[Lucene评分算法解释](http://www.hankcs.com/program/java/lucene-scoring-algorithm-explained.html)

[码农场](http://www.hankcs.com/) > [编程开发](http://www.hankcs.com/program/) > [Java](http://www.hankcs.com/program/java/) 2014-02-20 阅读(3417) [评论(2)](http://www.hankcs.com/program/java/lucene-scoring-algorithm-explained.html#comments)

目录

* **[总匹配分值的计算](http://www.hankcs.com/program/java/lucene-scoring-algorithm-explained.html" \l "h2-0)**
* [查询语句在某个域匹配分值计算](http://www.hankcs.com/program/java/lucene-scoring-algorithm-explained.html" \l "h3-1)

Lucene的IndexSearcher提供一个explain方法，能够解释Document的Score是怎么得来的，具体每一部分的得分都可以详细地打印出来。这里用一个中文实例来纯手工验算一遍Lucene的评分算法，并且结合Lucene的源码做一个解释。

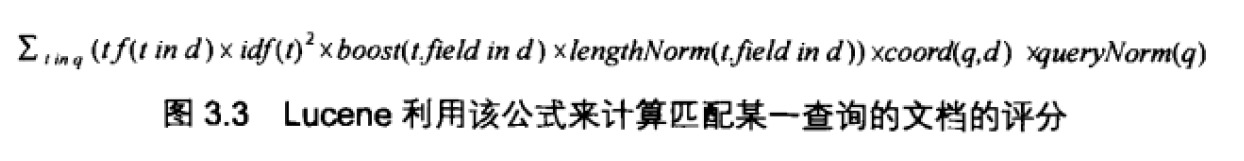
首先是测试用例，我使用“食品安全”来检索一个含有title与content域的文档。

然后是是输出，注意它有缩进，代表一个个的层级：

1. 5.6394258 = (MATCH) sum of:
2. 5.3901243 = (MATCH) sum of:
3. 3.2243047 = (MATCH) weight(title:食品 in 361) [DefaultSimilarity], result of:
4. 3.2243047 = score(doc=361,freq=1.0 = termFreq=1.0
5. ), product of:
6. 0.66116947 = queryWeight, product of:
7. 5.5733356 = idf(docFreq=14, maxDocs=1453)
8. 0.11863084 = queryNorm
9. 4.876669 = fieldWeight in 361, product of:
10. 1.0 = tf(freq=1.0), with freq of:
11. 1.0 = termFreq=1.0
12. 5.5733356 = idf(docFreq=14, maxDocs=1453)
13. 0.875 = fieldNorm(doc=361)
14. 2.16582 = (MATCH) weight(title:安全 in 361) [DefaultSimilarity], result of:
15. 2.16582 = score(doc=361,freq=1.0 = termFreq=1.0
16. ), product of:
17. 0.5418835 = queryWeight, product of:
18. 4.5678134 = idf(docFreq=40, maxDocs=1453)
19. 0.11863084 = queryNorm
20. 3.9968367 = fieldWeight in 361, product of:
21. 1.0 = tf(freq=1.0), with freq of:
22. 1.0 = termFreq=1.0
23. 4.5678134 = idf(docFreq=40, maxDocs=1453)
24. 0.875 = fieldNorm(doc=361)
25. 0.24930152 = (MATCH) sum of:
26. 0.17587993 = (MATCH) weight(content:食品 in 361) [DefaultSimilarity], result of:
27. 0.17587993 = score(doc=361,freq=13.0 = termFreq=13.0
28. ), product of:
29. 0.43032452 = queryWeight, product of:
30. 3.6274254 = idf(docFreq=104, maxDocs=1453)
31. 0.11863084 = queryNorm
32. 0.40871462 = fieldWeight in 361, product of:
33. 3.6055512 = tf(freq=13.0), with freq of:
34. 13.0 = termFreq=13.0
35. 3.6274254 = idf(docFreq=104, maxDocs=1453)
36. 0.03125 = fieldNorm(doc=361)
37. 0.073421605 = (MATCH) weight(content:安全 in 361) [DefaultSimilarity], result of:
38. 0.073421605 = score(doc=361,freq=11.0 = termFreq=11.0
39. ), product of:
40. 0.28989288 = queryWeight, product of:
41. 2.4436553 = idf(docFreq=342, maxDocs=1453)
42. 0.11863084 = queryNorm
43. 0.2532715 = fieldWeight in 361, product of:
44. 3.3166249 = tf(freq=11.0), with freq of:
45. 11.0 = termFreq=11.0
46. 2.4436553 = idf(docFreq=342, maxDocs=1453)
47. 0.03125 = fieldNorm(doc=361)

这个看起来可真是头疼，尝试解释一下：

首先，需要学习Lucene的评分计算公式——



分值计算方式为查询语句q中每个项t与文档d的匹配分值之和，当然还有权重的因素。其中每一项的意思如下表所示：

|  |  |
| --- | --- |
| 表3.5 | 评分公式中的因子 |
| 评分因子 | 描 述 |
| tf(t in d) | 项频率因子——文档**（**d)中出现项**（**t)的频率 |
| idf(t) | 项在倒排文档中出现的频率：它被用来衡量项的“唯一”性.出现频率较高的term具有较低的idf,出现较少的term具有较高的idf |
| boost(t.field in d) | 域和文档的加权，在索引期间设置.你可以用该方法 对某个域或文档进行静态单独加权 |
| lengthNorm(t.field in d) | 域的归一化（Normalization)值，表示域中包含的项数量.该值在索引期间计算，并保存在索引norm中.对于该因子，更短的域（或更少的语汇单元）能获得更大的加权 |
| coord(q,d) | 协调因子（Coordination factor),基于文档中包含查询的项个数.该因子会对包含更多搜索项的文档进行类似AND的加权 |
| queryNorm(q) | 每个査询的归一化值，指毎个查询项权重的平方和 |

**总匹配分值的计算**

具体到上面的测试来讲，每个文档有两个域：title和content，最终匹配分值=查询语句在两个域中的得分之和。即最终结果5.6394258 = 5.3901243 + 0.24930152。

**查询语句在某个域匹配分值计算**

这个5.3901243是如何来的呢？查询语句有两个项t:"食品"和"安全"。所以计算结果等于这两部分的和：“食品”在title中的匹配分值 + “安全”在title中的匹配分值。即 5.3901243 = 3.2243047 + 2.16582 。

某个项在某个域的匹配分值的计算

接下来我们看看“食品”在title中的匹配分值 3.2243047 是怎么算出来的。t在field中的分值score = 查询权重queryWeight \* 域权重fieldWeight，即 3.2243047 = 0.66116947 \* 4.876669 。

queryWeight的计算

queryWeight的计算可以在TermQuery$TermWeight.normalize(float)方法中看到计算的实现：

1. public void normalize(float queryNorm) {
2. this.queryNorm = queryNorm;
3. //原来queryWeight 为idf\*t.getBoost()，现在为queryNorm\*idf\*t.getBoost()。
4. queryWeight \*= queryNorm;
5. value = queryWeight \* idf;
6. }

其实默认情况下，queryWeight = idf \* queryNorm，因为Lucene中默认的boost = 1.0。

查询权重queryWeight 0.66116947 的计算方法：查询权重queryWeight = idf \* queryNorm，即 0.66116947 = 5.5733356 \* 0.11863084。

idf的计算

idf是项在倒排文档中出现的频率，计算方式为

1. /\*\* Implemented as <code>log(numDocs/(docFreq+1)) + 1</code>. \*/
2. @Override
3. public float idf(long docFreq, long numDocs) {
4. return (float)(Math.log(numDocs/(double)(docFreq+1)) + 1.0);
5. }

docFreq是根据指定关键字进行检索，检索到的Document的数量，我们测试的docFreq=14；numDocs是指索引文件中总共的Document的数量，我们测试的numDocs=1453。用计算器验证一下，没有错误，这里就不啰嗦了。

queryNorm的计算

queryNorm的计算在DefaultSimilarity类中实现，如下所示：

1. /\*\* Implemented as <code>1/sqrt(sumOfSquaredWeights)</code>. \*/
2. public float queryNorm(float sumOfSquaredWeights) {
3. return (float)(1.0 / Math.sqrt(sumOfSquaredWeights));
4. }

这里，sumOfSquaredWeights的计算是在org.apache.lucene.search.TermQuery.TermWeight类中的sumOfSquaredWeights方法实现：

1. public float sumOfSquaredWeights() {
2. queryWeight = idf \* getBoost();             // compute query weight
3. return queryWeight \* queryWeight;          // square it
4. }

其实默认情况下，sumOfSquaredWeights = idf \* idf，因为Lucune中默认的boost = 1.0。

上面测试例子中sumOfSquaredWeights的计算如下所示：

sumOfSquaredWeights = 5.5733356 \* 5.5733356 + 4.5678134 \* 4.5678134 + 3.6274254 \* 3.6274254 + 2.4436553 \* 2.4436553 = 71.05665522523017；

上面的四个weight分别来自 {食品, 安全} \* {title, content} 这四个组合。

然后，就可以计算queryNorm的值了，计算如下所示：

queryNorm = (float)(1.0 / Math.sqrt(71.05665522523017) = 0.11863084386918748683822481722352；

fieldWeight的计算

在org/apache/lucene/search/similarities/TFIDFSimilarity.java的explainScore方法中有：

1. // explain field weight
2. Explanation fieldExpl = new Explanation();
3. fieldExpl.setDescription("fieldWeight in "+doc+
4. ", product of:");
6. Explanation tfExplanation = new Explanation();
7. tfExplanation.setValue(tf(freq.getValue()));
8. tfExplanation.setDescription("tf(freq="+freq.getValue()+"), with freq of:");
9. tfExplanation.addDetail(freq);
10. fieldExpl.addDetail(tfExplanation);
11. fieldExpl.addDetail(stats.idf);
13. Explanation fieldNormExpl = new Explanation();
14. float fieldNorm = norms != null ? decodeNormValue(norms.get(doc)) : 1.0f;
15. fieldNormExpl.setValue(fieldNorm);
16. fieldNormExpl.setDescription("fieldNorm(doc="+doc+")");
17. fieldExpl.addDetail(fieldNormExpl);
19. fieldExpl.setValue(tfExplanation.getValue() \*
20. stats.idf.getValue() \*
21. fieldNormExpl.getValue());
23. result.addDetail(fieldExpl);

重点是这一句：

1. fieldExpl.setValue(tfExplanation.getValue() \*
2. stats.idf.getValue() \*
3. fieldNormExpl.getValue());

使用计算式表示就是

fieldWeight = tf \* idf \* fieldNorm

tf和idf的计算参考前面的，fieldNorm的计算在索引的时候确定了，此时直接从索引文件中读取，这个方法并没有给出直接的计算。如果使用DefaultSimilarity的话，它实际上就是lengthNorm，域越长的话Norm越小，在org/apache/lucene/search/similarities/DefaultSimilarity.java里面有关于它的计算：

1. public float lengthNorm(FieldInvertState state) {
2. final int numTerms;
3. if (discountOverlaps)
4. numTerms = state.getLength() - state.getNumOverlap();
5. else
6. numTerms = state.getLength();
7. return state.getBoost() \* ((float) (1.0 / Math.sqrt(numTerms)));
8. }

这个我就不再验算了，每个域的Terms数量开方求倒数乘以该域的boost得出最终的结果。