Text Technologies for Data Science Coursework1

s2020314

October 2020

1 Tokenisation and Stemming

I call function *preprocess*(*text*) to handle the text of each document and have it return a list of tokens after necessary processing.

The process includes:

1) Strip and lowercase the text;

Before complex processing, first remove the spaces at the beginning and end of the text using *str.strip*(). Then convert letters into lower case using *str.lower*().

2) Use regular expression to split the text into a list of tokens;

Compile a regular expression pattern using $r'[\backslash s\{\}]+'.format(re.escape(string.punctuation))$. Within which, $\backslash s$ is used to denote all whitespace characters and *string.punctuation* is used to represent all sets of punctuation. The pattern is used to recognize one or more occurrence of whitespace and punctuation. *re.split*() is then used to split the text by abovesaid characters. Heretofore, tokenisation is implemented.

3) Check stopword and stem the token.

Each token should first be checked whether it is a stopword before performing stemming. Stopwords are loaded from *englishST.txt* and transformed into a list. If a token is not in the stopword list, it can be stemmed using Porter stemming — *stem*(*token*) and added to the token list using *list.append*().

2 Inverted Index

All inverted index needs are *term*, *documentid* and *occurrence position*. I use a nested dictionary *termIndex* to store the inverted index with *term* being primary key, *document id* being secondary key and *occurrence position* being corresponding value.

I process each document in a for loop and for each document have received from preprocessing all terms that should be recorded in inverted index. Thereby, I call function *createInvertedIndex*(*tokens*, *docno*) for each document to add to inverted index. In this function, *index* is used to record the current position of term in document text, which is incremented by one in a for loop, and it is added to the value of the key *document id* under the primary key *term*.

After all documents are processed, the dictionary *termIndex* is fully prepared. Function *outputIndexTxt*() is then called to write the inverted index into *index.txt*.

3 Search Functions

Boolean search, phrase search and proximity search are three basic search methods and are implemented in a combined measure in function *processQueries*(). While ranked IR is processed via another function *processRankedQueries*().

3.1 Three Basic Search Functions

It is necessary to distinguish the abovesaid three basic methods. The first one that can be easily distinguished is proximity search, which has a particular form of query, using a # in the beginning of the query. After ruling out the possibility of proximity search, I use *shlex.split*() to split the query, which can protect phrase from being seperated and list all components of query.

The components of query now have two categories — *boolean relevant* like AND, OR, NOT and *informative context*. Phrase search is a part of informative context, along with plain word search. Now, I use *mid*, *neg*, *booleanChoice* and *preprocessedList* to paraphrase the query. Integer *mid* records the position of AND or OR in the query, which can be zero due to no boolean search.

Accordingly, *booleanChoice* records which one of AND and OR is used, using a structure of enum *Binary* to clarify the difference. List *neg* records all positions that NOT occurs in, which can be the length of zero to two. The *preprocessedList* is created by replacing all *boolean relevant* term list into a empty list and preprocess $informative\ context$. It's worth noting that boolean relevant term should not be deleted, otherwise the positions reside in *mid* and *neg* will have to be altered. Then temporarily ignoring all boolean relevant terms and process all component to fetch for corresponding documents.

In the whole, after reading the query file and having queries split, I process each query into its query number and the remaining query context. The processing of query context is demonstrated below. All the methods return a list of possible document id, which is then sorted and written into file *results.boolean.txt* by calling $writeIntoResults(file, query, answer)$.

3.1.1 Proximity Search

I first check the initial of query. If it is a #, I can directly preprocess the query context into list of tokens and call function $processProximitySearch(q)$. List q should contain three elements — $distance\ limit$, $former\ term$ and $latter\ term$.

I then need to find documents that contain both terms and have their distance within the limit. To reduce the processing time, I choose the term with fewer relevant documents as the range basis for for loop and iterate all possible documents to check firstly whether this document contains the other term. Distance checking only occurs when the document contains both terms.

In distance checking, I use $p1$ and $p2$ to record the current index of position in each position set. To check the distance, I will have to check the position of all index $p1$ and $p2$ that are close enough. Thereby, I use a while loop. It stops when any one of the position index comes to an end. If the distance between the positions corresponding to index $p1$ and $p2$ is smaller than or equal to distance limit, this document is valid and added to the list *proxiDocList*. Otherwise, to proceed the check, since the distance between those two positions are logically greater than distance limit, I need to increment by one the index with smaller position, to chase after the greater position — narrowing the distance. The function returns *proxiDocList*, which is a list of ids of all possible documents.

3.1.2 Phrase Search

After using *shlex.split*() as said above, $informative\ context$ is divided into two kinds — *phrase* and *single word*, in the form of list. If the list contains more than one word, this component can be seen as a phrase. I call function *phraseSeach*(*component*) to execute phrase search.

Similar to proximity search, I first need to find a list of document which contains both terms. Then I need to check the position. Instead of using absolute value of position distance in proximity search, phrase search requires exact order. In a for loop going through all possible documents, I use a flag *yesDoc* to denote whether the document contains phrase and another flag $yesFlag$ to denote the consistency of phrase from current starting position — *start*. *Index* is used to record the term position in the phrase, which allows the phrase to have more than two words. Only when $yesFlag$ is checked existent that *yesDoc* is marked positive and thus corresponding document is accepted and added to the $finalDocList$ to be returned.

3.1.3 Boolean Search

Heretofore, all the list of document retrieved for each component, ignoring NOT, are provided. I need to combine all the results and use boolean search to get a final result.

If *mid* equals to zero, no occurrence of AND or OR. The final result can be still — *neg* is [], or the complement set of all documents — *neg* exists and creates no conflict. It is the simplest situation and shows the main idea of handling boolean search:

1) *mid* can be zero, meaning no AND or OR is used, or be one in regulated format of query, meaning one AND or OR is used.

2) *neg* can be an empty list, meaning no NO is used, or a list with one or two elements, recording every occurrence of NO in query.

3) I print *Clash* to report possible conflicts among *mid*, *neg* and all list already retrieved. For example, when *mid* equals to one, length of *neg* equals to one and $neg[0]$ equals to $mid + 1$, the $mid + 2$ position of token list of a query should not be empty. It is noteworthy that *Clash* should not occur when the pattern of query is accurate.

4) After categorizing every possible query types in an if-else statement and ruling out *Clash*, function $boolSearch(choice, former, latter)$ is called. Within which, $former$ and $latter$ are already

transformed into the complement set using $notbutinD(notinlist)$ if NOT occurs in front of it. *choice* is the $booleanChoice$ mentioned before.

Function $notbutinD(notinlist)$ and $boolSearch(choice, former, latter)$ share the same idea. The former returns a list of document in the whole document set but not in the *notinlist*. The latter will find a list of document both occurs if *choice* is AND and find a list of document occurs in either one if *choice* is OR.

3.2 Ranked IR Based on TFIDF

The core idea of ranked IR is calculating the TFIDF. After doing the similar process in basic search methods, I now have *qwords* — a list of already preprocessed terms in each query context. For each query, I use a dictionary *score* to save the score of each term-relevant document corresponding to the query. For each term in query, iterate through all the documents and adding up the weight of term under $score[doc]$. Having results sorted and then written to file *results.ranked.txt* by calling $outputRankedResults(qno, rankedResults)$ for each query. *topN* is the upper bound for the number of output for each query and *index* is counted to help restrict to the bound.

4 Commentary and Gains

The system I implemented contains the basic functions expected from an information retrieval system. The total time consumed to create inverted index is around one minute, which is acceptable for a large dataset, and the querying process only take seconds, which is adequate for basic usage. However, the whole system can be improved to be more robust, flexible and user-friendly.

I have learned a lot from implementing the system — both theoretically and practically. In order to implement the system, I have to go through every detail of search methods which helps enhancing my control over knowledge and correcting misunderstandings. For example, in ranked IR I need to review the equation of TFIDF and I found myself confused over term frequency. In practical manner, I learned to parse the xml file, learned about mmap and shlex etc. and now can use python more confidently. I use visual studio code to handle code, git to version control my files, DICE and linux system to run my file on. Gradually, I find myself more fluent in handling new tools.

5 Challenges

1) Concept misunderstanding. When the code didn't work for the first time, I found it hard to realize it's something to do with wrong perception of an equation etc. 2) Logical thinking. In handling different query methods, it's exhausting to make my mind clear over which search method to handle first and what would happen if there are conflicts. Although the current code can only handle a limited amount of possibilities, it is still a challenge when I try to fit the system into a bigger scenario. Also, when I'm implementing phrase search, the dynamic change of two different positions and how to distinguish the position and the index of position in the position list are a pain.

6 Improvement and Scaling Up

1) In preprocessing the text, I now simply subtracting all special characters, which makes meaningful text containing special characters something meaningless. For example, time, certain names, URLs etc. Also, I changed all characters into smaller case, which may confuse words when different use of capitals can mean different things, like Windows and windows.

2) Corresponding to preprocessing improvement, the query should also allow distinguishment in the case of named entity, different capitals etc. Also, the query only allow certain pattern of input which is not flexible and can be further applied to fit into more situation.

3) The storage of data in the disk is not efficient and space-saving. When the system need to process large amount of documents, both the space and time will face challenge.

4) More user-friendly system would require an user interface and allow typos in query. Auto-check the query and allow more interaction with the user can be considered.