Information Retrieval Project 2, Group 11

Michael Merki & Markus Greiner - December 2016

# Installation

See README.md how to install and run using sbt.

# Preprocessing / Tokenizing

**TODO**

# Term-based scoring models

Our term-based model consists of two different scoring approaches. On the one hand we implemented the scoring based on tf-idf weights described in the lecture (lecture 4, slide 14). In a first attempt we directly used raw term frequencies which resulted in a bias towards longer documents. In order to avoid such biases we extended the standard tf-idf weighting scheme by using the augmented term frequency instead of the raw term frequency. The augmented term frequency was calculated according to the following formula:

With this extension we were able to increase the performance metrics of ranking the provided training documents for the 40 test queries as shown in the following table:

|  |  |  |
| --- | --- | --- |
| Metrics | TFIDF with raw tf | TFIDF with augmented tf |
| Precision | 0.314 | 0.400 |
| Recall | 0.328 | 0.415 |
| F1 Score | 0.319 | 0.405 |
| MAP | 0.206 | 0.277 |

Table 1: Comparison between raw term frequency and augmented frequency weighting

On the other hand we used the idea of the vector space model described in the lecture (lecture 4, slides 15 – 17) and computed the cosine similarity between the documents and queries represented as vectors.

Both scoring approaches can be found in the class "TermBasedModel" and can be run with and without inverted index. In case of running the scoring with indexing we compute the needed extra information like the maximum frequency per document (in case of the augmented tf-idf weighting) or the norms of the document vectors (in case of the cosine similarity measure) directly based on the information available in the inverted index. In case of not using indexing we make two entire iterations over all documents. In the first round we precompute document frequencies for all terms in order to be able to compute the tf-idf score in the second round. During the second iteration over the document collection we compute the score for each document and keep a sequence of the top-n ranked documents.

# Language scoring models

As our language model we implemented the maximum likelihood estimation presented in the lecture (lecture 6, slides 5-8) and applied the proposed Jelinek-Mercer smoothing to get better estimates for small frequencies. Experimenting with varying the tuning parameter λ showed optimal precision, recall, f1 and MAP measures for λ = 0.2.

The implementation of the language model can be found in the class "LanguageModel". It is possible to run our language model with and without indexing. In case of using the inverted index the required information like maximum frequency per document is directly computed based on the index. For the variant without indexing we make two entire iterations over the document collection. During the first round we precompute collection frequencies as well as the total amount of terms in the document collection in order to be able to apply the described scoring with smoothing in the second iteration. Like in the case of the term-based models we just keep the top-n ranked documents after each iteration of the second round.

# Training data performance

Evaluating the performance of our three scoring models for the provided 40 test queries and 100'000 training documents resulted in the following precision, recall, f1 score and mean average precision (MAP) metrics:

|  |  |  |  |
| --- | --- | --- | --- |
| Metrics | TFIDF | Cosine Similarity | Language Model |
| Precision | 0.400 | 0.342 | 0.407 |
| Recall | 0.415 | 0.355 | 0.421 |
| F1 Score | 0.405 | 0.346 | 0.412 |
| MAP | 0.277 | 0.198 | 0.280 |

Table 2: Precision, recall, f1 score and MAP metrics for our three scoring models

Note: For computation of recall and MAP we bounded the denominator by

# Approaches for improving training data performance

We implemented several extensions to the above described scoring models with the goal to improve the training data performance. These included:

* Character n-grams: One of the major challenges when working with the tipster data set is in our opinion spelling errors. In order to become more tolerant against such spelling mistakes we tried to create an inverted index with character n-grams. Our class "TipsterParseSmart" has an option "ngramSize" which allows specifying the size of the n-grams to be extracted from the document tokens. Testing various n-gram sizes unfortunately didn't result in improving the training data performance.
* Query expansion through synonyms/terms from top ranked documents: Because the provided training queries were rather short (most often just a couple of terms) we had the idea to use query expansion to improve training data performance. Our first approach was based on doing a first round of scoring and only returning the top n ranked documents with n being quite small (e.g. 10). From these "very" relevant documents we than extracted the most frequent words and appended them to the query terms. After that we did a second round of scoring and returned the top 100 documents. Because this didn't lead to an improvement we also tried query expansion by synonyms. For this we used a dictionary with synonyms from the WordNet dataset. Before evaluating a query we extended the original query terms by synonyms from this dataset. Unfortunately also this didn't result in an improvement of the training data performance.

# Challenges during development

Our initial challenge was limited heap size. During all our developments we set maximum allowed heap size to 4 GB which seemed to be too restrictive when working directly with the class "TipsterStream" from the provided library "tinyir". In order to work around this problem we extended this class and …. **TODO**

After solving the problem with the limited heap size the creation of the entire index took around 750 seconds which was not very practical for implementing and testing our scoring models. That was the reason why we made use of "LevelDB" and introduced a class "PersistentFreqIndex" which has the functionality to make the inverted index persistent and recreate the index from disk again. Recreating the inverted index for the entire document collection from the database takes around 50 seconds.

# Running times

The comparison of running the three scoring models with and without the inverted frequency index resulted in the following average running times per query:

|  |  |  |  |
| --- | --- | --- | --- |
| Running Time | TFIDF | Cosine Similarity | Language Model |
| Creating inverted index | ~ 750 secs | | |
| Recreating inverted index from the database | ~ 50 secs | | |
| Computing additional statistics based on the inverted index (\*) | < 20 secs | | |
| Average query running time **with** inverted frequency index | < 1 sec | < 1 sec | < 1 sec |
| Average query running time **without** inverted frequency index | ~ 3650 secs | ~ 2250 secs | ~ 2100 secs |

Table 3: Running times for index creation and running queries with/without index

Note: For these experiments we used a Windows Machine with Intelcore i/ 4500u processor, 1.80 GHz with 8 GB RAM. The maximum heap space was set to 4 GB.

\* This includes calculating statistics like document frequencies which was done directly on the information available in the index (see chapters term-based/language scoring model for more details). Because running this routine didn't take longer than 20 seconds we didn't bother about improving the performance of this step. Obviously ideal would be to also create these statistics directly when creating the inverted index and potentially also make them persistent in LevelDB in order to faster respond to queries.