CS 4300 Information Retrieval 11:59 PM Nov 3, 2014

Project 1

Alok Kumar Prusty (akp77)
Nikhil Anand Navali (nn259)
Prashanth Basappa (pb476)

1) Mean Average Precision for CACM collection: 0.2733
Mean Average Precision for Medlars collection: 0.4638

2) We proceeded with the following approach for the mini implementation of search engine:

- First, from a collection (CACM/Medlers) each document is considered and applied the Standard Tokenizer to perform stemming and stopping. Porter Stemmer already implemented in Lucene is used for Stemming. Stopwords are read from the file "*data/stopwords/stopwords_indri*" and copied into '*stopwords'* of type CharArraySet. The Standard Tokenizer in Lucene then uses the above '*stopwords*' list and Porter Stemmer to perform Stopping and Stemming. Lowercasefilter implemented in Lucene is also used to convert all the tokens to lowercase.
- Once the words in each document is tokenized, we then add it to a direct index which is following data structure:
  HashMap<String, HashMap<String, Integer>
  The *String* in outer *HashMap* holds the name of the document and the <*String*, *Integer*> in the inner *HashMap* holds the word, frequency count in that particular document. The above steps are followed for each document. Thus at the end of this step the above data structure will be holding the Term-Frequency count of each document in a collection. All these details are stored in "*directIndexMap*" which is the type discussed above. The following table shows the format of our direct index for first few documents of CACM collection:
  { manag=2, secur=2, 1=1, cryptograph=1, system=1, 1980=1, destroi=1, 5.6=1, 23=1, scheme=1, …….. }
  { access=3, januari=1, languag=1, rule=2, matrix=2, repeatedli=1, execut=2, measur=1, row=1, 31=1, ….. }
  and so on
- While calculating the score for each document, the document vector containing the term-freq count is got by HashMap<String, Integer> docVector = directIndexMap.get(document): document is the name of the document and docVector contains the term-freq vector.
- directIndexMap once calculated, is present in memory throughout the execution, since it is used for each query and each document for calculating tf*idf values.

3)

(a) atc.atc weighting:
Mean Average Precision for CACM collection: 0.3095
Mean Average Precision for Medlers collection: 0.4974

(b) atn.atn weighting:
Mean Average Precision for CACM collection: 0.3188
Mean Average Precision for Medlers collection: 0.4949

(c) ann.bpn weighting:
Mean Average Precision for CACM collection: 0.2909
Mean Average Precision for Medlers collection: 0.5002

(d) ltn.ltn weighting:[2]

l: sub linear tf	tf component is 1 + log tf
t: idf	idf component is log (N/n)
n: none	no normalization

Mean Average Precision for CACM collection: 0.3573
Mean Average Precision for Medlers collection: 0.4914

We have taken atn.atn and made slight variations to give more weightage to the term frequency. In augmented tf the max value a tf can take for any term is 1. But using a logarithm function, the tf value increases with increasing frequency. We just thought that giving higher importance to the term frequency like in BM25, we can actually improve the results and it is demonstrated for CACM collection. The MAP value is almost 0.05 higher than other tf*idf variants.

(4) BM25 Similarity Measure:

Mean Average Precision for CACM collection: 0.3742

Mean Average Precision for Medlers collection: 0.5202

(5) The possible problems in implementing methods 3a and 3b with an inverted index and dynamic collection are:

Unlike direct index, inverted index has the format <term – doc list> which is not only very slow to build but would require traversing whole index to get some statistics.

- To add one instance in inverted index, i.e. to get each <term-doc list>, an inverted index should visit the entire corpus. This makes it very slow to construct. But for a direct index, which is of format <doc – term freq count> , only that document is needed to be searched to come up with an instance.
- The calculation of tf and maxtf while calculating tf*idf values would not be efficient with an inverted index since we would have to traverse the whole inverted index to calculate these information for each document.
- In a dynamic collection, when the documents are added in real time, the whole index has to be visited to build the new index which might act as a bottleneck. Direct index would hardly take any time in adding a new document to the index.
- If a simple inverted index is used, then the position and frequency count of a term would be absent and in that case tf for 3rd question would not have been able to be calculated. Whereas maintaining a positional inverted index would be complex and time consuming.
- In the case of bottleneck the search then fails to provide the latest results as the index would still not be built and the new document details might have not been added to the collection.

Some of the approximations:

Few approximations can be made to allow the usage for simple inverted index efficiently:

- For efficient lookup, 2 level index can be used. A simple inverted index has just two parts: one for maintaining the terms and other maintaining the documents, offsets and frequency. Now on adding an extra level of to get to each term, a B+ tree can be used to quickly get to the term which is required. Even though this speeds up search, will still be slower than direct index.
- For dynamic collections, a supplementary index could be used which would be built as and when the new documents are added to the collection. This supplementary index would be smaller and periodically would be merged into the main index. But the problem that would arise here is that the program calculating tf*idf score then would have to search both the indexes.

  Combining such techniques inverted index could be used efficiently.

(6) After looking at the Mean Average Precision values from Question 3 and 4, we can say that BM25 similarity measure performs better than "atc.atc", "atn.atn" and "ann.bpn" in general for both CACM and Medlers collection. But this does not mean that BM25 similarity measure performs well over all queries. For example: BM25 similarity measure performs badly on the Query 9 of CACM collection but tf*idf variants perform better which is supported from the Fig 6.1

Let's compare the results from tf*idf vairants and BM25 similarity measure for Query 9 of CACM collection. Query 9 is *"Security considerations in local networks, network operating systems, and distributed systems"*. BM25 did not even fetch the document 3177 but other tf*idf vairants fetched it. To understand why, understanding of the BM25 formula is required.

BM25 is given by: *math.log* **((N - ni + 0.5) / ((ni+0.5)))** * **(((k1+1)*fi) / (K+fi))** * **(((k2+1)*qfi) / (k2+qfi))**

i ii iii

i – This term is related to the fact that how rarely a term occurs in the corpus.

ii – This term corresponds to frequency of term in document and iii – corresponds to frequency term of term in query.

Now coming back to document 3177 not being picked by BM25, there are two words i.e. "*system*" (occurs only once) and "*secur*"(occurs twice) that is matching the query terms and document terms. "system" occurs in 730 documents out of 3204 documents in CACM, so we can say that it does not occur rarely. This means that math.*log* **((N - ni + 0.5) / ((ni+0.5)))** is

very less. (1.22 in this case). But idf component of atc.atc gives (math.log(N/n)) 1.47 which is little higher than BM25. The idf component for the word "secur" (Occurs 30 times in 3204 collection) is also little higher in atc.atc. We can say that BM25 comparatively assigns less weightage for the rareness of the word.

Now the reasoning for document 2372 which is ranked lower than in tf*idf vairants:
The matching terms "secur" and "system" is repeated 6 and 7 times respectively. The component (ii) in BM25 is designed a such a way that, after 3 – 4 occurrences of term, additional occurrences will have little impact.[1] But the term frequency component in the tf*idf variants is (0.5 + 0.5 * (tf/max_tf in doc)) is directly proportional to the occurrences of word. So more the occurrences more the score. So in the tf*idf scoring scheme this document would be ranked better.

Now, let's consider a case where BM25 does much better than tf*idf variants.
BM25 similarity measure performs considerably well for Query 51 which is *"Algorithms for parallel computation, and especially comparisons between parallel and sequential algorithms*."
We will compare the (ii) term in BM25 and tf term in atn.atn: The max that tf can take in augmented tf used in tf*idf variants is 1 (i.e. when tf equals max_tf in doc). But in BM25, the tf increases as the number of occurrences increases. BM25 assigns score 1.69 and 1.57 for "*parallel*" and "*comput*" which is almost 1.5 times the weight atc.atc is assigning for the term frequency for these words. Thus, we can say that though BM25 gives more weightage for term frequency, it's designed such a way that it more term frequency have less impact. After few (3 or 4) occurrences of a term in the document, additional occurrences have less impact. So in actuality BM25 doesn't only check the similarity, it also checks the relevance. If a document just had repetitive matching words BM25 wouldn't linearly assign the weights to term frequency, instead it balances the weight.

Thus we can say few things about BM25 similarity from above observations:

- BM25 assigns relatively less value for the rareness of the word than atc.atc
- Average document length of CACM corpus is 51.44. Observing from Fig 6.1, the number of words in the documents of relevance, we can see that BM25 performs badly with the increasing length in documents.
- After few (3 or 4) occurrences of a term in the document, additional occurrences have less impact.

Supporting documents:
Query 9 – atn.atn performing better than BM25
Average precision for Query 9 using following variants:
BM25 = 0.1328 atn.atn = 0.3090 atc.atc = 0.3035 ann.bpn = 0.2909

atn.atn is the best performing among the tf*idf variants. So comparing it with it.

Relevant Documents for Query 9 in CACM collection	Position of relevant docs using BM_25	Position of relevant docs using ATN.ATN	Number of words per doc
CACM-2372	Position 23	Position 13	70+
CACM-2632	Position 43	Position 25	73+
CACM-2870	Position 44	Position 22	61+
CACM-2876	Not fetched	Not fetched	66+
CACM-3068	Position 13	Position 8	52+
CACM-3111	Position 11	Position 2	72+
CACM-3128	Position 66	Position 19	66+
CACM-3158	Position 4	Position 3	42+
CACM-3177	Not fetched	Position 69	52+

Fig 6.1

Direct index for 3177 Document - {ca791105=1, reveal=1, inform=1, half=1, data=1, 48=1, expos=1, olut=1, easili=1, complet=1, construct=1, shamir=1, interpol=1, techniqu=1, function=1, cryptographi=1, 1979=1, 11=1, remain=1, 5.39=1,

reconstruct=1, manag=2, **secur=2**, 1=1, cryptograph=1, **system=1**, 1980=1, destroi=1, 5.6=1, 23=1, scheme=1, januari=1, show=1, secret=1, novemb=1, kei=2, paper=1, share=1, divid=1, wai=1, enabl=1, breach=1, piec=5, robust=1, d=3, cacm=1, k=2, n=1, misfortun=1, knowledg=1, reliabl=1, db=1}

Direct index for 2372 Document - {explain=1, inform=3, 4.12=1, data=4, depend=1, featur=1, april=1, bank=1, 1972=1, morgan=1, four=1, function=1, implement=2, model=2, decis=2, propos=1, 1978=1, element=1, 3.50=1, item=1, conwai=1, 4.22=1, serv=1, distinguish=1, 59=1, gener=1, check=1, **secur=6**, manag=2, respect=1, repres=1, translat=2, exist=1, 1=1, **system=7**, jb=1, privaci=1, today'=1, ca720401=1, indic=1, access=3, januari=1, languag=1, rule=2, matrix=2, repeatedli=1, execut=2, measur=1, row=1, 31=1, independ=1, enabl=1, 3.73=1, set=1, cacm=1, column=1, perform=1, h=1, control=1, l=2, confidenti=1, r=1, framework=1, maxwel=1, w=2, 4.39=1, **oper=2**, time=3, user=1, pm=1}

Query 51 - BM25 performing better than atn.atn
Average precision for Query 9 using following variants:
BM25 = 0.5264 atn.atn = 0.2452 atc.atc = 0.2457 ann.bpn = 0.3097

Again comparing the positions of BM_25 and "atn_atn"

CACM Document	BM_25	ATN.ATN
CACM-0950	Position 3	Position 14
CACM-1601	Position 18	Position 35
CACM-1795	Position 34	Position 54
CACM-1811	Position 15	Position 57
CACM-2266	Position 11	Position 55
CACM-2289	Position 9	Position 2
CACM-2557	Position 24	Position 50
CACM-2664	Position 12	Position 25
CACM-2714	Position 10	Position 67
CACM-2973	Position 1	Position 4
CACM-3075	Position 16	Position 40
CACM-3156	Position 14	Position 13

Fig 6.2

Direct index for 0950 document - {standard=1, 09=1, analysi=1, consist=1, entir=1, ordinari=2, decemb=1, near=1, replac=1, propos=1, 1978=1, order=1, **algorithm=2**, proposit=1, method=3, real=1, integr=2, take=1, 3=1, recast=1, futur=1, jb=1, 9=1, full=1, result=1, **comput=3**, differenti=2, paper=1, **parallel=4**, ca641221=1, nievergelt=1, advantag=1, meant=1, avail=1, process=1, cacm=1, j=1, subtask=2, dedic=1, march=1, highli=1, expect=1, 1964=1, knowledg=1, form=1, serial=2, numer=2, equat=2, exampl=1, time=1, pm=1}

(7) From comparing MAP values from parts 1, 3, 4 we observed that the BM25 model works best in our system.

- In the Question 1, default Lucene logic was implemented on the collection without stopping and stemming. As expected, this performs badly since the stop words are not that useful while fetching the results and these has to be removed. Stemming is also important which groups similar words before building an index and for query matching. If the query has a word "computer" and the document had the word "computer". Though these words are similar and important in document score calculation, without stemming, these would not have matched.
- For Question 3 we used, stopping and stemming and used 3 variants to fetch the results from the collection. The first observation for the reason that this did better was due to the use of Stopping and Stemming.

  atn.atn / atc.atc – Augmented term frequency is used here which performs better than Lucene's tf calculation (sqrt(tf)). Lucene model doesn't have an upper bound for the term frequency.

  ann.bpn – This performed relatively poor due to the fact that ann.bpn just uses the binary 0-1 to calculate term frequency. In actuality if the term occurs more than once, it means that it is important and that needs to be accounted. ann.bpn does not consider even the idf component which is very important since if there is a rare term especially in such technical and medical collections it needs to be given some weight to account for its importance.

So from Question 3 we made following observations:

- Stemming and stopping is important and indeed improves the performance.
- Using logarithmic (Question 3.d) value for term frequency is better than having an upper bound for term frequency and not using any bound. That is balanced approach between upper bound of tf*idf variants and sqrt(tf) of Lucene.
- idf (inverted document frequency) should also be used to account for the rareness of the word.

- BM25 outperformed all the tf*idf variants due to many reasons which are listed below:
  BM25 is designed based on probabilistic models but tf-idf works in a vector space model. There are number of things which BM25 introduces which indeed makes it a better similarity measure than tf*idf measures. BM25 takes into consideration that longer documents has more words, thus words might have higher frequency. This might inversely affect the score each document is accumulating. A document might just be lengthier with few matching words relevant to query thus assigning this document a higher score. But just having few matching words with the query in no way assures of document being relevant to the query. So this is regulated by the parameter b, the document length dl and the average document length avdl.
- The other reason which we observed BM25 is doing better was the design it is using to calculate the term frequency. As explained in Question 6, in BM25 after few (3 or 4) occurrences of a term in the document, additional occurrences have less impact.
- BM25 also perform better on shorter documents. Average length of document in CACM collection is 51.44 and 101.11 in Medler collection. So we saw a significant increase in MAP value for CACM collection (lesser document length) but not that much for Medler collection.
- We also observed that BM25 performed bad for some queries, so we can't say that BM25 is the undisputed winner, rather it depends on many parameters such as the collection we are running on, how rarely important words appear in the collection and so on.

(8) Alok – Most of the JAVA coding was handled by him and played a major role in cleaning and modularizing the code.
Nikhil – Some part of the coding, variant for question 3.d, failure analysis, reasoning and documentation done by him.
Prashanth - Worked on Question 1, code for stemming and stopping and Question 7 done by him.

[1] – The BM25 Ranking Algorithm – Search Engines and Information retrieval - Croft
[2] - http://nlp.stanford.edu/IR-book/html/htmledition/sublinear-tf-scaling-1.html
[3] – ngram/2L approximation - http://infolab.dgist.ac.kr/~mskim/papers/CSSE07.pdf