

The following documentation explains the algorithms that were used while constructing different python programs through which tasks like inverted indexing and vector space retrieval were made possible. These are given in full details below -

1. **invidx_cons.py** –

This python program helps in the construction of two files namely *indexfile.dict* and *indexfile.idx*. Both of these files are binary files and these are saved with the help of the library pickle, by using the command `pickle.dump()`. The *indexfile.dict* is a dictionary (unordered) data structure containing keys as words present in the documents and the values being the posting lists corresponding to each word. The *indexfile.idx* is a nested dictionary (unordered) data structure containing the keys as document IDs and values as dictionaries (each containing keys as words present in the corresponding document and values as term frequency, *f*, of the word).

Each file that is present in the directory and contains documents is pre-processed in order to change it into a xml file. This is done by adding "<root>" tag at the start and "</root>" tag at the end of each file, and by the deletion of symbols - ["&", "\"", "\""] present in file text. This process generates a new set of files without altering/deleting the previous files.

The xml files are read using the `xml.etree.ElementTree.fromstringlist()` command and its root is stored. The children of the root are retrieved using the `root.getchildren()` command. As each file contains many documents with each document having its own document ID and the text under <DOCNO> and <TEXT> tags respectively, the program iterates over each document and stores its document ID and the text. As the text is a large string also consisting of named entities, the program splits the string on the basis of space between the words and these words are then stored in an array. If the word is other than <ORGANIZATION>, <ORGANISATION>, <LOCATION>, <PERSON>, </ORGANIZATION>, </ORGANISATION>, </LOCATION> and </PERSON> then all the punctuations that are part of the word are removed before storing it or if the word contains any of the above given tags in itself, then the tag part of the word is only stored (this helps in establishing a uniformity in order to store the named entities separately in the future, for eg. - "</ORGANISATION>," will be stored as "<ORGANIZATION>").

The next step is to form both the dictionaries (*indexfile.dict* and *indexfile.idx*). The program iterates through the list containing the words. If the word is not in the sets (data structure) ["<ORGANIZATION>", "<LOCATION>", "<PERSON>"] and ["</ORGANIZATION>", "</LOCATION>", "</PERSON>"], then it is checked if it is already present in the dictionary or not and if it is not present then a set is created corresponding to the word that contains the document ID for the document which is currently in the focus. If the word is already present in the dictionary then the document ID is added to its set. Also, the document's (which is in focus at present) document ID has been stored as a key in the *indexfile.idx* dictionary and corresponding to it a dictionary has been initiated which contains the word and its frequency as a key-value pair. If the word is already present in *indexfile.idx*[document ID].keys() then its value is increased by 1 otherwise it is added as a key with the initial value of 1. Also, if it happens that the word is a part of the set ["<ORGANIZATION>", "<LOCATION>", "<PERSON>"], then all the text between this word and the corresponding closing tag is saved

in a similar way (explained) to the dictionaries indexfile.dict and indexfile.idx. Along with this, if the next word after the closing tag is present in the set [“<ORGANIZATION>”, “<LOCATION>”, “<PERSON>”] and is also equal to the previous tag then the text between the new tag is stored individually as well as in combination with the text that was in between the previous tag. For eg. – if we have “<ORGANIZATION> Vietnam </ORGANIZATION>”, “<ORGANIZATION> News </ORGANIZATION>” and “<ORGANIZATION> Agency </ORGANIZATION>”, then the indexfile.dict and indexfile.idx will contain “vietnam”, “news”, “agency”, “vietnam news agency”. Please note that the each word is stored in its lower case form.

2. printdict.py –

In order to read the content of indexfile.dict in human readable format (“<indexterm>:<df>:<offset-to-its-postingslist-in-idx-file>”) this function first sorts the dictionary keys and iterates over the sorted dictionary keys outputting the indexterm as the key itself, df as the length of its postings list and offset as the index position in the sorted dictionary/sorted dictionary key list.

3. vecsearch.py –

This program implements the vector space retrieval model by taking in the parameters as the path to the queryfile, top k cut off, name/path to the resultfile, indexfile.idx and indexfile.dict.

In order to measure the similarity (sim) between a document and query we convert the document and query to N dimensional vectors, where N is number of unique words or the length of the dictionary indexfile.dict, and take the dot product of both the vectors. Through the help of dot product we compute the value of $\cos\theta$, where θ is the angle between query vector and the document vector. A query is more similar to the document if θ is small or if $\cos\theta$ is large.

Let the query vector be represented by \mathbf{q} and document vector be represented by \mathbf{d} , then

$$\cos\theta = \frac{\mathbf{q} \cdot \mathbf{d}}{|\mathbf{q}| \cdot |\mathbf{d}|}$$

The program first computes a dictionary (unordered) that contains the magnitude of each document as its value and document id as the key. To assign values(weights) to each dimension(term/word) of a document vector we find the term frequency f of each word and its inverted term frequency. The role of indexfile.idx comes into play here as the keys are document IDs and values contain dictionaries with keys being terms/words of the document and values being the term frequency f. We iterate over every document ID present in the indexfile.idx and for that document’s terms we compute term frequency(tf) = $1 + \log_2(f)$ and inverted document frequency(idf) = $\log_2(1 + n/df)$, where n = total number of documents or the length of indexfile.idx.keys() and df is the length of the postings list for term/word which can be calculated with the help of indexfile.dict. For every term of the document square of tf*idf is computed and all these values are added together. The square root of this sum is equal to the magnitude of document vector.

The next part is to process the queries. As the queryfile also contains query number or query ID under the "<num>" tag and query text under the "<title>" tag, the file is iterated and the query IDs are stored in a python list and query texts are stored in another python list. The query texts are now pre-processed before the computation of sim as the query texts contain named entities as well as prefix terms. The named entities and words other than prefixes in a query are stored in a similar way in which the named entities and words in the documents were stored. For eg.- "O: Vietnam", "O: News", "O: Agency" are stored as "vietnam", "news", "agency", "vietnam news agency". The punctuations in between the texts are removed and these are finally stored in a set. Also, in order to recognize a prefix term, we search if the word[-1] == "*" or not and if this is true then all the words in sorted(indexfile.dict.keys()) containing the word as prefix are searched and added in place of the prefix word to the set containing the terms of the query. Please note that these words are stored in the lower case form. While doing this a dictionary is also constructed containing the keys as the terms of the query and values as the term frequency of the word in the query itself.

Please also note that the idf value for the words in the query will be the same as $\log_2(1 + n/df)$ if the word is present in the indexfile.dict otherwise it will be 0. The only difference is the value of tf as now f in the formula of tf will be equal to the frequency of occurrence of the word in the query but not the document with which it is being compared.

Now as the program has pre-processed all the terms of the query and stored them into a set as well as stored their term frequency in the query in a dictionary, the next part is to compute sim values or $\cos\theta$.

We initiate a new dictionary to store the dot product of query vector and the document vector.

The program iterates over each word of the set (containing terms of the query) and finds its postings list. For each document in the postings list if the document ID is not present in the new dictionary then for the key document ID the product of tf-idf of the query term and tf-idf of the same term in the document is stored. If the document is already present in the dictionary the product is added to the previous value. The same process is repeated for all the words in the set. Also during the whole process the magnitude of the query vector is also computed by adding the squared values of tf-idf of each query term. After the completion of the iteration, the square root of the total sum gives the magnitude of the query vector.

After the completion of the above process the values of the new dictionary containing the dot product are divided by the product of the corresponding document's magnitude and query's magnitude. The resultant dictionary contains the document IDs as keys and sim as the values. This dictionary is sorted in reverse order and the top k document with the highest scores are written into the resultfile.

Following the above procedures, the wall running clock time of the `invidx_cons.py` program is 162.67

```
In [5]: start = time.time()
invidx_cons("col764_a1\docs1_a1", "col764_a1\indexfile")
end = time.time()
print(f"Runtime of the program is {end - start}")

C:\Users\titan\Downloads\anaconda\lib\site-packages\ipykernel_launcher.py:42: DeprecationWarning: This method will be removed in future versions. Use 'list(elem)' or iteration over elem instead.

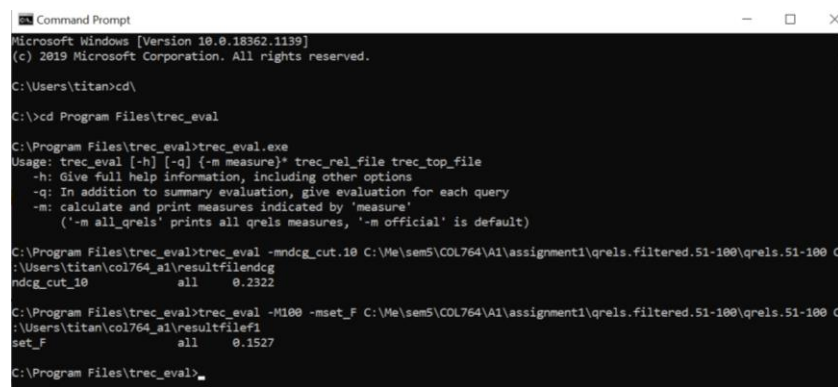
Runtime of the program is 162.6727774143219
```

The `indexfile.dict` contains 467793 unique words as keys and `indexfile.idx` contains 81946 unique document IDs as keys.

Also, the size of the `indexfile.dict` is 108MB and the size of `indexfile.idx` is 321MB.

The `ndcg` value obtained for $k = 10$ is 0.2322

The `F1` score obtained for $k = 100$ is 0.1527



```
Microsoft Windows [Version 10.0.18362.1139]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\titan>cd\
C:\>cd Program Files\trec_eval
C:\Program Files\trec_eval>trec_eval.exe
Usage: trec_eval [-h] [-q] [-m measure]* trec_rel_file trec_top_file
  -h: Give full help information, including other options
  -q: In addition to summary evaluation, give evaluation for each query
  -m: calculate and print measures indicated by 'measure'
      ('-m all_qrels' prints all qrels measures, '-m official' is default)

C:\Program Files\trec_eval>trec_eval -mndcg_cut.10 C:\Me\sem5\C0L764\A1\assignment1\qrels.filtered.51-100\qrels.51-100 C:\Users\titan\col764_a1\resultfile.ndcg
ndcg_cut_10      all      0.2322

C:\Program Files\trec_eval>trec_eval -M100 -mset_F C:\Me\sem5\C0L764\A1\assignment1\qrels.filtered.51-100\qrels.51-100 C:\Users\titan\col764_a1\resultfile.f1
set_F           all      0.1527

C:\Program Files\trec_eval>
```