# COMP0084_ICA1

Information Retrieval and Data Mining

Student number: 20101171
Department: Statistical Science
UCL
ucakdli@ac.uk.com

## TEXT PRE-PROCESSING

In the whole project, the same text pre-processing was implemented to both query text and document (passage) text so that they could pair each other. To extract terms from a whole text for further processing, "word_tokenize" under "nltk" package was used and it is more effective in dealing with symbols in the sentence compared with simple split function. After extract different terms, processing is still needed.

With regards to term pre-processing, four parts and their order were mainly considered:

1. Delete the terms which do not contain alphabet: It could be observed that there is no rare str consists of number or symbol in the query text so that such str appeared in the document part was not considered

2. Remove the stopwords (to improve efficiency): Although the IR (information retrieval) methods could avoid the impact of these common words, it is still necessary to remove them to improve efficiency as there are so many terms. To accurately recognize the stopwords, words were lowered before the processing and stemming was designed to be carried out after this part (e.g. because → becaus by stemming).

3. Stemming: Combining the same word with different forms could improve both effectiveness and efficiency. And the algorithmic type was applied here under "nltk package".

4.Other issues: It was also found that the "word_tokenize" function sometimes would separate a whole word. For example:

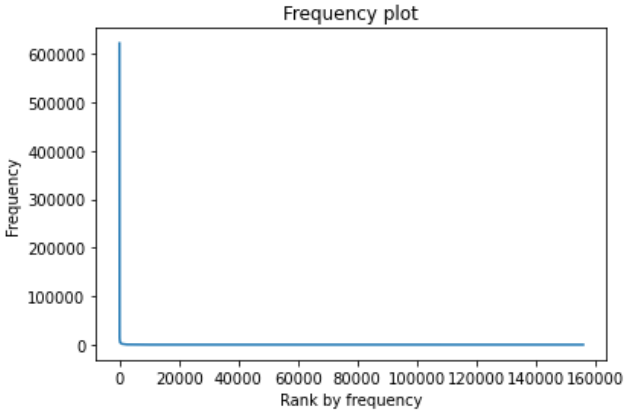
```
In [55]: word_tokenize("they're")
Out[55]: ['they', "'re"]

In [56]: word_tokenize("apple'tree")
Out[56]: ["apple'tree"]
```

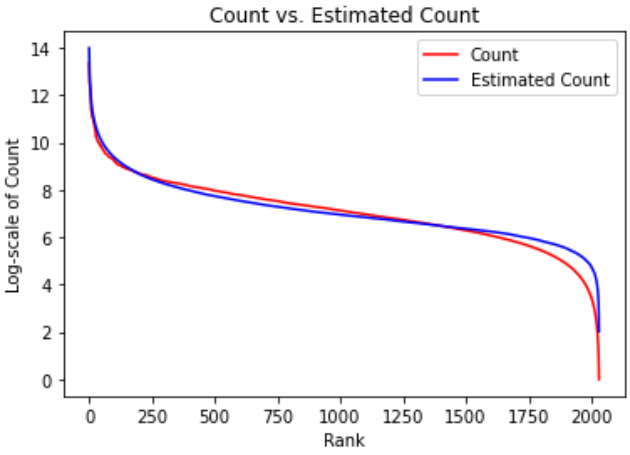
Four of them which appeared fairly frequently were removed and they are 're, 's, 'll, 've.

## ANALYSIS OF ZIP'S LAW

Before the analysis, it is necessary to mention that stemming and deleting some useless words were still implemented as the former step could gather the same terms and the later could delete terms generated by some errors. However, the step about removing the stopwords was not implemented as stopwords actually belong to collection and removing it would disturb the zip's law.




The above is the plot for the counts of terms from the provided dataset and the frequency ranked by higher value is put in the left. From the shape of the line, it seems that zip's law is probable to be true. However, that's not enough.




Firstly, note that if term counts of several terms are the same, then all these terms would have the same rank value and there are

about 2000 unique rank value. For each unique term frequency, according to the formula of zip's law, there could be an estimated k (term frequency times ranked value). And the mean of these estimated k (equal to 1182226) could be regarded as the parameter of zip's law. Using this parameter, the term count could be estimated inversely with a given ranked value. The above plot shows the relationship between the term count and the estimated count in log scale, which could reflect the consistency of the estimated parameter of zip's law. As the two lines could generally fit each other, we could say the zip's law exists here.

## IMPLEMENTATION OF INVERTED INDEX

During going through each term in the collection, after implementing the pre-processing, the inverted indexes of terms and the document information were both stored in the "dictionary" in python.

For the inverted index dictionary, each unique term in the collection were reserved in the key and its position (passage id) as well as the counts in that position were stored in the values. If the counts of that term in the collection is needed, simply sum these counts could give the answer. The inverted index could provide help for the following method like idf which needs the information about position. The reserved count was not used but it could reflect the original text if not considering the order of terms.

For the document information dictionary, each document (passage) id was reserved in the key and for each document, the unique term as well as its count were reserved in the value. This dictionary was even more practical when calculating the score in a certain document.

The inverted index dictionary and the document information dictionary actually contain the same information, which means given one of them, it could be converted to another one. And the document information dictionary could be regarded as a transpose version of the inverted indexes dictionary. The reason to store the inverted index in two types is to simplify the procedures to retrieve terms in the following re-ranking systems. It would be easy to check terms under a certain document or find where and how many a certain term appears.

The above procedures were actually realized by a function named "dict_con" in the code file so that for each different sub collection (under the same query), two dictionaries could be calculated immediately.

Example:

Key in inverted index dictionary (left side)

| Key | Type | Size | Value |
|---|---|---|---|
| acceptor | Counter | 38 | Counter object of collections module |
| accer | Counter | 1 | Counter object of collections module |
| accesori | Counter | 1 | Counter object of collections module |
| access | Counter | 1530 | Counter object of collections module |

Value in inverted index dictionary (a count dictionary)

| Key | Type | Size | Value |
|---|---|---|---|
| 6079 | int | 1 | 1 |
| 55381 | int | 1 | 1 |
| 111953 | int | 1 | 2 |
| 172754 | int | 1 | 1 |
| 377657 | int | 1 | 1 |
| 595715 | int | 1 | 1 |

Key in document information dictionary (left side)

| Key | Type | Size | Value |
|---|---|---|---|
| 301 | Counter | 11 | Counter object of collections module |
| 304 | Counter | 17 | Counter object of collections module |
| 321 | Counter | 9 | Counter object of collections module |
| 350 | Counter | 12 | Counter object of collections module |

Value in document information dictionary (a count dictionary)

| Key | Type | Size | Value |
|---|---|---|---|
| agreement | int | 1 | 2 |
| contractor | int | 1 | 1 |
| hire | int | 1 | 1 |
| includ | int | 1 | 1 |
| perform | int | 1 | 1 |
| relationship | int | 1 | 1 |

## IMPLEMENTATION OF VS/BM25 MODEL

In the beginning, one thing needs to be clarified is that "Term frequency" here means the term count divided by the number of words in the document.

**For VS model:**

A term frequency function was firstly created to calculate the tf of each term in each document using the document information dictionary. And then the inverse document frequency function was created to calculate the idf of each term appeared in the

collection using the inverted index dictionary. The version of idf here is $\log_{10}(N/df)$.

In the final function, in each document, according to the occurrence of terms, the tf-idf vector was generated by multiplying the tf value and idf value of each term. Similarly, the query vector could also be produced. The idf value of each term in query is same as that in document while the tf value is its term frequency in the query text. Then the cosine similarity between two vectors could be calculated by using the inner product divide the length of vector.

In the codes, a simple trick was used that during calculating the inner product of two vectors, the vector space of document vector was just selected to be the vector space of query vector as otherwise, the multiplication of two elements would also be zero. But the length of document vector was still the real length.

Although in tf part the term frequency was used instead of term count, during normalizing the vector, the effect of "divide by the sum number of words" was eliminated, which means they could actually produce the same result.

**For BM25 MODEL:**

Similar to the idf in the vector space model, the log part in the BM25 was firstly calculated using the inverted index dictionary. Note that there was not relevance information, so the relevance parameter was zero. Then going through each document, the BM25 score could be calculated respectively.

The main discussion is about the parameter selection (k1, k2 and b). k1 and b represents the effect of term frequency in document as well as the effect of document length. As there is the empirical optimal value for k1 (1.2) and b (0.75), here these values were used. From the formula of BM25, it could also be found that k2 represents the importance of query term frequency in query. And it was noticed that when each term only appears once, k2 as well as this part would be one which means they have no effect. For this project, as stopwords were eliminated, k2 and qf (query frequency) importance part do not play an important role. Based on the query size in this project, here we chose k2 to be 5, which suggests the importance of query term increases slowly as the qf increases.

## IMPLEMENTATION OF LANGUAGE MODEL

No matter which smoothing method we use, they all belong to the query likelihood language model and the ideas are the same. It is just to calculate the log probability of the joint (independent) distribution of query text, while the parameters are estimated by document. And if a term in query appears more than once, the score value (log-probability) would be added several times according to the logic to calculate probability.

The difference between three smoothing methods is just that the log-probability of each certain word is different. And it could be noticed that the Laplace smoothing method is equivalent to using Lindstone correction with e=1, as a result, the functions of these two methods were combined.

In general, two functions were created and the mutual inputs were text of query and the inverted index dictionary with its transpose version of the sub-collection. Laplace/Lindstone function needs an additional value of "e". The output would be the scores of different documents. The concrete procedure could be seen in the codes which were noted detailedly.

One finding is that for the query whose qid is 1063750, there is a term named "volunterilay" and it never appeared in the whole collection and as a result, Dirichlet smoothing would fail in this query.

In terms of the performance of these three models, it is considered that the model which could most accurately measure the weight of different terms would behave best. The problem of plain Laplace smoothing is that it gives too much weight to unseen terms. In this project, the number of words in a certain document is much smaller that it in the entire collection (D<<V), which means the information of one document is underestimated and suggests that the smoothing with Lindstone correction is expected to behave better. However, it still treats the unseen word equally, while with the background information as weight, the Dirichlet could measure each term more accurately. In addition, after processing the text of the query text, each query is short, which also supports Dirichlet smoothing to work better if the given parameter ($\mu$=2000) is suitable. However, as mentioned before, there is one case that Dirichlet smoothing would fail so that smoothing with Lindstone correction may work best in that document. In general, it is expected that the ranking of three methods is Dirichlet > Lindstone > Laplace.

In the running program (python file), codes were written efficiently so that the ranking system could be finished in 5 minutes. And codes were written in "spyder".