Assignment 1 – Document Retrieval

Brief

The main goal of this project was the retrieval of the most relevant documents based on a series of queries. The techniques involved use the *bag of words* model, using term frequencies and inverted term indices to weight the terms and create vectors to be compared for similarity.

Weighting Methods

Binary

A Boolean search considers a document relevant simply if it contains a term in the query. As the data passed to the function as index is stored in a dictionary, checking a terms presence can be done efficiently. Scores for accuracy are fairly low compared to other retrieval systems, the main weakness of a Boolean search is that factors such as frequency and rarity of a term aren't considered in this sort of search, and this results in a lot of irrelevant documents containing common words to be returned.

Term Frequency

The term frequency was the first proper weighting method implemented, using the number of occurrences of a word in a document as weight. The data can be manipulated further to obtain normalized values. In my solution I've explored three different variations for term frequency: using the raw tf value, adding 1 to the log of the tf, using the most frequent term's frequency to normalize all term frequencies in a document.
The dictionary provided as index has structure *{term: {docID: count}}*, where term is a unique word, mapped to a dictionary of all the documents containing the term and the term's respective occurrences in the document. This was equivalent to the term frequency and thus needed no further manipulation.
Logging the values has the effect of normalizing the data and reduces the weight of really common words; these were calculated through the inbuilt library *math* and the *math.log(tf)* function.
Using the top frequency of a document, involved creating a new dictionary in the form of *{doc: {term: tf}}* so that the top value could be extracted; this value was then passed into a function *a + (1 – a) * (tf/top_tf),* where a is an arbitrary value (set to 0.4 for this solution) and which returns a term frequency normalized against the most frequent term.

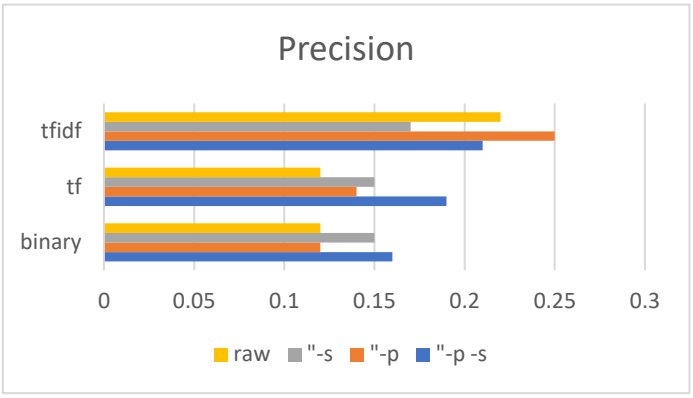
Term Frequency - Inverse Document Frequency

Adding the inverse document frequency as a factor to calculate a term's weight, results in higher scores for more impactful terms. This means common words such us 'the', 'and', 'to' etc. will score lower scores than rarer words. Calculating the idf of a term only requires the total number of documents in collection and the number of documents containing the term. My solution builds a dictionary with structure *{term: {doc: idf}}* in the class constructor using a list comprehension. These can be accessed by any function in the class, to compute both documents and queries weights.

Different solutions and combinations of tf and idf were explored, leading to more or less accurate document retrievals. Overall, the solution which scored highest for all three measurements of relevance used the formula *1+log(tf) * idf.*

Results

The binary implementation scored the lowest out of the three weighting methods, term frequency was better, especially in combination with stop words and stemming, however the tfidf method by far outscored the other two methods in all categories. For reference, the chart on the right compares precision of the different weighting methods.



Precision			
	binary	tf	tfidf
"-p -s	0.16	0.19	0.21
"-p	0.12	0.14	0.25
"-s	0.15	0.15	0.17
raw	0.12	0.12	0.22
Recall			
	binary	tf	tfidf
"-p -s	0.13	0.15	0.17
"-p	0.09	0.11	0.2
"-s	0.12	0.12	0.14
raw	0.09	0.09	0.17
F-Measure			
	binary	tf	tfidf
"-p -s	0.14	0.17	0.18
"-p	0.1	0.13	0.22
"-s	0.14	0.14	0.15
raw	0.1	0.1	0.19

The results were expected, using the document frequency to filter out irrelevant words does the most impact in retrieving relevant documents.

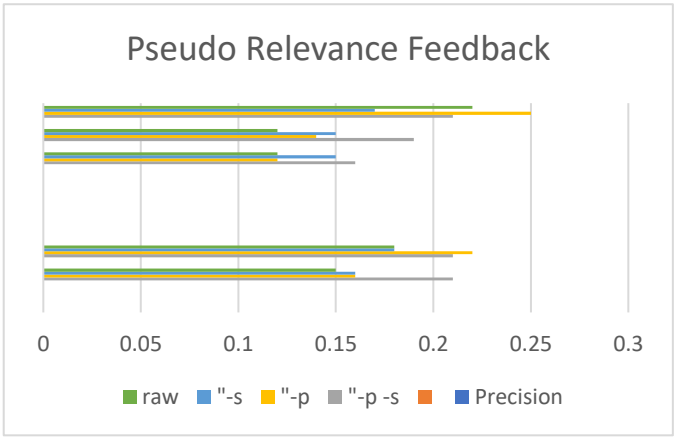
While stemming used in conjunction with stop words produced the best results for binary and tf weighting methods, it wasn't the case when using tfidf; stemming only of the words produced the highest scores of all tests carried out.

The runtime for any of the weighting schemes wasn't higher than 0.5s in any of the tests, there is ever only two nested for loops at most, giving us a time complexity of $O(n^2)$. Most values are computed and stored before the queries are processed to reduce computation time. Dictionaries are the main data structure used, these provide instant access to items and their data, making the software run considerably faster.

Pseudo Reference Feedback

As an attempt to improve relevance of the documents retrieved, pseudo reference feedback was implemented. This takes the results produced by the document retrieval, extracts the most relevant terms using tfidf to produce a new updated query, which is now used again against the documents collection to identify the new most relevant documents.

To achieve this, the function builds a dictionary of structure {doc: {term: tfidf}}, containing the top n documents extracted from the original results. The top terms t of each document are considered and added to the original query. The cosine of this new query is then computer against the documents collection to produce a new set of results.



Using pseudo reference feedback had a positive effect on retrievals using TF only. This however could also be attributed to the fact that the second query search is performed using tfidf.

A major drawback of this method is the execution time of the program doubling, as for every query the term weights have to be calculated twice.