SummerOfCode2011ProjectRankingTerrier

Return to main page

A short overview of Terrier's scoring architecture

Terrier is another Java-based, open source search engine developed at the School of Computing Science, University of Glasgow. It is positioned as a platform for IR research, and as such, has an extensible ranking architecture, with many ranking functions, such as LM, BM25 and the DFR framework already implemented. Therefore, it may be worthwhile to include a short summary and analysis of their architecture here.

The Terrier ranking architecture

The interface `Model` is top of the model (*Similarity*) hierarchy.

`WeightingModel`: abstract implementation of `Model`. It duplicates the information in the `CollectionStatistics` and `EntryStatistics` objects.

- `Idf` class as a member field
- Query term frequency: `keyFrequency`
- `score()` is abstract
- 2 {{score()}}s:
  - `tf`, `docLen`
  - `tf`, `docLen`, `df`, `tf(Corpus)`, `keyFrequency`
    What I find strange in this implementation is that they felt the need for the second `score()` method. In case of e.g. BM25, the two-parameter version takes the required values, such as *df*, from `WeightingModel`, while some of these can be supported directly to the five-parameter method. However, the choice of parameters seems rather arbitrary.

There are two classes responsible for providing the statistical parameters to the model.
`CollectionStatistics` stores the collection-level information.

- `numberOfFields`
- `fieldTokens`: tokens in each field
- `avgFieldLengths`
- `numberOfDocuments`
- `numberOfTokens`
- `numberOfPointers`: total number of pointers in the inverted file $= sum(df)$
- `numberOfUniqueTerms`: $len(lexicon)$
- `averageDocumentLength`

`EntryStatistics` contain some information about the term.

- *frequency*: the total number of occurences (`tf(Corpus)` above)
- *df*
- *term id*

The models are rather simple: they override the scoring methods with their own formulas. There are also methods for setting the parameters (e.g. *b* for BM25).

The last class of interest is `Idf`. All kinds of idf functions are defined in it, such as `idfDFR()`, `idfENQUIRY` (seems to be buggy in 3.0), etc. I have two gripes with this:

- For one, it gives the idea that if a new formula is added, its idf part should be implemented in this class. However, someone who just wants to use the library may not feel like modifying a core class.
- Also, even this idea is false. BM25, for instance, does not have its own `idfBM25()` method in `Idf`; it is computed in the `BM25` class directly.

`Idf` also has methods that compute the logarithm of number(s). This is because they use $\log_2$ instead of $\log_n$.

There are several sub-hierarchies under `WeightingModel`. These will be introduced in the next sections.

The DFR Framework

`DFRWeightingModel` is the base class for the modular DFR framework. The scoring formula has been divided into three parts, following the original paper:

- `BasicModel` subclasses represent the basic randomness models.
- `AfterEffect` subclasses compute the gain.
- `Normalisation` is applied on the "raw" term frequencies before they are passed to the basic model.

These classes define their own custom interfaces, which have little in common with the interface of the `WeightingModel` class. It is an interesting question if it is possible to maintain a consistent interface (e.g. *Similarity*) across scoring components and ranking frameworks.

It is worth mentioning that the basic model, aftereffect and normalization implementations are selected by class name and are initialized via the class loader mechanism. This might or might not be necessary in Lucene.

Per-field Normalisation

Field-aware scoring algorithms are implemented as subclasses of `PerFieldNormWeightingModel`. Currently two such models are implemented: BM25F and PL2F.

As far as I understand, per-field scoring is the default in Lucene, so this particular sub-hierarchy might not be of little interest to us.

Statistics availability in Lucene

The content of the `EntryStatistics` class, i.e. the term statistics, is conveniently mirrored by the `TermContext` class. The relevant fields are

- `docFreq` corresponds to *df*,
- `totalTermFreq` corresponds to *frequency*.

If I understand correctly, these statistics are extracted from the index on the fly, per field, so they depend on the fields searched in the query.

Also, as far as *df* goes, there is also `IndexReader.docFreq()`.

Collection-level statistics seem to be harder to come by.

- *number of fields*: `IndexReader.fields()`, **BUT** this statistic is only for normalization, which is performed outside of the `Similarity` in Lucene; hence, we don't need it;
- *no. of tokens in a field*: `IndexReader.getSumOfNorms()`; it's a bit different than the real length; it may be worth to have both, since the more options, the more possibilities to experiment with;
- *avg. field length*: has to be computed as in `MockBM25Similarity.avgDocumentLength()` from the no. of tokens in each field;
- *no. of documents*: `IndexReader.numDocs()` (for some reason, `maxDoc()` is used in `MockBM25Similarity`) from the context;
- *tf(Corpus)*: `TermsEnum.totalTermFreq()` via `IndexReader.totalTermFreq()`;
- *no. of tokens*: `Terms.getSumTotalTermFreq()`;
- *no. of unique terms*: `Terms.getUniqueTermCount()` via `IndexReader.getUniqueTermCount()`;
- *average document length*: has to be computed as in `MockBM25Similarity.avgDocumentLength()` from the avg. field lengths.

Conclusion

I have found the scoring hierarchy of Terrier very straightforward and easy to extend. However, I am not really convinced of the merits of the way idf is handled. I would rather have separate *Tf*, *Idf* and *Query weight* (and maybe *Smoothing*) parts that could be combined freely with each other. The obvious solution is to have separate class hierarchies (however flat) for all of them. This is up for debate, of course.

Return to main page