Apache Lucene - Scoring

Grant Ingersoll

Table of contents

1 Introduction

Lucene scoring is the heart of why we all love Lucene. It is blazingly fast and it hides almost all of the complexity from the user. In a nutshell, it works. At least, that is, until it doesn't work, or doesn't work as one would expect it to work. Then we are left digging into Lucene internals or asking for help on java-user@lucene.apache.org to figure out why a document with five of our query terms scores lower than a different document with only one of the query terms.

While this document won't answer your specific scoring issues, it will, hopefully, point you to the places that can help you figure out the what and why of Lucene scoring.

Lucene scoring uses a combination of the Vector Space Model (VSM) of Information Retrieval and the Boolean model to determine how relevant a given Document is to a User's query. In general, the idea behind the VSM is the more times a query term appears in a document relative to the number of times the term appears in all the documents in the collection, the more relevant that document is to the query. It uses the Boolean model to first narrow down the documents that need to be scored based on the use of boolean logic in the Query specification. Lucene also adds some capabilities and refinements onto this model to support boolean and fuzzy searching, but it essentially remains a VSM based system at the heart. For some valuable references on VSM and IR in general refer to the Lucene Wiki IR references.

The rest of this document will cover Scoring basics and how to change your Similarity. Next it will cover ways you can customize the Lucene internals in Changing your Scoring -- Expert Level which gives details on implementing your own Query class and related functionality. Finally, we will finish up with some reference material in the Appendix.

2 Scoring

Scoring is very much dependent on the way documents are indexed, so it is important to understand indexing (see Apache Lucene - Getting Started Guide and the Lucene file formats before continuing on with this section.) It is also assumed that readers know how to use the Searcher.explain(Query query, int doc) functionality, which can go a long way in informing why a score is returned.

2.1 Fields and Documents

In Lucene, the objects we are scoring are Documents. A Document is a collection of Fields. Each Field has semantics about how it is created and stored (i.e. tokenized, untokenized, raw data, compressed, etc.) It is important to note that Lucene scoring works on Fields and then combines the results to return Documents. This is important because two Documents with the exact same content, but one having the content in two Fields and the other in one Field

will return different scores for the same query due to length normalization (assumming the DefaultSimilarity on the Fields).

2.2 Score Boosting

Lucene allows influencing search results by "boosting" in more than one level:

- Document level boosting - while indexing - by calling document.setBoost() before a document is added to the index.
- Document's Field level boosting - while indexing - by calling field.setBoost() before adding a field to the document (and before adding the document to the index).
- Query level boosting - during search, by setting a boost on a query clause, calling Query.setBoost().

Indexing time boosts are preprocessed for storage efficiency and written to the directory (when writing the document) in a single byte (!) as follows: For each field of a document, all boosts of that field (i.e. all boosts under the same field name in that doc) are multiplied. The result is multiplied by the boost of the document, and also multiplied by a "field length norm" value that represents the length of that field in that doc (so shorter fields are automatically boosted up). The result is decoded as a single byte (with some precision loss of course) and stored in the directory. The similarity object in effect at indexing computes the length-norm of the field.

This composition of 1-byte representation of norms (that is, indexing time multiplication of field boosts & doc boost & field-length-norm) is nicely described in Fieldable.setBoost().

Encoding and decoding of the resulted float norm in a single byte are done by the static methods of the class Similarity: encodeNorm() and decodeNorm(). Due to loss of precision, it is not guaranteed that decode(encode(x)) = x, e.g. decode(encode(0.89)) = 0.75. At scoring (search) time, this norm is brought into the score of document as norm(t, d), as shown by the formula in Similarity.

2.3 Understanding the Scoring Formula

This scoring formula is described in the Similarity class. Please take the time to study this formula, as it contains much of the information about how the basics of Lucene scoring work, especially the TermQuery.

2.4 The Big Picture

OK, so the tf-idf formula and the Similarity is great for understanding the basics of Lucene scoring, but what really drives Lucene scoring are the use and interactions between the Query classes, as created by each application in response to a user's information need.

In this regard, Lucene offers a wide variety of Query implementations, most of which are in the org.apache.lucene.search package. These implementations can be combined in a wide variety of ways to provide complex querying capabilities along with information about where matches took place in the document collection. The Query section below highlights some of the more important Query classes. For information on the other ones, see the package summary. For details on implementing your own Query class, see Changing your Scoring -- Expert Level below.

Once a Query has been created and submitted to the IndexSearcher, the scoring process begins. (See the Appendix Algorithm section for more notes on the process.) After some infrastructure setup, control finally passes to the Weight implementation and its Scorer instance. In the case of any type of BooleanQuery, scoring is handled by the BooleanWeight2 (link goes to ViewVC BooleanQuery java code which contains the BooleanWeight2 inner class) or BooleanWeight (link goes to ViewVC BooleanQuery java code, which contains the BooleanWeight inner class).

Assuming the use of the BooleanWeight2, a BooleanScorer2 is created by bringing together all of the Scorers from the sub-clauses of the BooleanQuery. When the BooleanScorer2 is asked to score it delegates its work to an internal Scorer based on the type of clauses in the Query. This internal Scorer essentially loops over the sub scorers and sums the scores provided by each scorer while factoring in the coord() score.

2.5 Query Classes

For information on the Query Classes, refer to the search package javadocs

2.6 Changing Similarity

One of the ways of changing the scoring characteristics of Lucene is to change the similarity factors. For information on how to do this, see the search package javadocs

3 Changing your Scoring -- Expert Level

At a much deeper level, one can affect scoring by implementing their own Query classes (and related scoring classes.) To learn more about how to do this, refer to the search package javadocs

4 Appendix

4.1 Algorithm

This section is mostly notes on stepping through the Scoring process and serves as fertilizer for the earlier sections.

In the typical search application, a Query is passed to the Searcher , beginning the scoring process.

Once inside the Searcher, a Collector is used for the scoring and sorting of the search results. These important objects are involved in a search:

1. The Weight object of the Query. The Weight object is an internal representation of the Query that allows the Query to be reused by the Searcher.
2. The Searcher that initiated the call.
3. A Filter for limiting the result set. Note, the Filter may be null.
4. A Sort object for specifying how to sort the results if the standard score based sort method is not desired.

Assuming we are not sorting (since sorting doesn't effect the raw Lucene score), we call one of the search methods of the Searcher, passing in the Weight object created by Searcher.createWeight(Query), Filter and the number of results we want. This method returns a TopDocs object, which is an internal collection of search results. The Searcher creates a TopScoreDocCollector and passes it along with the Weight, Filter to another expert search method (for more on the Collector mechanism, see Searcher .) The TopDocCollector uses a PriorityQueue to collect the top results for the search.

If a Filter is being used, some initial setup is done to determine which docs to include. Otherwise, we ask the Weight for a Scorer for the IndexReader of the current searcher and we proceed by calling the score method on the Scorer .

At last, we are actually going to score some documents. The score method takes in the Collector (most likely the TopScoreDocCollector or TopFieldCollector) and does its business. Of course, here is where things get involved. The Scorer that is returned by the Weight object depends on what type of Query was submitted. In most real world applications with multiple query terms, the Scorer is going to be a BooleanScorer2 (see the section on customizing your scoring for info on changing this.)

Assuming a BooleanScorer2 scorer, we first initialize the Coordinator, which is used to apply the coord() factor. We then get a internal Scorer based on the required, optional and prohibited parts of the query. Using this internal Scorer, the BooleanScorer2 then proceeds into a while loop based on the Scorer#next() method. The next() method advances to the next document matching the query. This is an abstract method in the Scorer class and is thus overriden by all derived implementations. If you have a simple OR query your internal

Scorer is most likely a DisjunctionSumScorer, which essentially combines the scorers from the sub scorers of the OR'd terms.