**COMP34711 Coursework 1: Inverted index Report**

Minjun Choi

**Index term choices (Task 1)**

With regards to the selection of terms, there are methods that have been used to exclude the unnecessary words. Such as removing the set of words that appear in all the text files, excluding meaningless punctuations, removing stop-words, retaining multi-word phrases for characters and locations as provided in the .csv files, and more. It was not easy to execute these methods, but the general strategy for this is eliminating texts that does not have any useful information.

First of all, the names of locations and names in the given .csv files were stored as multi-word index terms, as these terms are very likely to become candidates for queries. This would preserve any stop words in names, such as ‘Bureau of Engraving and Printing’. For such types of names and locations, it is most likely to search for the exact matches. However, this makes the construction of the index complicated, this is one of the reasons to create index for single words and it will be discussed in following sections. Considering stop-words, nltk.stopwords.words(’english’) has been used to exclude stop-words from the list of index terms. It was a simple decision, since stop-words do not have significant meaning by themselves.

According to Luhn’s hypothesis, removing any terms that had a frequency below a certain threshold was considered once, but this idea has been rejected due to certain key words that are mentioned only once in the corpus. For example, the ‘couch gag’ appears only once in the opening scene of the show, since it is one of the most memorable moments of the show from each episode, there is a high chance for someone to look for it. This indicates that the one of the lowest occurring terms must be stored.

Next, in terms of non-alphabetical words, the numeric values were not removed for similar reasons to low-frequency words. Some of the numbers in this show could represent specific time or date that has meanings to a certain episode. For example, the number ‘2525’ is part of a reference to the song ‘In the Year 2525’ by Zager and Evans. In addition, each episode has a unique 4 letters of alphanumeric production code. Therefore, these terms are also kept as index terms in the inverted index.

In addition to non-alphabetical words, symbols such as ‘ # and – are the exceptions from removing. For instance, apostrophe can be genitive markers or enclitics and hyphen can be lexical hyphen. All of these examples are should be saved as index terms because some words without them will lose or have different meanings. # has a special role in this particular situation. Because some of the locations and names given from .csv files have # and numbering that makes a difference between each entity. Thus, these symbols are stored in the inverted index.

In conclusion, multi-word terms that are present in both the corpus and .csv files and single words that are not stop-words and present in the corpus has been chosen as index terms. Since there are two types of terms stored in the inverted index the volume of the storge is large and time cost both for constructing the index and searching has increased. Nevertheless, in my opinion, these decisions have conserved critical functionality of the inverted index. Elimination of other terms would have led to disturbing the search of relevant articles. Also, the time consumption of the construction of this version of inverted index is about 15 seconds which is an acceptable for given task. So, I believe that reasonable decisions have been made for choosing the indices for inverted index.

**Pre-processing Choices**

Pre-processing is essential and important, because after this, terms of the inverted index will be fixed. So, in order to reduce its size while preserving the strength, many actions has been taken.

Firstly, as much repetitive and meaningless phrases in the corpus were deleted before the tokenization phase. Phrases such as ‘From Wikipedia, the free encyclopedia’ were in all of the .txt files. Storing such kind of set of words would be a waste of time and storage. Phrases closed in square brackets were manually identified by going through more than 15 .txt files and found out that they were repeated in all of them without specifying anything to the documents. These measures did reduce the amount of storage needed at this stage.

Furthermore, tabs and any other symbols besides the exceptions that have been referred previously are replaced with whitespaces to prevent problems with formatting. Then the POS tagger has been used later for lemmatization. However, I have done some experiments and found that the nltk POS tagger did not show any meaningful difference in performance whether terms had punctuations or symbols. On the one hand, lemmatizing before removing symbols led to an insignificant reduction in index size, but a significant increase in time creating the index. deleting the symbols at this step before the process of tokenizing has led to cheapening the time cost.

The whole text has converted to lowercase at this phase. This enables search terms to be case insensitive which will provide correct answers regardless of minor capitalization errors.

The processed text is transformed in to single-word tokens by using TweetTokenizer. This tokenizer maintains symbols attached to their main word creating single word tokens. After this, a copy of this list of tokens is converted into a mix of multi-word and single-word tokens through the use of WordNetLemmatizer and MWETokenizer which is initialized using patterns from the .csv files. There are actually stemming method and lemmatization method. Stemming is a faster process than lemmatization as stemming chops off the word irrespective of the context, whereas the latter is context-dependent and also stemming is a rule-based approach, whereas lemmatization is a canonical dictionary-based approach. Lemmatization has higher accuracy than stemming.

The index was about 18,000 terms large before choosing lemmatization to perform on the tokens. When stemming was done, there was an increasing recall while precision got lower. Given that our index is large and it is very likely to contain too many relevant terms for any query, it would be a better choice to focus on improving precision for search queries. Thus, lemmatization was chosen over stemming. Doing so reduced the size of the index from around 18,000 words to 13,088 words, resulting in larger than 27% reduction in size. However, the average index construction time has increased from 3.1 seconds to 14.2 seconds. However, this was an acceptable arrangement as the indices are accessed more often than it was, so the 11.1 seconds increase in build time is quite negligible in the long run, and it is worth the quality increase in user experience.

For making a decision on which one to do first between doing lemmatization or removing punctuation, both of the methods has been executed. When lemmatization was firstly done, the result has shown that the total number of words in inverted index is 13,104, which is similar to the other method. However, in terms of time taken, this method has taken 16 seconds to be executed. So the latter method have been chosen.

Finally, stop-words are transformed into placeholder tokens instead of being removing. Stop words were eliminated for the reasons mentioned above, but the placeholder tokens were added so that the positions of each index term in the corpus would be preserved. This creates a drawback of increasing storage cost, proximity search becomes considerably more useful since users may base their queries on results that are more closely related to the original, unprocessed corpus. Additionally, as they are duplicate entries with terms from the list of single-word terms, any single-word tokens contained in the list of multi-word tokens are simply disregarded when generating the inverted index.

**What is Stored in the Inverted Index**

In order to benefit from the inverted index's effectiveness when retrieving objects under certain indices, it have been set as a dictionary. The defualtdict has been used for simpler handlings of terms. However, there were problem that defaultdict would add the entry that does not exist inside itself. This would cause the defaultdict's size to rapidly increase over time, which is not helping. Additionally, due to the nature of the dictionary data structure, all new entries must be initialised with the appropriate datatype, requiring a check to see if a word has previously been added or not. This adds a little complexity to the implementation but has little effect on how long it takes to build an index. All indices in the inverted index are keys with matching value of 2 item list. The first item of the list is the number of the document containing the term which is known as “document frequency”. The second item is a dictionary with names of files as keys and a list. There are two types of items containing the list. The first is the number of index term occurring in the file. The second is a list of positions of the term in current file.

Even though it is simple to calculate, the number of items in a dictionary or list is stored separately, since doing so can save time when it comes time to rank any entries according to frequency. The document frequency is kept in the first "layer," which may be used to determine which index terms are most frequently used in the corpus. This may be used, for instance, to compile, using the document frequencies of the characters, how frequently each character appears in a specific collection of episodes. The second "layer" records the word frequency of a given phrase within a given document, which is more visibly helpful by ranking papers that include a certain term in based on occurrence, and demonstrating how essential the term in each document. An article that says "Homer" ten times, for example, is considerably more likely to be centred on him than one that does so only once. The amount of storage space needed to hold the index does, however, rise as a result.

Instead of a shortened file id, file names are used as keys in the dictionary that is maintained under each index term. Although it is debatable if using integers that are uniquely mapped to each file name would be more efficient, file names were utilised as keys in this case to improve the readability of the dump function's results. One may argue that the underlying structure of the index might not be accessible to the user at all in a typical situation when a fully featured front-end is offered, negating the necessity for it to be readable. However, for the sake of this coursework, I have opted to directly utilise file names as keys to avoid the bother of the marker having to understand a list of a lot of sub-4-digit integers. In this scenario, I would use a collection of integers mapped to file names as keys instead.

This positional index gives additional significance in the proximity search application, which will be covered in more detail in following parts, and can also more easily be transformed into an actual position in the original corpus. This does require our implementation to keep a large number of placeholder tokens, but some of this is alleviated by utilising cumulative offsets to determine the placements of multi-word words. In order to reduce some computation time, we also simply skip all instances of placeholder tokens while creating the index. Because all punctuation should be erased at this time, the presence of angled brackets can only indicate that the token is a placeholder token that represents a stop word that has been deleted.

**Proximity Search**

Using two search terms and a window size, the proximity search function looks for any instances of the two terms in the same document inside the chosen window size. Any matches that are found are entered into a dictionary that is constructed similarly to the inverted index and are then returned.

The definition of the window size must be specified before discussing any of the implementation of the function.

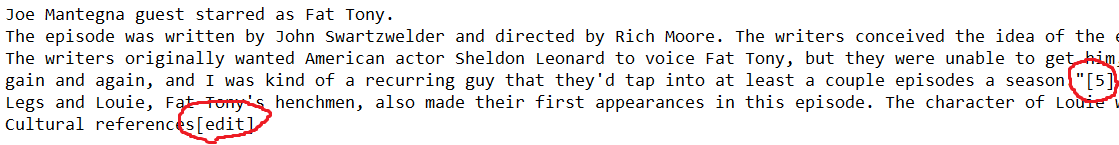
The phrase "window size" in the context of proximity search refers to "the interval in which all the search terms should occur" or "the width of the smallest window in a document that includes all the query items." Since my approach adheres to this definition of window size, I anticipate a window size of at least 2. However, it does technically work as it was intended when given smaller window sizes. Similar to the dump function, the two search terms entered into this function go through the same pre-processing as the tokens taken from the corpus. In actuality, it utilises the lemmatize terms() method. The proximity search function first verifies if both phrases are present in the index following pre-processing, and if they are not, it returns no results. Thus, exceptions are avoided. The function then scans all positional indices under each term's corresponding file name for any document file names that could exist under both search terms in the index.

The function then calculates the difference in the positional index of each term after iterating over each index for each of the two terms. The two phrases are contained inside the size of the window if the difference is smaller than the window size that has been selected. In this instance, the two terms' positions are noted as a pair and entered into the dictionary.

As an alternative to indexing multi-word terms, I think proximity search can absolutely enable looking for relevant items. In addition to requiring a predetermined list of multi-word words to search for, indexing multi-word terms makes it more difficult to generate the inverted index. Additionally, not all entities of relevance may be included in the pre-defined list of multi-word terms, such as "Gordy Howe," whose name is not referenced anywhere in the.csv files despite making many appearances in the programme. As the user can also find results when looking for a subset of the multi-word phrase, proximity search offers greater flexibility than indexing multi-word terms.

This will enable users to find the multi-word terms without needing to limit their search to precise matches, as my dump function presently does.

Appendix

[Figure 1] 

Stemming and Lemmatization comparison

<https://stats.stackexchange.com/questions/523066/should-stemming-and-lemmatization-both-be-used-together-or-not-what-is-best-pra>

Luhn’s hypothesis(idea)

https://www.dcs.gla.ac.uk/Keith/Chapter.2/Ch.2.html