Retrieval models on the complete works of Shakespeare

- **Shibin George**, email: **shibingeorge@(cs.)umass.edu**

I. Structure of source code (only the retrieval evaluation part)

The source code for retrieval and scoring is present in 2 packages:

i. **retriever**: This package provides the query-retrieval functionality and also the implementation of various models to score/evaluate documents w.r.t queries. Its main classes are:

   a. **Retriever** is the abstract super class that every Retrieval-sub-class must implement. It remains the same as the one from the previous assignment.

   b. **Evaluator** is the abstract super class that all scoring/evaluation models should extend. It exposes 2 methods which all sub-classes must implement: **getDocScoreForQueryTerm(String queryTerm, int termFrequency, int docId)** which scores a document w.r.t a query term, and **assignsBackgroundProbability()** which returns true if the evaluator sub-class scores even those documents which don't have a query term in it by assigning some background probability, else false.

   c. **BM25Evaluator** is a sub-class of Evaluator that scores documents w.r.t a query-term using the BM25 model. Its **assignsBackgroundProbability()** returns **false** since BM25 only scores documents which have the query-term in it. Its constructor takes the complete query as argument along with the index. Using the complete query, it tracks term-frequency of query-term within the query and avoids scoring duplicate words.

   d. **JelinekMercerEvaluator** is a sub-class of Evaluator that scores documents w.r.t a query-term using the Query-likelihood model with Jelenik-Mercer smoothing. Its **assignsBackgroundProbability()** returns **true** since JM assigns background probability scores to documents which don't have the query-term in it, so essentially it scores every document in the collection. It scores a query term every time it appears in the query, unlike the BM25.

   e. **DirichletEvaluator** is another sub-class of Evaluator that scores documents w.r.t a query-term using the Query-likelihood model with Dirichlet smoothing. Its **assignsBackgroundProbability()** returns **true** since Dirichlet assigns background probability scores to documents which don't have the query-term in it, like the JM model.

   f. **RawCountEvaluator** is the basic evaluator that just returns the term-frequency in the dpcument as the score of that document. Its **assignsBackgroundProbability ()** returns **false**.

   g. **DocAtATimeRetriever** is the sub class that implements document-at-a-retrieval retrieval using the InvertedFileIndex index and is pretty much the same from assignment 1. The one major change in this class appears in the **retrieveQuery(String[] terms, int k)** method which has been extended to accept a 3rd argument of type **Evaluator**. retrieveQuery(String[] query, int k, Evaluator evaluator) decides whether to score a document based on return value for **evaluator.assignsBackgroundProbability()** and if true (or if the query-term appears in the doc), calls **evaluator.getDocScoreForQueryTerm()** for the document.

ii. **apps**: This is the package with the different applications for index-creation, query-retrieval and evaluation. The newly introduced applications in this class are:

   a. **BM25QueryRetriever**: This application supports accepts just the path to the index as the argument and retrieves documents for queries (saved in variable QUERY_SET). The output is of the same format needed by the trecrun file. When calling retriveQuery(), it sends the **BM25Evaluator** as the evaluator instance to the **DocAtATimeRetriever**.

   b. **JelinekMercerQueryRetriever**: Same as above, except that the evaluator is an instance of **JelinekMercerEvaluator.**

   c. **DirichletQueryRetriever**: Same as above, except that the evaluator is an instance of **DirichletEvaluator.**

This completes the structure of the newly introduced classes.

II. To be or not to be?

Results for this query were alarmingly disappointing. Not a single model even featured the scene "hamlet:2.0" in their top-10 results. My guess is that because the 3 models are not considering phrases but treat each term in the query independently and all the terms ("to", "be", "or", "not") being very frequent in all the scenes, this might have thrown the models off of their game!

III. "setting the scene"

"scene" appears in all of the scenes. "the" appears in 747 of the 748 scenes. "setting" appears in 24 scenes.
All the top 10 results of the 3 models had the term "setting" in them (the other terms were virtually in all the scenes so they

didn’t quite matter..). So this query performed decently?

IV. Relevance judgement on top ten results for Q3 from all models

From the 30 scenes that retrieved by the 3 models (BM25, JL & Dirichlet), there were 13 unique scenes.

(Since the relevance judgement was open to interpretation, I decided to use the results from the 3 retrieval models to assign relevance.)

To assign a relevance score, I computed the reciprocal rank of each scene in each model. Then I summed up the reciprocal ranks from 3 models, for a scene. For instance, if the scene ”richard_iii:4.2” features in the 3 models at rank 1, 2 and 1 respectively, i.e.

```
Q3   skip  richard_iii:4.2        1    5.7386903564439065   shibingeorge-bm-25-k1=1.2-k2=100.0-b=0.75
…
Q3   skip  richard_iii:4.2        2    -19.66774251106544   shibingeorge-ql-jm-lambda=0.2
…
Q3   skip  richard_iii:4.2        1    -19.89330540938452   shibingeorge-ql-dir-mu=1500
```

Then, the reciprocal rank score for ”richard_iii:4.2” would be (10/1 + 10/2 + 10/1) = 25.0.
Having done this for all the scenes, I noticed that scores of all scenes fell in the range [1.0, 25.0].

Next, I had to transform them from [1.0, 25.0] range to [1, 3] (I didn’t want to assign 0-score to any scene because if the scene appeared in the top 10 results in any of the 3 models, I assumed that they would be “somewhat” relevant).
To transform the score ***x*** from [1.0, 25.0] to [1, 3], I used the following: $\frac{(x-1)(3-1)}{(25-1)} + 1$
This brought all the scores to range of [1, 4]. So, ”richard_iii:4.2” with reciprocal rank score of **25** got assigned a relevance score of **3** (i.e. most relevant) since all 3 models featured this scene at top of their retrieval.
The scores for the 13 scenes are included in the zip in the **judgments.txt** file.
The app RelevanceJudge.java implements the mechanism I described.

V. Changes to support phrase queries or other structured query operators

Conjunctive doc-at-a-time retrieval can be used so that positional information of terms within a document is taken into consideration, to support phrase queries.
In general, for phrases and other structured query operator, the system needs to understand a Query language mechanism. An evaluation tree (described in Pg. 179, fig. 5.23 of the textbook) would help in breaking down queries into sub-parts, independently processing them and then combining them to return to the user.

VI. How does your system do?

For queries where the terms are to be treated independently, BM25/JM/Dir do a good job. On queries which have very frequently appearing terms like “to be or not to be”, the 3 models technically do their job but since these are frequently appearing terms in almost all the documents, the relevance of the results is not that great.
On queries like “setting the scene” where one of the terms is very infrequent compared to the others, all 3 models perform good enough since they retrieve the documents which have all 3 of the terms.