INLS 490-154: Information Retrieval Systems Design & Implementation. Spring 2009.

5. Retrieval models-1

Chirag Shah*
School of Information & Library Science (SILS)
UNC Chapel Hill NC 27599
chirag@unc.edu

1 Introduction

A model of information retrieval that we have been following is shown in Figure 1. So far we have covered (1) indexing as a way to represent the text documents, (2) parsing the query as a way to represent the user's information need, and (3) matching the processed query to the index for retrieval. We will now focus on the matching and retrieving part. For this, we will look at a variety of retrieval models. We have already seen the vector space model. In this class, we will explore boolean retrieval as well as language modeling approach to retrieval.

Before we proceed, it is important to note what we mean by a retrieval model here. A model, in general, is an abstraction of an actual process. In that sense, a retrieval model is a kind of representation of the retrieval process. Since retrieval process is dependent on various components such as indexing and similarity function, it is difficult to separate the retrieval only part from other parts of the system. Therefore, while talking about retrieval models, we would be referring to the whole underlying process of pre-processing to matching and retrieving.

2 A matter of matching

There are two major approaches to matching a document with a query: (1) exact match, and (2) best match. The former is typically realized by the means of boolean matching and retrieval, and the latter is implemented with different models that account for approximations and probabilities. Let us explore these two methods in detail.

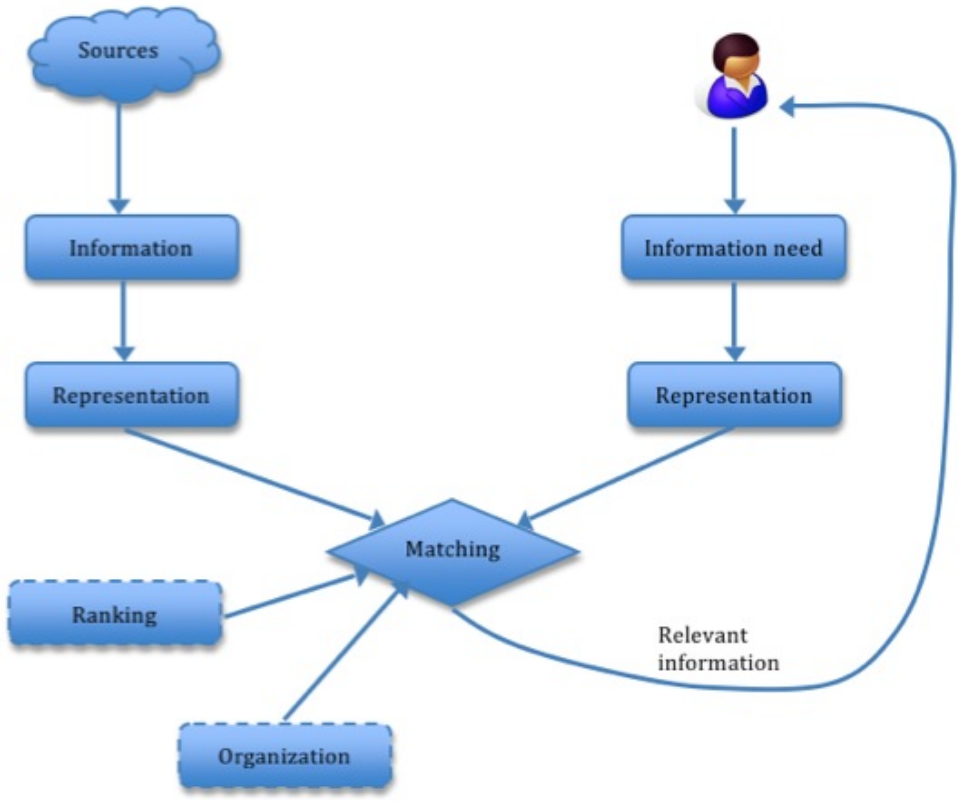

Figure 1: A model of Information Retrieval (IR)

2.1 Exact match (boolean)

This is a model where the query specifies a very precise matching criteria. A typical example of a boolean query is *java AND coffee*. This query indicates that a matching document has to have both *java* and *coffee*. It means if a document has only one of these terms, it will not be matched. This can be an advantage or a disadvantage depending on the situation. Also, in the pure exact match situation, the result for a given query is a set of documents without a ranking. This is because for a exact match situation, a document either matches to the query or it does not (binary relevance).

2.2 Best match

In this model, a query specifies what a good or best matching document could be. Every document has some chance of matching the query. This "chance" could be measured using a probability or expectation, and used to rank the results. Thus, with this model, we get some kind of score for each document with respect to the query that can be used to rank or organize the matching documents.

Each of the above models have their pros and cons. They are summarized in Table 1.

Table 1: Comparison of exact match and best match retrieval models

Exact match	**Best match**
Easy to understand and explain.	More difficult to convey the cognitive model of "relevance".
More useful to an advance searcher.	Quite usable even for a novice searcher.
Can be efficiently implemented.	Less efficient. Cannot reject documents early.
Difficulty increases with collection size.	Scales well.
Higher precision comes at the cost of recall.	Getting higher precision does not result in harming the recall much.

3 Boolean retrieval with Lemur

Lemur allows structured queries execution. We will now see how we can use this feature to create boolean queries. It is important to note that this feature does not do pure exact match. A structured query for Lemur is represented as

```
#q<id> = <operator> (<query_terms>);
```

A query file can have several queries and each query should have a unique identifier represented here as `<id>`. Possible operators for constructing a boolean query in Lemur are *#band* (boolean and), *#not,* and *#bandnot*. Operator *#bandnot* takes two arguments: the first one to be considered as *#band* and the second one as *#not*. Once a query file is created, it can be parsed using *ParseInQueryOp*. This application, which is similar to *ParseToFile*, takes the name of a parameter file as its first argument and name(s) of the query file(s) as consecutive arguments. A sample parameter file is shown in Code 1.

Code 1: Parameter file for ParseInQueryOp

```
<parameters>
    <outputFile>query_parsed.txt</outputFile>
    <docFormat>trec</docFormat>
    <stopwords>stopwords.list</stopwords>
    <stemmer>porter</stemmer>
</parameters>
```

The output of *ParseInQueryOp*, in the above case, will be stored in a file named `query_parsed.txt`. This parsed query file can now be used for doing the retrieval. This time, instead of *RetEval*, we will use application named *StructQueryEval*, which takes one argument - name of a parameter file. Such a file is shown in Code 2. The output of this application is similar to the one resulting from the execution of *RetEval* - first column with the query ID, second column with the document ID, and the third column with the score.

Code 2: Parameter file for StructQueryEval

```
<parameters>
    <index>myindex</index>
    <textQuery>query_parsed.txt</textQuery>
    <resultFile>results.txt</resultFile>
    <resultFormat>3col</resultFormat>
    <resultCount>5</resultCount>
    <retModel>inq_struct</retModel>
</parameters>
```

4 Language modeling

Now we will consider another approach to retrieval - language modeling. This approach is inspired by a whole stream of research in speech processing. The core idea here is to evaluate the probability distribution of a given string in a language. Let s be a string with terms $t_1, t_2, ..., t_k$ in a language L. Considering the terms in this string independent, the probability of s being generated by L can be calculated as

$$\begin{aligned} P(s|L) &= P(t_1, t_2, ..., t_k|L) && (1) \\ &= \Pi_{i=1}^{k} P(t_i|L) && (2) \end{aligned}$$

Now imagine s is a query and L is a document. Thus, we could calculate the probability of query q generated by document D can be calculated as

$$\begin{aligned} P(q|M_D) &= P(q_1, q_2, ..., q_k|M_D) && (3) \\ &= \Pi_{i=1}^{k} P(q_i|M_D) && (4) \end{aligned}$$

Here, M_D is the document model, a representation of the document. In the above formulation, $P(q|M_D)$ measures the probability or the likelihood of query q being generated by document D. Therefore, this approach is also called the *query likelihood model*.

5 Language modeling with Lemur

The idea of language modeling presented above can be implemented in many ways. In Lemur, this is done by a measure called Kullback-Leibler (KL) divergence. In general, KL divergence is used to evaluate the difference between two probability distributions. In our case, this will be between the probability distributions of a document and a query. Further explanation of this measure is beyond the scope for us. We will go ahead and use it in Lemur.

In order to use a language model in Lemur, we will use *RetEval* with 'kl' as the retrieval model. A sample parameter file is shown in Code 3. The output of *RetEval* should be similar to what you have seen before, except the last field with the scores, which should have negative numbers. This is due to the way KL divergence is calculated (involving log function of a fraction).

Code 3: Parameter file for RetEval with KL retrieval model

```
<parameters>
    <index>myindex</index>
    <textQuery>query_parsed.txt</textQuery>
    <resultFile>results.txt</resultFile>
    <resultFormat>3col</resultFormat>
    <resultCount>5</resultCount>
    <retModel>kl</retModel>
</parameters>
```

Often, KL divergence scores are proportional to those of TFIDF and so, it may not be very surprising if you see similar ranking of the documents for both TFIDF and KL approaches of retrieval models.

6 Summary

- Two major ways of matching a query with a collection of documents: exact matching and best (approximate) matching.
- Boolean queries facilitate exact matching.
- Statistical language models are very appealing for best matching scheme.
- Language models estimate the probability that the given query is generated by a document.