Text Technologies for Data Science
Coursework 1 Report

Zhan Wang (s2247161)

28 October 2021

1 Introduction

This report describes the methods used to complete Coursework 1 for Text Technologies for Data Science, where we are asked to implement a simple Information Retrieval (IR) system.

The IR system modules include preprocessing, creating a positional inverted index, searching and ranking. In the report, I will explain my implementation of each module and discuss the knowledge I learned, some challenges I faced and how to improve and scale my implementation.

In Section 2, the preprocessing of the text applying tokenisation, stopping and stemming is presented, together with the pros and cons for the chosen approaches. Next, Section 3 discusses the structures and algorithm used to build the positional inverted index. Then Section 4 presents how to use the positional inverted index to perform Boolean search, Phrase search and Proximity search. Section 5 explains the implementation of Ranked IR based on TFIDF. Finally, Section 6 is about a brief commentary on the system as a whole and what I learned from implementing it and a reflection on some problems and possible improvements that can be brought to this simple search engine.

2 Text Preprocessing

Before the inverted index can be built, we need to preprocess the text, it is essential because producing a list of normalised index for multiple forms of the same term can make a better retrieval in terms of performance and time.

To preprocess a text we need the following steps:

- Tokenization and Casefolding
- Stopping
- Stemming

The same methods for text preprocessing are applied for both the collection of documents and the queries subsequently run on the inverted index.

2.1 Tokenization and Case folding

Tokenisation is the process of splitting the text into smaller parts of the initial one, which are called tokens.

For Tokenization, I used the package `re()` and the simple regular expression `[\w+]` to split at non-letter characters and separate the text into word tokens, which means all the non-letter and non-digital characters are discarded. I didn't delete the numbers because I think that numbers in a document are important and may contain information such as year, specific values, etc.

This method simplifies the process, but also has some issues because the texts may contain hyphens and URLs. For example, a document containing composed words such as "high-school" might appear in searches such as "Environment of primary school". However, phrase searches such as "high school" (without the hyphen) would return the correct results.

Next, we turn each resulting word token to lowercase by using the built-in `lower()` function.

2.2 Stopping

Stopping is the process of filtering the list of words to remove all stopwords. We use the provided stopword list, which is not exhaustive but a good starting point for a simple search engine. The logic is simple: iterate through all the words and if it is in the stopwords, delete it, otherwise keep it.

2.3 Stemming

Stemming is the process of reducing morphological variations of words to a common stem, which usually involves removing suffixes. I chose the `nltk` package and the `PorterStemmer()` function to implement this and get a new list of tokens which will now help match words of similar form disregarding their plurality, case or conjugation.

3 Indexing

The next step is to build the positional inverted index incrementally, which will allow us to do fast full text searches and it is easy to develop.

3.1 Load XML

When we are reading through the document collection and a document in the collection contains three tags we are interested in:

- `<DOCNO />` - tag containing the document id
- `<HEADLINE />` - tag containing the headline of the document
- `<TEXT />` - tag containing the actual body of the document

So we can parse `trec.5000.xml` file through the `xml.etree.ElementTree()` package and stored the document IDs and their contents (which consists of HEADLINE and TEXT sections in the original XML file) in the `docs_df` , a `dataFrame()` structure from the package `pandas` of which the locs are document IDs and the corresponding values are the contents.

3.2 Data Structures

For each row of `docs_df`, which is each `HEADLINE` and `TEXT` tag corresponding to a document ID, we preprocess the contents and get the tokens. Remember to count the position of a term *after* after removing the stopwords because this reduces the number of positions in a document considerably and also reduces the time of merging postings.

The data structure I used for implementing positional inverted index is an `OrderedDict()` from the package `collections` called `index`, the keys are the preprocessed tokens, and the corresponding value is the posting of the token, an `OrderedDict()` of which the keys are document IDs where the token appeared and the values are lists of positions of the token in each relevant document.

4 Boolean Search

There are three types of queries that the `boolean_search` module can support, and the user can read the queries through the `queries.boolean.txt` file. The query terms need to go through the same preprocessing process as text before they are retrieved, this is to ensure that the search results are valid.

For each query, the boolean search is implemented in the following steps: first segmentation, then search for each part, and finally processing of the search results according to logical operators.

4.1 Splitting

We can split at `AND/OR`, if we get one term, which means that the query does not include logical operator `AND` or `OR` and then we only need to search one term, so the result is obtained directly after the `search()`.

If we get more than one subpart, we write them together with the intermediate operators to `querywords`, and then search for query term `word1` and `word2` to get `result1` and `result2` respectively.

4.2 Searching

During the search, define a bool value `haveNot`, if the query term you searched starts with `NOT`, define `haveNot = True` and remove the `NOT` to search, if there is no NOT in the query term, define `haveNot = False`.

There are three types of enquiries.

4.2.1 Phrase search

If the query term has `"` in it, we will use the `phrase_search()` which query for two consecutive words, remember that the order of phrase search is very important, so the result of the "word1, word2" query should be different from the result of the query "word2, word1".

First use `strip('"')` to remove the useless punctuation, then use `split()` to split the words and preprocess each word to get two tokens.

According to the `index` created, through `index[token1]` we can get the list of all document ids and the corresponding positions where token1 exists, and similarly we can get token2's.

If token1 and token2 appear in the same document and

$$position2 \; - \; position1 \; = \; 1$$

(note that the order of the two tokens is important here), then we can consider this document is a match of the query.

4.2.2 Proximity search

If the query term has `#` in it, we will use the `proximity _search()` which is similar to `phrase_search()`, and the difference is for the proximity search, we only care about whether the distance of the words is less than or equal to the given number, regardless of their order.

So also first remove all punctuation and obtain two query words and the maximum distance required which is noted as `num`, and preprocess each word to get two tokens.

If token1 and token2 appear in the same document and

$$abs \; (\; position2 \; - \; position1 \;) \; \leq \; num$$

then we can consider this document is a match of the query.

4.2.3 Single search

If it is a single word search, we will use `single_search()` where the input word is preprocessed to get the token for searching, and the `index[token].keys()` is output according to the established `index` to get all the corresponding `docids` of the token.

If a term is negated, which means we `haveNot = True`, we return all documents apart from the relevant ones for the non-negated term. For this, we keep a variable `alldocs` which is just a list of all the document IDs in the collection.

4.3 Logical Operators

After performing the searching for each term, we obtain two lists of relevant documents. We map each logical operator to `&` and `|`, and if the boolean term of the query is `AND`, we intersect the two lists, and if the term is `OR`, we perform a union of the list of documents (handled in Python by `set` operations).

AS we only expect one AND or one OR, and the terms can be negated with NOT and can be either simple words, phrases or proximity searches.

All boolean queries can be of the form:

$$Query \; \rightarrow \; term_1 \; (\; AND/OR \;) \; term_2$$

and term can be of the form:

$$term \; \rightarrow \; NOT \; term \; / \; word \; / \; phrase \; / \; proximity$$

The above three steps provide flexibility for any combination of queries and the results of Boolean queries can be found in `results.boolean.txt`.

5 Ranking

With simple Boolean or Phrase search, the documents retrieved either match or do not match. For a better understanding of

the user's needs when they search for a phrase, we use a scoring system to rank the documents that are most likely to satisfy the user's query.

5.1 TFIDF

We consider two metrics to calculate this score. The first one is called *term frequency* and is a count of how many times a term occurs in a document. The more times a term appears in a document, the more important that term is to that document.

The second measure is *inverse document frequency*. This is derived from a term's *document frequency* - the number of document the term appeared in. The more documents the term is seen in, the lesser its importance in the collection. Alternatively, the higher the *inverse document frequency* of a term, the rarer that term is in the collection. Consequently, the document containing the rare term will have a higher score in the query results.

TFIDF term weighting combines the two measures in a single formula:

$$w_{t,d} = (1 + log_{10} tf(t,d)) \cdot log_{10}(\frac{N}{df(t)})$$

Then, I take the union of the documents that contain at least one of the tokens in the query and get `subindex`, so for each document in `subindex`, we calculate the TFIDF of each term and add it up to get the final Score(q,d), which indicates how well the document d matches the query q.

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

5.2 Ranked IR

We first preprocess the query and for each term in the tokens, we compute the retrieval score of each document that the tokens in the query appear in using the TFIDF (term frequency - inverse document frequency) formula, which assigns a weight to each term that belongs to a document.

The higher the weight, the more relevant is the document for a specific query, so the results displayed to the user are sorted in a descending order by their query score, and can be found in `results.ranked.txt`, which includes at most 150 documents per query.

6 Summary and Outlook

I have learned a lot from this simple IR system and there are several improvements that can be brought to the system.

6.1 Lessons learned

Implementing this system from scratch gave me and in depth understanding of the algorithms presented in class. Choosing a good data structure for the inverted index was challenging, but `OrderedDict()` helped organise the code better.

6.2 Challenges

The challenges I faced were mainly related to the variability of the queries for the boolean search. Queries can include multiple terms connected with the logical operators `AND`, `OR` and `NOT`, as well as phrases. So it is important to clarify the code logic and make it as concise as possible.

6.3 Future works

For preprocessing, we could improve the tokenisation by retaining some punctuation marks that are parts of numbers, such as dots, commas, or the ones that are parts of a URL or an email.

The establishment of the positional inverted index is still very time-consuming and cannot be updated. If we use Delta Encoding or B-Trees to store the index, I think it will work better.

The current boolean search can only target special queries. It would be better if a more general search can be achieved, that is, it can handle queries containing multiple `AND`, `OR` and `NOT`, or make phrase search and proximity search can handle query with more than two words.

At present, I can only process the given file, and then I hope to realize automatic search for users through `argparse()` package.