COM 4115/6115 Text Processing (2016/17)
Assignment: Document Retrieval
Registration Number: 160122440

1 INTRODUCTION

A document retrieval system is defined as the matching of some queries chosen by a user to a set of free-text words. The aim of this assignment was to create such a system, based on an inverted index which was created from the collection of documents provided. The retrieval system supports both conjunctive boolean retrieval and ranked retrieval and it also allows the user to pass new queries to the system. The design and implementation of the system was developed under the guidelines provided in the assignment description handout. All the required tasks have been completed and will be explained in the following sections of this report.

2 DESIGN AND IMPLEMENTATION

First of all, the system was designed in two separate programs; one for indexing and the other for document retrieval. Even though there is some common functionality required by both programs (such as tokenisation and stemming), and it is also better to avoid redundancy in the code, it was still preferable to create two separate programs for two reasons. Firstly, all real-world document retrieval systems have the inverted index stored in a conventional format and load it when required; instead of computing it every time a document needs to be retrieved. So as an intention to illustrate a real-world document retrieval system, one program was developed to create the inverted index and another to perform the retrieval. The second reason for this decision was in order to satisfy the principal of separation of Concerns; which states that a computer program is separated into distinct sections, each addressing a separate concern.

2.1 Indexing

The first program developed (*indexing.py*) is for the construction of the inverted index. The inverted index was created using two of the files provided; the first one being the collection of documents (*documents.txt*) and the second one being the stop list (*stop_list.txt*), which contains all the words that must not be included in the index. The steps followed for the creation of the inverted index are explained below:

- **Tokenise** the content in the documents file using the NTLK's word tokenizer.
- **Normalise** all the words (terms collected from the tokenise step) to lower case.
- Ignore all the terms that appear in the **stop list** file.
- **Stem** all the remaining terms using Porter Stemmer.
- Store terms, document IDs and counters in a **two level dictionary** to create the inverted index.
- Create a **text file** containing the inverted index (*inverted_index.txt*).

The inverted index was created as a two-level dictionary in order to be able to store terms, document IDs and counters in an efficient and accessible format. The outer dictionary maps terms to the embedded dictionary which contains document IDs as keys and the corresponding counts as values.

2.2 Retrieval

The second program (*retrieval.py*) is responsible for reading the inverted index, and for each query (in the query text file *queries.txt*) to retrieve the documents relevant with that query. This retrieval can be done either using a boolean retrieval or a ranked retrieval. The user can also pass new queries by typing new queries in the command line.

Queries must also be preprocessed before relevant documents must be retrieved. This is done so that query terms are of the same manner as document terms, in order to be able to compare them. Queries are preprocessed using the same steps as for documents, described in section 2.1. Queries are first tokenised, then normalised, then the stop list is used to ignore the stop words and finally tokens are stemmed using the Porter Stemmer. Queries are then stored in a normal dictionary with keys being the queries IDs and values being all the tokens in the query.

2.3 Similarity

The next step is the retrieval of relevant documents. As mentioned above, the system allows both boolean and ranked document retrieval. The main aim of this assignment is Ranked retrieval therefore Boolean retrieval will not be further explained. Additionally, Boolean retrieval code has been commented out in the program and you can see in the *README* file how to find and use it if needed.

The first step in ranked retrieval is to measure the frequency of document terms in order to be able to measure the similarity between a particular document and a given query. Also, it is not necessary for all the query terms to be present in the document, since it is a ranked similarity measurement. A vector space model was implement in order to measure the similarities, and the documents with the highest similarity appear first. In the vector space model each document is a point in a high-dimensional vector space and each term within the index is one dimension; also using vector-notation to represent the queries. Furthermore, in Vector Space Model the similarity is calculated from the cosine angle between the two vectors. Several parameters needed to be calculated first, which are explained in the steps that follow (most parameters are found from the index created):

- $|D|$: the number of documents in the collection, by gathering together the full set of document identifies for the collection.
- *dfw*: the document frequency for each term, found from the inverted index (keys of embedded dictionary)
- $log(|D|/dfw)$: the inverse doc frequency of each term, found by division of the first two parameters.
- *tfw*: term frequency, in order to calculate *tf.idf* (values of embedded dictionary are the counters)

Finally, the cosine angle is calculated from:

$$sim(\overrightarrow{q}, \overrightarrow{d}) = cos(\overrightarrow{q}, \overrightarrow{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} {q_i}^2} \sqrt{\sum_{i=1}^{n} {d_i}^2}}$$

Where $\sqrt{\sum_{i=1}^{n} {d_i}^2}$ is the size of each document vector, and $d = tf.idf$

The component $\sqrt{\sum_{i=1}^{n} {q_i}^2}$ is constant for a given query and so it can be dropped without affecting the ranking. Therefore, for the denominator, only the sizes of the non-zero document vectors were considered. For the numerator, only the terms that appear in the query were included in the calculation; since the terms that are not in the query will have a term frequency of zero. Finally, the results of similarity calculations are stored in a descending order (only the 10 highest ranked documents) in a text file called *similarities_calculations.txt*.

3 DISCUSSION AND EVALUATION

The performance of the system has been evaluated using the script provided (*eval_ir.py*) along with the (*cacm_gold_std.txt*). Results obtained were Precision: 0.25, Recall: 0.20, Relevance: 0.23. Relevance evaluation is based on various factors such as effectiveness, ease of use, response time and form of presentation which will be discussed below. First of all, alterations of term manipulations were examined as shown in the table below:

stop list	stemming	capitalisation	**Precission**	**Recall**	**Relevance**
Yes	Yes	Yes	**0.25**	**0.20**	**0.23**
No	Yes	Yes	0.25	0.20	0.23
Yes	No	Yes	0.19	0.15	0.17
Yes	Yes	No	0.20	0.16	0.18

Table 1: Term manipulation alterations to evaluate the accuracy of the system.

As shown in table 1, the best accuracy was obtained when all term manipulation techniques were used. Even though excluding the stop list did not affect the accuracy, it increased the response time; since more terms had to be processed.

Moving on, the response time of the system is extremely impressive, as it can process all 64 queries from *queries.txt* in just 2.5 to 3 seconds. Also, response time for new queries happens almost instantaneously.

The system is also user friendly and can be used by non-expert programmers. A help option is available to explain how to run the system, which only requires a few parameters to be typed. The user can also pass in a new query which will be processed instantly and similarity results will be printed in a presentable format. Additionally, both the inverted index and similarity results are printed in a text file, allowing the user to examine the results at any time without having to run the code again.

4 CONCLUSION

In document retrieval systems there is no perfect solution on how the system retrieves the relevance. Some systems might be better than other in certain tasks, but have a disadvantage in other areas. For example, in our system we are removing capitalisation, which will provide a higher similarity measure for general words, but will have certain disadvantages like loosing acronyms. Every alteration might have a positive or a negative impact. Concluding, each retrieval system must be designed in such a way as to support the task it was destined for; after implementation of several configurations to examine which provides the best outcome.