Information retrieval

Project

Fall’16

Dr. Nada J. Naji

Aly

Akhtar

Lalit

Pathak

Tushar

Gupta

Contents

[INTRODUCTION 3](#_Toc468982378)

[LITERATURE AND RESOURCES 4](#_Toc468982379)

[IMPLEMENTATION AND DISCUSSION 5](#_Toc468982380)

[Phase 1: Introduction and Initial Setup 5](#_Toc468982381)

[On Your Marks! 5](#_Toc468982382)

[Get Set Go! 5](#_Toc468982383)

[Phase 1: Task-1 (Implementation) 6](#_Toc468982384)

[Vector Space-Cosine Similarity Model 6](#_Toc468982385)

[TF-IDF Model 6](#_Toc468982386)

[BM25 Model 6](#_Toc468982387)

[Lucene Model 7](#_Toc468982388)

[Phase 1: Task-1 (Analysis) 7](#_Toc468982389)

[Phase-1: Task-2 (Implementation) 8](#_Toc468982390)

[Choice of Algorithm 8](#_Toc468982391)

[Other Nuts and Bolts 8](#_Toc468982392)

[Phase-1: Task-2 (Analysis) 8](#_Toc468982393)

[Phase-1: Task-3 (Implementation) 9](#_Toc468982394)

# INTRODUCTION

# LITERATURE AND RESOURCES

# 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-

#### 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, sans the decorations(using BeautifulSoup) 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.

#### 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.

#### 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,*

*Where K is,*

1. First, we calculate the value of 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) was 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 ri is calculated, where ri 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)

## 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.

### Nuts and Bolts

* 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-
* 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-
* = 1.0
* = 1.0
* = 0.5
* The resultant vector is sorted and top ten fields sorted on weight are picked for expanding the query.

## Phase-1: Task-2 (Analysis)

## 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 implemented **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 slightly modified to implement query-time stopping. Here, when reading queries from the file, we searched for stopwords in the query, and if found, they were removed from the query.
* Finally, the processed query was sent to BM25 Model for retrieving results.