Text Processing: COM4115/6115. Autumn Semester. William Briggs

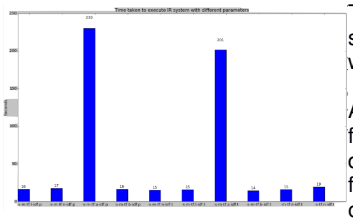
System Brief

- Built with Python 3
- Full implementation using vector space model
- Support for optional stemming and stop list
- Different flavours of tf-idf weightings (tf: natural, boolean, logarithmic, augmented) (idf: term, probability)
- Support for additional 3 different stemmers + lemmatizer
- Optimized result set mode (set -n to -1)(see README.txt for details)
- Rocchio Relevance feedback algorithm implementation
- Support for both running on-the-fly and storing tf-idf values against a document (-F)

Keys -Here are the command line options. Some might help in interpreting this report. (Can also be found in README.txt)

```
-h prints help message and exits program
-s use the stoplist file[OPTIONAL]
-c the collection file
-i the index file [this will output a new index file with the given name]
-I use an existing index file[OPTIONAL][Name of index file is taken from -i param][The index file must be in the same directory]
-m use stemming [OPTIONAL][OPTIONS: p, s, l, lem][porter, snowball, lancaster, wordnetlemmatizer respectively]
-q a query id [OPTIONAL][Used to execute a single query]
-o output name for results file
-tf the flavour of tf [OPTIONAL][Default is n(natural)][OPTIONS: b, l, a][boolean, logarithmic, augmented respectively]
-idf the flavour of idf [OPTIONAL][Default is t][OPTIONS: p][prob idf]
-n number of documents to return. Higher n == higher recall but lower precision.[Default is 10][set to -1 to get more accurate optimized result set]
-r use relevance feedback mode[OPTIONAL][OPTIONS: n][a numerical value of how many documents you will have to make as relevant. default is 10]
-F generate output on the fly [OPTIONAL] In order to implement the relevance feedback, I had to store the tf.idf vectors for later modifaction. This flag will stop the system from storing the tf-idf values (cannot be used with rr mode)
```

Performance



The graph shows the average execution time (in seconds) of each output type. These timings should be taken with a grain of salt as they change under different systems. They also assume that we are not computing results on-the-fly.

A huge performance spike can be seen when running the application with augmented term frequency types. This is because each vector has a dimension of the vocabulary of all the documents. The augmented
frequency means each individual vector will always have a value of at least 0.5. With this factored out, the average execution times are quite well rounded.

Summary of Standard Results

OuputType	Precision	Recall	Harmonic Mean
'-s-m-tf:l-idf:p'	0.25	0.20	0.22
'-s-m-tf:n-idf:p'	0.26	0.21	0.23
'-s-m-tf:a-idf:p'	0.24	0.19	0.21
'-s-m-tf:b-idf:p'	0.21	0.17	0.18
'-s-m-tf:n-idf:t'	0.27	0.22	0.24
'-s-m-tf:l-idf:t'	0.25	0.20	0.23
'-s-m-tf:a-idf:t'	0.24	0.19	0.22
'-s-m-tf:b-idf:t'	0.21	0.17	0.19
'-m-tf:n-idf:t'	0.26	0.21	0.23
'-s-tf:n-idf:t'	0.22	0.18	0.20
binary-m-s/-s/-m/binary	0.15/0.12/0.09/0.07	0.12/0.10/0.07/0.06	0.14/0.11/0.08/0.07
term-freq-m-s/-s/-m/term-freq	0.18/0.11/0.140.07	0.14/0.09/0.11/0.06	0.16/0.10/0.13/0.07

The table shows the average return measures from outputting the top 10 documents for each query The best standard run configuration is when using a stop list, the porter stemmer and using natural tf as well as idf (logN/dw). Using natural tf.idf over binary and term-freq based vectors improves our performance significantly. The harmonic mean drops as low as 0.07 when using binary vectors without a stop list or stemmer.

Using more optimized output modes, we can improve on these results. We explore this possibility later in the report.

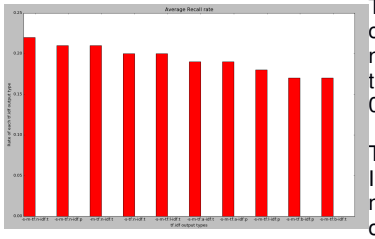
In-depth Results

tf.idf vs tf vs binary

Using the raw tf values as vectors gives us a lowly 0.16 average f-measure. Using a pure binary value vectors naturally gives us even worse results with an average f-measure of 0.14. NOTE: these computations were disregarded pretty quickly as this report focuses more on the different formats of tf.idf weightings and returning an optimum number of documents.

Recall

Using the gold standard result set as a base line, I was able to evaluate the results produced by my IR system. To calculate recall we compared the number of documents returned for each query contained within gold standard data set.



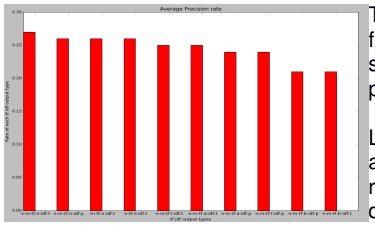
The graph shows the average recall rate of each run type for the top 10 documents. This is computed as the sum of the recall rates for each run type / the total number of queries. Using natural tf and idf produced the highest average recall rate of 0.22. Ignoring the stop list dropped the average recall rate down to 0.21. Removing stemming dropped the rate down even further to 0.18.

This calculation is not going to produce the most accurate reflection of recall. This is because my IR system pulls back a constant result count, (top 10) whilst the gold standard presents only the most relevant documents. An example of this can be seen for query 64; the gold standard shows only 1 relevant document, meaning the chance of a perfect recall is very high.

Precision

We can calculate the precision of the system by dividing the relevant retrieved documents by the total number of documented retrieved. This was again calculated using the gold standard data set to identify the relevant documents.

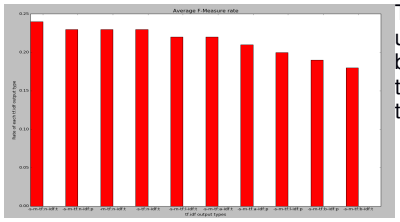
Text Processing: COM4115/6115. Autumn Semester. William Briggs



The chart shows the average precision rate. This is computed as the sum of the precision rates for each run type / the total number of queries. The best run type for precision is from using stemming, and the stop list, this also assumes a natural tf.idf weighting. The worst rate for precision is when using a boolean tf weighting.

Like the recall calculation, the results are somewhat inaccurate. This is because my IR system always pulls back N documents for a query, the precision will be out of N. The gold standard returns one result for query 64, even though my IR systems detects this result as the most likely document, another 9 results are also returned and thus the precision is calculated as 1/10.

F-measure
As a way of combining both the recall and the precision together, we can compute the F-Measure.



The graph shows that the highest rate of retrieval is achieved under a run configuration of using a stoplist, stemming and a natural tf.idf weighting. It's also worth noting that the boolean based term frequency produce the lowest rate of recall, precision and average between the two. This suggests the most efficient retrieval type for processing the query set is from using tf.idf rankings for each term along with the vector space model.

Stemming/Lemmatizing
Using the tf.idf natural weighting along with a stop list we can compare which of the three stemmers and lemmatizer gives us the highest harmonic mean.

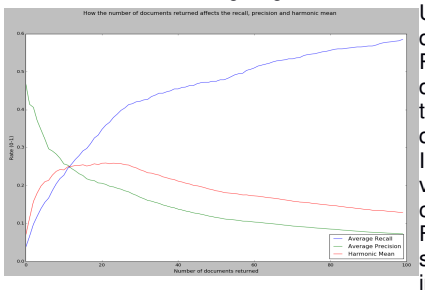
Stemmer/Lemmatizer Type	Precision	Recall	Harmonic Mean
Porter Stemmer	0.27	0.22	0.24
Lancaster Stemmer	0.26	0.21	0.23
Snowball Stemmer	0.27	0.21	0.24
WordNetLemmatizer	0.25	0.20	0.22
No Stemmer	0.22	0.18	0.20

The snowball stemmer gives the highest rate of precision, but the porter stemmer is our best overall choice.

Tokenisation
Our formalisation of what we constitute as words will impact the performance of the system. Some considerations would be if we should store numbers, hyphenated terms and words with apostrophes as singular terms. These results assume a natural tf.idf weighting along with stoplist and porter stemmer.

Type of Match	Regex	Precision	Recall	Harmonic Mean
Words only	[A-Za-z]+	0.26	0.21	0.24
+Hyphenated Words	[A-Za-z]+-?[A-Z-a-z]*	0.27	0.22	0.24
+Numbers and apostrophes	'?\w[\w\d']*(?:-\w+)*'?	0.27	0.22	0.24

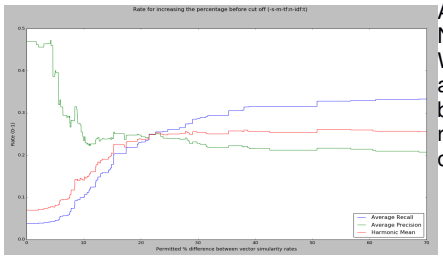
How many documents should we return for each query?
An issue arises when computing the recall, precision and f-measure when our IR system returns a constant amount of documents for each query. The above results assume each query returns 10 documents. It would be better if we could work out the optimum quantity of documents to return, giving us the best f-measure.



Using our preferred output type (-s-m-tf:n-idf:t) we can look at how the number of returned documents affects the average recall, precision and f-measures.
Recall naturally goes up as we return more documents, but as we start to reach 100 documents the rate starts to level out. This suggests that the similarity with query vector at this point is negligible and does not contribute enough weighting to associate itself with the query.
Inversely, the precision rate drops as we start to return more documents. This is because very few/if any gold-standard data set results return more than 20 documents, thus most of our later return results negate the precision rate.
Finally, the f-measure peaks around the 20 returned documents mark, but remains relatively stable between 7-25 documents. The rate gradually declines as the quantity of returns is increased, this is due to the effect of the precision. The interception point is at 11.5 documents where all three averages will be around 0.25. Using this information, we can now tweak our original configuration to output 12(-n 12) documents rather than 10, which gives us a new f-measure high of 0.25.

There is still room to improve this result even further, in cases with a limited amount of relevant documents. For instance, suppose I had query like so: “I want document 2781”. The idf weighting for the value “2781” will skyrocket, document 2781 would most likely finish on top, but we would also need to pull back another 9 results, which would have a detrimental effect on our precision rate, and have no effect on our recall rate. NOTE: This is just an example; the document ids are not actually considered when returning.

Returning an optimum number of documents



As it stands, our result set takes the format of rectangular multi-dimensional array. That is, N by Q. Where N is the number of documents returned and Q is the number of queries. We need our result set to take the format of a jagged array, where N becomes dynamic for a particular query. As a naïve approach to this problem we can assess the distance between vectors and determine if the range is great enough to cut off the rest of the results deemed to be irrelevant. Note: these results are computed using our preferred output type of (-s-m-tf:n-idf:t)

In order to determine the distance between two document-query scores I simply checked if the first score is a set percentage greater than the other. The lower the percentage; the lower the cut-off point and thus less documents will get returned.

The graph shows the recall, precision and f-measure between the return set and the gold standard as we increase our range percentage for vector similarity.

I started the process at 0.01% distance between vectors; as this is the point in which more than 1 document starts to get returned for each query. I also decided to cap the max returned documents to 20(as queries for some documents were very similar and 100s of documents would get returned very early on in the process)

The rate was increased by 0.01% each time I ran the output. This improves the f-measure to a *spellbinding* 0.2608; which is higher than any of the previous constant count run configurations, regardless of the run parameters. We reach this peak around the 55% vector similarity cut-off mark.

While it is satisfying to see that we were able to improve the quality of output using this method, it's evident there would be better ways in determining when we should cut-off the rest of the returned document set for a query. With a much larger dataset, we would also need to consider returning more than 20 documents.

An option to get this more optimized result set has been added as a command line flag when running the IR system. (set -n to -1).

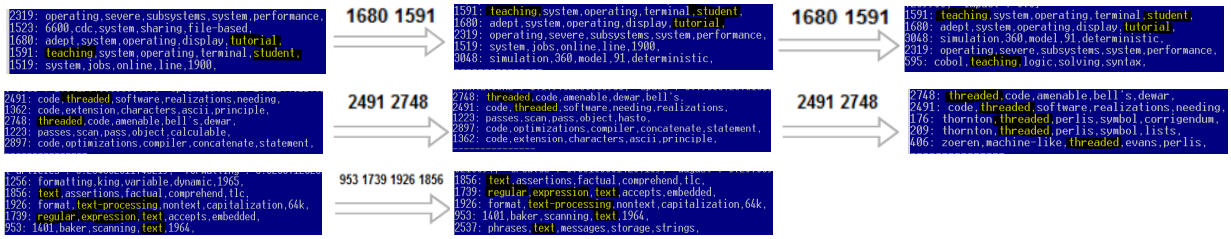
Rocchio Feedback algorithm

The IR system features an implementation of the Rocchio feedback algorithm. This can be ran using the -r option followed by a number of how many documents you will be presented with to deem as relevant. This will only work with a single query and it's normally best to run it without stemming to identify the words better. e.g:

Python main.py -s stop_list.txt -c documents.txt -i william.txt -I -o f_run_results.txt -tf n -idf t ***-r 5 -q 1***

This feature uses the tf.idf vectors for the documents and the query as well as cosine similarity using the vector space model. This is a recursive call and will keep asking for more relevant documents until -1 is specified. The highest ranking documents for the query will then be printed out to file in the usual format.

Here is an example of running the system in relevance feedback mode. After the first cycle we have 5 documents. I have high-lighted 3 terms which are relevant to our search. The system will then ask us to input document ids which we see as relevant to the query. Ids 1680 and 1591 are input. The first cycle moves the two documents we deemed most relevant to the top. Inputting the same document ids again results in document 595 appearing as the 5th most relevant, which contains one of our terms. A couple more examples are then shown below.



As it's not feasible for the user to select all documents they deem to be relevant from the full data set, we simply present only the top n documents. Documents that are selected will increase the vector values for the query, conversely the unselect documents will decrease the values. We also present the top 5 ranked words within the document to the user to give them an idea of what the document is about.

Additional Notes

- The program features a progress bar which displays how far the current task is to completion. **I do not claim any credit for the implementation of this.** It was taken from: http://stackoverflow.com/questions/3173320/text-progress-bar-in-the-console
- The file used for analysing the optimized outputs can be found in the results folder named 'outputFile.txt'
- There is support for querying using a boolean model I.E 'Dog AND Cat OR Snake'. This is implemented in the boolean_query_model class, but is not accessible from using command line arguments. This was mainly implemented to just test my inverted index.

Improvements

- Optimize the 'on the fly' generation in the vector_space_model class. The decision to store the tf.idf weights in a separate class alongside the inverted index was influenced by the need to modify the vectors when implementing the relevance ranking feature. **You can still run the IR system without storing any tf.idf weights against the document by setting the -F flag**.

- There is a slight problem when wishing to generate an index file for a stemmed output, then generate a non-stemmed output with the same index file. My system really should detect that the index file is not compatible with the current run type; or at least offer a warning to the user of this problem.