Flávio Nuno Fernandes Martins

Licenciado em Engenharia Informática

Improving search engines with open Web-based SKOS vocabularies

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador: Prof. Doutor João Magalhães, Prof. Auxiliar, FCT/UNL

Júri:

Presidente: Prof. Doutor João Carlos Gomes Moura Pires

Arguente: Prof. Doutor Bruno Emanuel da Graça Martins

Vogal: Prof. Doutor João Miguel da Costa Magalhães

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2012

Improving search engines with open Web-based SKOS vocabularies

Copyright © Flávio Nuno Fernandes Martins.
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Acknowledgements

First, I would like to express my sincere appreciation to everyone that contributed directly or indirectly to the success of this project. As is often the case, this master's thesis is not an individual enterprise and is shaped by individuals that I was fortuned enough to cross paths with during my formation years and later academic endeavors.

I would like to express my gratitude to my advisor Professor João Magalhães for introducing me to an area of research that turned out to encompass everything that I am interested in and for his continuous support of my MSc studies and research.

I would also like to thank Bernhard Haslhofer for his knowledge, for his reviews, for developing the first version of Lucene-SKOS and for making it open source.

Thanks to Fundação para a Ciência e Tecnologia for financing my research, during the preparation of this thesis, through the ImTV research project, in the context of the UTAustin-Portugal collaboration (UTA-Est/MAI/0010/2009). In addition, I am thankful to Quidgest for allowing myself to mature this thesis, during my internship with them, and for providing financial support, through the QSearch project (FCT/Quidgest).

I would like to thank my family, especially my mother, Maria Goreti, for encouraging me to be a hard working student and pursue graduate studies. I want to thank my siblings for fast tracking me constantly as the next big thing and for their support.

I extend my thanks to Ana Figueiras, for taking this journey with me. I am grateful to her for giving me lots of support and for listening carefully to intricate expositions of computer science subjects that I was often over enthusiastic to share with her.

Finally, I am grateful for the good times in Universidade Nova de Lisboa and I would like to extend my thanks to my Professors and colleagues. To all of you, thank you for the opportunity to learn and contribute to scientific endeavors.

Abstract

The volume of digital information is increasingly larger and even though organizations are making more of this information available, without the proper tools users have great difficulties in retrieving documents about subjects of interest. Good information retrieval mechanisms are crucial for answering user information needs.

Nowadays, search engines are unavoidable - they are an essential feature in document management systems. However, achieving good relevancy is a difficult problem particularly when dealing with specific technical domains where vocabulary mismatch problems can be prejudicial. Numerous research works found that exploiting the lexical or semantic relations of terms in a collection attenuates this problem.

In this dissertation, we aim to improve search results and user experience by investigating the use of potentially connected Web vocabularies in information retrieval engines. In the context of open Web-based SKOS vocabularies we propose a query expansion framework implemented in a widely used IR system (Lucene/Solr), and evaluated using standard IR evaluation datasets.

The components described in this thesis were applied in the development of a new search system that was integrated with a rapid applications development tool in the context of an internship at Quidgest S.A.

Resumo

O volume de informação digital é cada vez maior e embora as organizações estejam a disponibilizar cada vez mais informação, sem as ferramentas adequadas os utilizadores têm grandes dificuldades na recuperação de documentos sobre assuntos de interesse. Bons mecanismos de recuperação de informação são cruciais para responder às necessidades de informação do utilizador.

Hoje em dia os motores de pesquisa são inevitáveis, eles são um recurso essencial em sistemas de gestão documental. No entanto, conseguir boa relevância é um problema difícil, particularmente quando se trata de domínios técnicos específicos, onde o problema de discordância de vocabulário pode ser prejudicial. Numerosos trabalhos de pesquisa descobriram que explorar as relações lexicais ou semânticas dos termos numa coleção atenua este problema.

Neste trabalho, pretendemos melhorar os resultados de pesquisa e experiência do utilizador, investigando o uso de vocabulários da Web potencialmente ligados nos motores de pesquisa da informação. No contexto de vocabulários Web abertos baseados em SKOS propomos um sistema de expansão de consulta implementado num sistema IR amplamente utilizado (Lucene/Solr) e avaliado utilizando conjuntos de dados de avaliação padrão de IR.

Os componentes descritos neste trabalho foram aplicados no desenvolvimento de um novo sistema de pesquisa que foi integrado com uma ferramenta de desenvolvimento rápido de aplicações no contexto de um estágio na Quidgest S.A.

Contents

Figures

Tables

Acronyms

AP	Average Precision
API	Application Programming Interface
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DIH	Data Import Handler
DSK	Domain-specific Knowledge
DYM	Did You Mean
EJB	Enterprise JavaBeans
FTP	File Transfer Protocol
GMAP	Geometric Mean Average Precision
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IDF	Inverse Document Frequency
IMAP	Internet Message Access Protocol
IQE	Interactive Query Expansion
IR	Information Retrieval
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
KOS	Knowledge Organization System
LETOR	LEarning TO Rank
MAP	Mean Average Precision
MeSH	Medical Subject Headings
MFC	Microsoft Foundation Classes
ML	Machine Learning
MVC	Model-View-Controller
nDCG	Normalized Discount Cumulative Gain
NLM	National Library of Medicine (US)

NoSQL	Not Only Structured Query Language
OCR	Optical Character Recognition
PDF	Portable Document Format
PLSQL	Procedural Structured Query Language
RAD	Rapid Application Development
RDBMS	Relational Database Management System
RDF	Resource Description Framework
REST	Representational State Transfer
RoR	Ruby on Rails
SAPO	Serviço de Apontadores Portugueses
SFTP	Secured File Transfer Protocol
SIGIR	Special Interest Group on Information Retrieval (ACM)
SKOS	Simple Knowledge Organization System
SPARQL	Simple Protocol and RDF Query Language
SQL	Structured Query Language
TF	Term Frequency
TREC	Text REtrieval Conference
UKAT	UK Archival Thesaurus
UMLS	Unified Medical Language System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VSM	Vector Space Model
WCF	Windows Communication Foundation
XML	Extensible Markup Language
XSLT	XML Stylesheet Language for Transformations
ZIP	Zip file format

1

Introduction

Information management was initially limited to the maintenance of physical information, such as paper-based files. The proliferation of information technology and the ever increasing amount of digital data that is produced and exchanged nowadays steered information management processes to embed digital data storage and maintenance tasks. This boosted the development of dedicated systems for the management of electronic documents.

A document management system is a computer system that facilitates the collection and management of digital data and provides the means for storing the electronic documents in an efficient structured way. These systems often provide document history tracking, management of document metadata, security services and support for collaborative work. Furthermore, the inclusion of advanced indexing and retrieval capabilities into document management systems has become a powerful resource for many organizations. Document retrieval is greatly improved by using a full-text search engine and metadata extractors to index the document's content as well as the metadata bundled with the document. Even traditional paper-based documents are being scanned and introduced in modern document management systems, as-is in any image file format or PDF and alternatively processed through an OCR engine to recreate the textual content in a machine-readable format.

In summary, a document is any electronic data file, that is almost invariably semi-structured, and that contains some kind of textual content and textual metadata associated. The state-of-the-art document retrieval systems collect and extract data elements and features, in a given collection of documents, to improve document retrieval significantly.

1.1 Context and motivation

Due to achievements such as the World Wide Web and to the connected nature of virtually every modern organization, document management systems, the electronic counterparts of the archaic paper-based information management systems, currently have to deal with an overwhelming flow of information. A clever structuring and organization of files and folders might have sufficed in the past, but nowadays the quantity of data maintained in the document management systems of a large number of organizations and businesses is getting massive. In addition, rapidly calling up valuable documents is, of course, a kind of competitive advantage and therefore efficient information retrieval has become an essential tool for information owners and keepers in exploiting the value of their assets.

Higher speed of search, the time it takes for the results to come up, is of course a welcomed trait in a search engine. However, the real usefulness of a search engine is estimated better by its ability to find the right piece of information, which turns out to be a larger, harder problem. With a very large number of documents, a poorly ranked result set is not useful, even if these are returned instantly, because the user must then sift through a sizable amount of results in order to find the precise document. Search engines for the Web are a good example of this phenomenon, because any given query can match millions of web pages. To solve this problem, search engines go to great lengths, employing advanced ranking methods to return the most valuable results first. Google for example analyses the hyperlink structure of web documents, and with the PageRank algorithm, attributes a value of importance to pages and its domains based on the number of incoming links and based on the rank of each incoming link as calculated by the PageRank algorithm, in a recursive fashion. This allows Google to weight in PageRank, along with other relevance factors and to make a better determination of the relevance of the each search result. In document management systems however, there is no link structure like on the Web, so we cannot use it as a factor to determine document's relevancy, thus making it harder to rank.

To alleviate this fact, document retrieval systems and enterprise search engines use other relevance factors, albeit weaker, such as document freshness and popularity for ranking results. To help the user further, to find what they need in a large collection, modern IR systems explore domain specific knowledge to improve retrieval effectiveness. This knowledge can be encoded by a domain expert that creates hierarchical and organized thesauri to assist user searches.

1.2 Objective

This thesis aims to explore the possibilities of incorporating open Web-based Simple Knowledge Organization System vocabularies (SKOS) in document retrieval systems and improving retrieval performance with this data in the context of document management systems. The effect anticipated is an overall improvement of search results due to the increased document recall and ranking performance.

1.3 Proposed framework and contributions

In this framework (see Figure 1), the search system imports its data from an application's relational database. Results are then indexed and retrieved using improved retrieval functions. Queries are expanded using available SKOS thesauri and an interactive query expansion aware search interface guides users in the query formulation.

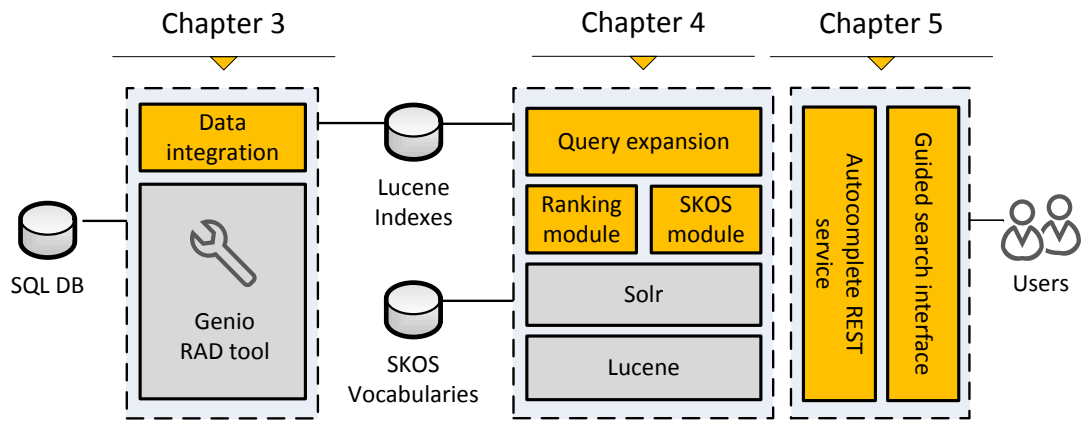


Figure 1. Proposed Framework diagram.

The main contributions of this thesis are:

- Integration with the automatic code generation tool Genio from Quidgest – implementation of a custom search engine, a management interface and an automatic deployment and data import configuration process. Implementation of state-of-the-art retrieval functions in a widely used production IR system.

- Domain-specific knowledge-based query expansion – development and evaluation of a query expansion technique in the context of open Web-based SKOS vocabularies, implemented in a widely used IR system Lucene, and evaluated using standard IR evaluation datasets.

- Search interface with guided query expansion – development and analysis of an interactive query expansion interface implemented directly in the search input box, using a REST service. This interface allows users to be guided in the query formulation phase, by presenting them with alternative closely related concepts.

1.4 Organization

The structure of this document is as follows:

- **Chapter 2 – Related work:** the main building blocks of search engines are presented with special emphasis to ranking, probabilistic retrieval models and learning-to-rank approaches. Query expansion techniques for improved recall and relevancy are presented; the architecture and interfaces of Apache Solr are detailed.

- **Chapter 3 – Data integration:** this chapter describes the integration of the Solr search engine functionality with Quidgest's Genio system and the extensions built around it. This includes new retrieval function implementations and management interfaces.

- **Chapter 4 – Domain-specific knowledge-based query expansion:** this chapter presents how to apply this technique in the context of open Web-based SKOS vocabularies and presents evaluation results and discussion using standard IR evaluation datasets.

- **Chapter 5 – Search interface with guided expansion:** this chapter presents a search interface implementation for guided query expansion using SKOS vocabularies and concept expansion.

- **Chapter 6 – Conclusions:** this chapter presents the conclusions of this thesis.

2

Related work

Several advances in Information Retrieval (IR) have allowed a specific type of IR system, the so-called search engine, to become the primary and preferred form of information access today. A search engine is characterized by facilitating the search in a large collection of semi-structured documents, in a fast and effective fashion. Effectiveness is related to the relevance quality of the results returned in regards to the given query. In Section 2.1, we will show the typical inner workings of search engines.

2.1 Ranking with heuristics

Search engines are a very specific kind of IR system, since they have a functional requirement to return the list of the most relevant documents to a given query in order of importance. Therefore, ranking techniques are very much central to search, while in filtering, classification, clustering and other Information Retrieval fields [14] these are not as important.

Another very important trait of search engines is the large amount of data they have to process, which makes simple filtering approaches or raw data not feasible computationally. Therefore, search engines employ pre-processing indexing techniques to cope with query processing demands in a large document collection. The conventional technique used in search engines is an inverted index and its simplest form can be augmented to build the foundations needed for rank ordering the results and to allow matching operations that are more flexible.

2.1.1 Search engine components

A search engine or document retrieval system maintains an index to support the query functionality. Indexing usually encompasses four main phases:

1) Collect the documents to index
2) Tokenize the text
3) Linguistic pre-processing of tokens
4) Index the documents for each term

In the document collection phase, we gather the documents to index. These documents can be collected in a variety of ways. They can be located in a repository, referenced by a database or, in the case of a Web search engine, by using a Web crawler.

Each document's content is then transformed into a histogram of words or bag-of-words representation through a tokenization and normalization process. This process puts the documents content through an array of text processing steps: charset conversion; blank space removal; markup filtering (often used for HTML content tag cleaning); character replacing or filtering (used for example to replace Latin accentuated characters). Finally, the text goes through a tokenizer that splits the text up into a series of tokens, which will be the final terms after some linguistic processing.

Linguistic pre-processing captures certain text patterns and uses heuristics to simplify the documents histogram of words. These processes enclosed are language specific. Therefore, the document's language is detected and this information is passed to the remaining linguistic processing steps. One of the simpler heuristic pre-processing is stop-word removal, by which we discard tokens that are too common, and thus would not yield sufficient value if indexed. In the English language, the stop-words list might contain common words such as "a" and "the" which are found in the majority of documents. Another usual process is stemming, which consists in reducing each inflected or derived word to their stem or root such as the morphological root of the word. A stemmer would for instance reduce "fishing", "fished" and "fish" simply to the root word "fish". A search engine benefits greatly from linguistic pre-processing as it reduces the number of distinct words to index and has the effect of reducing the index size. It also improves search effectiveness by better handling queries with different word inflections and opens the way to a type of query broadening called conflation in which words with the same stem are treated as synonyms by the search engine.

When reaching this last phase there is one histogram of words representation for each document which results from the earlier processing. With this information the

actual index is built. The index must support quick evaluation of search queries to return the list of documents containing each word in the query fast. To that effect search engines incorporate an inverted index, which is a dictionary of all the words and allows direct access to the list of documents containing each word.

This index is said to be a *Boolean index*, because it is capable of determining which documents match a query, but it cannot return a ranked list of the matched documents. However this type of index is usually not very useful, since the list of documents matched is invariably much larger than the number of results we can expect a user to sift through, even more prominently so in the case of a Web search engine.

2.1.2 Vector space model

The vector space model (see Figure 2) is the representation of a collection of documents as vectors in a common vector space. In search engines, this is achieved by considering each document as a vector, with a component for each term in the dictionary, together with a weight for each component calculated according to the ranking model.

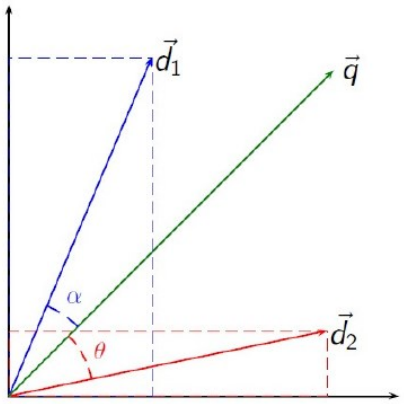


Figure 2. Vector space model.

In this model, the relevance degree can be more accurately calculated when both queries and documents are taken as vectors in the same Euclidean space, such that the inner product between the query and each document can be used to calculate the similarity. To calculate the relevance of a document in regards to a query, the *cosine similarity* between the two is used. Given a query q and a document d, the cosine similarity can be determined, as follows:

$$cos(q,d) = \frac{q \,.\, d}{\|q\| \,.\, \|d\|} \quad (1)$$

2.1.3 Scoring documents

We have seen how the Boolean model is supported in a search engine using an inverted index, and how this model cannot predict the degree of relevance of the documents. Being able to make this prediction and rank-order the results is vital for a search engine, and we will present in this section how the system described can be extended to rank-order the results. This means that the search engine will be able to compute a score for a query-document pair and return the list of results accordingly.

Until this point, the document structure is not explored by the search engine, which treats documents as a stream of terms instead. However, most digital documents are semi-structured, their content is often laid out in sections or zones and there is usually associated metadata. For instance, in the case of web page documents the content is laid out by zones, like the header, the body and footer. The publishing date is usually also an available piece of associated metadata. In a search engine, these are encoded into either fields or zones. A field, such as a publication date, is characterized by having a finite domain, and of possibly sortable nature, that is why there is a parametric index for each field where B-trees are often used. In contrast, a zone contains arbitrary free-form text, and a simple solution is to store each zone in its own inverted index. A more efficient way to index using zones is to have only one inverted index instead and to encode the zone in the postings, alongside the document id. This encoding is useful not only because the indexes size is reduced but also because it allows the system to weight zones. In weighted zone scoring, a weight g is set for each zone such that the weights of all zones sums up to one. The score of a document d in regards to a query q is the sum of the weights of all the zones for which a query word matches. Setting the title zone to be given more weight and setting a low weight for an author zone is the type of possibility enabled by weighted zone scoring. This model is usually also known as *Ranked Boolean retrieval*, as the matching continues to be Boolean for each zone. Now, however, there is also document scoring based on the number of zones where the query keyword appears and the weight that is defined for each zone. The *Ranked Boolean retrieval* model is not capable of predicting the documents rank very well, since it does not use term frequency as a relevance indicator. The next step is to use term frequency for scoring in zones on the assumption that a zone that mentions a term more times is bound to be more relevant to a query containing that term.

A number of weighting schemes take into account term frequency in the calculation of the score. The simplest approach is the term frequency scheme denoted by $tf_{t,d}$ which sets the weight to be the actual number of times a term t occurs in document d,

or a function *f* of its *frequency* as follows:

$$tf_{t,d} = f(frequency_{t,d}) \tag{2}$$

However, search engines also take into account the term scarcity in the collection of documents. The inverse document frequency, a document-level statistic, is used to attenuate the weight contribution of terms that are too common in the collection and therefore are not as useful for relevance determination. Denoting the total number of documents in the collection by *N* and df_t as the number of documents that contain a term *t*, the inverse document frequency (*idf*) of a term *t* is given by:

$$idf_{\mathrm{t}} = \log \frac{N}{df_t} \tag{3}$$

2.1.4 TF-IDF weighting

The term weighting scheme that combines term frequency *tf* and inverse document frequency *idf* is denoted by *tf-idf* [14]. The *tf-idf* weight of a term *t* in document *d* is given by:

$$tf\text{-}idf_{t,d} = tf_{t,d} \,.\, idf_t \tag{4}$$

This weighting scheme assigns the highest weight to a term if it occurs many times in a small number of documents, and the lowest weight to a term that occurs in most documents. The score for a document is calculated using the overlap score measure modified to use *tf-idf*. In the overlap score measure, the score of a document *d* is the sum, over all query terms, of the number of times each query term occurs in *d*. In the modified version, the *tf-idf* weight of each term in *d* is used instead, as follows:

$$\sum_{t \in Q} tf\text{-}idf_{t,d} \tag{5}$$

2.1.5 Okapi BM25 weighting

The Okapi BM25 retrieval function [22][23] is the state-of-the-art probabilistic ranking model for nearly two decades. It ranks documents by the log-odds of their relevance. The score for a document *D* with respected to a query *Q* is calculated as follows:

$$\sum_{t \in Q \cap D} \frac{(k_3+1)c(t,Q)}{k_3+c(t,Q)} \,.\, f(t,D) \,.\, \log \frac{N+1}{df_t + 0.5} \tag{6}$$

where k_3 is a free parameter often set to 1000, $c(t,Q)$ is the number of terms in *Q*, *N*

is the total number of documents and df_t is the document frequency of t. The main component is the $f(t, D)$ formula, a sub-linear *tf*-normalization presented below:

$$f(t,D) = \frac{(k_1+1)c(t,D)}{k_1(1- b + b\frac{|D|}{avdl}) + c(t,D)} = \frac{(k_1+1)c\prime(t,D)}{k_1+ c\prime(t,D)} \quad (7)$$

Where $|D|$ represents document length, often the number of words of the document d, *avdl* is the average document length in the collection and $c(t, D)$ is the raw *tf* of t in D. The values for the free parameters *k1* and b can be set empirically. Alternatively, machine-learning algorithms can learn the optimal values for a given collection. In the simplified formula $c'(t, D)$ represents the normalized *tf* by document length using pivot length normalization [25].

$$c'(t,D) = \frac{c(t,D)}{1- b + b\frac{|D|}{avdl}} \quad (8)$$

2.2 Learning to rank

Learning to rank [9] is a machine-learning problem where the goal is to learn a ranking model from training data. The academic community and the commercial search engines have expressed a large interest in learning-to-rank in the past several years. Learning to rank methods allows search engines to leverage a large number of features for ranking their search results. With learning to rank, we want to be able to learn the optimal way of combining several features extracted from query-document pairs using discriminative training.

Learning to rank algorithms are used to find a ranking model by using supervised and semi-supervised training data. A corpus of documents for training contains a number of queries and the documents associated with each query are annotated with relevance judgments. A ranking model is determined by a learning-to-rank algorithm, such that it can predict the ranked lists in training. The objective is to obtain a ranking model that is able to predict a good ranking for new unseen queries.

2.2.1 Features

In learning to rank all the documents are represented by feature vectors reflecting the relevance of the documents in regards to the query. In the "Gov" corpus (LETOR collection [19]), there are 64 features extracted for each query-document pair. The features available are typical features used in learning to rank including the frequencies of

the query terms in the documents, the outputs of the BM25 model and other IR features such as *tf-idf* and *idf*. Some features are query dependent (Q), some are document dependent (D), and some depend on both the query and the document (Q-D). An excerpt of the learning features for the "Gov" data set is shown in Table 1.

ID	*Feature Description*	*Category*
1	$\sum_{q_i \in q \cap d} tf_{q_i,d}$ in body	Q-D
3	$\sum_{q_i \in q \cap d} tf_{q_i,d}$ in title	Q-D
6	$\sum_{q_i \in q} idf_{q_i}$ in body	Q
8	$\sum_{q_i \in q} idf_{q_i}$ in title	Q
11	$\sum_{q_i \in q \cap d} tf_{q_i,d} \cdot idf_{q_i}$ in body	Q-D
13	$\sum_{q_i \in q \cap d} tf_{q_i,d} \cdot idf_{q_i}$ in title	Q-D
16	LEN(body)	D
18	LEN(title)	D
21	BM25 of body	Q-D
51	PageRank	D

Table 1. Learning features from the "Gov" corpus.

2.2.2 Learning approaches

The **Pointwise approach** [9] to the problem of learning-to-rank assumes that the machine-learning methods are directly applied and that the relevance degree of each document will be what we want to predict individually. Pointwise approaches include the use of regression-based algorithms that output real-valued scores; classification algorithms that output non-ordered categories; and ordinal regression based algorithms that output ordered categories.

The **Pairwise approach** [9] does not try to predict the relevance score of each document, but instead operates on two documents at a time to try to predict the relative order between the document pairs. The ranking problem is reduced to a classification problem on document pairs where the objective is to minimize the miss-classified document pairs.

The output space for a **Listwise approach** [31] is one that contains either the relevance degrees for the documents associated with a query or the ordered list of documents associated with a query. In this approach, the output space contains the permutation of the documents associated with the same query, and a loss function measures

the difference between the permutation given by the hypothesis and the ground truth permutation.

2.3 Query expansion

In the context of information retrieval, query expansion [14] is a process by which queries are reformulated to include new and/or better terms with the objective of improving the quality of the search results. The defining aspect of query expansion is that it applies a number of techniques to increase the total recall of a search query. In other words, it helps the system find a larger number of matching documents. This process is particularly important in the case of user-formulated queries, which could otherwise not match any documents or leave out, from the search results, a large number of relevant documents.

2.3.1 Morphological query expansion

Perhaps the most straightforward query expansion technique is *stemming*, a technique in which query terms as well as document terms are stemmed. This means that document terms are processed into their morphological root or stem before indexing, so that alternate word forms for a term are matched. A closely related technique is called *lemmatization* in which terms are processed into their dictionary form instead, the lemma. A lemmatizer is slower than a stemmer because it needs to do dictionary lookups and its operation requires the knowledge of the context of each word in the text and not of each word separately. However, it does provide slightly better precision than when using stemming [14].

2.3.2 Knowledge-based query expansion

Another popular technique is to expand the query by finding the synonyms of the query terms and adding these terms to the query in the search process. For this effect, synonyms or related words are retrieved from a thesaurus. A thesaurus can be constructed in many ways: it can be edited manually by humans, derived automatically from word-occurrence statistics or by taking advantage of query log mining to suggest query reformulations that were executed by users. Research includes [1, 4, 5, 8, 12, 13, 18, 20, 29, 30, 33–35]. In section 4.1 we expand this discussion.

2.3.3 Statistical query expansion

Other family of approaches rely on the statistical aspects of data, [15]. Techniques such as automatic clustering of documents or results to build groups of related terms or documents. [24] A notable query expansion method Latent Semantic Indexing (LSI) [3] takes advantage of the semantic structure of documents and uses methods such as single value decomposition to identify patterns or relationships between terms in a collection. Tag co-occurrence has also been exploited [7]. Research includes [17, 27, 32]

2.4 Solr and Lucene

Apache Solr[1] is a popular enterprise search server from the Apache Lucene project. Solr provides a REST-like API, which supports XML, JSON and binary over HTTP, for indexing, searching and administrative functionalities, which allows easier integration. Solr is cross-platform and the whole codebase, written in the Java programming language, is available as open source software under the Apache License.

The Apache Lucene library is at the core of Solr and provides its full-text search functionality. Lucene is an information retrieval library, which powers a large number of search applications either through its standard Java implementation or through index-compatible ports in other programming languages.

It provides an API for full-text indexing and searching, which features high-performance indexing and efficient search algorithms with support for ranked and fielded searching and a number of different query parsers. While Lucene is a barebones library, Solr, on the other hand, is a full-featured search application built on top of it. The most prominent additions of Solr, are the REST-like API, the HTML administration interface, caching and replication.

Solr makes everything configurable using external XML files, allowing great flexibility. It also extends Lucene's existing features with more advanced text analysis, extensions to the Lucene query language, faceted search and a number of other improvements.

[1] http://lucene.apache.org/solr/

2.4.1 Schema

The details about how and for what to use Lucene's indexes in Solr are defined through a XML configuration file, called the Schema. This schema maps the source data zones or metadata into Lucene's structure. This is accomplished by defining in a *schema.xml* file the field types, such as numeric and date types that will be used, and the fields of those types that will store the data extracted from the documents.

Field Types

Terms are the fundamental unit that Lucene and consequently Solr actually indexes and uses for searching. For this effect, the definition of a field type includes one or more Analyzer configurations, to control the text analysis steps to be taken for processing terms at index or query time for a given field type. This means that tokenization, stemming and other analysis components are field type specific.

Fields

Fields are defined by declaring the value of a number of attributes. The name of the field is used to uniquely identify it and the type should refer to a field type that has been declared earlier in the schema.

Solr is flexible when defining fields. It allows for instance to control whether to store the data extracted for a field with a boolean attribute. There are a number of other boolean attributes for fields to control details about how the data is to be processed:

- **indexed** – enables searching and sorting in this field. This in turn implies that this field will be processed by analyzers and stored in an index.
- **stored** – enables the storage of the data so it can later be returned in the search results. This can be set to false to avoid redundancy, if for example the data has already been stored in another field.
- **multiValued** – this should be enabled if this field can have more than one value associated. Solr takes care of the order of the values at index-time.

In the case of fields with indexing enabled, an additional number of options are available:

- **omitNorms** – enabling this option affects document scoring and reduces memory usage for a field. This option can be used when the length of the field should not affect the scoring and for fields that will not be used for scoring at all.

- **omitPositions** – enabling this option disables phrase queries because it omits term position information from the index for this field. Doing so, can save some index space.
- **omitTermFreqAndPositions** – enabling this option affects search effectiveness negatively and disables phrase queries. This omits terms positions and term frequency from the index.
- **termVectors** – enabling this option improves search performance for features like *MoreLikeThis*, which is often referred in information retrieval literature as *SimilarTo search*. This option will probably make the index grow considerably as well as increase the indexing time.

Dynamic Fields

Fields can be created on the fly by doing a wildcard match of the name supplied for the field with dynamicField definitions. Dynamic field definitions have all the same attributes and options of regular Fields, apart from the different naming, which is a wildcard. They are only matched if the provided field name did not match any named field definition.

Unique key

This declares the field as the unique identifier for each document and helps Solr determine if an added document is new or already exists in the indexes and should be updated.

CopyField

This declaration allows copying field data into another field that might have different indexing behaviour. Another very common practice is to copy most fields into a common field to improve performance at query time.

Other declarations

There are other declarations that are in the Schema, that warrant mentioning since they influence the search behaviour. They are *defaultSearchField*, for setting the name of the default field for searching in queries that do not explicitly set it, and *solrQueryParserDefaultOperator* which declares the default search operator which is either *AND*, or *OR*.

2.4.2 Indexing details

To obtain the terms that Lucene uses for indexing and search, each field goes through a text analysis phase that involves a number of text processing steps and results in the breaking up of the field's data into terms that are then put in and index. A query usually is subject to the same or slightly modified process before issuing a search request. In Solr, the field types have an analyzer configuration associated and thus control the text analysis of fields of the given type.

An analyzer defines an ordered sequence of text processing steps to turn some stream of text into a sequence of terms or tokens. The steps can be roughly divided in three categories, character filters, tokenizers and token filters.

- **Character Filter** – manipulate data at the character level, often used to remove punctuation and replace Latin accentuated characters.
- **Tokenizer** – is the only required element in an analyzer. The tokenizer outputs a stream of tokens by processing the input text. Tokenizers often use straightforward algorithms that split on whitespace and punctuation.
- **Token filter –** token filters manipulate tokens and output what we can call the final terms that are added to the index. Simple token filters do token lowercasing, stop-word removal and stemming.

2.4.3 Data import handler

The Data Import Handler framework (DIH) provides facilities for easing the process of importing data from the most often used data sources. Most prominently, Solr offers a handler that can import data from databases but it also includes importers for indexing popular data types such as XML files, rich document formats and it can also access IMAP servers and index email, including attachments. Moreover, this framework can be used to build handlers for new data sources and is able to combine data from different data sources.

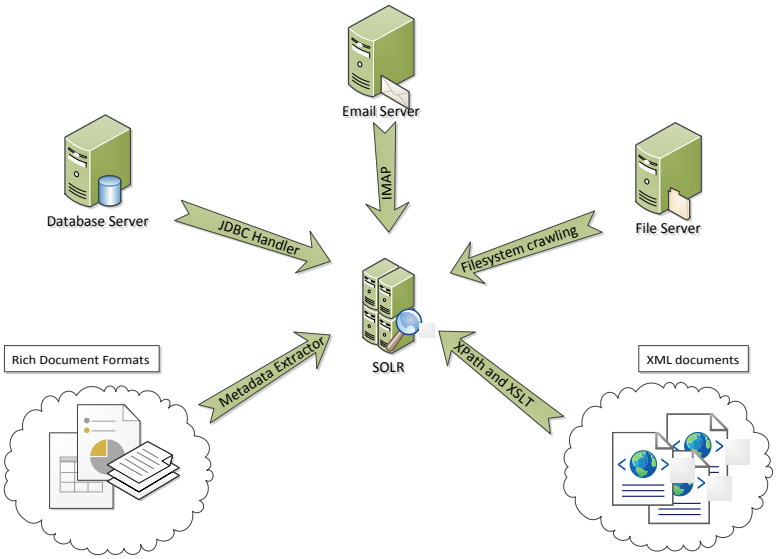


Figure 3. Solr data import Handlers.

Databases

Searching over relational database data is a very common use-case and Solr can do full imports as well as delta-imports if the database has a last-updated date field.

XML files

The handler for XML files supports XPath extraction on XML data and can execute XSLT transformations.

Rich documents

Solr makes use of Apache Tika to extract text as well as metadata from popular document formats such as the ones produced by Word, Excel and PDF files.

2.5 Summary

This chapter presented the related work for this thesis: it covered some of the information retrieval concepts related to search engines including the problem of ranking in IR; the probabilistic models; the different learning-to-rank approaches; Multiple approaches to query expansion; and a description of the Apache Solr IR system.

3

Search and Data Integration

The integration of the Genio RAP tool with the work developed in this thesis and the Solr search engine was my central activity at Quidgest. The main required feature of this integration is the creation of facilities that ease the deployment, configuration and maintenance of the full-text search functionality. The goal was to produce a stable, ready to use solution that has advanced search functionality with autocomplete and facets to replace the current search implementation based on SQL Server, which proved ineffective.

The use of Solr in this effort transfers the time complexity of the search functionality from query-time (with SQL) to index-time. The transfer of complexity into the indexing phase improves the experience in relation to the previous solution by orders of magnitude in both response time and search quality. In the end, this translates into a search function that returns results virtually without delay, in contrast to the solution faced initially where the response times, for the same data, were greater than 10 seconds. The performance gains uphold even though supplementary search features can now be provided because of the use of Solr and Lucene plugins.

There are numerous aspects to consider, in the engineering challenge of integrating a search engine into an existing framework. In the case of Solr, there are questions that arise such as the creation of a suitable data model, configuring handlers, mapping table rows to a specific data type, database access method and others. In addition, interactions with the search component (Lucene Java) had to be performed through an abstraction layer in the form of a REST API provided by Solr, which runs independently from other components.

There are usually two main strategies, for the data import. They are known as

"push", where the application is the one responsible for updating the index, and "pull" where this responsibility lies with the search engine, which has to fetch the data itself from the application's data source. We had to resort to and hybrid pull model, where the search engine component has to do the import from the database but is also assisted by the application with notifications to keep the indexes fresh. The disadvantages associated with the pull model are minimized since the system is able to request only the latest changes based on a last modified timestamp and execute a partial import.

3.1 The Genio RAD tool

Genio is a rapid application development tool (RAD) for the Windows desktop solely developed within Quidgest for internal use and licensed for use to other companies. Rapid application development is a software development methodology where development is speeded up by using iterative development and software prototyping techniques. Genio, Figure 4, is the development environment from Quidgest that provides end-to-end automatic application or prototype generation. It can generate applications in C++/MFC for back office, C#/MVC, Silverlight for Web based applications and also PLSQL and WCF Web Services. There is also partial support for the generation of mobile applications with PhoneGap, Microsoft-Office task panes, SharePoint and Enterprise Java Beans (EJB).

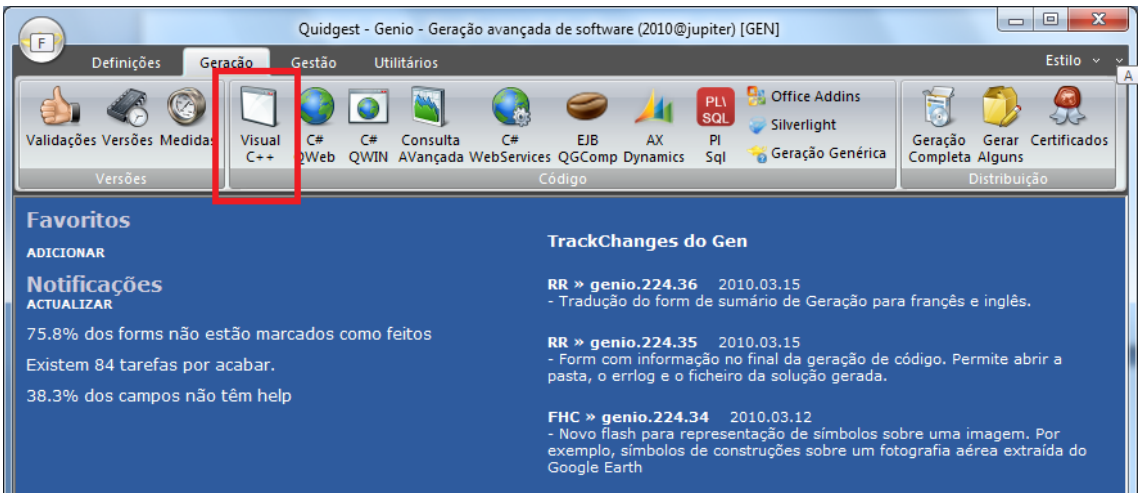


Figure 4. Genio IDE highlighting C++/MFC generation.

Quidgest has been developing the Genio tool since 1991 to be able to come to market rapidly with initial software prototypes for its clients and improve upon them with

specialized development subsequently. With Genio and its associated RAD methodology, Quidgest has been developing a range of different products such as systems for local and global Windows networks, Web portals, Web services, systems for mobile devices, Microsoft Office add-ins, and business intelligence tools. It has been used in a variety of areas such as Human Resources, Supply Chain, Financial, Document Management, Health, etc.

3.2 Integration requirements and constraints

To bring the data from the Genio databases into Solr we evaluated initially the push method as it is considered the cleanest method and should be less wasteful than a pull approach. However, there is no software layer or middleware in Genio we can use to piggyback on to create hooks for CRUD actions, which would make the task of implementing a push approach less intrusive. It would only be possible to replicate the hook functionality by duplicating some code across all the controller classes in the system. With that said, implementing the push method for every type of application that Genio can create was not feasible due to time constraints. Genio supports numerous backends and languages, and therefore it would have to generate code to serialize application data into XML or JSON to execute create and update documents for each one of them. This issue is aggravated by the fact that this simply would not be possible in the case of some backends or produce different behavior across solutions.

Therefore, a final solution that works across the board with a minimal footprint of structural changes in Genio was considered preferable due to maintainability concerns and for future compatibility with new backends. This in turn means that a pull approach fits the requirements better. However, we want to keep some of the advantages of the push method, such as faster update times and immediate feedback. It was important in some applications that the search system would return the most up to date results, especially in the case of delete actions that leave behind inexistent documents in a pure pull approach. For that reason, the solution that we took is a hybrid one.

Genio is now able to deploy a custom Solr installation and add generated pull scripts for each search configuration defined in the Genio's application configuration. It adds hooks to the application controllers in order to be able to call the search engine through REST to execute imports and other simpler actions. The scripts generated can perform a partial import with only the data modified since the last indexing time, which allows for much shorter periodic updates of the index. Deletions in the final ap-

plication are mirrored immediately by issuing a delete command to the search engine with a unique id.

3.3 Importing data from Genio

3.3.1 The Schema

As described in section 2.4.1, the schema file (XML) maps the source data zones or metadata into Lucene's field types. This allows us to index table attributes differently depending on their type and usage. Instead of generating a different schema for each application, we opted to define the needed field types and assign special dynamic field definitions to each one of them. This means that we can decide the field type of each attribute through its field name, and thus decide its analysis process. In other words, field names generated by Genio are a composition of field name and the field type suffix as defined in the dynamic field definitions in the schema.

```
<!-- Polish -->
<fieldType name="text_pl" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
   <tokenizer class="solr.StandardTokenizerFactory"/>
     <filter class="solr.LowerCaseFilterFactory"/>
     <filter class="solr.StempelPolishStemFilterFactory"/>
  </analyzer>
</fieldType>
<dynamicField name="*_pol" type="text_pl" indexed="true" stored="true" multi-
Valued="true"/>
```

Figure 5. Polish analyzer definition.

Language	*dynamicField*
Portuguese and Tetum	_por
English	_eng
French	_fre
Spanish	_spa
Catalan	_cat
Danish	_dan
Polish	_pol
German	_ger
Simplified Chinese	_zho

Table 2. Language support dynamicField mapping.

Because these applications are used in many different regions of the globe, we added to the schema various dynamic field definitions with language-specific analyzers for each language supported. This accommodates all the locales supported in Genio, see Table 2. When generating a final application, the search engine configuration uses the appropriate analyzers for the text fields according to the main language of the application, for instance the configuration for Polish text is shown in Figure 5.

This schema with its dynamic field definitions is the foundation that enables the generation of the search engine configurations for any application, since there is a matching Solr field type for every type of field used. For instance, we can decide to index a document *Person* with a *Social Security Number* attribute without having predefined a specific field for it. The field could be named "ssn_i" and due to the field name suffix the type of the field is assumed a number.

To support the search through Solr in Genio it is necessary to import the data into search indexes. In our case we want to skip communicating with Genio-generated applications and import directly from the application's data source taking advantage of the field definitions from the schema. This data source is usually a SQL Server or Oracle database. Therefore, we have resorted to the Data Import Handler (DIH) module. DIH is a contributed module in Solr that facilitates index building from a number of data source types. It can not only import data from relational databases, which is our main concern, but also from other sources such as the filesystem, FTP, SFTP and HTTP as well. It contains a number of plugins that allow pre-processing XML with Xpath and a metadata extractor for all kinds of files through Tika[2].

```
#if ($db.Equals("S") || $db.Equals("Q") || $db.Equals("L"))
<dataSource name="db" driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
responseBuffering="adaptive"
url="jdbc:sqlserver://{host}:1433;databaseName=${system}" user="{user}" pass-
word="{password}"/>
#elseif ($db.Equals("O"))
<dataSource name="db" driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@//{host}:1521/${system}" user="{user}" pass-
word="{password}"/>
#end
```

Figure 6. JDBC data source definition.

[2] https://tika.apache.org/

To use the DIH with a relational database we have to provide the Java-based database driver (JDBC) that corresponds to the RDBMS system in use by the application. This allows the system to issue SQL commands to this database. This step is executed in a per-application basis by the deployment process by copying the relevant driver into the classpath of the search engine. The relevant data source definition is then written to the dataconfig.xml file, which rules over DIH behavior, with the database host, name, and authentication.

3.3.2 Genio Solr Extension

The rest of the DIH dataconfig.xml generation is carried out with the help of a new extension to Genio (GenioSolrExtension). This extension is able to generate the SQL query that returns all fields that will be indexed based on the definitions of textual research only. We use only LEFT JOINs to preserve incomplete "documents". The generated SQL query is able to execute delta-imports, indexing only the items changed since the last index time, reducing greatly the time of periodic indexing. To choose the data types of the fields used to index we built a file that maps the types of Genio to the corresponding types in the Solr schema. Additionally Tika is used to extract the metadata from documents that exist in the database as file streams such as Office .doc or PDF.

Type	*Description*	*fieldType*
+	Primary key	string
CE	Foreign key	string
C	Simple text. Highly used; text, ids or numbers.	string
MO	Multi-line text	text
MM	Multiline text with RTF	text
N	Decimal numbers	string
$	Currency	currency
L	Boolean	boolean
D	Date	date
Y	Year	integer
OD	Last modified date	date

Table 3. Genio data types Solr fieldType mapping.

This extension implements different methods to compose specific parts of the full SQL statement necessary to import data to the index. The extension is coded in C# and is part of the Genio codebase now. Due to this, its methods can be called from within the normally used Velocity templates to generate the final XML data configuration.

Reducing the number of requests to the database is desirable. Thus, for multivalued table attributes we use the group_concat feature present in most database systems. This allows for a single SQL statement to be made to get all the records. The different values for a multivalued attribute are agregated in a single column. In SQL Server however, an open source implementation of the group_concat feature has to be installed first in the database server since this database system does not provide an implementation.

GenioSolrExtension's main purpose is to generate data import scripts for each "Search component" defined in the Genio application specification. This mechanism allows the system to periodically update its index. It is also able to ping the server for a partial upgrade when a change is detected by the final application. Partial update is based on last indexing date, supported by date fields across all tables. An overview of the structure of the configuration script generated can be seen in Figure 7.

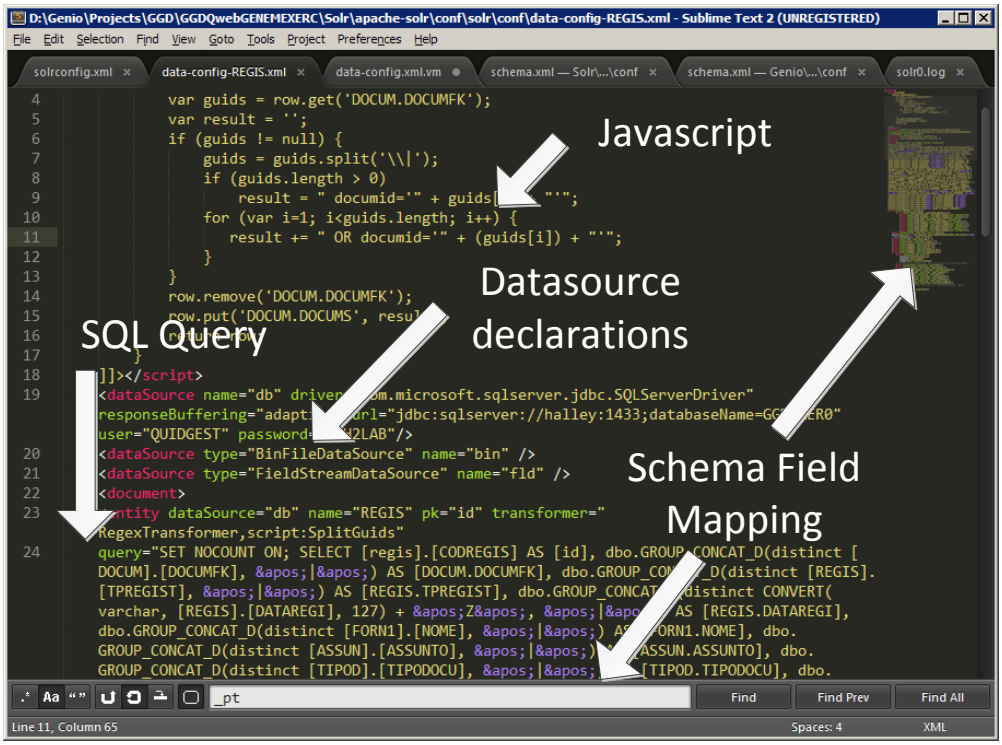


Figure 7. Data import script structure overview.

3.4 Text-based search

Solr's decoupled structure can offer a degree of separation of concerns that has the potential to allow easier scaling by allowing the search component of an application to be deployed and scaled separately. We are able to improve the overall search engine text retrieval performance and offer more features by expanding the capabilities of the search engine with extensions and other modules.

Thus, we were able to implement new retrieval models and evaluate them using standard test collections separately.

3.4.1 Implemented ranking functions

Lucene and consequently Solr have implementations of some of the most prevalent algorithms for ranking results. New retrieval functions and improvements to existing ones have been published in scientific papers, however there are no implementations of these algorithms and for production search engines.

In this work, we implemented BM25L [11] and BM25+ [10], two modifications of the state-of-the-art retrieval function Okapi BM25 for use with Lucene and Solr. Okapi BM25 provides an excellent ranking model that remains virtually unchanged for nearly two decades. However, BM25 penalizes long documents too much and both BM25L and BM25+ eliminate this known problem. Therefore, they allow this system to provide better search results, especially in a heterogeneous collection scenario.

An implementation of the LTC variation of TF-IDF with logarithm tf-normalization was also implemented. This improves the ranking performance of this model considerably, which may be suitable for use in less powerful systems.

BM25L

The BM25L ranking function is a modification of the classic Okapi BM25 with an improved *tf* normalization formula $f'(c, D)$. It is a shifted version of the original *tf*-normalization, which effectively avoids overly-penalizing long documents. A new parameter δ is introduced to control the shift. Evaluation has shown that $\delta = 0.5$ is a good default [11]. The TF-normalization is given by:

$$f'(t, D) = \begin{cases} \frac{(k_1+1) \cdot [c'(t,D)+\delta]}{k_1+[c'(t,D)+\delta]} & if\ c'(t, D) > 0 \\ 0 & otherwise \end{cases} \quad (9)$$

BM25+

This model is very similar to the previous and avoids overly-penalizing long documents like the classic BM25 formula. It also introduces a new parameter δ to directly influence the lower-bound of the *tf*-normalization formula. Evaluation has shown that $\delta = 1.0$ is a good default [10]. The formula is as follows:

$$f'(t,D) = \frac{(k_1+1)c(t,D)}{k_1(1- b + b\frac{|D|}{avdl}) + c(t,D)} + \delta \quad (10)$$

TF-IDF LTC

Logarithmic TF-normalization helps TF-IDF improve its retrieval metrics, since term frequency is upper-bounded. This means that the score difference between two documents with a large number of occurrences of a query word is attenuated. The formula is shown below:

$$tf_{t,d} = 1 + \log(frequency_{t,d}) \quad (11)$$

Evaluation

These similarities were evaluated using the TREC-9 Filtering data (OHSUMED dataset) and have shown to improve retrieval across various metrics as follows:

	DefaultSim	*LTC*	*BM25*	*BM25L*	*BM25+*
P@10	0.254	0.260 (+2.36 %)	0.265	0.294 (+10.94 %)	0.290 (+9.43 %)
nDCG@10	0.350	0.356 (+1.71 %)	0.374	0.388 (+3.74 %)	0.387 (+3.48 %)
MAP	0.145	0.151 (+4.13 %)	0.164	0.173 (+5.49 %)	0.172 (+4.88 %)

Table 4. Evaluation results of new ranking functions.

```
<!-- This loads the custom lucene-relevance module. You can comment it out
      completely if you need to upgrade to a new/incompatible Solr version -->
<similarity
class="pt.unl.fct.di.lucenerelevance.search.similarities.BM25SimilarityFactory
">
    <str name="model">PLUS</str>
    <str name="k1">1.2</str>
    <str name="b">0.75</str>
    <str name="d">1.0</str>
</similarity>
```

Figure 8. Solr similarity setting for BM25+.

Each of these retrieval functions or similarities can be used in Lucene and in Solr from similarity factories. There are sane defaults for each model and therefore only the model needs to be set explicitly because good performance defaults are put in effect when each parameter is ommited. The new BM25SimilarityFactory can instantiate the classic BM25, BM25L and BM25+ and initialize the parameters for each model. A Solr similarity configuration enabling the use of BM25+ is pictured in Figure 8.

3.4.2 Facets and filtering

Faceted search is a technique that allows the user to explore the available results by successively applying filters according to a faceted classification system. User-generated tags can be browsed using this technique to narrow-down results: as an example, a user can find the movie Rocky by browsing to the tag “action” which filters further tag navigation to the tags that occur alongside the tag “action”, the user could then browse to tag “boxing”, for example, and find the movie “Rocky” in the search results. The application interface also allows the user to filter the results by a number of fields, such as the content type and rating.

3.4.3 Fuzzy search

The search functionality supports approximate string matching in two modes. Solr can match full words with only approximate substrings and can correct spelling mistakes with the Spellcheck component by finding dictionary words that match an entered term. In the Spellcheck component, the dictionary words are sorted using the Levenstein distance.

3.5 Management interface

We built a control panel interface (see Figure 9) using Ruby on Rails (RoR) to support the configuration of the most customized features. It was important for this search engine to be monitored and tweaked for different uses and clients.

The page displayed in Figure 10 allows the search engine administrator to inspect the current state of a search core, having access to information about the data import status, number of total documents and index size. This is useful maintenance information especially in the case of this type of tool integration, where components such as the database access can fail independently.

In Figure 11, we can see the interface for adding and removing stop words, for a specific search core. This modifies the behavior of the stop word filter to remove these words. There are similar pages for the configuration of synonyms and spellings.

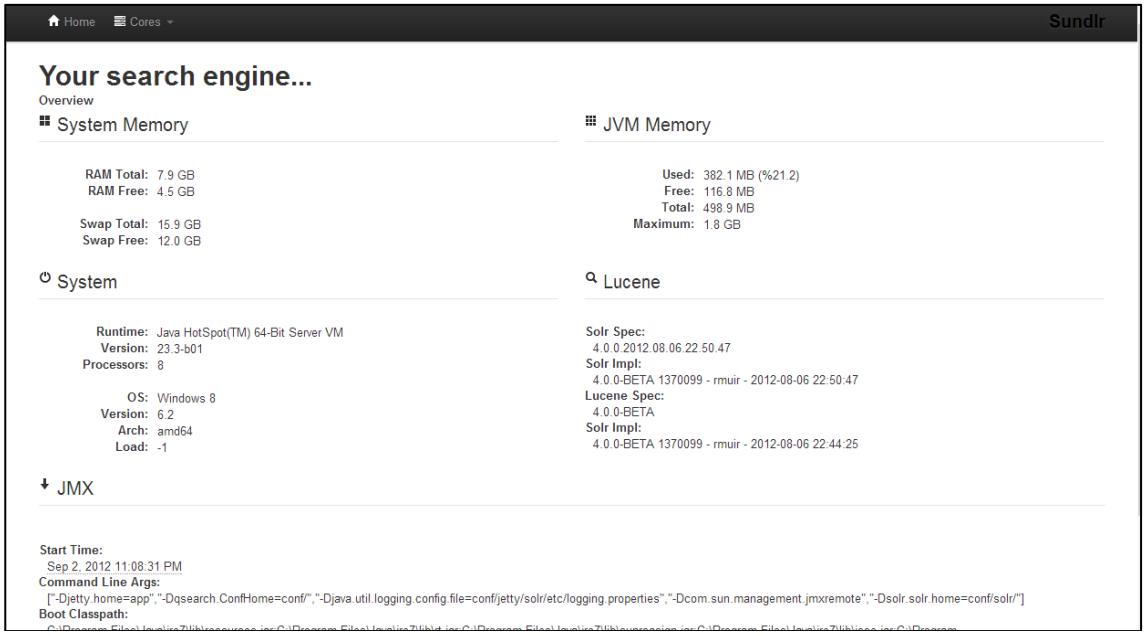


Figure 9. Sundlr system overview page.

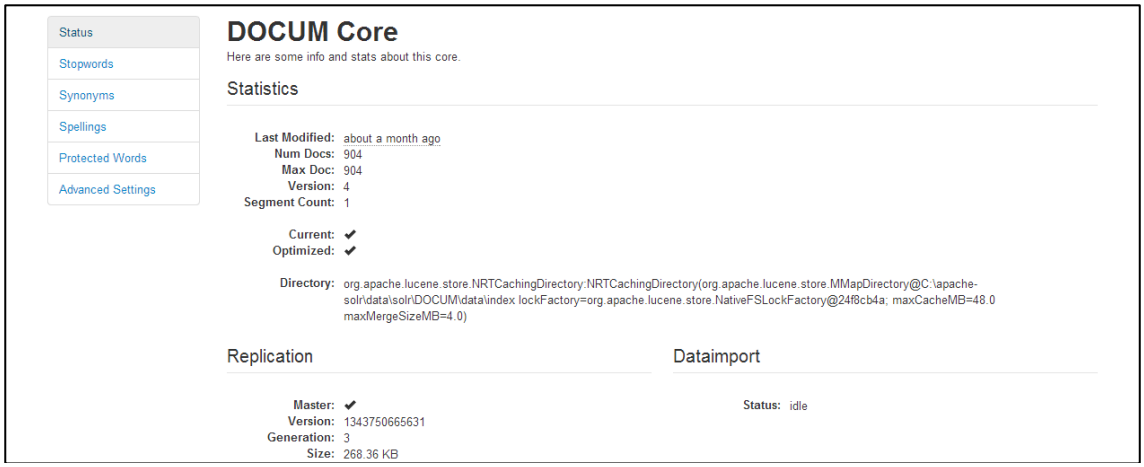


Figure 10. Sundlr search core status page.

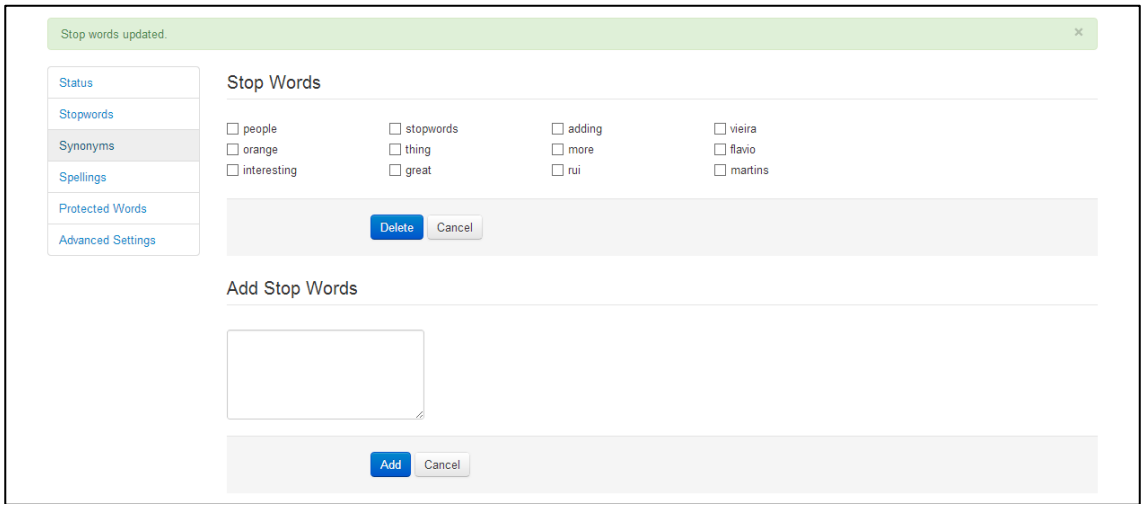


Figure 11. Sundlr stop words editing.

3.6 Summary and discussion

In this chapter, we described the integration points between Genio and our search framework. The development of an extension to Solr was required to accommodate the system requirements of Genio.

The heterogeneity of the indexed data in the system encountered lead to the necessity to implement new retrieval functions that are able to cope better with collections of documents of different types and sizes. In addition, due to this variety, faceting and filtering was enabled for text search activities.

The management interface was crucial in delivering an overview and a simple configuration interface. Because, Genio is used in completely different areas such as health and customer relations, this ability to allow clients to fit the search engine to their needs was very desirable.

4

Domain-specific knowledge-based query expansion

The Simple Knowledge Organization System (SKOS) [26] is, since 2009, a World Wide Web (W3C) recommendation to represent controlled structured vocabularies, such as thesauri, classification schemes and taxonomies. It is a standardized and multi-language way to represent vocabularies commonly known as *Knowledge Organization System* (KOS). Most often, KOSs are built within a community of practice for a specific domain, and as a result are tailored for the specific needs in that domain. The SKOS is a modular and extensible family of languages that allows the representation of KOSs from various domains. It provides a migration path for existing vocabularies representations, therefore it reduces the fragmentation problems of multiple *organization system* standards. SKOS is also seeing an uptake in adoption as a lightweight yet powerful representation format for thesauri. Due to the qualities of SKOS and the availability of thesauri in SKOS format, it makes sense to incorporate them in retrieval systems. Query expansion is an obvious application that can help to minimize significantly retrieval problems arising from mismatches between user input and indexed terms.

In this chapter, we propose a query expansion framework that makes use of SKOS thesauri. The framework evaluates the input terms entered by the users and adds terms sourced from SKOS thesauri concepts with the objective of improving the general retrieval performance. At some point, because a thesaurus in SKOS format can have labels for a concept in many languages, they will also present an opportunity for cross-language expansion and will possibly enhance multi-language search.

4.1 Background

Query expansion techniques have been researched for decades. Usual techniques involve taking advantage of synonyms, the knowledge of the morphology of words, and the use of dictionaries to improve the quality of search results. Much of the research work in the IR field delves into query expansion techniques using linguistic resources. There are various papers published about using thesaurus and thesaurus-like vocabularies such as WordNet. In a paper Voorhees [29] found that expanding using synonym sets from WordNet automatically can degrade performance, while hand picking concepts improves less well developed queries significantly. Later works seem to have focused on the medical field to avoid the problems verified by Voorhees with WordNet, a linguistic resource that is too broad. In [6] Hersh et al. presented OHSUMED a medical test collection with judgments, based on journals from the NLM. In [1] Aronson and Rindflesh combined expansion using the UMLS thesaurus with an automatic retrieval feedback method to achieve better Average Precision (AP). Later in [5] Hersh et al. would also try thesaurus-based query expansion with UMLS. In this paper they used explosions (expanded all child terms) since in OHSUMED the journals are indexed using the narrowest indexing terms. They also found that simply adding the new expansions to the query is a simple and effective approach in IR statistical systems. In [28] Theobald et al. developed a system that performs expansion only when thematic similarity is above a threshold, but found that it implies a high execution cost and hard-tuning of parameters. Later works [8] [35], by Lin and Demner-Fushman and Zhou et al. investigate the creation of custom similarities to score concept matches with more emphasis than word matches. In the work by Zhou they use the MeSH thesaurus and use only one level hierarchical relationship and synonyms.

4.2 Simple Knowledge Organization System (SKOS)

SKOS [16] is an application of the Resource Description Framework (RDF) that provides a model for expressing the basic structure of most concept schemes such as thesauri, classification schemes, taxonomies, and other types of controlled vocabulary. SKOS documents can be published on the Web and exchanged between different software applications and computer systems in an interoperable way just like other RDF documents. The main reason for this is that SKOS takes advantage of Uniform Resource Identifiers (URIs) to identify uniquely and universally each resource/concept.

The use of URIs makes SKOS decentralized and augments its interoperability, enabling interaction between resources across a network. The *SKOS Core Vocabulary* defines the RDFS classes and RDF properties to represent *concept schemes* as an RDF graph.

A complete description of the SKOS specification is outside the scope of this thesis, for more details the reader is referred to the SKOS Core specification [16].

```
<rdf:Description rdf:about="http://www.nlm.nih.gov/mesh/D018599#concept">
   <rdf:type rdf:resource="http://www.w3.org/2004/02/skos/core#Concept"/>
   <skos:inScheme rdf:resource="http://www.nlm.nih.gov/mesh#conceptScheme"/>
   <skos:prefLabel>Witchcraft</skos:prefLabel>
   <skos:altLabel>Sorcery</skos:altLabel>
   <skos:altLabel>Sorceries</skos:altLabel>
   <skos:broader rdf:resource="http://www.nlm.nih.gov/mesh/D008273#concept"/>
   <skos:broader rdf:resource="http://www.nlm.nih.gov/mesh/D026443#concept"/>
      <skos:scopeNote>An act of employing sorcery (the use of power gained
    from the   assistance or control of spirits), especially with malevolent
    intent, and the exercise of supernatural powers and alleged intercourse
    with the devil or a familiar. (From Webster, 3d ed)
   </skos:scopeNote>
</rdf:Description>
```

Figure 12. SKOS concept extracted from the MeSH vocabulary.

4.2.1 Concepts

In the *SKOS Core Vocabulary*, the fundamental building block is the *concept*: a resource for which the value of the *rdf:type* property is *skos:Concept*. Each *concept* has an unique URI identification and is characterized using RDF properties, to declare the preferred indexing term, alternative terms, notes and semantic relations.

The concept with the URI *http://www.nlm.nih.gov/mesh/**D018599**#concept* is presented in Figure 12. This definition corresponds to the notion of "Witchcraft" or "Sourcery" since these are the values for the skos:*prefLabel* and skos:*altLabel* properties assigned to this resource (detailed in the next section). The value of the property *skos:scopeNote* contains a short text with the definition of the concept.

The *concept* "Witchcraft" is linked to a broader *concept* "Magic" with a *skos:broader* property. The use of broader-narrower relationships between different concepts creates a hierarchically organized graph of conceptual resources.

4.2.2 Labels

SKOS Core has three properties to assign natural language lexical labels to a *concept*: *skos:prefLabel*, *skos:altLabel* and *skos:hiddenLabel*. These in turn are sub-properties of *rdfs:label*, which means their value can include a language tag (e.g. Witchcraft@en).

There can be one preferred label (*skos:prefLabel*) per language tag. These are called

the *indexing terms* or *descriptors* of the concept. It is recommended that no two concepts in the same *concept scheme* have an identical preferred label since some applications could use these as a concept identifier. The *skos:altLabel* property is generally used to declare synonyms for the preferred label, abbreviations and acronyms. Finally, assigning hidden lexical labels with *skos:hiddenLabel*, contemplates text-based indexing applications that can use these labels as entry points to correct common misspellings of preferred or alternative lexical labels. For instance, the "Witchcraft" concept could have "Whitchcraft" as one of its hidden labels.

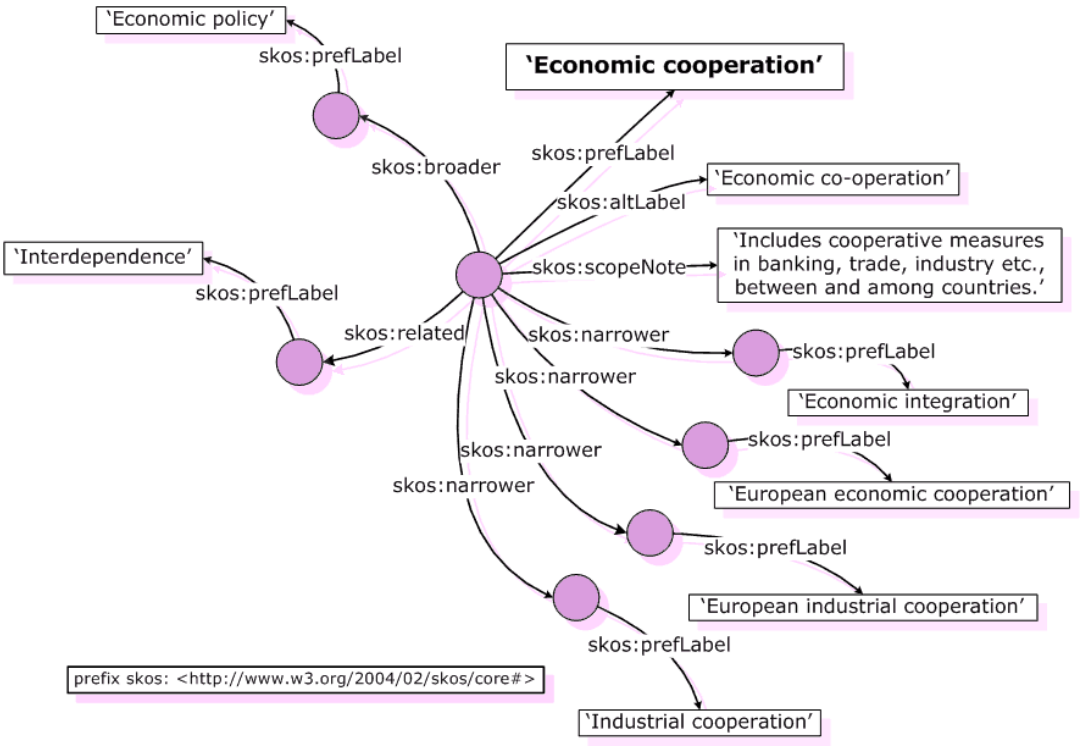


Figure 13. UKAT thesaurus RDF graph visualization.

4.2.3 Semantic Relations

It is important to be able to assert semantic relationships between concepts. A number of disconnected concepts is a poor KOS, since the meaning of a concept is also established by its links to other concepts in the KOS. The *SKOS Core Vocabulary* has three properties to express semantic relationships: *skos:broader*, *skos:narrower* and *skos:related*.

The property *skos:broader* is used to assert that one concept is broader than a given concept. For instance, "Economic policy" is a broader or more general concept than "Economic cooperation" (see Figure 13). On the other hand, the *skos:narrower* property

asserts that one concept is narrower or more specific than the given concept. For instance, "European economic cooperation" is more specific than "Economic cooperation". Both these properties *skos:broader* and *skos:narrower* establish relations that originate a hierarchical graph organization.

The property *skos:related* asserts non-hierarchical semantic links between concepts. It is used when two concepts are neither more general nor more specific than the other is.

4.3 Lucene SKOS-based query expansion

The implemented system is based on the Lucene[3] information retrieval library from the Apache Foundation. It is possibly the most commonly used library to add search capabilities to applications and is the foundation of the enterprise search engine Solr[4].

Lucene is a modular Java-based library, which at the core implements the most essential mechanisms and algorithms necessary for the implementation of search engines. It can be extended with packages and plugins that implement features that are more advanced. The core Lucene library exports interfaces and implementation classes that can be implemented or extended to build new functionalities. The core interfaces are *QueryParser*, *Analyzer* and *TokenFilter*, these allow the implementation of new analysis components and tokenizers that are compatible with Lucene.

Usually with Lucene, the common practice is to do query expansion at indexing time using a Filter or an Analyzer. However, this means that it indexes each document using the document's terms and the expanded terms in addition. This results in a large index and in a computationally lighter query-processing phase. This is an acceptable solution when the number of synonyms is small. However, when using a thesaurus, most terms will have a large number of expanded terms resulting in efficiency problems for indexing time and index size. Due to this fact, we argue that query expansion at query-time is the right choice allowing changes in the thesaurus without re-indexing.

We developed a Lucene plugin to implement query expansion based on the data encoded on SKOS thesauri. To accomplish this task the plugin contains various Lucene

[3] http://lucene.apache.org
[4] http://lucene.apache.org/solr

component implementations, such as a custom analyzer that adds alternative terms sourced from a SKOS thesaurus and a query parser that provides a query-time query expansion solution that uses SKOS thesauri sourced terms and optionally boosts them according to their type.

4.3.1 SKOS Engine

This component loads SKOS vocabularies from a remote URL or from a file, which uses one of the standard RDF syntaxes. It can also load files compressed with ZIP.

It provides methods to access all KOS data needed for the expansion process. The main methods are *getConcepts(label)* which returns all the concepts (URIs) matching a given label and methods such as *getPrefLabels(uri)* which return the labels of the requested type for a given concept URI. Performance was a primary concern, since these methods are going to be executed several times.

Therefore, we evaluated the performance of Apache Jena TDB and other RDF triple stores and databases but ended up settling on a combination of Apache Jena, for loading RDF and doing simple entailment and Lucene for storage and querying.

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
INSERT { ?subject rdf:type skos:Concept }
WHERE  {
          { ?subject skos:prefLabel ?text } UNION
          { ?subject skos:altLabel ?text } UNION
          { ?subject skos:hiddenLabel ?text }
       }
```

Figure 14. SPARQL query to infer concepts without explicit type.

The first time a specific SKOS file is opened with the engine it is loaded into memory using Apache Jena and a simple entailment (see Figure 14) is executed to improve our chances of finding all the available concepts. Then it creates a new Lucene index and iterates over all the concepts it found and for each concept adds one new document (see Figure 15) in the newly created index.

Finally, the engine saves the index to disk so that subsequent loads of the same SKOS file are close to instantaneous since they will reuse the existent index. In addition, the engine will add a suffix to the newly created index, indicating the languages indexed. If the languages are later changed, the engine creates another index with the new requested languages.

```
stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<uri:http://www.nlm.nih.gov/mesh/2006#D001678>
stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<pref:biogenesis>
stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<alt:origin of life>
stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<hidden:biogeneses>
stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<hidden:life origin>
stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<hidden:life origins>
stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<broader:http://www.nlm.nih.gov/mesh/2006#D001686>
```

Figure 15. SKOS concept as indexed by SKOS engine.

In summary, indexing SKOS data into a Lucene index allows the engine to use Lucene as a NoSQL database and take advantage of its performance and caching mechanisms to execute the necessary queries extremely fast.

4.3.2 SKOS Label Filter

A *TokenStream* is a chain of steps starting with a *Tokenizer* that splits characters into initial tokens, followed by any sequence of filters (*TokenFilter*) that pre-process and further modify the tokens produced. The Lucene API exposes the *TokenStream* to filters as an iterator. Filters call *incrementToken* to advance to the next token, and for each token have access to specific attributes of the token. There are a number of core token attributes such as the text content of the token (*CharTermAttribute*) or the start and end offset in the original not tokenized text (*OffsetAttribute*). In addition, a custom *Tokenizer* or *TokenFilter* can make use of custom attributes. These Lucene abstractions allow the creation of a number of features that involve processing tokenized terms one by one, in our case term expansion.

The main logic to perform query expansion using SKOS is implemented through a Lucene *SKOSLabelFilter*, a Lucene *TokenFilter* that adds additional terms for each recognized concept while traversing the token stream. This filter is parameterized with a SKOS engine that will be used for querying for SKOS data and a list of expansion types. In addition, to enable multi-term expansion you can parameterize it with the maximum number of tokens it will join to search for a multi-term concept.

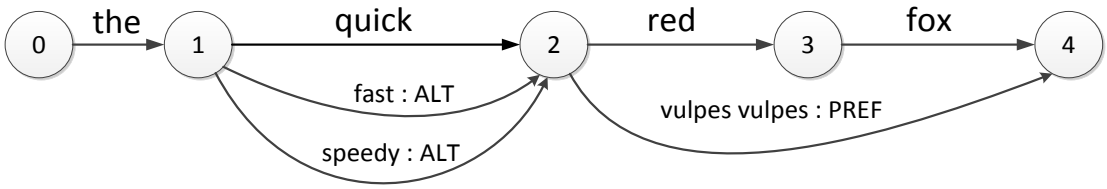


Figure 16. Example TokenStream graph after SKOSLabelFilter.

In Figure 16, we can see an example of a directed acyclic graph that results from the application of the *SKOSLabelFilter* to the text "the quick red fox". In this example the SKOS engine returned the alternative labels "fast" and "speedy" for the term "quick", while "red fox" is the alternative name for the species "vulpes vulpes". Because of the new *TokenStream* graph, with additional arcs, resulting from the expansion process, new queries that previously would not match, such as "the speedy red fox", do match this time. A custom attribute (*SKOSTypeAttribute*) is added to the new tokens created in the expansion process to signal the type of the expansion that originated them. The types of expansion are PREF, ALT, HIDDEN, BROADER, NARROWER and RELATED. To be able to encode this information in the file system index we assigned an integer representation to each type of expansion, which is stored as a *Payload.*

4.3.3 SKOS Analyzer

The SKOS Analyzer is a ready to use Lucene analyzer specifically designed to use with *SKOSLabelFilter* for SKOS-based term expansion. It can be used to index documents, which will be expanded using the provided SKOS thesaurus or used directly with a Lucene *QueryParser* for query-time expansion.

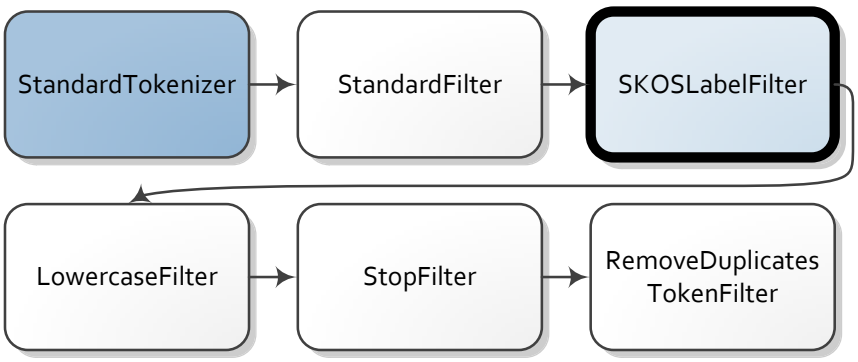


Figure 17. SKOS Analyzer filter chain.

The chain of components of *SKOSAnalyzer* (see Figure 17) serves as a reference for possible new analyzers to use with other languages or with other specific particularities. The chain starts with the *StandardTokenizer* and *StandardFilter* which tokenizes the text into a *TokenStream* using the word break rules from the *Unicode Text Segmentation algorithm* and does some normalization of the tokens extracted. These were chosen be-

cause they are suitable for many languages.

SKOSLabelFilter, traverses this *TokenStream* and tries to add new tokens. As described in the previous section this filter queries a SKOS engine, that provides the lexical labels linked to a *concept* through SKOS properties, and adds, alongside the original token, a new token for each label found, accompanied with a *SKOSTypeAttribute*.

All the terms in the token stream are then normalized into lowercase and common terms are deleted, such as "the", "a", etc. with a *stopword* filter.

Just like the *SKOSLabelFilter*, the analyzer can expand using a configured set of languages, enabling it to perform a something that could be called *concept translation*. Because of this feature, the SKOS engine needs to returns preferred lexical labels as expansion terms and thus we added the *RemoveDuplicatesTokenFilter* to remove duplicated tokens that arise from this fact and from labels which are orthographically equal in various languages.

4.3.4 SKOS Query Parser

Lucene's standard query parser (*StandardQueryParser*) is the default query-parsing framework. It parses queries in clear text which contain simple syntax that we are accustomed to from Web search engines and additional Lucene specific syntax, and processes it into a tree of *QueryNode*s.

A Lucene *Query* is a tree of *QueryNodes* of various types, such as *BooleanQueryNode*, *WildcardQueryNode* or *NumericQueryNode*. This tree is processed and reprocessed in various steps until the structure of the original query in clear text is mirrored in Lucene's internal *Query* representation. This chain of steps is named *QueryNodeProcessorPipeline* and each step is called a *Processor*. There are *Processors* that carry out the task of splitting the query string into its query clauses, processing Boolean operators, wildcard terms, numeric ranges and other query syntax.

We created a new *SKOSQueryParser* by extending the *StandardQueryParser* and inserted a new SKOS *Processor* in the pipeline, before the analysis step. The new processor uses the *SKOSAnalyzer* and produces new *QueryNodes* with the additional tokens.

As can be seen in Figure 18, the SKOS processor creates multiple *QueryNodes* for each token: a *QueryNode* for each one of the lexical labels found in a SKOS thesaurus. Each term in the query is expanded using both preferred and alternative lexical labels for the term in the thesaurus and the labels for other semantically related *concepts*.

Also in Figure 18, we can see the motivation for the development of this query parser. This query parser was developed to allow the manual configuration of boost val-

ues for each type of expansion, (*PREF, ALT, HIDDEN, BROADER, NARROWER, RELATED*). This feature allows information curators or domain experts to fine-tune the influence of the expanded terms, to fit their needs or information retrieval scenario, and allows dynamic changes of boost values or user based values. In this example, we can see that PREF and ALT expansion terms are given a boost of 0.5 while the expansion of "fox" to the BROADER term "canine" is given 0.25. The boost values are intimately related to the scoring in Lucene and thus will be explained further in the next sections.

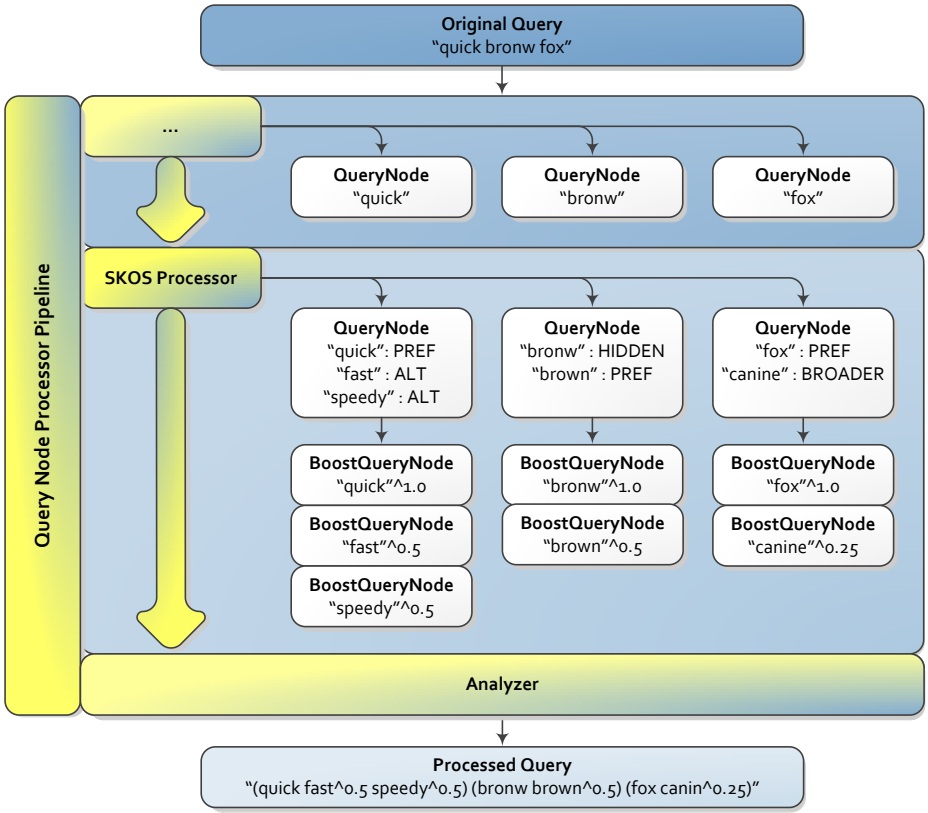


Figure 18. Query parsing pipeline.

4.3.5 Scoring

We decided to use existing ranking models and weave concept weighting into these models. Since the release of version 4.0, Lucene ships with a number of advanced scoring algorithms but continues to use by default a slightly non-classical *tf-idf* model as the default ranking method. In the next sections, we detail the scoring method in regards to the default Lucene ranking model (the scoring in other models is analogous).

Baseline scoring method

Denoting the number of occurrences of term t in document d by $frequency_{t,d}$, the $tf_{t,d}$ computation is given by:

$$tf_{t,d} = \sqrt{frequency_{t,d}} \tag{12}$$

Denoting the total number of documents by N and df_t as the number of documents that contain a term t, the inverse document frequency (idf) of a term t is given by:

$$idf_{\mathrm{t}} = 1 + \log \frac{N}{df_t + 1} \tag{13}$$

The term weighting scheme resulting from combination of term frequency $tf_{t,d}$ and inverse document frequency (idf) is denoted by *tf-idf*. The *tf-idf* weight of a term t in document d is given by:

$$tf\text{-}idf_{t,d} = tf_{t,d} \cdot idf_t \tag{14}$$

The *tf-idf* score of a document d is thus the sum, over all query terms, of the *tf-idf* weight of each term t in d, as follows:

$$Score(q,d) = \sum_{t \in q} \left(tf_{t,d} \cdot idf_t \right) \tag{15}$$

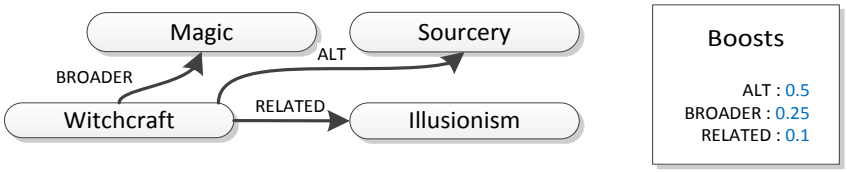
Expansion scoring method

The effective scoring function in Lucene augments the simple *tf-idf* scoring described above. Its formula can be succinctly conveyed as follows:

$$Score(q,d) = coord_{q,d} \cdot \sum_{t \in q} \left(tf_{t,d} \cdot idf_t \cdot boost_t \right) \tag{16}$$

where $coord_{q,d}$ is a score factor calculated using the count of query terms that are found in the specified document and $boost_t$ is a boost factor of term t in the query q.

With the $coord_{q,d}$ factor we ensure that a document with a higher count of matched query terms will score higher than a document with a low count. This is especially interesting in this expansion framework, since the score for documents that match the original query terms and match the alternative terms, broader and narrower concepts or even related concepts derived from the original query will be higher.



Score = coord(q,d) * (tf-idf(witchcraft) + tf-idf(sourcery)*0.5 + tf-idf(magic)*0.25 + tf-idf(illusionism)*0.1)

Figure 19. Expansion scoring example.

However, not all matches have an equal importance. The inclusion of a $boost_t$ factor tabulated according to the expansion type that originated the term can be used to ensure that original query terms are weighted more heavily than the expansions. In addition, it can be used to ensure that an expansion with a weaker semantic relation type does not weight as much as an alternative term, for example (see Figure 19).

4.4 Evaluation

In this section, we evaluate the query expansion framework proposed in this chapter as well as the implemented ranking functions described in chapter 3.

4.4.1 Evaluation metrics

- **Precision at n ($P@n$)** – measures the relevance of the top n positions of a ranked list; it represents the fraction of documents that are relevant in the top n positions, and is expressed as

$$P@n = \frac{\#\{\text{relevant docs in top } n \text{ results}\}}{n} \tag{17}$$

- **Mean Average Precision (MAP)** – is defined as the mean of the Average Precision (AP) over the set of all queries and can be expressed as

$$MAP = \frac{1}{|Q|} \sum_{q=1}^{Q} AP_q \tag{18}$$

where Q denotes the number of queries and AP_q is given by

$$AP_q = \frac{\sum_{n=1}^{N_q} P@n \ . \ rel(n)}{\#\{\text{total relevant docs for } q\}} \tag{19}$$

where N_q denotes the number of documents retrieved, and $rel(n)$ is 1 if the n-th document is relevant and 0 otherwise.

- **Geometric Mean Average Precision ($GMAP$)** – is defined as the geometric mean of the Average Precision (AP) over the set of all queries and can be expressed as

$$GMAP = \sqrt[|Q|]{\prod_{q=1}^{Q} AP_q} \tag{20}$$

- **Normalized Discount Cumulative Gain ($nDCG@n$)** – is a measure for evaluating the top n positions of a ranked list when there is multiple level relevance judgment. It is defined as follows

$$nDCG@n = Z_n \sum_{j=1}^{n} \frac{2^{r_j} - 1}{\log(1 + j)} \tag{21}$$

where r_j denotes the relevance level of the j-th document, and Z_n is a normalization factor chosen such that the perfect ordering gets an $nDCG@n$ of 1.

4.4.2 Evaluation data

Datasets play an important role in IR research, freeing researchers from the burden of creating their own test collection, and allowing them to focus on researching IR methods. In addition, standard datasets ease the comparison of results between different approaches and algorithms. In our experiments, we used the data from the TREC-9 track[5], which is a modified version of the OHSUMED dataset.

TREC-9 Filtering data (OHSUMED dataset)

The OHSUMED collection contains a subset of the MEDLINE database, a set of queries and relevance judgments. The document data is comprised of about 350,000 references from 270 journals covering the years from 1987 to 1991. Each reference contains the usual fields for bibliographic records, including title, authors and abstract. Abstracts longer than 250 words are truncated and about 1/3 of the documents lack an

[5] http://trec.nist.gov/data/t9_filtering.html

abstract and contain only the title. In addition, each document contains human-assigned indexing terms taken from the Medical Subject Headings (MeSH) vocabulary. The collection contains 101 queries generated by physicians in the course of patient care. In addition, there are absolute relevance judgments of documents with respect to the queries, generated by human assessors. The relevance judgments have three levels 0, 1 and 2 for, *not relevant*, *possibly relevant* and *definitely relevant*. The data used is split into a training set with all of the 1987 data, and a test set with the data of the period of 1988–1991. In addition, 63 queries (topics) were picked from the original 101 OHSUMED topics establishing the smaller OHSU set. These 63 queries were specifically picked in order to get at least two *definitely relevant* documents in the training set, for each query. In our experiments, we used the data in the test set and the OHSU set of queries.

MeSH vocabulary

The Medical Subject Headings (MeSH)[6] is a controlled vocabulary managed by the U.S. National Library of Medicine (NLM) compiled initially from multiple sources. It has 12 levels, structured hierarchically, establishing thesaurus-like relationships. MeSH is very useful for the NLM, which uses its descriptors to index millions of articles in the MEDLINE database, from about 5,400 biomedical journals. Therefore, it is also very useful for physicians and medical librarians, which can use the MeSH vocabulary, in their search activities to find relevant documents.

MeSH is not available in the SKOS format, but is distributed in other formats such as XML and simple text files. When considering the XML format, the structure of MeSH can be easily mapped onto SKOS relationships and converted to a SKOS thesaurus as described in Van Assem et al. [2].

4.4.3 Methodology

Our system contemplates two different scenarios. In the first scenario, which we called simply "MeSH", we perform query expansion using the MeSH SKOS thesaurus and using preferred and alternative labels. For each query term *t*, we collect all the concepts that have *t* in any of those labels plus the hidden label and add the labels to

6 http://www.nlm.nih.gov/mesh/

the query without boosts. In the second scenario, "MeSH-Boost", we add a manually set boost factor for each type of expansion: 0.5 for *PREF* and 0.5 also for *ALT* terms, effectively giving greater relevance to the original query terms.

The system was evaluated using the modified OHSUMED dataset from TREC-9. We use the average performance over the set of (63) queries provided to measure the overall performance of this approach. The baseline performance of the retrieval system corresponds to the results obtained by the system without the query expansion framework.

To evaluate our query expansion approach, we focused on two measures, precision at n ($P@n$), and $nDCG$ at n ($nDCG@n$). These metrics emphasize the relevancy of the top n results, in contrast to MAP, which considers all retrieved document. Some metrics depend on the definition of which levels of relevance judgements are considered relevant. We only considered relevant the "*definitely relevant*" relevance judgement level. In regards to the number of results retrieved, we limited it to the top 1000 documents for each query.

4.4.4 Results and Discussion

Initial validation and testing

The results of the system's retrieval performance benchmark at position 1, 3 and 10 are presented in Figure 20, Figure 21 and Figure 22. In these figures we compare the performance of the baseline against simple expansion with MeSH, and MeSH-Boost expansion using boosts of 0.5 for PREF terms and 0.5 also for ALT terms. Introducing boosts to the query expansion process has a positive effect in the ranking of the documents as can be seen by the increased MAP with MeSH-Boost. When looking at the increase in MAP, of 5.7% between the Baseline and MeSH-Boost, one must keep in mind that a even a small change like this requires a lot of effort, since this measure takes into account all retrieved documents (in our case the first 1000). The introduction of a boost of 0.5 is beneficial since it effectively makes the original query terms twice as important. The advantage is more noticeable in the $P@3$ and $nDCG@3$ benchmarks, therefore we chose to further analyze the MeSH-Boost scheme.

In Table. 5 we present the raw-change and percent-improvement in document retrieval for the top position in the MeSH-Boost scenario. We can see that our system provides 6.0% and 5.4% improvement for P and $nDCG$. When we consider the top 3 retrieved documents, there are still sizable improvements. However, while the P@3

value is improved by about 12.3%, $nDCG$ seems to take a small hit and does not improve as much. Finally, considering the top 10 results, both metrics see an improvement of 5.1%.

	@1		*@3*		*@10*	
	Precision	$nDCG$	*Precision*	$nDCG$	*Precision*	$nDCG$
Baseline	0.365	0.429	0.302	0.374	0.254	0.350
MeSH-Boost	0.387	0.452	0.339	0.406	0.268	0.368
Raw Change	0.022	0.023	0.037	0.032	0.014	0.018
Improvement	6.0 %	5.4 %	12.3 %	8.6 %	5.1 %	5.1 %

Table 5. Retrieval improvements MeSH-Boost.

These first results, not only validate the framework but also corroborate the soundness of the SKOS-powered query expansion intuiton. Reformulating the query by looking up alternative terms in open Web-based SKOS vocabularies and adding these additional terms to the query improves precision and nDCG. Furthermore, the initial intuition that the original terms should be weighted in a different way, to achieve better performance, is confirmed here by the fact that the results are improved when using the MeSH-Boost scheme. That is, the retrieval metrics improved further by simply halving the weight assigned by the retrieval model for each term that is introduced in the expansion process. In this case the *tf-idf* weight as implemented by Lucene. In the end, this effectively assigns more importance to the input terms in the original query.

In this experience, we learned that for terms that originate from expansions, setting the boost to a value that lessens their contributions to the total score of each document improves performance. However, the 0.5 boost used in the previous experience is just one of the boost values that improve performance, but it is not the optimal value.

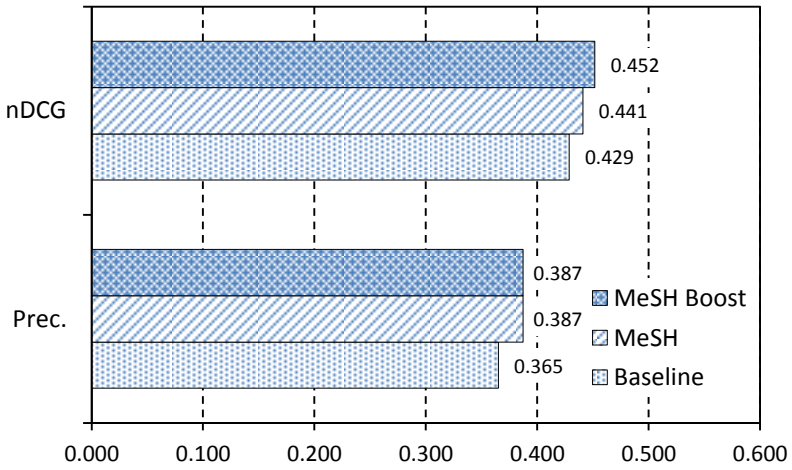


Figure 20. Expansion scheme comparison @1

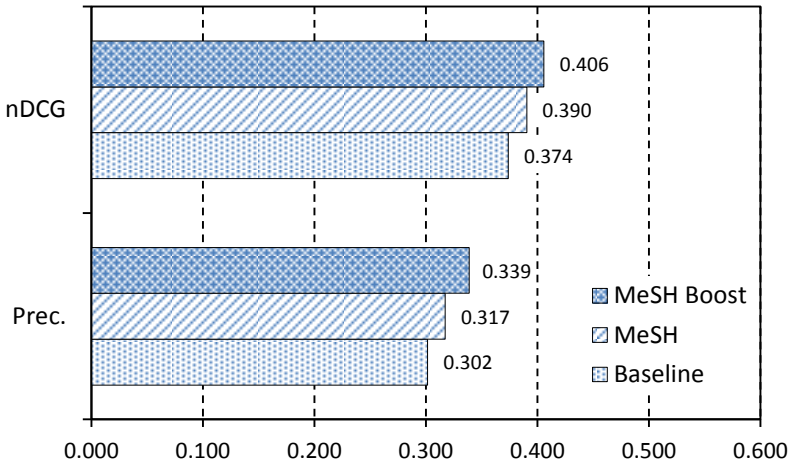


Figure 21. Expansion scheme comparison @3

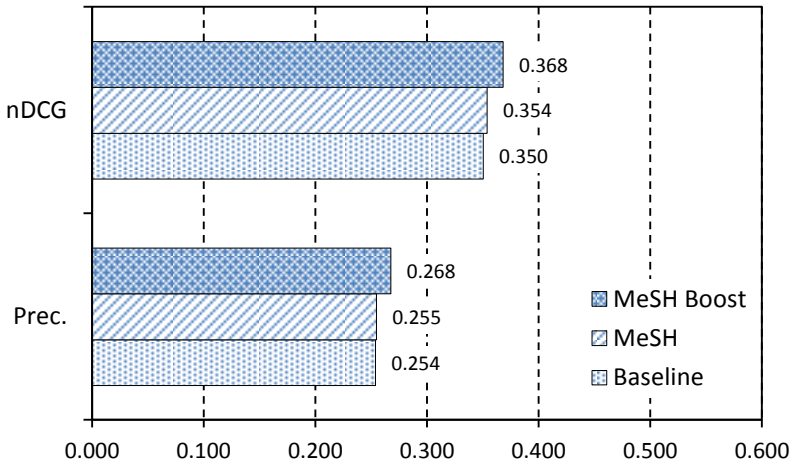


Figure 22. Expansion scheme comparison @10

Analyzing the influence of boost weights

In the following set of experiments the influence of different boost values is examined. The boost variables balance the contributions of the original query terms and the expanded terms to the final ranking computation. In the implemented framework, the boost values can be tuned independently for each expansion type. An optimal setup for a specific retrieval metric is therefore a combination of boost values that maximize it.

The choice of retrieval model should also play a role in achieving the best performance from this query expansion framework. Different retrieval models reach different performance levels, and the optimal boost values are specific to each. Furthermore, there could be retrieval models that are more robust and are able to maintain good retrieval performance in larger ranges of boost values, which is not only desired for more easily finding the optimal values when using a test set, but is also very useful to find a base setup: a combination of retrieval model and corresponding boost values that can blindly offer good results when there is no test set available.

In order to assess the framework and find higher performance gains we performed a series of benchmarks and the results are presented in the next few pages. In these tests we restricted the expansion types to PREF and ALT terms. The tests measure the performance of the system for each boost value in a range from 0.0 to 2.0 in 0.05 steps. The metrics presented are precision and *nDCG* (at 1, 3 and 10), MAP and GMAP.

With this evaluation, we can easily compare the performance of the *tf-idf* and BM25 models and the variations implemented LTC, BM25L and BM25+ as described in Chapter 3.

In our tests, the best results for P@1 is 0.500 with BM25 and a boost value of 0.6. The second best is 0.468 with BM25L and a boost of 0.55. The other functions have a lower P@1 value close to 0.410. The best boost values seem to be in the range of 0.5 to 0.65.

Both BM25L and BM25+ reached a maximum for P@3 of 0.382. This value is obtained with a boost in the range 0.35-0.4 for BM25L and 0.25-0.35 for BM25+. BM25 and LTC managed to reach P@3 of 0.366, while tf-idf delivered the lowest result with 0.339.

P@10 is best with BM25L and BM25+, which reach 0.305 and 0.302 respectively. In Figure 25, we can easily see that all functions start giving good results with a boost of 0.3 forward. Then there is a constant decline of precision in the BM25-based functions when the boost value passes 0.8 and around the 0.4 mark for tf-idf-based functions.

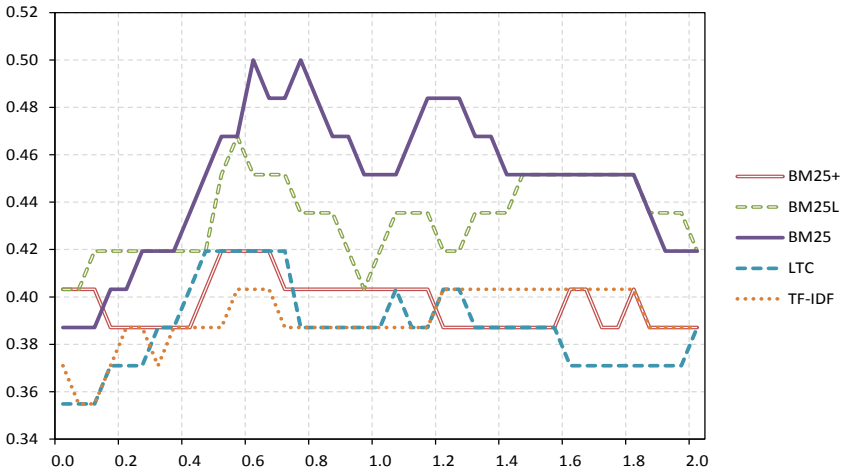


Figure 23. Optimal boost for P@1

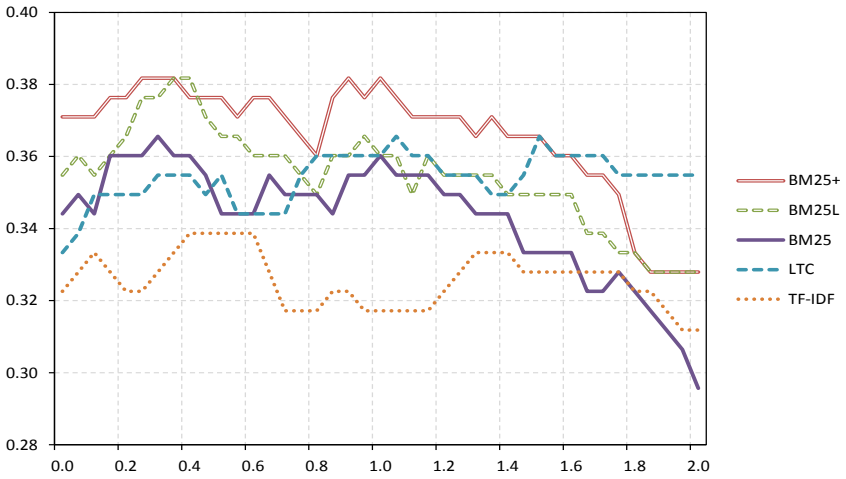


Figure 24. Optimal boost for P@3

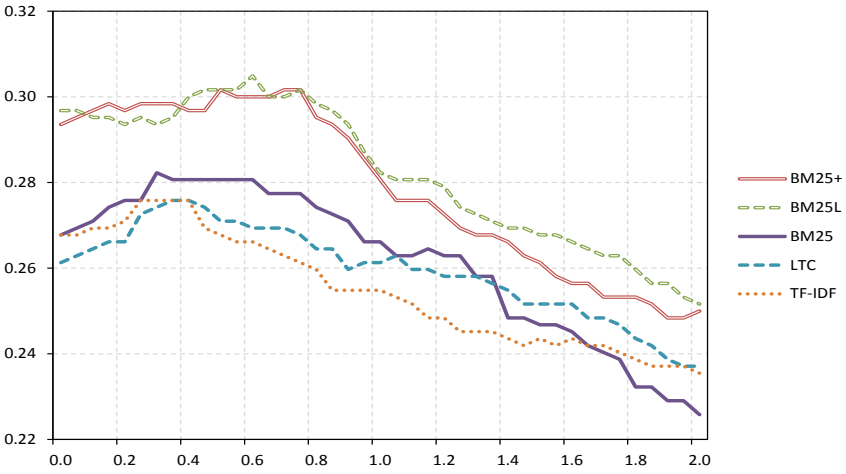


Figure 25. Optimal boost for P@10

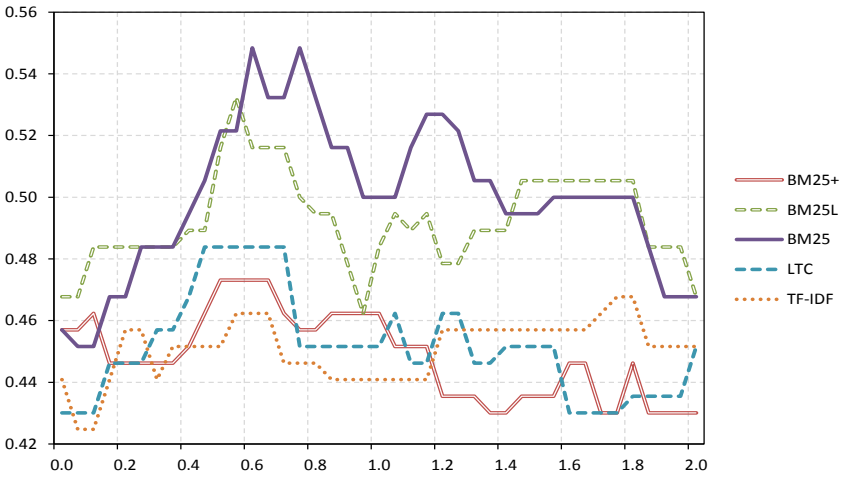


Figure 26. Optimal boost for nDCG@1

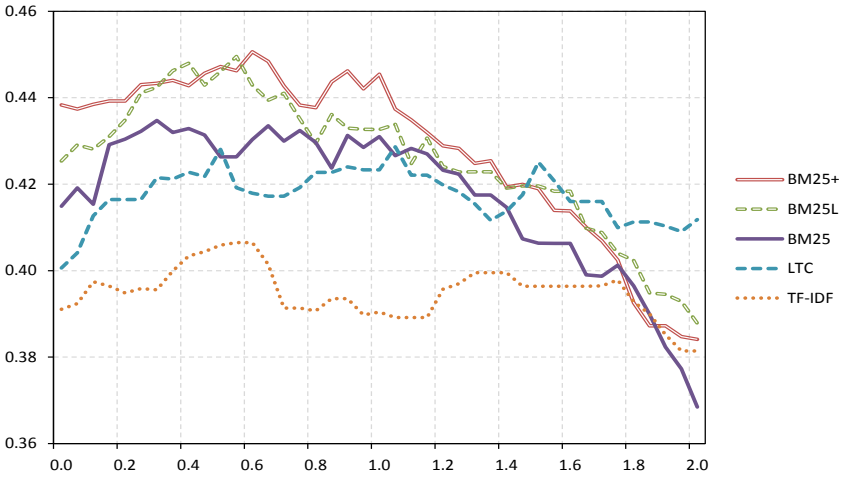


Figure 27. Optimal boost for nDCG@3

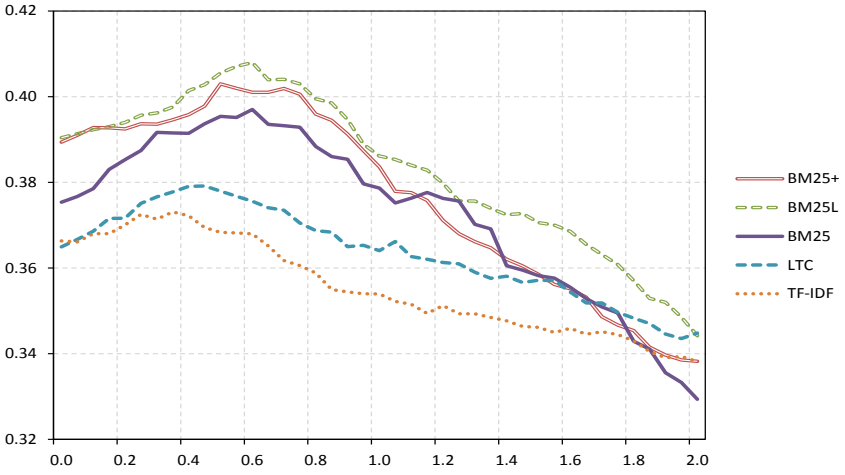


Figure 28. Optimal boost for nDCG@10

With nDCG@1, the BM25+ function disappoints with a lower performance than with LTC. Again, as with P@1, BM25 takes the first place with 0.548 with a boost of 0.6 and BM25L follows with 0.532 with a boost of 0.55. As seen in Figure 26, the functions seem to get a better result with a boost in the interval 0.5-0.65.

Again, as with P@3, the best functions for nDCG@3, with expansion, are BM25L and BM25+ reaching 0.450 and 0.451 for a boost of 0.55 and 0.6 respectively. In Figure 27, LTC is competitive with regular BM25 and does not suffer from the steady decline seen in the higher boosts for the BM25-based functions. The best results with LTC are 0.429 with a boost of 1.05, 0.428 with a boost of 0.5 and 0.425 at a 1.5 boost.

When measuring nDCG@10 the lines in Figure 28 smooth out and allow us to see that BM25L marginally edges out BM25+. BM25L and BM25+ reach nDCG@10 of 0.408 and 0.403, with a boost of 0.6 and 0.5 for respectively. The performance of the BM25-based functions starts to decline for boost values greater than 0.7, and a bit earlier for the tf-idf-based functions.

Overall, while BM25 leads the performance of P@1 and nDCG@1, by a considerable margin, BM25L is the best function because it offers superior performance at positions 3 and 10 and got the second best performance at position 1. The best performance with BM25+ is in P@3 and nDCG@3, where it maintains the best results for in most boost values. However, the results for the first position are very weak. With LTC, the performance seems to vary less than the other functions for different boost values, even outperforming the BM25-based functions for larger boosts. This makes it a good option when building a system for good performance at 3.

In the end, the best function overall seems to be BM25L with the best boost around the values of 0.55 and 0.6. This boost value results in a slightly higher than half factor in the scoring function, which means expansions are weighted at around 55%.

Evaluating the influence of boost for MAP

Proceeding with the evaluation of the expansion framework, we present the progression of MAP for boost values in the interval of 0.0 to 2.0 in 0.05 steps like with did with precision and nDCG. MAP differences are usually a very good indication of the improvement brought by a new IR method. Small differences can mean high gains since this metric takes into account all of the documents returned by a search. GMAP [21] is presented as well because literature favors it in query expansion research.

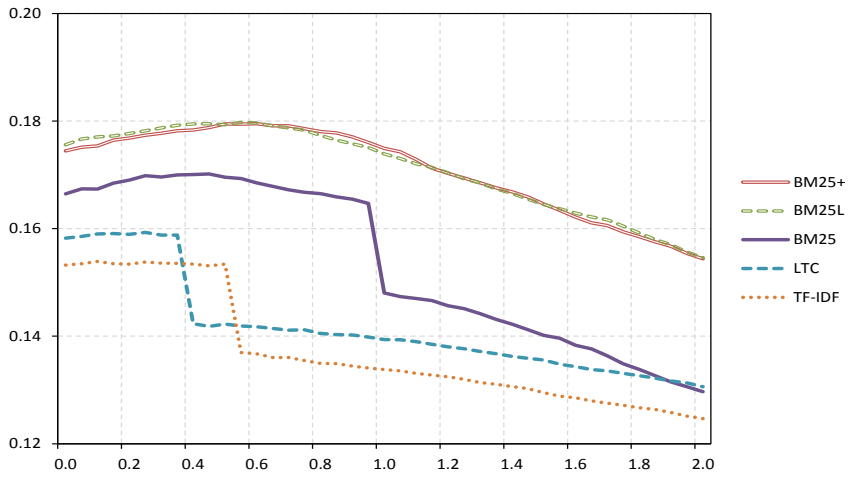


Figure 29. Optimal boost for MAP

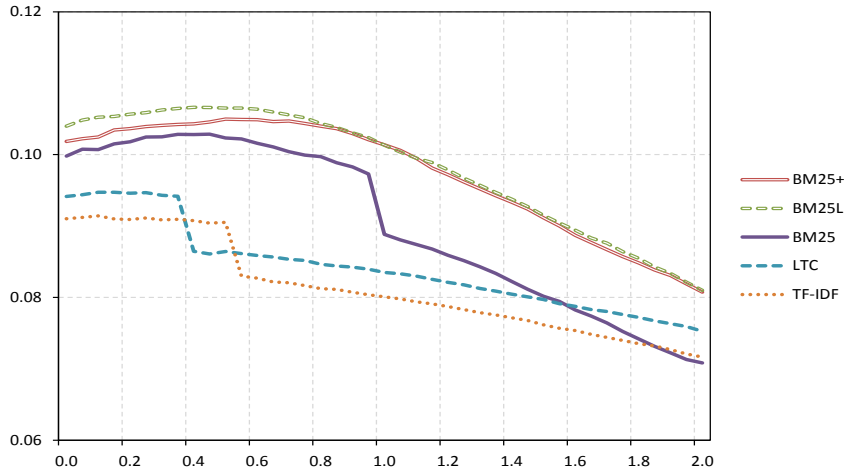


Figure 30. Optimal boost for GMAP

In Figure 29 and Figure 30, the first thing that is noticeable is the large performance degradation common to three of the retrieval functions at a specific boost value for each. BM25L and BM25+ do not suffer from this problem, resulting in a smooth curve. Furthermore, BM25L and BM25+ are the functions with the best MAP, with a value of 0.18 and 0.179 for a boost of 0.6, making them the best choice to use with the framework.

In the GMAP chart presented in Figure 30 the difference between the BM25L and BM25+ curves means that BM25L is marginally better at overcoming the mismatch between the original query and the information need, when associated with the query expansion framework.

4.5 Summary and discussion

The proposed query expansion framework makes use of Simple Knowledge Organization System (SKOS) thesauri. By finding semantically related words for each query clause, and adding these as additional query terms, the system is able to overcome mismatches between the users' vocabulary and the indexed data gracefully.

Evaluation on the OHSUMED dataset demonstrated the ability of the system to improve on the baseline Lucene retrieval metrics. The improvements are across the board, over all the evaluated retrieval metrics. Weighting the expanded terms in a different way, depending on the type of SKOS semantic relation, seems to provide further improvement of the search quality. As future work, we intend to improve the efficiency of the system by replacing our SKOS engine with a high performance triple store to improve the efficiency of the analyzer.

5

Search interface with guided query expansion

New Information Retrieval research is always being incorporated into search engines leading to improvements, sometimes sizable, in retrieval performance. Modern Web search engines are able to find the results for the exact information need of the users. This power is so ingrained into our minds that a powerful search engine's name has even become synonym with the general concept of search.

In chapter 4 we presented an advanced automatic query expansion mechanism, that makes use of open Web-based vocabularies to narrow the gap between the users' queries and their information needs. The other side of the coin, however, is that a retrieval system relies on users to set the retrieval goal. Several studies have concluded that users have problems formulating effective queries. Users tend to write queries that are either short or too general, hampering the search engine ability to deliver the best results due to lack of context. For instance, in the Web context, regardless of the domain, queries tend to have on average 2 or 3 query terms only, since users expect that the search engine will extrapolate the right information need from ambiguous and incomplete queries.

This mismatch between the capabilities of current search engines and the users' expectations that the search engine will be able to infer or predict the information need, from ill-formulated queries alone, is aggravated in non-Web search engines. This gap is widened further in information retrieval systems that support the search on document collections of the more technical domains, because users tend to have a limited knowledge of the domain-specific vocabulary used for instance in journals of Medicine, legislation documents and others. This limitation of the users leads to ill-formulated queries that are short and do not use the best terms that will lead to the

desired results. For instance, Health information retrieval in Internet accessible knowledge bases, are accessed by millions of normal users that are untrained to use them, leading to the construction of simplistic or misleading queries, such as a general term "antidepressants" or one common symptom such as "fever" [34].

Query formulation rises as a major area that is in need of improvement in information retrieval, and some aids have been integrated in Web search. An example of query formulation aid is the "Did You Mean" (DYM) feature of Web search engines that corrects queries' misspellings and suggests a better query for a number of information needs as well. Autocomplete features are also common in search engines, making suggestions as-we-type of popular search queries obtained through query log analysis. Interactive query expansion (IQE) is another way to assist people in query formulation. Like the (DYM) feature, users are only aided after having formulated and issued an initial query, but gain the ability to adjust the query with alternative terms made available to select as a recommendation by the system and related to the initial query. This tool can assist users reduce the number of interactions required to complete a search task, however, it is not optimal. Users still have to provide an initial query to get to a query-building interface that allows them to explore available concepts, and just then construct queries that are more accurate.

To help users articulate better their information needs, we have developed a novel system that moves IQE into the search box allowing users to formulate queries in one sweep. This is implemented like the autocomplete feature in a way such that the autocomplete suggestions are actually the labels for concepts in a specific SKOS thesaurus. With the interface users can find out what are the preferred terms in a certain domain vocabulary, choose from recommended terms and, when combined with the work from chapter 4, discover the automatic query expansions that the system will perform. This makes the choice of terms and informed one, allowing the user to decide which terms to include in the query and therefore to decide the expansions also.

5.1 Search Interface Implementation

This interface was implemented in two parts: backend and frontend. For the backend we used a Java REST web service to expose two resources. One resource for obtaining suggestions given a string and another for fetching the expansions of a given term. To implement this service we use the REST API building framework, Dropwizard, which contains a metrics module and allows inspecting the response times of re-

quests which are very important in the user experience. The metrics output for a run of a number of queries on the suggestion resource can be seen below. In this, we can see that this component is able to send a response on average in 27 milliseconds. The frontend is set so that the user receives instant feedback so this value is very well within the requirements.


```
  },
  "pt.unl.fct.di.suggestskos.resources.SuggestResource" : {
    "getSuggestions" : {
      "type" : "timer",
      "duration" : {
        "unit" : "milliseconds",
        "min" : 6.406835,
        "max" : 152.970185,
        "mean" : 27.351427818181815,
        "std_dev" : 24.303891287373723,
        "median" : 22.145276000000003,
        "p75" : 31.542952,
        "p95" : 60.9272845,
        "p98" : 152.970185,
        "p99" : 152.970185,
        "p999" : 152.970185
      },
      "rate" : {
        "unit" : "seconds",
        "count" : 44,
        "mean" : 0.06854489365365439,
        "m1" : 5.227524847840772E-5,
        "m5" : 0.02159673844870502,
        "m15" : 0.025808210978855068
      }
    }
  }
}
```


Figure 31. Autocomplete requests performance metrics.

We used jQuery-Autosuggest in the frontend and modified it heavily to support our backend and expansion feature. The interface presents an ordinary search box, but when the user starts to type, requests are sent to the backend 150ms after the last keystroke, triggering an autocomplete suggestion obtained from the backend service. To allow the users to decide if the suggestion is valuable, we show a popup window with the terms that will be included in the query, by that term's expansion. Users can also enter terms that are not in the vocabulary and these will not be expanded.

5.2 Guided query expansion

Using the Eurovoc (European Parliament) thesaurus, which has labels in 22 languages for the concepts contained, we experimented with cross-language expansion. In the figure below we can see the user typing the word fragment "spir" and getting a suggestion for "spirits" a concept which in the case of this thesaurus expands to a

number of different types of spirits. In addition, in this case, the expansion are in both English and Portuguese, and the interface recognizes either one.

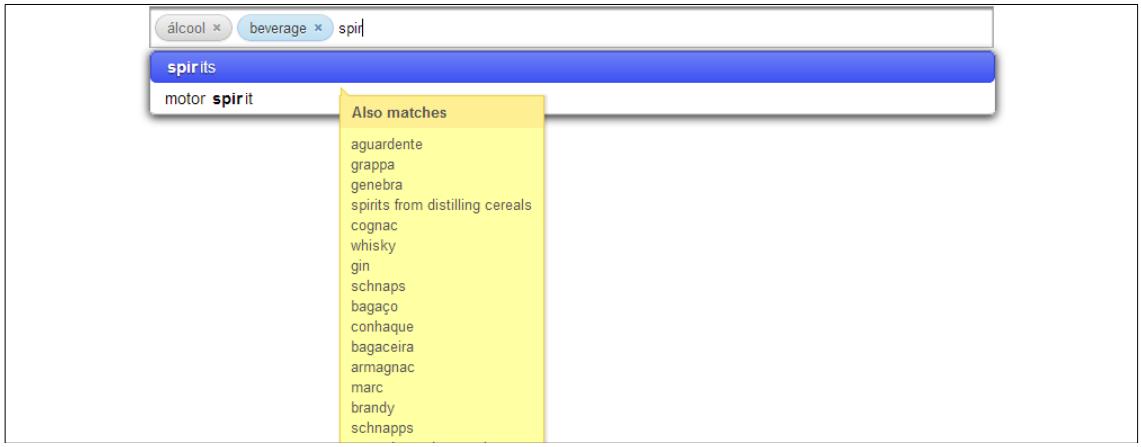


Figure 32. Guided expansion cross-language example.

In this figure, the MeSH Medicine vocabulary is used, which contains only English labels. This example search illustrates the benefits of this approach in this specific domain, where users usually have difficulties and the full recall is often necessary.

In this specific case, the user is searching for drug interactions that could induce acute kidney failure. This user is especially interested in the contribution of "acetaminophen", but is only aware of the alternate name "paracetamol". The system does not suggest the term "paracetamol", when the user enters a fragment of this word, because it is not the preferred term. This should have the effect of training users to use the better query terms. However, when the user enters the full name, the concept is identified and expanded to the various forms.

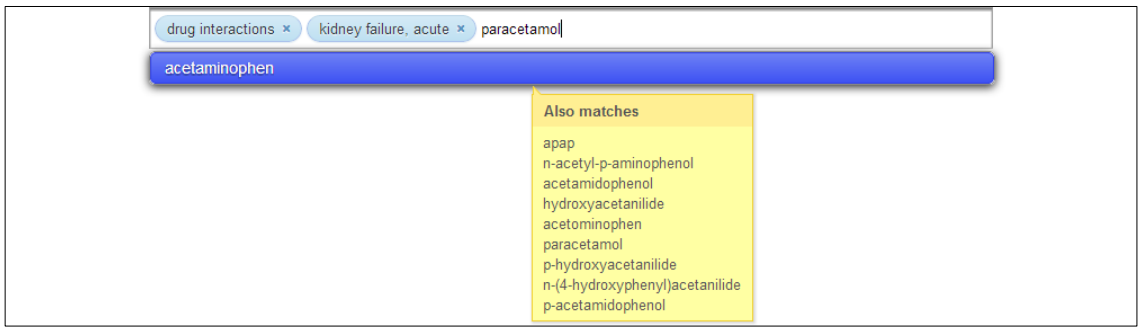


Figure 33. Guided expansion medical use-case example.

This type of system is not only useful for the layman but also for the highly skilled technical staff that are interested in getting the best and the most complete results, to

avoid complications due to poor recall. Another area that has similar requirements is law, in its various forms, such as case law, and patents which are areas in which it is vital to have the most complete set of results to gain advantage.

To allow matching word fragments with concept labels we index the SKOS thesaurus labels using the analysis in Figure 34. The last filter EdgeNGramTokenFilter is what generates the tokens for these word fragments, usually also called *grams*.

```
        final StandardTokenizer src = new StandardTokenizer(matchVersion,
            reader);
        TokenStream tok = new StandardFilter(matchVersion, src);
        tok = new StandardFilter(matchVersion, tok);
        tok = new LowerCaseFilter(matchVersion, tok);
        tok = new ASCIIFoldingFilter(tok);
        tok = new EdgeNGramTokenFilter(tok, Side.FRONT, 1, 20);
        return new TokenStreamComponents(src, tok);
```

Figure 34. Guided expansion analyzer.

5.3 Hacker Search demo application

We built a Web application to demonstrate the guided query expansion technology at Codebits 2012. Codebits is the biggest Portuguese event for programmers, technologists and hardware geeks, that is organized annually by SAPO from the Portugal Telecom group. Every year, speakers from all over the world come to speak at the event or give workshop classes.

The main activity in this event is a 48 hour programming contest. In this contest a jury awards prizes to the best projects after a final presentation session. In addition, prizes are awarded to the most popular projects by public vote. Projects usually fall in these categories: robotics/hardware, automation, web applications, video games, open data and API mashups and sometimes even software libraries. Teams that come up with clever business ideas are often offered mentoring through an incubation initiative called Codebits Labs.

To apply to this event all the participants create an account at the event's site and fill out their profile with simple personal information. In addition, to be eligible for the contest they also have to fill out a biography field and choose their skills from a static set of skills. Finally, all the profiles go through a reviewing process and the users with the best profiles get accepted to the event. The information about the participants is accessible through the site's search feature, but that only supports finding participants by username and filtering by a static set of skills.

In this year's edition our research group was offered a booth at Codebits, and event participants played with demos that demonstrated the research developed in the past year. The methods presented in this chapter and chapter 4 and were demonstrated through a new customized Web application we named HackerSearch. This application allowed participants to search for other participants with the required expertise for their projects and thus facilitated the staffing of teams. This Web application is particularly useful in the context of this programming challenge, which is time-limited, where not finding productive people for a certain project can lead to the project's demise. The data used in this demo was fetched using the API from the Codebits main site, which already had all the information about the accepted participants in the event.

We decided to do a limited evaluation of this demo application to take advantage of the large number of participants in the event, which were potential users of the application. Therefore, in addition of having the application available at our booth, we hosted the application online and collected Web analytics information about the usage of the site. By making available two unlabelled options (see Figure 35), with and without guided query expansion, we wanted to see what option users at the event preferred.

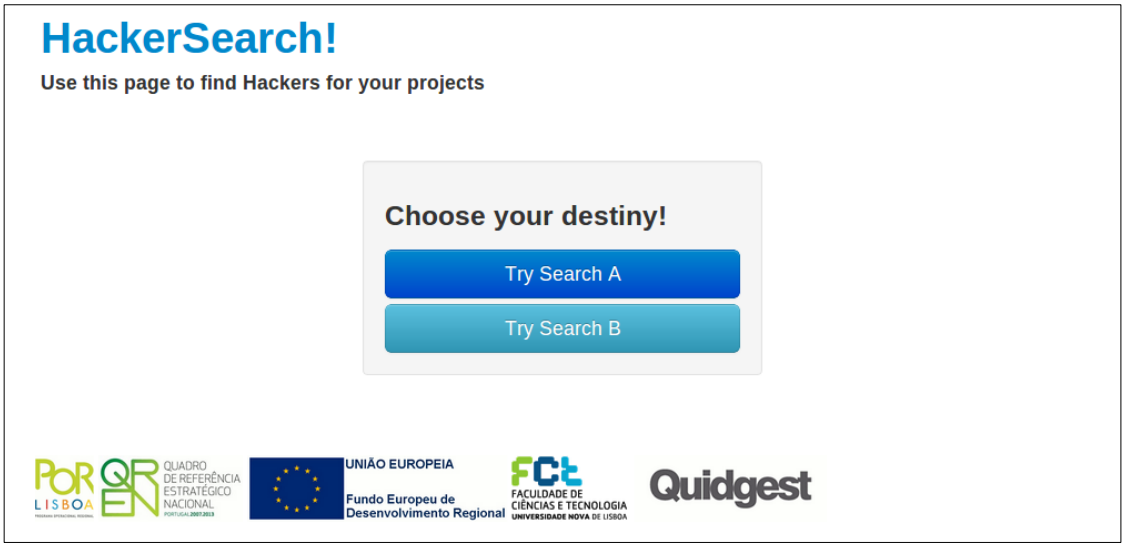


Figure 35. Hacker Search home page.

5.3.1 Search

For HackerSearch we created our own small custom SKOS thesaurus with concepts that correspond loosely to the static skill set in the Codebits site: PHP, Perl, Ruby, Python, Erlang, C/C++, Cocoa/objC, NET/Mono, Java, Javascript, CSS, APIs, Web, Embedded, Hardware, Microformats, Security and others. For each concept in our thesau-

rus, the approach was to add alternative labels so that searches for alternative terms would map to a main skill. For instance, in Figure 36 the definition of a SKOS concept from our thesaurus with the preferred label "Security" is pictured. We equate terms like "Hacking" and "Cryptography" with the main skill "Security".

```
<!--     http://www.codebits.eu/ontologies/2012/10/hackers.owl#security     -->
<skos:Concept rdf:about="&hackers;security">
     <rdf:type rdf:resource="&owl;NamedIndividual"/>
     <skos:altLabel>Cryptography</skos:altLabel>
     <skos:altLabel>Phreaking</skos:altLabel>
     <skos:altLabel>Hacking</skos:altLabel>
     <skos:altLabel>Exploit</skos:altLabel>
     <skos:altLabel>Vulnerability</skos:altLabel>
     <skos:altLabel>Cracking</skos:altLabel>
     <skos:prefLabel>Security</skos:prefLabel>
</skos:Concept>
```

Figure 36. Security concept: excerpt of hackers.rdf.

In Figure 37, we can see the result of applying this SKOS thesaurus to the search in the demo application with guided query expansion. When the user enters the word "design" the system suggests 3 main skills: database design, CSS and HTML/CSS.

Figure 37. Hacker Search with guided expansion.

In this case, the user was initially interested in "Web design", which is one of the matches in the tooltip when hovering the "CSS" suggestion. This interface guides the user, informs it in regards with the mapping between technologies (CSS) and needs (Web design). In addition, the system alerts the user for the fact that a search for the term "CSS" will search additionally for "web design" and "html".

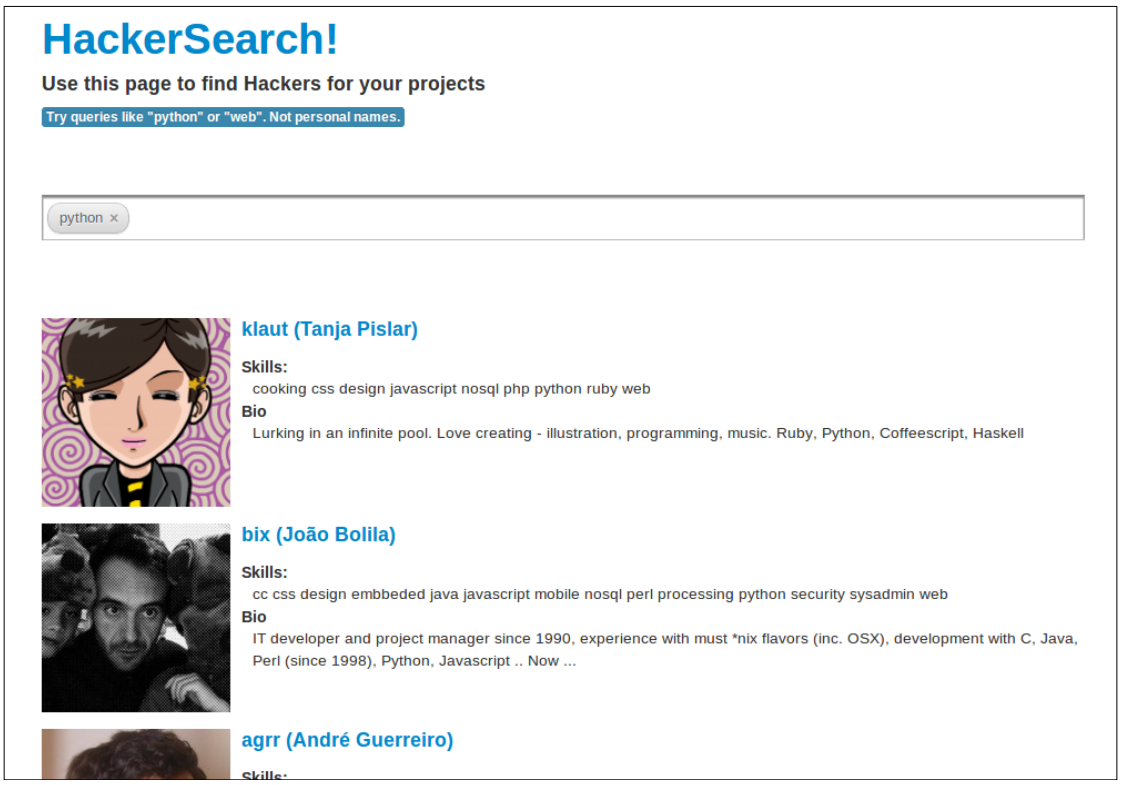


Figure 38. Hacker Search results page for Python.

The page of results for the query "python" is picture in Figure 38. The results list shows only the 10 users with better ranking. Even though we only query the bio, we can see in the picture that the top ranked users have checked the Python checkbox, indicating themselves that they are good at Python. In this case, the top user ranked highest because she lists Python in her bio, but she also listed Coffeescript, which is a Python inspired language that compiles to Javascript.

This demo uses Solr as the backend for searching and as a user database also. All the information about a user is indexed and/or stored. In addition, to better score the documents we index the biography of each user in two different fields, one field does simple English analysis while the other adds a *SKOSLabelFilter* (see section 4.3.2), which configures it to perform expansion of concepts at query-time. Having two different fields allows us to configure Solr's query parser to weight each field differently. Since we were using the BM25+ similarity, we decided to weight the expansions at 35%.

5.3.2 Similar Users

This application provides another feature that takes advantage of expansion using the SKOS thesaurus. When a user clicks through a search result to open it, they are tak-

en to a parallel demonstration that we called HackerBro. Instead of showing only the profile for the user selected we present a page with similar users to the one selected.

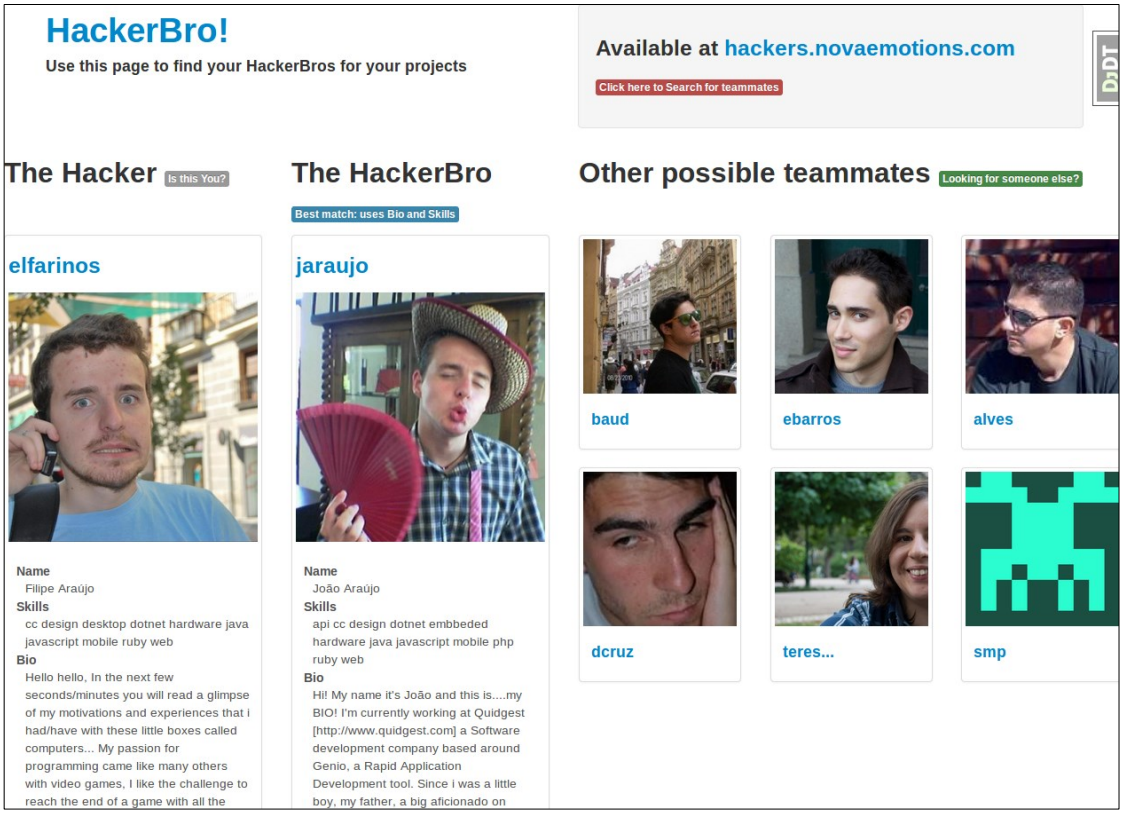


Figure 39. Hacker Brother feature.

As can be seen in the example in Figure 39, the page for the user "elfarinos" shows the information about the user requested but also shows the information of a second user "jaraujo", which is the most similar participant at the event. This example is interesting because these two event participants are actually twin brothers with a similar skill set and the list of possible teammates, which is just the other top ranked users, happen to be people they know from university and work.

This feature showed the possibility of using SKOS-based query expansion in combination with relevance feedback (*MoreLikeThis*), however we can't offer hard evaluation data since we do not have judgments for this dataset. Due to the way *MoreLikeThis* works, we had to do the expansion at index time. In this case, we weighted expansions at 25%. In addition, in contrast with the search feature we also considered here the skills that the participants selected in their profiles, weighting them at 10%.

5.3.3 Analysis of Web analytics

Instead of asking users to execute pre-determined search tasks and answer a survey, we decided to evaluate this demo application using the data gathered using Google Web Analytics. During the event, 92 unique visitors (approximately 12% of the event participants) visited the site 176 times and generated 1676 page views. About 47.73% (84) visits were from returning visitors. Approximately 25% of the user navigation sessions have a page depth of more than eight pages, a sign of user engagement. The HackerBro feature accounts for 55% (925) of all page views, which might be a sign that users preferred hopping from similar user to similar user, since search page views only account to 20% page views. In regards to search, the distribution of page views is 8.05% (135), for the query-expansion enabled search, and 5.43% (91) for the simple search without expansion. The time on page for the guided expansion is 16 seconds, in contrast with the 46 seconds needed on average in the non-guided option.

5.4 Summary and discussion

This chapter presented a prototype to demonstrate the usefulness of SKOS vocabularies in retrieval activities. Having used in the previous chapter the SKOS vocabularies for automatic query expansion, we found the need to make this process more transparent to the user. Therefore, we combined the autocomplete feature with data sourced from SKOS thesauri to aid users in the process of query formulation.

The system allows inline expansion of SKOS labels and suggests the preferred terms for the search. In addition, the user can inspect the alternate labels for term added to the query, or before adding a concept to the query and decide which concept is closer to the information need.

In addition, we developed a real world application and demonstrated it at a relatively big venue, Codebits 2012. We integrated in the demo all the methods described in this chapter in association with the other research described in this thesis.

6

Conclusions

The aim of this thesis was the exploration of open Web-based SKOS vocabularies to improve search engines properties in the context of document management systems. We anticipated that the use of these resources could translate into an increase of recall and other retrieval performance metrics and also user satisfaction.

This work is part of a bigger framework, which posed some integration challenges that we were able to overcome and sometimes turn into our advantage. We took the Genio's own generation capabilities and used them for deploying, configuration of the search system. Removing the burden of configuring data import configurations for the possibly various search cores of an application produced by Genio. In the end, the system developed respects the requirements of rapid development methodologies allowing for rapid prototyping of search engines functionality through automatic generation from given specifications. The performance gains of this solution in comparison with the solution it replaces are astounding, especially in the query response time aspect. In addition, search quality is very much improved with specialized analysis processes for different languages and new retrieval functions providing the most impact.

Finally, the work developed with SKOS thesauri, further improves the search system capabilities. This is done in two ways: by using domain-specific vocabularies with corresponding collections, to automatically add new related terms to a search query, with a query expansion module; and by using the same data to mirror the automatic expansion to help the user formulate the queries and understand the final query meaning. Future work could encompass the use of Linked Data sets, and the exploration of learning-to-rank techniques to learn the best way to weight expanded terms, as well as using multiple information sources for expansion.

References

[1] Aronson, A.R. and Rindflesch, T.C. 1997. Query expansion using the UMLS Metathesaurus. *Proceedings : a conference of the American Medical Informatics Association / ... AMIA Annual Fall Symposium. AMIA Fall Symposium*. (Jan. 1997), 485–9.

[2] Van Assem, M. et al. 2006. A Method to Convert Thesauri to SKOS. *The Semantic Web Research and Applications*. 4011, c (2006), 95–109.

[3] Deerwester, S. et al. 1990. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*. 41, 6 (1990), 391 – 407.

[4] Eichmann, D. et al. 1998. Cross-language information retrieval with the UMLS metathesaurus. *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '98* (New York, New York, USA, Aug. 1998), 72–80.

[5] Hersh, W. et al. 2000. Assessing thesaurus-based query expansion using the UMLS Metathesaurus. *Proceedings / AMIA ... Annual Symposium. AMIA Symposium*. (Jan. 2000), 344–8.

[6] Hersh, W. et al. 1994. OHSUMED: an interactive retrieval evaluation and new large test collection for research. (Aug. 1994), 192–201.

[7] Jin, S. et al. 2009. Query expansion based on folksonomy tag co-occurrence analysis. *2009 IEEE International Conference on Granular Computing* (Aug. 2009), 300–305.

[8] Lin, J. and Demner-Fushman, D. 2006. The role of knowledge in conceptual retrieval. *Proceedings of the 29th annual international ACM SIGIR conference on*

Research and development in information retrieval - SIGIR '06 (New York, New York, USA, Aug. 2006), 99.

[9] Liu, T.-Y. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends® in Information Retrieval*. 3, 3 (2009), 225–331.

[10] Lv, Y. and Zhai, C. 2011. Lower-bounding term frequency normalization. *Proceedings of the 20th ACM international conference on Information and knowledge management - CIKM '11* (New York, New York, USA, Oct. 2011), 7.

[11] Lv, Y. and Zhai, C. 2011. When documents are very long, BM25 fails! *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information - SIGIR '11* (New York, New York, USA, Jul. 2011), 1103.

[12] Mandala, R. et al. 1999. Combining multiple evidence from different types of thesaurus for query expansion. *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '99* (New York, New York, USA, Aug. 1999), 191–197.

[13] Mandala, R. et al. 2000. Query expansion using heterogeneous thesauri. *Information Processing & Management*. 36, 3 (May. 2000), 361–378.

[14] Manning, C.D. et al. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[15] Metzler, D. and Croft, W.B. 2005. A Markov random field model for term dependencies. *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '05* (New York, New York, USA, Aug. 2005), 472.

[16] Miles, A. et al. 2005. SKOS Core: Simple knowledge organisation for the Web. *International Conference on Dublin Core and Metadata Applications*.

[17] Mitra, M. et al. 1998. Improving automatic query expansion. *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '98* (New York, New York, USA, Aug. 1998), 206–214.

[18] Nemeth, Y. et al. 2004. Evaluation of the real and perceived value of automatic and interactive query expansion. *Proceedings of the 27th annual international conference on Research and development in information retrieval - SIGIR '04* (New York, New York, USA, Jul. 2004), 526.

[19] Qin, T. et al. 2010. LETOR: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*. 13, 4 (Jan. 2010), 346–374.

[20] Qiu, Y. and Frei, H.-P. 1993. Concept based query expansion. *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '93* (New York, New York, USA, Jul. 1993), 160–169.

[21] Robertson, S. 2006. On GMAP. *Proceedings of the 15th ACM international conference on Information and knowledge management - CIKM '06* (New York, New York, USA, Nov. 2006), 78.

[22] Robertson, S.E. et al. 1994. Okapi at TREC-3. *Proceedings of the Third Text REtrieval Conference TREC94* (1994), 109–126.

[23] Robertson, S.E. and Walker, S. 1994. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. (Aug. 1994), 232–241.

[24] Salton, G. and Buckley, C. 1990. Improving retrieval performance by relevance feedback. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*. 41, (1990), 288 – 297.

[25] Singhal, A. et al. 1996. Pivoted document length normalization. *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '96* (New York, New York, USA, Aug. 1996), 21–29.

[26] SKOS Simple Knowledge Organization System Reference: 2008. *http://www.w3.org/TR/skos-reference/*.

[27] Tan, B. et al. 2007. Term feedback for information retrieval with language models. *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07* (New York, New York, USA, Jul. 2007), 263.

[28] Theobald, M. et al. 2005. Efficient and self-tuning incremental query expansion for top-k query processing. *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '05* (New York, New York, USA, Aug. 2005), 242.

[29] Voorhees, E.M. 1994. Query expansion using lexical-semantic relations. (Aug. 1994), 61–69.

[30] Wollersheim, D. and Rahayu, J.W. 2005. Ontology based query expansion framework for use in medical information systems. *International Journal of Web Information Systems*. 1, 2 (Jan. 2005), 101–115.

[31] Xia, F. et al. 2008. Listwise approach to learning to rank. *Proceedings of the 25th international conference on Machine learning - ICML '08* (New York, New York, USA, Jul. 2008), 1192–1199.

[32] Xu, J. and Croft, W.B. 1996. Query expansion using local and global document analysis. *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '96* (New York, New York, USA, Aug. 1996), 4–11.

[33] Zazo, Á.F. et al. 2005. Reformulation of queries using similarity thesauri. *Information Processing & Management*. 41, 5 (Sep. 2005), 1163–1173.

[34] Zeng, Q.T. et al. Assisting consumer health information retrieval with query recommendations. *Journal of the American Medical Informatics Association : JAMIA*. 13, 1, 80–90.

[35] Zhou, W. et al. 2007. Knowledge-intensive conceptual retrieval and passage extraction of biomedical literature. *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07* (New York, New York, USA, Jul. 2007), 655.