**SET HW1**

The search engine works on a local corpus- in this case the Cranfield corpus. It builds an inverted index and returns the relevant documents for a user entered query.

The basic flow of the program is as follows:

1. mySearchEngine.py

* This script acts like a driver flow, calling the functions from other scripts to perform the functionalities as required.
* This script takes the path which will contain the corpus as an input from the user.
* Given this path, it calls the parsing function followed by the indexing and querying functions.

1. parsing.parse\_file()

* This function uses the ‘xml.dom.minidom’ module to parse the xml file and extract relevant data from it.
* Task1: It creates an intermediate directory to store the extracted text files i.e files containing just the data under <TEXT> element.
* Also for each cranfield file, this function extracts the title information and stores it in a separate metadata dictionary. It can support any future requests to store any other metadata too.
* Pickle module is used to dump this dictionary which is used by other functions as desired.

1. indexing.create\_index()

* This function takes all the extracted text files from the intermediate directory and creates an inverted index for the corpus.
* It does tokenization on word boundaries.
* Converts all words to lower-case
* Performs stemming of the words using the nltk PorterStemmer function. (This reduces the index length by around 2500-3000)
* This is followed by inserting each word into the index along with information such as the document id and the positions it appears in.
* Lastly, it remove the stopwords from the index. (This reduces the index length by around 150)
* This index is then pickled for use by other functions.

1. querying.get\_query()

* This function returns the results based on the request type.
* A query is taken from the user and based on the request type, it is routed to one of the functions which extracts metadata or extracts similar words or gets statistics or retrieves the documents.
* To retrieve documents pertaining to the query, a ‘pre\_process\_query’ function is called to understand if the query contains words or phrases or negated phrases or negated words and extract the same.
* The get\_documents function creates a dictionary which is updated when the required word or phrase appears in the corpus. The scores are also calculated based on the number of occurrences and stored.
* Separate dictionaries are maintained for the negated phrases and negated words.
* A fuzzy-or operation is applied as per the requirements. E.g. query- cat !dog

The documents retrieved will consist of the documents containing cat union documents not containing dog. If there are documents containing both cat and dog, the search engine returns this common set. Intuitively, it may be thought that it should not return this common set of document as the user has requested not to see documents containing dog. But as the requirements suggest a fuzzy-or operation, it returns this common set.

* This function also supports finding phrases for multiple length words(more than 5 possible). It stores the documents containing those phrases and their start positions.
* Detailed comments in the script.
* The ranking function is called next .

1. ranking.ranking()

* This function takes the dictionaries storing the result docids and positions and ranks them based on the scores as per the requirement.
* Each result is then printed as- docid \t snippet. The snippet is extracted by reading each text file from the intermediate directory and printing the (n=8) words preceding and following the word/phrase positions as calculated in the querying function.
* For docids which indicate the documents not containing the negated word/phrases, only the docid is printed.

1. Similarity

* Query for similarity is supported for words. The logic behind implementing similarity is based on context. I assume that for a word, if we derive it’s context through preceding and following words, we can find similar words using this contextual information. Implementation is as follows-
* For the query word, get the words that appear next to it i.e. preceding and following it in the corpus. So for every occurrence of w1 query-word w2, extract the phrase ‘w1 query-word w2’. For every such phrase, check the corpus for words that have the same words w1, w2 preceding and following it. Output will be the set of these words such that w1 word w2 occurs in the corpus.

1. Some brief information about the implementation:
2. User needs to enter the path for the corpus.
3. As stemming is done on the index, all query words are stemmed before processing.
4. ‘tf’ request type only supports the word requests.
5. ‘df’ request type supports both word and phrase requests.
6. ‘freq’ request type supports both word and phrase requests.