University of California Santa Barbara

Information Retrieval and Web Search
CS 293D

Re-ranking with RankLib

Team members:
Kristoffer Saastad
Navjot Singh

Supervisor:
Tao Yang

Fall 2018

Contents

1 Introduction

As a short justification for why the report is some pages too long: There are a lot of results and tables to be shown, as well as this project has alot of aspects to it, many features to be derived and models to be compared. For results, go to section 5.

1.1 Our Objective

During the assignments in this course, we have learned to rank documents using Lucene. Re-ranking Lucene results with RankLib to achieve good ranking.

1.2 Tools and Libraries Used

Lucene Lucene is an information retrieval software library, which was used for ranking.

RankLib RankLib is a learning to rank library [3], which was used for the re-ranking.

2 Flow of the Re-Ranking

Figure 1 and Figure 2 illustrates and explains the core flow on how all of this is done. The training flow figure shows where the TREC data is processed and where the features are derived using Lucene with top n documents. The features are then written to a file in LETOR format and passed to Ranklib, where the models are trained.

The prediction flow figure shows how Lucene only retrieves the top 10 documents based on a single query. Then writes the same features as for the training data to a predict file in LETOR format. At the end it loads the trained models and re-rank the retrieved documents.

2.1 Training

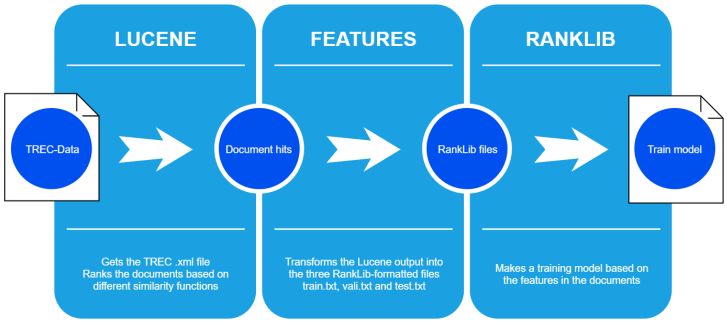

Figure 1: Training flow

2.2 Prediction

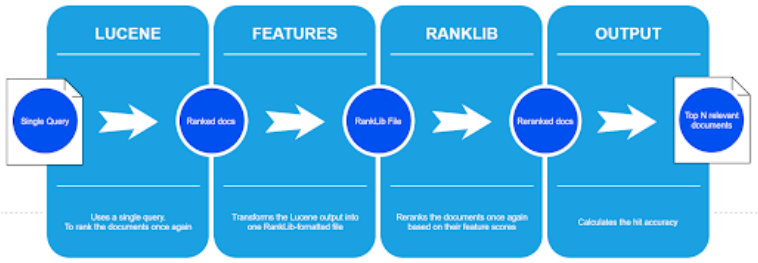

Figure 2: Prediction flow

3 Data Used

We acquired the TREC disk 4 and 5 dataset (**trec-disk4-5.xml**), with corresponding judge labels (**qrels.trec6-8.nocr**) and queries (**title-queries.301-450**). This xml-file contains 550k documents. We used the whole dataset for our results. For the corresponding judge labels and queries we usesd 150 queries and had about 1500 judge labels per query.

4 Deriving Features with Lucene

Throughout the whole project we used the newest version of Lucene, version 7.5.0. Our first task was to make Lucene work with RankLib. The features used are example wise bigrams, n-grams, cosine similarity and more, which are discussed more in detail in the underlying sections.

4.1 Data Processing

4.1.1 Offline and Online

In order to acquire a better match for words in the queries and the documents, one removes/filter away certain high frequent words that decreases the context of the two. Another aspect to increase the match chance is to use stemming. This transforms the words to a simpler form of the same word. Lucene provides a built in class called PorterStemming, which do the stemming. These two methods are implemented offline for the whole document dataset while it is indexed, and online for the query words before comparing the two. Only using stopword filtering or stemming for one of the query or document will not make the result better, since the form of the words and documents might have been transformed. Prior to these methods, the query terms are transformed to lower case.

4.1.2 N-Gram

To increase the context of the query to a given document, the positioning of the query terms to each other matters. Using n-grams is a solid method for this. Instead of using each query term alone to check for query document matches, one has to acquire a hit for if the words are next to each other, i.e. creating shingles of the query. Given that the query length is not constant, the shingle length used in the implementation are bigram and max-gram. The meaning of max-gram is that one creates one shingle for the whole query, and if this gets a hit, the whole query in the same order in the document.

4.1.3 Relevancy Label

RankLib has an option to use different levels of relevancy for each query-docuemnt pair. The original source code only has binary judge labels. Using n-grams, the hits can be justified of being more relevant if the match is between one word, or multiple words next to each other. By using max-gram to determine the relevancy label, the relevancy label is multiplied by 2 if the document retrieved has a match with the whole query. The irrelevant judge labels (relevant = 0) does not get affected by this.

4.2 LETOR Format

This is a way to format the input data RankLib uses. Since RankLib is an advanced learning to rank library, it is crucial that the input given is in the right format. This input should contain a multiple of query → document pairs. For each query → document pair there are judge labels and feature weights, which are used to rank documents. This format is called LETOR (LEarn TO Rank). The amount of weights used is completely up to the implementer and the data available, but the more features used, the more weights the models has to determine a success or not.

Below is an example of the input format required by RankLib:

```
Each Line should contain this
<relevancy/judge label> qid:<query id> <feature>:<value> <feature>:<value> ...
<feature>:<value> # <document number or other info>
```

4.3 Lucene General

Lucene uses a file called build.xml as config file to initialize correctly. This specifies the default command line for both BatchSearch and IndexTREC, which are the java classes used in this project. The build file also instantiate and compile all the java classes.

4.3.1 IndexTREC-Class

Lucene uses index files to look up and search the documents in the dataset. IndexTREC creates these index files by going through each document. Changes to be made

or fields to be stored is added in the IndexTREC.java and TrecDocIterator.java classes. IndexTREC is only run when files or datasets are changed.

4.3.2 BatchSearch-Class

Batch search is, as the name of the class implies, searching through batches of the dataset by using a Lucene class called IndexSearcher, which uses the indexed files from IndexTREC. One batch means one query to all matched documents. The code implemented to extract the features is based on the trec-demo code used in HW1.

The original code only has enough skeleton code to index and do a basic batch search and return the similarity score for the hit documents. The length of the code was changed from 137 to 761 lines of code.

4.3.3 Automating the Process

In order to automate the process of indexing the data, using the right queries, sending the outputs to inputs, training the models, predicting etc, we had to make a file that could do it all for us! main.py is the core of all this. Here is a list of some key features:

1. Holds the right parameters given to both Lucene and RankLib.
2. Controls the interaction between these two libraries.
3. Keeps track of the right input/output files.
4. Generates training models and ranked models with the usage of RankLib

4.4 Feature Extraction

So for the part where the actual features are derived. There are a total of 34 features derived from the documents. To start of lets talk about the training parameters used for the extraction.

Table 1 shows the parameters used for the extraction. To justify the content, lets jump into the parameters. The max number of hits means how many the maximum amount of document retrieved for the batch search will be. The number of 250 is the middle ground of creating a sizable training data set, yet keep it small enough for the training process to not be too long, in addition to not overfit the models for the 150 queries available. This means that each query has a maximum of 250 hits, and in total the dataset generated will contain 37,500 training examples.

The chosen fields are according to where the queries would more likely give a query hit. The whole document, meaning every field, would give us the highest rate of hits, but will tell us nothing about where it actually hit. By searching through specific field, one can get a sense of which fields give a relevant result.
The similarity functions used are the standard, build in Lucene similarity functions.

Parameter	Value
Max number of hits	250
Number of queries	150
Fields	Whole, Title, Body, In
Similarity functions	TFIDF, BM25, DFR, LM

Table 1: Parameters used for the feature extraction

4.4.1 General Features

For each of the similarity functions, for each part, the following different features are derived: Similarity score for part, Term Frequency and Inverse Document Frequency.

4.4.2 Lucene .explain()

Lucene's "explain"-function [1] is a very useful method to acquire useful information that can be used to derive the features, but at a cost of time complexity, since explain accesses the index directory. Explain passes in the query and the desired document you would want to explain. It returns an overview of a term's score to the document, which contains the term frequency and inverse document frequency, as well as the normalization factor.

4.4.3 Part of Document

To index the specified field, one would have to create tag, matchers and groups for each of the fields, and then add the field to the indexed document. The content for each field is stemmed in this process. To justify the part of the documents chosen as features, lets start of with the title field. Given that the queries given is named title-queries.300-405, how each document hit scores for only the field title is valuable as a feature. The body of the document contains the most text and would be the most likely field for a random given query term to hit. The In field contains in which article the document was found. Having a matching query for this field would result in a lot of information. For each similarity, all the features are derived.

4.4.4 N-gram as Features

As explained in section 4.1.2, the n-grams will change the judge label into a higher relevancy label for the query–>document pairs as the match in shingles is a very valuable feature since it tells the score when the words appear together. The similarity scores for all the similarity functions for both the bigrams and the max-grams are derived and passed as features.

4.4.5 Cosine Similarity

Another valuable feature is to know how the query is similar to the document. To calculate this one uses the information extracted from explain, using the term frequency in the document and the term frequency in the query, as well as the inverse document frequency. Imagine the terms being represented in a n-dimensional vector space,

where each term represents an axis. The cosine similarity shows the angle between the document and the terms, which is a similarity measure.

Formula: [4]

*Cosine Similarity(Query,Document) = Dot product(Query, Document) / ||Query|| * ||Document||*

4.4.6 Document Length

The document length is easily derived since the document is already retrieved. Basically what is done is taking the string length of the document.

Feature	Value	Feature	Value
1	TFIDF of whole document	18	DRF of title
2	TFIDF of title	19	DRF of body
3	TFIDF of body	20	DRF of in
4	TFIDF of in	21	LM of whole document
5	TF of whole document	22	LM of title
6	TF of title	23	LM of body
7	TF of body	24	LM of in
8	TF of in	25	Bigram TFIDF of document
9	IDF of whole document	26	Bigram BM25 of document
10	IDF of title	27	Bigram DRF of document
11	IDF of body	28	Bigram LM of document
12	IDF of in	29	Maxgram TFIDF of document
13	BM25 of whole document	30	Maxgram BM25 of document
14	BM25 of title	31	Maxgram DRF of document
15	BM25 of body	32	Maxgram LM of document
16	BM25 of in	33	Co-similarity of query $\rightarrow$ document
17	DRF of whole document	34	DL of whole document

Table 2: List of the 34 features extracted

5 RankLib and Results

An option to choose from when using RankLib is using k-fold cross validation, which splits the data three times in train, validation and test data. Each folder (k number of folders) uses the whole set, but different in which part is used for what file, in other words the model will train k number of times on k different sets.
We used two different LETOR files, all using 3-fold cross validation. The two train sets were used for generating the training models, one containing shuffled query results, and the other being sorted based on queries. Each line in the train sets contains a set of features with their values, and is marked with a relevancy label between 0 and 3, which is based on different factors, as explained in section x.x.

The reason for using shuffled was to assumption of generalizing the dataset, since our assumption was that the constituent order of the query-document pairs is biased. The consistent order of the pair turned out to be a requirement for RankLib, and by that concludes the shuffled data useless (for curiosity: training the models using shuffled data gave an average of 20% accuracy).

We used three tree-based models (AdaRank, MART and LambdaMART) and one neural-net-based model (RankNet) for the training. Each using 3-fold cross validation and NDCG@10 as metric. The tree based models trained 1000 trees, and used 256 threshold candidates. RankNet is a neural network. In order to test the network its extent, we used 2 hidden layers with each hidden layer containing 30 nodes. RankNet was trained for 100 epochs.
Talking a little bit about over fitting, RankNet and AdaRank didn't improve much after the earlier stages of the training, but MART and LambdaMART kept increasing the validation accuracy throughout the 1000 epochs.

Before looking at the results, an assumption to be made is that one can expect LambdaMART to give the best result, based on previous experience and from the lectures.

As seen in the figures below, the results gave us the following value:

NDCG@10	3-Fold	
	Train	Test
Baseline	-	0.2967
LamdaMART	0.8036	0.3725
MARTA	0.6964	0.3454
AdaRank	0.3316	0.3343
RankNet	0.3616	0.3616

Table 3: 3-fold cross validation training and test accuracy results

As we can see, all the trained models was an improvement over the baseline NDCG@10 score, which means the features derived and passed to RankLib actually increased the accuracy of the predictions. As the assumption that LambdaMART would give the best results holds up, the results are satisfying. As a quick note, we looked at a paper matching models for ad-hoc retrieval [2], and the baseline result for BM25 was 0.418, meaning the models could be better trained using more features, or more labeled data, but all in all having a best score of 37% accuracy is ok.

	LamdaMART	MART	AdaRank
Feature nr.1	6	5	17
Feature nr.2	23	7	23

Table 4: Top 2 scoring features for the models

Simply stating the explanation for each feature:
LambdaMART: Feature 6 TF of Title, Feature 23 LM of Body
MART: Feature 5 TF of whole document, Feature 7 TF of body

AdaRank: Feature 17 DRF of whole document, 23 (LM of body)
The top features can be read from the model files. For LamdaMART and MART, a heat map is derived to display the most important features at a given state of the training.

The objective of this project was to rerank the top 10 documents retrieved from Lucene. Below is a list of the resulted position and reranking of the models. Given lambdaMART being the best model, one would use this as a final pass to a potential user.

DOCNO	Lucene	LamdaMART	MART	AdaRank
FBIS4-33622	0	0	0	0
FT922-9507	1	5	5	1
FBIS4-21356	2	7	6	2
FBIS4-28692	3	6	4	4
FBIS3-47058	4	4	2	3
FBIS3-20608	5	3	8	6
FBIS3-6545	6	9	7	7
FBIS3-29527	7	2	1	8
FBIS4-48928	8	8	3	5
FBIS4-33116	9	1	9	9

Table 5: Reranked lists for each of the trained models compared to the retrieved documents returned by Lucene. 0 is the highest rank, 9 is the lowest

6 Challenges and Problems

Since features are a crucial part of generating models using RankLib, they had to be extracted properly and efficiently in Lucene. This was challenging. We had to rewrite huge parts of the code directly in Lucene, without messing it up. And we had to make sure that we got the right results that we needed, especially when we were working with this huge amount of data. Getting the relevancy labels was also a challenge.

7 Conclusion

We have used a lot of different techniques that we have learned throughout the course during this project. Here is a list:
The usage of open source search engines like Lucene. Indexing, query processing, co-similarity measurement, usage of different algorithms like TF-IDF, BM25, DRF and LM. Search evaluation using NDCG@10 (Normalized Discounted Cumulative Gain), usage of training and test sets, scoring based on different features, document ranking and re-ranking, classification with the usage of k-fold, usage of n-grams in query search, online and offline data processing, stemming and stopword removal and the importance of formatting correct data (learning to rank-format). Using all of this, we have derived

34 features, giving a better accuracy that the baseline NDCG@10 score and created a script to re-rank the top 10 documents for any given query.

8 Setup and User Manual

Just follow these steps

1. Make a folder. This folder should contain the repository mentioned below.
2. Get this repo by using git clone in your folder

   `https://github.com/RovelMan/293d-Project-Reranking`
3. Before running main.py, there are 3 parameters to be set:

   (a) Feature Extraction
   feat_extract = True or False
   This will run the Lucene part of the script, with executing the build file, IndexTREC and BatchSearch. This may take some time, base on the size of the dataset used. The dataset is also partitioned into train and set data. Since we use k-fold cross validation, it does not generate a validation data.

   (b) Train models
   train_models = True or False
   This will train all the models listed in the *train_ model* list, test it with the test file, and compare the models.

   (c) Predict
   predict = True or False
   This will retrieve the top 10 documents from Lucene, and re-rank them using the trained models. The models to include can be put in the *pred_ model* list.

   (d) These are the default settings if parameter input is missing:
   feat_extract = False
   train_models = True
   predict = False
4. We have already extracted the features, but the models still need to be trained and predicted. To do this run the main.py file using the commands in the given order below:

   ```
   # Train models - assuming you've already extracted the features
   python main.py --feat-extract False --train-models True --predict False

   # Predict - assuming you have trained models
   python main.py --feat-extract False --train-models False --predict True
   ```
5. If you want to extract features from an other trec dataset run this command before doing the ones listed in the previous step

```
# Extract features - assuming you have the trec file in the following folder
# trec-demo-master/test-data/<your trec file>.xml
python main.py --feat-extract True --train-models False --predict False
```

References

[1] IndexSearcher, Lucene 7.5.0 API, 2018.
https://lucene.apache.org/core/7_5_0/core/org/apache/lucene/search/IndexSearcher.html

[2] Jiafeng Guo, Yixing Fan, Qingyao Ai and W. Bruce Croft.
A Deep Relevance Matching Model for Ad-hoc Retrieval
University of Massachusetts, 2016.
http://www.bigdatalab.ac.cn/ gjf/papers/2016/CIKM2016a_guo.pdf

[3] RankLib, The Lemur Project, SourceForge
https://sourceforge.net/p/lemur/wiki/RankLib

[4] Jana Vembunarayanan
If-Idf and Cosine similarity
Seeking Wisdom, October27, 2013
https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/