Mechanics of Search: Assignment 1

MARCEL AGUILAR GARCIA, National University of Ireland, Galway, Ireland

Three information retrieval systems, Vector Space Model, BM25, and Language Model with Jelinek-Mercer Smoothing are implemented in Python and evaluated using a collection of 12,208 documents and 160 queries. Evaluation of the documents with judged results and eval_trec shows that BM25 performs better in terms of efficiency but Jelinek-Mercer Smoothing provides very similar results x4.5 faster.

Additional Key Words and Phrases: information retrieval,vector space model, language model, BM25

Link to GitHub Repository: https://github.com/marcelagga/Assignment_1_MoS.git

INTRODUCTION

In this report, three different information retrieval systems are evaluated on a collection of 12,208 documents extracted from newspapers. The implementation of these information retrieval systems has a common architecture:

- extraction and pre-processing of the documents from the collection
- computation of inverted index
- calculation of similarity between documents and query

The three systems evaluated are Vector Space Model, BM25, and Language Model with Jelinek-Mercer Smoothing. The main difference between these systems is the method used to calculate the similarity between the documents and the query.

A sample of 160 topics and a file with judged results is used to perform the final evaluation using eval_trec.

This report has been split in the following sections:

- **Indexing**: Pre-processing of the documents and main data structured used to store required information.
- **Search and ranking**: Methods used to calculate the similarity score for each system.
- **Evaluation**: Collection and topics used to evaluate the systems, and summary of the results obtained by eval_trec.
- **Conclusions:** Final conclusions from evaluation section and possible future work.

Author's address: Marcel Aguilar Garcia, National University of Ireland, Galway, Galway, Ireland.

Manuscript submitted to ACM

INDEXING

Document Analysis and Pre-processing

All documents from the collection are in XML format. Each document has three main tags: DOC_ID, HEADLINE, and TEXT. For each document, HEADLINE and TEXT are extracted together as one text and pre-processed. This pre-processing involves:

(1) **Removal of non-valid words**: Non-valid words are considered English stopwords or any word containing non-alphabetical characters (e.g. numbers or symbols).
(2) **Text tokanization**: Splitting the text in words. This has achieved by using word_tokenize from NLTK Python library.
(3) **Lowercasing**: Each word has been converted to lowercase
(4) **Stemmization**: Reducing the words to the root. This has achieved by using PorterStemmer from NLTK Python library.

As only the document word counts are required, the text processing part of the implementation returns a dictionary with the document terms (keys) and their term frequency (values) for each document. A Counter has been used to provide this calculation in an efficient way.

Finally, a dictionary is used to link each document (DOC_ID) to its related Counter. In the following sections, I refer to this dictionary as **docs_processed**.

0.1 Data Structures and Index Construction

The index is used to efficiently access the data required to compute the similarity between the query and a document. In order to compute this similarity, these systems need to access only documents containing at least one of the words from the query. While a document-term matrix seems to be a natural way to find this information, is not efficient for this computation. Additionally, it can be expected that this matrix may contain many zero elements and therefore would utilise more RAM than needed. For this reason, the index used in this implementation is **inverted index**.

Inverted index provides a link between words and the documents where they occur. Additionally, it may contain metadata for each of the documents. In this implementation, the inverted index is stored in a dictionary that has, by keys, all words from the collection and, by values, dictionaries with the documents where they occur and the term frequency associated.

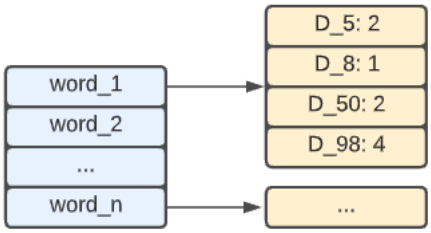

Fig. 1. Example inverted index with document term frequencies

The implementation of the inverted index has been done by using the following algorithm:

Algorithm 1 Calculation Inverted index

```
inv_index ← dict()
for doc in docs_processed do
    for word in doc do
        inv_index[word][doc] = docs_processed[doc][word]
    end for
end for
```

where **docs_processed** is the dictionary storing all pre-processed documents as explained in the previous section. Note that **docs_processed[doc][word]** is the term frequency of **word** in document **doc**.

SEARCH AND RANKING

All these IR systems rank the documents from the collection based on the similarity score between the query and each document. In this section, I intend to explain how this similarity is computed.

Vector Space Model

Let $\mathcal{V} = \{t_1, ..., t_n\}$ be the vocabulary of the collection of documents C. A document d can be represented as a vector within the n-dimensional vector space given by $\mathcal{V}$ by providing the term frequency of each word. In this way, if $t_{d,i}$ is the term frequency of t_i in the document d for $i = 1, ..., n$ then $d = (t_{d,1}, ..., t_{d,n})$.

The vector space model calculates the similarity of a document d and a query Q based on the cosine similarity of their vector representations. This is

$$\text{sim}(d, Q) = \frac{|d \cdot Q|}{||d|| \cdot ||Q||} = \frac{|\sum_{i=1}^{n} t_{d,i} \cdot t_{q,i}|}{||\sum_{i=1}^{n} t_{d,i}^2|| \cdot ||\sum_{i=1}^{n} t_{q,i}^2||}$$

In order to achieve better results, this implementation uses tf-idf weighting scheme. The tf-idf weight of a term t in a document d can be calculated as

$$\text{tf-idf}_{d,t} = \text{tf}_{d,t}\text{idf}_{d,t}$$

where $\text{tf}_{d,t}$ is the normalised term frequency of the term t in d and $\text{idf}_{d,t} = \log \frac{N}{df_t}$ with N being the total number of documents in the collection and df_t the number of documents where term t occurs. Normalising tf allows the algorithm to be sensitive to document lengths, while the inverse document frequency, penalises words that occur too often within the collection.

It's important to mention that $\text{sim}(d, Q)$ has to be calculated just for documents that contain at least one of the terms from the query Q. For a term $q \in Q$, all documents from C that contain this term can be easily found by checking the keys of $\text{inv}[q]$ where inv here represents the inverted matrix from the implementation.

The algorithm used in the implementation is as follows:

Algorithm 2 Cosine similarity

```
doc_tfidf ← calculate_tf_idf_components(doc)*
query_tfidf ← calculate_tf_idf_components(query)
norm_doc ← calculate_norm(doc_tfidf)
norm_query ← calculate_norm(query_tfidf)
for word in query_tfidf do
    if word in doc_tfidf then
        dot_product+= doc_tfidf[word] · query_tfidf[word]
    end if
end for
return (dot_product \(norm_doc · norm_query))
```

where **calculate_tf_idf** calculates the tf-idf weights of each document, and **calculate_norm** calculates the Euclidean norm for a given vector. *In the implementation, all collection document weights tf-idf are pre-computed and cached to improve the performance of VSM.

BM25

BM25 combines the weight w^{RSJ}, the term frequency derived from the 2-poisson model, and the document length normalisation into one single weight.

As in this collection there is no relevance information, Robertson and Sparck Jones weight gets simplified to a df weighting schema:

$$w^{RSJ} = \log \frac{N - n + 0.5}{n + 0.5}$$

where N is the total number of documents in the collection and n the number of documents containing the term t.

This weight can be used to provide an estimation of the 2-Poisson Model as seen below:

$$w(tf) = \frac{tf}{k + tf} w^{RSJ}$$

where k is an unknown constant and tf represents the term frequency of term t.

However, the 2-Poisson model assumes that all documents have the same length. For this reason, the term frequency can be normalised by using:

$$B = \left((1 - b) + b \frac{dl}{avdl} \right)$$

where $b \in [0, 1]$

Applying 2-Poission Model estimation and this normalisation, we can finally define the BM25 weighting function as:

$$w^{BM25} = \frac{tf}{k((1 - b) + b \frac{dl}{avdl} + tf)} \log \frac{N - n + 0.5}{n + 0.5}$$

note that, as explained in the lectures, existing experiments recommends to use $0.5 < b < 0.8$ and $1.2 < k < 2$. For the purpose of this assignment, I have used $k = 1.5$ and $b = 0.6$.

The algorithm used to implement BM25 is:

Algorithm 3 BM25 weighting

```
b ← 0.6
k ← 1.5
N ← len(collection)
for word in query_processed do
    if word in inverted_index then
        tf ← inverted_matrix[word][doc_id]
        dl ← docs_length[doc_id]
        avdl ← mean(docs_length.values)
        n ← len(inverted_matrix[word])
        bm25_score+= (tf/(k((1-b)+b(dl/avdl))+tf))log((N-n-0.5)/(n+0.5))
    end if
end for
return bm25_score
```

Language Model with Jelinek-Mercer Smoothing

A language model is able to assign a probability $P(w_1, ..., w_m)$ for a sequence of words to be generated in a given language. In a similar way, we can think of a document as a language by using its vocabulary, structure, etc. and calculate the probability $P(w_1, ..., w_m|d)$ for a sequence of words to be generated by document d. If we assume that all terms are independent, we can approximate this probability as

$$P(w_1, ..., w_m|d) = \prod_{i=1}^{m} P(w_i|d)$$

However, if the document is large enough, we could expect these probabilities to be close to zero and the computation of this product could cause underflow issues. For this reason, we can use the logarithm function:

$$\log P(w_1, ..., w_m|d) = \sum_{i=1}^{m} \log P(w_i|d)$$

The probability of a term occurring in the document d can be easily calculated as

$$P(w_i|d) = \frac{t_{d,i}}{|d|}$$

where $t_{d,i}$ is the term frequency of t_i in d and $|d|$ represent the length of document d.

While this is a good approximation, note that $P(t|d) = 0$ for all terms t that are not in d. For this reason, Jelinek-Mercer smoothing technique provides a balance between this probability and the probability $P(t|C)$ of the term t being generated by the language given by the collection of documents. This balance is brought by a paramater $\lambda \in [0, 1]$ as seen below:

$$P(t|d) = (1 - \lambda)\hat{P}(t|d) + \lambda\hat{P}(t|C)$$

In general, for shorter queries we may prefer to give more weight to $\hat{P}(t|d)$ and therefore select a lower λ and the opposite for longer queries. For the purpose of this assignment, and as I am taking TITLE and DESC from each query, I have set $\lambda = 0.7$.

Finally, by combining both formulas we have

$$\log P(w_1, ..., w_m|d) = \sum_{i=1}^{m} \Big((1-\lambda)\hat{P}(w_i|d) + \lambda \hat{P}(w_i|C) \Big)$$

In a similar way than in previous cases, the inverted index provides an efficient way to check only documents that contain query terms and find the term frequency of each word in a document. The algorithm used to implement JMS is:

Algorithm 4 Jelinek-Mercer Smoothing

```
λ ← 0.7
Dl ← sum(docs_length.values)
dl ← docs_length[doc_id])
for word in query_processed do
    if word in inverted_index then
        tfD ← sum(inverted_matrix[word].values)
        tfd ← inverted_matrix[word][doc_id]
        JMS+=log((1-λ)(tfd/dl)+λ(tfD/Dl)
    end if
end for
return JMS
```

Implementation Structure

The implementation uses two different classes TextProcessor and InformationRetrieval. The first class contains all methods used for the document pre-processing, and the second one has all the methods required to compute relevant documents for VSM, BM25 and LM. The pseudo-code used in this section are methods from InformationRetrieval.

EVALUATION

A collection of 12,208 documents extracted from newspapers and a sample of 160 topics with judged results have been used for evaluation. The processing of the documents is explained in detail in section **Indexing**.

Queries are in XML format and have four main tags: QUERYID, TITLE, DESC, NARR. For the purpose of the assignment, the evaluation uses the content of TITLE and DESC for each query. The query is processed in a similar way than the document, Counter is used to count words after ongoing the preprocessing seeing in **Index**.

Each IR system generates a file for the 1000 more relevant documents (ordered by similarity) to each query. This file is then compared to the judged results, test_qrels.txt, by using eval_trec.

A summary of the final results can be seen in the following table:

Metric	VSM	BM25	LM
MAP	0.194	0.28	0.269
P5	0.173	0.251	0.233
NDCG	0.369	0.442	0.432
Time Execution	172.97s	950.34s	229.06s

See below a description of the metrics used to evaluate the three IR systems:

- MAP (Mean Average Precision): Arithmetic mean of per-topic average precision. MAP is important in cases where the user is interested in finding many relevant documents for each query.
- P5 (Precision after five documents retrieved): P5 considers only the precision over the first five documents retrieved in each query. P5 is more relevant in cases where the user requires to find important information in the top relevant documents.
- NDCG (Normalized Discounted Cumulative Gain). Gain is accumulated starting at the top of the ranking and has lower value at lower ranks. NDCG considers top ranked documents more useful to the user and the otherway around.
- Time Execution: I thought it would be good to include the time that takes for each system to return the documents for all 160 queries.

By looking at these metrics, we can see that BM25 can be considered the best information retrieval system followed by LM, and finally VSM. However, BM25 and LM have very similar results and the implementation for LM is around x4.5 faster than BM25. In certain cases where the time for query execution is required, LM might be a better candidate to use.

By looking with more detail to BM25, we can say that, as P5 is 0.251, for the queries evaluated, around 1.25 out of 5 documents are relevant. When considering the whole set of documents relevant to BM25 and the precision of these documents, the system returned an average of 28% of relevant documents from the judged set. Finally, a NDCG of 0.442 tells us that less than half of the relevant documents have been positioned in higher ranks. The results are quite similar for LM. Unfortunately, VSM results are not great, for example, P5 being 0.173 leads to the conclusion that, in general, not even one relevant document is on the top five ranking.

CONCLUSIONS

From the three IR systems evaluated, VSM has not achieved good results. On the other hand, BM25 and LM have both shown much better performance than VSM. While BM25 performs slightly better than LM, it has shown a much worse time execution. For this reason, we may consider LM as the most suitable information retrieval system for most of use cases. There are many techniques that could be used to improve the performance of this system:

(1) The pre-processing techniques applied to text can be enhanced. The implementation uses stemmization to get the root of the word. However, I believe that by applying pos tagging techniques and lemmatization the systems could perform much better.
(2) The time execution has not been optimised properly. Using Numpy library and reviewing the complexity of the code, the time execution could probably perform much better.
(3) The time execution of BM25 could be improved by caching information in a similar way than the current implementation does for VSM.

(4) All results could be improved by using bigrams
(5) BM25 and LM depend on some parameters that have been fixed to a constant. It would be interesting to use parameter tuning to see if other values would perform better.

Note references: This report does not have references as I have not used any external information, just the content from the lectures.