1 Introduction

This paper introduces about the high-level descriptions of four types of search engines, including a boolean search, a phrase search, a proximity search, and a ranked Information Retrieval based on Term Frequency-Inverse Document Frequency(TFIDF). In this paper, I will discuss data pre-processing processes, a positional inverted index, the four types of search functions, and the evaluation of the programme.

2 Implementation

2.1 Environment

This programme was examined under Python 3.8.12. The search result was tested on a iMac with 3.3 GHz 6-Core Intel Core i5 processor and AMD Radeon Pro 5300 4 GB Graphics. The system was tested with 5000 The Financial Times articles and their relevant query.

2.2 Output

A total four files were included relevant to this system; one python file, two text files, and one report document.

Python file: *copy.py*, which contains a SearchEngine class and a RankedIR class that includes all relevant functions and variables for the systems.

Text files: results.boolean.txt, results.ranked.txt, which contain the outputs of the system within a certain query and documents, as mentioned above.

Report file: this paper is the report document.

2.3 Preprocessing

The preprocessing process was performed under the following steps:

1. Tokenisation;
2. Casefolding;
3. Stemming;
4. Stopping.

The common premise for preprocessing is that both documents and queries were preprocessed on the a homogeneous way. In this section, we explain document preprocessing.

2.3.1 Tokenisation & 2.3.2 Casefolding.

Although it depends on the query, the sequence of word characters is generally extracted as a token by splitting white space. This code processes each character in the input string token, and removes any non-alphabetical letter or a space. Lowercasing the tokens uses the built-in *token.lower().*

2.3.3 Stemming.

Stemming used the PorterStemmer which is a package provided by NLTK. This package is the most common stemming algorithm for English.

2.3.4 Stopping.
Stopping used the given stop words list.

2.4 Indexing

The system selected a positional inverted index for the easier and faster text retrieval of search engines. However, there are several disadvantages to the indexing methods, such as increased storage requirements due to the extra positional information stored, and slower indexing and so on. First, the system used a regular expression to find all textual information inside each tags (HEADLINE and TEXT) in xml format. The information retrieval using a regular expression was performed in document units. The retrieved textual data were preprocessed and stored in a list data structure before indexing. The list data structure gave us positional information of each token in a single document. Afterwards, the positional inverted index employed *defaultdict(list)* data structure from the package *collections* for indexing. The key is the preprocessed token extracted from the corpus 5000 *The Financial Times* articles, and the corresponding values are the document ID and the positions of the token in each relevant document. Positions for each document begin with 1, not 0.

2.5 Search Engine

Each engine used the corresponding query. The corresponding query has a different format, and we preprocessed each query for the corresponding engine. In a high-level description, we have a SearchEngine class and a RankedIR class. The SearchEngine class deals with a boolean search, a phrase search, and a proximity search. The RankedIR class handles with ranked IR based on TFIDF search. Two query files were assigned to the system, such as queries.boolean.txt and queries.ranked.txt. Each class was built for the corresponding query.

2.5.1 SearchEngine class
The objects of the class works on the term-based which locates in-between operators (‘AND’, and ‘OR). A single query(i.e. a single term) or each term of a query has a mark indicating which search engine it corresponds to. For instance, a normal term without any marks is used for a normal boolean search. A term starting with ‘\”’, for a phrase search. And, a term beginning with ‘#’, for a proximity search. In the preprocessing stage, the system separated each term based on the operators. Each separated term was preprocessed. The operator was moved to the end of the query for the later operation of each term.

2.5.1.1 Normal Search(*boolean_normal_search* function)
This function finds a list of documents where the term appeared. As the system memory has the positional inverted index of all documents, it can find the document where term appeared in from the index. Furthermore, this function can operate the NOT operator. We can subtract the list of all documents of the corpus from the list of documents where the term, which we want to exclude, appeared.

2.5.1.2 Phrase Search(*phrase_search* function)
After preprocessing of the term and removing ‘\”’, the system checks if the first term of the query is followed by the second term in the index. If so, the function returns a list of all document IDs corresponding to documents having the phrase in the index.

2.5.1.3 Proximity Search(*proximity_search* function)
The input of the function looks like #number(term1, term2). The number of the input refers to the proximity distance between term1 and term2. After we separate the proximity distance from the input, we preprocessed each term. If the distance between each term is less than or equal to the proximity, the function returns a list of all document IDs satisfying the proximity distance from the inverted index.

2.5.2 RankedIR class

The Ranked IR (TFIDF) search was built based on the following formula:

$$w_{t,d} = (1 + \log_{10} tf(t,d)) \times \log_{10}\left(\frac{N}{df(t)}\right) \tag{1}$$

$$\text{Score}(q,d) = \sum_{t \in q \cap d} w_{t,d} \tag{2}$$

,where
q refers to query input.
d refers to a single document.
$tf(t,d)$ refers to the number of times term t appeared in document d.
$df(t)$ refers to the number of documents term t appeared in.
N refers to the number of documents in a collection.

In the Ranked IR search, we want to find the best scored documents based on TFIDF for a query. In the equation (2), the score was calculated by having an input, including a single query and a single document. Each query consists of terms. When calculating the score, as depicted in the equation (2), we considered a term found both in a query and a document, and we retrieved all documents which include at least one of the terms from a query. Other parts of the calculation for getting the score are quite straightforward, which were illustrated on the code.py well.

3. Evaluation

The whole system took 76.75563 seconds for running all processes, mentioned above, with 5000 news articles and the corresponding queries to search engines, as shown in Appendix A. The system speed may be relatively slower than the optimal speed based on the coursework 1 guideline of the Text Technology for Data Science module at The University of Edinburgh(Al Hariri, 2023). According to the guideline, my system should take around 50-60 seconds. We assume that reducing the number of loading indexes from memory may increase the system speed for the project. Also, potentially, using other index methods would improve the system speed. Selecting proper index methods should be determined by the characteristics of the search engine purpose.

3.1 Challenge.

This challenge test results are based on 5000 news articles and the corresponding queries. As mentioned above, the system has two search classes for each query. Each class indexes 5000 new article documents for each their query search. SearchEngine class took 14 minutes for building a positional inverted index while RankedIR class took time 31 minutes for the loading index and searching process. It is an insane increase. Although we expected the running time for indexing to increase, we did not anticipate it being exponentially expensive. One intriguing finding is that the size of index without stopping is merely twice as with stopping. As stop words represent the most common words in a collection, the system indexes rather uninfluential words regarding aboutness. For instance, the word ‘a’ was appeared in 4721 documents, which is 94% of the total document. We know stopping are sometime very important, so we need to properly handle stopwords for the purpose of search engine system.

Appendix A

Measurement = seconds
index_running_time = 37.71451282501221
search_indexload_running_time = 1.9073486328125e-06
search_running_time = 2.933112144470215
rankedIR_indexload_running_time = 5.0067901611328125e-06
rankedIR_running__time = 36.119012117385864

References

Al Hariri, Y. (2023). ttds_2023_07_ranked_retrieval. Presentation presented at the University of Edinburgh