**AIR assignment 1 report**

**Name** : Harsha Vardhan M A

**USN** : 1PI14IS017

**Introduction**

This is the report of a submission that attempts to build an Information Retrieval system using two techniques namely, Boolean retrieval model and Vector Space model. I intend to give out the documentation and analysis of the programs written formally as a part of this report. The following content of the report is organised as, Implementation details of both the models built which explains the nuances of both the programs and the justifications for the type of implementation used. This is followed by results and observations of the model, then by performance details later by references pertinent to the programs.

**Implementation**

*Boolean retrieval model* :

To implement the Boolean retrieval model, an inverted index was created using the dictionary data structure of Python with key being a term (tokenised word after normalisation from the data files), and value being a set of file names in which the term exists. Every file is traversed word by word in this process of index creation. For every new word encountered, a new dictionary <term, set> pair is added. If the term is already seen and if it is revisited, the file name in which it is present is added to the set of the term in the index dictionary. This dictionary is later used in handling of Boolean queries. Note that before adding a word to the dictionary, it is tokenised using word\_tokenize of nltk and is stemmed using PorterStemmer of nltk. Norving’s spell checker is used in Boolean Retrieval model as a part of normalisation. Query terms are stemmed as well before query processing. Any quotations before and after the query term or the index term is also removed as a part of normalisation.

Once the dictionary is ready, it can be used for handling Boolean queries. In this approach, I handled queries left to right (though initially I was doing it disjunctively, but since the code was becoming unnecessarily complex after introducing NOT operator, and since both AND and OR have same precedence, I changed the implementation to left-right query processing)

The algorithm for processing the Boolean queries is briefly explained below:

For each query in the query file, do the following:

1. Have a temporary set (name it tempest)
2. Have a set of operators (name it opers, whose contents will be “AND”, “OR” and “NOT”)
3. Populate the tempset initially with the set contents of the first term in the query (obtained from the index dictionary, provided that the first term is not in opers set)

Handle various cases like query starting with a NOT etc

1. Start a loop from the first AND or OR operator till the end of the query string. Intersect ,for AND or union ,for OR (after any NOTs, present) the terms in the loop traversal with the tempest contents and put the result back into the tempest.
2. Print the final answer which is in tempest

Note that , NOT is performed using the set difference operator in python. NOTs are done before unions or intersects are performed to maintain the correct arity of the operators (NOT is a unary operator while the other two are binary)

*Vector Space model:*

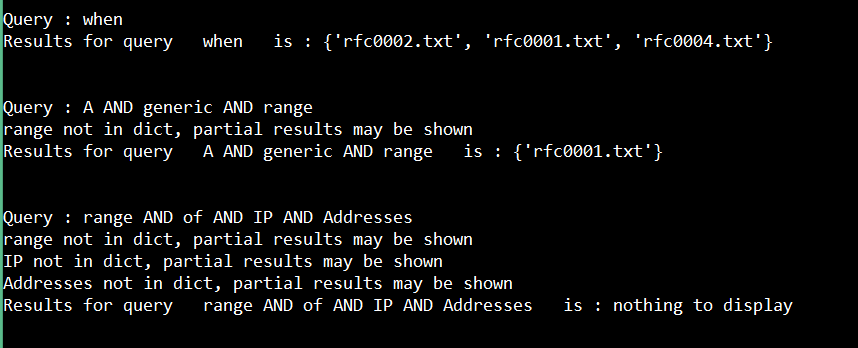
This is a model where the query can be a free flowing text rather than having some predefined format of operators and operands. Tf-idf score for each document is calculated (including the query file) and this results in a vector of numbers which has number of dimensions equal to the number of words in the document. Tf and df (document frequency which is later used for idf) can be calculated in one traversal across files in the data folder. This is made possible by having two dictionaries which has key,value pair equal to <filename,<term,tf>> and <term,df> for tf dictionary and df dictionary respectively. Since document frequency is a collective entity, the dictionary of df of a term is calculated incrementally as and when the document folder is parsed where as tf being an individual entity, it is calculated in each iteration. Once both the dictionaries are populated, tf idf product scores are calculated for each term with in a document , which eventually comprises a document vector. These tf-idf scores for each document is also stored as a dictionary. Once document vectors of both the query and documents are calculated, cosine similarity of query file vector is calculated with every other document vector and top k (here 10) values of cosine similarity is considered and their corresponding file names are shown. Cosine similarity calculation is done manually, without libraries.

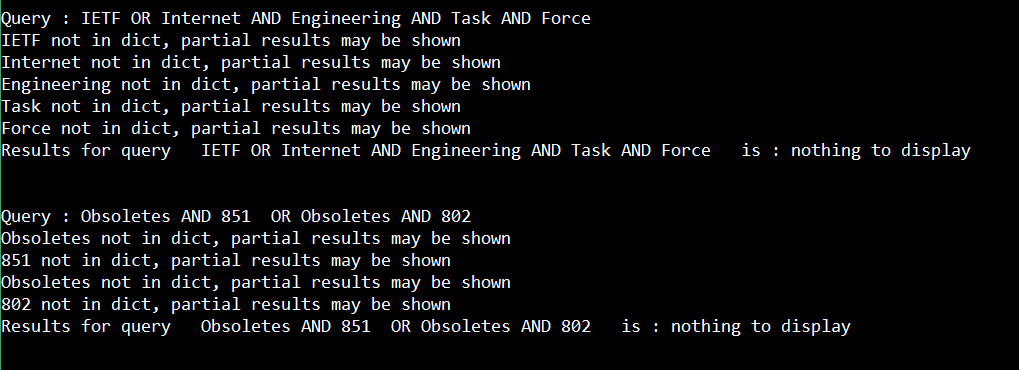
Porter stemmer of nltk is used.

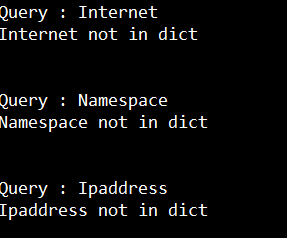
Both the models can be run as follows : python filename.py <path to docs folder> <path to query file>

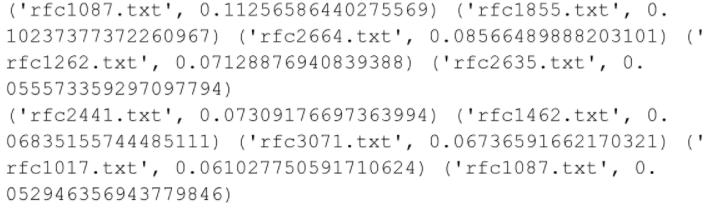
**Results and observations**

Results of the Boolean model is shown in form of snapshots. Varying outputs are shown. Documents folder being the rfcs dataset.

****

****

****

Results of Vector space model 

**Performance**

For running both the models using the rfcs dataset, Vector space retrieval model took almost 50 minutes for vector calculations and 2 minutes for querying, and I did not have much time (insufficient RAM maybe) to run the Boolean model for the entire rfcs dataset. For 20 files, the index creation and querying(10 queries) took approximately 3 seconds.

System specifications: Intel i5 4th gen, 8GB DDR3 RAM, Windows 8.1

**References**

Stemmers - <http://www.nltk.org/howto/stem.html>

Basic python tutorial - <https://youtu.be/YYXdXT2l-Gg>

Python dictionary specifics - <https://www.tutorialspoint.com/python/python_dictionary.htm>

Spell checker - <https://github.com/mattalcock/blog/blob/master/2012/12/5/python-spell-checker.rst>