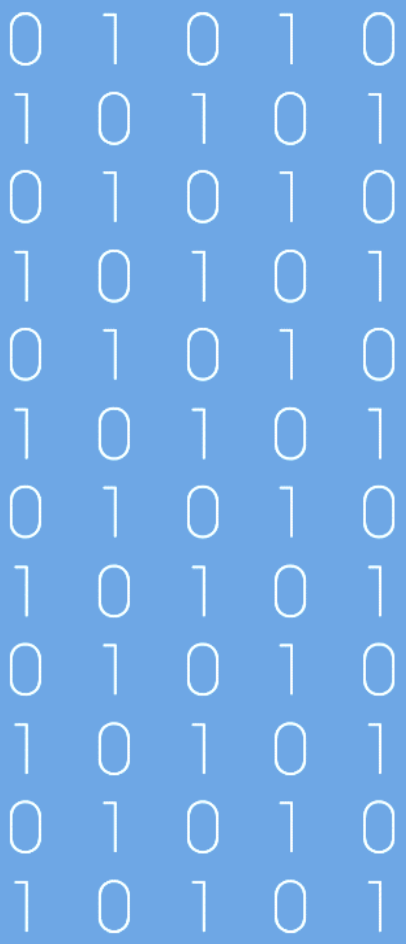
INFORMATION RETRIEVAL

PROJECT

FALL'16

Dr. Nada Naji



Aly Akhtar

Lalit Pathak

Tushar Gupta

Table of Contents

INTRODUCTION

This project provides a very brief and scope-limited overview of Search Engines. The goal of the project is to design and build our information retrieval systems, evaluate and compare their performance levels in terms of retrieval effectiveness. For this project, we have used the CACM test-collection. In the first phase, we begin with Indexing the corpus, and performing four Retrieval runs (BM25, Cosine, tf-idf, and Lucene). Then, we perform query expansion on the BM25 model, using Rocchio's Algorithm. The final part of Phase-1 involves performing stopping and running the model with the stemmed index and queries.

In the second phase, we first combine query expansion with stopping to perform a *seventh run,* and then evaluate and assess the performance of our search engines. To evaluate, we use Mean Average precision, Mean Reciprocal Ratio, Precision at Rank and, Recall and Precision. While Precision and Recall are query-level metrics, Mean Average Precision and Mean Reciprocal Rank give us an idea about the whole system.

Next up, we performed the "Extra Credit" task, which involved snippet generation and query term highlighting within the retrieved results.

Finally, we wrote this documentation, which includes a detailed analysis and report of the project in about 2,500 words. It covers expansively, the techniques we used, the steps we followed, the challenges we overcame and the final conclusion of the project.

Contributions of team members to this project:

1. **Aly Akhtar**: Performed Phase-1: Task-2 and Extra Credit task
2. **Lalit Pathak**: Performed Phase-1: Task-1 and the second part of Phase 2(Evaluation)
3. **Tushar Gupta**: Performed Phase-1: Task-3 and the first part of Phase 2(Expansion with Stopping)

Contribution to the documentation was equal on all our parts. We wrote our respective tasks to the documentation and then went over it together.

LITERATURE AND RESOURCES

The following literature and resources were referred to, during the course of the project-

- We used Rocchio's Algorithm for Query Expansion. It is an algorithm for Pseudo-Relevance Feedback. It is mentioned in *Search Engine: Information Retrieval in Practice, Croft et al.*
- We used BM25 Algorithm as our choice for Retrieval Model, as it is better for long queries, as mentioned here- http://www.ercim.eu/publication/ws-proceedings/DelNoe02/hiemstra.pdf
- We used the following third-party libraries in Python for this project-
    - NLTK
    - BeautifulSoup
- We studied about the Rocchio's Algorithm from https://en.wikipedia.org/wiki/Rocchio_algorithm

IMPLEMENTATION AND DISCUSSION

Phase 1: Introduction and Initial Setup

On Your Marks!

In the first phase, we perform indexing and retrieval. We use the CACM dataset. This phase is divided into three tasks-

1. Task 1: Build search engines using four different retrieval models.
2. Task 2: Perform query expansion using Rocchio's Algorithm.
3. Task 3: Perform stopping and stemming.

Get Set Go!

The following tasks were done in order to prepare the system, and get relevant initial files-

1. *Extract queries to a simple readable format*
    - "extract_queries.py" is a python program which takes the "cacm.queries" file as input and returns a file "queries.txt" which contains the queries given in the file, so that it is easier to feed them to the retrieval system.
    - This program cleans the queries by performing case folding and removing punctuations.
    - "queries.txt" is the file that all our systems use for processing queries.
2. *Tokenize the Corpus*
    - "task1_tokenizer.py" is a python program which takes the CACM Corpus in html format as input and processes it to generate tokenized corpus in .txt format.
    - We have used the NLTK library to perform tokenization.
    - NLTK library leaves behind some punctuations, which have been taken care of, manually.
3. *Index the Corpus*
    - "task1_indexer.py" is a python program which takes the tokenized corpus as input and creates the inverted index for the corpus.
    - It saves the index to the file "Inverted_Index.txt" in alphabetically sorted order.

Phase 1: Task-1 (Implementation)

In this, we performed four base runs on the following retrieval models. One for each model-

- Vector-Space-Cosine-Similarity Model
- BM25 Model
- TF-IDF Model
- Lucene Model

Vector Space-Cosine Similarity Model

This is principally similar to Task-2 of Homework 4. The result of this run is published in the next section.

TF-IDF Model

This model implements the retrieval ranking with tf-idf. In this,

1. First, the normalized term frequencies of the document terms and the query terms is calculated.
2. Next, the Inverse Document Frequency is calculated for the documents.
3. Finally, these two values are used to calculate the tf-idf score of the document.
4. Higher score refers to better ranking in the result.

The result of this run is published in the next section.

BM25 Model

This model implements the BM25 model as described in the textbook, *Search Engine: Information Retrieval in Practice, Croft et al.*

In that, the BM25 Score is calculated with the following formula,

$$\sum_{i \epsilon Q} \log \frac{(ri + 0.5)/(R - ri + 0.5)}{(ni - ri + 0.5)/(N - ni - R + ri + 0.5)} * \frac{(k1 + 1)fi}{K + fi} * \frac{(k2 + 1)qfi}{k2 + qfi}$$

Where K is, $K = k1((1 - b) + b.\frac{dl}{avdl})$

1. First calculate R, which is the total number of relevant documents for a query.
2. Then from the inverted index, the lengths of documents(dl) and the average document length (avdl) is calculated.
3. Next, the value of K is calculated by taking b = 0.75 and k1 = 1.2. These values provide the best results for TREC evaluations.
4. After that, the value of K is plugged in the BM25 formula along with the value of R that was calculated in step 1, and taking k2 as 100(empirically).
5. Then for each document, query terms are taken one by one, and for each query term, the value of r_i is calculated, where r_i is the number of relevant documents, in which that particular query term appears. This step is repeated for all query terms.

6. Next up, the BM25 Score is calculated for all query terms, and it is summed up to get the final BM25 Score for one document for one query.
7. This process is repeated for all documents, and for all query terms.

The result of this run is published in the next section.

Lucene Model

Lucene model implements the already built Lucene Search Engine. We re-use the code provided in Task-1 of HW4. The result of this run is published in the next section.

Phase 1: Task-1 (Analysis)

In Task-1, we implemented four runs. Time complexity is important, as running 64 queries on 3204 documents, if not handled properly, could have gobbled up a huge chunk of time. In order to reduce time complexity, we implemented dictionaries in Python which use hashmap instead of lists.

Similarly, in BM25 Model, we empirically chose the given values of **b, k1 and k2**, such that the results were as close to relevant results as possible. We tried on a few different values and finally arrived at the chosen values.

For further processing in upcoming tasks, we choose the BM25 model as its results have the best recall.

Amongst the four models, the MRR and MAP values of BM25 are the highest followed closely by Lucene, with Cosine and tf-idf trailing far behind.

Phase-1: Task-2 (Implementation)

In this we performed query expansion using **Pseudo Relevance Feedback** technique. We have taken **BM25 Model** as our go-to model for the rest of the tasks.

Choice of Algorithm

We have implemented **Rocchio's Algorithm**, which involves re-weighting terms to produce better results. Queries are automatically expanded by adding all the terms not in the original query that are in the relevant documents and non-relevant documents, using both positive and negative weights based on whether the terms are coming from relevant or non-relevant documents.

Actual Implementation

- To expand queries automatically, we run them through BM25 Model. The top 10 results are considered to be relevant documents for query expansion.
- The following formula is used for calculating Rocchio's relevance feedback-

$$\vec{Q}_m = (\alpha \,.\, \overrightarrow{Q_o}) + \left(\beta \,.\, \frac{1}{|D_r|} \,.\, \sum_{\overrightarrow{D_J} \epsilon D_r} \overrightarrow{D_J}\right) - \left(\gamma \,.\, \frac{1}{|D_{nr}|} \,.\, \sum_{\overrightarrow{D_k} \epsilon D_{nr}} \overrightarrow{D_k}\right)$$

- In this, the first term is the Query Vector, the second term is the Relevant Documents Vector and the third term is the Non-Relevant Documents Vector.
- We have assumed the following values-
  - α = 1.0
  - β = 1.0
  - γ = 0.5
- The resultant vector is sorted and top ten fields sorted on weight are picked for expanding the query.
- Finally, the expanded query is re-run with the BM25 Model.

The result of this run is published in the next section.

Phase-1: Task-2 (Analysis)

In this task, we implemented Rocchio's algorithm because-

1. It is easy to implement and provided in the textbook referred.
2. Pseudo-Relevance-Feedback provides satisfactory query expansion without employing external structures such as thesauri, dictionaries etc.

We observed that the result with query expansion on BM25 model deteriorated slightly, in comparison to pure BM25 model, in terms of MAP and MRR.

Phase-1: Task-3 (Implementation)

In this, we perform **Stopping** and **Stemmed Corpus Run** on the BM25 Retrieval Module. Stopping involves removal of common words from our query as well as our index. In Stemmed Corpus Run, we index the given stemmed index, and run two queries from the given stemmed query file.

Stopping

Here, we implement **Stopping** in our search engine's base run. The following steps were performed in order to achieve the objective-

- First, the python program for indexing was modified to account for stopwords, i.e., the words in the common_words.txt file were not indexed.
- Next, the BM25 model was modified to implement query-time stopping. While reading queries from the file, we searched for stopwords in the query. If found, they were removed from query.
- Finally, the processed query was sent to BM25 Model for retrieving results.

The result of this run is published in the next section.

Stemmed Corpus Run

Here, we implement the stemmed corpus and stemmed queries to see how the results vary with stemming. To implement this, we did the following-

- First, the python program for indexing was modified to account for stemmed corpus file, i.e., the words in cacm_stem.txt
- Then, the seven queries from the cacm_stem.query.txt file were run on the stemmed index with the BM25 Model.

Phase-1: Task-3 (Analysis)

We compare the stemmed and non-stemmed run on two queries-

1. **Stemmed Query**- parallel algorithm
   Non-Stemmed Query- Parallel Algorithms
   Query-by-Query Analysis: Performed on Top-10 documents retrieved from both runs.
   - In stemmed-query, 4 relevant documents were obtained in top 10 documents.
   - In non-stemmed-query, 5 relevant documents were obtained in top 10 documents.
   - Between stemmed and non-stemmed query, 6 documents are common in top 10 documents.
2. **Stemmed Query**- code optim for space effici
   Non-Stemmed Query- code optimization for space efficiency
   Query-by-Query Analysis: Performed on Top-10 documents retrieved from both runs.
   - In stemmed-query, 1 relevant document was obtained in top 10 documents.
   - In non-stemmed-query, 4 relevant documents were obtained in top 10 documents.
   - Between stemmed and non-stemmed query, 1 document is common in top 10 documents.

Phase-2: Seventh Run

Query Stemming and Stopping Combined

Here, we perform a seventh run (The stemmed corpus run does not count) where Query Stemming performed in Phase-1: Task-2 is combined with Stopping performed in Phase-1: Task-3.

- First, the stopwords index was used instead of the regular one.
- Next, query time stemming was performed on every query, and then Rocchio's algorithm was implemented on the queries and the corpus to retrieve the expanded results.

Phase-2: Assessing the Performance

In this, we now measure the performance of our system on the seven runs performed till now. We use four metrics to judge the performance.

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

Mean Average Precision

MAP is calculated by calculating the mean of average precisions for all 64 queries. The result is provided in the next section.

Mean Reciprocal Rank

MRR is calculated by averaging the reciprocal ranks of all 64 queries. The result is provided in the next section.

Precision at 5 and 20

It is the precision at a particular rank (5 and 20). The result is provided in the next section.

Precision and Recall

Here we calculate precision and recall as mentioned in the book *Search Engine: Information Retrieval in Practice, Croft et al.*

Extra Credit: Snippet Generation and Query Time Highlighting

We have performed snippet generation by taking query words, highlighting them in HTML bold tag. For snippet generation the phrase preceding and following the query words was published. The source code can be found in file named bonus.py

RESULTS

	MAP	MRR
BM25	0.386841	0.811172
TF-IDF	0.160140	0.446495
Cosine Similarity	0.169175	0.537316
Lucene	0.282510	0.683367
Query Expansion	0.288458	0.662286
Stopping	0.395802	0.768635
Expansion+Stopping	0.322894	0.612309

The following results are present in the following files in the submission folder-

Result for BM25 Model: task1_query_result_BM25.csv

Result for tf-idf Model: task1_query_result_TFIDF.csv

Result for Cosine Similarity Model: task1_query_result_VSCS.csv

Result for Lucene Search Engine: task1_query_result_LUCENE.csv

Result for Query Expansion: task2_expansion_result.csv

Result for Stopped Word List: task3_result_stopping.csv

Result for Stemmed Run: task3_result_stemming.csv

Result for Query Expansion with Stopping: task4_result_expansion.csv

Result for Precision and Recall of BM25: task4_result_BM25.csv

Result for Precision and Recall of tf-idf: task4_result_TF-IDF.csv

Result for Precision and Recall of Cosine Similarity: task4_result_VSCS.csv

Result for Precision and Recall of Lucene: task4_result_LUCENE.csv

Result for Precision and Recall of Query Expansion: task4_result_queryexpansion.csv

Result for Precision and Recall of Stopped Word List: task4_result_stopping.csv

Result for Precision and Recall of Expansion with Stopping: task4_result_expansionstopping.csv

We see from the above evaluations, that BM25 is the best system, closely followed by Stopwords list, while tf-idf appears at the bottom, with the worst MAP and MRR values.

CONCLUSION AND OUTLOOK

From the individual analysis provided above and the Results, we conclude that BM25 Model works best, as it has the best Precision and Mean Reciprocal Rank Values. It is followed by the BM25 Model with Stopwords removed. Lucene and Query Expansion follow after that. Tf-idf comes in the end and it has least precision value.

Outlook-

Given more time, we can improve the BM25 Model, by experimenting with the values of k1 and k2. In future, we can implement **hidden markov model or artificial neural network,** which can predict the next word in the query by using the context of the previous word. This can be further expanded to implement Contextual Search as currently employed by Bing Search Engine.

BIBLIOGRAPHY

1. http://www.ercim.eu/publication/ws-proceedings/DelNoe02/hiemstra.pdf/
2. https://en.wikipedia.org/wiki/Rocchio_algorithm
3. *Search-Engine:Information-Retrieval-in-Practice, Croft et al.*
4. http://nlp.stanford.edu/IR-book/html/htmledition/results-snippets-1.html
5. http://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/
6. http://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/
7. http://www.cs.cmu.edu/~wcohen/10-605/rocchio.pdf
8. https://www.cs.cornell.edu/people/tj/publications/joachims_97a.pdf
9. http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
10. http://www.site.uottawa.ca/~diana/csi4107/cosine_tf_idf_example.pdf