TTDS CW1 Report

October 2020

1 Pre-processes Text

This section introduces the text preprocessing.

1.1 Tokenisation

Tokenisation is a process to transfer a text into tokens by splitting on certain characters. A token is an instance of a sequence of characters. I use the regular expression to split a text on non-letter and non-number characters. Then unify all tokens to lowercase which is called case folding.

1.2 Stemming

After getting lowercase tokens, I delete the stop words according to the given stop word list. Then carry out the stemming operation. Stemmers attempt to reduce morphological variations of words to a common stem. For this step, I use PorterStemmer in NLTK library.

2 Inverted Index

I use nested dictionaries to store the inverted index. First, parse the file stored in XML format. Traverse the DOC nodes in the file, that is, each document. Get the number and content of each document (including HEADLINE). Then preprocess the content to get tokens.

For each document, iterate through its list of tokens. For each token, find its position in the document first. The returned position is a dictionary whose key is the document number and the value is a list of positions. Then check whether there is a key equal to the token value in the inverted index dictionary. If not, use the returned location dictionary as the value and token as the key to create a new key-value pair. If so, use the returned location dictionary to update the value of this key.

In this way, the inverted index dictionary is created. Next, write the inverted index into a text file according to the project format requirements.

3 Search Functions

This section describes four search functions. For each query, use regular expressions to split the statement according to 'AND' or 'OR' to get the token list (exclude the serial number). Get the query serial number and the first token. If the first token starts with ["] and the query only has one token, enter the phrase search. If the first token starts with [#], enter the proximity search. Otherwise, enter the boolean search.

Except for 'AND' and 'OR' in each query, the remaining tokens are preprocessed before being used. Including case folding, determining whether it is a stop word, and stemming.

3.1 Boolean search

Enter the boolean search section. First, determine whether the length of the token list is one. If it is, enter a single token search. This step directly searches the inverted index according to the token and returns the corresponding document number list. If not, then consider whether it contains a phrase.

Perform phrase search on the phrase part, and perform a single token search on the non-phrase part. Both searches will return a list of document numbers. Convert these two lists into sets, and perform the corresponding intersection, union, and complement operations according to the operators contained in tokens. If no phrase is included, enter the normal search.

According to tokens and operators, determine whether it contains an 'OR' operation. If it is, and a certain token is not in the index, it will directly return the search result for another token. If neither exists, return an empty list. If it is 'NOT', 'AND' or the 'OR' operation where both tokens exist, get the list of document numbers corresponding to each token, and then perform the corresponding operation according to the operator.

3.2 Phrase search

Enter the phrase search section. Remove the ["] before and after the phrase, and then split it by spaces to get the token list. Get the corresponding value of each token in the inverted index, that is, a dictionary that contains the document number and the specific positions in each document. For each document number in the location dictionary of the first token, determine whether it is a key value in the location dictionary of the second token. If so, get two lists of the specific positions of these two tokens in the document. Traverse the first list to determine if there is a value equal to the first value plus one in the second list. If there is, add the document number to the returned list, and immediately exit the loop and start searching for the next document number.

3.3 Proximity search

Enter the proximity search section. First, split the query according to [#], [( ], and [ )] to get tokens and distance value. Get the corresponding value of each token in the inverted index, that is, a dictionary that contains the document number and the specific positions in each document. For each document number in the location dictionary of the first token, determine whether it is a key value in the location dictionary of the second token. If so, get two lists of the specific positions of these two tokens in the document. Traverse the first list to determine whether there is a value in the second list so that the absolute value of the difference between the two values is less than or equal to the distance value. If there is, add the document number to the returned list, and immediately exit the loop and start searching for the next document number.

3.4 Ranked IR based on TFIDF

When in the TFIDF section, I use the regular expression to split a text on non-letter and non-number characters.

Define a list for storing IDF values and a list for storing the index of each token. Traverse the token list (not including the request sequence number), and do the rest of the preprocessing operations for each token. Look up the token in the inverted index. If it exists, get the number of values corresponding to the key, that is, the DF value. Calculate the IDF value according to the formula and save it in the list. At the same time, the value corresponding to the key (the dictionary storing the document and the specific locations) is stored in the list. If it does not exist, the IDF value of the token is equal to 0. The list of sub-indexes is updated with an empty dictionary.

Next traverse the document number. For each dictionary in the sub-index list, if the current document number is the key in it, get the number of values corresponding to the key, that is, the TF value of the token in the document. Then calculate according to the formula. If this key does not exist, the value of this part is equal to 0. Then use the same subscript to get the IDF value corresponding to the token in the IDF list. Add the two values to get the TFIDF value of the current token, and then update the total TFIDF value.

The returned results are sorted in descending order of TFIDF value. When TFIDF values are equal, they are sorted in ascending order by document number. The TFIDF value retains four decimal places. Finally, it is stored in the file in the format required by the project.

4 Commentary and Gains

The system implements a simple IR tool. Features include preprocessing text, creating the inverted index, boolean search, phrase search, proximity search, and ranked IR based on TFIDF. The results of the system in the three labs are the same as the answers published on Piazza. The test set in coursework 1 can run normally.

In the process of implementation, I learned how to parse the XML file. Learned how to preprocess text and write files in a specific format. Also learned how to use a variety of data structures to store data. Based on the understanding of the algorithm, the system was finally implemented with Python.

5 Challenges

This section describes the challenges I encountered in the process of implementing the system.

1. When extracting the stored inverted index file, I have the problem of how to restore it to a dictionary. Later, by studying the data structure and experimenting, I solved this problem.

2. The document number (DOCID) doesn't equal the number of documents. At the beginning of lab3, I used the DOCID of the last document as the total number of documents, which caused the IDF value to be wrong.

6 Improve and Scale

This section describes some ideas on how to improve and scale the implementation.

1. When the file is very large, it is a waste of resources to read the index into the memory at once and store it in a dictionary. More efficient storage methods should be considered to improve efficiency.

2. The program integrates most of the overlapping logic. A small part of the context and data structure can be further optimized. Using a hash table is a possible way.