.

7:#AND(#NEAR/5(air travel) information)
The user was looking for information related to air travel, but there may be additional words in between air and travel that we still want to match, so we relax the NEAR operator.

15:#OR(espn.body espn.title sports)
Matching sports in the title would result in too many results, so we only search espn in both the body and title.

16:#AND(arizona #OR(game and fish))
Game and fish are two different entities of the same category, so we care about matching either one.

17:#AND(poker.body poker.title tournaments)
We want to find an article that talks about poker tournaments in the body but also mentiones poker in the title.

18:#AND(wedding budget #OR(calculator.body calculator.title))
We are looking for a calculator, so we search both body and title for that. However, we want a specific wedding budget calculator, so we search body for those terms.

21:#OR(volvo.body volvo.title)
Since this is a one-word query, we search both the title and the body.

25:#OR(euclid.body euclid.title)
Since this is a one-word query, we search both the title and the body.

26:#AND(lower #NEAR/1(heart rate))
We want to treat “heart rate” as a phrase, so we match with the NEAR operator.

29:#AND(ps-2 games)
The hyphenation gets converted to #NEAR/1. It is kept as a hyphen to demonstrate that hyphenated words work correctly.

31:#OR(atari.body atari.title)
Since this is a one-word query, we search both the title and the body.

32:#AND(website #OR(design hosting))
We want to return results for both design and hosting websites, but they don’t necessarily have to be website design hosting.

36:#OR(gps.body gps.title)
Since this is a one-word query, we search both the title and the body..

37:#NEAR/1(pampered chef)
This is referring to the company Pampered Chef, so we want to use the NEAR operator to ensure the words are adjacent.

38:#AND(dogs for adoption)
We want to find dogs up for adoption, and the stopword for will get dropped.

39:#NEAR/1(disneyland hotel)
We are looking specifically for hotels in disneyland, and don’t care about articles just about disneyland or only hotels.

41:#AND(orange county convention center)
We want an article that simply has all of these terms. It would be rare to find an article that has all these terms and isn’t what the user is looking for.

42:#NEAR/1(the music man)
This seems to be a search for a specific entity titled “The Music Man”, so we make sure to use the NEAR operator.

46:#NEAR/1(alexian brothers hospital)
This is the name of a specific hospital, and so to find all articles about it, we use the NEAR operator.

54:#AND(president.title president.body of the #OR(#NEAR/1(united states) us))
We are looking for the POTUS, and so we check both title and body for president, drop stopwords, and check for either the phrase “united states” or “us” to make sure it is the correct president.

56:#AND(#NEAR/1(uss yorktown) #OR(charleston sc))
This seems to be a query for a ship (USS Yorktown) located in Charleston, SC. We want to match articles that mention the ship, but also just Charleston or SC.

59:#AND(how to build a fence)
After dropping stopwords, build and fence are left, so we simply do a fullbody search for both terms.

62:#OR(texas #NEAR/1(border patrol))
We care about articles about Texas or “border patrol”, and using AND here would reduce the number of results too much.

73:#OR(#NEAR/1(neil.title young.title) #NEAR/1(neil.body young.body))
Neil Young is a name, so we have to use the NEAR operator, but it can appear in either title or body, so we include both.

80:#AND(keyboard reviews)
We want to find results with both terms keyboard and reviews.

82:#OR(joints.body joints.title)
Since this is a one-word query, we search both the title and the body.

91:#AND(er #OR(tv show))
We are looking specifically for ER, the tv show. We don’t care if the result has TV or show, as long as it has ER and one or the other.

97:#NEAR/1(south africa)
This is the search for the location of “South Africa”, so we use the NEAR operator.

Experimental Results

	AND_BOW_RANKED	AND_BOW_UNRANKED	OR_BOW_RANKED	OR_BOW_UNRANKED	OR_STRUC_RANKED	OR_STRUC_UNRANKED
runtime (ms)	12254	12226	15801	15249	21648	21573
num_q	30	30	30	30	27	27
num_ret	2789	2789	3000	3000	2425	2425
num_rel	531	531	531	531	499	499
num_rel_ret	377	30	287	4	356	31
map	**0.3715**	**0.0422**	**0.1532**	**0.0001**	**0.3491**	**0.0141**
gm_ap	0.0266	0.0001	0.0013	0	0.0306	0.0001
R-prec	0.3623	0.0416	0.1681	0.0005	0.3641	0.0191
bpref	0.4501	0.068	0.2189	0.0021	0.4338	0.0459
recip_rank	0.5275	0.0737	0.2096	0.0016	0.495	0.0491
ircl_prn.0.00	0.5707	0.0742	0.2746	0.0021	0.58	0.0522
ircl_prn.0.10	0.5523	0.0523	0.2731	0	0.5686	0.0274
ircl_prn.0.20	0.5349	0.0516	0.2594	0	0.5543	0.0235
ircl_prn.0.30	0.5232	0.0516	0.201	0	0.5253	0.023
ircl_prn.0.40	0.4669	0.0486	0.1795	0	0.4292	0.0183
ircl_prn.0.50	0.4336	0.0486	0.1642	0	0.3967	0.0183
ircl_prn.0.60	0.3902	0.0348	0.1499	0	0.3527	0.003
ircl_prn.0.70	0.3098	0.0301	0.1351	0	0.2721	0.003
ircl_prn.0.80	0.2402	0.0301	0.1081	0	0.2294	0.003
ircl_prn.0.90	0.153	0.0301	0.0653	0	0.1227	0.003

ircl_prn.1.00	0.1114	0.0301	0.0415	0	0.0839	0.003
P5	0.3867	0.04	0.16	0	0.363	0.0074
P10	0.3767	0.04	0.1667	0	0.3481	0.0185
P15	0.3333	0.04	0.1644	0	0.3185	0.0173
P20	0.29	0.0317	0.1633	0	0.2833	0.0167
P30	0.2544	0.0244	0.1489	0	0.263	0.0136
P100	0.1257	0.01	0.0957	0.0013	0.1319	0.0115
P200	0.0628	0.005	0.0478	0.0007	0.0659	0.0057
P500	0.0251	0.002	0.0191	0.0003	0.0264	0.0023
P1000	0.0126	0.001	0.0096	0.0001	0.0132	0.0011

Analysis of Results

The three different approaches of forming queries are immensely different. The first method, combining a bag of words via the OR operator, is primarily used for high recall. This means that the query will match many different documents. However, these matched documents may or may not be relevant. The second approach was to combine a bag of words via the AND operator. This was used for high precision/relevance in the results. The benefit of this was that the results were much more relevant to the original query. However, it was possible that an extremely relevant document was discarded simply because it did not have one of the search terms. So, while the OR operator was often too loose in its specificity and AND was often too restrictive, it is difficult to find a middle-point. The third query method was to create structured queries that represented the query in such a way that it would hopefully match the original information need more closely.

Along with those three query formation methods, there were also two available retrieval methods: ranked and unranked. In unranked retrieval, if a document matched the criteria of the query, it was included in the result. However, in a ranked retrieval, documents that matched the query criteria were included in the results, but the document was also given a score. This score was determined by how well it matched the query. For example, a simple implementation of scoring for OR and AND was to count the number of times a specific search term appeared in the document. Then, when combining inverted lists, OR used the max of the scores while AND used the min. In the case of the NEAR operator, a point was given for every time the desired query string appeared in the corpus with the words being the specified distance apart. For example, if the query had #NEAR/1(white house), then every time the phrase “white house” in its entirety appeared in a specific document, a point would be added to the score of that document. By getting scores for documents, they could be sorted by their score, and so relevant results could be returned. We will now look at the three query operators in depth.

The first operator we will consider is the OR operator. This operator takes two or more arguments and return all documents that match any of its arguments. This is equivalent to the set operation of union. This operator is effective in that it does not need to have every term appear in a document in order to return it in the results. This way, the result set will be larger and thus result in a higher amount of recall. This is occasionally desired in cases, for example,

where the user may not be certain of the keywords they are using to search and so some words may or not may not appear in certain documents. However, this operator also comes with some drawbacks. The most common weakness of the OR operator is the low amount of precision it attains. Since the operator is matching documents so loosely, it may be that a specific document only matched one of the search terms, but it was still ranked very high because that one word appeared many times. However, that article may not be at all related to what the original query was searching for. For example, if a user searched for the term “apple computer”, and that phrase was combined via the OR operator, it could be that a document that mentioned the fruit apple many times would be ranked high, even if it never mentioned the word computer. This is backed up by the experimental data. The MAP for the ranked OR-combined bag-of-words queries was only 0.15, which is not very high. The ranked AND-combined bag-of-words queries had a MAP of more than twice as high at 0.37. When creating the structured queries, I had to be careful not to use it too liberally. Otherwise, the precision of the results would drastically be reduced. However, if I try to avoid using OR altogether, then the results would not be as widely distributed across various articles as desired. The OR operator was most useful to search in various fields. For example, if there was a one-word query, it was converted to an OR structured query that searched both the title and the body for that keyword. This allowed the search to occur in both fields, and so more results could be returned. One thing I noticed about the implementation of OR was that it was relatively memory-hungry. Since it required taking the union of two or more (possibly very large) sets to get the result, it became very easy to run out of heap space. Unfortunately, this is an unavoidable issue because the data must be stored in memory at some point. Luckily, it was not an obscene amount of extra memory. However, because it had to deal with more data and larger data structures, the OR operation seemed to run slower than the AND operation. From the experimental data, the bag-of-words queries that were OR’d together took around 15-16 seconds to parse all the queries, execute them, and write the results to disk total while the bag-of-words queries that were AND’d together only took 12-13 seconds to do the same thing.

The second operator we look at is the AND operator. It works similarly to the OR operator in that it takes two or more arguments, but instead of returning all documents that match any of the arguments, it only returns the documents that contain all of the arguments. This is analogous to a set intersection operation. This operator is quite powerful at enhancing result precision as it ensures that all requested query terms are present in the document. This means that if a document has a high score (i.e. all the terms exist in the document and they occur many times), chances that it is a relevant document. This is backed up by the experimental data, where the ranked AND-combined bag-of-word queries had a MAP of 0.37, which is the highest of all the experiments that were run. However, a weakness of the AND operator is that since it automatically discards any document that doesn’t contain all of the query terms, it may happen that it discards an extremely relevant document because it did not have one keyword. For example, if the search query was “obama family tree”, the relevant article that talked about Obama’s family may have been thrown out simply because it never mentioned the word “tree”. The AND operator was useful to create structured queries for strings where it was obvious that if all the terms were not present in a document, then the document would probably not be relevant. For example, the query “dogs for adoption” would result inaccurate results for anything but the AND operator. With OR, articles that simply talk about dogs would be in the result list as would articles about adoption in general. Since the information need was to find dogs for adoption, by using AND, we know that the resulting documents will contain both keywords. The AND operation is relatively fast since we can perform an intersection directly on the backing data structures’ sets, and thus the entire runtime for all the queries was about 12 seconds.

The third operator is the NEAR operator. This operator is slightly different in that when combining inverted lists, it acts as an AND operator. However, when determining the

score, it has the additional stipulation that the words must be at most k words apart (where k is an argument to the NEAR operator). This means that with the operator #NEAR/1(white house), every time the phrase “white house” appears in an article increases it’s relevance score by one. However, it differs from AND in that while AND would also match a document that has the phrase, for example, “white paint on the side of the house”, NEAR would not. This operator is the most inefficient of all the operators because of its nature; it has to not only perform the set intersection for viable documents, but it also has to then go into each word’s position list for each document and ensure that the positions for the words are valid distances apart. However, using this operator along with AND and OR in structured queries allowed for the best of all the worlds. Results were broad enough that relevant articles weren’t mistakenly discarded, but they were relevant enough that precision was still comparable to simply using AND. This is demonstrated by the experimental data of ranked structured data achieving a MAP of 0.35 (while AND had a MAP of 0.37). This operator was useful in constructing the structured queries because it allowed words that had to be together in order to make sense to be enumerated. For example, the query “south africa” would have been worthless as OR (or even AND) because then the results may have had documents about South America, etc. By using the NEAR operator, the results were guaranteed to have the phrase “south africa”, thus increasing the relevancy of the results. One failure of using the NEAR operator was that occasionally it would be too restrictive and so either not enough (or even no) results would be returned.

Software Implementation

This implementation of a boolean search engine is relatively straightforward. First, we shall consider the control flow of the program. The main method, found in SearchEngineRunner.java, starts with first reading the file with the queries. This file is parsed into a Map<Integer, String>, where the key is the query ID and the value is the actual query itself. After the queries have been parsed, they are looped through and operated on. Each query is treated independently, and so we can discuss the flow of a single query now.

The query is currently just a flat String, so we have to parse it into a tree representation. To do this, we call QueryParser.parse() on the query string. This method first cleans up the input query string by removing extra whitespace, replacing punctuation (i.e. hypens with NEAR, comma/semicolon/periods with space, etc) by using simple regexes. Note that periods are valid if they are surrounded by words on both sides, indicating that they are to be used as a field search marker. Those are not modified. A new Node object is then created with the default operator. The interface design pattern is used within the Node class so that different operators can be easily added in the future. Then, the cleaned query string is parsed into a query tree. If there is no operator on the top level, then the default operator (the root we created earlier) is retained. If there is, then we discard the original default operator root and simply return the top level operator. This parsing is done recursively, where the base case is a non-operator term (query string term) and the recursive step occurs on operators. Stopwords are removed on the parsing of base cases. After the tree has been constructed, it is returned to the main method.

The main method then takes the constructed query tree and calls QueryRunner.run on it. In this method, a depth first traversal of the tree occurs where at the leaf nodes, the corresponding inverted list is read from disk into an InvertedList object, which contains all the information about the inverted list. Then, for every inverted list, the score is determined (depending on if it is a ranked or unranked run) to be either the number of hits for that word in the document for ranked or simply 1 for unranked. This is inserted into a map that maps from docid to integer score. A map is used (as opposed to another data structure, such as a 2d list) because we need fast lookup and modification - total ordering only matters all the way at the end. At non-leaf nodes, the docIDs and scores for its children have already been calculated, so it combines them from a collection of maps (as mentioned above) into a single map by doing the appropriate combine method (depending on the type of operation), as implemented by the

combine() abstract method of the Operator class. For example, the OR operation performs set unions while the AND operation performs set intersections. The NEAR operator is treated as a special case and so only pulls documents from the inverted list if all the terms in the near operator are properly satisfied, which is done in a pairwise recursive fashion. This combining of scores keeps going up the tree until it finally reaches the root node, at which point the final map of docIDs to scores is calculated. However, this map is currently unordered, and before returning, the map is sorted via descending score and then ascending docIDs (done with a SortedSet given a custom comparator). This sorted data is inserted into a two dimensional array with the first element being the docID and the second element being the score. This two dimensional array is finally returned to the main method.

The main method takes this array and simply formats it to the specified text file format (i.e. trec_eval parsable). The array is sorted in the ranked order with the highest rank (best result) being the first element in the array.

The entire project was developed using Eclipse as an IDE, with its debugging tools and interface being invaluable. Git and Github were used for version control. No external Java libraries were used.

One good design decision for this implementation is that the operators use the interface pattern. This means that if we need to add more operators in the future, it will require only a small amount of additional work. Another good decision was that in many cases where I had to make a decision as to what data structure would back some object, I chose to use a map instead of a simple list. While a list may have been easier to implement and would maintain ordering, the cost of performing constant lookups and modifications would have ruined the program's efficiency. This is particularly noticeable in the implementation of the NEAR operator. I specifically kept the positional list for each document as a HashSet because I knew that the most common operation would be a contains() call (to ensure that a term is succeeded in a valid position by another term). If this had been implemented with a list, searching would have been much slower. Furthermore, the order of the entries did not matter because I would be iterating over all of the positional values anyway, so it using a HashSet was clearly a better design choice. This also ensured that the overall runtime for the program was fast. It was able to do 50 structured queries in ~20 seconds, which is fast for the hardware it is running on and the corpus it was running against.

A design decision that could have been improved was how it was inefficient at using memory in many cases. There are definitely duplicates of data kept in memory (such as the same data being stored in a list, then in a set, and then in a map). However, trying to optimize for memory would have made the code much more difficult to understand and follow, so memory efficiency was a tradeoff I had to make for understandability. One problem that the system encountered was that when running an entire set of queries in a single run, the JVM would get OutOfMemoryExceptions because the max heap size was not large enough. Luckily, this problem was alleviated by simply increasing the max heap size at runtime. (For specific numbers, the default heap size was originally 256mb, but the process would run well by simply bumping up to as little as 300mb. To be safe, the max heap size should be set to 512mb.)

Another fault that could have been improved on was the implementation of the NEAR operator. Right now, it is implemented as a special case. This means that extending the search engine with more operators similar to NEAR is not viable in its current state. A better design would have to be developed so that operators like NEAR can be implemented and extended easily.