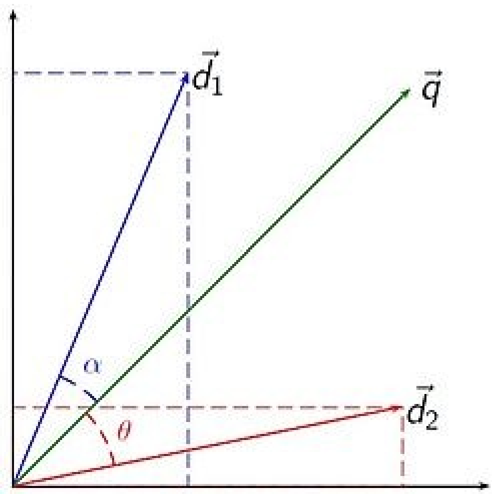
VSM Model

Vector space model or term **vector model** is an algebraic **model** for representing text documents (and any objects, in general) as **vectors** of identifiers, such as, for example, index terms. It is used in information filtering, information retrieval, indexing and relevancy rankings.

In VSM, documents and queries are represented as weighted vectors in a multi-dimensional space, where each distinct index term is a dimension, and weights are Tf-idf values.

VSM does not require weights to be *Tf-idf* values, but *Tf-idf* values are believed to produce search results of high quality, and so Lucene is using *Tf-idf.*



Applications

Relevance rankings of documents in a keyword search can be calculated, using the assumptions of document similarities theory, by comparing the deviation of angles between each document vector and the original query vector where the query is represented as the same kind of vector as the documents.

In practice, it is easier to calculate the cosine of the angle between the vectors, instead of the angle itself:

$$\cos\theta = \frac{\mathbf{d_2} \cdot \mathbf{q}}{\|\mathbf{d_2}\| \, \|\mathbf{q}\|}$$

Scoring

Tf and *Idf* are described in more detail below, but for now, for completion, let's just say that for given term *t* and document (or query) *x*, *Tf(t,x)* varies with the number of occurrences of term *t* in *x* (when one increases so does the other) and *idf(t)* similarly varies with the inverse of the number of index documents containing term *t*.

VSM score of document *d* for query *q* is the Cosine Similarity of the weighted query vectors *V(q)* and *V(d)*:

$$\text{cosine-similarity}(q,d) = \frac{V(q) \cdot V(d)}{|V(q)|\ |V(d)|}$$

VSM Score

$$\text{score}(q,d) = \text{coord-factor}(q,d) \cdot \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot \text{doc-len-norm}(d) \cdot \text{doc-boost}(d)$$

Lucene Conceptual Scoring Formula

$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} \left(\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot \text{t.getBoost}() \cdot \text{norm}(t,d) \right)$$

Lucene Practical Scoring Function

Advantages

The vector space model has the following advantages over the Standard Boolean model:

1. Simple model based on linear algebra
2. Term weights not binary
3. Allows computing a continuous degree of similarity between queries and documents
4. Allows ranking documents according to their possible relevance
5. Allows partial matching

Limitations

The vector space model has the following limitations:

1. Long documents are poorly represented because they have poor similarity values (a small scalar product and a large dimensionality)
2. Search keywords must precisely match document terms; word substrings might result in a "false positive match"
3. Semantic sensitivity; documents with similar context but different term vocabulary won't be associated, resulting in a "false negative match".
4. The order in which the terms appear in the document is lost in the vector space representation.
5. Theoretically assumes terms are statistically independent.
6. Weighting is intuitive but not very formal.

Results

We are comparing results of our VSM Model with efficient performance tuning and without efficient performance

Efficient System

```
num_q          all   14
num_ret        all   10516
num_rel        all   210
num_rel_ret    all   199

map            all   0.6543
gm_map         all   0.5804
Rprec          all   0.6222
bpref          all   0.7590
ndcg           all   0.8909
recip_rank     all   1.0000

P_5            all   0.8000
P_10           all   0.5857
```

Without custom Search handler

num_q	all	14
num_ret	all	6455
num_rel	all	210
num_rel_ret	all	185
map	all	0.6739
gm_map	all	0.6183
Rprec	all	0.6678
bpref	all	0.7580
ndcg	all	0.8744
recip_rank	all	1.0000
P_5	all	0.8286
P_10	all	0.6071

Without custom fields

num_q	all	14
num_ret	all	4782
num_rel	all	210
num_rel_ret	all	112
map	all	0.5764
gm_map	all	0.4855
Rprec	all	0.5617
bpref	all	0.6431
ndcg	all	0.7700
recip_rank	all	1.0000
P_5	all	0.7857
P_10	all	0.5286

Performance Tuning

We have done performance tuning as below

Custom Fields

```xml
<!--   These are custom fields, defined to maximize query results -->
<field name="text_custom_ru" type="text_custom_ru" indexed="true" stored="false" multiValued="true"/>

<!-- Custom Russian -->
<fieldType name="text_custom_ru" class="solr.TextField" positionIncrementGap="100">
    <analyzer type="index">
        <charFilter class="solr.MappingCharFilterFactory" mapping="mapping-ISOLatin1Accent.txt" />
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.TrimFilterFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.WordDelimiterFilterFactory" splitOnCaseChange="1" splitOnNumerics="0"
                generateWordParts="1" stemEnglishPossessive="0" generateNumberParts="1"
                catenateWords="1" catenateNumbers="0" catenateAll="0" preserveOriginal="1"/>
        <filter class="solr.EdgeNGramFilterFactory" minGramSize="2" maxGramSize="15"/>
       <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
    </analyzer>

    <analyzer type="query">
        <charFilter class="solr.MappingCharFilterFactory" mapping="mapping-ISOLatin1Accent.txt" />
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.TrimFilterFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.WordDelimiterFilterFactory" splitOnCaseChange="1" splitOnNumerics="0"
                generateWordParts="1" stemEnglishPossessive="0" generateNumberParts="1"
                catenateWords="1" catenateNumbers="0" catenateAll="0" preserveOriginal="1"/>
    </analyzer>
</fieldType>


<!-- copyField commands copy one field to another at the time a document
    is added to the index.  It's used either to index the same field differently,
    or to add multiple fields to the same field for easier/faster searching.  -->

<copyField source="text_ru" dest="text_custom_ru"/>
```

Custom Search Handler

```xml
<!-- This is custom search handler for Russian queries which gives more weight to Russian documents because query is also in same
     language -->
<requestHandler name="/ramanVsmSearchRu" class="solr.SearchHandler" default="true">
    <lst name="defaults">
        <str name="echoParams">explicit</str>
        <str name="defType">edismax</str>
        <str name="mm">30%</str>
        <str name="bf">recip(ms(NOW,created_at),3.16e-11,1,1)</str>
        <str name="qf">
            text_ru^6.0 text_custom_ru^4.0 text_en text_custom_en text_de text_custom_de tweet_hashtags
        </str>
        <str name="pf">
            text_ru^6.0 text_custom_ru^4.0 text_en text_custom_en text_de text_custom_de tweet_hashtags
        </str>
        <str name="pf2">
            text_ru^6.0 text_custom_ru^4.0 text_en text_custom_en text_de text_custom_de tweet_hashtags
        </str>
        <str name="pf3">
            text_ru^6.0 text_custom_ru^4.0 text_en text_custom_en text_de text_custom_de tweet_hashtags
        </str>
        <float name="tie">0.1</float>
        <int name="ps">10</int>
        <int name="qs">10</int>
        <str name="q.alt">*:*</str>
    </lst>
</requestHandler>
```

Multilingual Query Processing

```
if (lang.equals("ru")) {
        translated_en = Translate.execute(q1, Language.RUSSIAN, Language.ENGLISH);
        translated_de = Translate.execute(q1, Language.RUSSIAN, Language.GERMAN);

        q2 = "text_custom_ru:\"\"" + q1 + "\"\""
           + "ORtext_custom_en:\"\"" + translated_en + "\"\""
           + "ORtext_custom_de:\"\"" + translated_de + "\"\"";

        searchEngine = "ramanVsmSearchRu";

}
```

BM25 Model:

This model is a probabilistic model which has two parameters to tune, k1 and b. It ranks a set of documents based on the query terms appearing in each document, regardless of the inter-relationship between the query terms within a document (e.g., their relative proximity).

B: This parameter controls how much effect field-length normalization should have. A value of `0.0` disables normalization completely, and a value of `1.0` normalizes fully. The default is `0.75`.

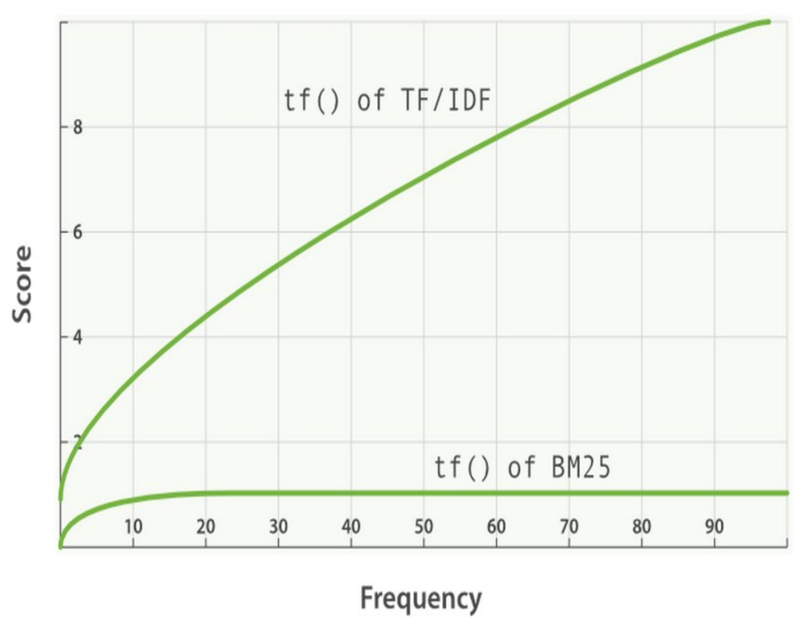
Tuning depends on the dataset we have. The dataset that was indexed was a set of tweets with a limit on the maximum length (140 chars). So, we don't have to set a very big value of length normalization. Due to the upper limit on the document length we set a small value of b.

K1: This parameter controls how quickly an increase in term frequency results in term-frequency saturation. The default value is `1.2`. Lower values result in quicker saturation, and higher values in slower saturation.

Keeping a higher value is feasible with respect to our dataset as we have less terms in the document and we don't want the saturation to happen very quickly.

Advantage of BM25 Model over default model:

Terms that appear 5 to 10 times in a document have a significantly larger impact on relevance than terms that appear just once or twice. However, terms that appear 20 times in a document have almost the same impact as terms that appear a thousand times or more.



Tuning the model:

We have to define the Similarity Model in the schema and restart the solr service and reindex the documents, the following is to be included in the schema, where we tune X.X.

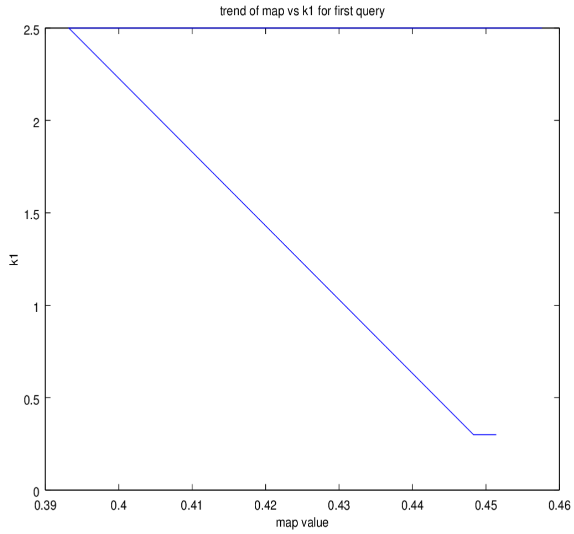
<similarity class=”solr.BM25SimilarityFactory”>
<float name=”k1”>X.X</float>
<float name=”b”>X.X</float>
</similarity>

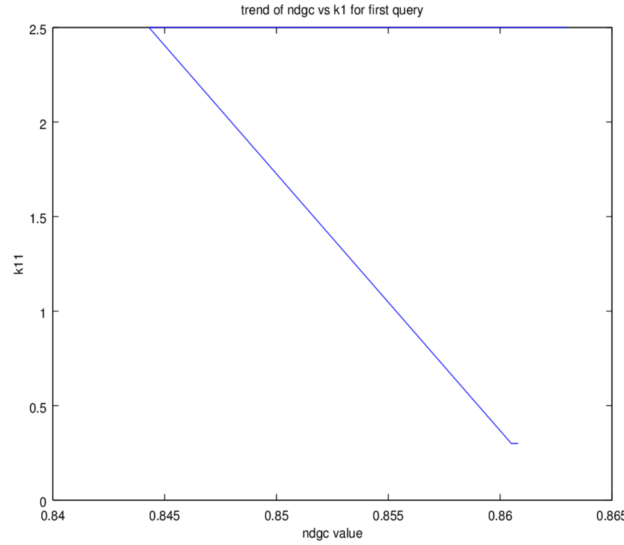
We wrote a simple program in python to try a range of values of k1[0.1, 3.0] and b[0.1, 0.9]. For each of the combinations, the schema was updated using a file pointer. Using commands module, the solr services were restarted and index refreshed. After that, the queries were fired and output written to a file. This file is now input for trec evaluation, trec evaluation is also done in the program and result is appended to a file. Now we check for the file and find trends in the trec evaluation output to get the best result.

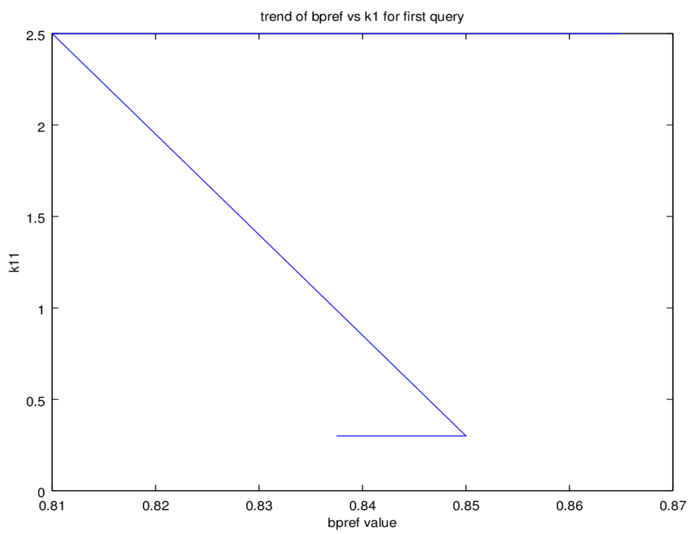
Figure 2 – Program to automate the tuning process

```python
import re
import os
import json
from urllib import urlencode
import urllib2
import commands
def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r += step
def main():
    k1_list=drange(0.1, 3.0, 0.1)
    b_list=drange(0.1, 0.9, 0.1)
    schema_pointer=open('schema.xml', 'r')
    schema_lines = schema_pointer.readlines()
    schema_pointer.close()
    queries_pointer=open('queries2.txt', 'r')
    k1_l=[]
    b_l=[]
    for k in ["%g" % x for x in k1_list]:
        k1_l.append(k)
    for b1 in ["%g" % x for x in b_list]:
        b_l.append(b1)
    for each_query in queries_pointer:
        for k1 in k1_l:
            for b in b_l:
                print 'Searching for b = ',b,'and k1 = ',k1
                schema_lines[126] = '\t\t\t<float name="k1">'+ str(k1) +'</float>\n'
                schema_lines[127] = '\t\t\t<float name="b">'+ str(b) +'</float>\n'
                schema_pointer=open('schema.xml', 'w')
                schema_pointer.writelines(schema_lines)
                schema_pointer.close()
                (status1, output1)=commands.getstatusoutput("~/solr/solr-5.3.0/bin/solr stop -all")
                print output1
                (status2, output2)=commands.getstatusoutput("~/solr/solr-5.3.0/bin/solr start -s ~/solr/solr-5.3.0/booksdemo/solr")
                print output2
                (status3, output3)=commands.getstatusoutput(' curl http://localhost:8983/solr/booksdemo/update?commit=true -H "Content-Type: text/xml" --data-binary \'<delete><query>*:*</query></delete>\'')
                print output3
                (status4, output4)=commands.getstatusoutput(" curl 'http://localhost:8983/solr/booksdemo/update/json?commit=true' --data-binary @$(echo ~/solr/solr-5.3.0/booksdemo/Train_Data.json) -H 'Content-type:application'")
                print output4
                outfn = 'file1.txt'
                qid = each_query[:3]
                IRModel='BM25'
                outf = open(outfn, 'w')

                a=unicode(each_query[4:], 'utf-8')
                params = {'where': 'nexearch', 'q': a.encode('utf-8')}
                params = urlencode(params)
                url = "http://localhost:8983/solr/booksdemo/BM25SearchEn?q=%22%22" + params[2:] + "%22%22&fl=id,text_en%2Cscore&wt=json&indent=true&&rows=1500&mm=2%3C1%208%3C2"
                data=urllib2.urlopen(url)

                docs = json.load(data)['response']['docs']
                rank = 1
                for doc in docs:
                    outf.write(qid + ' ' + 'Q0' + ' ' + str(doc['id']) + ' ' + str(rank) + ' ' + str(doc['score']) + ' ' + IRModel + '\n')
                    rank += 1
                outf.close()
                res2_pointer = open('res2.txt', 'w+')
                res2_pointer.write(str(k1)+", "+str(b)+"\n================================================================================\n\n\n")
                res2_pointer.close()
                os.system("cat res2.txt >> res.txt")
                os.system("trec_eval.9.0/./trec_eval -q -c -M1000 -m set_F.05 -m ndcg -m map -m bpref qrels.txt file1.txt >> res.txt")

if __name__ == '__main__':
    main()
```

Trends:





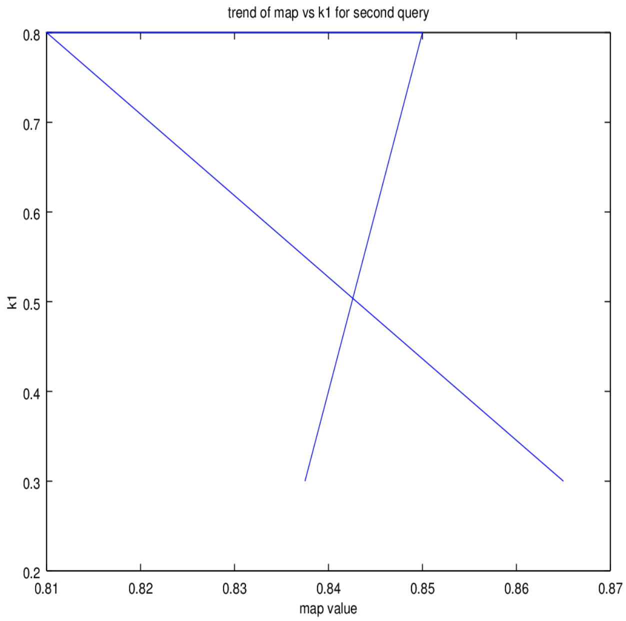


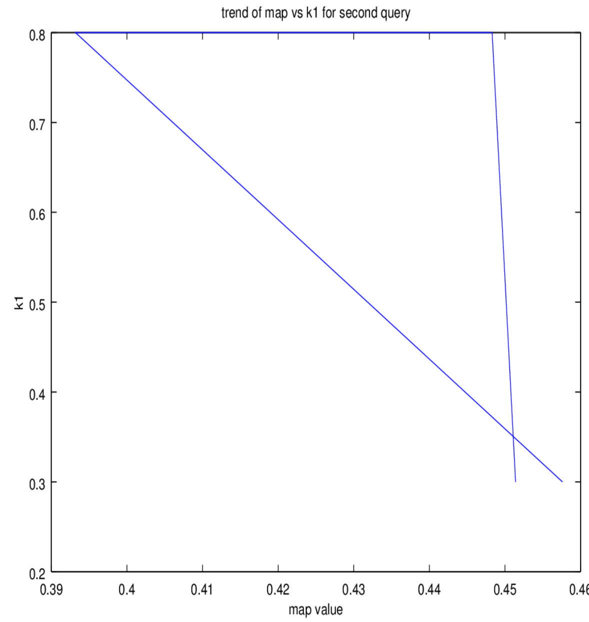
k1 = [0.30000 0.30000 2.50000 2.50000]
b = [0.30000 0.80000 0.80000 0.30000]
map = [0.45140 0.44830 0.39320 0.45760]
bpref=[0.83750 0.85000 0.81000 0.86500]
ndgc=[0.86080 0.86050 0.84430 0.86300]

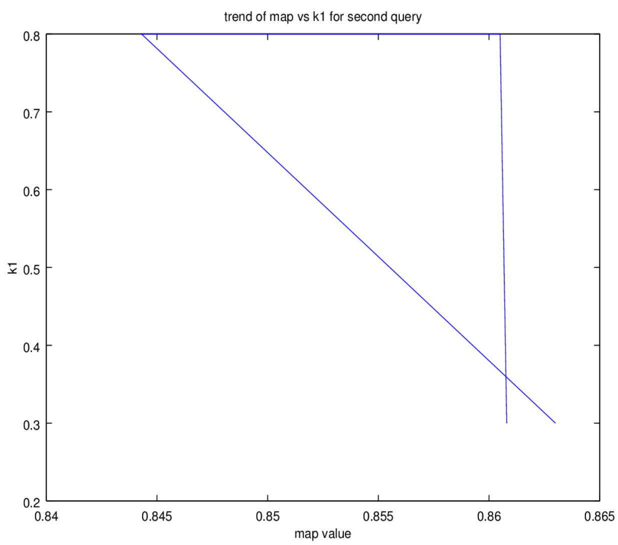
Thus, for first query, we can say that a higher value of k1 and a small value of b should be used.

max_map = 0.4719
max_bref = 0.8700
max_ndgc= 0.8663

we do a tradeoff with max value and see what is the closest to it.

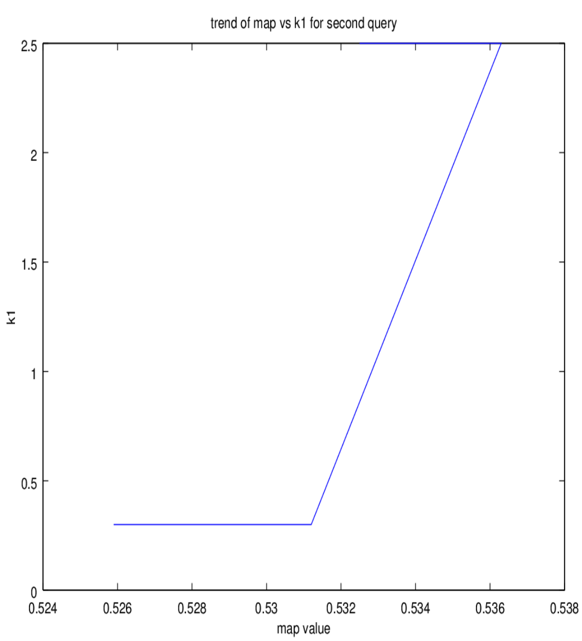


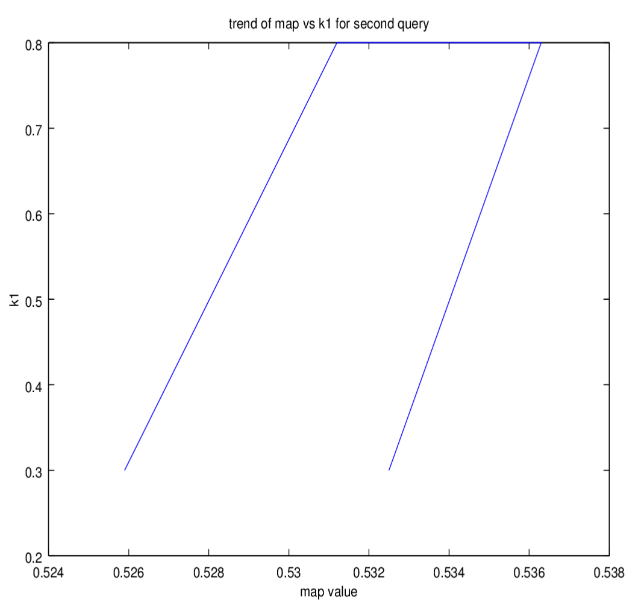


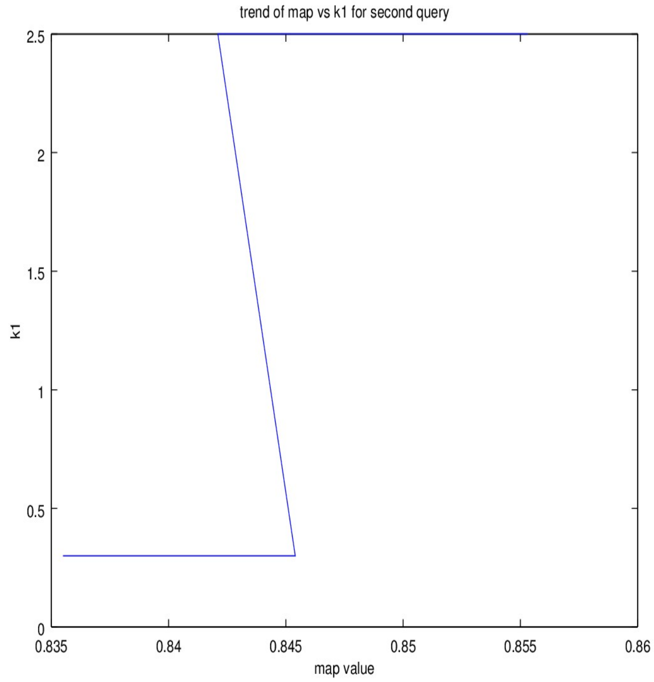


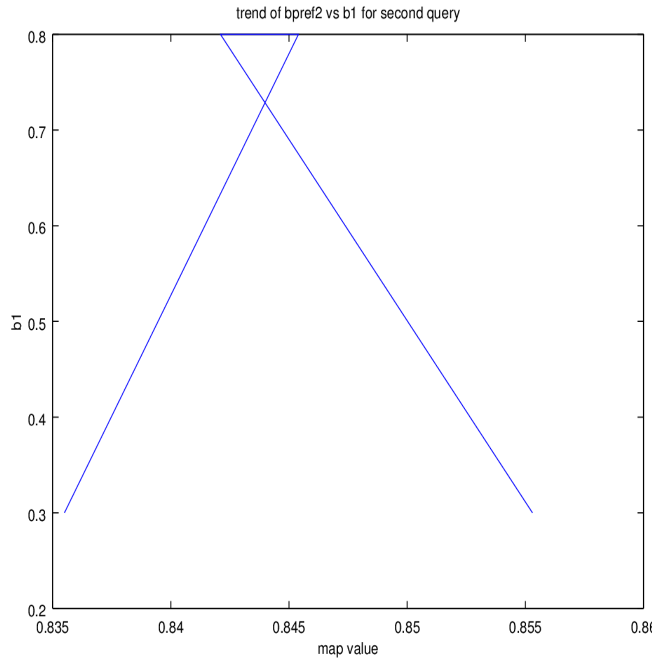
We see that map increases with increase in the value of b and other factors decrease with it. K1 normalizes this. So, we use a small value of b and a large value of k1.

Second test query:









we see the same trend in the second query too. Thus we choose a high value of k1 and low of b.

Final similarity after analyzing all the queries is:

<similarity class="solr.BM25SimilarityFactory">

<float name="k1">2.4</float>

<float name="b">0.2</float>

</similarity>

BM Model is better than the default model as the parameters values that we get in BM is better than the ones in default model.

LM Model:

The language modeling approach to IR directly models that idea: a document is a good match to a query if the document model is likely to generate the query, which will in turn happen if the document contains the query words often.

It has two subclassses: LMDirichletSimilarity, LMJelinekMercerSimilarity

We have used the first one in the project. It has one parameter to tune known as mu.

<similarity class="org.apache.solr.search.similarities.LMDirichletSimilarityFactory">

<float name="mu">XX</float>

</similarity>

The default value of mu is 2000, we tune this according to the results from trec.

Tuning:

started from 400 to 4000 with a step of 400; used the same python program as in BM25 Model after tweaking it for this model. The script is automated, thus the schema get updated, solr is restarted and refreshed, results obtained in a file and trec evaluation done. Then we analyize the result.

400:

map	001	0.5352
bpref	001	0.8700
ndcg	001	0.8911
set_F_05	001	0.1387

800:

map	001	0.5673
bpref	001	0.8725
ndcg	001	0.9021
set_F_05	001	0.1387

1200:

map	001	0.5968
bpref	001	0.8600
ndcg	001	0.9130
set_F_05	001	0.1387

1600:

map	001	0.5897
bpref	001	0.8450
ndcg	001	0.9112
set_F_05	001	0.1387

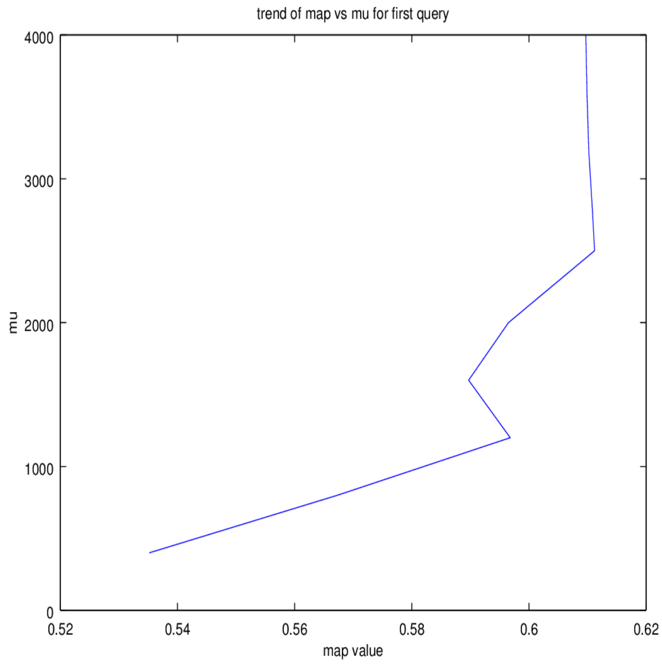
we see that for lower mu values, map is low but other parameters are high. When mu is high, then map is high but other parameters tend to be low. So, we choose an optimum value between them.

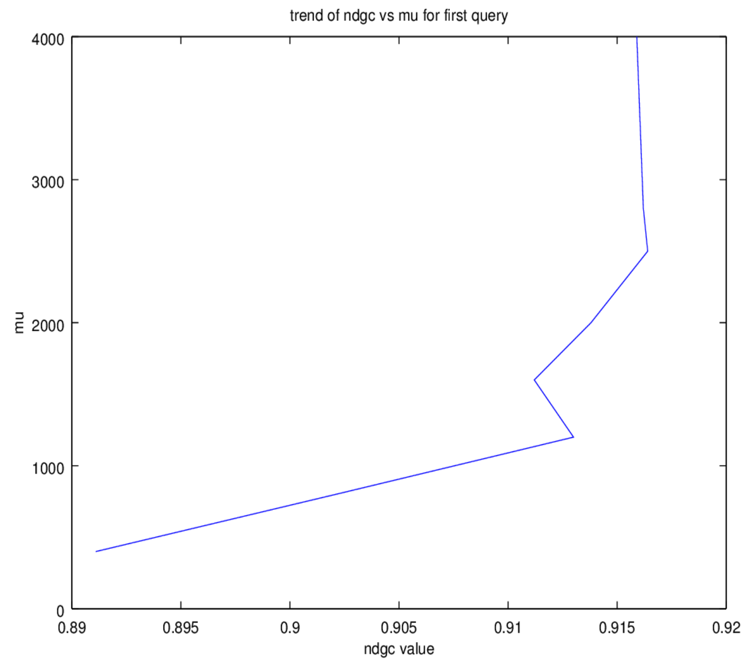
We chose mu as 1200 for our model. Thus,

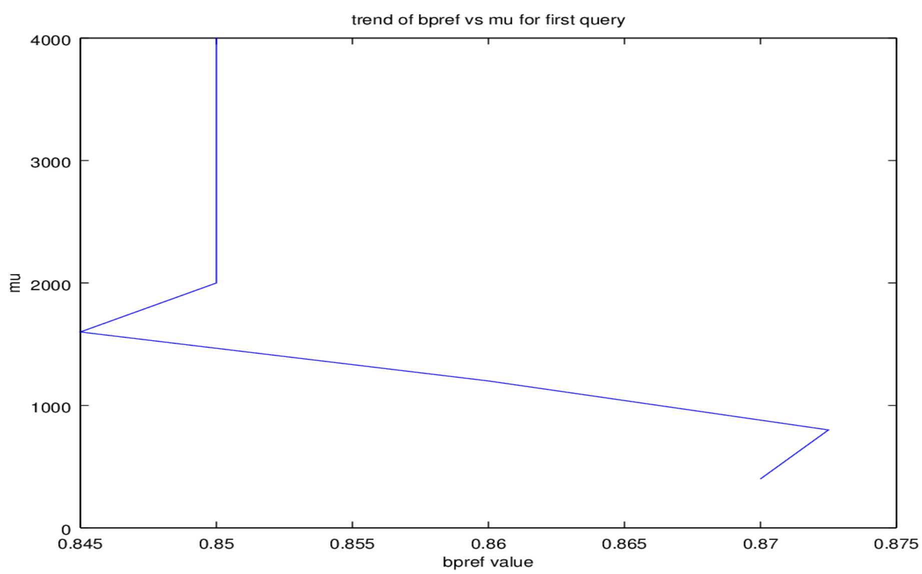
<similarity class="org.apache.solr.search.similarities.LMDirichletSimilarityFactory">

<float name="mu">1200</float>

</similarity>







LMSimilarity model gives the best parametric results among the three models.

THE END