CA6005

Assignment 1

Thomas Cronin
Student Number: 23105260
University of Galway

Link To Code : https://github.com/Tom-Cronin/CA6005-

ABSTRACT

The following paper focuses on Indexing, ranking and evaluation of models in context of Information Retrieval. The models used: Vector Space Model, BM25 and Language Model Unigram are explored in terms of theory and implementation. The effectiveness of each model is compared using trec_eval for the Cranfield set.

INTRODUCTION

Information Retrieval is used for managing large quantities of text and in a digital age this requires the ability to scrape data in different formats. The Cranfield dataset is an example of such a format for documents using tags to identify and categorize the texts. This paper gives a comprehensive overview of the indexing of such text, explores methodologies and implementation of models that can use the index to retrieve and rank documents. Finally an evaluation of the models is given via the trec_eval specifically looking at the MAP performance and precision at 5 documents.

1 Indexing

The first step in information retrieval is to index all documents available. indexing is a form of data processing which provides a quarriable and more efficient method for searching through documents.

Indexing speeds up the querying process. If indexing were not available for each token/term the application would have to scan every document for each query required which dramatically slows the computation times of any search. Indexing allows for the quick retrieval of relevant documents for a given query.

For this assignment an Inverted Index was used however there are several other forms of indexing implementations, such as:

- Direct Index
- Document Index
- Lexicon

All being a form of indexing, they differ in implementation from term queries and content / attribute-based queries.

1.1 Inverted Index

The Inverted Index is a method of indexing the documents in a collection by providing a token to document mapping, the reverse of the document to token mapping giving the term 'inverted' index.

The Inverted Index allows for the querying of tokens and returns the list of documents associated with each token.

The basic structure for the Inverted Index is that of a key, value pair:

- **Key**: Each key is a token
- **Value**: A list of documents / references to documents that contain the key.

When the application requires it uses the Inverted Index by querying the key value pairs to quickly generate the list of documents for the query by using each token within the query.

1.1 Inverted Index Implementation

The implementation of the Inverted Index itself is relatively simple but requires two key stages, **Corpus Processing** and **Index Generation.**

1.1.1 Corpus Processing

In order to index the documents, the document data needed to be extracted from the *cran.all.1400.txt* document. Each individual document within the text was formatted with the following tags

doc	The start/end of a document
docno	The identification number of the document
title	The title of the document
author	The author of the document
bib	The reference for the document
text	The text of the document

1.1.1.1 Data Crawling

To extract this information the documents were iterated over and each tag's context extracted and stored in a dictionary with the following keys: title, author, bib, text, full_text, doc_length. 'full_text' and 'doc_length' were generated variables to be used in the document indexing. 'full_text' is the concatenation of "title, author, bib, text" and doc_length is the length of the document after it has been tokenized, normalized and stop words have been removed.

1.1.1.1 Tokenization

The full text of the document is first split based on spaces into individual words and stored as a list of word ['word', 'word', …]. For each token in the list, the token is removed of most of its punctuation except for apostrophes, hyphens and brackets to avoid the splitting of words such as (O'Connell). This leave a list of tokens to be used in normalization and stop word removal

1.1.1.2 Normalization

To normalize the tokens each token is converted to lower case. This was the only form of normalization done during this assignment.

1.1.1.3 Stop word removal

In order to remove the stop words, a set of stop words from the English language was taken and stored in a set. The stop words were taken from https://www.cs.cmu.edu/~cburch/words/top.html and were calculated using Zipf's Law.

For each token a check was performed to verify that the token was not a stop word. If the token was a stop word it was ignored, if the token was not a stop word, it was added to a secondary list containing only non stop word tokens. This non stop word list of tokens is then returned to be used how the application sees fit.

1.1.1.4 Processed Documents

The final result of the data processing is a dictionary of documents where the key for each document is the docno (document id) with a value of a sub dictionary 'doc_parts' containing title, author, bib, text, full_text, doc_length.

1.1.2 Index Generation

After the pre-processing is completed the generation of the Inverted Index was straightforward. From the 'full_text' variable for each document, the tokenization, normalization and stopword removal is preformed, then for each token, if the token is not in the inverted index dictionary yet, the token is added and value is set to an empty set to ensure duplication of document Id's do not occur, then the document id that the token is in is added to the set.

The result of this process is a dictionary of tokens where the key is a set of docid's referring to the documents where these tokens were found i.e an Inverted Index

1.1.3 Implementation Retrospection

Upon writing of this document, the author realized redundancies in the code where the tokenization process is being unnecessarily duplicated to calculate the document length and instead the token versions should be stored where the 'full_text' variable is.

2 Ranking

In information retrieval 'Ranking' is the process of determining relevance of each document to a specific query. There are several ways of ranking documents such as:

- Vector Space Model
- BM25
- Language Model (LM)

Each of these methods rank's the relevance of each document to the query based on certain criteria. The resulting ranks of the documents determine relevance and the higher the rank e.g 1,2,3 where 1 is the highest value the more relevant a document is.

2.1 Vector Space Model

The Vector Space Model is a ranking model that determines the relevance of documents and queries based on the token weight and similarity of document to query. In the Vector Space Model both queries and documents are represented as points within a vector space. Each document and query can be represented as a weighted vector of the index terms. The vector can be represented as the TF-IDF (term frequency – inverse document frequency).

The resulting vectors of the TF-IDF can be compared for similarity using functions such as cosine similarity to determine the similarity of the vectors.

2.1.1 Vector Space Model Implementation

The Vector Space Model implementation is comprised of the following steps:

- Inverse Document Frequency (IDF) calculation
- Vector Space Generation
- Query Preprocessing
- Cosign Similarity Calculation
- Document Ranking

2.1.1.1 Inverse Document Frequency

The Inverse Document Frequency (IDF) is required to reduce the wait of tokens that appear frequently across documents, bloating the search query and are effectively less informative due to the number. There for a inverse document frequency dictionary was created to store term frequency, for each term in the inverted index, the log of the overall number of documents divided by the length of the term documents (i.e. the amount of documents the term was in) is calculated and stored for the term, giving a representation of key value pair where the key: term and value: IDF was stored and returned.

2.1.1.2 Vector Space Generation

To generate the Vector Space Model (the vector space for the documents) the Inverted Index and processed documents were used to calculate the IDF per term (See 2.1.1.1). Each document was then tokenized and processed similar to that of the Vector Space Model.

For each token per document the token frequency was calculated for later use where the token is the token is the key and the count of the token (frequency) is the value of the token.

Finaly the Token term frequency (TF) and the IDF scores per token per document is used to calculate the TF-IDF. First the normalized version of the term frequency was calculated to reduce the bias of the short and large documents. Then the normalized term frequency is multiplied by the inverse document frequency giving the TF-IDF which is stored in the vector_space_model in the following format:
Vsm = {document_id: {token: tf_idf}, token: tf_idf}, …, token: tf_idf}}

Giving the vector space for each document.

2.1.1.3 Query Preprocessing

Similar to the Vector Space Generation (See 2.1.1.2) The query is first tokenized, and the term frequency is calculated in the same way as the vector space generation. Rather than recalculating the IDF, the vector space model returns the IDF to be used in the query generation.

The TF-IDF is calculated for the query by the same method as the Vector space model and the resulting vector of the query is returned to be used for cosine similarity calculations.

2.1.1.4 Cosine Similarity Calculation

The cosine similarity calculation takes two vectors (vector_1 and vector_2) which represent the query vector and the document vector. The dot product is calculated for the two vectors, then the sum of the squares of each vector is calculated and scored. The resulting summed squares are then square rooted. Finaly the dot product is divided by the square root vectors multiplied by each other giving the cosine similarity of the vectors. The result is returned by the function to be used in the document ranking

2.1.1.5 Document Ranking

For a given query the document ranking first uses the query preprocessing to get the vector. Then for each document vector stored by the Vector Space Model the score is calculated using the cosine similarity calculation and scored. The scores are stored in a dictionary where the key is the document id and the value is the cosine similarity score.

Finaly the scores are sorted based on the score of each document and returned.

2.1.2 Vector Space Model Retrospection

Similar to the Inverted Index there was a lot of unnecessary code repetition. In future creating more generic code for reuse would be better. The storing of the vectors is over engineered, updating the document dictionary to contain a list of the vector space would have been more efficient.

2.2 Best Match 25

The Best Match 25 (BM25) model uses a mix of term frequency, inverse document frequency and normalized document length to rank documents for a given query.

The BM25 model is a modification of the 2-Poisson and uses the 2-Poisson model as its base however it uses a formula that is similar in behavior to w^{elite} .

The key concept behind the BM25 model is that it ensures that the contribution that each term per document relevancy does not continuously increase and reduce the effect of term saturation.

The BM25 model calculates the inverse document frequency and term frequency and length normalization to reduce bias from shorter and longer documents.

The overall BM25 function is as follows.

$$w^{BM25} = \frac{tf'}{k_1 + tf'} w^{RSJ} = \frac{tf}{k_1\left((1-b) + b\frac{dl}{avdl}\right) + tf} w^{RSJ}$$

2.2.1 Best Match 25 Implementation

The BM25 model is implemented like so:

- BM25 Score Calculation
- BM25 Query Scorer
- Ranking
- Query Processing

2.2.1.1 BM25 Score Calculation

The Inverse Document Frequency (IDF) is calculated from the number of documents with the terms (n) and the total number of documents (N)

The normalization factor is calculated to reduce the impact of the larger documents that cause bloating the terms.

tf prime is then calculated by dividing the term frequency provided to the function with the normalization factor.

Finaly the BM25 score was calculated by multiplying the IDF with the tf' divided by the tf' plus the saturation parameter k1.

The resulting score for the term is then returned.

2.2.1.2 BM25 Query Scorer

This stores the BM25 scores for the documents per token. It is stored in the format {docid: score}.

For each token in the query, it checks if the token is in the inverted index. Provided the token is in the inverted index the document id's are taken from the Inverted Index and the total number of documents containing that term is taken.

For each document found, the term frequency is calculated and the document length is calculated as before and the BM25 score is calculated for that token. The token score is added to the bm25 scores dictionary which contains score to document id in the format docid: score

2.2.1.3 Document Ranking

For this code the ranking and running of the model is combined, the extra variables for the bm25 such as average document length at the beginning.
The returned scores for the query is sorted based on the query score and given a rank from 1 to n where 1 is the best and n is the worst Similar to how the ranking was done with the Vector Space model. (See 2.1.1.5)

2.2.1.4 Query Processing

Similar to the Inverse Index (See 1.1.1.1 to 1.1.1.3) The query is tokenized in the same fashion.

2.2.2 BM25 Retrospection

Similar to the Vector Space Model there was a lot of code duplication across both models. The query creation and variable creations were done in the same code block and would of benefited from being extracted into two separate functions.

2.3 Language Model

Language models are the basic stepping stones for Natural Language Processing. By using probabilities of word sequences they are able to determine the likelihood that words appear together allowing models such as information retrieval, machine translation and more.

For this assignment a Unigram model (1 ngram) model was used for its implementation.

Smoothing is required to handle unseen words in the corpus that would normally give a zero probability. The Jelinek Mercer smoothing technique solves this issue by using lambda a value from 0-1 to modify the probability of unseen words to ensure they have a low value but not a zero value for their probability.

2.3.1 Unigram Implementation

The Unigram model is implemented like so:

- Corpus Training
- Smoothing
- Document Scoring
- Document Ranking

2.3.1.1 Corpus Training

For each document the text was extracted and tokenized in the same process as before. After being tokenized a Counter is used to count the occurrence of the tokens.

Once all the documents have been iterated over the counter for each token is then summed and the total of number of terms in the corpus is calculated.

Finally the collection of tokens and individual counts and the total count is returned.

2.3.1.2 Smoothing

The term frequency is retrieved from the document count for the token, the token collection probability is calcued by getting the total times the document appears and dividing it by the total collection of tokens

2.3.1.3 Document Scoring

The query is tokenized as before. Similarly the Documents are tokenized and a counter for the tokens is created to store a count of all the tokens in the documents.

For each token I the query the token probability is calculated using the smoothing technique. Provided the probability is not 0 the log of the score is aggregated for the tokens. Finally the score for each document is stored to be used in the Document ranking phase.

2.3.1.4 Document Scoring

The Document ranking and running is also tied together similar to that of the BM25 model.

2.3.2 BM25 Retrospection

Similar to the Vector Space Model and BM25 there was a lot of code duplication across both models.

The author is unsure if the implementation of this model is correct as there were a lot of division by 0's that should have been handled by the lambda in the smoothing technique and as such to get a ranking model if statements catching these were implemented.

3 Evaluation

3.1 Cranfield Collection

For this assignment the Cranfield Query and Documents were provided. The reading of the Cranfield documentation extraction can be found in (1.1.1 Corpus Preprocessing). A similar technique was used to read the Query as the documents however a modification was required as the fields 'num' and 'title' were used for the queries with 'top' separating each query.

Each model was fed the querys in the format {query_id: query_text} and ran. The models then output their results top 100 rankings for each query to their designating files 'model_name_results.txt'.

These text files were then used with the trec_eval execution file resulting in the outputs of each model.

3.2 Model Results

Table 1: Vector Space Model:

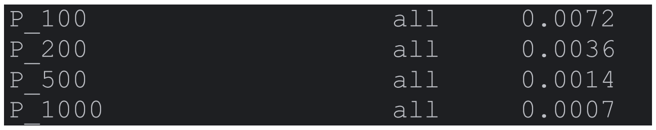
```
runid                   all
vsm_result
num_q                   all     152
num_ret                 all     15200
num_rel                 all     1074
num_rel_ret             all     119
map                     all     0.0080
gm_map                  all     0.0001
Rprec                   all     0.0068
bpref                   all     0.0685
recip_rank              all     0.0276
iprec_at_recall_0.00    all     0.0314
iprec_at_recall_0.10    all     0.0273
iprec_at_recall_0.20    all     0.0170
iprec_at_recall_0.30    all     0.0120
iprec_at_recall_0.40    all     0.0094
iprec_at_recall_0.50    all     0.0048
iprec_at_recall_0.60    all     0.0012
iprec_at_recall_0.70    all     0.0012
iprec_at_recall_0.80    all     0.0003
iprec_at_recall_0.90    all     0.0003
iprec_at_recall_1.00    all     0.0003
P_5                     all     0.0132
P_10                    all     0.0118
P_15                    all     0.0101
P_20                    all     0.0109
P_30                    all     0.0092
P_100                   all     0.0078
P_200                   all     0.0039
P_500                   all     0.0016
P_1000                  all     0.0008
```

Table 2: BM25:

```
runid                   all     BM25
num_q                   all     152
num_ret                 all     15200
num_rel                 all     1074
num_rel_ret             all     111
map                     all     0.0087
gm_map                  all     0.0001
Rprec                   all     0.0092
bpref                   all     0.0622
recip_rank              all     0.0315
iprec_at_recall_0.00    all     0.0350
iprec_at_recall_0.10    all     0.0304
iprec_at_recall_0.20    all     0.0189
iprec_at_recall_0.30    all     0.0145
iprec_at_recall_0.40    all     0.0060
iprec_at_recall_0.50    all     0.0035
iprec_at_recall_0.60    all     0.0019
iprec_at_recall_0.70    all     0.0007
iprec_at_recall_0.80    all     0.0003
iprec_at_recall_0.90    all     0.0003
iprec_at_recall_1.00    all     0.0003
P_5                     all     0.0158
P_10                    all     0.0112
P_15                    all     0.0123
P_20                    all     0.0105
P_30                    all     0.0092
P_100                   all     0.0073
P_200                   all     0.0037
P_500                   all     0.0015
P_1000                  all     0.0007
```

Table 3: Language Model:

```
runid                   all
run_LM_1ngram
num_q                   all     152
num_ret                 all     15200
num_rel                 all     1074
num_rel_ret             all     110
map                     all     0.0081
gm_map                  all     0.0001
Rprec                   all     0.0100
bpref                   all     0.0642
recip_rank              all     0.0298
iprec_at_recall_0.00    all     0.0344
iprec_at_recall_0.10    all     0.0287
iprec_at_recall_0.20    all     0.0183
iprec_at_recall_0.30    all     0.0127
iprec_at_recall_0.40    all     0.0046
iprec_at_recall_0.50    all     0.0037
iprec_at_recall_0.60    all     0.0018
iprec_at_recall_0.70    all     0.0018
iprec_at_recall_0.80    all     0.0003
iprec_at_recall_0.90    all     0.0003
iprec_at_recall_1.00    all     0.0003
P_5                     all     0.0118
P_10                    all     0.0118
P_15                    all     0.0110
P_20                    all     0.0092
P_30                    all     0.0092
```



```
P_100                    all     0.0072
P_200                    all     0.0036
P_500                    all     0.0014
P_1000                   all     0.0007
```

3.2.1 Model Result Interpretation

Table 1:
The VSM performed poorly the mean average precision is exceeding low at 0.0080 the precision at 5 document retrieval is also low at 0.0132 overall the model suggests that the VSM is running extremely poor for the documents.

Table 2:
The BM25 performed poorly the mean average precision is low at 0.0087 the precision at 5 document retrieval is also low at 0.0158.

Table 3:
The BM25 performed poorly the mean average precision is low at 0.0081 the precision at 5 document retrieval is also low at 0.0118.

Overall Results:
The bm25 model performed the best across all aspects of the trec evaluation it performs particularly better at the precision 5 recall point.

Overall the modle performances goes from Bm25 > language mode > vector space model.

CONCLUSION

Each model had difficulty in ranking documents effectively however the BM25 model performed the best. The author believes that their implementation of the models is the issue. The code was primarily created in a function based methodology which causes redundancies and difficulties in refactoring. Class based models specifically for the Language model would most likely increase performance as well as using a more complex model for the Language model.