**CS6200 Information Retrieval Project 03**

Jiachi Liu

# Problem

In this project, I will replicate the functionality of the Lemur index used in Project 2, and in conjuction with the code created implementing various retrieval functions for Project 2.

# How to Run

This project is using BeautifulSoup to parse the query file. There are total six runnable scripts. To run the system, first run the index script to create index file and doclist. The index scripts are the following two:

1. run\_index.py:
   1. command: python run\_index.py
   2. description: reading all documents in cacm and create doclist and index file. These two files will in the same directory as run\_index.py. The index do not contains stopwords.
2. run\_index\_nostop.py:
   1. command: python run\_index\_nostop.py
   2. description: reading all documents in cacm and create doclist and index file. These two files will in the same directory as run\_index\_nostop.py. The index contains stopwords.

To test the database with removing stopwords, please use run\_index.py, and to test database without removing stopwords, please use run\_index\_nostop.py before run query scripts. After created the index, running the query scripts to test retrieval models. The scripts as following:

|  |  |
| --- | --- |
| Model id | Model |
| 1 | OKTF |
| 2 | OKTF\_IDF |
| 3 | Max likelihood with Laplace |
| 4 | Max likelihood with Jelinek-Mercer |
| 5 | BM25 |

Table 1 Model\_id and model

1. run\_query.py:
   1. command: python run\_query.py MODEL\_ID OUTPUT\_FILE.
   2. Example: python run\_query.py 5 bm25.txt
   3. Description: this script will read the query file and use given model to generate output file. It removes the stopwords in query.
2. run\_query\_nostop.py:
   1. command: python run\_query\_ nostop.py MODEL\_ID OUTPUT\_FILE.
   2. Example: python run\_query\_ nostop.py 5 bm25.txt
   3. Description: this script will read the query file and use given model to generate output file. It will not remove the stopwords in query.
3. run\_query\_100times.py:
   1. command: python run\_query\_100times.py MODEL\_ID OUTPUT\_FILE.
   2. Example: python run\_query\_100times.py 5 bm25.txt
   3. Description: this script will read the query file and use given model to generate output file 100 times. It removes the stopwords in query.
4. run\_query\_nostop\_100times.py:
   1. command: python run\_query\_ nostop\_100times.py MODEL\_ID OUTPUT\_FILE.
   2. Example: python run\_query\_ nostop\_100times.py 5 bm25.txt
   3. Description: this script will read the query file and use given model to generate output file 100 times. It will not remove the stopwords in query.

# System Overview



Figure 1 System Overview

The system contains four components shows in Figure 1:

* The **common** component contains data structures and utility that both will be used in query and index process, such as stop list and stemming.
* The **Index** component implements the logic to statistic and create index for a given corpus.
* The **query** component comes from Project 2, which contains five retrieval models and some auxiliary modules to parse the given query file index search.
* The **evaluation** component use evaluation script and qrel file to provide precision for query results.

# System Implementation

## Common



Figure 2 Common Component

This component has four classes. The StopList saving the stop words information and provides methods to test whether a given word is a stop word. The Stemming class reads the stemclass file and provides method convert a given word to its stem. In this project, the KStem is used and I chose the loose version. TextFilter is a tool to convert the given content text to a list of terms. It splits the text into words and remove all punctuation, stop words, and convert the word to stemming terms. The NonStopTextFilter overrides the filter method in TextFilter, which will preserve the stop words in terms.

## Index



Figure 3 Index Component

Index component response for indexing the given corpus and generate the statistic information for the corpus database. For each document in corpus, a Document class will be created to save all processed terms and their frequencies in this document. After that, the Indexer will merge all the terms in Documents into a single invert index list and passing the result with all Documents into Database class. The Database will statistic the information just like the ‘db’ command of Lemur, such as number of terms, number of documents and so on. It also formats the invert index list and doc list and writes them to disk.

## Query

The query component is directly comes from project 2. The majority of this component is not changed. Only the InvertListParser class and QueryParser class has a small change in order to support new format of index and query file.

Since this time the index information will be read from index file on disk instead of from web interface, I created a Lemur class, which provides a query function. This method takes a term and an index file path, returns the index information in programming model as same as Project 2. The InvertListParser will use the new Lemur class to get the index for each term.

The QueryParser is using BeautifulSoup library to parse the given query file since this time the file is in xml format.

## Evaluation

The evaluation component uses the same script to evaluate the result. The qrel file will be cacm.rel.

# Evaluation and Analysis

The query file is cacm.query. Using the same model as we did in project 2. The result is shown as following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Database(STOP STEM), Query(STOP STEM) | | | | | |
| ID | Model | MAP | R-Precision | Precision at 10 | Precision at 30 |
| 1 | OKAPI TF | 0.2973 | 0.3182 | 0.3212 | 0.1872 |
| 2 | OKAPI TF\*IDF | 0.3767 | 0.3932 | 0.3654 | 0.2147 |
| 3 | Laplace | 0.2258 | 0.2362 | 0.2615 | 0.1558 |
| 4 | Jelinek-Mercer | 0.3242 | 0.3457 | 0.3327 | 0.2051 |
| 5 | BM25 | 0.3660 | 0.3765 | 0.3558 | 0.2147 |

Table 2 Result for Database with Stop

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DB = 3 (STOP,STEM) | | | | | | | | | |
| ID | Model Name | MAP | | R-Precision | | Precision at 10 | | Precision at 30 | |
| NIST | IRClass | NIST | IRClass | NIST | IRClass | NIST | IRClass |
| 1 | OKAPI TF | 0.1863 | 0.2538 | 0.2208 | 0.2622 | 0.3240 | 0.2800 | 0.2680 | 0.2040 |
| 2 | OKAPI TF \* IDF | 0.2570 | 0.3150 | 0.2813 | 0.3159 | 0.3880 | 0.3400 | 0.3187 | 0.2427 |
| 3 | Laplace | 0.1848 | 0.2501 | 0.2316 | 0.2358 | 0.3608 | 0.3200 | 0.2880 | 0.2493 |
| 4 | Jelinek-Mercer | 0.2181 | 0.2483 | 0.2472 | 0.2621 | 0.3200 | 0.2608 | 0.2707 | 0.1920 |
| 5 | BM25 | 0.2503 | 0.3035 | 0.2770 | 0.3025 | 0.3560 | 0.3080 | 0.3120 | 0.2333 |

Table 3 Result from Project 2

We use NIST result in project 2 to analyze with project 3. Compared the MAP results with Project 2, the relative position of models is remain same. In other words, OKAPI TF\*IDF is still got the best accuracy followed with BM25. The Jelinek-Mercer has third best accuracy where OKAPI TF is at fourth and Laplace is the worst.

Although the relative position of MAP remains same, the R-Precision has a small difference. The OKAPI TF is a little lower than Laplace in project 2 where it is much higher than Laplace in project 3. See the precision at 10 docs, Laplace model has less accuracy in project 3. In fact, compare all the accuracy before 30 docs, Laplace model has less accuracy than OKAPI TF. But after 30 docs, these two models got almost same accuracy. Thus in project 3, OKAPI TF has a better performance at beginning, but drop fast whereas Laplace remain a relative steady drop rate.

In order to test the effect on running time of stopwords, running tests on database with and without stopwords.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DB | NUM\_DOCS | NUM\_TERMS | NUM\_UNIQUE\_TERMS | AVG\_DOC\_LEN |
| STOP | 3204 | 164420 | 12112 | 51 |
| NOSTOP | 3204 | 241662 | 12384 | 75 |

Table 4 Database Summary

|  |  |  |  |
| --- | --- | --- | --- |
| Model | DB(STOP) Query(STOP) | DB(NOSTOP) Query(STOP) | DB(NOSTOP) Query(NOSTOP) |
| OKTF | 17m7.181s | 20m14.467s | 33m3.532s |
| OKTF\*IDF | 17m59.463s | 20m56.339s | 32m57.247s |
| Laplace | 17m13.696s | 20m21.524s | 39m47.816s |
| Jelinek\_Mercer | 18m4.584s | 23m11.937s | 36m24.758s |
| BM25 | 16m17.341s | 19m32.545s | 32m16.559s |

Table 5 Running Time

From the table, we can see that when database and query both remove stopwords, it requires the least running time. Without removing stopwords on database, the running time becomes slightly increasing. Since the size of index becomes a little bit greater (from 12112 to 12384), each time we getting the invert list from index file need a little more time.

However, when the query also not removing stopwords, the running time greatly increased. For each stopword in query, it needs to get the invert list and calculate the rank score on this term in retrieval models. And stopwords usually have long invert list. We can see in Table 3, the number of terms becomes greatly increased, from 164420 to 241662, without removing stopwords. Thus they require a lot more time when compute the rank score. And the total running time in query 100 times becomes larger.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DB(NOSTOP STEM) Query(STOP STEM) | | | | | |
| ID | Model | MAP | R-Precision | Precision at 10 | Precision at 30 |
| 1 | OKTF | 0.2955 | 0.3125 | 0.325 | 0.1846 |
| 2 | OKTF\*IDF | 0.3824 | 0.3865 | 0.3712 | 0.2179 |
| 3 | Laplace | 0.2273 | 0.2357 | 0.2712 | 0.1564 |
| 4 | Jelinek-Mercer | 0.3233 | 0.3374 | 0.3269 | 0.2077 |
| 5 | BM25 | 0.3701 | 0.3701 | 0.3635 | 0.2147 |

Table 6 Result for database without removing stopwords

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DB(NOSTOP STEM) Query(NOSTOP STEM) | | | | | |
| ID | Model | MAP | R-Precision | Precision at 10 | Precision at 30 |
| 1 | OKTF | 0.2075 | 0.2097 | 0.2673 | 0.1494 |
| 2 | OKTF\*IDF | 0.3496 | 0.3694 | 0.3596 | 0.2 |
| 3 | Laplace | 0.1345 | 0.1564 | 0.1673 | 0.1115 |
| 4 | Jelinek-Mercer | 0.2969 | 0.3231 | 0.3346 | 0.1968 |
| 5 | BM25 | 0.3383 | 0.3503 | 0.3423 | 0.1974 |

Table 7 Result for Database and query with stopwords

Figure 4 Comparing Result

STOP SOP=DB(STOP),Query(STOP);

NOSTOP,STOP=DB(NOSTOP),Query(STOP);

NOSTOP,NOSTOP=DB(NOSTOP),Query(NOSTOP)

Table 6 shows the result for database without removing stopwords whereas the query still has no stopwords. Compared this table with Table 2, the result has only a small difference for all five models. Since the query still removes stopwords, when calculate the rank score, the stopwords do not have significant effect on final results.

However, when queries contain stopwords, as the Table 7 and Figure 4 shows, the MAP and R-Precision has dropped a lot. When queries have stopwords, all those stopwords will be considered as key word of query and searching the corpus to find relevant documents. And at same time, the database also has indexed the stopwords. In this case, documents that have same stopwords will be considered as relevant even thought their actual content is completed irrelevant. And according to Zipf’s law, the stopwords always has higher frequency in documents. Thus the precision of retrieval models drops a lot.