Introduction

Our implementation is very straightforward - yet highly modularized such that individual pieces can be easily replaced. The architecture is described in more detail in the following section. As a term-based model we use a tf-idf (vectorspace) model and as language model we use a simple implementation of an unigram language model, similar to the one discussed in the lecture. What sets our ranking apart is that we use a fancy hit similar to the initial Google paper for words which appear in the title of a document and that abbreviations are preprocessed.

System Architecture

A `DataReader` goes through all the documents and builds the dictionary, inverted index (if desired) and other data structures. The class `InvertedIndex` uses those data structures to provide information to the `RankingModel`s which implement the actual ranking.

We found early on that the inverted index using a HashMap is large but still fits into memory (it needs around 2.1-2.2 GB)[1], so we did not need to optimize for the size of it. Furthermore the only part that lasts a long time is the actual creation of the inverted index. Queries are reasonably fast (on average 0.1 seconds in the default implementation). Hence we for example use naive list intersection without further optimization.

As we ran into a memory issue at a point during the development we also created a version that uses levelDB. We did not use levelDB to store the inverted index in the end, but only for normalization factors in the term-based model. However we sill benchmarked the inverted index with levelDB.

Running the model with and without inverted Index

When we create the inverted index, we save the words and the words counts and hence have no need to store whole documents. So we only need one pass through the data. To measure the performance of queries without an inverted index we create the `PassThroughInvertedIndex`, which provides the same interface as the `InvertedIndex`, but does not use an inverted index to answer queries. Without an inverted index every query requires a full pass through all the documents. Therefore $n+1$ passes through all the documents are needed where n is the number of queries (the first pass is to create the dictionary). In order to decrease the number of passes required for the case without an index, we added batch mode to our ranking model. This allows to perform multiple queries at once. In batch mode we just need 2 passes to answer the n queries, one to build the dictionary and one to find the relevant documents for the queries. An exception to the above is when running the model for the very first time. An additional pass is needed to create the vector normalization database.

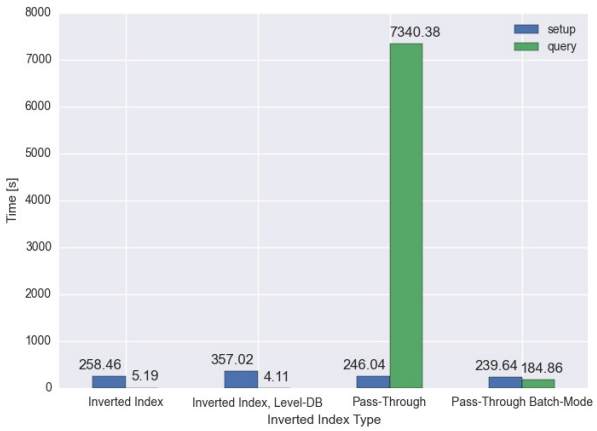

Figure 1: The data is averaged over 2 runs for the pass-through mode and 5 runs for all others. measurements were performed on Ubuntu with OpenJDK 8, and Java heap constrained to 4 GB. If we run the model without memory constraints it needs about 17 GB and is slightly faster. **setup** shows the time for the first pass through the data (creating the inverted index), **query** shows the time to run all training queries.

Figure 1 shows the time breakdown for the various configurations using the language model. From an information retrieval standpoint all of those settings result in the same output; all report the same orders and metrics. The time differences for the term-based model are very similar.

The setup time is for the initial pass through the data and dictionary creation (and inverted index creation, if wanted). For the default implementation we need 258 seconds on average to perform the setup and only around 240 seconds if we don't create an inverted index (pass-through mode). Storing the inverted index in levelDB takes slightly longer. Answering the queries is very fast when using an inverted index (4-5s for 40 queries). Without an inverted index we roughly need 180 seconds per query as we need to pass through all the documents for every query. In batch mode we also need around 180 seconds for one pass through the documents, but can answer all queries in that time.

This section shows the importance of an inverted index. Were we to implement an end-user application the obvious choice would be to use a database. The setup time is slightly higher, but only needs to be run once. Queries are the fastest and this has less impact on the memory when running in production. Our levelDB has 2.6 GB on disk. This is slightly higher than the actual inverted index, but could probably be lowered with more efficient serialization. If we

[1] It is hard to accurately measure this in the JVM, as heap analysis tools need ages to calculate the overall size of an object.

were to build a data warehousing application that answers a lot of queries, but we don't care about the latency of a single one, we could actually run in batch mode without an inverted index.

For the development we used the simple in-memory inverted index as it has the lowest overall running time and it is by far the simplest.

Data preprocessing and Index construction

As the first preprocessing step we replace certain abbreviations (e.g. "U.S.A", "U.S", "United States of America", ...) in the raw documents by a special expression (e.g. "united-states-america"). The goal of this approach is to get more matches for different terms which share the same meaning. We then read the tokens from the documents using the tinyIR library, remove tokens not consisting of letters, stem every word and remove stopwords. While preprocessing each document we directly create an inverted index in memory storing for each word a list of the documents it appears in. Furthermore we store the number of occurrences of the word in the document as well as whether the word appears in the title of the document.

Ranking models

Vector space model

We first explore the term-based vector space model. Each document and query can be represented as a vector, where the indexes of the vector represent the words considered, and the weights are the tf-idf scores of the given word. Then, we rank the documents according to the following formula : $\text{score}(q,d) = \frac{\vec{V}(q)\cdot\vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)|}$. The different ways of representing vectors as well as normalizing them leave room for flexibility. More precisely, we can use combinations as described by the SMART notation[2]. For tf weights we consider natural tf, logarithmic tf, and boolean weighting. For idf we consider logarithmic idf, probabilistic idf, and no idf-weighting at all. We normalize vectors using their cosine length, or not at all. Considering that these combinations can be applied to both documents and queries, this results in 18^2 possible combinations of vector representations and normalizations, although a lot of these representations lead to the same result. We achieve the best scores on the provided data using the *lnn.npn* vector representation (logarithmic tf for documents, probabilistic idf for queries, and no normalization) with a fancy hit bonus of 10.0. The corresponding Mean Average Precission (MAP) score was 0.219 when using the bounded Average Precission (AP) denominator.

To simplify the model, the vector dimensions we use are the terms present in the query as opposed to the whole dictionary. This does not change the product $\vec{V}(q)\cdot\vec{V}(d)$ and leads to much faster computation. However, this requires pre-computation of the cosine vector norms $|\vec{V}(d)|$ for the 9 different vector representations, as they are dependant on the full vector dimensionality. We store all these values in a levelDB database called "VECTORNORMS".

Language model

One option for computing scores for the individual documents is to use a language model. We use the following formula as a base for our implementation : $score(d,q) := P(d|q) = \sum_{t\in q}\log((1-\lambda)P(t|M_c) + \lambda P(t|M_d))$ where M_d is the language generated by document d, and M_c is the language model genereated by the whole collection (same for every document). More precisely, $P(t|M_d) = \dfrac{tf_{t,d} + \gamma(t,d)}{L_d + \zeta}$ and $P(t|M_c) = \dfrac{cf_t}{T}$.

There are three hyper-parameters to be optimized : λ balances the document and collection languages and allows for smoothing. γ is the fancyhit bonus awarded if the term is present in the document title. ζ is added to the document length to decrease the unfair advantage of short documents compared to longer ones. ζ also matches the intuitive explanation that some words implicitly belong to any document's language, even if they are actually not present in the document. We performed a grid-search to find the optimal values for these hyper-parameters : $\theta = 0.75$, $\zeta = 200$, $\gamma(t,d) = 3.0$ (if t present in d's title): This results in a MAP score of 0.318 when using the bounded AP denominator.

Comparison of models

Apparently the Language model leads to much better results than the term-based Vector Space model, even though the latter allowed for a lot of flexibility. One drawback of the vector-space model is the normalization choice : If we do not normalize, longer documents have an advantage as the term frequencies are generally higher. If we do normalize, we only consider how the term frequencies relate to each other. A short document that contains, by chance, one of the query terms, will have a very high score. Both solutions are very extreme and are therefore not optimal. Using the pivoted normalization could have lead to better results, even though we did not have time to try these. The language model is intuitive, simple, and allows smoothing of the model with the hyper-parameters ζ and λ. This leads to a better handling of extreme documents lengths, leading to overall better results.

[2] `http://nlp.stanford.edu/IR-book/html/htmledition/document-and-query-weighting-schemes-1.html`