CS 6200 : Information Retrieval

Spring 2019

Final Project Report

Professor : Rukmini Vijaykumar

Prepared By :

Sameer Desai

Matthew Piorko

Sushmita Chaudhary

1. Introduction

We have designed, implemented, evaluated, and compared our own Information Retrieval systems and their performance in terms of their retrieval effectiveness. CACM dataset is used as input.
The three project phases are:

Phase 1: Indexing and Retrieval:
The following retrieval systems constitute the first four *Baseline runs*. Each return the top 100 results for each query.

- *Tf/idf*
- *Query Likelihood Model (JM Smoothed)*
- *BM25*
- *Lucene (from HW4 of CS6200 course)*

Two more models were built based on query expansion techniques. These constitute two query expansion runs. The pseudo relevance feedback was inspired by HW3 of this course, however we reimplemented the code in python. These were:

- *Query time stemming*
- *Pseudo relevance feedback*

Performed four runs on the BM25 and Lucene baseline runs . Two of these used Stopping (removing stop words) and two used Stemmed version of corpus and query. Since , the relevance information was available , it was used in one of the runs.

Phase 2: Displaying results:
S*nippet generation* and *query term highlighting* implemented for Lucene baseline.

Phase 3: Evaluation:
To assess the performance of the different retrieval systems, the effectiveness is judged based on the following techniques:

- MAP - Mean Average Precision
- MRR - Mean Reciprocal Rank
- P@K, K=5 and 20
- Precision & Recall

Phase 4: Extra Credit:
Implemented spelling correction engine (using Levenshtein/edit distance for corrections).

2. Effort Division

Design and Blueprint	Everyone
Phase 1 (Task 1) : Build your own Retrieval System	Sushmita
Phase1 (Task 2) : Query expansion	Matthew - Query Expansion + Relevance feedback, query time stemming
Phase1 (Task 3) : Retrieval with variations	Sameer - 2 runs on baseline + two query expansion techniques
Phase 2 Displaying Results	Sameer - Snippet Generation
Phase 3 Evaluation	Matthew - MAP/MRR Sushmita - P@K, Precision, Recall
Extra Credit	Matthew
Integration and Documentation	Everyone

3. Literature and Resources:

- Lucene: The program uses Lucene 4.7.2 to retrieve the top 100 documents for a given query. Standard analyzer is integrated for tokenizing the corpus files.
- TF-IDF: Idea is to keep track of frequency of term in each document (TF) and document frequency (IDF) across corpus.

$$\mathrm{tfidf}_{i,j} = \mathrm{tf}_{i,j} \times \log\left(\frac{N}{\mathrm{df}_i}\right)$$

$\mathrm{tf}_{i,j}$ = total number of occurences of i in j
df_i = total number of documents (speeches) containing i
N = total number of documents (speeches)

- Smoothed Query Likelihood: The formula for QLM with JM smoothing:

$$p(qi|D) = \frac{(1-\lambda) fqi\ D}{|D|} \div + \frac{\lambda cqi}{C}$$

$p(qi\|D)$	: score
λ	:smoothing parameter which is set to 0.35
$\|D\|$	: document length
$fqi\ D$	: term frequency
C	: Corpus length

- BM25: BM25's formula is an extension of binary independence model. It includes the document and query term weights to provide more accurate results for scoring and retrieval.

$$\sum_{i \in Q} \log \frac{(r_i + 0.5)/(R - r_i + 0.5)}{(n_i - r_i + 0.5)/(N - n_i - R + r_i + 0.5)} \cdot \frac{(k_1 + 1) f_i}{K + f_i} \cdot \frac{(k_2 + 1) q f_i}{k_2 + q f_i}$$

where:
r_i = no. of relevant documents containing the query term
R = no. of relevant documents for that query
N = total no. of documents in the collection
qf_i = frequency of the query term in the query
K = k1(b*L+(1-b))
f = term frequency in that document
n_i = total number of documents in which the term appears
L = document length / average document length

- Pseudo Relevance Feedback: Used the traditional method for pseudo relevance feedback namely, doing an initial search to collect more keywords, and repeating the search with those keywords. We used *k* = 7 and *n*= 10, and this seemed to function well.
- Query Time Stemming: Performed stemming by taking possible stems for a word and appending them to the query before searching. We included all possible stems rather than taking only a few, since we don't have enough information to limit the size of the query.
- Snippet Generation : Used Luhn's approach to generate snippet for *Lucene Baseline Run*. The snippet for each query is stored in a seperate text file with query terms highlighted.
  Reference :https://urlzs.com/ygZE , https://urlzs.com/ELWJ
- Mean Average Precision: The results are scored according to the following formula:

$$\text{MAP} = \frac{\sum_{q=1}^{Q} \text{AveP(q)}}{Q}$$

  Where *AveP(q)* is the average precision of the query *q*. The precision value at every relevant document is summed up and averaged over all the relevant documents for that query. Those averages are then averaged overall to reach a mean average precision.
- Mean Reciprocal Rank (MRR): The results are scored according to the following formula:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}.$$

  Where *rank(i)* is computed as the rank of the *i-th* relevant document. These reciprocals are summed and averaged over all the queries to get the mean reciprocal rank.
- Precision and Recall:

$$Precision = \frac{|Relevant \cap Retrieved|}{|Retrieved|}$$

$$Recall = \frac{|Relevant \cap Retrieved|}{|Relevant|}$$

- Precision at K =5 and K=20:
  Tests the performance of the search system by obtaining the retrieved number of relevant documents at higher ranks than K. Higher precision values implies greater efficiency of the retrieval system.
- Levenshtein Edit Distance: The following formula was used to calculate the Levenshtein distance between two strings.

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

  Where the final term tests if the character at positions i and j in the two strings are the same or not.
- Additionally, many parts of our code used these common libraries:

  1) Lucene : lucene-core-4.7.2.jar
  lucene-queryparser-4.7.2.jar
  lucene-analyzers-common-4.7.2.jar
  2) NLTK - Natural Language toolkit for tokenization
  3) BeautifulSoup - Parsing HTML documents

4. Implementation and Discussion

Phase 1:

- As a part of preprocessing step , we generated
  i. Clean corpus
  ii. Clean query (stop words removed using Beautful Soup)
  iii. Inverted Index
  iv. Term Count in document
- Task 1: CACM html files were parsed, cleaned, tokenized and generated as text files using Parser.py. Indexer.py file opens each file in clean corpus and generates a unigram index (*'unigram_index.txt'*), keeping count of frequency of words. 'task1.py' compartmentalizes functions for each retrieval system (TFIDF, QLM and BM25). The process flow redirects to method for the particular user selected retrieval system. For Lucene, code from HW4 is used to generate index and print top 100 results.
- Task 2: Reused most of the code from task 1 to minimize computations. The index was reloaded into memory, and the tf-idf retrieval model was used as a baseline. In the case of pseudo relevance feedback, the search was re-run with *k* more keywords added based on analysis of the *n* top results from the search. These keywords are then added, and the search is done again. For query time stemming we used *PorterStemmer* used to get more stem words which are added to the query before searching.
- Task 3 : The outputs from preprocessing were used .For the first run we removed the stop words from corpus and query by and for the second run we used stemmed version of the corpus and query provided to us.

Phase 2:

- Displaying Results :
  We used Luhn's approach of text summarization to generate snippets. Sentences in each document are ranked using a significant factor(SF). A word is a significant word if it's not a stop word (which eliminates high freq words) and its frequency is
  SF is calculated by number of significant words occurring in the sentence.

$$f_{d,w} \geq \begin{cases} 7 - 0.1 \times (25 - s_d), & \text{if } s_d < 25 \\ 7, & \text{if } 25 \leq s_d \leq 40 \\ 7 + 0.1 \times (s_d - 40), & \text{otherwise,} \end{cases}$$

  where s_d is the number of sentences in the document.
  Due to the nature of corpus, we have chosen to subtract from 3 instead of 7. This was outcome of empirical process, since each document in the corpus has less than 10 sentences.
  SF is calculated by squaring the number of significant words and divided by the total number of words. We ranked the sentences in descending order based on the SF and chose top two sentences to generate the snippet. Since average number of sentences in each document of the corpus is 5, the snippet should cover atleast 40% of the words, hence we have chosen two sentences.
  Partial sentences were generated for these two sentences by using first 18 words then joining them with a delimiter(...) to generate the snippet. The 18 words were based on the fact that Google produces snippets with average number of 175 characters. (Ref : https://urlzs.com/eP4f).
  Query Highlighting: The query words appearing in the snippet is sandwiched between this symbol ***.

Phase 3:

- For the ninth run, we used BM25 (as opposed to tf-idf used earlier) with pseudo-relevance feedback. We used very similar logic to the task in phase 2, except swapping out the retrieval model of tf-idf with BM25.
- MAP and MRR are implemented as discussed in class. Every result file is looked at, examined for where the relevant documents appear, and the scores are calculated using the formula listed earlier. All nine result files are evaluated in this manner.
- P@K, K=5 and 20 and Precision & Recall: *metrics.py* consists of code to calculate relevance of documents. If relevant, the count is incremented in the *relevance* variable and count for all files is incremented in the *retrieved* variable. This is then used to calculate the precision, recall and P@K for each query.

Extra Credit:

- In order to find suitable replacements, the Levenshtein distance between the word and all words in the inverted index are considered. To ease the number of computations, we make the assumption that there is unlikely an error in the first character, so the list of potential replacements is reduced to words that start with the same letter. Additionally, modifications were made to the Levenshtein distance algorithm to stop searching if the distance was already greater than 2. We did this because it shouldn't matter if the distance is 10 or 100; if it's larger than 2, it is already too far apart to be a good replacement. The corrections are ranked by edit distance and limited to only six corrections. These modifications made the code run fast enough to compute every correction for every word in the queries. In order to showcase the case, the corrected queries are in a separate file, where [*word1, word2, ...]* indicates possible spelling corrections in order of confidence, and [*?*] indicates no spelling corrections found. These formats would not be included in the actual queries should they be executed (instead, the most likely correction would be chosen and no correction words would be removed), but for sake of example, they are included in the output file. Most words are already in the index, and so have not been corrected.

5. Results:

Phase 1

1. Task 1:
   In the Task1/Outputs path, the top 100 results for each of the 64 queries for the four baselines:
   - TFID.txt
   - QLM.txt
   - BM25.txt
   - Lucene_scores.txt
2. Task 2:
   Under task2/, the top 100 results for each query expansion technique are in:
   - pseudo_relevance_feedback.txt
   - query_time_stemming.txt
3. Task 3:
   Inside folder Run1:
   BM25_Score.txt - Top 100 retrieved documents (BM25) ,
   Lucene_Results.txt - Top 100 retrieved documents (Lucene)
   with ***stop words*** removed from both query and corpus.
   Inside folder Run2:
   BM25_Scores_Stem.txt - Top 100 retrieved documents (BM25).
   Lucene_Results_Stem.txt - Top 100 retrieved documents(Lucene)
   for ***stemmed*** corpus and query.
   Query Analysis : Analysed the following queries for four runs.
   Query1 : portabl oper system vs portable operating systems
   BM25 : Even though the top two ranked documents are same (CACM-3127,CACM-2246), the stemmed version has a low score compared to the stopped version .The lowest ranked document for non stemmed version makes no appearance in the top 20 ranks of stemmed query retrieval.
   In CACM-3127 (Non Stemmed) the term portable appears only twice where are in its stemmed counterpart the word ‘portabl’ appeared four times.Another observation from this comparison is only six documents from top 20 ranked documents were common to both .The only possible explanation for this divergence is stemming might have high impact on query terms.
   Lucene : The top ranked document for both version is CACM-3127 with score difference of 0.08.The lowest ranked documents . The lowest rank document do appear in top 20 ranks for either of the version. Also only documents overlap which reiterates our obervation about query stemming.
   Query2: parallel algorithm vs parallel algorithms
   We found this interesting because we wanted to analyse the effect of using singular nouns instead of plural nouns
   BM25: The top ranked document is CACM-2714(stemmed) vs CACM-3075.In the stemmed version word parallel appeared 7 times and algorithm appeared 5 times.This gives us clear indication why BM25 ranked this document high.Another observation was that the document CACM-3075 which was ranked 19th when retrieved with stemming.The stememd version of this corpus had only 3 occurrences of parallel and 4 occurrences of algorithm respectively. Also CACM-2714 had frequent occurrences of query term ‘ parallel’. We observed that a slight change in the query almost affected ranking of 50 percent of documents retrieved.The document CACM-2685 was ranked same in both the versions and score was also identical.
   Lucene : The top ranked document is CACM-2973(stemmed) and CACM-3075(unstemmed).However the document CACM-0950 was ranked 2 when retrieved with both versions with identical ranks.Stemming affected ranking of 60 percent of document.

It’s a good finding that both models agree on document CACM-3075 when stemmed.
Query3: *appli stochast process vs Applied stochastic processes*
BM25: The top ranked documents were CACM-1696(stemmed) vs CACM-0020.This query was interesting because all the quey words were stemmed.Only eight documents overlap with top ranked documents retained in top with top 20 with drop in score. This was obvious because of effect of stemming the query.
Lucene: The top ranked documents were again pleasantly agreed with the findings from BM25 model above.The document CACM-1696(stemmed) was ranked highest vs Q0 CACM-1410 . One surprising finding was none of the documents ranked after ten overlap with each other. This shows how stemming affects retrieval.In contrast most of the documents in top 10 ranks overlap with drop in rankings.The top ranked document Q0 CACM-1410(unstemmed) dropped to rank 2 when stemmed whereas the top ranked document CACM-1696 was ranked at 7 before stemming.

Phase 2 : The folder Snippets has all snippets generated for each query.Each file has all snippets for the top 100 documents retrieved for *Lucene* baseline run.

Phase 3:
- The results are under pseudo_relevance_feedback_with_bm25.txt.
- MAP (Mean Average Precision) are under map.txt.
- MRR (Mean Reciprocal Rank) are under mrr.txt.
- P@K, K=5 and 20 are in the Metrics Folder, under the respective run.
- Precision and Recall are in the Metrics Folder, under the respective run.

Extra Credit:
- Results are in: extra_credit/corrected_queries.txt.
- Please refer to section: **Implementation and Discussion** for more details.

6. Conclusions and Outlook

- According to our MAP and MRR results, the stemmed Lucene is the clear winner out of all retrieval systems. Interestingly, it uses Lucene as a baseline thus showing that Lucene is indeed a powerful information retrieval engine. Some models like BM25, BM25 (pseudo-relevance feedback), and BM25 (stemming) almost perform to the same level, however do not reach the same efficiency as Lucene. This shows that BM25 is likely the strongest model that were implemented, and is not really rivaled by any of the other engines..
- *Snippet generation* can be improved by incorporating the query terms to decide the significant factor, place special emphasis to words appearing in title of the document .
- Use machine learning frameworks like *TensorFlow* to generate relevant snippets
- Using ML algorithms for relevance-feedback.
- Perform better query expansion using *clickthrough data* from query logs.

7. BIBLIOGRAPHY:

Beautiful Soup: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
Search Engines by Croft, Metzler and Strohman: http://ciir.cs.umass.edu/downloads/SEIRiP.pdf
Snippet Generation :
https://nlp.stanford.edu/IR-book/html/htmledition/results-snippets-1.html
https://urlzs.com/ELWJ/
https://urlzs.com/eP4f
Python Programming :
https://urlzs.com/Ptwe
https://urlzs.com/1geX
https://docs.python.org/2/library/string.html
https://urlzs.com/nxvz
Zhai, Chengxiang, and John Lafferty. www.iro.umontreal.ca/~nie/IFT6255/zhai-lafferty.pdf
Okapi BM25: a Non-Binary Model,
nlp.stanford.edu/IR-book/html/htmledition/okapi-bm25-a-non-binary-model-1.html
Using Query Likelihood Language Models in IR,
nlp.stanford.edu/IR-book/html/htmledition/using-query-likelihood-language-models-in-ir-1.html
cdn-images-1.medium.com/max/800/1*mu6G-cBmWlENS4pWHEnGcg@2x.jpeg.