TREC 2007 Million Queries Track - IBM Haifa Team

The Million Queries Track ran for the first time in 2007.

Quoting from the track home page:

- "The goal of this track is to run a retrieval task similar to standard ad-hoc retrieval,
  but to evaluate large numbers of queries incompletely, rather than a small number more completely.
  Participants will run 10,000 queries and a random 1,000 or so will be evaluated. The corpus is
  the terabyte track's GOV2 corpus of roughly 25,000,000 .gov web pages, amounting to just
  under half a terabyte of data."

We participated in this track with two search engines - our home brewed search engine Juru.

The official reports and papers of the track should be available sometimes in February 2008, but here is a summary of the results and our experience with our first ever Lucene submission to TREC.

In summary, the out-of-the-box search quality was not so great, but by altering how we use Lucene (that is, our application) and with some modifications to Lucene, we were able to improve the search quality results and to score good in this competition.

The lessons we learned can be of interest to applications using Lucene, to Lucene itself, and to researchers submitting to other TREC tracks (or elsewhere).

Training

As preparation for the track runs we "trained" Lucene on queries from previous years tracks - more exactly on the 150 short TREC queries for which there are existing judgments from previous years, for the same GOV2 data.

We build an index - actually 27 indexes - for this data. For indexing we used the Trec-Doc-Maker that is now in Lucene's contrib benchmark (or a slight modification of it).

We found that best results are obtained when all data is in a single field, and so we did, keeping only stems (English, Porter, from Lucene contrib). We used the Standard-Analyzer, with a modified stoplist that took into account that domain specific stopwords.

Running with both Juru and Lucene, and having obtained good results with Juru in previous years, we had something to compare to. For this, we made sure to HTML parse the documents in the same way in both systems (we used Juru's HTML parser for this) and use the same stoplist etc.

In addition, anchor text was collect in a pre-indexing global analysis pass, and so anchors of (pointing to) pages where indexed with the page they point to, up to a limited size. The number of in-links to each page was saved in a stored field and we used it as a static score element (boosting documents that had more in-links). The way that anchors text was extracted and prepared for indexing will be described in the full report.

Results

The initial results were:

Run	MAP	P@5	P@10	P@20
1. Juru	0.313	0.592	0.560	0.529
2. Lucene out-of-the-box	0.154	0.313	0.303	0.289

We made the following changes:

1. Add a proximity scoring element, basing on our experience with "Lexical affinities" in Juru.
   Juru creates posting lists for lexical affinities.
   In Lucene we used augmented the query with Span-Near-Queries.
2. Phrase expansion - the query text was added to the query as a phrase.
3. Replace the default similarity by Sweet-Spot-Similarity for a better
   choice of document length normalization. Juru is using
   pivoted length normalization
   and we experimented with it, but found out that the simpler and faster sweet-spot-simiarity
   performs better.
4. Normalized term-frequency, as in Juru.
   Here, tf(freq) is normalized by the average term frequency of the document.

So these are the updated results:

Run	MAP	P@5	P@10	P@20
1. Juru	0.313	0.592	0.560	0.529
2. Lucene out-of-the-box	0.154	0.313	0.303	0.289

3. Lucene + LA + Phrase + Sweet Spot + tf-norm	0.306	0.627	0.589	0.543

The improvement is dramatic.

Perhaps even more important, once the track results were published, we found out that these improvement are consistent and steady, and so Lucene with these changes was ranked high also by the two new measures introduced in this track - NEU-Map and E-Map (Epsilon-Map).

With these new measures more queries are evaluated but less documents are judged for each query. The algorithms for documents selection for judging (during the evaluation stage of the track) were not our focus in this work - as there were actually two goals to this TREC:

- the systems evaluation (our main goal) and
- the evaluation itself.

The fact that modified Lucene scored well in both the traditional 150 queries and the new 1700 evaluated queries with the new measures was reassuring for the "usefulness" or perhaps "validity" of these modifications to Lucene.

For certain these changes are not a 100% fit for every application and every data, but these results are strong, and so I believe can be be valuable for many applications, and certainly for research aspects.

Search time penalty

These improvements did not come for free. Adding a phrase to the query and adding Span-Near-Queries for every pair of query words costs query time.

The search time of stock Lucene in our setup was 1.4 seconds/query. The modified search time took 8.0 seconds/query. This is a large slowdown!

But it should be noticed that in this work we did not focus in search time, only in quality. Now is the time to see how the search time penalty can be reduced while keeping most of the search time improvements.

Implementation Details

- Contrib benchmark quality package was used for the search quality measures and submissions.
- In order to experiment with various length normalizations the most straight forward way would have been to create a separate index for each option. But this was unreasonable for the amount of data, and for the flexibility that was required in order to try new things. So we indexed and searched like this:
  - Omit norms for the (single) text field.
  - index document length (number of terms) in a dedicated field.
  - index document unique length (number of unique terms) is another dedicated field.
  - index number of in-links (see anchors above) in a third dedicated field.
  - Create an !indexReader that implements norms() by reading (caching) the document length field and either:
    - Imitate the stock Lucene by normalizing the length with DefaultSimilarity and then compressing it to a single byte.
    - Similarly imitate SweetSpot similarity.
  - For pivoted length normalization tests (which are not listed in the results below because they were outperformed with the simpler sweet-spot similarity) we used a regular index reader (so norms were 1) and used a CustomScoreQuery (search.function) with a ValueSourceQuery part that did the normalization - it read the unique length field and used the average on it to compute the pivoted norm.
  - In-links static rank scoring was also implemented with CustomScoreQuery - with a FieldScoreQuery that read the cached in-links count.
- At some point I tried to accelerate the search and improve its quality by creating a new query - an OR with Phrase with Proximity query. This query should have been faster (I hoped) because it read each posting just once. This is in oppose to creating a SpanNear query for each pair of query words, in addition to a phrase query and an or query. But the results were very disappointing: search time was not improved, and quality was hurt. Might just be a bug... But I learned to appreciate even more the modularity of Lucene queries.

Here is an example Lucene Query. For the input query (topic):

```
    U.S. oil industry history
```

the following Lucene query was created:

```
oil industri histori
(
  spanNear([oil, industri], 8, false)
  spanNear([oil, histori], 8, false)
  spanNear([industri, histori], 8, false)
)^4.0
"oil industri histori"~1^0.75
```

This demonstrates that:

- U.S. is considered a stop word and was removed from the query text.
- Only stemmed forms of words are used.
- Default query operator is OR.

- Words found in a document up to 7 positions apart form a lexical affinity. (8 in this example because of the stopped word.)
- Lexical affinity matches are boosted 4 times more than single word matches.
- Phrase matches are counted slightly less than single word matches.
- Phrases allow fuzziness when words were stopped.

⚠ (1) todo: refer to payloads (2) todo: describe tf normalization implementation.

More Detailed Results

Following are more detailed results, also listing MRR, and listing various options.

Run	MAP	MRR	P@5	P@10	P@20	Time
1. Lucene out-of-the-box	0.154	0.424	0.313	0.303	0.289	1.349
2. LA only	0.208	0.550	0.409	0.382	0.368	5.573
3. Phrase Only	0.191	0.507	0.358	0.347	0.341	4.136
4. LA + Phrase	0.214	0.567	0.409	0.390	0.383	6.706
1.A Sweet Spot length norm	0.162	0.553	0.438	0.400	0.383	1.372
1.B tf normalization	0.116	0.436	0.298	0.294	0.286	1.527
1.C Sweet Spot length norm + tf normalization	0.269	0.705	0.562	0.538	0.495	1.555
4.A LA + Phrase + Sweet Spot length norm	0.273	0.737	0.593	0.565	0.527	6.871
4.B LA + Phrase + tf normalization	0.194	0.572	0.404	0.373	0.370	7.792
4.C LA + Phrase + Sweet Spot length norm + tf normalization	0.306	0.771	0.627	0.589	0.543	7.984

There are some peculiarities, for instance the fact that tf-normalization alone (without sweet-spot length normalization) hurts MAP, both on top of stock Lucene (run 1.B vs 1) and on top of "LA + Phrase" (run option 4.B vs. 4) but other than that results seem quite consistent. For instance Sweet spot similarity improves with almost no runtime cost - this shows in both 1.A vs. 1 and 4.A vs. 4.

Possible Changes in Lucene

- Move Sweet-Spot-Similarity to core
- Make Sweer-Spot-Similarity the default similarity? If so with which parameters?
  In this run we used steepness = 0.5, min = 1000, and max = 15, 000.
- Easier and more efficient ways to add proximity scoring?
  For example specialize Span-Near-Query for the case when all subqueries are terms.
- Allow easier implementation/extension of tf-normalization.

⚠ To be completed & refined...