Text Processing

Document Retrieval

Diego Alejandro Cerda Contreras

DESCRIPTION

The objective of this IR System is, given an information necessity and a set of documents or collection, to sort this documents in descending order by relevance and present the information to the user. This Information Retrieval system is a Python program based on the vector space model.

This model measures the distance between a document and a query/user request. The system considers a term as the defining characteristic of a document. Binary weighing and frequency weighing are used in this model to compute the angle between a set of words. Several configurations may be used as well such as Stemming and Stopwords. Finally, the data is presented in console and/or in a text file containing the top 10 search result for each query.

IMPLEMENTATION

The first task in this project was to clean up and preprocess the data inside each document. For this task we use the *ReadDocuments* script provided in the assignment material. This script happens to be very useful because it allowed us to retrieve every line inside every document by its *doc.docid.* Once we were able to get this line information the next task was to separate each word, tokenization, from each line with a Regex expression to find the maximal alphabetic sequence.

Creating the inverted Index: A two level dictionary was defined to store every word inside the collection. The first level contains each word and the second level contains the documents with its term frequency. *(Code: def indexingInvertedIndex*). This is where the *Stemming* and the *Stoplist* are processed. There are also two more methods that creates a two level dictionary, one for the manual search and a second for the query, however this could be simplified to avoid redundancy. The final result is a variable that contains every word and T.F. (term frequency). This method contains a total of three nested loops which gives a complexity of ***O(n3)*** that's why is very important to keep it efficient. To avoid this complexity an export option is provided. A *Json* format file is produced if enabled to store the dictionary in an external file, if this option is disabled then it will compute the inverted index and store it in the specified file (Refer: README.txt).

Once we got the inverted index now we can continue to find which terms are contained in the two set of words (query and document), in other words, we need to find the intersection of the two sets. This intersection is used to reduce the amount of calculations that the system has to compute, we achieve this with a specific method *(Code: def booleanRetreival)* that represents a simple Boolean Retrieval model.

For the rest of the implementation a divider and conquer approach was implemented. The methods used are listed below:

- `inverseDocumentFrequency`: A word may appear more than once in the same document, and some words may also have more relevance than others that's why it is also important to normalize the vectors so we avoid privileging words against small and large documents. By doing the IDF we make sure that a word is relevant for the document in the collection. This measure is often used as a weighting factor. This method compares the number of all available documents with the number of documents containing the term. And finally, the logarithm is responsible for compressing the results.
- `termWeighting`: Determines the relative frequency of a specific term. This method contains both Binary and Frequency approach. If option '-b' is enabled it will compute for T.F. binary values, this is done by given a value of 1 is the word appeared in the document, otherwise this method will return how many times a word repeats itself.
- `getTotalDocumentsCollection`: Iterates through the collection and return the total documents.
- `documentSize`: This method represents sum of squared weights for terms appearing in the document and returns a dictionary with the document ID and its size. This method was implemented to improve processing efficiency.
- `Qidi`: The value increases proportionally to the number of times a word appears in the document, but it is compensated for by the frequency of the word in the document collection, allowing to handle the fact that some words are generally more common than others. This method finally returns a dictionary of each document ID and its TF.IDF value.

Form the previous methods now we are able to compute the Cosine of every document and print the result. We store the result in a new dictionary and implement a Sorted method to loop for the first 10 results and present the data.

RESULTS

To evaluate our system we use the Gold Standard data against the Result Example provided in the assignment material and the results computed by our own IR System. In all test cases we got the following number of documents over every query:

Retrieved:	796
Relevant:	640

BINARY

Total Words in Index	**Stemming**	**StopWords**	**Rel_Retr**	**Precision**	**Recall**	**F-measure**
6963	YES	YES	122	0.19	0.15	0.17
7218	YES	NO	122	0.19	0.15	0.17
10356	NO	YES	99	0.15	0.12	0.14
10769	NO	NO	98	0.15	0.12	0.14
example_results_file.txt			174	0.27	0.21	0.24

From the previous table we can conclude that Stemming does not contribute as much as *Stopwords*, being the use of *Stopwords* more important. The use of a *Stopwords* reduces significantly the number of words in the inverted index as well.

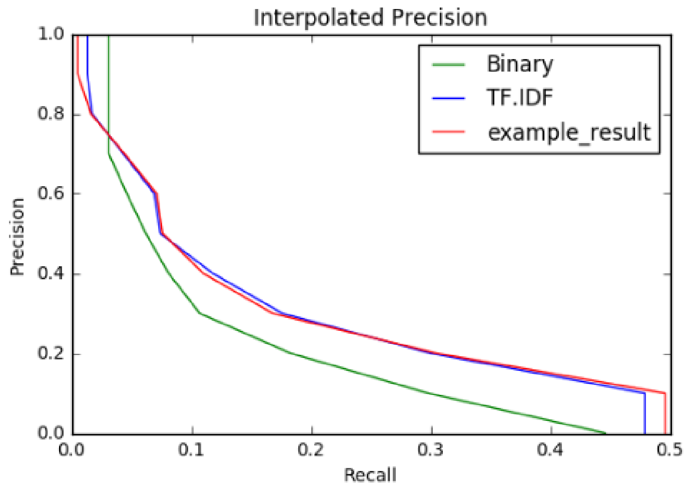
TF.IDF

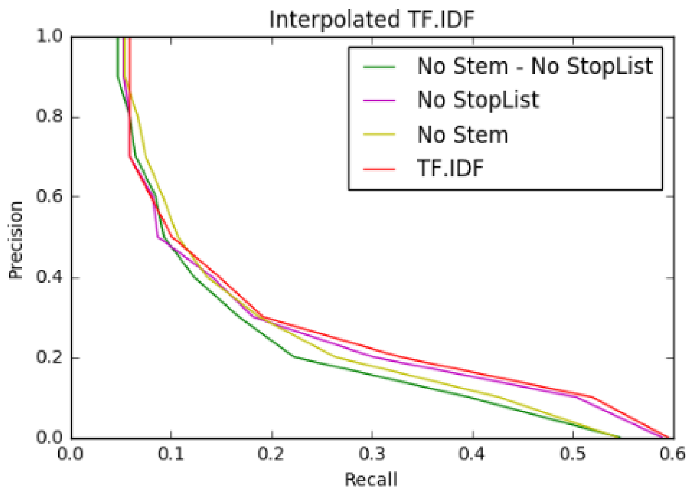
Total Words in Index	Stemming	StopWords	Rel_Retr	Precision	Recall	F-measure
6963	YES	YES	169	**0.26**	**0.21**	**0.24**
7218	YES	NO	166	0.26	0.21	0.23
10356	NO	YES	140	0.22	0.18	0.19
10769	NO	NO	132	0.21	0.17	0.18
example_results_file.txt			174	0.27	0.21	0.24

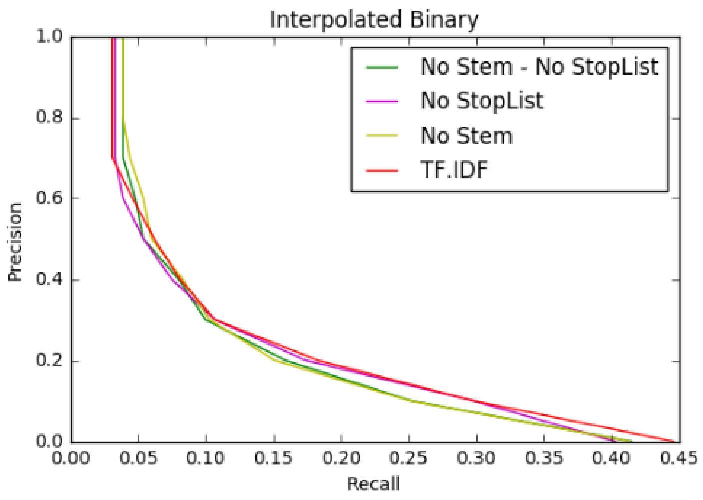
It makes sense that using Stemming and Stopwords helps to increase the precision and recall of our system as we increment preprocess tasks we increase our F-Mean and our precision (proportion of retrieved documents that are relevant).

The graph on the right shows a comparison between the Binary weighting and the TF.IDF method. We can observe that our TF.IDF system behaves almost exactly to the example provided, and also we can show that the Binary method has less precision and recall in contrast to the TF.IDF

In conclusion, there are some preprocessing algorithms that aids in information retrieval systems. From the graphs below we can conclude that the use of *Stemming* and *Stopwords* will always help to improve or IR system by giving higher precision and recall.


Interpolated Precision
Binary
TF.IDF
example_result
Precision
Recall


Interpolated TF.IDF
No Stem - No StopList
No StopList
No Stem
TF.IDF
Precision
Recall


Interpolated Binary
No Stem - No StopList
No StopList
No Stem
TF.IDF
Precision
Recall