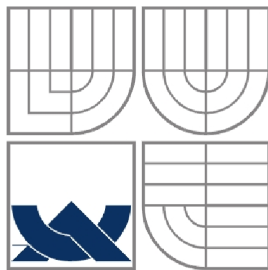
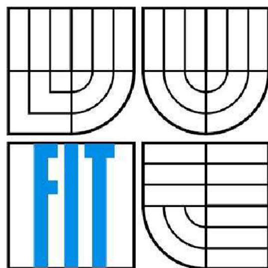


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

Hledání souvisejících dokumentů na webu

Similarity Search in Document Collections

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Dimitar Jordanov

VEDOUCÍ PRÁCE
SUPERVISOR
RNDr. Pavel Smrž, Ph.D

Brno 2009

Zadání

Abstract

The main objective of this work is to estimate the efficiency of the available software for similarity search in document collections and on two in particular, Semantic Vectors and Lecene's class MoreLikeThis. The paper provides a comparison of those two approaches and introduces methods that can lead to improving the quality of the results generated by a search.

Keywords

Semantic Vector , Random Projection algorithm, Apache Lucene, Matching technologies, Natural Language Processing, Text Clustering, MoreLikeThis.

Declaration

I confirm that this final project is my own work, except where I have explicitly indicated otherwise. The list of all sources I have used in the process of creating this work are provided at the very end of the paper.

Acknowledgments

I would like to thank my supervisor, doc. RNDr. Pavel Smrž, Ph.D for the, encouragement, the patient guidance and advices that he provided throughout the time I was preparing this work. I have been extremely lucky to have a supervisor who cared so much about my work, and who was responding to my questions and queries so effectively.

Brno, 26 May 2009

…………………………
Bc. Dimitar Jordanov

Contents

1. Introduction ... 11
2. Motivation ... 13
3. Information retrieval and text mining ... 15
3.1. Document indexing. ... 15
3.1.1. Document linearization ... 15
3.1.2. Filtration ... 15
3.1.3. Stemming ... 16
3.1.4. Weighting ... 16
3.2. Term frequency and Inverse document frequency ... 16
3.3. Inverted index ... 17
3.4. Cosine-distance ratio ... 17
4. Apache Lucene ... 19
4.1. Fundamental Lucene's classes for indexing text ... 19
4.2. Lucene's index structure ... 20
4.3. Searching related documents with Lucene ... 21
5. Semantic vectors ... 25
5.1. Wordspace model ... 25
5.2. Probabilistic dimension reduction methods ... 25
5.3. Software architecture ... 27
6. Similarity search with Semantic vectors ... 29
6.1. Search type arguments ... 29
6.2. Graphical user interface application ... 30
6.3. Automated testing ... 31
6.3.1. Bash scripts set ... 31
6.3.2. Searching algorithms' productivity ... 31
6.3.3. Training cycles ... 32
6.4. Indexing performance parameters ... 32
7. MoreLikeThis vs. Semantic vectors ... 35
8. Similarity search in collections according to user's feedback ... 39
9. Future work and clustering of document collections ... 43
9.1. Clustering methods ... 43
9.1.1. Hierarchical agglomerative clustering ... 43
9.1.2. Clustering without a precomputed matrix ... 44
9.1.3 Efficiency issues related to the document clustering ... 45
9.2. Clustering with Semantic vectors ... 46
10. Summing up ... 49
Appendix A ... 51
References ... 53
List of attachments ... 57

Chapter 1

1. Introduction

One of the major problems our days in the subject of text information storage and retrieval is how to represent the content of text documents in a manner that provides not computationally expensive information retrieval and documents comparing.

The main subjective of this work is to present two approaches of similarity search. The Semantic Vectors package as a prominent approach of deploying modern matching technologies and Lucene's class MoreLikeThis. The source of motivation can be found in the rising number of scientific articles provided to the community of researches and the difficulty to organize them in a way the will provide time saving information retrieval.

Chapter 3 goes through the basic issues and techniques related to the information retrieval and text mining. In deep are discussed the main steps of the document's indexing process and the relational values between the items in the index.

Chapter 4 brings up details specific to Apache Lucene package along with discussion about the parameters of the index that can be created by the package supported by interesting test data. The last part of the chapter is dedicated to Lucene's class class MoreLikeThis.

Chapter 5 is a brief overview over the main features of the Semantic Vectors package. The used approach for similarity search is discussed emphasizing on the probabilistic dimension reduction methods including its base concepts, application areas and future development.

Chapter 6 is again dedicated to the Semantic Vectors this time rather than theoretically the chapter is trying to provide the reader with some test data and conception conclusions.

Chapter 7 is trying to compare the efficiency of the both approaches discussed in the previous chapters, MoreLikeThis and Semantic Vectors.

Chapter 8 is chasing another challenging task. Its objective is to investigate the possible approaches of implementing the similarity search system that is to take manual user's feedback in account.

Chapter 9 is looking forward in the future. The chapter is looking for the answer whether Semantic Vectors package can be used for managing the task of clustering text documents that it self could be considered as important unsupervised learning problem.

Chapter 2

2. Motivation

This paper was motivated from the everyday increasing number of scientific articles and memos provided by magazines and different scientific events and the issues that this fact provokes.

The fact that we have to deal with a large number of documents implies a demand of a system that can maintain relations between the documents based on meta data or some other method. Web based application like CiteSeer and Google Scholar in fact comply fully with this idea and provide a full text search in a large number of documents along with basic relation between the items stored in the system. If we take as a particular example the CiteSeer web application we can notice that it keeps a track of the number of articles that quotes certain document. This parameter makes difference mainly in the order number of the article in the results list. If an article is quoted often then can be assumed that the quality of article is high. This idea is well know already. The most popular Search engine today Google uses almost the same technique but instead of quotes are used the number of hyperlinks from other web sites that point to the ranked web site.

However those methods are working pretty well but we are highly motivated to look on the problem from a different aspect. We will try to come up with a new idea and approach the matter from a prospective that differs for the once mention above. In this relation we are to investigate the possibility of creating a system that can obtain manual user feedback and use this data to parameterize a similar following searches.

For this purpose we are to make a small research over the features and matters related to the efficiency of two freely distributed projects that could provide similarity document search for our system. Crucial here will be the speed with witch the products provide the results and the index size and structure.

As a final step the software product that performs better will be employed in out system. In case the results are encouraging we could chase another challenging task. Its objective will be to implement clustering of text documents using the method of user's feedback metadata. Understanding the complexity of the text documents clustering task we assume the work on this issue as a preparation for future research in the area of the Information retrieval.

Chapter 3

3. Information retrieval and text mining

3.1. Document indexing

During indexing documents are prepared for use by an Information Retrieval system. This means preparing the raw document collection into an easily accessible representation of documents. This transformation from a document text into a representation of text is known as indexing the documents. The indexing is normally done in the following four steps

3.1.1. Document linearization

Document Linearization is the process by which a document is reduced to a stream of terms. This is usually done in two steps and as follows:

- ✓ **Markup and format removal**

  During this phase, all markup tags and special formatting are removed from the document. Thus, for an html document all tags and text inside these are removed. This normally would include all element attributes, scripts, comment lines and text placed into these. Some commercial search engines may keep text placed inside the title tag, image alt attribute, table summary attribute and meta description tag. Other systems may not care for element attributes or meta data at all.

- ✓ **Tokenization**

  During this phase, all remaining text is parsed, lowercased, all punctuation removed along with strange alphanumeric characters and Cascading Style Sheets (CSS) instructions.

3.1.2. Filtration

Filtration refers to the process of deciding which terms should be used to represent the documents so that these can be used for:

- ✓ describing the document's content.
- ✓ discriminating the document from the other documents in the collection.

Frequently used terms cannot be used for this purpose for two reasons. First, the number of documents that are relevant to a query is likely to be a small proportion of the collection. A term that will be effective in separating the relevant documents from the non-

relevant documents, then, is likely to be a term that appears in a small number of documents. This means that high frequency terms are poor discriminators. The second reason is that terms appearing in many contexts do not define a topic or sub-topic of a document.

3.1.3. Stemming

Stemming in its base is process of reducing terms to their stems or root variant. Thus, "computer", "computing", "compute" will be modified to "comput" and "walks", "walking" and "walker" is reduced to "walk". Not all implementations use the same type of stemmer. The specifics of every language or at least group of languages will demand specific stemmer [29]. For English, the most popular stemmer is Martin Porter's Stemming Algorithm [13].
On one hand stemming process reduces the size of the inverted file but on the other hand too much stemming is not practical and can be annoying for the user.

3.1.4. Weighting

Weighting is the final stage in most Information Retrieval indexing implementations. Terms are weighted according to a given weighting model which may include local weighting, global weighting or both. If local weights are used, then term weights are normally expressed as term frequencies (tf). If global weights are used, the weight of a term is given by inversed document frequency(idf) values. The most common (and basic) weighting scheme is one in which local and global weights are used (weight of a term = tf*idf). This is commonly referred to as tf*idf weighting.

3.2. Term frequency and Inverse document frequency [26]

Term frequency can be defined in several ways but one of the most common used is the number of occurrence of the term divided by the sum of the occurrence of all terms in the document.

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

Inverse document frequency like term frequency have many modifications but always the purpose is on: to measure the general importance of the term. One of the most popular definition follows:

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

Where D is the number of documents and $|\{d : t_i \in d\}|$ is the number of documents that contain the term.

The product of the multiplication of the term frequency and inverse document frequency is usually used to score a term in a document according to an index.

$$\mathrm{tfidf}_{i,j} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i$$

The score has a height value when the term has a height frequency in the document and appears rarely in the rest of the documents.

As an example we can consider a document containing 1000 words and the word Lucene appears 5 times. Following the previously defined formulas, the term frequency for Lucene is then 0.005. Now, assume we have 5 million documents and Lucene appears in five hundred of these. Then, the inverse document frequency is equal to log(5 000 000 / 500) = 4. The tf-idf score is the product : 0.03 * 4 = 0.12.

3.3. Inverted index [27]

Inverted index is a index data structure that allows full text search. The data structure stores mapping of the location of words in a set of documents. This main feature makes it most popular structure for the purposes of information retrieval.

There two types of inverted indexes:

- Record level inverted index – maps a term to a list of documents that comprise this term.
- Full inverted index – maps a term to couples of digits. The first digit of each couple provides the document that comprise the term and the second digit provides the position of the term in the document.

3.4. Cosine-distance ratio

This ratio is used as a similarity measure between any two vectors representing documents or queries. The ratio defines the cosine angle between the vectors, with values between 0 and 1 and this was normalize the DOT product.

$$\mathrm{Sim}(A, B) = \mathrm{cosine}\ \theta = \frac{A \bullet B}{|A||B|} = \frac{x1^*x2 + y1^*y2}{(x1^2 + y1^2)^{1/2}\ (x2^2 + y2^2)^{1/2}}$$

When the angle between two vectors is getting smaller the cosine product approaches 1. The angel between the two vectors can express similarity or other relation of whatever the vectors presents. A cosine

product approaching 1 means more common ground for what ever the vectors represent.

This is a convenient way of ranking documents; in other words by measuring how close their vectors are to a query vector. However this method has one drawback. Longer documents are given smaller term weights and smaller documents are favored over longer ones. Pivoted Unique Normalization [28] tries to correct it based on the document length, the probability that a document is relevant and the probability that the document will be retrieved.

For the purposes of ranking we should create term vector model. Vector space model [6](or term vector model) is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers, such as index terms. It is used in information filtering[8], information retrieval[7], indexing and relevancy rankings.
A document is represented as a vector. Each dimension corresponds to a separate term. If a term occurs in the document, its value in the vector is non-zero. Several different ways of computing these values, also known as (term) weights, have been developed. One of the best known schemes is tf-idf weighting.

The cosine similarity (cosine angle) between query and documents is represented as follows:

$$\mathrm{Sim}(Q, D_i) = \frac{\sum_i w_{Q,j} w_{i,j}}{\sqrt{\sum_j w_{Q,j}^2} \sqrt{\sum_i w_{i,j}^2}}$$

where the sigma symbol means "the sum of", Q is a query, D is a document relevant to Q and w are weights. Weights can be defined in terms of variants of tf and idf, each one with their own customized definition and theoretical interpretation.

In short Term Vector Theory is applying the Vector Analysis technique to the Information Retrieval problem.

Chapter 4

4. Apache Lucene

Lucene is a free Java solution providing indexing and searching for text documents. Lucene is not complete application ready to use but is a library implemented in a way that implies easy deployment to different kind of applications intended to work with versatile text documents. Further on is provided a list of typical application software that can take advantage of Lucene.

- **Web pages** – weblog, wiki software.
- **E-mail clients** – full-text mailbox search and email log indexing.
- **Specific Search engines** – intended for developers for searching source code, job offers, shopping.

4.1. Fundamental Lucene's classes for indexing text.

- **IndexWriter** - IndexWriter is used to create a new index and to add Documents to an existing index.
- **Analyzer** - Before text is indexed, it is passed through an Analyzer. Analyzers are in charge of extracting indexable tokens out of text to be indexed, and eliminating the rest. They are also used when searching. Because the search string has to be processed the same way that the indexed text was processed, it is crucial to use the same Analyzer for both indexing and searching. Not using the same Analyzer will result in invalid search results.
  The Analyzer class is an abstract class, but Lucene comes with a few concrete Analyzers that pre-process their input in different ways. Should we need to pre-process input text and queries in a way that is not provided by any of Lucene's Analyzers, we will need to implement a custom Analyzer. If we are indexing text with non-Latin characters, for instance, we will most definitely need to do this.
- **Document** - An index consists of a set of Documents, and each Document consists of one or more Fields.
- **Field** - Each Field has a name and a value. Lucene offers two different classes that specifies the fields from which a developer can choose.
  - Field.Index - Specifies whether and how a field should be indexed.
  - Field.Store - Specifies whether and how a field should be stored.

4.2. Lucene's index structure

Typical Lucene index is stored in a single directory in the filesystem on a hard disk.
The core elements of such an index are segments, documents, fields, and terms. Every index consists of one or more segments. Each segment contains one or more documents. Each document has one or more fields, and each field contains one or more terms. Each term is a pair of Strings representing a field name and a value. A segment consists of a series of files. The exact number of files that constitute each segment varies from index to index, and depends on the number of fields that the index contains. All files belonging to the same segment share a common prefix and differ in the suffix. Each segment is as a sub-index, although each segment is not a fully-independent index.

- **Indexing speed factor**

One of the settings that have impact on the searching speed is how often the changes buffered in memory are flushed to the index on the hard disk. The default is to flush when random access memory usage is 16 megabytes. For better indexing speed flushing should by done by usage of a large random access memory buffer. An additional issue is that flushing just moves the internal buffered state via IndexWriter into the index, but these changes are not visible to IndexReader until either commit() or close() is called.

- **Merging indexes**

To optimize an index, method optimize() should be called on an IndexWriter instance. This will cause all documents in the memory to be flushed to the disk and all index segments to be merged into a single segment, reducing the number of files in the index. On the other hand, optimizing an index does not help improve indexing performance. Actually, optimizing an index during the indexing process will slow things down. Despite this, optimizing may sometimes be necessary in order to keep the number of open files under control. For instance, optimizing an index during the indexing process may be needed in situations where searching and indexing happen concurrently, since both processes keep their own set of open files. A good rule of thumb is that if more documents will be added to the index soon, calling optimize() should be avoided. If, on the other hand, the index will not be modified for a while, and the index will only be searched, it is a good time to optimize it. That will reduce the number of segments (files on the disk),

and improve search performance. The number of files that Lucene should open during the search influence directly searching speed.

- **Indexing performance parameters**

  - **Normal index**

Files number	10 000
Words number	53 100 173
Time elapsed	219881 milliseconds (3 min 39 sec)
Milliseconds per file	21,98 milliseconds per file.
Size on disk	73 MB

  - **Positional index**

Files number	10 000
Words number	53 100 173
Time elapsed	474218 milliseconds (7 min 54 sec)
Milliseconds on file	47,42 milliseconds per file.
Size on disk	182 MB

4.3. Searching related documents with Lucene

Lucene's class MoreLikeThis uses all the methods described in Chapter 3 to find related documents in an index to the one provided. For this purpose the content of the source document is analyzed using content of the index. The result of this procedure is a query that is executed and this way a result list of related documents is obtained.

There a two main options that MoreLikeThis offers for providing the source file. The file can be in the Lucene index or can be parsed from an external source. We tested both methods and noticed some difference in the score results that sometimes exceeded 30 %. Further on we will discuss only the case when we use as input a document from the index. The documents in the index are identified by unique document number (docNum) that we will employ to identify our source document.

Along with the source file, MoreLikeThis offers an option a set of parameters to be provided that will guide the parsing process. One of them is to provide a set of field names that are to be taken in account. If no fields are provided, MoreLikeThis will obtain all fields from the index that are marked as "indexed".

The next step that MoreLikeThis undertakes is to loop over the list of the fields (method retrieveTerms) obtaining the term frequency vec-

tor for each field, if during the indexing process term frequency vector for the field is created, or obtains directly the terms in the field calculating their frequency. Both options are dismissing all stop words from the result list. The structures obtained this way are accumulated in a HashMap, where the key is the term and the value is the count of the term in the document (method addTermFrequencies). Here is checked the parameter maxNumTokensParsed. This parameter is not for the whole document but only for a field.

The HashMap provided from retrieveTerms method is passed to the createQeueu method. The method computes five parameters related to a term. All of them are inserted into an array of Objects. All arrays are sorted in a PriorityQueue that is returned as a result from createQueue method. The PriorityQueue is sorted according the score parameter. The other important parameter except socre is top field for a term. This is a field in that the term is most often used. The rest of the parameters are saved for debug purposed (IndexSearcher.explain)

The createQueue method loops throughout all words in the HashMap provided as an input. First checks if a term appears in the document more than minTermFreq parameter. If this is the case, the next step is to be determined the topfield parameter and docFreq parameters. The topField parameter is the field in the index in which the term is used most often. The docFreq is sum of the number of time used terms has been used in this fields in all the documents in the index. The docFreq parameter computed in this way must be greater than minDocFreq parameter.

When we have the docFreq parameter we need the number of documents in the whole index as well in order to be able to calculate the inverse document frequency. How the idf will be exactly calculated is defined in a class that inherits the abstract class Similarity. As an example in Apendix A is described the implementation of the DefaultSimilarity class where can be found more detailed information of how idf is calculated.

The last and most important parameter that should be calculated is the score. The score is calculated as a product of the number of times a term is used in the source file (term frequency) multiplied by the idf (inverse document frequency).

As a last step the five parameters (topfield, score, idf, docFreq,tf) and the term are inserted in the priority queue.

Now, when we have all the terms in the priority queue, can go further on and create a Lucene query. As we mentioned before to accomplish this task we will use only the name of the term, the topfield and

the score. How many terms from the top of the priority queue will be taken in account in the creating process is set in the setMaxQuery-Terms parameter. The other parameter that is relevant to the Lucene query creation is boost flag. If the flag is set to “true” every term in the query will be boosted by the product of the score of the term divided by the score of the term form the top of the priority queue.

Lucene has many Query types (TermQuery, BooleanQuery, ConstantScoreQuery, MatchAllDocsQuery, etc.) but the query parser does not create all types. Most of the queries are mapped to basic queries like TermQuery and BooleanQuery. This is the case in MoreLikeThis Class as well where BooleanQuery type is employed.

Ones the query is generated is passed to the Search method of the IndexSearcher class. In the Search method first is called the method BooleanQuery.createWeight. This function returns an object of type BooleanWeight for the query.(BooleanWaight is a subclass of BooleanQuery and implements interface Weight). As second step is called method BooleanWeight.score that returns a BooleanScorer2 class (BooleanScorer2 inherits Score class). As third and last step is called method BooleanScorer2.score that will iterates over documents matching a query and return a result list of related documents. Scores are computed using a given Similarity implementation(see Appendix A).

Example of scoring :

```
Score: 0.21167752 is sum of :
    0.052198052 = weight(content:program in DocID), product of:
        0.4598019 = queryWeight(content:program), product of:
            1.0953102 = idf(docFreq=9, numDocs=11)
            0.4197915 = queryNorm
        0.11352291 = fieldWeight(content:program), product of:
            6.6332498 = tf(termFreq(content:program)=44)
            1.0953102 = idf(docFreq=9, numDocs=11)
            0.015625 = fieldNorm(field=content, doc=8)

    0.039045364 = weight(content:model in DocID), product of:
        0.38326484 = queryWeight(content:model), product of:
            0.9129886 = idf(docFreq=11, numDocs=11)
            0.4197915 = queryNorm
        0.10187567 = fieldWeight(content:model), product of:
            7.1414285 = tf(termFreq(content:model)=51)
            0.9129886 = idf(docFreq=11, numDocs=11)
            0.015625 = fieldNorm(field=content, doc=8)
```

Chapter 5

5. Semantic Vectors

The Semantic Vectors package is trying to fill the gap between the promising research results and the practical implementation of the semantic idea. The package creates semantic vectors for words and documents applying a Random Projection algorithm to term-document matrices created using Apache Lucene. The result of those sets of mathematical transformations is that similar concepts vectors are near to each other in the space. This advance technique make possible applying in practice to some extend the idea of automatically matching related concepts.

5.1. Wordspace model [3]

The Semantic Vectors package creates a WordSpace model . The concept of the WordSpace models lays on representing target items in a high directional vector space.

Semantic Vectors build the model in three stages:

- ❖ Create basic random vectors for each document.
- ❖ Create term vectors by summing the basic document vectors the term occurs in.
- ❖ Create new document vectors by summing the term vectors of the terms that occur in each document.

For the default indexes there is an interesting possibility called training cycles that returns the result from stage three to stage two.

5.2. Probabilistic dimension reduction methods [1]

There are several methods in favor of dimensions reducing. The most popular among them our days is the Latent Semantic Analysis[25].

Latent Semantic Analysis is completely straight forward mathematical method for finding relations between words in text documents. It does not use dictionaries, grammars or semantic parsers. The input stream is a raw text parsed into words separated into meaningful parts as sentences or paragraphs.

LSA [12] represents the text document as a matrix in which each row stand for a unique word and every column stands for a text paragraph or other meaningful passage. Each cell in the matrix represents the number of appearance of a word in a paragraph defined from the column index.[5] As a next step a preliminary transformation is ap-

plied on the matrix in which each cell is estimated by a function that expresses both the word's importance in the particular text domain and the degree to which the word provides information in the text passage.
Further the LSA applies singular value decomposition (SVD) to the matrix. SVD [11] decomposed the matrix to three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed. There is a mathematical proof that any matrix can be so decomposed perfectly, using no more factors than the smallest dimension of the original matrix. When fewer than the necessary number of factors are used, the reconstructed matrix is a least-squares best fit. One can reduce the dimensionality of the solution simply by deleting coefficients in the diagonal matrix, ordinarily starting with the smallest. (In practice, for computational reasons, for very large corpora only a limited number of dimensions can be constructed.)

Regardless of the give robust performance of the LSA the authors of the Semantic Vectors package choose to use another method for their purposes. This change was demanded due to a performance issues. The goals that were set in front of the Semantic Vectors demanded computationally less expensive approach as Random projectoin.

Random projections [1][4] is a powerful Locality Sensitive Hashing (LSH) method for dimensionality reduction. The main idea of those type of methods is to separate the input in a way that similar items will fall to the same group where the number of groups is much smaller than the possible universe of the input items.
Typical application areas are the processing of both noisy and noiseless images, and information retrieval in text documents.

Whether this kind of experiment will bring the expected good performance of the package is still under question. Even though it is assumed that the random projection algorithm will be effective enough to give robust performance the author have left a backdoor solution. The package is implemented in way that the change of the dimension reduction method can be done with minimum effect to the other modules.

It is a hard task to compare LSA and Random Projection but mainly Random Projection was preferred because of the following arguments:

- ❖ Random Projection performs comparably well as Latent Semantic Analysis.
- ❖ The Random Projection algorithm is the simpler, and therefore best for implementation, for testing and for collaborating with the rest of the code.
- ❖ It is easy to update a basic Random Projection model incrementally.

5.3. Software architecture

There are several reasons why Semantic Vectors Package becomes so popular in such a short time.

- ❖ It is written in Java, that make it platform independent
- ❖ It has only one dependency (Apache Lucene)
- ❖ It is easy to use

The package has two main functions

- ❖ Building a WordSpace Model. - The main utility is *BuildModel*. It creates a termvectors and docvectors files. Can be started with the following options:
  - o -d vector length or number of dimensions
  - o -s [seed length] number of non-zero entries in basic vectors
  - o -m [minimum term frequency]
  - o -tc [training cycles]
  - o -docs [incremental|inmemory] Switch between building doc vectors incrementally" (requires positional index) or all in memory (default case).

  An option of the default model building is building a Posisional Index with the *BuildPositionalIndex* utility. This utility will requite a positional index created by Lucene's tool *IndexFilePositions*. Creating the model this way will take account of the word order. A specific argument that must be provided is the window size where it is a odd number defining the count of words in both sides of the word. The Positional Index created by Lucene can be used by the default model just will take more place on the disk.

  Further enhancement is the Permutation index. It is created in the same manner like the standard Positional

Index with option -indextype permutation or -indextype directional. A permutation index encodes the position of each term relative to each other term within a sliding window, while a directional index encodes whether a term occurs before or after each other term in this sliding window.

- Document Search – Semantic Vectors package offers a versatile set of searching utilities. Their detail description and performance is objective of the next chapter.

Chapter 6

6. Similarity search with Semantic vectors

The major subject of this section is to revise in detail the searching features provided by the package and to try to summaries their efficiency of finding related documents. This fact implies good grasp of the ideas related to the search implementation. This chapter will pay more attention exactly to those modules of the package.

6.1. Search type arguments

- ***SUM - default option***
  - ✓ Searching for Documents using Terms

    java pitt.search.semanticvectors.Search -q termvectors.bin -s docvectors.bin term

  - ✓ Using Documents as Queries

    java pitt.search.semanticvectors.Search -q docvectors.bin -s termvectors.bin - lowercase false bible_chapters/Genesis/Chapter_1

- ***SPARSESUM***

  Build a query as with SUM option, but quantize to sparse vectors before taking scalar product at search time.

  java pitt.search.semanticvectors.Search -searchtype sparsesum term1 term2

- ***SUBSPACE***

  "Quantum disjunction" - gets vectors for each query term, create a representation for the subspace spanned by these vectors, and score by measuring cosine similarity with this subspace.

- ***MAXSIM***

  "Closest disjunction" - get vectors for each query term, score by measuring distance to each term and taking the minimum.

- ***TENSOR***

  A product similarity that trains by taking ordered pairs of terms, a target query term, and searches for the term whose tensor product with the target term gives the largest similarity with training tensor.

- ***CONVOLUTION***

  Similar to Tensor, product similarity that trains by taking ordered pairs of terms, a target query term, and searches for the

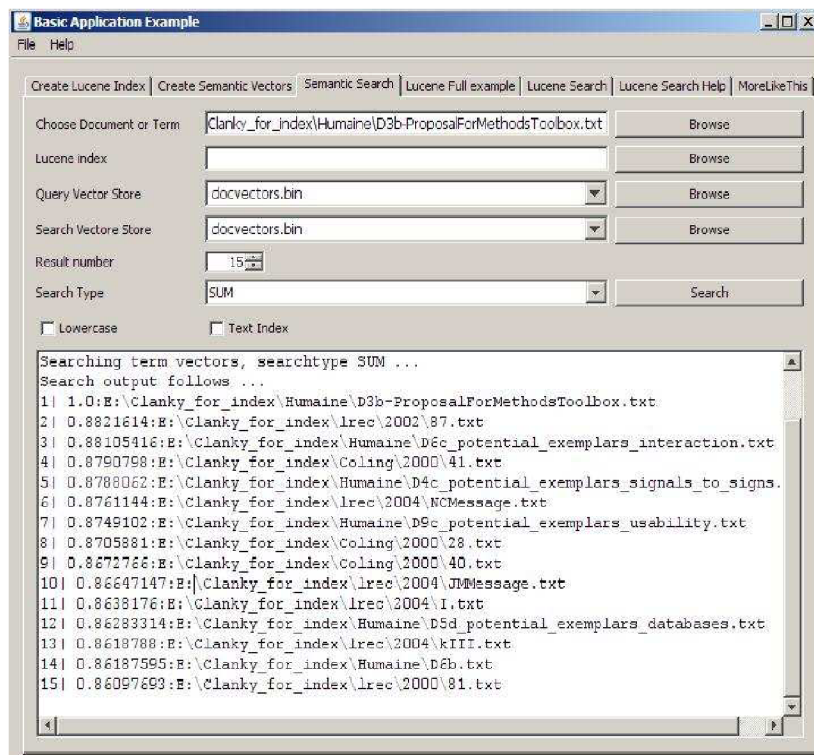
term whose convolution product with the target term gives the largest similarity with training convolution.

❖ ***PERMUTATION SEARCH***

The technique of using vector coordinate permutations (also interpreted as rotations) to investigate the effect of word order on vector semantics [2]

6.2. Graphical user interface application

The first thing that occurred to our minds at the begging of this work was that some graphical user interface will make monitoring of the process of indexing, creating a model and searching the model much more comprehensive. The Semantic Vectors package and Lucene Apache provide as with a lot of alternatives not only for searching but even how to create the index and the model. Some of the alternative options are connected through dependency relations. For example certain type of searches expected a certain type of index. That is why as a first step we created a Java application that is to provide us with all possible options trough a graphic user interface and at the same time will look after the dependences between the parameters throughout the creation and searching process. Not at the last place it will provide us with the ability for faster testing and opportunity to put outs new ideas and new knowledge gained about Semantic Vectors package capabilities right in the application.



6.3. Automated testing

The GUI application give us a good start but it turned out that it will be hard to implement automated tests for Semantic Vectors package in this environment. The results produced by Semantic Vectors are in plain text. It made us think about an environment that will provide us with powerful tools for text manipulation.

A popular decision of our concern are the Unix shells. Our choice was the Bash shell. This shell is one of the most popular and creating a scripts is relatively easy.

The corpora used for testing comprise a large amount of scientific publications presented on the different kind of events. The fact that those articles were prepared for events with certain topic of relatively narrow scientific research areas make us think that due to this fact we could assume that the articles from one event is more likely to be recognized by the Semantic Vector Package as documents that have common semantic ground.

Test cases presented further on are motivated from the common issues currently attracting the attention of the researches working in the field of natural languages linguistic computer representations.

6.3.1. Bash scripts set

The bash scripts set implemented for those tests are comprised from three main files and has one dependency. The script expected that will find “pdftotext” executable file in the current directory. This file is part of the poppler-utils rpm package.

The *auto_sem_vec_test.sh* scripts provide a preprocessing of the documents set and convert the pdf files to text using pdftotext tool.

The *start_sem_test.sh* script parses the input parameters from the command line estimate them and execute the *sem_test.sh* script with the appropriate parameters.

The sem_test.sh script looks for the most similar documents for each document from the index with the provided searching option, parses the output and estimates the results.

6.3.2. Searching algorithms’ productivity

The first test compares the SUM, SUBSPACE and MAXSIM searching options provided by Semantic Vectors package. The specifics’ of each of the searching options was already discussed.

The test file set comprises 237 docs (1 036 508 words) separated by topic relevance to ten folders. Each folder comprises at least fifteen

documents and every document comprises at least five hundreds words. The Semantic vectors index is created in ten training cycles.

To each result line is assigned a number from 8 to 1. The test scripts run a search for every document from the set and add the number of the result line that provides a document from the folder where the file resigns to a common sum. At the very end the sum is divided by the number of the files processed.

Method	Results
SUM	16.443037974
SUBSPACE	16.443037974
MAXSIM	16.443037974

The result of the test is quite weird. The output of all three different types of searches is the same. The authors were informed about this issue.

Additionally the test provides the average number of similar documents found in the result set of documents that resigned in the same folder. The value is 4.772151898 for list results of ten documents. It is a little less than 50 %.

6.3.3. Training cycles

In this test we will try to investigate the impact of more training cycles on the Semantic Vectors' search performance.

Method	One cycle	Five cycles	Ten cycles	Twenty cycles
SUM	0.796762389	1.896564933	2.758944822	2.795733724
SPARSESUM	14.045226130	17.145728643	25.256281407	27.185929648
SUBSPACE	2.160514266	1.156582368	2.758944823	2.795733716
MAXSIM	2.870141037	1.139624238	2.758944822	2.795733724

The result of the test seems quite optimistic. The test proves the efficiency of the model training feature. From the results can conclude that a training of ten cycles is optimal.

6.4. Indexing performance parameters

- **Semantic Vectors index**
  - ***Normal index***

Number of documents	10 000
Number of terms	688677
Training Cycles	1
Time elapsed	55 seconds
Size on disk Tems	55 MB
Size on disk Docs	8 MB

Number of documents	10 000
Number of terms	688677
Training Cycles	10
Time elapsed	378 seconds
Size on disk Tems	55 MB
Size on disk Docs	8 MB

- ***Positional indexes***
  - ***Permutation index***

Number of docs	10 000
Number of terms	68477
Vector length	200
Window size	3
Time elapsed	164 seconds (2 min 44 sec)
Size on disk Perm	55 MB
Size on disk Random	55 MB

Number of docs	10 000
Number of terms	68477
Vector length	200
Window size	11
Time elapsed	509 seconds (8 min 29 sec)
Size on disk Tems	55 MB
Size on disk Docs	55 MB

Number of docs	10 000
Number of terms	68477
Vector length	100
Window size	3
Time elapsed	138
Size on disk Tems	28 MB
Size on disk Docs	28 MB

- ***Incremental index***

Number of docs	10 000
Number of terms	68477
Vector length	100
Window size	3
Time elapsed	162 (2 min 42 sec)
Size in Disk drxnterm	55 MB

For building this index is used the Lucene's index that provided the numbers discussed in Chapter 4. The BuildIndex class created for this index 1257308 terms and 28643 docs in 135 seconds. As we can see the Semantic Vectors' index comprise almost one thousand documents less than the Lucene's index. This is because the conversion process from pdf to text is not on hundred percent successful. Some times the result text document has not enough recognizable words and the Semantic Vectors failed to build a vector for this document. In case that we provide such document that has no vectors in the index as a search parameter the Sum, Sparsesum and Maxsim search types generate no output. On the other hand searching with type Subspace will generate error output that can cause problems in automates test as unpredictable output. This problem is handles in function test_Search_res in sem_test.sh script.

Chapter 7

7. MoreLikeThis vs. Semantic Vectors

In this chapter we will try to estimate some test data that should give us an answer of the question how well perform the advanced techniques used by Semantic Vectors versus the MoreLikeThis approach.

The test was done on a set of 237 text files that were clustered into ten folders. This way each folder claimed to comprise related documents that are less related to the files in the rest of the set of folders.

We are to look for similar documents for each of the 237 text documents. We will expect that most of the results will resign in the same directory as the one provided as an input. This fact we will employ to estimate the performance of the methods for searching related documents. The method that places more documents from the directory where the source document resign in the result list will be estimated as more successful. The result list will be comprised of ten document.

There is one specific thing related to the Semantic Vectors package that should be taken into account. The Semantic Vectors package not always provide the same result list in case the index files were recreated. Due to this fact we decided to run the test ten times and compute an average set of results based on the results of every single execution.

For all of the tests bellow we created a Lucene's index (class IndexFiles) with default values for all options. This was not the case with the Semantic Vectors' indexes. Trying to improve the performs of the Semantic Vectors package we experiment on vectors with different length.

In the searching procedure for Semantic vectors we used the standard approach for related documents search pointing both query searching vectors to docvectors*.bin.

For searching with MoreLikeThis class we created our own class that in addition obtaineds the "path" field for the Lucene's index for the purposes of the test. The other change that we made was based on results of our previous research work on MoreLikeThis. In this work we found out that the MoreLikeThis performs best when the number of terms in the query is about one hundred.

The test results are presented in a simple table displaying only the average number of times when for a search for a certain source file a given method obtained more files from the folder were the source file resign that the other method. The last row is reserved for the number of time when both methods performed equal.

- The table below represent the result of the first test. In this test case Lucene's class MoreLike this performed much better than Semantic Vectors.

Training Cycles	1 (default)
Vector length or number of dimensions	200 (default)
Vector Seed Length - Number of non-zero entries in basic vectors	20 (default)

MoreLikeThis	119.4
Semantic Vectors	70.5
Equal	47.1

- The results from the previous test made us thinking of a way how to improve the performance of the Semantic Vector package. We started with the most logical approach, increasing the values of the Vector Length and Vector Seed Length. In addition put in practice the training index feature of the package and trained the index ten times (see the tests for SV).

Case 1:

Training Cycles	10
Vector length or number of dimensions	200
Vector Seed Length - Number of non-zero entries in basic vectors	50

MoreLikeThis	123.1
Semantic Vectors	69.4
Equal	44.5

Case 2:

Training Cycles	10
Vector length or number of dimensions	200
Vector Seed Length - Number of non-zero entries in basic vectors	50

MoreLikeThis	123.1
Semantic Vectors	69.4
Equal	44.5

Case 3:

Training Cycles	10
Vector length or number of dimensions	400
Vector Seed Length - Number of non-zero entries in basic vectors	20

MoreLikeThis	117.9
Semantic Vectors	71.7
Equal	47.4

Case 4:

Training Cycles	10
Vector length or number of dimensions	400
Vector Seed Length - Number of non-zero entries in basic vectors	20

MoreLikeThis	117.9
Semantic Vectors	71.7
Equal	47.4

As we can see despite our efforts to provide all possible combination of options the results do not differ much.

- As a last attempt we decided to dramatically increase the values of the Vector length and Vector Seed Length to a levels ten times higher than the default ones.

Training Cycles	10
Vector length or number of dimensions	2000
Vector Seed Length - Number of non-zero entries in basic vectors	100

MoreLikeThis	111.1
Semantic Vectors	79.5
Equal	47.4

In the results of this search we can see that Semantic Vectors starts to perform better but the question is for what price. As we can see we go ten time over the normal default values. This cause the creating of the index to be extremely slow and to need a lot of system resources. This fact makes this case nearly impossible to be put in practice.

The software (Java, Perl) that provide the test can be found on the CD attached. The software is independent from the directory structure or filenames so the test can be executed on other files sets as well.

Chapter 8

8. Similarity search in collections according to user's feedback

In this chapter we will go for chasing another challenging task. Our objective will be to investigate the possible approaches of implementing the similarity search system that is to take manual user feedback in account. Further on we can even dare thinking of making a record of the user feedback and use it for automatically improved similarity search.

In the previous parts of this work we presented two approaches of similarity search along with some test performance data. In those studies the Semantic Vectors, as a representative of a new attractive semantic similarity search method, were performing well but still less effective than Lucene's class MoreLikeThis. When we were thinking about which method to employ in our system, this previous experience convince as that Lucene and MoreLikeThis will be the more appropriate way to achieve our goal.

The system architecture is based on a three level model. The first stage is Searching followed by Retrieving user's feedback and once again Adjusted Searching according to the data retrieved in stage two. As we already mentioned above for the searching part we intent to employ Lucene's MoreLikeThis class. The main crux here with high probability will be the middle part or how to retrieve the user's feedback in order the following search to be more accurate.

Retrieving the user's feedback will be in close relation with the way how MoreLikeThis implements the similarity search. A good grasp of this could help us to find out what kind of information the user could provide for us the can be used for more accurate search.

As we described in details in the previous chapters MoreLikeThis requires a source text file as an input parameter and produces a query. The query itself is a set of terms along with their weights. Then this query is applied to the whole index and result documents list is obtained. The documents in the list are documents for which the terms in the query are estimated with height weight.

What first occurred in our minds was to ask the user to select one document from the result list obtained from a single search that according to him is not related to the document provided as an input.

Having this document we can find the terms in the selected document that match with the query. We will provide this list to the user

with the position of the term in the selected document and the poison of term in the source document. The user can then select a bunch of terms that think that are not appropriate. Those words will be automatically added to the stop words list. Stop words list is a feature of MoreLikeThis that comprises words that should not be taken in account in comprising the query.

The words that probably should be added to the stop words list are words that have small order number in the selected document and big order number in the source document. This will means that the term is important for the selected document and not so important for the query.

If we look in how the score is calculated in Lucene, we will noticed that the number of overlapping terms between the query and the document is important from it self.

Experimenting with this method it turned out that if we select all terms that have position greater than half of the query length in the source document, the selected document step down several position in the result list.

Basic Application Example

Insert File to the Index | More Like This Search

Document : :\aaa\NetBeans\Workspace\Lucene_Index_Fields\CEUR_for_Index_all\Folder_0\folder_0_file_10.txt Browse

Min Term Freq 2 Min Word Len 3 Max Query Terms 100 Fields : Title authors

Min Doc Freq 5 Max Word Len 0 Max Num Tokens Parsed 5,000 content summary

Stop Words logic programs formula variable

Boost words Number of results 25 Search

Results

Number	Score	DocID	Doc Folder	File Name	Doc Tite
1	0.91892666	8	folder_0	folder_0_file_10.txt	Automated Formal Verification of PLC Programs Written in IL
2	0.39955494	4	folder_0	folder_0_file_6.txt	Symbolic Fault Injection
3	0.36098072	5	folder_0	folder_0_file_7.txt	A Termination Checker for Isabelle Hoare Logic
4	0.251216	7	folder_0	folder_0_file_9.txt	Fully Verified JAVA CARD API Reference Implementation
5	0.22182801	9	folder_0	folder_0_file_11.txt	Combining Deduction and Algebraic Constraints for Hybrid System Analysis
6	0.22044659	3	folder_0	folder_0_file_5.txt	A History-based Verification of Distributed Applications
7	0.20309332	10	folder_0	folder_0_file_12.txt	A Sequent Calculus for Integer Arithmetic with Counterexample Generation
8	0.2012648	6	folder_0	folder_0_file_8.txt	The Heterogeneous Tool Set
9	0.19727838	0	folder_0	folder_0_file_2.txt	Formal Device and Programming Model for a Serial Interface
10	0.17544544	1	folder_0	folder_0_file_3.txt	Combinations of Theories and the Bernays-Sch¨nfinkel-Ramsey Class
11	0.1565813	35	folder_3	folder_3_file_1.txt	Modeling Data & Processes for Service Specifications in Colombo
12	0.14172813	2	folder_0	folder_0_file_4.txt	An Advanced Logic for Interactive Component Engineering

Interesting terms in the selected document that are common with the query Common: 22

Position in the selected document	Term	Position in the query
16	logic	30
20	programs	2
21	can	25
23	system	32
30	formula	40
32	safety	34
34	variables	1
36	systems	36
40	variable	21
46	symbolic	41
47	states	7
50	state	9
53	have	78
60	which	24

Example: We made a test search with our system for publication "Using Formal Concept Analysis Using Formal Concept Analysis for Heterogeneous Information Retrieval" from workshop Vol-162. For our surprise the best result for the search was publication "LSI vs. Wordnet Ontology in Dimension Reduction for Information Retrieval" from workshop Vol-98.

We were determined to find out why this happens. As a first step we decide to remove all words that are common for both publications but obviously do not bring any meaning. Those words were :

document, concepts, concept, recall, calculated, answer, from.

The following adjusted search places our document one position down in the list. This was unsatisfying result for as. So we decide to go further and remove another set of words that were not topic spesific:

model, precision, set, threshold.

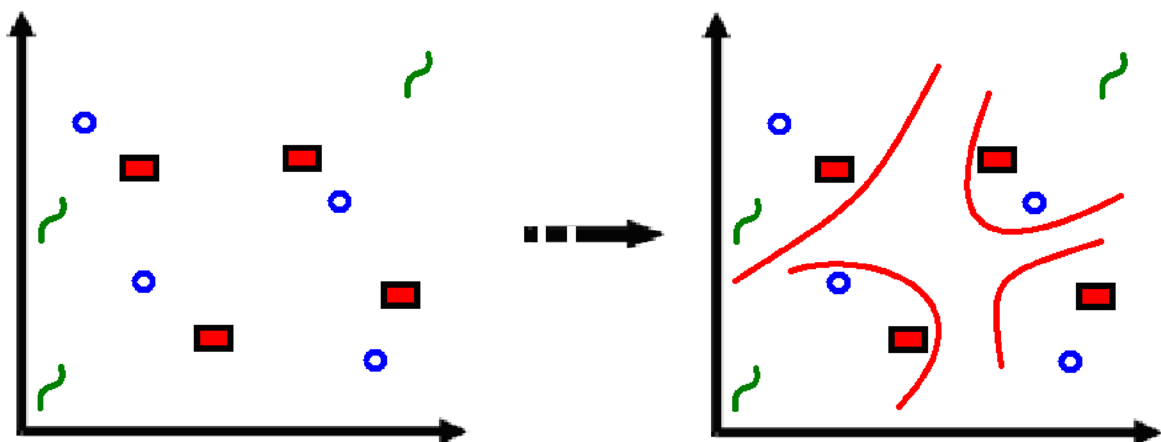
The result of this action was that the document was moved seven positions down. Given that both publications have common ground, the Information retrieval, we were satisfied from the result. The words that left were:

retrieval, vector, query, collection, similarity, information, data-base, search.

Chapter 9

9. Future work and clustering of document collections

Clustering can be considered as the most important unsupervised learning problem [22]. The purpose of a clustering algorithm is to add structure to a set of data. A loose definition of clustering could be "algorithm that groups together objects with similar features". Often similarity is estimated as distance between the items. Each item within a cluster would be similar, and dissimilar between elements in other clusters.



In the case displayed on the scheme above is easy to be identified four clusters into which the data can be divided. The similarity measure is distance. The main objective is to put together in one cluster items that are "close" in geometrical matter. This approach is well known as distance-based clustering.

9.1. Clustering methods

9.1.1. Hierarchical agglomerative clustering

Hierarchical algorithms find clusters using previously established clusters. Hierarchical algorithms can be agglomerative ("bottom-up") or divisive ("top-down"). Agglomerative algorithms begin with each element as a separate cluster and merge them into larger clusters. Divisive algorithms begin with the whole set and proceed to divide it into smaller clusters.

Steps of the Hierarchical Clustering Algorithm are :

- First N*N document similarity matrix is formed. Each document is placed into its own cluster.
- The following two steps are repeated until only one cluster exists.

- The two clusters that have the highest similarity are found.
- These two clusters are combined, and the similarity between the newly formed cluster and remaining clusters recomputed.

- As the larger cluster is formed, the clusters that merged together are tracked and form a hierarchy.

There are different methods to calculate the similarity measure between two clusters like Single Link Clustering, Complete Linkage, Group Average etc.

Once the hierarchy is generated, it is necessary to determine which portion of the hierarchy should be searched.

A top-down search starts at the root of the tree and compares the query vector to the centroid for each subtree. The subtree with the greatest similarity is then searched. The process continues until a leaf is found or the cluster size is smaller than a predetermined threshold.

The alternative method is a bottom-up search starts with the leaves and move upwards.

9.1.2. Clustering without a precomputed matrix

In this case data are grouped in an exclusive way, so that if a certain object belongs to a certain cluster then it could not be included in another cluster. As an example of this group of algorithms can be pointed out K-means.

K-means algorithm was presented by MacQueen in 1967. This is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set to a certain number of clusters (assume k clusters) fixed before hand. The main idea is to define k centroids, one for each cluster. These centroids should be placed in a specific way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is left, the first step is completed and early clustering is done. At this point we need to re-calculate k new centroids as centers of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more.

Finally, the algorithm objective is to minimize a function, in this case a squared error function.

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2$$

Where $\left\| x_i^{(j)} - c_j \right\|^2$ is a chosen distance measure between a data point $x_i^{(j)}$ and the cluster centre c_j, is an indicator of the distance of the n data points from their respective cluster centers.

Although it can be proved that the procedure will always terminate, the k-means algorithm does not necessarily find the most optimal configuration, corresponding to the global objective function minimum. The algorithm is also significantly sensitive to the initial randomly selected cluster centers. The k-means algorithm can be run multiple times to reduce this effect.

9.1.3. Efficiency issues related to the document clustering

In our days when an efficiency problem is faced the first thing that comes to mind is to try to divide the processing on independent parts and execute each of them simultaneously on several processors. This of course in not always a trivial task.

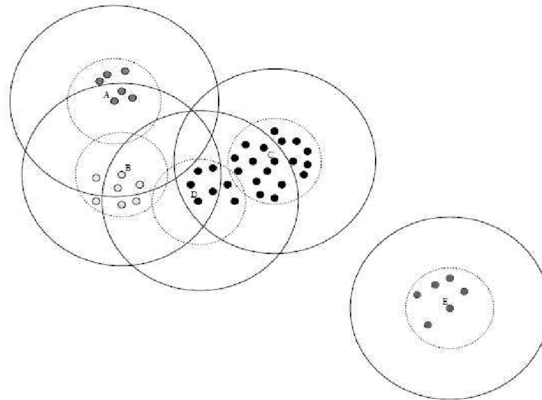
From the common algorithms used for clustering on first sight the hierarchical clustering seems to have potential for parallel processing [20]. According to some research papers these algorithms often have large computational overhead or where the results are acceptable the hierarchical clustering algorithm is applied on not text data.

The light in the tunnel brings the Arrays with Reconfigurable Optical Buses(AROB) and the Parallel Random Access Machine (PRAM) [14]. These algorithms have better performance measures than existing algorithms

However there is one more quite attractive possibility that can speed up the clustering. Canopy clustering [24] is completely new concept for which I heard for the first time don a lecture about MapReduce. MapReduce is a Google software framework that supports distributed computing on large data sets on clusters of computers. The functionality and the name are inspired by the map and reduce functions that are commonly used in functional programming.

The idea is to perform clustering in two stages, first a rough and quick stage that divides the data into overlapping subsets we call canopies, then a final stage in which expensive distance measurements are only made among points that occur in a common canopy. This differs

from other clustering methods in that it uses two different distance metrics for the two stages, and forms overlapping regions.



Clustering based on canopies can be applied to many different underlying clustering algorithms, including K-means[23].

All the very fast distance metrics for text used by search engines are based on the inverted index. An inverted index is a matrix in which, for each word, there is a list of documents containing that word. When we want to found all documents close to a given query, we do not need explicitly to measure the distance to all documents in the collection, but need only examine the list of documents associated with each word in the query. The documents, which have no words in common with the query will never be considered. Thus we can use an inverted index as a distance metric that is based on the number of words two documents have in common.

9.2. Clustering with Semantic vectors

In the first part of this work we tried to examine the robustness of the main features of the Semantic Vectors package. The results were encouraging and we decided to investigate the possibility of real applications deployment. We needed an application that is both solving up to date issue and related to the area of document comparison.

With the increasing number of documents and web resources accessible on Internet the document clustering of documents becomes very attractive. The enormous number of results provided by the modern searching machines implies the need of categorization. An example of this idea is “Clusty” (http://clusty.com/)[21], a searching engine developed in Carnegie Mellon University that is trying to discover the most important word and phrases for each result and put it in a separate category named according to those core expressions. The author of the

project expects that this feature will be very attractive to the user and will make the searching easier and more efficient.

Our initial goal was to try simulating the search and the categorization process as a part of a searching engine.

For a document set for the search we carefully selected documents and separate them in ten categories. After that we started a testing process, which aim was to discover the documents that comprise the core of each the category. We assumed for core documents, those documents that are similar at least to for other document from the same category according to the Semantic Vectors package result search. All documents that were not in this range were dismissed.

Given that now we had well preformed categories we could go for the next step of the test. In order to check the capability of the Semantic Vectors package to perform clustering we were to pull out two files from each category. The set collected this way will use for input of a cluster algorithm.

The algorithm that was to determine witch input file to witch category will belongs is as most intuitive and simple. The separation was to be done according to the number of files from certain category in the result set of documents. The category that comprises the biggest number of file from the result set will be destination folder of the input file.

For the purposes of the test a naming convention was used for naming the files and the directories. This way from the name was clear to witch category each file belongs. This approach was to help as a lot with estimating the results of the test.

The estimation method was based on how many files were categorized to different folder than the folder that they used to belong.

During the tests a weird behavior of the Semantic vectors package was noticed. After recreating both index files, the result of the same search were similar, but not the same. This made as worry, weather the Semantic vector package will manage the clustering task due to this light variations of the results.

To estimate the results we start the test ten times and average mishints were less then 10 % that we take as acceptable.

The approach that we used actually does not implement a real clustering, because we used predefined categories[9][10]. What we did is more or less categorization[19] but according to the results we can claim that implementation of K-mean algorithm on the base of Semantic Vectors package will perform well. There is only one crucial issue ahead of us where we could face problems and this is the speed. The

combination of K-means and Canopy that was discussed above can probably solve this issue.

Chapter 10

10. Summing up

At the end we will try to make a summary of the benefits that brings this work.

At first place we hope that the results of the tests and the applications that were created during preparing this work will be valuable for other researches working in the area of the Information retrieving and especially for those who intend to use the Semantic Vectors package or Lucene's class MoreLikeThis.

With MoreLikeThis we showed that obtaining a feedback from the user is not "causa perduta". Collecting this data and use it in similar searches can be a topic for a future work.

We also managed to show that Semantic Vectors package is a reliable software product that can be benefit for the academic community and with its scalability implies commercial usage as well. Semantic Vector package could help researches and developers to concentrate their efforts on the linguistic properties of source text. There were several light problems that we provided to the authors of the package as a feedback. We did appreciate their fast and detailed replies that once more convinced us that there is not doubt that the project is run by team full of enthusiasm. We hope that this communication was useful for both sites and will help Semantic Vectors package to come up with new ideas and push the limits further.

Appendix A

The search scoring in Lucene is defined in the abstract class Similarity. DefaultSimilarity.java inherits Similarity and implements a default weight scheme for scoring the queries and documents.

The score of query q for document d is compute as cosine-distance or dot-product between document and query vectors in a Vector Space Model (VSM) of Information Retrieval. A document whose vector is closer to the query vector in that model is scored higher. The score is computed as follows:

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

The detail description of the default implementation from the DefaultSimilarity class follows:

- **tf(t in d)** computes the term's frequency, defined as the number of times term t appears in the currently scored document d. Documents that have more occurrences of a given term receive a higher score. The default computation for tf(t in d) in is:

$$\textbf{tf(t in d)} = \textbf{Frequency}^{1/2}$$

- **idf(t)** stands for Inverse Document Frequency. This value is equal to the inverse of docFreq (the number of documents in which the term t appears). This means rarer terms give higher contribution to the total score. The default computation for idf(t) in is:

$$\textbf{idf(t)} = 1 + \log\left(\frac{\textbf{numDocs}}{\textbf{docFreq+1}} \right)$$

- **coord(q,d)** is a score factor based on how many of the query terms are found in the specified document. Typically, a document that contains more of the query's terms will receive a higher score than another document with fewer query terms. This is a search time factor computed in coord(q,d) by the Similarity in effect at search time.
- **queryNorm(q)** is a normalizing factor used to make scores between queries comparable. This factor does not affect document ranking (since all ranked documents are multiplied by the same factor), but rather just attempts to make scores from different queries (or even different indexes) comparable. This is a search time factor computed by the Similarity in effect at search time. The default computation is:

$$\text{queryNorm}(q) = \text{queryNorm}(\text{sumOfSquaredWeights}) = \frac{1}{\text{sumOfSquaredWeights}^{1/2}}$$

The sum of squared weights (of the query terms) is computed by the query Weight object. For example, a boolean query computes this value as:

$$\text{sumOfSquaredWeights} = \text{q.getBoost}()^2 \cdot \sum_{t \text{ in } q} \left(\text{idf}(t) \cdot \text{t.getBoost}() \right)^2$$

- **t.getBoost()** is a search time boost of term t in the query q as specified in the query text, or as set by application calls to setBoost().There is really no direct API for accessing a boost of one term in a multi term query, but rather multi terms are represented in a query as multi TermQuery objects, and so the boost of a term in the query is accessible by calling the sub-query getBoost().
- **norm(t,d)** encapsulates a few (indexing time) boost and length factors:
  - **Document boost** - set by calling doc.setBoost() before adding the document to the index.
  - **Field boost** - set by calling field.setBoost() before adding the field to a document.
  - **lengthNorm(field)** - computed when the document is added to the index in accordance with the number of tokens of this field in the document, so that shorter fields contribute more to the score. LengthNorm is computed by the Similarity class in effect at indexing.

When a document is added to the index, all the above factors are multiplied. If the document has multiple fields with the same name, all their boosts are multiplied together:

$$\text{norm}(t,d) = \text{doc.getBoost}() \cdot \text{lengthNorm(field)} \cdot \prod_{\text{field } f \text{ in } d \text{ named as } t} \text{f.getBoost}()$$

However the resulted norm value is encoded as a single byte before being stored. At search time, the norm byte value is read from the index directory and decoded back to a float norm value. This encoding/decoding, while reducing index size, comes with the price of precision loss - it is not guaranteed that decode(encode(x)) = x. For instance,

decode(encode(0.89)) = 0.75. Also that search time is too late to modify this norm part of scoring, e.g. by using a different Similarity for search.

References

[1] Dominic Widdows, Kathleen Ferraro, Semantic Vectors : A Scalable Open Source Package and Online Technology Management Application University of Pittsburgh, 2008.

[2] Magnus Sahlgren, Anders Anders Holst, Permutations as a Means to Endcode Order in Word Space. Available on : http://www.sics.se/~mange/papers/permutationsCogSci08.pdf (02.2009) Swedish Institute of Computer Science, Kista, Sweden.

[3] Marnus Sahlgren, The Word – Space Model . Department of Linguistics, Stockholm University (2006).

[4] Magnus Sahlgren , An Introduction to Random Indexing Swedish Institute of Computer Science, 2005. http://www.sics.se/~mange/papers/RI_intro.pdf (04.2009)

[5] Landauer, T. K., Dumais, S. T, The Latent Semantic Analysis theory of the acquisition, induction, and representation of knowledge (1997). http://lsa.colorado.edu/papers/plato/plato.annote.html

[6] Magnus Sahlgren, Towards pertinent evaluation methodologies for word-space models, SICS, Swedish Institute of Computer Science, Kista, Sweden (2006). http://www.sics.se/~mange/papers/lrec2006.pdf (04.2009)

[7] Amit Singhal ,Modern Information Retrieval, Google, Inc (2001). http://singhal.info/ieee2001.pdf (04.2009)

[8] Nicholas J. Belkin, W. Bruce Croft, Information filtering and information retrieval: Two sides of the same coin?(1992). http://www.ischool.utexas.edu/~i385d/readings/Belkin_Information_92.pdf

[9] David D. Lewis, An evaluation of phrasal and clustered representations on a text categorization task, Center for Informaiton and Language studies University of Chicago.

[10] ZHOU Qiang, ZHENG Yabin, Integrate Text Clustering Features in Text Categorization System, Dept. of Computer Science and technology, Tsinghua University, Beijing.

[11] Virginia Kleman J. Laub, The Singular Value Decomposition:Its Computation and Some Applications.

[12] S. Deerwester, S Dumais, Indexing by latent semantic analysis, Journal of the American society for information science, 1990.

[13] M.F. Porter, ‚An algorithm for suffix stripping, *Computer Laboratory, Cambridge, UK.*

[14] Sanguthevar Rajasekaran, Efficient Parallel Hierarchical Clustering Algorithms.

[15] Kristina Lerman, Document Clustering in Reduced Dimension Vector Space, Information Sciences Institute.

[16] Makoto Iwayama, Cluster - Based Text Categorization: A Conparison of Category Search Strategies, Department of Computer Science, Tokyo.

[17] Eric C. Jensen, Steven M. Beitzel, Parallelizing the Buckshot Algorithm for Efficient Document Clustering, Illinois Institute of Technology, Chicago.

[18] Sugato Basu, Semi-supervised Clustering: Probabilistic Models, Algorithms and Experiments, The University of Texas at Austin.

[19] Paolo Rosso, Edgardo Ferretti, Text Categorization and Information Retrieval UsingWordNet Senses, National University of San Luis, Argentina.

[20] Manoranjan Dash, Simona Petrutiu, Peter Scheuermann, Efficient Parallel Hierarchical Clustering, Department of Information Systems, Nanyang Technological University, Singapore, 2004.

[21] Florian Beil, Martin Ester, Xiaowei Xu, Frequent Term-Based Text Clustering, Institute for Computer Science Ludwig-Maximilians-Universitaet Muenchen Munich, Germany.

[22] Magnus Rosell, Introduction to Information Retrieval and Text Clustering, Magnus Rosell, 2006.

[23] Paul S. Bradley, Usama M. Fayyad, Refining Initial Points for K-Means Clustering, Microsoft Research, 2006.

[24] Andrew McCallum, Kamal Nigam, Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching, School of Computer Science Carnegie Mellon University Pittsburgh, PA USA.

[25] Michael W. Berry, Susan T. Dumais and Gavin W. O'Brien,Using Linear Algebra for Intelligent Information Retrieval, Society for Industrial and Applied Mathematics, 1995.

[26] Karen Spärck Jones, A statistical interpretation of term specificity and its application in retrieval, Computer Laboratory, University of Cambridge, Cambridge, UK, 1972

[27] Justin Zzobel, Alistair Moffat, Inverted Files for Text Search Engines, The University of Melbourne, Australia.

[28] Amit Singhal, Chris Buckley, Mandar Mitra, Pivoted Document Length Nornalization, Department of computer science, Cornell University, Ithaca.

[29] Asuncion Honradot, Ruben Leon A Word Stemming Algorithm for the Spanish Language, Laboratorio de Linguistica Informitica, Universidad Autonoma de Madrid.

List of attachments

Attachment 1: CD.