Biomedical Information Retrieval based on Document-Level Term Boosting

Dagur Valberg Johannsson

Master of Science in Informatics
Submission date: June 2009
Supervisor: Herindrasana Ramampiaro, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Biomedical information retrieval based on document-level term boosting

Dagur Valberg Jóhannsson

June 8, 2009

Abstract

There are several problems regarding information retrieval on biomedical information. The common methods for information retrieval tend to fall short when searching in this domain. With the ever increasing amount of information available, researchers have widely agreed on that means to precisely retrieve needed information is vital to use all available knowledge. We have in an effort to increase the precision of retrieval within biomedical information created an approach to give all terms in a document a context weight based on the contexts domain specific data. We have created a means of including our context weights in document ranking, by combining the weights with existing ranking models. Combining context weights with existing models has given us document-level term boosting, where the context of the queried terms within a document will positively or negatively affect the documents ranking score. We have tested out our approach by implementing a full search engine prototype and evaluatied it on a document collection within biomedical domain. Our work shows that this type of score boosting has little effect on overall retrieval precision. We conclude that the approach we have created, as implemented in our prototype, not to necessarily be good means of increasing precision in biomedical retrieval systems.

Preface

This thesis is my final work on my masters degree, and marks the end of my study on information management at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

Thanks goes to my supervisor, Heri Ramampiaro for valuable feedback, suggestions and ideas. I would also like to thank my colleagues at Searchdaimon for their feedback. Last, but not least I would like to thank my parents, as well as the rest of my family, for their support and motivation; particulary to my father Jóhann who has genuinely shown interest in this thesis.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Background and motivation

In [7] Ramampiaro describes several problems regarding information retrieval on biomedical information. The common methods for information retrieval tend to fall short when searching within this domain. Searches tend to be either too restrictive, causing low recall, or they tend to become to broad, giving low precision. Thus they become unsatisfactory with respect to user needs.

Several methods have been used to customize retrieval technology to exploit domain-specific attributes of the document collection being searched in, increasing recall, precision or both. These have several challenges to overcome. In biomedical information there exists a lot of domain-specific terminology [18]. Mixing natural English and biomedical-specific terms can be challenging due to high ambiguity. A word can have several meanings, and gain more with time[19]. In addition there is a lack of terminology standards, thus new ones are created as seem fit and there often exist several typographical and lexical variants[18].

1.2 Problem definition

Given the quality of information retrieval using common retrieval methods on biomedical information, or lack there of, we believe it's possible to improve information retrieval to better suit user needs. We do this by using the domain-specific attributes of the document collection we search within. In this thesis our main focus is on improving precision. We attempt to increase precision by introducing an additional weighting factor when calculating term relevance, a factor we generate by analyzing the terms context within the document.

1.3 Layout of the thesis

This thesis is divided into 5 parts. These are as follows

Intoduction This part, a brief introduction to the thesis explaining it's background and problem definition.

Concepts and definitions Introduces concepts and definitions used throughout the the thesis. These are concepts that we build and evaluate our work upon, thus deem as important background information.

Related work Introduces other work and research done within the same field that we relate this work to.

Approach Describes our thought out approach on how we wish to solve the problem definition.

Implementation Contains details on how we have implemented the prototype that we created to test our approach.

Experiment and results Presents how we evaluate our prototype, and presents the evaluation results and interpretation of these results.

Discussion and conclusion This part discusses our work and concludes on our approach based on our evaluation.

Part II

Concepts and definitions

Chapter 2

Ranking models

A central topic of information retrieval is document ranking. Ranking is important, as it limits the amount of documents a human has to sift through to find the document he is after; especially when a large amount of documents are recalled. Ranking allows us to assign a score to a (query, document) set for the means of attempting to give better score to the more relevant documents. Results are then sorted showing the highest scoring documents to the user first. This section briefly introduces common ranking models, Okapi BM25 and Vector Space Model (VSM) on which we build our work upon.

2.1 tf-idf

tf-idf is a statistical measure used to evaluate how important a term is in a document collection. This measure works by increasing the weight of a document by how often it is found within the document (term frequency), but is offset by how often the term is found within the entire collection (inverse document frequency). Variations of tf-idf are widely used by search engines as a central part of their ranking strategy. The td-idf weight for term t_i in a document d_j is defined as

$$w_{i,j} = tf * idf$$

where tf is the *term frequency* and *idf* is the *inverse document frequency*. Term frequency is a normalized measure of how often the term is found within the document. The measure is normalized as so it does not favor longer documents, where the term may have a high term count regardless of its importance.

2.2 Okapi BM25

Okapi BM25 is rather than a single function ranking model, a collection of several models. It disregards term inter-relationships and weights them as a bag-of-words, meaning it does not look at term order, nor grammar. The following combination of models called BM25 is as follows [4]

$$score(D,Q) = \sum_{i=1}^{n} \frac{idf(q_i) * tf(q_i, D) * (k_1 + 1)}{tf(q_i, D) + k_1 * (1 - b + b * \frac{|D|}{avdl})}$$

Where $f(q_i, D)$is term frequency in document D, $|D|$ is the length of document D and $avgdl$ is the average length of documents in the collection. $k_1$and b are tuning constants. This formula ensures that the effect of the term frequency is not to strong, and that the weight for a term occurring once in a document of average length is just idf of that term[4].

The constant $k_1$may be tuned to influence the effect of term frequency, in Text REtrieval Conference (TREC)[1] this constant was found to be effective at $k_1 = 2$. The constant b is used to modify the effect of the document length. Setting $b = 1$ would mean that documents are long because they are repetitive, while setting it to $b = 0$ assume they are long because they are multitopic. In TREC, this value was deemed helpful at $b = 0.75$.

In Okapi BM25, idf is calculated as

$$idf(q_i) = log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

Where N is the number of documents in the collection, and $n(q_i)$ is the number of documents containing q_i.

2.3 Vector Space Model

Vector Space Model (VSM) is representation of a set of documents in a common vector space, first introduced by Salton[11]. It has, and still is widely used in information retrieval for scoring, as well as document classification and clustering[5].

When using VSM, each document is given a vector measure $\vec{V}(d)$, with a component for each of the terms within the document. Components are usually computed using tf-idf. When scoring a document against a query, the query is treated as a small document containing the terms represented in the query, giving us vector measure $\vec{V}(q)$.

One way of measuring ranking score is to use cosine similarity between the vectors, defined as[5]

$$score(q,d) = \frac{\vec{V}(q) * \vec{V}(d)}{|\vec{V}(q)| * |\vec{V}(d)|}$$

This similarity measure may also be used to calculate the similarity between two documents.

[1]TREC is a series of workshops with focus on information retrieval within different research areas. A research area is also called *track*.

Chapter 3

Evaluation

In order to compare our results with existing ranking schemes, we require a mean of evaluating the retrieval results. This section introduces well known methods of evaluation.

Buckley and Voorhes [6] have shown that common evaluation measures are not robust when working with incomplete judgments sets. As we're evaluating against a collection with incomplete relevant judgments, we will evaluate against mean average precision (MAP) (section 3.3) which has been shown to be more stable and robust measure for this type of sets [6].

3.1 Recall and precision

Many of the most frequently used evaluation methods are derived from recall and precision[6]. Recall and precision is the most common way to measure retrieval performance. The method calculates how many of the relevant documents were retrieved, and how well the relevant documents were ranked, among the non-relevant ones. Precision is the proportion of the documents that are relevant. Recall is the proportion of the relevant documents that were retrieved. Given the a cutoff point r, the *precision* P_r is calculated as

$$P_r = \frac{number\,retrieved\,that\,are\,relevant}{total\,number\,of\,retrieved}$$

Here $P_r$will indicate the proportion of the documents that are relevant. For example, if 5 out of 10 documents retrieved are relevant, then $P_r$will become $P_r = 5/10 = 0.5$.

The *recall* $R_r$is calculated as

$$R_r = \frac{number\,relevant\,that\,are\,retrieved}{total\,number\,relevant}$$

Here $R_r$will indicate the proportion of the relevant documents that were retrieved. If 5 out of 10 documents, then R_r becomes $R_r = 5/10 = 0.5$.

3.2 R-precision

R-precision is the precision after R documents have been retrieved where R is the number of relevant documents for the query.

To be able to use R-precision, one has to have a list of relevance judgments before performing the retrieval. The judgments can be complete or not, but the measure is not robust on incomplete judgment sets [6].

3.3 TREC mean average precision (MAP)

Mean average precision (MAP) has become the most standard measure to evaluate information retrieval within the TREC community [5], and has been used in TREC Genomics Track evaluation [2]. MAP is mean of the precision scores after each of the relevant document have been retrieved, using zero as precision for relevant documents not retrieved. MAP uses a lot more information than simpler measures, such as R-precision and *precision at k*. This makes it a more powerful and more stable measure [6]. The drawback of this is that it is harder to interpret. A measure of 0.3 can be caused by many factors with MAP, while the same R-precision value on a query having 10 relevant documents would mean 3 relevant documents were recalled. MAP is defined as[5]

$$MAP(Q) = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{m_j}\sum_{k=1}^{mj}Precision(R_{jk})$$

where the set of relevant documents for query$q_j \in Q$ is $\{d_1...d_{m_j}\}$ and $R_{jk}$is the set of ranked retrieved documents down to document d_k.

Chapter 4

Document indexing

Document indexing is the task of processing a document collection for the purpose of building data structures used during document retrieval. In this section, we introduce one means of building a structure for retrieving documents based on query terms; *inverted index*.

4.1 Inverted index

The basic idea of a inverted index is to keep a dictionary of terms, and for each of the terms keep a list of records for which documents the term is found in[5]. Given documents $d_0 = ''Kent\,is\,superman''$, $d_1 = ''superman\,is\,strong''$ and $d_2 = ''Lane\,likes\,Kent''$, an index could be constructed as in table 4.1.

If a user wanted to look up documents having the term *superman*, the system would look up the term in the index and return the documents listed; in this case d_0, d_1 and d_2. In more complete information retrieval systems, additional values are often stored within the inverted index, such as the terms position within the document.

Term	Documents
Kent	d_0, d_2
is	d_0, d_1
superman	d_0, d_1, d_2
strong	d_1
from	d_1
Krypton	d_1
Lane	d_2
likes	d_2

Table 4.1: Term and documents as stored within an inverted index.

Part III

Related work

Chapter 5

Increasing recall and precision

5.1 MeSH Query expansion

The effectiveness of using Medical Subject Headings (MeSH)[1] for automatic query expansion in PubMed is investigated in [8]. In their experiment, they constructed queries by selecting keywords from questions on 55 topics. These queries were then expended using Automatic Term Mapping (ATM), by making use of MeSH field of indexed MEDLINE[2] citations. ATM works by comparing terms in the user query, to pre-indexed terms in translation tables, such as MeSH table. When user for instance searches for *tumor*, the query will be automatically mapped to the MeSH term *Neoplasms*. This means that query will in addition to retrieving articles containing the term tumor, return articles containing Neoplasms.

Their experiment suggests that query expansion on PubMed can improve retrieval performance, generally resulting in more relevant documents. Though this improvement may not be useful for users that do not look past the topmost results.

5.2 Relevance feedback

The basic idea of relevance feedback (RF) is that the user submits an initial query, and the system returns ranked results. The user then judges what documents in the results are relevant. Based on the user judgment, the system constructs a new query based on terms in the original query, and terms found in the documents found relevant. The system then returns results based on the constructed query. This iteration can then be repeated if needed. Relevance

[1]MeSH is a comprehensive controlled vocabulary created and updated by United States National Library of Medicine. In 2008 it contained 27,767 subject headings arranged in a hierarchy, this makes MeSH suitable to be used as a thesaurus in information retrieval.

[2]MEDLINE (Medical Literature Analysis and Retrieval System Online) is a literature database of life sciences and biomedical information.

feedback exploits the idea that it is difficult to express a good query when the document collection is unknown to the user, but it's easy for the user to deem if documents relevant or not.

Some assumptions are needed for RF to be useful. The user has to have sufficient knowledge to make the first query retrieve some relevant documents. Relevant documents need to be similar to each other, as in cluster together, and the irrelevant documents should not be similar. The user has to be willing to provide feedback and be willing go through with prolonged search. The result set from a restructured query may also confuse the user, as he may not understand why the particular documents were retrieved.

Variations to explicit user judgments exist. *Pseudo relevance feedback* assumes that topmost k documents are relevant, and uses these as basis to construct a new query. *Indirect relevance feedback* keeps track of what documents in the query results the user looks at. The assumption that a document is relevant to a query, when many users view this document.

RF has been shown to be very effective at improving retrieval relevance[5].

5.3 Boosting specific document parts

Boosting specific document parts is a means of exploiting the structure of structured documents to increase precision. As an example; citations provided by MEDLINE has been structured in a way that allows one to extract title, abstract and author along with other fields separately. Research has been done on weighting specific document parts differently, showing that for instance weighting the *title* field of a MEDLINE citation twice as much as the *abstract* yields better ranking[7].

Chapter 6

Retrieval systems

6.1 PubMed

PubMed is a search engine publicly available for searching within MEDLINE citations. In addition to providing citations, it provides link to articles full-text where available. PubMed provides Boolean queries, in addition to using MeSH headings to expand queries and giving access to related articles. As of 2009, PubMed indexed 17,764,826 citations[1]. PubMed does not seem to rank searches by document relevance, [7] suggests that PubMed seems to order search results based on publication date, author names and journals.

6.2 Textpresso

Textpresso[17] is a text-mining system allowing specialized search of scientific literature. Textpresso implements categories of terms, these being classes of biological concepts (genes, cell), objects relations (association, regulation) or classes that describe biological concepts (biological process). These categories form an ontology. Textpresso splits papers into sentences, and these into word or phrases that are labeled to their ontology. This allows expressible searches that exploit the categorization. Textpresso ranks their results using frequency of queried index term, which ranks documents that contain the largest number of queried terms at the top.

6.3 BioTracer

BioTracer is a proof-of-concept prototype that has has implemented several information retrieval techniques in means to investigate how to improve the ability for a system to find and rank relevant documents[7]. The ranking in BioTracer

[1]Citation totals retrieved from http://www.nlm.nih.gov/bsd/licensee/2009_stats/2009_Totals.html

is built up on existing similarity models, such as BM25. It extends these models factoring in the techniques boosting specific document parts, allowing users to affect the ranking trough user relevance feedback (URF), adding a query-document correlation factor to the model as well as supporting additional query expressions. BM25 was originally designed for keywords-based queries, in BioTracer the model has been extended allowing Boolean operators (AND, OR, NOT). BioTracer also supports the use of wildcards.

Based on evaluation against a subset of TREC 2004 Genomics[2], the extensions implemented in BioTracer has been shown to help improve retrieval performance.

[2]TREC Genomics Track data is available at http://ir.ohsu.edu/genomics/data.html

Part IV

Approach

Chapter 7

The idea

The basic idea is to use text mining techniques to give each term within a document a weight based on their biomedical information context. When a document is retrieved as it matches a query term, we use this weight as an additional measurement to determine how relevant the document is to the query.

In our approach we define the context of the term to be the sentence it is found in within a document; giving each sentence in a document a weight. The weight is determined by pin pointing biomedical relevant information in the sentence.

The purpose of this is to give us means to boost the term match if the context of the query match has strong biomedical context; as well as to give it a negative boost if it's context is in essence just natural English. Doing this will boosting the document itself during ranking on strong biomedical context matches.

By doing this, we attempt to find out if this additional context factor will increase the overall precision of an ad-hoc information retrieval query. When the terms in a user query have a strong biomedical context within a given document, the document might be more likely to be relevant to the query. This hypothesis is strengthened by the fact that the domain searched within is scientific in nature, and thus users performing the searches are likely to be researchers, scholars and there like; generally users that are able to make educated searches.

7.1 Document collection

To evaluate our prototype, a *relevance judged* document collection within biomedical domain is preferred. A set of relevance judgments is a list of relevant documents, so called "right answers" for a given topic or topics. This gives us a baseline to evaluate against, and allows us to use evaluation methods such as R-precision and mean average precision (described in chapter 3). TREC regards a document relevant if one were to write a report on a topic, and would use some information in the document[1].

7.2 Document-level term boosting

As stated, we have defined the context of a term to be the sentence it is used in, and this context has to be weighted. This weight is then to be used to boost, or negatively affect the documents rank. To achieve this, the following needs to be done

- Extract each sentence out of the document.
- Run text mining techniques on the sentence to determine biomedical information.
- Determine a weight for the sentence.
- Use this weight during ranking.

This section describes how we solve each of these individual problems.

7.3 Sentence extraction

Automatically extracting sentences out of a document is a subject of natural language processing called *sentence segmentation*. We assume the contents of the documents are in English; thus dividing the document at punctuations might seem reasonable approximation. It is however not as trivial as that. Challenges within sentence segmentation are; amongst others

- Abbreviations and acronyms such as “Mr.”
- Sentences within sentences, such as: *He said: “I need a cup of Earl Grey”, and went into the kitchen.*

Although sentence segmentation is out of the scope for this thesis, we will briefly describe the method used in the prototype, an approach loosely based on an article by Mikheev[12].

Having done lexical analysis and divided the document into a set of tokens, it uses a heuristic model with three sets of token types

- Possible stops: A set of tokens that are allowed to be the final token, such as periods (.) and double quotes (“).
- Impossible Penultimates: A set of tokens that may *not* be the second-to-last token of a sentence, such as “Mr”.
- Impossible Starts: A set of tokens that may *not* be the first token of a sentence, such as punctuation characters like end quotes (”) that should be a part of previous sentence.

[1]As per definition on the TREC website; http://trec.nist.gov/data/reljudge_eng.html

This is the base heuristics used, in addition to other conditions[2].

7.4 Weighting by biomedical relevance

For measuring biomedical relevance of each sentence, we use name entity recognition (NER) for recognizing biologic terms and concepts. NER is a sub-task of information extraction used to identify and classify within a document collection all instances of names for a specific type of thing, such as genes, name of persons, locations and so forth. It has been suggested that solving the task of proper NER will allow more complex text-mining tasks to be addressed[13].

NER has been challenging for biomedical texts[13], as

- There does not exist a complete dictionary for most types of biomedical named entities.
- The same word or phrase can have different meanings, based on context (ferritin can both be a biological substance, and a laboratory test).
- Biological entities can have several names (PTEN and MMAC1 are the same gene), as well as the names can have multiple words.

The main approaches to NER is lexicon-based, rules-based and statistically based. One of the models used in our prototype is a statistical model that uses a hidden Markov model[3]. The other uses a statistical machine learning technique that generates vectors from token shapes[4].

7.4.1 Using Name Entity Recognition for weighting

Having done sentence extraction, we now have a set of sentences S for a document D_i. We then weight each of the sentences $s_j \in S$ using

$$sw(s) = |R|$$

$sw(s)$ is a function to weight sentence $s_{i,j}$ and where R is a set of NER recognized terms and concepts in the sentence, $|R|$ being the size of this set.

7.4.2 Normalize weighting

In order for our weight factor to merely influence the current ranking models, rather than dominating the the ranking with our sentence weight, we need the weights to be normalized. Also, if weights are not normalized, terms within

[2]We use a library in LingPipe for this task. The method is described in more detail in LingPipes JavaDoc at http://alias-i.com/lingpipe/docs/api/com/aliasi/sentences/HeuristicSentenceModel.html

[3]GeneTag in LingPipe is built upon hidden Markov model chunker, described in http://alias-i.com/lingpipe/docs/api/com/aliasi/chunk/HmmChunker.html

[4]GENIA in LingPipe is built on token shaping, extended from the class TokenShapeChunker http://alias-i.com/lingpipe/docs/api/com/aliasi/chunk/TokenShapeChunker.html.

longer sentences will be favored due to more recognized entities, giving biased weighting. For this, we have used the following normalization formula

$$nsw(s) = \frac{(k_1 + 1) * (sw(s) + 1)}{sl(s) + (sw(s) + 1)}$$

where $k_1$is a free constant, $sw(s)$ is the function to weight sentence $s = s_{i,j}$ in document d_i and $nsw(s)$ gives us the new normalized sentence weight.

When nothing of biomedical relevance has been identified within a sentence, the weight will be $sw(s_{i,j}) = 0$, thus we increment all weights by 1 in our normalization; this so the boosting factor doesn't zero out the document score. $sl(s)$ is our means of including the sentence length in the boost. $sl(s)$ is defined as

$$sl(s) = k_1(1 - b + b(\frac{|s|}{avgsl}))$$

Where b is a free constant, $|s|$ is the length of the sentence and $avgsl$ is the average sentence length in the collection.

This normalization is derived from Okapi BM25 (section 2.2). We set the constants to $k_1 = 2$ and $b = 0.75$ as we would with Okapi BM25 for lack of better tuning values to use for this type of normalization. Using this formula, our boosting factor will usually float between 1.0 and 2.0 with a few exceptions for extreme cases.

7.5 Using term boosting

Having created means of weighting each sentence within a document, we can use these weights to create a term boosting value. We set the boosting factor $tb_{d,t}$ to be the average normalized weight for all the sentences $S \in d$ the term is found within in document d; defined as

$$tb(d,t) = \frac{\sum_{i=0}^{tf(d,t)} nsw(ts(d,t_i))}{tf(d,t)}$$

Where $tf(d,t)$ is the term frequency of term t in document d, $ts(d,t_i)$ *(term sentence)* is the sentence the term t_i is located in within d , and $nsw(s)$ is the normalized sentence weight for this sentence.

7.6 Term boost combined with existing ranking models

We can now combine our term boost with existing ranking models. Given $score(d,t)$ is the score calculated by a ranking model, such as Okapi BM25 or VSM (described in chapter 2), we combine the weights as follows

$$bs(d,t) = score(d,t) * (tb(d,t) * k_1)$$

Where $bs(d,t)$ is our new boosted score for term t in document d and k_1 is a free constant used to tune the influence of our term boost factor. We set $k_1$to be $k_1 = 1$.

Chapter 8

Storing context weights

Our methods for generating context based weights for terms require advanced text mining techniques. In addition, the entire text of the document needs to be loaded, most often from primary storage systems. Both are performance expensive tasks and when combined can consume great amount of CPU- and I/O-time. Performing context based weighting during retrieval would therefor be to resource demanding and slow in practice.

Generating the context based weights during document however, is reasonable. Indexing does not have the same requirements to performance, nor demands on response time. In addition, the weights are static, and need only to be done once, rather than every time the document is retrieved.

This does however require that the weights are stored for later use; stored in a way that they can be looked up on a per-term-basis within a document. In this section we suggest two methods of storage and compare these to each other.

8.1 Store in an inverted index

We described in section 4.1 how an inverted index is built up. It is possible to extend an inverted index to store the term boost value, along with the document reference. Building up on the term *superman* in the example used in table 4.1; we set the term weights for the documents d_0, d_1, d_2 to be w_0, w_1, w_2 respectively. Our new inverted index would now be built up as

$$superman \rightarrow [d_0, w_0], [d_1, w_1], [d_2, w_2]$$

This requires that a weight is calculated for every term within a document during indexing. It may increase the storage size of the inverted index significantly. During retrieval, finding the term boost would be the simple task of looking up the boost in the inverted index.

8.2 Store in a separate index

An alternative to storing term boost weight in the inverted index, is to have a separate sentence index. A document needs to have a identifier that can be looked up in the index; and it should point to a list of structures that hold the end position and weight for sentences in that document. A single index look up is illustrated with the following

$$document \rightarrow [endposition, weight], [endposition, weight] ...$$

where *document* is the document identifier pointing to a list of sentence pairs; each containing an *end position* and the *sentence weight*.

Given that this set is sorted by end position, and that we have term frequency and position available during lookup, we can calculate the weight as described in section 7.5 by

- Iterating over the sentences, matching the term to it using it's position.
- Summing up the sentence weights.
- Dividing the total weight with the term frequency.

The following is a pseudocode demonstrates this

```
Iterator<Sentence> sitr = sentenceIndex.iterator(document);
Sentence sentence = sitr.next();

float totalWeight = 0;

for (int i = 0; i < term.freq(); i++) {

        // match term to its sentence
        while (sentence.endPosition() < term.nextPosition()) {
                sentence = sitr.next();
        }

        totalWeight += sentence.weight();
}
return totalWeight / term.freq();
```

where *sitr* is used to iterate over the sentences in a single document, *term* holds document term frequency and position, and *sentence* holds the sentence position and weight.

8.3 Comparison

Summarized comparison is shown in table 8.1. Using inverted index, the index may become larger than when using a separate index, as weight values have to

Inverted index	Separate index
Weights are stored on per-term basis.	Weights are stored on per-sentence basis.
Per-term weighting is done during indexing.	Per-term weighting is done during retrieval.
Weights can co-exist with existing architecture.	New index requires additional architecture.
May require larger changes to existing system.	May require lesser changes to existing system.

Table 8.1: Comparison of storing weights in inverted index vs. separate index.

be stored for all terms in each document; rather than for all sentences. This, however, allows all weighting to be done during indexing. When using a separate index, term weighting has to be done during retrieval, using the sentence weights.

Inverted indexes are widely used, architecture and techniques have already been developed for maximizing performance, redundancy, distribution and so forth. We argue that including weights in existing inverted index architecture is likely to be better than introducing a separate index; requiring these techniques to be re-implemented. Introducing a separate index may however not require altering existing indexed data structure, if document identifiers are used as lookup values.

Part V

Implementation

Chapter 9

Overview

We have in previous chapter described our approach to creating a document-level term boosting factor. This chapter describes implementation specific details and decisions of a prototype made for the purpose of testing and evaluating this idea.

9.1 Technology overview

To ease, shorten development time and refrain from “reinventing the wheel”, our implementation takes advantage of libraries previously developed for the common task of search retrieval and text mining. We settled on using Apache Lucene for search, and LingPipe for text analysis. Both are technologies written in Java, thus Java has become our language of choice for the prototype.

9.1.1 Apache Lucene

Apache Lucene[1] is a full-featured text search engine library. It’s not a search engine by itself, but provides library and API to rapidly build one. We base our prototype fully on Lucene, using it both for indexing and retrieval, as well as modifying and expanding on it.

9.1.2 LingPipe

LingPipe[2] is a suit of Java libraries for linguistic analysis of human language. We use LingPipe for *sentence segmentation* (described in section 7.3), biomedical *name entity recognition* (section 7.4) and for parsing and handling MEDLINE citations.

We use LingPipes sentence segmentation rather than the native library *BreakIterator* in Java, as LingPipe has an sentence model created specifically

[1]Apache Lucene is available at http://lucene.apache.org/java/docs/index.html
[2]LingPipe is available at http://alias-i.com/lingpipe/

for handling MEDLINE citations, having 99% accuracy on detecting sentence boundaries according to their own evaluation[3].

9.2 Architecture overview

This section gives a basic overview of the architecture of our prototype. Although we base our work up on Lucene, we will not include lucene internals, merely our implementation around it. We describe the architecture of *document indexing* and architecture *document search* by them self. Although they work on the same data and indexes, we deem these to be two unrelated processes.

9.2.1 Document indexing

As the prototype is based on Lucene, we've based it on the indexing mechanism provided by Lucene itself as much as possible. However, as noted in chapter 8, weighting sentences by text mining them for biomedical information is performance intensive work and not feasible to do ad-hoc during document search. We therefor do this during indexing.

In chapter 8 we described two methods of storing weights, one being in the inverted index and the other in a separate index. As we wish to make as few modifications to the Lucene library itself, we decided up on using a separate index. This does require some modifications to Lucene, however in an lesser extent, and less error prone.

Thus, two indexes need to be built, the regular Lucene index and our *sentence index*. Our implementation of the sentence index differs somewhat from the method described in section 8.2 in that we do not use document identifiers too look up weights. The way lucene works, is that documents are divided into *fields*, for example one field for *topic*, one for *body* and so on. Each of these fields are indexed by them selves. In order to be able to use sentence weights on several fields within the same document, we create a unique lookup value for each field when indexing. This is done by reserving a *position* within our sentence index, using this position as our lookup value and storing it within the field along with other field data. This required some modifications to lucene and is described in more details in section 10.2.1. More modifications done to lucene in order to build our index are found in section 10.2.3.

9.2.1.1 Building lucene index

Our part in document indexing is implemented in two steps. The fist step is common in Lucene-based applications, and is centered around reading the documents and adding them to the index. The fist step is illustrated in figure 9.1[4]. Using LingPipes MedlineParser, we traverse the TREC XML file with

[3]Evaluation can be viewed at http://alias-i.com/lingpipe/demos/tutorial/sentences/read-me.html

[4]The internal architecture of Lucene indexing is deemed out of scope for this thesis, and illustrated with a cloud in the figure.

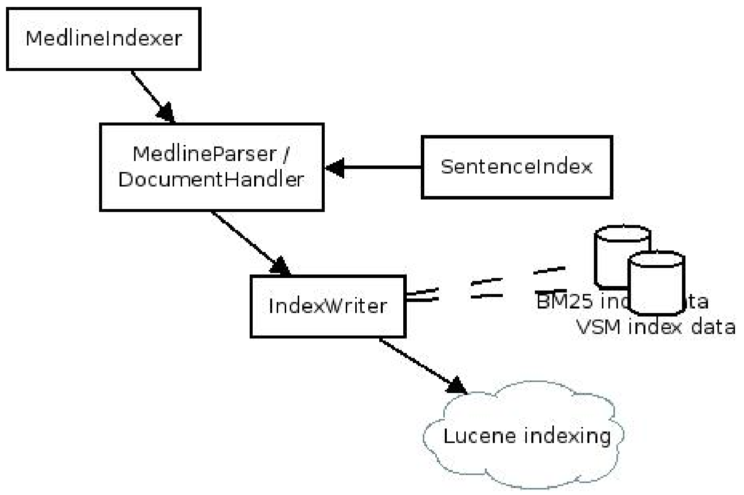


Figure 9.1: MEDLINE documents parsed and added to lucene index.

citations and pass them on to a Lucene IndexWriter using DocumentHandler, a class extended from LingPipes MedlineHandler.

Noteworthy architecture decisions are

1. Invoking sentence index during document handling.
2. Having separate indexes for each of the main ranking models (BM25 and VSM described in chapter 2) we evaluate against.

Sentence index is invoked when indexing the abstract, in order to reserve a position within the index that can be used when weighting sentences in step two; storing this position in the abstract field.

Having separate indexes for each ranking model is necessary as these models have fundamentally different similarity models. Building a single index based on one of the models, and using it with the other would have a negative impact on its ranking.

9.2.1.2 Building sentence index

The second part of document indexing is done during Lucene token consumption, and is shown in figure 9.2. DocInverterPerField holds all the data needed to weight sentences and match them to internal positions of tokens as stored in Lucene index. Thus we make a call from DocInverterPerField to our indexer to weight the sentences in the abstract field. Implementation details are described in section 10.2.3. As with the first step, two separate indexes are created, one for each ranking model.

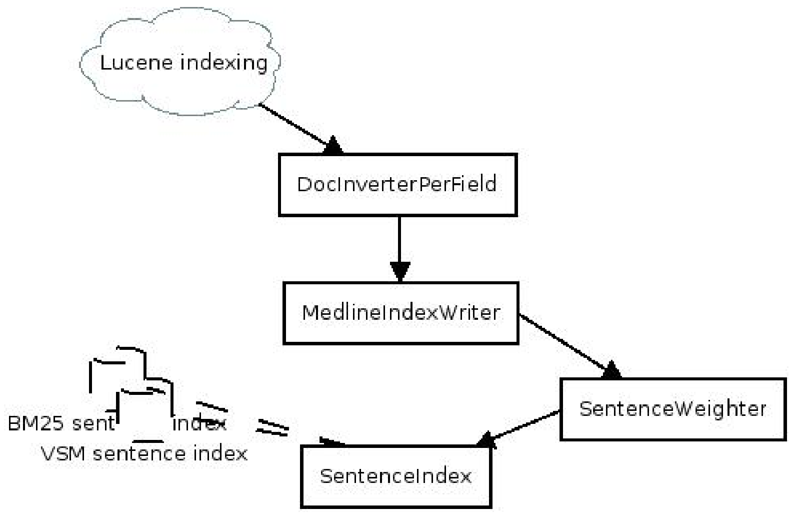


Figure 9.2: Sentences in abstracts weighted and weights stored in an index.

9.3 Document retrieval

When searching an index in Lucene, on it's simplest one has to

1. Create an IndexSearcher, pointing it to the location of the index.
2. Create a Query instance, by running search string through the query parser.
3. Run search on the IndexSearch using our query instance.

As with document indexing, two noteworthy architectural changes have had to been made during retrieval. First is the ability to select what index to use, based on the ranking model. The second is less trivial, and has to do with how document retrieval and scoring is done internally in Lucene. A *Query* instance holds a *Weighter* instance, and a *Scorer* instance, both central in how documents are ranked. Thus for each new model one creates, those three classes have to be implemented as well.

This also means we have to send our custom Query implementation to Lucene index searcher. Rather than working on Lucene's QueryParser, a seemingly complicated task, we implemented a query rewriter. Our rewriter initially parses the query as we would before, but then works on the query instance returned by QueryParser, replacing it with the selected models respective implementation. This is illustrated in figure 9.3.

Since we use a separate index for weights, two of the ranking models implemented need to access this index and calculate the term boost score as in the manner described in section 8.2.

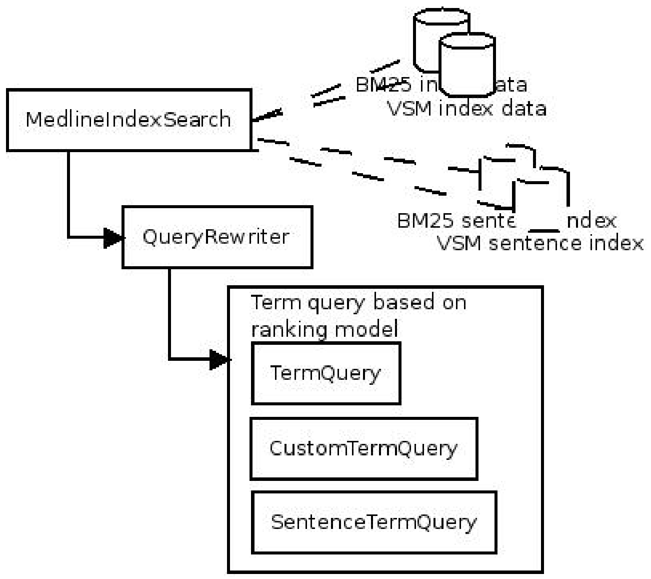


Figure 9.3: Depending on what ranking model is used, indexes are selected, query is parsed and rewritten for selected model.

Chapter 10

Implementation details

10.1 Preparing collection

The MEDLINE citations are available as 5 XML files approximately 4GB, a total size of 20GB.

As we wanted the judged subset of these documents to work with, a mechanism of filtering out all unjudged documents had to be created. A tool was written for this purpose[1]. It parses the judgment file, keeping track of citation identifiers[2]. It then uses those identifiers to filter out unjudged documents when traversing the 20GB of MEDLINE citations. As traversing a 20GB XML file is time consuming, the tool is used to create a new file, containing only the judged citations. This file is compatible with the MEDLINE files provided, thus can be parsed in the same way with our prototype.

10.1.1 Parsing citations

As described in section 10.1, the documents we work on are citations from MEDLINE, available in XML-files. Although Java has good support for XML-parsing, the task of parsing the citations is made even more trivial by using Ling-Pipes MedlinePaser implementation. This implementation provides an event-driven API, that lets us process each citation as it is parsed. This puts no limitation on how large the XML we're working on can be, as opposed to DOM handling where the entire XML is stored internally before it can be worked on; thus limited by the amount of internal storage. The following code snippet demonstrates how a MEDLINE XML file can be parsed using LingPipe. Here the method handle is called on each citation stored in the file MEDLINE_XML.

[1]This filtering tool is located in package `org.ntnu.masteroppg.tools.filter` in the prototype source code.

[2]Documents are identified in the judgment file, as well as in the XML citations using PMID (PubMed Identifier). PMID is a unique number assigned to each PubMed citation.

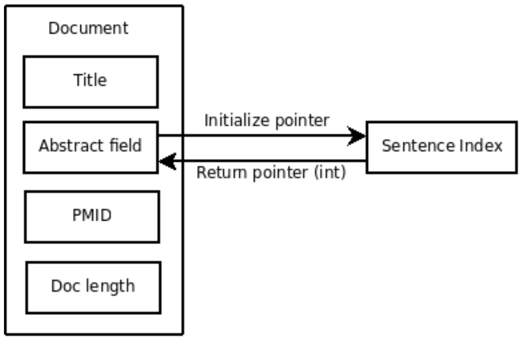


Figure 10.1: Document instance as when added to the index.

```
MedlineParser parser = new MedlineParser(false);
parser.setHandler(new MedlineHandler() {
    public void delete(String pmid) {
        throw new UnsupportedOperationException();
    }

    public void handle(MedlineCitation citation) {
        System.out.println(citation.article().articleTitle());
    }
});
parser.parse(MEDLINE_XML);
```

A MEDLINE citation contains richly structured data about a publication having author, affiliations, title, abstract, grants and medical subject headings (MeSH), amongst other data. This is available through the *MedlineCitation* instance sent to the *handle* method. In our prototype, we make use of pmid, abstract and title; the rest is ignored.

10.2 Document indexing

10.2.1 Sentence index API

The sentence index is used to hold the weight of every sentence indexed from the abstract in the document collection. Data is stored and accessed by having every abstract field hold a a reference to the internal position within the sentence index. When adding a document to the index, we call on the sentence index to initialize a position for given field. The sentence index then returns a reference to the initialized position. This pointer is then stored in the field that requires sentence weighting. In our experiment, we use sentence weighting for the abstract field, and figure 10.1 shows the fields of a document instance in our implementation.

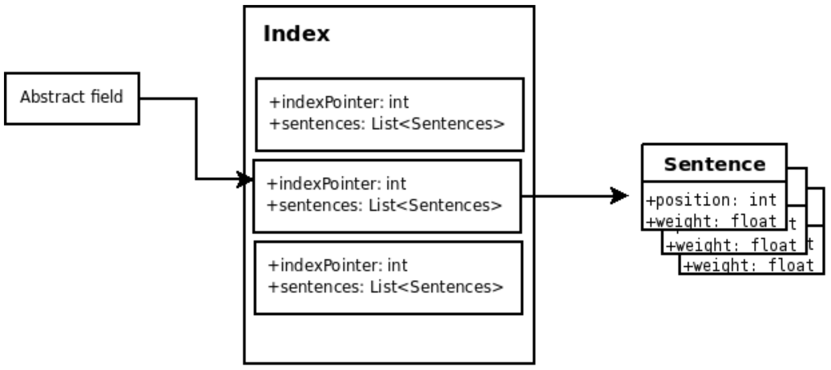


Figure 10.2: Current implementation of the sentence index.

10.2.2 Sentence index internals

Internal representation is as shown in figure 10.2. The position reference points to a indexPointer; which again holds a reference to an array of structs having the sentence weights, and the sentence positions within a document field.

We have not made any effort to distribute this index in our prototype implementation. The entire index is held in internal storage. This suffices for this experiment, as the size of the document collection indexed is of limited size. The index is built in internal storage during indexing, and stored when indexing is complete. During retrieval, the index is loaded into internal storage before search proceeds. The index is stored as a Java serialized object.

10.2.3 Building sentence index

Indexing sentence weights are done in a finit state machine. In Lucenes class *DocInverterPerField* each term within the field is processed in a sequential order. When processing the term, the field data the term belongs to, as well as the terms string offset are available. During term processing, the term is sent to the sentence index, along with this additional data. This is a small modification made to Lucene, described in detail in section 11.2. The sentence index forwards the call to our sentence weighter, which performs the task of building the sentence index itself.

10.2.4 Sentence weighter

The *sentence weighter* is our finit state machine implementation. The flow works as follows; when being called upon with a new term in a given field and the sentence weighter has not encountered the field before, it extracts the sentences out of the field, weights them using NER technique and stores them temporarily. It then works on each of the fields sentences, and attempts to match the character offsets of the terms to the character offset of the sentence to find

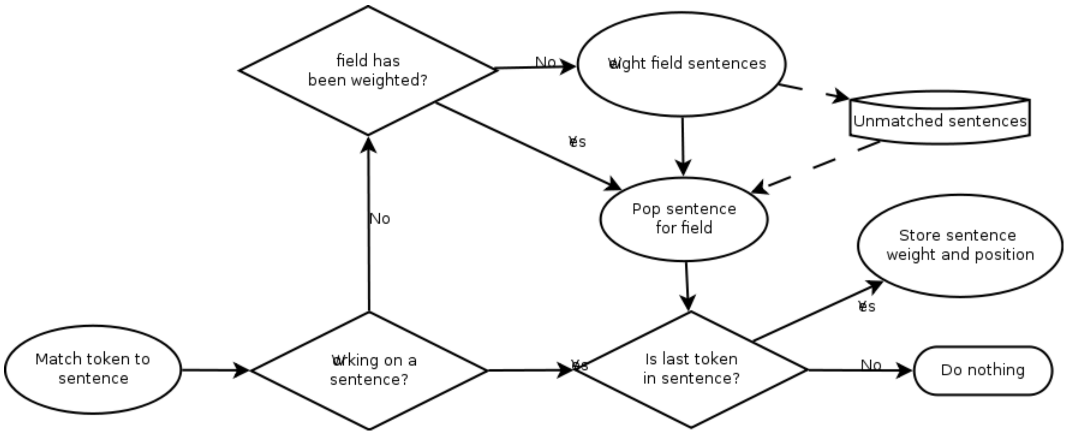


Figure 10.3: Flow chart that shows how sentences are weighted during indexing.

the last term of the sentence. When it gets a match, it uses the terms position as the sentence position and stores it into the index. This flow is illustrated in figure 10.3. Here we see how sentence weighter attempts to match terms to an unmatched sentence, and store the sentence in index if there was a match; otherwise just ignore term.

10.2.4.1 Sentence extraction

Sentences have to be extracted from the abstract during indexing. We tried two approaches to *sentence segmentation*, Javas native BreakIterator and LingPipes SentenceModel; creating a common interface to be able to switch between for testing. The interface is as follows

```
public interface SentenceIterator {
        public void run();
        public void handleNext(String sentence, int start,
int end, int tokens);
}
```

where handleNext should be overridden, as it's called on every sentence within the document. The parameter *sentence* has the sentence string itself, *start* and *end* hold the character offset positions to the start and end of the string respectively and *tokens* holds the number of tokens in the sentence. Having start end end positions allows us to match sentences to internal term positions in Lucene, while token count is used to normalize sentence weights. Basing our work on LingPipes sentence segmentation, we can use our implementation iterate over sentences in a document as in this example:

```
SentenceIterator sitr = new LangPipeSentenceIterator(doc) {
        public void handleNext(String sentence, int start,
int end, int tokens) {
```

```
                System.out.println(sentence);
        }
};
sitr.run();
```

The example shows how we can iterate over the sentences in text string *doc*, printing them out one at the time.

After some testing, we ended up using LingPipes implementation, as it gave better results. Before sentence segmentation, the abstract needed to be tokenized. For this we used the *IndoEuropeanTokenizerFactory* which tokenizes Indo-European languages. The sentence model we used is *MedlineSentenceModel*, which is specifically implemented to operate over biomedical abstracts[3].

10.2.4.2 NER

LingPipe provides two different biomedical NER models, GENIA and GeneTag, both which we use and evaluate in our experiment. These models are used to automatically extract useful information within scientific texts.

GENIA works on the micro-biology domain, although it's goal is to be used within other domains as well. It contains biologic concepts, identifying both generic entities such as *dna sequence* and more classified concepts *I-L2 gene*. GENIA 3.0 corpus contains 93 293 biologically meaningful terms annotated by two domain experts[10]. GeneTag annotation is more restrictive, although having a wide definition of gene/protein entities, a biologic term must refer to a specific entity. This means that *tat dna sequence* will be identified, whilst *dna sequence* will not[9].

LingPipes provides these models as downloadable serialized Java Objects that are loaded during application run-time[4]. Both classes implement the same interface, the interface *Chunker*; thus both have the same API for identifying entities, allowing proper code reuse. Although the models classify entities into categories[5], we do not weight them different. We determine the biomedical relevance of the sentence solely by how many entities the NER models identify within the sentence.

10.3 Document retrieval

Document ranking is done during document retrieval. In our prototype we support the following two models

- *Vector Space Model (VSM)* - Lucenes default of-the-shelf similarity measure is built upon Vector Space Model[6]. We use this implementation in

[3]How MedlineSentenceModel works is described in its JavaDoc at http://alias-i.com/lingpipe/docs/api/com/aliasi/sentences/MedlineSentenceModel.html

[4]Both models are available for download at http://alias-i.com/lingpipe/web/models.html

[5]GENIA has 47 biologically relevant nominal categories[10].

[6]Implementation details are described in Lucene documentation at http://lucene.apache.org/java/2_4_1/api/org/apache/lucene/search/Similarity.html

our prototype.

- *Okapi BM25* - We've added a implementation of Okapi BM25 based on the work of BioTracer[7] to our prototype. Lucene does not include this ranking method by default.

In addition we have extended both ranking models to include our term boost implementation. Both have been added by subclassing each of these, and altering the scoring method. In order to be able to access the sentence index and read out term positions, some amount of boilerplate code was needed when extending these model extending. It however basically boiled down to overriding the score as shown in the following code

```
public float score() {
        return super.score() * (sentenceBoost() * K1);
}
```

Where *super.score()* calls the scoring method of the original model, *sentenceBoost()* calculates the term boost (implemented as shown in section 8.2) and K1 is our tuning constant, set to $K1 = 1.0$.

Implementing our extended models by overwriting the score method means that we need not alter the behavior or code base of the original models in any way. Our extended classes can can be used by Lucene in the same way as the original classes.

[7]BioTracer is the work of Ramampiaro and described in [7].

Chapter 11

Modifications to Lucene

This section describes modifications done to the Lucene library itself. Some modifications have been done to the Lucene library in cases where Lucene has not been extendable to the extent needed to implement our prototype.

11.1 Changes in FieldsWriter

Each document stored in lucene can contain several fields. We use a field to store title, another to store pmid, one to store the abstract and one to store document length (number of tokens). Internally, Lucene has support for storing several binary options as bitmasks. It it not possible to define custom options to store through the API. In order to be able to lookup sentence weights from the sentence index, we modified Lucene to add an additional option; option to indicate whether the field was sentence indexed. This option has been named *FIELD_HAS_SENTENCEINDEX*, as shown below in the list of options now supported in each field. When this option is set, an additional integer referencing the position within the sentence index is written to the index a long with the field data. When this option is set, the additional integer is also read during field lookup. Minor changes in several other classes within lucene had to be made for this change to be properly propagated upwards the API. The options now supported are as follows

```
static final byte FIELD_IS_TOKENIZED = 0x1;
static final byte FIELD_IS_BINARY = 0x2;
static final byte FIELD_IS_COMPRESSED = 0x4;
static final byte FIELD_HAS_SENTENCEINDEX = 0x8;
```

11.2 Changes in DocInverterPerField

DocInverterPerField is a class used for holding state for inverting all occurrences of a field in a document. The class doesn't do anything itself; rather it forwards

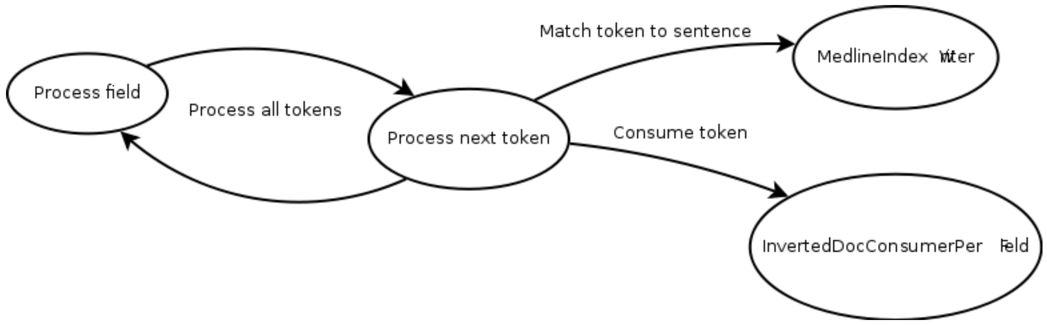


Figure 11.1: State diagram showing simplified view of the processFields method in the DocInverterPerField class.

the tokens produced by analysis to a consumer [1].

As sentence segmentation of the field is done after Lucene has tokenized the field, we do not know what internal positions the sentences should hold. To work around this, we have added an extra step when processing fields within this class; match the token to a sentence. This extra step is shown in figure 11.1. The internal positions for the sentences is done by matching token character offsets to sentence offsets in our sentence weighter, as has been described in section 10.2.4.

11.3 Altered class/method/field access

We have had the need to extend Lucene to more extent than the API has been able to provide. We have needed access to several classes, methods and variables within the lower Lucene libraries. Many of these have however been private within classes, or private lucene packages. In order to be able to make use of these classes, and extend up on those; still keeping our own code base out of Lucenes packages, several Lucene classes, methods and variables have needed to be changed to either protected or public. This is not a preferred solution, however we argue that it's better than mixing our codebase within Lucene's codebase. This way, our code stays within packages separate from Lucene.

[1]The class is a private class within the org.apache.lucene.index package, thus not included in the Java API docs. However, the source code can be viewed at http://svn.apache.org/repos/asf/lucene/java/tags/lucene_2_4_1/src/java/org/apache/lucene/index/DocInverterPerField.java

Part VI

Experiment and results

Chapter 12

Introduction

12.1 Document collection

In our approach we described a preference of what type of document collections would fit our evaluation (section 7.1) . We find that TREC Genomics Track test collection for 2004, which is made available for research purposes, suits our needs. The TREC Genomics Track was held for the purpose of evaluating information retrieval and related technologies within genomics domain. Their 2004 collection is a test collection with 4,5 million MEDLINE citations. Out of these 4,5 million citations 42,225 have been judged as relevant against 50 topics; each topic consisting of *title*, *need* and *context* field.

We have in our evaluation, rather than using the collection in its entirety, extracted the judged subset for indexing and retrieval purposes. Indexing the entire collection would be overly time consuming with the current implementation, and we argue that this subset, with its diversity to be large enough for this type of evaluation.

12.2 Method of evaluation

We use ranking measurements to evaluate the ranking models. Instead of rolling out our own tools for this, we've implemented our prototype in a way that makes its result output compatible with the tool *trec_eval*[1]. This gives us basis to compare our results with other research within the TREC community. The tool trec_eval is the standard tool used by the TREC community to evaluate ad hoc retrieval runs. When providing trec_eval with a result set, along with a standard set of judged results, the tool will give us several measures for retrieval performance, including MAP, R-precision and bpref. We put focus on TREC mean average precision, as it is the same approach as implemented by TREC.

[1]trec_eval is available for download at http://trec.nist.gov/trec_eval/index.html. Our evaluation is based on version 8.1

	Hardware and Java environment
CPU:	Intel(R) Pentium(R) D CPU 3.00GHz
Memory:	1x1GB + 1x2GB
Disk:	250GB S-ATA 7200RPM
Operating system:	Kubuntu GNU/Linux Jaunty 9.04
Java version	Sun Java(TM) 6 (1.6.0_13)
Java VM arguments	-Xms128m -Xmx512m[2]
Perl version	5.10.0

Table 12.1: Lists hardware environment, and Java VM environment.

MEDLINE documents provide in addition to article abstracts; title and several metadata attributes like Mesh headings and author names. We have only made use of the document abstracts during indexing and retrieval.

12.3 Test environment

The specifications for the platform used in our experiment is listed in table 12.1. We note that this is a desktop computer, running a full desktop environment in addition to our development platform Eclipse. Non the less, this platform has been suitable and testing and evaluation has not been limited by our test platform.

12.4 Experiment environment

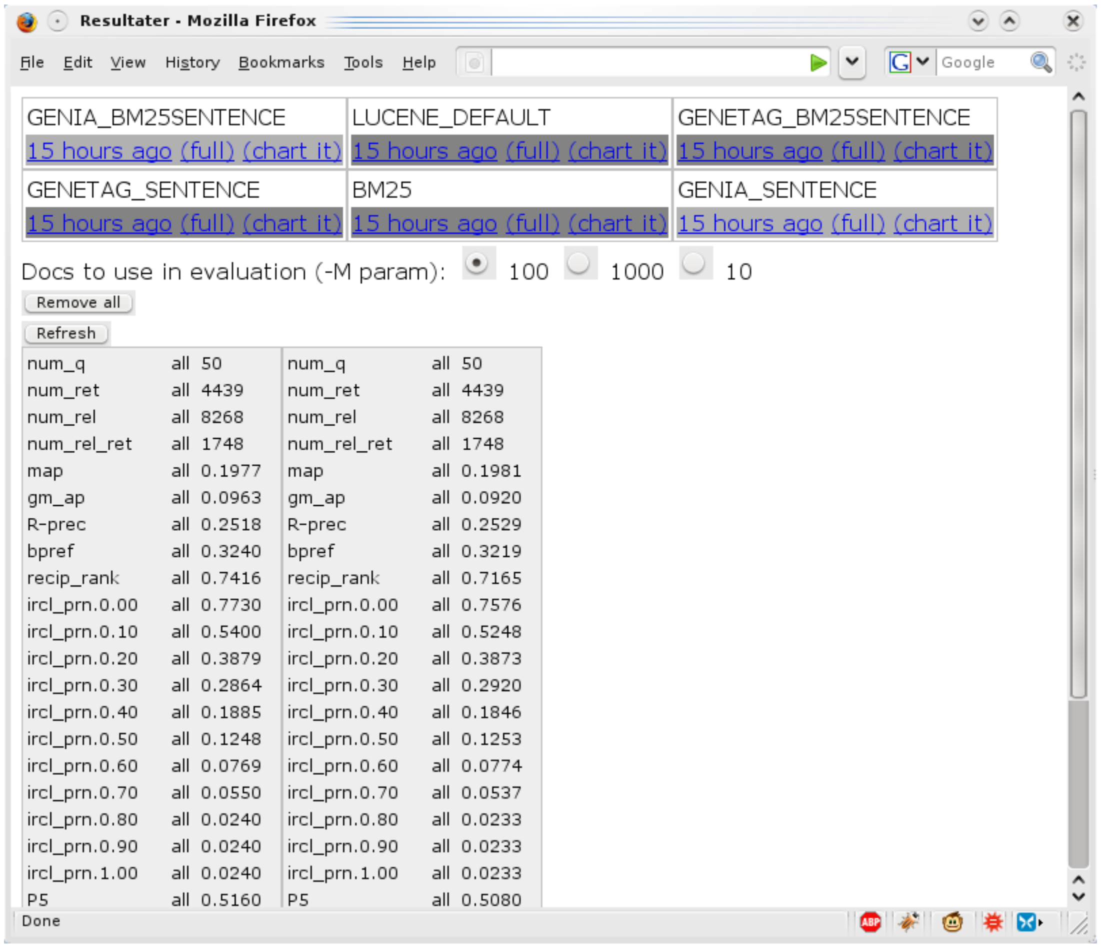
All evaluation is run within the development environment Eclipse[3] (version 3.2). The runs have been done within Eclipse as there were no constrains that required the prototype to be run in a separate environment. Evaluating this way allowed for quicker problem solving and rapid development.

In addition, a user interface was made to simplify the task of running *trec_eval* against the result sets the prototype produced, shown in figure 12.1. The user interface allows on-the-fly trec_eval execution of results, and displays them side-by-side for easy comparison. This interface is developed using Perl as the CGI-backend and HTML/Javascript for the interface itself. Perl has also been used for minor tasks, such as converting data for graph input and working on XML-documents.

12.5 Topics and judgments

TREC Genomics Track has defined 50 different topics to be searched for in their document collection. A total of 48,753 document have been judged against these topics, on average 975 per topic. A document can be judged not relevant,

[3]Eclipse is an integrated development environment, available at http://www.eclipse.org/


Resultater - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Google
Remove all
Refresh
Done

GENIA_BM25SENTENCE	LUCENE_DEFAULT	GENETAG_BM25SENTENCE
15 hours ago (full) (chart it)	15 hours ago (full) (chart it)	15 hours ago (full) (chart it)
GENETAG_SENTENCE	BM25	GENIA_SENTENCE
15 hours ago (full) (chart it)	15 hours ago (full) (chart it)	15 hours ago (full) (chart it)

Docs to use in evaluation (-M param): [✓ 100] 1000 10

num_q	all	50	num_q	all	50
num_ret	all	4439	num_ret	all	4439
num_rel	all	8268	num_rel	all	8268
num_rel_ret	all	1748	num_rel_ret	all	1748
map	all	0.1977	map	all	0.1981
gm_ap	all	0.0963	gm_ap	all	0.0920
R-prec	all	0.2518	R-prec	all	0.2529
bpref	all	0.3240	bpref	all	0.3219
recip_rank	all	0.7416	recip_rank	all	0.7165
ircl_prn.0.00	all	0.7730	ircl_prn.0.00	all	0.7576
ircl_prn.0.10	all	0.5400	ircl_prn.0.10	all	0.5248
ircl_prn.0.20	all	0.3879	ircl_prn.0.20	all	0.3873
ircl_prn.0.30	all	0.2864	ircl_prn.0.30	all	0.2920
ircl_prn.0.40	all	0.1885	ircl_prn.0.40	all	0.1846
ircl_prn.0.50	all	0.1248	ircl_prn.0.50	all	0.1253
ircl_prn.0.60	all	0.0769	ircl_prn.0.60	all	0.0774
ircl_prn.0.70	all	0.0550	ircl_prn.0.70	all	0.0537
ircl_prn.0.80	all	0.0240	ircl_prn.0.80	all	0.0233
ircl_prn.0.90	all	0.0240	ircl_prn.0.90	all	0.0233
ircl_prn.1.00	all	0.0240	ircl_prn.1.00	all	0.0233
P5	all	0.5160	P5	all	0.5080

Figure 12.1: Screenshot of the user interface developed to evaluate and compare results.

possibly relevant, or definitely relevant. A document that is either possible or definitely relevant will be considered as a relevant hit.

The evaluation topics are available in an XML-file containing title, what information is needed for this topic and a context for the search. Each topic is presented in XML as shown below.

```
<TOPIC>
        <ID>1</ID>
        <TITLE>Ferroportin-1 in humans</TITLE>
        <NEED>Find articles about Ferroportin-1, an iron
transporter, in humans.</NEED>
        <CONTEXT>Ferroportin1 (also known as SLC40A1; Ferroportin 1;
FPN1; HFE4; IREG1; Iron regulated gene 1; Iron-regulated transporter 1;
MTP1; SLC11A3; and Solute carrier family 11 (proton-coupled divalent
metal ion transporters), member 3) may play a role in iron transport.
        </CONTEXT>
</TOPIC>
```

User queries are not provided by TREC, it is up to whom evaluates the system to generate queries based on the information given for each topic, or other information he might find out about this topic. As for the example above, one might try to search for the synonyms given in addition to *Ferroportin-1*.

12.6 Query set

We have in our evaluation created a custom query set by hand for all 50 topics. We did not implement phrase query support, or support for other advanced query methods. Thus we have been limited to use Boolean term queries in our query set. The following is an example of one of our queries:

```
(oxidative OR oxidation) AND DNA AND (repair OR effect)
```

This query is for the topic "*DNA repair and oxidative stress*". Our full query set is available in Appendix A.

12.6.1 Other query sets

We attempted two other approaches at generating query sets. First was to parse the XML, extracting the title and using only the title as query for each topic. The second approach was to create a tool that used GENIA NER to extract biomedical terms from the topics and use as query. The flow of this tool is shown in figure 12.2, showing that it looked first for terms in the *need* field, and if non were found, used terms found in the *context* field. Both approaches were abandoned, as the queries produced provided too low recall to give proper measurements.

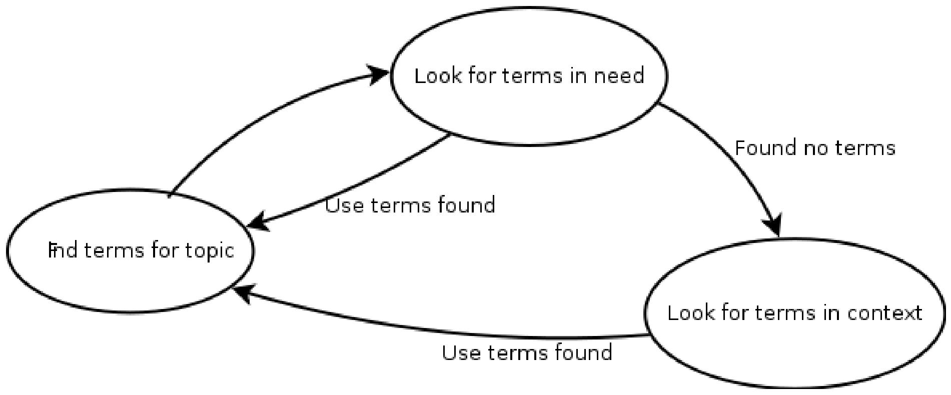


Figure 12.2: Automated query generation using GENIA NER.

Chapter 13

Results

13.1 Summary

Tables 13.1 and 13.2 provide an overview of our results. The best measure for each evaluation method has been emphasized. Table 13.1 shows measures when looking at only the first 100 results, while table 13.2 shows measures for first 1000 results. We note that it seems that no ranking method seems to outperform the others by any notable means.

13.2 Overall precision of the results

Our measurement is done on the first 100 results, as opposed to the default of 1000 in the *trec_ eval* tool. We find that using 100 results is more representative to user needs, as the user is likely to only look at the topmost results[14, 15]. We have in our evaluation focused on mean average precision (MAP), which has become the most standard measure of evaluation within the TREC community [5], and has as noted in section 3.3 been used before in TREC Genomics Track evaluation.

In table 13.1 shows that precision is mostly unaffected by extending the models with term-boosting. The table shows that extended BM25 model gives

	MAP	R-precision	P(5)	P(100)
VSM (Lucene)	0.1983	0.2524	0.5200	0.3500
Okapi BM25	0.1977	0.2518	0.5160	0.3496
Extended VSM (GeneTag)	0.1962	0.2490	0.5400	*0.3566*
Extended BM25 (GeneTag)	0.1983	0.2545	*0.5440*	0.3478
Extended VSM (GENIA)	0.1981	0.2529	0.5080	0.3496
Extended BM25 (GENIA)	*0.1992*	*0.2564*	0.5320	0.3480

Table 13.1: Evaluation measures at 100 first results.

	MAP	R-precision	P(5)	P(100)
VSM (Lucene)	*0.3057*	0.3452	0.5200	0.3500
Okapi BM25	0.3055	0.3490	0.5160	0.3496
Extended VSM (GeneTag)	0.3043	0.3437	0.5400	*0.3566*
Extended BM25 (GeneTag)	0.3045	*0.3512*	*0.5440*	0.3478
Extended VSM (GENIA)	0.3051	0.3470	0.5080	0.3496
Extended BM25 (GENIA)	0.3045	0.3505	0.5320	0.3480

Table 13.2: Evaluation measures at 1000 first results.

slightly better MAP measure, while extended VSM gives slightly worse.

Figures 13.1 and 13.2 show MAP measure for 49 of the topics[1]. We first compare Okapi BM25 and its extended versions based on GeneTag and GENIA NER weighting in figure 13.1, and Lucenes VSM with its extended versions in figure 13.2. Common for both are that no query is largely affected by term boosting.

13.2.1 Closer look at queries

In BM25 we see one particular topic, topic 15, that gains a slight precision boost when using the extended models. The measures for these results were 0.0664 with regular BM25, while it was 0.1644 and 0.1277 for GENIA and GeneTag respectively. The query for this topic was:

```
(ATPase OR ATPases) AND (apoptosis OR (cell death))
```

ATPase being a class of enzymes likely to be recognized by both GeneTag and GENIA, while apoptosis being the process of programmed cell death is likely to be recognized by Genia.

In VSM one topic gained precision lost with GeneTag, topic 28. The measures for these results were 0.3428 for regular VSM, 0.3463 using Genia and 0.2267 using GeneTag. The query for this topic was:

```
(autophagy OR (gene autophagic)) AND apoptosis
AND (proteases OR morphological)
```

Autophagy being a cellular process and proteases being an enzyme.

Both incidents seem isolated and neither are found on both of the ranking models BM25 and VSM.

13.3 Precision of top results

For measuring the precision of the top results, we use precision-at-k measure. Precision at k is a measure for how relevant the first k hits are to the query. This

[1]One of the topics, topic 18 was left out. The topic had measure of 1.0 for all ranking models, and was left out of the graph to get more detailed y-range.

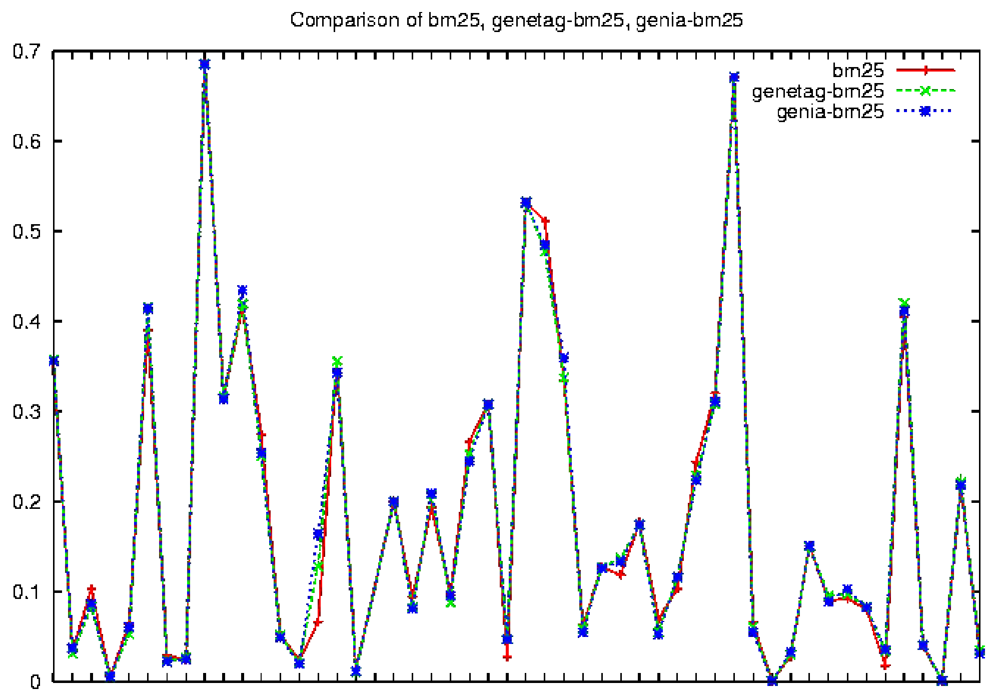


Figure 13.1: Comparison of MAP measure for BM25 and BM25 extended with GeneTag and GENIA

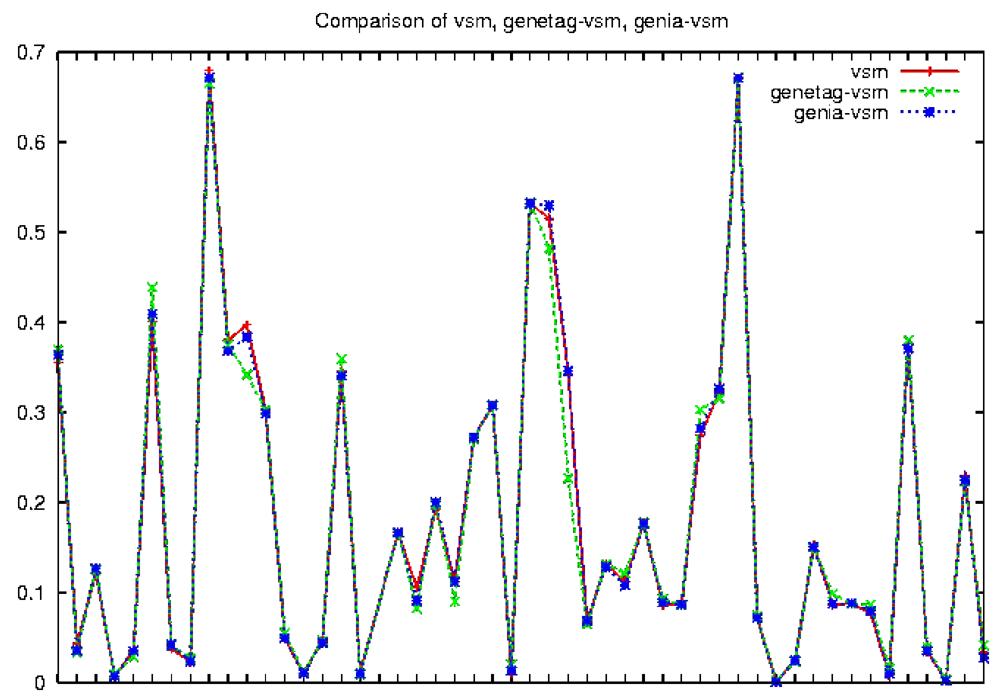


Figure 13.2: Comparison of MAP measure for VSM and VSM extended with GeneTag and GENIA

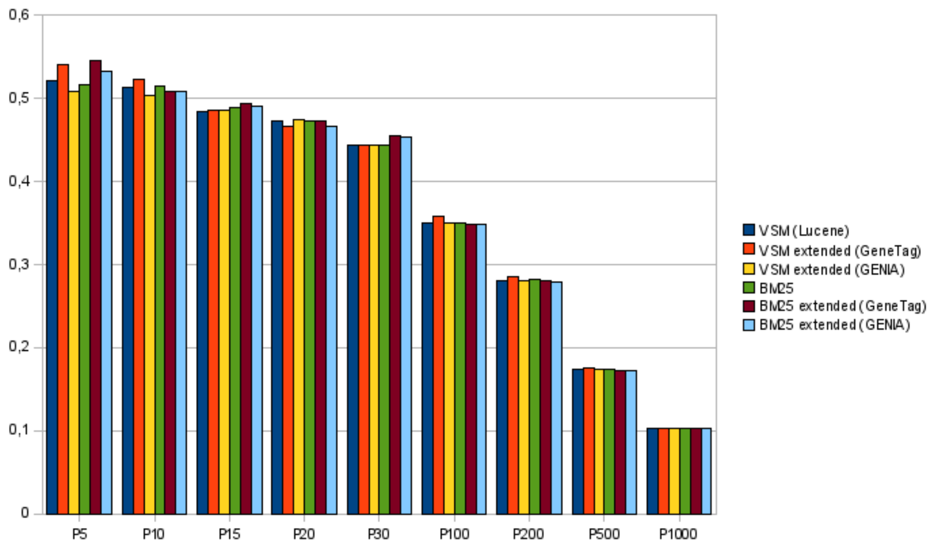


Figure 13.3: Precision-at-k for all of the ranking methods.

MAP measures	Baseline	$k_1 = 0.5$	$k_1 = 1.0$	$k_1 = 2.0$
Lucene VSM (GENIA)	*0.1983*	0.1981	0.1981	0.1981
Okapi BM25 (GENIA)	0.1977	0.1991	0.1992	*0.2010*
Lucene VSM (GeneTag)	0.1983	*0.1962*	*0.1962*	*0.1962*
Okapi BM25 (GeneTag)	0.1977	0.1981	*0.1983*	*0.1983*

Table 13.3: MAP measures when tuning term-boosting

measure is useful when attempting to increase user satisfaction, as users seldom look past the top results[14, 15]. This is a simple recall-precision measure that shows how many of the recalled documents are relevant. When 5 out of 10 first documents are relevant, the measure is $5/10 = 0.5$.

Figure 13.3 shows precision at different recall levels for all models. As shown on the figure, the extended models using term boosting with GeneTag have a slightly better precision at the first 5 documents. The precision of extended models versus original models seem mostly unaffected otherwise.

13.4 Tuning boost factor influence

Table 13.3 shows the result of tuning the term boost factor $k_1$as defined in section 7.6. These measures show that tuning this constant has little to no effect on the overall result, most promising being boost factor $k_1 = 2.0$ on extended Okapi BM25 using GENIA NER. The same factor had otherwise no effect on any other models compared to $k_1 = 1.0$.

13.5 Interpretation of results

Evaluation shows that document level term-boosting has little effect on the results. The effect it does have can be both positive and negative, and where it has effect seems isolated and not necessarily reproducible in the other model. The extended BM25 models give slightly better precision than BM25 baseline, however the extended VSM models have the opposite effect. Attempting to tune the results using tune boost multiplication factor has little to no effect.

Part VII

Discussion and conclusion

Chapter 14

Discussion

14.1 Context weighting

As noted, our context for a term is defined as the sentence which it is found within. This definition, while correct, may be extendable to a broader perspective. A properly structured research document will in most cases have at least a title, abstract, main content and a conclusion. Boosting specific document parts (section 5.3) has been shown to yield better ranking. This leads us to believe that extending our context from just the sentence the term is found within, to also factor in which part of the document it is found within, may yield better term boost weights.

Additionally, the sentence weighting may be improved. We weight a sentence by the number of entities the chosen name entity recognition model finds. This does not factor in how it classifies the entity. GENIA alone can classify entities into 47 biologically relevant nominal categories, from as vague as “other”, to as specific as “RNA substructure”. Having different weights based on classification may give more presentable sentence weighting. One might also try to combine GENIA and GeneTag.

14.2 Query sets

Our prototype implements Boolean queries, and is limited to that. A proper retrieval system may additionally support phrase queries, as well as other advanced querying techniques such as wildcard support. Not supporting advanced querying may limit the precision of our query set.

There has not been a notable difference in retrieval performance between the implementation of vector space model we use and Okapi BM25. Others have had BM25 give improved results[16, 7]. We suggest our measures differ as a result of the queries we use, which are Boolean in nature and generally suffer from low recall. This may also be because we do not search within documents topics, but only the abstract. Implementing additional query support and refactoring the queries to make use of additional query expressions is likely to improve performance; we do have however not have basis to predict whether the extensions made to the ranking models will positively affect the results.

14.3 Evaluation

In our evaluation we have made the assumption that our queries represent real user queries. We have also used a subset of 42.225 which we recognize, despite its diversity in topics, that it is a limited set for proper evaluation. In addition, the document set is a closed test collection. Thus the evaluation may not necessarily represent real life situations. A more proper evaluation may be one done on a larger document collection, involving human experts.

Chapter 15

Conclusion

We have in this thesis defined an approach to give all terms in a document a context value based on the domain specific data of this context. We have then created a means of using this value during the ranking process of document retrieval, by combining it with existing ranking models. We have proposed that when the queried term is located in strong domain context within a document, that this document is more relevant than in documents where the term is located in a sentence composed merely of natural English. Based on this proposal, we have attempted to increase precision of the retrieved documents by giving each term a document-level term boosting.

We have in this thesis, in means to test our proposal, implemented a full search retrieval system prototype that uses our defined approach. The implementation has made use two different named entity recognition models to create context weights, and the produced weights have been combined with two widely used ranking models that would otherwise rank terms in a document merely in a bag-of-words manner. We have then evaluated our prototype on a document collection that has been used for research on biomedical information retrieval, using ranking measurements well established within the same field; giving basis for comparison.

15.1 Final conclusion

Given the experience and data gathered in our evaluation of this prototype, we have determined that this type of document boosting has little effect on overall retrieval precision. The effect it has can affect precision both positively, and negatively. We see however that when combined with the ranking model *Okapi BM25,* the document boosting effect has more often than not contributed positively to the results, as seen in tables 13.1 and 13.3. As seen in the same tables, this is not the case when combined with Lucenes implementation of *vector space model.*

The positive effects on precision, where positive, are still small. The effect may or may not turn out to be positive on a different document collection, or different query set. In addition our approach requires significant additional processing for text analysis and increases complexity of a search retrieval system. Thus we conclude this approach, as it is now, to not necessarily be good means of increasing precision in biomedical retrieval systems.

15.2 Further work

We find the boosting effect our work has shown, although small, may be worth further research. One way of improving this effect may be improving context weighting, by both extending the context and more intelligent weighting schemes, as discussed in section 14.1.

Combining term boosting with the increased recall MeSH query expansion gives (described in section 5.1) may be worth research. Query expansion is considered a recall-favoring technique, and sometimes harmful to system performance[8]. However combining it with term-boosting may yield both better precision and recall. Better recall due to the query expansion itself, but also better precision because the extended terms are likely terms that are recognized by the NER models used in our context weighting. Thus giving additional biomedical terms to boost the relevant documents with.

Bibliography

[1] Yilmaz, E. and Aslam, J. A. 2008. Estimating average precision when judgments are incomplete. Knowledge and Information Systems 16, 2(July), 173-211

[2] Hersh, W.R., Bhupatiraju, R. T., Ross, L., Roberts, P., Choen, A. M., and Kraemer, D.F. 2006. Enhancing access to the bibliome: the trec 2004 genomics track. Journal of Biomedical Discover and Collaboration2006 1, 3 (March), pp. 10

[3] Ian H. Witten, Alistar Moffat, Timothy C. Bell, Managing Gigabytes, Morgan Kaufmann 1999

[4] Robertson, S.E. and Jones, K.S. 1994. Simple proven approaches to text retrieval. Tech. Rep. 356, University of Cambridge.

[5] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze 2008, Introduction to Information Retrieval

[6] Chris Buckley, Ellen M. Voorhees, Retrieval Evaluation with Incomplete Information 2004

[7] Heri Ramampiaro, Retrieving BioMedical Information with BioTracer: Challenges and Possibilities, Norwegian University of Science and Technology.

[8] Zhiyong Lu, Won Kim, W. John Wilbur, Evaluation of query expansion using MeSH in PubMed, 2008

[9] Lorraine Tanabe, Natalie Xie, Lynne H Thom, Wayne Matten, W John Wilbur, GENETAG: a tagged corpus for gene/protein named entity recognition, 2005

[10] J.-D. Kim, T. Ohta, Y. Tateisi, J. Tsujii, GENIA corpus - a semantically annotated corpus for bio-textmining, 2003

[11] Salton, G., Wong, A., and Yang, C. (1975). A vec- tor space model for automatic indexing.

[12] Mikheev, Andrei. 2002. Periods, Capitalized Words, etc. Computational Linguistics 28(3):289-318.

[13] A survey of current work in biomedical text mining, Aaron M. Cohen, William R. Hersh , 2004

[14] Spink, A., Wolfram, D., Jansen B.J., and Saracevic, T. Searching the web: the public and their queries. Journal of the American Society of Information Science and Technology, 52(3), 2001, 226-234.

[15] Spink, A., Jansen B.J., Wolfram, D., and Saracevic, T. From e-sex to e-commerce: web search changes. IEEE Computer, 35(3), 2002, 107-109.

[16] Robertson, S.E. et al. 'Okapi at TREC-3', 1995

[17] Muller, H.-M., Kenny, E. E., and Sternberg, P. W. 2004. Textpresso: an ontology-based information retrieval and extraction system for biological literature.

[18] Krauthammer, M. and Nenadic, G. 2004. Term identification in the biomedical literature. Journal of Biomedical Informatics 37, 6, 512-526

[19] Chen, L., Liu, H., and Friedman, C. 2005. Gene name ambiguity of eukaryotic nomenclatures. Bioinformatics 21, 2, 248-256

Appendix A

Custom queries

The following are our custom search queries for each of the 50 topics provided by TREC 2004 Genomics Track.

1. iron AND (Ferroportin1 OR (Ferroportin AND 1) OR SLC40A1 OR FPN1 OR HFE4 OR IREG1 OR (Iron AND regulated AND gene AND 1) OR MTP1 OR SLC11A3)
2. (transgenic OR (copy AND gene)) AND (mice OR mouse OR murine)
3. kidney AND (mice OR mouse OR murine) AND (develop OR developing OR development)
4. kidney AND ((gene OR genes) OR (expression AND (profile OR profile))) AND (mice OR mouse OR murine)
5. (isolate OR isolating OR fractionation OR purify) AND cell
6. FancD2 OR (Fanconi AND anemia) OR (group D2) OR (type AND 4 AND fanconi AND pancytopenia)
7. (oxidative OR oxidation) AND DNA AND (repair OR effect)
8. ((oxidative AND (disease OR diseases OR carcinogenesis)) OR (DNA AND repair)) AND (cancer OR cancers OR carcinoma)
9. (muty OR hmyh) AND -myoglobin
10. (NEIL1 OR (nei AND endonuclease)) AND (DNA OR repair)
11. hairless mice carcinogenesis skin OR UV
12. (gene OR genes) AND smad4
13. (TGFB OR (transforming growth factor beta)) AND (homeostasis OR angiogenesis)

14. (TGFB OR (transforming growth factor beta)) AND ((head and neck squamous cell) OR HNSCC) AND (cancer OR cancers OR carcinoma)
15. (ATPase OR ATPases) AND (apoptosis OR (cell death))
16. (AAA OR (ATPases associated activities)) AND (protein OR proteins OR lipids OR DNA)
17. (DO1 OR (p53 AND (antibody OR anti))) AND binding
18. (Gis4 OR YML006C)
19. (GAL1 OR SUC1) AND (repressors OR reprosessor OR activators OR activator)
20. (covalent OR attachment OR covalence OR substrate) AND (ubiquitin OR ubiquitously OR ubiquitylation OR ubiquitination)
21. (p63 OR TP63) OR (TP73 OR p73) DNA
22. p53 DNA (break OR damage)
23. Saccharomyces OR cerevisiae (protein OR proteins) (ubiquitin OR proteolytic OR pathway)
24. (mice OR mouse OR murine) AND (peptidoglycan OR PGRP or PGRPs)
25. scleroderma OR (autoimmune disease (genes OR gene))
26. (BUB2 OR BFA1) AND (cytokinesis OR yeast)
27. (autophagy OR (gene autophagic)) AND apoptosis
28. (autophagy OR (gene autophagic)) AND apoptosis AND (proteases OR morphological)
29. (gyrA OR (DNA gyrase)) AND (mutation OR mutations OR alteration)
30. Nkx OR Sax
31. TOR OR (Target AND Of AND Rapamycin) OR FRAP1 OR (FK506 AND associated AND protein)
32. Xenograft AND (tumorogenesis OR cancer OR cancers OR carcinoma)
33. (Histoplasmosis OR (blood borne hogen)) AND (mice OR mouse OR murine)
34. Cryptococcus AND (gene OR genes OR genome)
35. WD40 OR (protein AND repeats)
36. (RAB3A OR (RAS oncogene family)) AND ((synaptic plasticity) OR synapse)

37. PAM OR (peptide AND amidating AND enzyme) OR (peptidylglycine AND amidating)
38. (genetic AND loci) OR Stroke OR E4
39. Hypertension ((risk OR danger) AND stroke)
40. (antigen OR antigens) AND (epithelial OR epithelium)
41. (mutation OR mutations OR altered) AND ((Cystic AND Fibrosis) OR CF OR mucovoidosis OR muscoviscidosis)
42. (chromosome AND translocations) OR (chromosomal (rearrangement OR rearrangements))
43. (sleeping AND beauty) OR SB
44. (nerve AND growth AND factor) OR NFG
45. (MWH1 OR (mental health wellness)) OR (mental disorder gene)
46. RSK2 OR (ribosomal protein kinase)
47. BCL OR BCL2 OR (BCL AND 2)
48. (UNC OR BGS OR homologues) AND (gene OR genes)
49. (Glyphosate OR glycine) AND (tolerance OR tolerant OR immune)
50. low temperature ((E AND coli) OR escherichia)