Zeeker: A topic-based search engine

Magnús Sigurðsson
Søren Christian Halling

Kongens Lyngby 2007
IMM-2007-91

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Preface

This thesis is submitted as a fulfillment of the requirements for the degree of Master of Science in Engineering at the Technical University of Denmark (DTU), Lyngby, Denmark.

Authors are Søren Christian Halling (s021797) and Magnús Sigurðsson (s021980) under the supervision of Prof. Lars Kai Hansen of the Intelligent Signal Processing group at the Department of Informatics and Mathematical Modeling (IMM), DTU.

The thesis was completed at the Department of Informatics and Mathematical Modeling during the period April 2007 - October 2007.

Kongens Lyngby, October 2007

Søren Christian Halling (s021797)

Magnús Sigurðsson (s021980)

Acknowledgments

During the process of writing this thesis, numerous people deserve recognition for their contributions, time and knowledge.

First of all we would like to thank our supervisor, Prof. Lars Kai Hansen, for his constant enthusiasm and ability to guide us in the right direction when we were faced with difficult problems. We would also like to thank Anders Meng for very inspirational input at the early stages of our work.

Furthermore, we would like to thank Helga Björk Magnúsdóttir and Thelma Björk Gísladóttir for their thorough review of this thesis and constructive comments. Our fellow student and friend, Søren Knudsen, we thank for countless discussions and invaluable input.

Finally, we would like to express our gratitude to the people, who took the time and effort to test our search engine and fill out the questionnaire.

Thanks guys, your help is greatly appreciated.

Resume

Internettets hastige vækst og de massive datamængder har forøget betydningen og kompleksiteten af informations søgning. Mængden og forskelligheden af data introducerer brist i de metoder søgemaskiner bruger til rangering af resultater. Et resultat af dette er at søgemaskine optimerings-virksomheder blomstrer ved at udnytte søgemaskinernes svagheder og derved manipulerer med søgeresultater. Derudover præsenterer mange søgemaskiner brugeren for millioner af resultater ved søgninger hvor disse resultater ofte er skævvredet mod blogs og net butikker. Denne skævvridning stammer fra den link analyse, der ligger til grund for søgeresultaterne. Internettets link struktur er både styrken men også akilleshælen ved brugen af link analyse til rangering af søgeresultater.

I dette speciale foreslås at ændre søgemaskinernes adfærd, væk fra link analyse og hen mod en analyse af websiders faktiske indhold. Ved hjælp af datagrupperings algoritmer og den enorme mængde af information i Wikipedia er ideen at bygge kategorier, der er specifikke nok til at filtrere i søgeresultater. Udsigten til at lade brugeren filtrere søgeresultater ved et tryk på en knap vil forbedre relevansen af søgeresultaterne. Kategorierne vil give brugeren færre men mere relevante resultater.

De veldefinerede kategorier og artikler i Wikipedia har vist sig at være værdifulde som trænings sæt ved gruppering af Internet data. Den implementerede *Zeeker Search Engine* har præcise kategorier, hvilke kan forbedres væsentligt ved at drage nytte af yderligere information fra Wikipedia.

En brugerundersøgelse har vist at *Zeeker Search Engine* har høj relevans ved informations søgning, er nem at bruge og har stort potentiale som søgemaskine. Dette projekt har givet innovative ideer og måder at bruge information fra Wikipedia til at producere gode kategorier og finde relevante søgeresultater.

Nøgleord: Søgemaskine design, Informations søgning, Sprogteknologi, Data gruppering, Wikipedia, *Zeeker Search Engine*.

Abstract

The rapid growth and massive quantities of data on the Internet have increased the importance and complexity of Information Retrieval. The amount and diversity of data introduce shortcomings in the way search engines rank their results. As a result, Search Engine Optimization companies flourish by exploiting the search engine weaknesses in order to manipulate the search results. Furthermore, many search engines present several million results to queries and more often or not these results are biased toward blogs and on-line stores. This bias is due to the link analysis used to rank the search results. Internet link structure is the strength but at the same time the Achilles' heel of these ranking algorithms.

In this work it is proposed to push search engine behavior in a new direction, away from link analysis and toward actual content and topic analysis of web pages. With the use of clustering algorithms and the vast amount of information in Wikipedia the idea is to create categories that are good enough to be used to filter search results. The prospect of letting the user filter the search results by the push of a button would improve the relevance of the search results for a particular query. Categories will give the user fewer yet more relevant results.

The well-defined categories and articles of Wikipedia are shown to be valuable as a training set when clustering Internet data. The implemented *Zeeker Search Engine* has precise categories which can become even better by taking advantage of additional information available in Wikipedia.

A user-survey conducted has revealed that *Zeeker Search Engine* has good relevance when retrieving information, is easy to use and has great potential as a search engine. This work has suggested innovative ideas and ways of using the information in Wikipedia to produce good categories and retrieve more relevant search results.

Keywords: Search engine design, Information Retrieval, Natural Language Processing, Clustering, Wikipedia, *Zeeker Search Engine.*

Contents

Preface i

Acknowledgments iii

Resume v

Abstract vii

1 Introduction 1
1.1 Motivation: A new way of searching . . . 1
1.1.1 Known search engine problems . . . 1
1.2 Search engine anatomy . . . 3
1.2.1 Document processing . . . 4
1.2.2 Data Indexing . . . 9
1.2.3 Query Processing . . . 13
1.2.4 Ranking relevant results . . . 15
1.2.5 Summary . . . 19
1.3 Problem description . . . 19
1.3.1 Reading guide/Overview . . . 21

I Data Store and Preprocessing 23

2 Data store 25
2.1 Wikipedia . . . 25
2.2 Data store structure . . . 28
2.3 Test sets . . . 29
2.4 Summary . . . 31

3 Data processing 33
3.1 Processing data . . . 33
3.1.1 Index reduction . . . 33
3.1.2 Index size . . . 34
3.1.3 Tests . . . 34
3.2 Summary . . . 36

II Clustering 39

4 Clustering 41
4.1 Vector Space Model 41
4.1.1 Cosine Similarity Measure 42
4.1.2 Term weighting (TF x IDF) 43
4.1.3 Summary 44
4.2 Clustering 45
4.2.1 Overlapping vs. non-overlapping 46
4.2.2 Types of algorithms 46
4.3 Machine learning 47
4.4 Summary 49

5 Spherical *k*-means 51
5.1 Document representation 51
5.2 Algorithm 52
5.2.1 Cluster quality 53
5.2.2 Clustering step by step 54
5.3 Summary 55

6 Nonnegative matrix factorization (NMF) 57
6.1 Standard NMF-Algorithm 58
6.1.1 Initial problem 58
6.1.2 Updating rules 59
6.2 Summary 60

7 Frequent term-based text clustering (FTC) 61
7.1 Definitions 61
7.2 Algorithm 62
7.3 Summary 63

8 Clustering discussion 65
8.1 Algorithm choices 65
8.2 Algorithm pros, cons and similarities 66
8.2.1 NMF discussion 66
8.2.2 Spherical *k*-means discussion 66
8.2.3 Algorithm similarities 67
8.3 Current state of affairs 68

9 Clustering tests 71
9.1 Data set 71
9.2 Test strategy 72
9.3 Quality measure 72
9.4 Test results 73
9.4.1 Cluster quality and baseline measures 73
9.5 Discussion 75
9.6 Summary 77

III Retrieval 79

10 Retrieval 81
10.1 Query Processing . . . 81
10.1.1 Vocabulary pruning . . . 81
10.1.2 Misspelled queries . . . 82
10.2 Query operators . . . 82
10.2.1 Search operators . . . 82
10.2.2 Category filtering . . . 83
10.2.3 Query expansion . . . 84
10.3 Document retrieval . . . 85
10.3.1 Retrieval method . . . 85
10.3.2 Ranking . . . 85
10.3.3 Lack of results . . . 86
10.4 Summary . . . 86

11 Evaluating retrieval 89
11.1 Recall, Precision and F-measure . . . 89
11.2 User Feedback . . . 90
11.3 Summary . . . 91

12 Testing retrieval 93
12.1 Test strategies . . . 93
12.1.1 Selected test methods . . . 94
12.2 Test discussion . . . 94
12.2.1 Searching . . . 95
12.2.2 Overall evaluation . . . 97
12.3 Summary . . . 98

IV Implementation 101

13 Implementation 103
13.1 Lemur and Indri . . . 103
13.2 Zeeker Applications . . . 105
13.2.1 *Zeeker.Spider* and *Zeeker.DataGateway* . . . 105
13.2.2 *Zeeker.Base* (database) . . . 105
13.2.3 *DataManagementWizard* . . . 106
13.2.4 *BuildIndex* . . . 107
13.2.5 *Clustering* . . . 108
13.2.6 *Zeeker.Website* . . . 108
13.2.7 Test programs . . . 109
13.3 Summary . . . 109
13.3.1 Data flow . . . 110

V Conclusion 113

14 Future work 115
14.1 Known issues . . . 115
14.2 Future features . . . 116

14.3 Summary 117

15 Conclusion **119**

VI Appendix **123**

A User Guide **125**
A.1 Query syntax 125
A.2 Screen-shots 127

B Tests **129**
B.1 Test indexes 129
B.2 Questionnaire 131

C POS Tagging **139**
C.1 POS tagset 140

D Stopwords **143**

List of Tables

1.1 Punctuation groups 5
1.2 Inverted index 11
1.3 Full inverted index 11

2.1 Data store statistics 31

3.1 Included tags in index 34

9.1 Clustering statistics 75

12.2 Survey: How difficult was it to find? 96
12.3 Survey: Selected comments 97
12.4 Survey: Did you find the categories useful? 97
12.5 Survey: How relevant were the results to your queries? 97
12.6 Survey: How do you rate our search engine's overall performance? 98
12.7 Survey: How likely are you to use this kind of search engine again? 98

B.1 Test indexes - full index 129
B.2 Test indexes - POS tagged 130
B.3 Test indexes - Stopped and stemmed 130
B.4 Test indexes - POS tagged, stopped and stemmed 130

C.1 Test on distribution of the tags in the Brown/Penn-style tagset . 139
C.2 Variant of the Brown/Penn-style tagset - Part I 140
C.3 Variant of the Brown/Penn-style tagset - Part II 141

D.1 Stop word list (a-h) 144
D.2 Stop word list (i-y) 145

List of Figures

1.1 Linearization of table-based Web content 9
1.2 Important steps in the document processor 10

2.1 Wikipedia category and article hierarchy overview 27
2.2 Data intersection 28
2.3 Expansion of the data store 30
2.4 Test sets structure 30

3.1 Number of unique terms in documents 36
3.2 Document processing 38

4.1 Cosine simularity 42
4.2 Clustering example 45
4.3 Supervised vs. unsupervised learning 48

5.1 Normalizing vector space 52
5.2 Spherical *k*-means 54

6.1 Difference between NMF and LSI 58

9.1 Spherical *k*-means baseline- and cluster-quality 74
9.2 Wikipedia and Spherical *k*-means cluster quality 74
9.3 Wikipedia overlapping clusters 77

13.1 Zeeker Search Engine data and application flow 104

A.1 Zeeker Search Engine front-end 127
A.2 Zeeker Search Engine result page 128

Glossary

Notation	Description
bag-of-words	text is regarded as a bag of words - no compositional semantics are used. One word equals one term
Clustering	The process of finding natural groups in data
Compositional Semantics	The meaning of words in combination given by the rules used to combine the words
FTC	Frequent Term-based Clustering
ICA	Independent Component Analysis
Lexical Semantics	The meaning of a single word
LSI	Latent Semantic Indexing
NMF	Non-negative Matrix Factorization
NP-complete	Non-deterministic Polynomial-time problem (Problem that cannot be solved in polynomial time)
ODP	Open Directory Project
PCA	Principal Component Analysis
PLSI	Probabilistic Latent Semantic Indexing
Polysemy	A word with more than one meaning
POS	Part-of-Speech
QE	Query Expansion
SEO	Search Engine Optimization
Stemming	The process of reducing words to their stems
Stop Words	Words that add no semantic meaning to a sentence

Notation	Description
Tf x Idf	Term-frequency Inverse-document-frequency (term weighting scheme)
Vocabulary Pruning	The process of removing unnecessary terms from vocabulary
VSM	Vector Space Model
Wikipedia	Free online Encyclopedia

Chapter 1

Introduction

1.1 Motivation: A new way of searching

When using the best available search engines, we often find that they lack quite a bit in terms of actually providing the information needed for a given search query.

Today a whole industry of *"search engine optimization"* (SEO) companies have emerged with no other mission than to manipulate the search engine results using link farms, cleverly chosen meta information[1] etc. These tricks give their customers' web pages a better ranking in the search results[2]. On top of the enormous amounts of redundant information these companies provide, one usually gets as many as ten million results when searching. Many of the resulting web pages only contain one word from the search query but that does not mean the page has any useful information regarding this word or topic. Very few have the time, or desire, to go through all the search results to find relevant information. Often users merely skim the top $10-20$ results and choose the web pages that look promising before refining their queries. If the most relevant web page for a given search query is ranked 30 or lower, most users would probably not find it, at least not the first time around. Cheating search engines and most of all; getting too much useless information, is, as we see it, the biggest problem with search engines today.

1.1.1 Known search engine problems

When discussing search engine problems the place to start is Google. Not because Google is the worst search engine but because Google is the best search engine out there. The problems Google and its PageRank [10, 23] algorithm face, and have not yet conquered, other search engines face as well. Therefore, we will focus our attention on Google when describing some of the problems we

[1] Such as including popular keywords as hidden text

[2] Placing the web page higher in the list of results

find important to solve.

Whenever searching for something that can be sold on-line, Google's results skew toward on-line stores. Searching for *flowers*, eight of the top ten results are on-line stores trying to sell flowers. Wikipedia's article describing flowers is number three and a gallery is number eight[3]. These results are good if the user wanted a mothers day gift but if he/she wanted gardening information, a search within the search results would be necessary. Google creates a skewness in search results when people mention a product on their web page and link to a store selling the product. These *service links* generate enormous weights to stores because the Google PageRank algorithm deems pages with many links pointing to it important.

Another skewness occurs on synonyms. For example, searching for information regarding apples, a search using merely the term *apple* does not work. After skimming more than one hundred of the top results from Google, we had not found anything but links to Apple Inc. The PageRank algorithm finds Apple Inc. to be very important because of the many links to Apple.com whenever a web page mentions iPhone, iPod etc.

Again, skewness occurs when a certain topic is discussed by many people on blogs and forums. These threads tend to make it to the top of the search result list instead of the pages actually containing information describing the topic. This means that one gets pages with discussions of a topic but not pages defining the topic. Terms can even get different meanings after they have been through Google's algorithms - called *Google wash*. Using the example with the search query on *apples* the results deal with Apple Inc. not the fruit. This is an example of Google Wash. Effectively this means that results on Google refer to a set of opinions and certain uses of words, not necessarily the true meaning of the words. This can be seen in many queries where blogs and forum threads enter the top ten results instead of the definition of the search words.

It has been reported (unofficially) that Google only indexes the first 100 KB of web pages, probably due to the fact that the Internet is too big to store, even on Google's hardware. The problem with this restriction comes into effect if the relevant information is stored somewhere beyond those 100 KB, thus making it impossible to search for.

Many of these problems such as biased results toward on-line stores, topic skewness etc. arise from the fact that Google puts a lot of faith into the link structure of the Internet. It is assumed that a link points to some related information and that the link's anchor text describes that information. This is not always the case - and when big money is involved and there is a weakness in the search engine's structure, someone is going to take advantage of it. These weaknesses are used by link farms, *Google Bombs* etc. to get a particular web page higher in the result list.

In the discussion above we have identified some potential problems with

[3]Search query submitted May 2nd, 2007

search engines today which we would like to make better. However, before we start implementing a new search engine, we need to analyze how a search engine works and what issues need to be addressed. In the rest of this introduction we will discuss and analyze some of the problems and choices we need to make when implementing our search engine.

1.2 Search engine anatomy

The inner workings of a search engine is very complex and versatile. This section will describe the general data-flow in a search engine and analyze some of the problems arising when doing data processing, data indexing and query processing.

Based on the data-flow in a standard search engine, some common elements are needed to construct a search engine. The data-flow is as follows:

1. First a **Webcrawler** is used to download pages from the web and store them in the search engine's database. These pages are defined as *documents*.

2. When data is present in the database, a **Document Processor** parses the documents and formats them before indexing can take place.

3. An **Indexing Service** takes the parsed and formatted data and creates an index. The Indexing Service only indexes items that have been identified as relevant[4] thus making the data ready to be searched. These *items* are defined as *terms*.

4. Finally, a **Query Processor** processes the queries from users and searches the database for matches and presents the relevant results to the user.

Based on the above, the construction of a search engine can be split into three different categories:

- Document Processing
- Data Indexing
- Query Processing

The *Document Processing* deals with data preprocessing, *Data Indexing* handles the appropriate indexing of the downloaded data and *Query Processing* submits queries and retrieves the results from the search engine. These elements will be discussed in more detail in the following sections. The *Webcrawler* used in this work, called *Zeeker.Spider*, was implemented in a separate project[5] and will not be discussed in further detail here. Suffice to say it meets the requirements of a standard webcrawler.

[4] E.g. words, phrases, numbers, names, etc.

[5] By Søren Halling and Magnus Sigurdsson at the Danish Technical University in February and March 2007

1.2.1 Document processing

Since text cannot be directly interpreted by a search algorithm, an indexing procedure needs to map the text to a proper representation of its content. The document processor does exactly that. It prepares, processes, and inputs the documents (web pages or sites) that the search engine will add to its index. Many problems arise when processing large amounts of data. Even the simplest tasks can become complicated when data is of enormous size. Further, the parsing of data is difficult as there is very little (if any) structure in the various documents on the Internet. Just about anything goes "out there".

When a document processor is implemented, various problems arise regarding parsing and identifying indexable elements (terms). This section discusses how these problems could be addressed.

How a document processor represents text is a choice of which elements of the text it finds meaningful (*lexical semantics*) and what combinational rules it finds meaningful for these elements (*compositional semantics*). Usually, the compositional semantics of text is disregarded[36] and text is represented as a histogram of terms that occur in the text, hence only keeping the lexical semantics.

To create the term histogram, the document processor performs some or all of the following steps (see also figure 1.2.1 on page 10):

1. Normalizes the document stream to a predefined format.
2. Breaks the document stream into desired retrievable units.
3. Identifies potential indexable elements in documents.
4. Deletes stop words.
5. Reduces terms to their stems.
6. Extracts index entries.
7. Computes term weights.

These steps are very important in the process of creating a search engine as the terms the document processor identifies are the terms that later can be searched for.

Text is extremely dynamic and can contain hundreds of thousands of characters, symbols, punctuations, digits etc. As a result there are many problems that need to be addressed when dealing with such data and trying to identify indexable elements. Besides the different parsing rules and handling of data, the document processor needs to ease the load of later calculations by only selecting the terms that are important and relevant without loosing too much valuable information. This trade off is one of the real challenges in implementing a good document processor.

Punctuation example	**Result**
x:id, x;ix, x.id	x<a>id
x’id, x”id, xîd	x<b>id
x(id, x[id, x{id	x<c>id
x((id, x[(id, x{(id	x<c>id

Table 1.1: Punctuation groups

Normalizing and identifying indexable terms

First of all the text has to broken into terms and the obvious way would be to break on white spaces - i.e. define a single word as a term. This approach is called *bag-of-words* as text is seen as a bag of words, thus disregarding compositional semantics.

Punctuations can divide sentences but are also used in many other contexts, e.g. variable names in program code such as ’x.id’ or in ’510B.C.’. Punctuation cannot be removed uncritically from a sentence thus giving ’xid’ and ’510BC’. One solution could be to create punctuation groups such that punctuations would be replaced by special character sequences, e.g. ’<x>’ in the index giving ’x<x>id’ and ’510B<x>C’ making searches on ’x:id’, ’x;ix’, ’x.id’ etc. mean the same thing. The groups could be refined even further by generating several groups as shown in table 1.1, thus minimizing the number of indexed terms.

To avoid loosing too much of the individual words and adding to processing complexity it is advised to use white space as delimiters and remove punctuations only if a word begins or ends with a punctuation.

When finding indexable terms it is difficult to say when a word or group of words are important enough to be indexed. The case of the word says a lot about its importance and a decision whether ’PET’ and ’pet’ should mean the same thing has to be made. An easy approach is to add more importance to upper case words as they tend to be abbreviations of organizations, terms and/or concepts.

Nouns usually carry most semantic weight and other word groups could be removed. In [4] it is suggested that nouns situated side by side could be placed in noun groups (or *compound term*) and not just single terms. For example a sentence like ’In computer science we usually...’, ’computer science’ could be indexed as a single compound term. Syntactic distance between nouns, meaning the distance between two nouns where they are still considered a compound term, is suggested to be at a threshold of three. Only relying on nouns seems to disregard to much information and a term histogram of a mix of terms and compound terms is therefore preferred.

Digits and dates present yet another problem. In some cases it might be very useful to be able to search for a given year or date. Some dates and years are very important - e.g.: ’9/11’. Dates and years could be normalized and indexed, although it would require some parsing as dates can be in many formats. If digits are removed, searching for a specific year is not possible, but most of the time that does not make a lot of sense anyway as a year normally can not be connected with a single searchable item e.g. search query ’2000’ makes no

sense. One idea is to treat years as nouns and thus include them in the compound terms. If a noun and a year are syntactically close, the year may be important - e.g. searching for 'exploration 1492'. The danger of including digits is that it could lead to an extremely long term list in the index if documents contain a lot of distinct numbers.

Spell checking all terms and compound terms found is very difficult. Even if it is assumed that names start with a capital letter so they can be identified, there are plenty of other character constellations that do not fit spelling rules. Again, the program variable 'x.id' is a good example. No dictionary would allow such a term. Misspellings might be removed by their frequency, i.e. that terms which appear less than a given minimum threshold are removed (hopefully misspellings are rare). Likewise, very common terms could be removed if they appear more than a given maximum threshold.

Stop words - to be or not to be

Within the fields of Information Retrieval and Natural Language Processing, *stop words* are words that add nothing to the precision of a search query or to the semantics in the index. If a term occurs in more than 80% of documents it should be considered a stop word [4]. This includes a pre-calculation of the distribution of words in the documents retrieved. Such a calculation can be avoided by the use of a stop word list, also called a negative dictionary.

[42] lists 425 stop words and [17] lists 421 stop words found through analysis of general English texts. Stop words present a problem when searching for a phrase like 'to be or not to be'. Most likely, all words in such a search query would be removed as stop words.

Removing stop words does reduce calculation efforts - if 425 words are removed and one million documents are indexed, $425 * 10^6$ positions in the index are removed. If a single position is represented by only one bit,[6] it reduces memory usage by

$$\frac{425 * 10^6}{(1024 * 1024 * 8)} \approx 50 \text{ MB}$$

If represented by an integer it could get up to 32 times as big. Stop words reduce the document word counts considerably, but do not reduce the length of the term list in a significant way.

Lexical Analysis and Phrases

When trying to generalize text categorization, lexical analysis is an important tool. WordNet[7] is a lexical database containing lexical concepts that have been and are being used to improve classifications significantly.

The use of Part-Of-Speech (POS) taggers has shown a great improvement on text categorization and generalization[30]. POS taggers analyze the sentences and tag terms with their syntactical groups such as verbs, nouns, numbers,

[6] Usually represented as an unsigned integer - 8, 16 or 32 bits

[7] http://wordnet.princeton.edu/

punctuations etc. Hence, words with a weak contextual meaning can be distinguished from similar words with a stronger contextual meaning.

Another important aspect of lexical analysis is phrases. To be able to find phrases and include them in a term histogram, a predefined list of phrases needs to be available such that terms can be matched in the list. If such a list can be obtained or built, phrases should be included to obtain better results.

The synonymy part of WordNet has been used to expand the term list for each text category with good results[13] but attempts have been made to classify text based on word meanings with no significant improvement in accuracy[21]. When using POS taggers it has been shown that the vocabulary (term histogram) can be reduced by up to a staggering 98% [30] (in some cases: depending on the type of vocabulary).

As described in [30] the vocabulary was reduced significantly by the use of the POS tagger *QTag*. This tagger is a probabilistic part-of-speech tagger that has proved very useful in text categorization and is therefore a good choice if POS tagging is to be used[8].

Stemming

Stemming is used to remove word suffixes (and possibly prefixes). This reduces the number of unique terms and gives a user's search query a better recall. If taking the classical example from textbooks on stemming, words such as analysis, analyzing, analyzer and analyzed all stem to 'analy'. This example shows that stemming introduces an artificial increased polysemy[9]. Without stemming the term histogram could grow to an unmanageable size.

[42] compares benefits from eight stemming projects and finds conflicting results. Therefore, some search engines do not use stemming at all. [18] reports that stemming on average increases performance by 1-3% compared to no stemming and for some queries even better. Further, it is shown overall that prefix removal reduces the result yet specific queries perform better with prefix removal. When using stemming, it is not advised to find a word's 'true' root. Linguistic stemmers are simply not good enough to make such a stemming efficiently at this time.

Several stemming algorithms exist - two of the well known are Lovins [28] and Porter [31]. Both [4] and [18] report that the Porter stemmer is the best algorithm for stemming as is. The algorithm is simple and elegant with comparable results to more sophisticated algorithms.

Term weighting

The idea of term weighting is to give terms different weights when situated in different contexts. For example, if a term is located in a title tag, the term is assigned more weight than a term in a paragraph. Another way is simply to use

[8]Not to mention that QTag is freeware

[9]Polysemy means that a word has more than one meaning e.g. train

the term frequency (how many times does the term appear in a document) or a combination of the two.

There exist many term weighting algorithms, but the one generally used[36] and with consistently good results[34] is the *term frequency-inverse document frequency* ($Tf \times Idf$). $Tf \times Idf$ makes three basic assumptions[14][10]

1. Rare terms are no less important than frequent ones
2. Multiple appearances of a term in a document is no less important than single appearances
3. Long documents are no more important than short ones

Together these assumptions constitute *normalized* $Tf \times Idf$. Term frequency is simply the number of times a given term appears in a document divided by the number of terms occurring, and the inverse document frequency is a measure of the general importance of a term (how many documents does it appear in). Term weighting by $Tf \times Idf$ is described in more details in chapter 4.1.2.

Linearization

Linearization is the web designer's and SEO company's problem that really has nothing to do with how the search engine behaves but how the search engine might perform poorly with some web pages. When the search engine reads a page, it reads it line by line, but that is not always as it is shown on the page when viewed. See the example in figure 1.1 where it is obvious that poor web design (building a web page with tables) might affect the search engine's ability to index the page correctly. The terms *Stores Clients Visit our Partners* make no sense and as table complexity increases the incoherence does to. Linearization has an effect on many things such as term positioning, compositional semantics, phrases, thresholds for noun groups etc.

This is not a problem we will address any further. We only mention it to point out that even when taking all possible precautions there are still problems that search engines have no control over.

Summary

In this section we have listed many problems with regards to document processing but also presented a general analysis of the document processor. The most important steps of document processing can be seen in figure 1.2.1 on page 10. Many of the problems listed above are due to hardware and CPU limitations, considering storing and performing calculations on all documents and terms on the Internet. As these hardware limitations are not anywhere near getting to a feasible point, we need to sieve the data and still preserve the essence such that all the information in the index is searchable. Herein lies the true challenge.

This analysis has shown that such a sieve can be built from the use of stop word removal, stemming, punctuation groups, lexical analysis using POS taggers and other tools to give a satisfactory result. Since hardware and CPUs answer to Moore's law, the holes in the sieve can be made bigger and bigger as time passes.

[10]Practically all weighting methods make these assumptions in one way or another

Before linearization

<table>
<tr><td colspan="3">1
Colorado Pet Shop</td></tr>
<tr><td rowspan="3">2
Products

Services

Company</td><td rowspan="3">3
Books about dogs, cats and birds.</td><td>4
Stores</td></tr>
<tr><td>5
Clients</td></tr>
<tr><td>6
Visit our partners</td></tr>
</table>

After linearization

Colorado Pet Shop Products Services Company Books about dogs, cats and birds. Stores Clients Visit our Partners

Figure adapted from E. Garcia's 'The Keyword density of Non-sense'

Figure 1.1: Linearization of table-based Web content

1.2.2 Data Indexing

As search engines' indexes have grown very fast in the past few years, the Data Indexing part is becoming the most important part of the data processing within a search engine. If data is not indexed properly, the search engine can not be expected to yield good results to search queries. In the previous section we discussed what steps need to be considered in the data preprocessing, prior to the actual indexing and as a result the document processing plays a very important role in the data indexing. The results from the document processing should have identified which terms are to be indexed, but the data indexer must take the final decision of what should be included and what data can be excluded. The naive approach of simply indexing every term that occurs within the downloaded documents would require enormous amounts of data storage, and could also result in a slow response time to queries due to computational inefficiency. In this section we discuss how terms can be indexed in order to get positive results to search queries.

Traditional Indexing

A popular way of creating an index for search engines is to add the terms from the document processor to the index, sometimes calculate term-proximity within the documents and add that information to the index. This way, it is possible to search for combinations in the documents, e.g. for a query like 'computer science', the search engine would rank the documents highest that have the two terms side-by-side. However, this additional information, i.e. the term proximity, makes the index larger and might make the search engine inefficient. This is where the concept of an *inverted file* is introduced. Most traditional search engines use this structure to represent their index with great results.

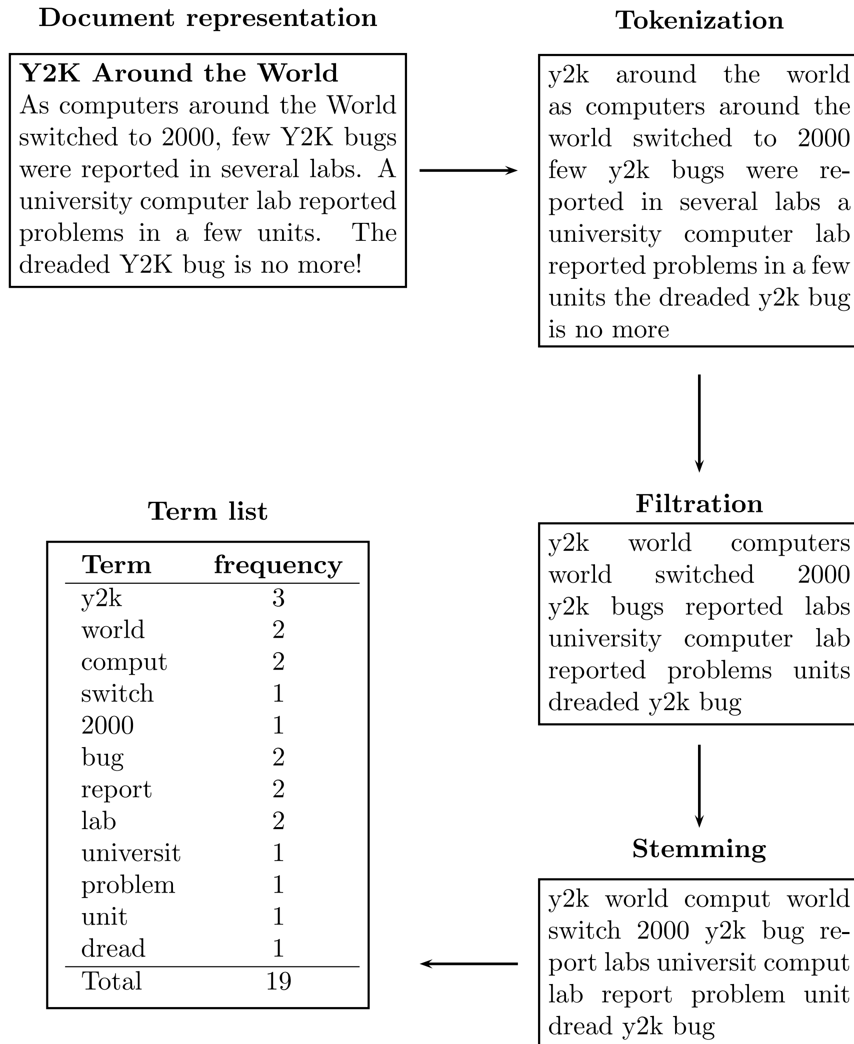

Figure 1.2: Important steps in the document processor[a]

[a]Figure adapted from E. Garcia's 'The Keyword density of Non-sense'

When representing such large amounts of data it is not feasible to list the words per document in an index. Instead an inverted index data structure is used which lists the documents per word. An inverted index is an index structure storing a mapping from words to their locations in a document or a set of documents, allowing full text search. This structure also optimizes the speed of the query as the query can look-up the word and find the documents containing it.

An inverted index has many variants but usually contains a reference to documents for each word and, possibly, also the position of the word in the document. If we have the set of texts:

$$T = \{\tau_0 = \textit{i love you}, \tau_1 = \textit{you love i}, \tau_2 = \textit{love is blind}, \tau_3 = \textit{blind justice}\},$$

we get an inverted index as can be seen in Table 1.2.

Word/Term	index information
i	{ 0, 1 }
love	{ 0, 1, 2 }
you	{ 0, 1 }
is	{ 2 }
blind	{ 2, 3 }
justice	{ 3 }

Table 1.2: Inverted index

When searching for the words *love* and *blind* we get the result set $\{0, 1, 2\} \cap \{2, 3\} = \{2\}$.

A full inverted index can be created from the same text set T by adding the local word number giving a full inverted index as seen in Table 1.3

Word/Term	index information
i	{ **(0, 0)**, (1, 2) }
love	{ **(0, 1)**, (1, 1), (2, 0) }
you	{ **(0, 2)**, (1, 0) }
is	{ (2, 1) }
blind	{ (2, 2), (3, 0) }
justice	{ (3, 1) }

Table 1.3: Full inverted index

When searching for the phrase *i love you* we get hits for both τ_0 and τ_1, but if we use the positioning we will only get τ_0 as seen in bold in Table 1.3.

The index might also include details such as:

- Position of a word
- Position of the starting character
- Term weights

- Term frequencies

Clustering

Another possible way of indexing or improving an index, is to arrange documents into clusters, or categories. In this way, the documents dealing with similar or the same subject would be placed in the same category. Using categorization, it would be possible to only index the terms that are descriptive for each category. This would decrease the size of the term histograms and in turn also decrease the size of the index.

Using a clustered index can hopefully give us a more detailed index, which in turn would give more precise and relevant search results. For example, if a query for 'computer science' was submitted, the search engine would try and predict in which of the predefined categories the search terms are a part of. Then the search engine could search within these categories and return the results categorized and by relevance. However, this categorization comes with added computational effort since the clustering requires additional information stored in the inverted file structure and not to mention the calculation of the clusters themselves.

Essentially, the clustering could be yet another detail in our inverted file as mentioned above. Using clustering as additional information in our index, the index would be more specific and contain the following details for each indexed term:

- List of documents where the term occurs
- The term weight
- Position within each document
- List of categories the word belongs to

As mentioned above, the indexer has to calculate the context for each document and decide which category, or even categories, it belongs to. Also, the categories that documents should adhere to must either be pre-calculated, in order to make this procedure faster, or they could be calculated in real-time.

Calculating the categories real-time is computationally expensive. This is because if a document is added to the index and does not match any of the already calculated categories, a new category is created. When a new category is added, all the already indexed documents need to be checked to see if they match the new category. If documents have been moved to new categories, all categories need to be recalculated. There are ways of doing such *add ins* more effectively, but it is very complex and still time consuming. Therefore, the predefined categories seems like the best way to go. However, this also comes at a price.

In order to predefine the categories, the indexer must be given a *training-set* to use for its categorization (training sets will be discussed in chapter 4.3). This training-set must be descriptive enough to be able to define all the categories of the downloaded documents. Such datasets are hard to come by, and even harder to create so this is not an easy task.

Clustering algorithms

Several algorithms have been developed and used to categorize text documents, some of which are listed below:

- Latent Semantic Indexing (LSI)
- Independent Component Analysis (ICA)
- Probabilistic Latent Semantic Indexing (PLSI)
- Bisecting k-means
- Spherical k-means
- k-Nearest Neighbor (kNN)
- Bayes-Classifier
- Principal Component Analysis (PCA)
- Artificial Neural Network
- Non-negative Matric Factorization (NMF)

Also, we found another algorithm, *Frequent Term-based Clustering (FTC)* [5], promising but not many articles discuss its effectiveness or performance. The most common algorithms are probably LSI, PLSI, ICA and Bisecting k-means. For a good description of LSI, PLSI and ICA, the reader is referred to Sune Birch's work [9] and a brief description of Bisecting k-means and its performance can be found in [35].

Summary

In this section we discussed two approaches to indexing. The traditional inverted index has given good results in the past, but here we propose adding additional details to the index in order to get better results. Our general assumption is that *more precise index* $\Rightarrow$ *more precise results.*

In this work we will not try all of the above mentioned clustering algorithms, but merely wanted to introduce them as possible algorithms. We intend to use some of them and compare their results and performance. The details of the selected algorithms along with the reasoning behind the choices will be discussed in chapter 8.

1.2.3 Query Processing

When using search engines, users tend to submit few but specific terms as the query. Often the queries are merely one term which, obviously, can make it very difficult for the search engine to return relevant results. Especially if the word has many different meanings, e.g. *jaguar*. If a user only submits *jaguar* to the query, the search engine has no way of knowing if the user is looking for the car, the animal or even a former Formula One racing team. In a study from 2004, Beitzel *et. al.* [6] found that the average query length was 1.7 terms for popular

queries and 2.2 for all queries. However, an article from Yahoo! shows that the average query length has increased over the last few years and was about 3.3 terms in 2006[11].

Query Expansion

With most search engines, longer queries usually give more relevant search results. Query Expansion (QE) is commonly used to improve search engine results by either extending the original query with additional terms and/or re-weighing the original query terms.

The process of adding terms to queries can be automatic, manual or user-assisted[8]. Automatic QE is, for example when an algorithm calculates the weights of all the terms in the top results of the initial query and then adds the terms with the highest weight to the initial query, submits it again and returns the results of the second query to the user. In the case of manual QE, the user adds terms to the initial query. Finally, when referring to user-assisted QE, the system calculates and presents a list of possible expansion terms to the user, where the user is then asked to select which terms to use in the expanded query.

The most common technique within QE is using term-term similarities and/or term re-weighting, as mentioned above. In [32], Qiu and Frei present a probabilistic query expansion model using an automatically created similarity thesaurus to find term-term similarities. Furthermore, their research uses domain knowledge based on the query concept to find the appropriate terms to add to the original query. Their tests on three different datasets yield a result of $18 - 29\%$ improvement of the results. They found that the search results improve for each term added to the query. However, when the added terms are more than ≈ 100, the results for some datasets start deteriorating again.

Joshi and Aslandogan [19] also present a model using parallel concept-based query expansion. Their idea is to predict the different domains (concepts) of a users query, expand them separately, submit them to the search engine and return the categorized results to the user. Taking the term *jaguar* as an example, the parallel model would create three different query expansions, submit them to the search engine and return the top results for each query. In their tests they use WordNet and WordNet domains along with their own category corpus to predict the concepts of the queries. Their results show great improvement in relevancy of the results but they also find that the average time spent per query evaluation[12] decreases a great deal. Surprisingly, the query evaluation is shortest when parallel query expansion is only used with their category corpus. However, it is important to note that this research only deals with short queries (1 or 2 terms), and the model performs best with 1-term queries whereas the precision decreases as soon as the second term is added.

[11] http://blogs.zdnet.com/micro-markets/index.php?p=27

[12] Here query evaluation refers to the time it takes a user to identify the first 10 relevant results among the top 30 results shown

Finally, Vechtomova and Wang [40] examine the effect of term proximity on query expansion. They investigate the technique which expands the queries using terms occurring at a certain distance from the query terms. No clear conclusion is given in their study, but it indicates that expanding queries using term proximity could improve results. However, the distance is of great importance and the maximal term-distance should be chosen with care.

Query Analysis

Besides adding more terms to the original query, some other techniques might be useful in order to get better search results. For example:

- Use Part-Of-Speech taggers to analyze the query
- Use stemming to find term stems
- Try and predict the domain (context) of the query
- Remove Stop-words
- Handle punctuations and special characters

All of the above mentioned items have been discussed in previous sections regarding the document processing and will not be described further here. Basically, the query processor could utilize the same rules and steps as the document processor to make the query even more specific. In fact, it is necessary to apply the same rules as for the document processor since the query terms have to have the same form as the ones in the index. For example, if the query processor did not stem words, but the document processor did, then query term 'computer' would be indexed as 'compute' in the index, and thus not matched to the query term.

Summary

The above mentioned studies all indicate that query expansion is a great way to improve the results of short queries. However, the number of expansion terms and their weighting is something that has to be chosen with care in order to improve the results. Adding the wrong terms or giving them the wrong weight could end up yielding worse results than the original query. Furthermore, using concept-based query expansion has given good results but requires the use of corpora for domain prediction and to find terms with similar meaning.

Query analysis is also a necessary part of the query processor. The rules and methods used in the document processor should also be applied in the query processor in order to get more precise queries and ambiguous query- and index-terms.

1.2.4 Ranking relevant results

Whenever a user submits a query to a search engine, the engine returns a sorted list of results. This list is what the ranking algorithm in the search engine sees

as the most relevant results, the first result being the most relevant one etc. In today's search engines these ranking algorithms can differ very much in design and performance. Any user familiar with the use of search engines knows that the results to a given query are rarely the same for the most popular search engines. In fact, these lists can be very different. This is in part due to which pages the search engines have indexed, but also due to the different underlying ranking algorithms.

Generally, ranking a web page is not as easy as it seems. For example, if a ranking algorithm is based solely on word matching and a user submits a query using very common keywords such as *sports* or *movies*, the algorithm ranks all pages containing these keywords equally and thus, possibly, gives the user a lot of useless results in random order. A more sophisticated algorithm would try and determine the relevance of the pages containing the keywords and rank them accordingly.

As the number of web pages and other data on the Internet increases, returning relevant results to search queries becomes more difficult. Much effort has been put into the development of ranking algorithms in order to return more relevant results to the user.

> ...the number of documents in the indices has been increasing by many orders of magnitude, but the user's ability to look at documents has not. People are still only willing to look at the first few tens of results.
>
> \- Brin & Page, 1998 [10]

A user is less likely to continue using a search engine which returns few relevant results within the first tens of the results.

There are mainly three strategies in practice today when it comes to ranking search results. Namely, *Link Analysis*, *Vector Space Model* and *Relevance Feedback*. In the following subsection these strategies will be briefly introduced.

Link Analysis

Link analysis in general is used to try and find a link between two subjects. For example, link analysis is used in law enforcement when a criminal's bank records, telephone calls etc. are investigated to try and find evidence of his or her crime. Banks and insurance companies also use this kind of link analysis to try and detect fraud. Within the field of search engines, the link analysis strategy to ranking is based on how the pages on the Internet link to each other. The assumption is that pages regarding a specific subject will, with good probability, link to other pages on similar or the same subject. This seems like a very reasonable assumption and has worked quite well in practice, e.g. *Google* uses this approach.

An example of an ranking algorithm based on link analysis is *Google's* own **PageRank** algorithm [10]. The following quote taken from Google's Technology

web page[13] explains the essence of the PageRank algorithm:

> PageRank relies on the uniquely democratic nature of the web by using its vast link structure as an indicator of an individual page's value. In essence, Google interprets a link from page A to page B as a vote, by page A, for page B. But, Google looks at more than the sheer volume of votes, or links a page receives; it also analyzes the page that casts the vote. Votes cast by pages that are themselves "important" weigh more heavily and help to make other pages "important".

So basically, a page's rank in Google's search results is higher if many, preferably important, pages link to that page. The higher the PageRank, the more relevant the page is (according to Google). The mathematics behind the PageRank algorithm is quite impressive but will not be explained here. For a good explanation of the mathematics and other design issues of the algorithm see [23].

Another example of a ranking algorithm using link analysis is the **HITS** algorithm. The HITS algorithm utilizes the link structure of the Internet like the PageRank algorithm. The HITS algorithm is based on *hubs* and *authorities*, i.e. the algorithm calculates two values for each query, a hub value and an authority value. The authority value estimates the content of the page while the hub value estimates the value of its links to other pages. This algorithm will not be discussed further here, but more information on the algorithm can be found in [22], a paper written by the author of the algorithm, Jon Kleinberg.

Vector Space Model

Ranking using *Vector Space Model* (VSM) is very simple. Basically, each document in VSM is represented as a column in a *term-document* matrix. Each row in the term-document matrix represents a term. The value at index $td_{i,j}$ says how many times term i occurs in document j. For example:

$$\mathbf{TD} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

is a VSM representation of three documents each containing one occurrence of one term. The total number of terms for these documents is three. When ranking documents using VSM, the *Cosine Similarity Measure* is the easiest one to use see also chapter 4.1.1. Cosine Similarity Measure calculates the angle between two vectors in the above matrix. The closer the angle is to zero, the more similar the two documents are. Therefore, when a query is submitted to a search engine, a term vector is constructed for that query in a similar way as it is done for documents. The best matches for that query are the documents with the highest similarity to the query vector.

Unlike the link analysis ranking algorithms, ranking results using the Cosine Similarity is very easy and requires little computational effort since the backbone of the calculation is based on the dot-product of two vectors. This makes

[13] http://www.google.com/technology/

the Cosine Similarity measure an attractive possibility when it comes to ranking.

We will not explain the vector space model in more details in this section, but section 4.1 contains more details about the mathematics of the model and section 4.1.1 contains information on the Cosine Similarity Measure.

Relevance Feedback

Relevance feedback is used to try and improve the relevance of the results returned by a search engine, where the search engine's original ranking usually follows one of the above mentioned schemes. The idea is that the search engine can learn which results are relevant for a given query. However, the machine has to get some feedback data in order to improve its results. Feedback information is used to either adjust the weights in a given query and/or add terms to the query to make it more specific. There are mainly two types of relevance feedback used, namely, *Explicit feedback* and *Implicit feedback*. Explicit feedback is where a user tells the search engine explicitly what results are relevant and which are not. Implicit feedback is where the search engine "monitors" which links a user follows for the given query. The search engine then makes the assumption that the links followed are more relevant than the others and in that way adjusts that page's rank in subsequent, similar queries.

It is intuitively clear that explicit feedback can be very useful to improve ranking results, given that the users are honest and consistent in their evaluations. In [44], Patil *et. al.* introduce a tool that can be used to get explicit feedback from users. Implicit feedback is not so intuitive since one can visit 10 search results before finding any useful result and therefore the other nine results should not get a higher ranking for the query. However, in [20], Jung *et. al.* found that considering all "click-data" in a search session has the potential to increase recall and precision of the queries. Also, in [3] Rohini and Ambati found that using implicit relevance feedback based on search engine logs and user profiles gave improved precision results.

Summary

In this section we have discussed three terminologies when it comes to ranking search results. The basic Vector Space Model is a relatively simple and cost efficient way of measuring similarity between query and document. However, this model has its limitations. One limitation is that long documents are represented poorly, i.e. when comparing two documents of different lengths, their dot-product might not be very high due to high dimensionality. Also, documents concerning the same topic, but with different vocabularies would not give a high dot-product. This could though be rectified with the use of a synonymy-dictionary.

Link analysis has also proved to be a very good way of searching, e.g. Google's enormous success. However, the indexing takes a long time and the link structure can be manipulated in such a way that a web page gets ranked

higher in the search results (using SEO companies as mentioned before).

The intuitively best way of ranking results is using explicit relevance feedback. Provided that the users are unbiased toward the web pages and/or search engines and willing to participate and give their honest opinion, the results would be very good. However, this procedure will take a very long time, since many queries would have to be ranked and there is no way of ranking all possible queries. Automating relevance feedback would also be useful and would not take as much effort from the users. Although the result would not be as precise as using explicit relevance feedback.

1.2.5 Summary

In this section we have introduced and discussed the various elements of a search engine's anatomy. We showed that each of the elements play an important role in the final result. For instance, if the data preprocessing is done inadequately, the search engine can not be expected to yield good results. The same goes for the indexing, query processing and ranking. All the elements must be designed carefully.

Our discussion of the search engine anatomy has been very general and broad. We have discussed many issues that need to be addressed in order to construct a usable search engine and in the following section we will give a more detailed description of what the goal of this work is, and what we hope the final results will give us.

1.3 Problem description

ICA, LSI, NMF etc. are all algorithms that have been used to cluster text and make terms indexable. We propose to use Wikipedia as a learning source (expert pages) to generate our clusters and act as supervisor. Wikipedia has strong context, data is in a labeled hierarchy and has a strict format making it easy to parse. All qualities that makes clustering easier and hopefully more precise.

Our vision is to build a search engine that can read the content of a web page and understand its topic, hereby classifying pages accordingly. When classifying pages based solely on their content we expect the use of link farms, added meta information and other methods used to manipulate search results become obsolete or minimized severely. Only the actual topic of a web page will matter when matching a web page to a search query and placing it in the result set. The ranking will be based on how close a search query is to the actual topic of a page and not on the pages link structure. A topic-driven approach will hopefully minimize the number of results to a search query and at the same time make them more relevant. The idea is to deliver few, but relevant results.

The thesis can therefore be divided into two equally important goals, namely topic calculation and search engine implementation. Neither of which can stand alone as topic calculation is useless without the ability to retrieve documents

from it and vice versa.

We want to be able to calculate the topic in any given English text and categorize it with the help of Wikipedia articles. The main challenge of topic calculation is to find suitable algorithms for our data set, implement them and perhaps fine tune them in order to get the desired results.

Our goal with the search engine is to create a general search engine that is capable of searching within any given topic. However, to begin with we will have to focus on one topic, here being music. We limit our data to musical pages in order to reduce the data amount. Despite the fact that we use music articles, the search engine should not be implemented in a special way with this topic in mind. We want to implement an engine that can be trained using any kind of data and still sort through the topics in the data.

Basically, the end result should be a general search engine capable of searching within any topic by using clusters. The search engine should be able to retrieve results with such a precision that users find it useful.

1.3.1 Reading guide/Overview

This section gives a brief overview of the structure of the thesis part by part.

Part I - Data Store and Preprocessing

Data store and preprocessing describes the design of the data store used in *Zeeker Search Engine*, i.e. what data is available and how have we chosen to limit the data to a manageable size while still getting good results. Furthermore, data processing and the various data processing tests are also discussed.

Part II - Clustering

Clustering introduces the basics behind clustering and the general principles. This part also introduces a few clustering algorithms such as Spherical k-means, Non-Negative Matrix Factorization (NMF) and Frequent Term-based Clustering (FTC). The algorithms differences are discussed as well as the tests on the implemented clustering as well.

Part III - Retrieval

The Retrieval part explains how the *Zeeker Search Engine* processes queries and how documents are retrieved, ranked and presented to the user. Various test scenarios along with the tests performed on the *Zeeker Search Engine* retrieval and the results of these are also discussed.

Part IV - Implementation

Implementation of the search engine is described along with the data flow i.e. from the time data is downloaded from the Internet to how it is presented on the web page.

Part V - Conclusion

The Conclusion part contains a discussion of the future of the *Zeeker Search Engine*, i.e. what *Zeeker Search Engine* is capable of and how we want to improve it. Furthermore, all the efforts put into this thesis are discussed and summed up in the conclusion.

Appendix

The Appendix includes a list of used stop words, a list of POS tags, description of test sets and a User Guide.

Part I

Data Store and Preprocessing

Chapter 2

Data store

Data store refers to what data is available for preprocessing, indexing, testing etc. In this chapter it will be discussed and explained what data from the internet has been made available in the data store. Data is downloaded and only stored if it belongs to the chosen data intersection, which will be described later in this chapter. Wikipedia has been mentioned before and since data from Wikipedia is used as the core data, it is necessary to take a closer look at how it is structured and what data is publicly available.

The downloading of data and web resources is handled by *Zeeker.Spider*. The webspider was created as a pre-project and is therefore not discussed in this thesis. *Zeeker.Spider* can best be described as a standard webspider, distributed and highly configurable, with an underlying database holding all downloaded data. Suffice to say that the *Zeeker.Spider* meets all the web-crawling needs of this thesis.

2.1 Wikipedia

Wikipedia is a well known on-line free encyclopedia that anyone can edit. It has had tremendous support from day one and is still growing. People all over the world co-author articles on whatever topic they find interesting and Wikipedia categorizes these articles and makes them publicly available on the Internet.

Wikipedia (wiki) has been chosen as a source of *expert pages* in the search engine because of it's enormous amount of information. Wikipedia includes both explicit information, such as articles, and implicit information, such as edit history, discussion pages etc. In fact, Wikipedia contains so much information that there is no way of utilizing it all[1]. Therefore a choice has to be made on which information is most suitable for the purpose of this thesis and scale it to a size that is possible to utilize within the given time frame.

[1] using the current hardware setup

Wikipedia structure

One of the powerful features of Wikipedia is the strong contexts of its articles (one article - one topic). All articles are labeled and categorized by many editors giving a very good categorization - which even improves over time as more people contribute.

Figure 2.1, page 27 illustrates a part of the Wikipedia structure. The category *Musical groups* is chosen and the figure shows the outline of the category tree as well as the relationship between categories and articles. Categories can have sub categories and articles associated with them. Articles belong to one or more categories and have several interesting article attributes as shown in the figure. Wikipedia does not enforce a strict parent-child relationship in its category structure and cycles and disconnected categories are therefore possible (yet rare)[46]. The categorization of Wikipedia is human made and as a result is very precise - or at least as precise as can be hoped for. The category precision is what can hopefully be exploited to get better results when the core is clustered. Using Wikipedia as a training set to create clusters is called *supervised machine learning* - which is exactly the intention of this thesis.
A more detailed discussion of the concepts *clustering*, *supervised-* and *unsupervised machine learning* and *training sets* is given in Part II. For now, suffice to say that the strong categorization of the Wikipedia articles is a unique opportunity to use Wikipedia as a data source to generate good clusters.

Other features

Besides the great categorization and the strong article contexts, Wikipedia has a lot of other useful attributes as shown in figure 2.1. All articles have a quality measure, such as *stub*, *normal* and *featured* that indicate how Wikipedia rates the context of the article.

Every article also has associating discussion pages and logs of what has changed, who has changed it and when. These article attributes give a lot of information about the article quality as well. Long discussion pages, many major and minor edits, the profile of the user doing the editing, the amount of links to other Wikipedia articles, the amount of external links, how many images are included and which images etc. can all be used as quality indicators for a specific article. These measures can even be used as a filter to remove the bad articles when trying to generate accurate clusters.

Yet another advantage of using the Wikipedia articles is that when clustering the articles, the article information might be incorrect but the vocabulary will probably still be correct. Meaning the words used to describe the topic inaccurately are presumably the same as the words normally used to describe that topic in an accurate way. Hence, the clusters will still give a very good description of the topic since the statistical properties of the individual words are used and not their exact meaning.

Furthermore, Wikipedia includes a download service which provides data

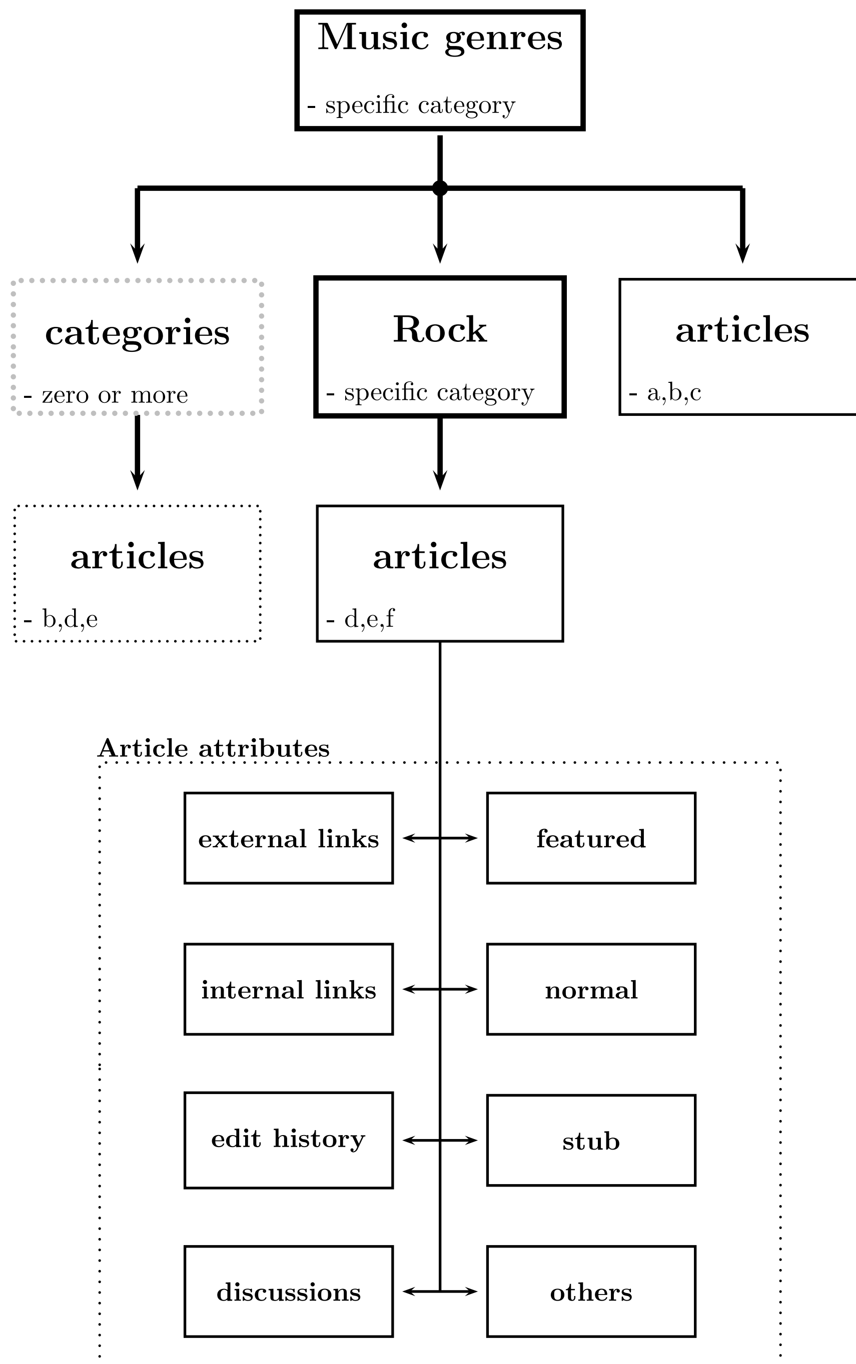

Figure 2.1: Wikipedia category and article hierarchy overview

dumps in text format. These data dumps include many important tables that the Wikipedia database consists of, such as page information, external links, image links, page to page links, page abstracts, articles, article titles, redirecting articles and much more.

Another interesting source for generating training sets and clusters was considered, namely the *Open Directory Project* (ODP). The problem with ODP is that one page does not necessarily mean one topic, and it is much easier to build a category from a page when it is known that it only has one topic. Therefore, given all its features and article attributes, Wikipedia seems like the best choice.

2.2 Data store structure

As mentioned above, Wikipedia has many good qualities and is therefore used as the core data in the data store. The entire Wikipedia article base has been downloaded[2] such that any Wikipedia article (in English) is available in the data store. However, Wikipedia has around 1.8 million articles[3] and such data volume is simply too big to work with in this project. Therefore, the size of the core must be reduced such that only the articles categorized under the category *Musical groups* are used. This is done by running through the hierarchy, see figure 2.1, and collecting all categories and articles from all hierarchy levels below the category *Musical groups*.

Ideally, all terms in all documents on the Internet would be indexed for future searches, but again the vast amounts of data makes that nearly impossible. Instead an intersection of data is created, as shown in figure 2.2, by only allowing web resources with certain Mime-types (Html) in English dealing with *Musical groups*.

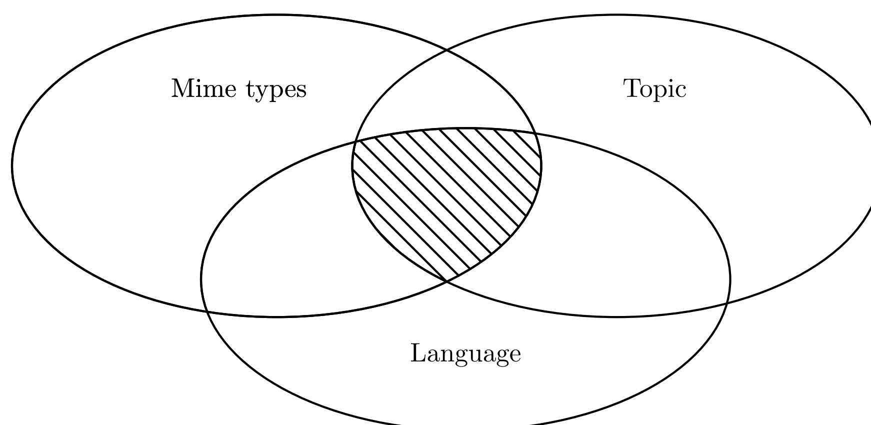

Figure 2.2: Data intersection

[2]Wikipedia can be downloaded from http://download.wikipedia.org/
[3]June 2007

Expanding the data store

The data store is constructed in a bottom-up fashion starting with the data core and then expanding it with additional data as can be seen in figure 2.3. Each layer of data adds more information to the data store and requires additional space. The trade-off between data amount and size is a fine line which needs to be determined. In the data store, each layer is given a *depth* which indicates the layer's distance from the core layer. The greater the distance from the core, the more noise (rubbish data) the layer will have. Therefore, the distance must be chosen wisely in order to get as much relevant data with minimal noise.

The chosen intersection gives us a core of 49.748 Wikipedia articles in 3.605 categories in the data store. The core will be used as a training set when clustering. The core is defined to have depth zero.

Most of the articles in the core, contain references to external links i.e. external to Wikipedia. These web resources have been downloaded and added to the data store. The external links are closely related to the information in the Wikipedia articles, thereby minimizing the noise in the data compared to the noise in the data had it been downloaded from other sites on the Internet. The external links provide 172.497 web resources of different kinds (html, pdf, word etc.). The external links are defined to have depth one.

Even though the core and the external links provide approximately 200.000 web resources, it would be best to download at depth two. The reason being that depth two will not only provide more data but also data with a lot of noise which can be very useful when testing the retrieval performance of the search engine. Depth two (Distant links), has well over 10 million web resources but the downloading has been limited to approx. 1 mill. web resources at depth two.

Figure 2.3 shows the bottom-up structure of the data store and how the data store is expanded as the depth of the downloading increases. The final result is a data store of approx. 1.2 mill. web resources.

2.3 Test sets

The Wikipedia article database is used to generate some general test sets. These test sets are created from general topics where the vocabulary is not restricted to any topic. General topics are chosen since the intent is to perform tests on general data to make sure the algorithms and word filtering is not fitted toward a less diverse vocabulary.

The test sets are constructed in an incremental fashion as can be seen in figure 2.4. This means that test set 2 is created from test set 1 plus an additional 10.000 randomly chosen articles, test set 3 is created from test set 2 plus an additional 10.000 randomly chosen articles etc. The test sets have sizes ranging from 10.000 to 100.000 articles. The test sets are created in an incremental fashion primarily because the intention is to test the scalability of the system as more articles are added to the index. The data sets could also be created by

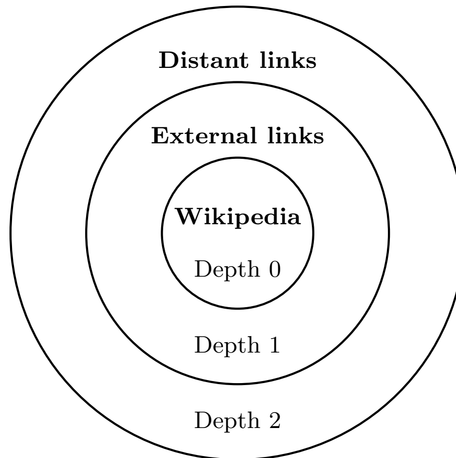

Figure 2.3: Expansion of the data store

selecting random articles for each set. This approach could introduce more noise in the data which might give an unrealistic image of the system's scalability. This effect is minimized when the test sets are built on top of each other.

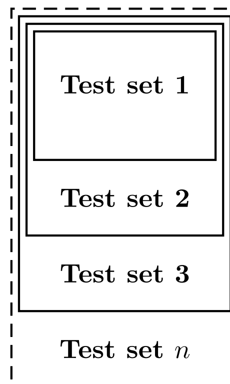

Figure 2.4: Test sets structure

Choosing the sizes for the test sets is a trade-off between indexing time and index size. Enough articles had to be chosen to properly test the system's scalability, but also the time factor had to be taken into account as it should be possible to create the test indexes within a period of a few hours. Furthermore, appropriate sets had to be chosen to see how they pressure the hardware as the set sizes grow up to and above 1 GB. Indexing such sets could pressure the memory and CPU of the indexing computer and it is interesting to see how this changes with growing sets.

2.4 Summary

Wikipedia has many great qualities which are useful when trying to cluster data and assessing the quality of the data. Wikipedia contains vast amounts of useful information and strong article contexts. Other resources on the net, such as ODP, do not have these qualities making Wikipedia the best choice for this work.

The data store core has been built up around the Wikipedia category *Musical groups* and extended further by adding the external links from the articles in the selected category and the distant links they refer to. This gives a layered data store with web resources of depth up to 2, viewing articles in *Musical groups* as depth 0. The core also delivers a training set created from the hierarchy of the labeled Wikipedia articles.

The many Wikipedia articles are used to generate several test sets of various sizes which contain a general vocabulary and are large enough to reveal any problems the filtering might encounter when working on large scale data. These sets are also used to estimate how the system will scale as more articles are added.

All this results in a data store built from the Wikipedia category *Musical groups* and test sets created from the Wikipedia article base. Table 2.1 shows the basic statistics for the data store.

Depth	Data	Web resources
0	The core	49.748
1	External links	172.497
2	Distant links	approx. 1 mill.
	Data store	approx. 1.2 mill.

Table 2.1: Data store statistics

Chapter 3

Data processing

In this chapter we will discuss how the data store and test sets are processed, i.e. what choices have been made and how data is prepared for indexing and clustering. Some tests of the quality of our various choices will also be presented.

3.1 Processing data

Data collected from the Internet is in various formats requiring normalization. As described in section 1.2.1, a predefined list of stop-words will be used along with the Porter stemmer to reduce the index size. The list of stop-words can be found in tables D.1 and D.2 in appendix D.

3.1.1 Index reduction

In order to reduce the index sizes further, a POS[1] tagger is used. After the POS tagger has categorized the terms it is easy to decide which word categories (POS tags) can be ignored thus reducing the index size. In [30] it is shown that pruning the vocabulary in this way does not necessarily affect the retrieval precision. However, the distribution of the POS tags still need to be tested to get an indication of how the choices made will affect retrieval accuracy and index size. All the POS tags can be found tables C.2 and C.3 in appendix C. Table C.1 in appendix C shows the distribution of the POS tags. Since the original texts consist of approx. 30% nouns and this being the most important group, the index size can never fall below 30% of its original size without the use of word association techniques.

The POS tag distribution also reveals that punctuation groups, as introduced in section 1.2.1 are not important. Punctuation groups, along with other

[1] Part-of-speech

similar groups, account for less than 2%[2] of the index.

Based on tests and basic natural language knowledge, the tags chosen to be included in the index are presented in table 3.1. All others are ignored.

Pos included	Description
CD	number, cardinal (four)
FW	foreign word (ante, de)
JJ	adjective, general (near)
JJR	adjective, comparative (nearer)
JJS	adjective, superlative (nearest)
NN	noun, common singular (action)
NNS	noun, common plural (actions)
NP	noun, proper singular (Thailand)
NPS	noun, proper plural (Americas, Atwells)
OD	number, ordinal (fourth)
RB	adverb, general (chronically, deep)
RBR	adverb, comparative (easier, sooner)
RBS	adverb, superlative (easiest, soonest)
RP	adverbial particle (back, up)
SYM	symbol or formula (US$500, R300)
???	unclassified

Table 3.1: Included tags in index

3.1.2 Index size

Due to hardware limitations, the index size is one of the most important considerations in the data preprocessing. [48] states that the size of an index consisting of texts[3] will be around 40% of the original size and the index size for Html-pages will be around 20% of the original size. This difference is due to the large amount of unindexed markup data (Html code). However, these sizes can not be taken as absolute values of how index sizes grow, but merely as an indication. Several index compression techniques are also shown and these might come in handy at a later stage, but in this project the data is restricted to a size that can be handled without compression.

3.1.3 Tests

The size of the index has been tested using four different approaches (test models):

- A full index (all terms indexed).
- A stopped and stemmed index.
- A POS tagged index.
- A POS tagged, stopped and stemmed index.

[2]punctuation groups alone only account for 0.7%
[3]Measuring the NewsWire data set

The ten test sets generated from the Wikipedia article database (see chapter 2.3 for more information) are used for creating indexes for each approach. This means that ten indexes are created for each test model giving 40 indexes in all, ranging from 10.000 to 100.000 articles. These 40 indexes have been used to test the functionality of the code, the building of the index and numerous tests to see how this vast amount of data behaves. All test results can be seen in appendix B.1. The tables (appendix B.1) show how many terms are in the indexes and the index sizes in question. The total number of unique terms is the most important statistic as it represents the actual number of searchable terms[4]. Unique terms also determine the dimensionality of the term-vectors and obviously the larger the dimensionality, the more complex the clustering calculations and other calculations become.
Figure 3.1 on page 36 shows, how the number of unique terms decreases when stop-words, stemming and POS tagging is applied to the 40 tests conducted. The slow rate of decrease is due to the conservative strategy used when removing terms. An aggressive stemmer is not used and POS tags are not removed too aggressively. This conservative strategy is enforced because it will be easier to decrease the index size later, by POS tagging or other means, when the complete search is implemented, as it is easier to see the effects at that time. If too many terms are removed from the beginning, it would be impossible to determine whether a bad search result is due to a bad retrieval technique or simply because too many terms were missing in the index. Conversely, it is easy to enforce a more aggressive strategy if the search results return too many useless results.

With the current strategy of moderate POS filtering, stopping and stemming it is possible to reduce the index from 1.352.497 unique terms when *full index* is used, to 1.167.676 unique terms. That is a decrease of 14% in unique terms and a deacrease of 17% in size on disk. Compared to the staggering results of 92% in [30] it is not impressive. However, as already mentioned, only a limited amount of terms are removed in order to have more control over data and retrieval. It should also be noted that the vocabulary used in this work is a completely different vocabulary than in [30] where the data was email correspondance - whereas in this case the vocabulary is more diverse.

The choices made in the preprocessing stage have given a flow diagram as shown in Figure 3.2 on page 38. First a parser tokenizes the documents and removes Html and/or Wikipedia Xml tags. Then the POS tagging is applied and filtered according to the rules of included tags in table 3.1. After POS filtering, the remaining stop-words are removed (POS tags should remove most of those). Stemming is then applied before the tokens are finally indexed along with the original document. The original document is stored in full in the index for future reference and the ability to retrieve text snippets to show along with the link results as known from other search engines on the Internet.

[4] If no other parsing or removal of terms is done

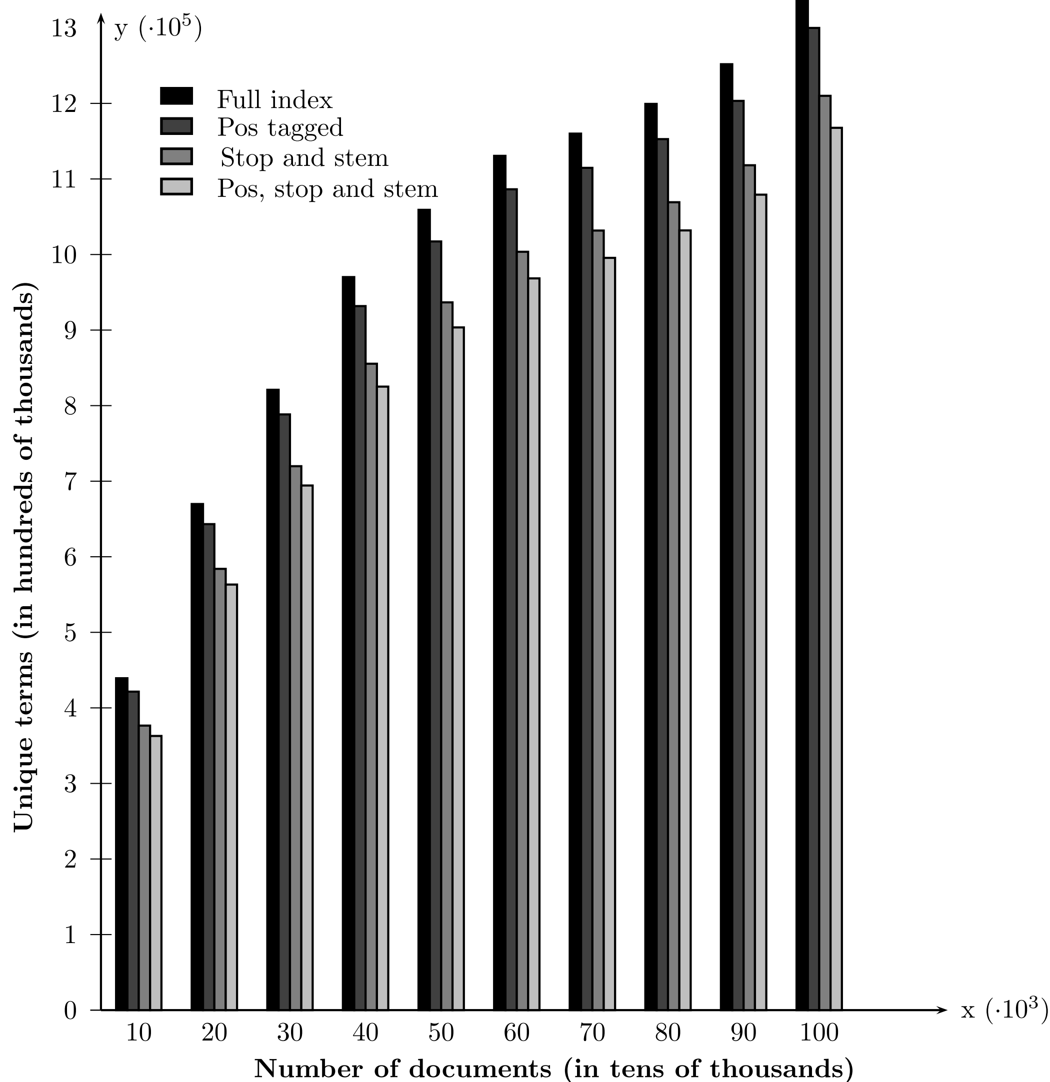

Figure 3.1: Number of unique terms in documents

3.2 Summary

To sum up the choices made, the data representation will be mentioned first. As stated in [48], and discussed in a previous chapter, an inverted file index seems to be the ideal choice for the data used. However, the *full-inverted index* (as shown in Table 1.3 on page 11) was chosen. The full inverted index seems better as it includes the term proximity information needed to facilitate a full text search.

As mentioned above, the list of stopwords can be found in appendix and the Porter stemmer is used as it seems to be well respected and comparable with more complex and sophisticated stemmers. Here we opted for the simple, but effective stemmer.

POS-tag filtering is also included, as it reduces the index size quite impressively, and has the possibility of reducing it even further without much additional work if it later is found necessary. The conservative strategy is found most appropriate to begin with. Implementing POS-tagging also gives the ability to later POS-tag texts in the index, thus giving users the ability to search for a particular word in a certain POS category. E.g. the ability to search for the noun *train*, thus excluding any documents containing the verb *train*. This feature is not something we expect to be a part of this project, but merely a possibility in a later version.

Finally we mention that the tests conducted on the index size, unique terms etc. are all performed on general articles from Wikipedia and the number of terms, number of unique terms and index sizes may be very different for a more specific category, like the one chosen, namely the *Musical groups*. The vocabulary is presumably smaller for these articles giving a smaller index. However, it was important to conduct the tests on general articles so that the choices made were based on the general vocabulary and not on the specific vocabulary used when describing music.

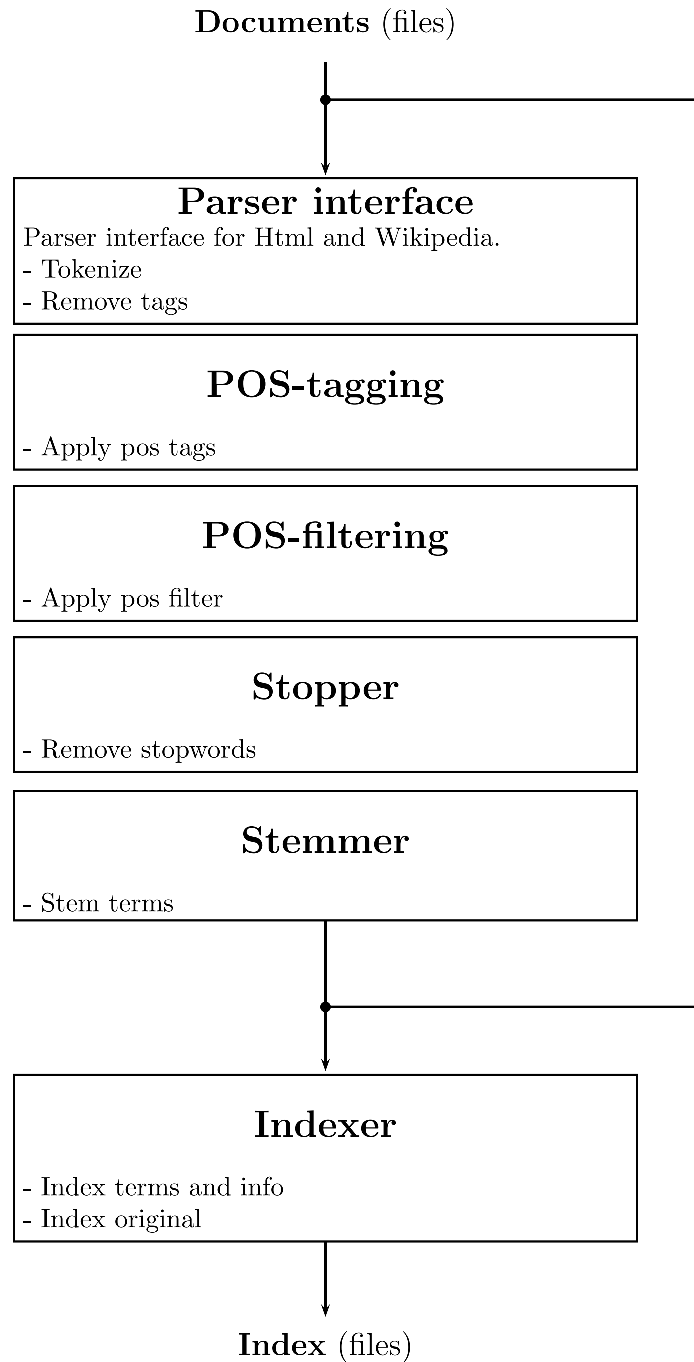

Figure 3.2: Document processing

Part II

Clustering

Chapter 4

Clustering

Document clustering is a very important part of automatic topic detection in machine learning and pattern recognition. The idea of clustering is to partition a set of objects into clusters such that any object within a cluster has more in common with the other objects in the same cluster, than any other object outside the cluster. In order to cluster objects, a proper representation of the objects is needed as well as a way to measure similarity between them.

In this chapter we will describe the representation of objects using the *Vector Space Model* and the object similarity measure used, namely Cosine Similarity Measure. We also discuss the basics of clustering and mention different types of clustering algorithms.

4.1 Vector Space Model

As mentioned in chapter 1.2.4, the *Vector Space Model* (VSM) refers to how documents are represented and ordered. In VSM, a *term-document* matrix, **TD**, is constructed in such a way that each row represents a term[1] and each column represents a document. For example:

$$\mathbf{TD} = \begin{bmatrix} t_{1,1} & \cdots & t_{1,m} \\ \vdots & \ddots & \vdots \\ t_{n,1} & \cdots & t_{n,m} \end{bmatrix}$$

where the column vector d_i is called the term-vector for document i. The term-vectors are often in very high dimensions as each term represents a single dimension, i.e. the richer the vocabulary, the higher the term-vector dimensions.

[1] A term can consist of one or more words, numbers, dates etc.

The values in the matrix can be binary (0 or 1), discrete $(0, 1, \ldots, n)$ or continuous $(0.45, 1.56, \ldots$ etc.). When using the binary values, a zero denotes a word not occurring in the document and the number one denotes an occurring word. Discrete values are typically used to represent word frequencies, meaning that if the value $d_{5,i} = 5$ then the term t_5 occurs five times in document i and so on. Continuous values are used when the terms in the term vector are weighted differently, for instance if a term is found to be more important within a certain context e.g. a title of a web page etc.

4.1.1 Cosine Similarity Measure

Measuring similarity between two term-vectors is frequently done using the *Cosine Similarity Measure*. The cosine similarity measure between vector a and b is defined as the angle between the two vectors:

$$\cos\theta = \frac{\mathbf{a} \bullet \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} \tag{4.1}$$

where $\mathbf{a} \bullet \mathbf{b}$ is the dot product between the two vectors and $\|\mathbf{a}\|$ refers to the length of the vector. The similarity is found by looking at the angle between the two vectors. The smaller the angle, the greater the similarity between the vectors. Looking at figure 4.1, it is easy to see that the smaller the angle θ, the more similar the vectors $\|\mathbf{a}\|$ and $\|\mathbf{b}\|$ are. When $\theta = 0$ the vectors are identical.

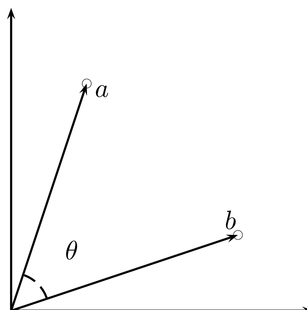

Figure 4.1: Cosine simularity

In [33], Salton and McGill discuss, in more detail, the Vector Space Model and the calculation and usage of the Cosine Similarity Measure along with other similarity measures.

Document similarity example

With documents represented by the VSM, measuring similarity between the documents using cosine similarity measure, is fairly simple, as shown in equation 4.1. However, since the models are simple, they also have their weaknesses. As an example, consider the following term-document matrix:

$$\mathbf{T} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 2 & 4 \\ 1 & 3 & 4 \end{bmatrix}$$

where $t_i \in \{jaguar, car, british\}$. Here, document d_1 only consists of one instance of term t_1 and one instance of term t_3. Now lets say a query is submitted to a search engine using only *jaguar* as a query term. In order to find the best match for that query, the cosine similarity needs to be calculated between the query vector, $q_v = [1\ 0\ 0]$, and each document in $\mathbf{T}$. Using equation 4.1 gives:

$$\begin{aligned}
\cos(\theta)_{d_1} &= \frac{\mathbf{q_v} \bullet \mathbf{d_1}}{\|\mathbf{q_v}\|\|\mathbf{d_1}\|} = \frac{1*1}{\sqrt{1^2} * \sqrt{2^2}} \approx 0.707 \\
\cos(\theta)_{d_2} &= \frac{\mathbf{q_v} \bullet \mathbf{d_2}}{\|\mathbf{q_v}\|\|\mathbf{d_2}\|} = \frac{1*2}{\sqrt{1^2} * \sqrt{2^2+2^2+3^2}} = \frac{2}{\sqrt{17}} \approx 0.485 \\
\cos(\theta)_{d_3} &= \frac{\mathbf{q_v} \bullet \mathbf{d_3}}{\|\mathbf{q_v}\|\|\mathbf{d_3}\|} = 0
\end{aligned}$$

The results above show that the cosine similarity measure clearly favors shorter documents with fewer unique terms or fewer occurrences of the query terms. This is not a good property since one can not in general assume that short documents with fewer terms are more descriptive and precise compared to longer documents. It is also noted that if the searched term does not occur in a document it has similarity 0. To counteract the effect/weakness of longer documents in the cosine similarity measure, term weighting is a good start. Using a term-weighting scheme such as *Tf x Idf* would certainly help to minimize this effect.

4.1.2 Term weighting (TF x IDF)

As mentioned above, weighting terms when using VSM is very important in order to represent documents properly. Longer documents are poorly represented with regards to cosine similarity in VSM and therefore a term weighting scheme is useful to rectify this.

In chapter 1.2.1, the rationale behind the *Tf x Idf* scheme was briefly discussed. In short the rationale is as follows; Rare terms are not less important than frequent ones and vice versa. Likewise, longer documents are not more important than shorter ones and vice versa. Thus, this term weighting scheme takes both word frequencies as well as document frequencies into account when assigning weight to the terms.

The mathematical specifications of *Tf x Idf* are briefly explained in the following (primarily adopted from [14]). The equation for the weight of a term is shown as:

$$\text{tfidf}(t_k, d_j) = \text{tf}(t_k, d_j) \log \frac{|D|}{|\{t_k \in d\}|} \tag{4.2}$$

where $|D|$ is the total number of documents and $|\{t_k \in d\}|$ is the number of documents where the term t_k appears and

$$\text{tf}(t_k, d_j) = \begin{cases} 1 + \log \text{freq}(t_k, d_j) & \text{if } \text{freq}(t_k, d_j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\text{freq}(t_k, d_j)$ denotes the number of times t_k occurs in d_j.

$\log \frac{|D|}{|\{t_k \in d\}|}$ is the inverse document frequency which is a measure of the general importance of a term in the document collection.

In order to satisfy the assumption that longer documents are not more important than short ones, equation 4.2 needs to be normalized. This is often achieved using cosine normalization[14]:

$$\mathrm{w}_{kj} = \frac{\mathrm{tfidf}(t_k, d_j)}{\sqrt{\sum_{s=1}^{|T|} \mathrm{tfidf}(t_s, d_j)^2}} \tag{4.3}$$

where $|T|$ is the number of terms.

Given the above, a term will be assigned a high weight if it occurs many times in a single document but rarely in the entire document collection.

Term weighting example

In order to demonstrate the effect of the *Tf x Idf* term weighting scheme, the matrix $\mathbf{T}$ from before is used. Applying equation 4.3 to each term in the matrix gives:

$$\mathbf{T_w} = \begin{bmatrix} 1 & 0.707 & 0 \\ 0 & 0.707 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Using the same query as above, $q_v = [1\ 0\ 0]$, the cosine similarity measure is found using 4.1 for each document:

$$\begin{aligned} \cos(\theta)_{d_1} &= \frac{\mathbf{q_v} \bullet \mathbf{d_1}}{\|\mathbf{q_v}\|\|\mathbf{d_1}\|} = \frac{1 * 1}{\sqrt{1^2} * \sqrt{1^2}} = 1 \\ \cos(\theta)_{d_2} &= \frac{\mathbf{q_v} \bullet \mathbf{d_2}}{\|\mathbf{q_v}\|\|\mathbf{d_2}\|} = \frac{1 * 0.707}{\sqrt{1^2} * \sqrt{0.707^2 + 0.707^2}} \approx 0.707 \\ \cos(\theta)_{d_3} &= \frac{\mathbf{q_v} \bullet \mathbf{d_3}}{\|\mathbf{q_v}\|\|\mathbf{d_3}\|} = 0 \end{aligned}$$

With the new weighting applied, the similarity values have changed. Longer documents have become more important. Although this is a simple constructed example, it demonstrates how the weighting improves the representation of longer documents. The example also illustrates that terms occurring in all documents (term t_3), regardless of frequency, are given weight zero thus phasing out frequent terms.

4.1.3 Summary

Representing documents using VSM and measuring similarity between documents using the cosine similarity measure is quite simple and computationally efficient. Unfortunately, the VSM model and the similarity measure favor shorter documents with fewer occurrences of the terms when compared to short queries.

Since short documents can not be assumed to be more relevant than the longer ones, the longer documents must be represented better using VSM. This is where a term weighting scheme such as *Tf x Idf* can be useful. The rationale behind the scheme is that longer documents are not necessarily more important

than the shorter documents and vice versa.

Therefore, dealing with documents of varying lengths using VSM and cosine similarity, applying the *Tf x Idf* term weighting scheme looks like a natural choice.

4.2 Clustering

As mentioned at the beginning of this chapter, clustering is the task of finding natural groups in data. In general, there is not only one solution to the problem of clustering. Therefore, the clustering algorithms seek to maximize some mathematical measures for the quality of the found solutions. It has been shown that the general problem of partitioning d-dimensional data into k sets is NP-complete[2] [1] which is why clustering algorithms can not find precise solutions, but only approximations to the problem.

Several clustering algorithms were briefly introduced in chapter 1.2.2. The introduced algorithms all find approximate solutions to certain minimization or maximization problems. For example, the k-means algorithm tries to minimize the Euclidean distance between the documents in a cluster to a given cluster center, Spherical k-means (as will be described in chapter 5) tries to maximize an angle between documents and cluster centers etc.

While documents are usually represented as vectors in a multi dimensional space using VSM, figure 4.2 illustrates a set of documents represented as dots in two dimensions for simplicity.

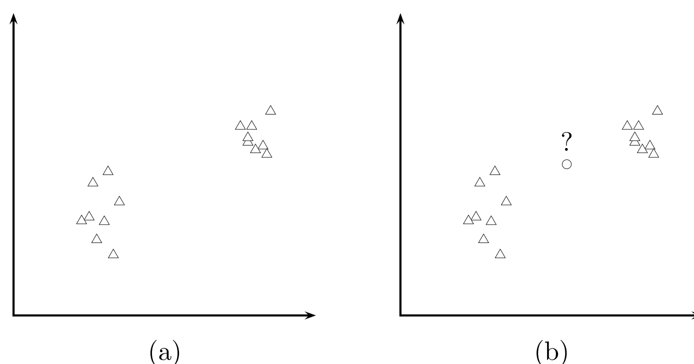

Figure 4.2: Clustering example

The clustering algorithms try to calculate which documents belong to which clusters. This can seem a trivial task when looking at the documents in figure 4.2(a), but as complexity grows with thousands of dimensions and millions of documents this is not an easy task. Even in two dimensions it can be difficult as can be seen in figure 4.2(b). What cluster does the document marked with

[2]See glossary for definition

the circle belong to? Even for humans it can be difficult to determine the right cluster or category a document belongs to and as there may be millions of documents it is not feasible for humans to categorize them all. This is where *supervised-* and *unsupervised machine learning* methods meet. These terms will be described in the following section.

Some of the problems with clustering are related to how documents are represented in the vector space, meaning that the clustering algorithm can be very good, but if the documents are not well represented in the vector space, the algorithm can only do so much. The creation of the vector space[3] is analyzed in the introduction of the thesis in chapter 1.2.1. There, the problems regarding which words are included in the index and which words are not included are described. And even the more basic question: What is a word?

Our considerations are based on the English language as pointed out in chapter 3. Other languages can have different problems, yet many, or all, of the considerations associated with English may also apply to other languages as well. We will not delve deeper into these considerations, but merely mention that different languages can pose different problems and these problems are also a very important part of creating usable clusters.

4.2.1 Overlapping vs. non-overlapping

Another aspect of clustering is the question of overlapping or non-overlapping clusters. Overlapping refers to when documents may overlap between clusters, that is, belong to more than one cluster. This is intuitively the best clustering method as many (or all) real-life documents are part of several categories. This of course depends on how the clusters are created. If there are only two clusters, *selling* or *not selling*, a document should only be classified into one cluster - it is either selling something or it is not. Such a simplistic view of clusters is not feasible when using the algorithms on Internet resources. For the example shown in figure 4.2 (b) it would be intuitively best if the circle belonged to both clusters with some probability.

4.2.2 Types of algorithms

When talking about clustering algorithms, there are three primary strategies used to find the clusters. Namely,

- Hierarchical clustering
- Partitional clustering
- Spectral clustering

The *Hierarchical clustering* approach builds a hierarchy (a tree) where the nodes in the tree represent the clusters. This approach can be used in either a bottom-up or top-down fashion creating a new level of clusters at each iteration. The *Bisecting k-means* algorithm can be modified to create a hierarchical clustering

[3]Meaning what is indexed and what is left out (stopwords, POS-filtering etc.)

by storing how the algorithm divides the dataset[4].

Using *Partitional clustering* means to partition the dataset into a number of parts (clusters). The number of parts is defined beforehand and the algorithms refine these parts at each iteration to improve these parts. The algorithms stop when they have converged or a number of iterations are done. An example of a partitional clustering algorithm is the original *k-means* algorithms. The unmodified version of Bisecting k-means can also be seen as a partitional clustering algorithm.

The last approach is *Spectral clustering*. The spectral clustering algorithms usually use dimensionality reduction techniques such as *Singular value decomposition* or *Non-negative matrix factorization* to reduce the dimensionality of the datasets so that they are easier to work with. Clustering of the dataset is then performed on the dimension reduced set. Example of spectral algorithms are *Latent semantic indexing* and *Probabilistic latent semantic indexing*.

4.3 Machine learning

Supervised machine learning is the task of classifying a collection of documents into a set of categories with the use of a training set. Such a training set is a large number of labeled training documents that can give the algorithm a sense of what kind of documents belong to a certain category. This means that the clusters are *learned* from the training set and the *real* documents are then added to the cluster(s) they are most similar to. Such training can give near *human-like* classified clusters if the training set is large enough. One problem is that it is very hard and time consuming to create such labeled training documents by manually categorizing them and, as mentioned, one piece of text can easily be categorized into different categories by different people.

Unsupervised machine learning is categorizing the clusters using statistical methods and/or clustering algorithms without any prior knowledge. This means that documents are added to different clusters based on a calculation on which cluster they most likely belong to. It is very problematic to create an unsupervised algorithm that works well on all varieties of data.

Figure 4.3 illustrates the difference between supervised and unsupervised learning - calculation of clusters is done by k-means (roughly).

The top left and top right plots demonstrate how supervised learning recognizes which documents belong to a cluster based on the training set and the cluster is then calculated from these documents. In the top left plot the dashed circles indicate the labeling of the clusters, i.e. what documents belong to each of the clusters. When such knowledge is present, it is fairly easy to compute the clusters and their centers as seen in the top right plot.

The bottom left and right plots demonstrate how the clusters are calculated from randomly chosen initial cluster centers[5] giving a completely different result

[4] Bisecting k-means divides the largest cluster into two parts at each iteration until a wanted number of clusters is reached

[5] k-means algorithm needs starting points for its cluster centers which are usually supplied as random points

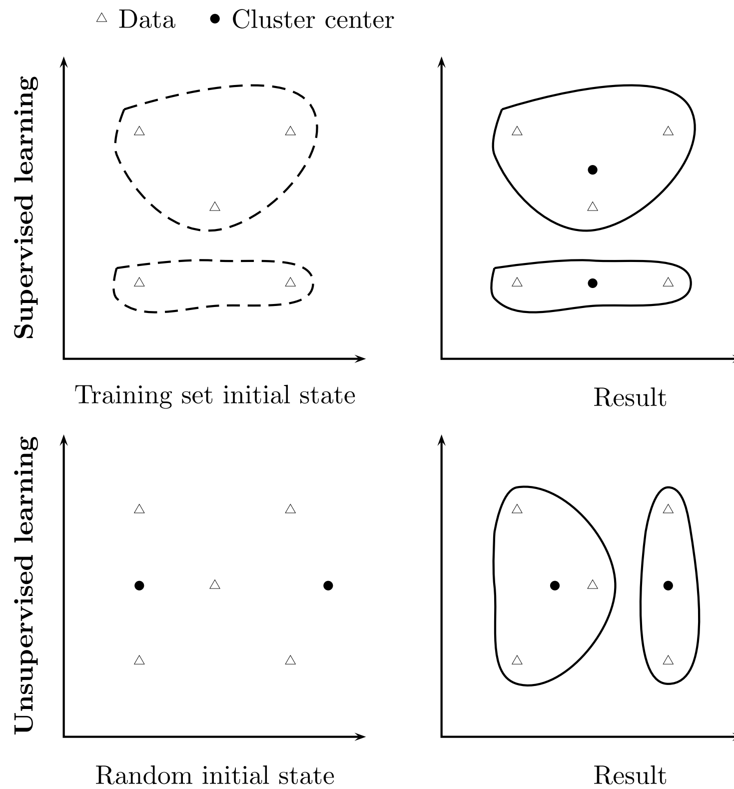

Figure 4.3: Supervised vs. unsupervised learning

than the supervised learning. A more detailed description of how clusters are calculated is given in the following chapters.

Figure 4.3 is of course a very simplistic look at clustering and meant only as an example. With different initial cluster centers, the unsupervised learning example could generate the same clusters as the supervised learning example. Figure 4.3 is meant to illustrate one of the possible short comings in unsupervised learning. However, it is not feasible to create training sets that are large enough to enable clustering of all Internet resources using supervised learning which is why unsupervised learning is interesting.

Most of the methods and techniques in supervised learning can easily be ported to unsupervised text categorizations while supervised learning allows more accurate performance measurements[6] and easy comparison with other methods. As mentioned in chapter 2, the intention is to use the strong contexts of the Wikipedia articles to create clusters, in order to (hopefully) get very good clusters that few or no other publicly available training sets can give.

[6] As data sets that have been tested thoroughly are available

4.4 Summary

So what is clustering good for? In this work we propose to use clustering in order to obtain better search result. The idea is to use Wikipedia as a training set (learning source), relying on its strong contexts and categories and run clustering on this set. Then, documents downloaded from the Internet will be indexed and categorized using the Wikipedia clusters such that similarities in the downloaded documents are easier to find. We feel that this could give more precise and relevant search results.

Based on the discussion in this chapter, the documents to be clustered from the index will be represented using the VSM. Similarity between documents is found using the cosine similarity measure and the term-vectors will be weighted using *Tf x Idf* in order to represent both long and short documents properly.

The algorithms we have chosen to look at are discussed in more detail in the following chapters. These are *Spherical k-means*, *Non-negative matrix factorization* and *Frequent term-based clustering*. The algorithms are chosen because of their different structure and features. Spherical k-means is a non-overlapping, partitional clustering algorithm while Non-negative matrix factorization provides overlapping clusters using the spectral clustering approach (dimensionality reduction). Finally, the Frequent term-based clustering algorithm has the ability to be an overlapping and a non-overlapping algorithm. The algorithm is also a greedy, partitional clustering algorithm, but looks simpler than the Spherical k-means algorithm and therefore an interesting algorithm to explore.

Chapter 5

Spherical k-means

Spherical k-means [15], introduced by Dhillon and Modha in 2001, is a partitional clustering algorithm based on the k-means algorithm[29] from 1966. The k-means algorithm has spawned many variants such as Bisecting k-means[38], Parallel bisecting k-means with prediction[27] and Spherical k-means [15] and others.

In this chapter we will discuss the details of Spherical k-means as it has been shown to be very efficient compared to other k-means variants[43]. Other k-means algorithms will not be discussed further in this work. Note that the theory in this chapter is adopted primarily from [15] and therefore we choose to maintain the author's mathematical notation for simplicity.

5.1 Document representation

When clustering, the documents can be represented using the *VSM* as described in chapter 4.1. When using Spherical k-means the document term-vectors are normalized using the l^2 norm (also known as Euclidean norm). The l^2 norm is defined as

$$|x|_2 = \sqrt{\sum_{(k=1)}^{n} x_k^2} \tag{5.1}$$

This norm is a vector norm that normalizes all document term-vectors to unit length. Visually, this means that the documents can be seen as lying on the surface of a hyper-sphere[1] with radius one. Such normalization is done to capture the direction of a document and to ensure that documents of different lengths, but with same direction, are located at the same place in space (approx.). The normalization also helps to ensure that documents pointing in the

[1] A high dimensional sphere with dimensions ≥ 4

same direction but having different lengths, do not get assigned to different clusters. Looking at the left part of figure 5.1, the documents can be divided into vectors having four directions, but different lengths, meaning that the vocabulary is similar but the term frequencies are different. The right part of figure 5.1 shows the effect of normalizing the documents to unit length. The documents align into four clusters on the edge of the unit circle. Although not perfectly, but clearly enough for the clusters to be identifiable. Had the documents not been normalized, as in the left part of the figure, the documents might have been divided into 5 clusters (two at the bottom and three at the top). These clusters would have more similar term frequencies but less similar vocabularies compared to the clusters with the normalized documents.

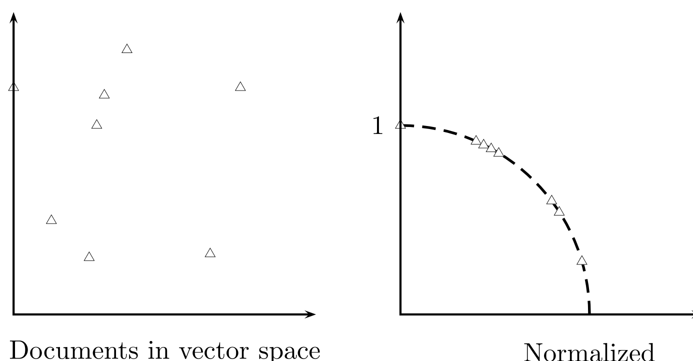

Figure 5.1: Normalizing vector space

With document vectors normalized, the *Cosine Similarity Measure* described in chapter 4.1.1 is used to measure similarity between them. The Spherical k-means algorithm performs clustering on the high dimensional sphere created by the document normalization and is therefore called *Spherical k-means*.

The algorithm is described graphically in figure 5.2 on page 54 and this figure will be referenced as the algorithm is explained further. Since the algorithm works on a hypersphere which is difficult to visualize, the workings of the algorithm has been projected down to two dimensions (along with other figures) for the sake of simplicity and visualization. This means that the surface of the hypersphere is projected down to the two-dimensional plane in figure 5.2.

5.2 Algorithm

Given n document vectors $x_1, x_2, \ldots, x_n$in $R^d_{\geq 0}$, d being the document vector dimensions and ≥ 0 denotes the positive part of R^d, then $\pi_1, \pi_2, \ldots, \pi_k$ are called the k disjoint clusters derived from the document vectors such that

$$\bigcup_{j=1}^{k} \pi_j = \{x_1, x_2, \ldots, x_n\} \text{ and } \pi_j \cap \pi_l = \phi \text{ if } j \neq l \tag{5.2}$$

Equation 5.2 means that the clusters generated are disjoint (non-overlapping) and the mean vector for a cluster π_j is easily computed as

$$m_j = \frac{1}{n_j} \sum_{x \in \pi_j} x \tag{5.3}$$

where n_j is the number of documents in cluster π_j. Thus, the direction of the mean vector is given by

$$c_j = \frac{m_j}{|m_j|_2} \tag{5.4}$$

where c_j denotes a *centroid* normalized to unit length. A centroid is defined as a vector that is closest in cosine similarity (in average) to all documents in cluster π_j.

5.2.1 Cluster quality

The coherence, or quality of a cluster can be measured by the dot product for each cluster $\pi_j, 1 \leq j \leq k$,

$$\mathcal{Q}_j = \sum_{x \in \pi_j} x^T c_j \tag{5.5}$$

If two documents are identical, the dot product equals 1 according to the definition of a the dot product with the lengths normalized to 1. Equation 5.5 returns a value between $0 \leq v \leq n$ where n is the number of documents in π_j. If $v = n$ then all documents are identical, meaning the angle θ between the documents is 0 degrees which implies that the closer v is to n the better the quality of the cluster.

The Cauchy-Schwarz inequality is defined as:

$$\sum_{x_i \in \pi_j} x_i^T z \leq \sum_{x_i \in \pi_j} x_i^T c_j \tag{5.6}$$

where z can be any vector in a high dimensional space.

Equation 5.6 states that combining all vectors in a cluster you are closer to the clusters centroid than any other vector. As equation 5.6 states, is is not possible to get closer to a minimum average distance (or angle) to all other vectors in the cluster than the centroid. Hence, the quality measure of any given cluster can by defined as

$$\mathcal{Q}\left(\{\pi_j\}_{j=1}^{k}\right) = \sum_{j=1}^{k} \sum_{x \in \pi_j} x^T c_j \tag{5.7}$$

Equation 5.7 is the function that Spherical k-means tries to maximize. Such a maximization is, as mentioned, NP complete and Spherical k-means is an approximation to this maximization problem[15].

The quality function, (equation 5.7) is used to measure the quality of the clustering between each iteration and as a stop criteria. The stop criteria for this algorithm is when the quality function improves less than a given value, ϵ, between iterations or after a predefined number of iterations.

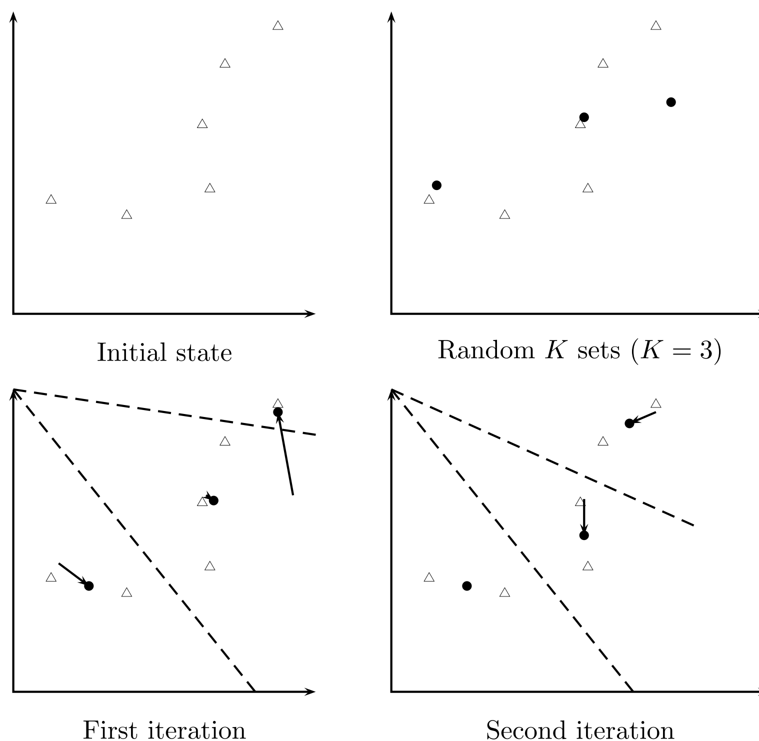

Figure 5.2: Spherical k-means

5.2.2 Clustering step by step

When the Spherical k-means algorithm starts, the documents (represented as triangles in figure 5.2) are either randomly assigned to centroids (represented as dots in figure 5.2) or with some prior knowledge and the index of iterations is set to $t = 0$. The algorithm proceeds as follows:

1. For each document find the closest centroid and assign the document to that centroid.

$$\pi_j^{t+1} = \left\{ x \in \{x_i\}_{i=1}^n : x^T c_j^{(t)} > x^T c_l^{(t)}, 1 \leq l \leq n, l \neq j \right\}, 1 \leq j \leq k.$$

2. Compute the new centroids for each partition of documents found in step one using equations 5.3 and 5.4. This moves the centroids closer to the maximum of the quality function in equation 5.5 (see figure 5.2).

3. If the stopping criteria is met, then stop, otherwise go to step one.

The number of centroids (clusters) to be created when running the Spherical k-means algorithm has to be known from the start. The starting positions of the centroids is also very important. Had the starting centroids been placed differently in figure 5.2 or the documents assigned to other clusters, the calculated centroids would probably converge differently. Hence, the algorithm

initialization is very important since a poor initialization converges slower and could yield worse clusters compared to a good initialization.

5.3 Summary

Spherical k-means creates a disjoint set of clusters (centroids) where all documents are included. This means that every document is assigned to a cluster and no document is assigned to more (or less) than one cluster. The clusters centroids returned by the algorithm can be seen as the direction of the most general document within the cluster. Hence, the centroids can be seen as the *topic vectors* describing the clusters.

A negative aspect of the algorithm is its non-deterministic nature. If initialized with different starting centroids, different resulting clusters are found. The starting centroids play a very important part in the creation of good clusters. Much research has been done with regards to the initialization of the algorithm and the resulting centroids from the Bisecting k-means algorithm have even been used as starting centroids for Spherical k-means [41]. However, the random initialization seems, in general, to be adequate and will therefore be used in our implementation.

Chapter 6

Nonnegative matrix factorization (NMF)

When clustering large datasets which are represented by the VSM, the dimensions of the term-document matrix can be enormous. In order to perform clustering on these matrices efficiently, matrix decomposition is used for rank reduction purposes. The general idea is to factor the large term-document matrix, $\mathbf{T}$, into smaller matrices, $\mathbf{W}$ and $\mathbf{H}$, in order to make calculation easier, that is:

$$\mathbf{T} \approx \mathbf{WH} \tag{6.1}$$

where $\mathbf{T}$ is a $n \times m$ matrix, $\mathbf{W}$ is a $n \times r$ matrix and $\mathbf{H}$ is a $r \times m$ matrix..

Many methods are used to accomplish this, such as *Independent Component Analysis* (ICA), *Probabilistic Latent Semantic Indexing* (PLSI) and many more. The methods use *Singular Value Decomposition* (SVD) to decompose the matrix and hereby try to reduce the term-document matrix to a certain rank, r, which corresponds to the number of clusters in the dataset. In order to find the best approximation, the SVD methods (such as ICA, PLSI etc.) minimize the Frobenius norm of the difference between the original matrix, and the approximation. However, the SVD methods allow negative values in their decompositions, which does not always make sense since documents in the semantic space are non-negative, i.e. an entry in the term-document matrix is either positive (if the word is in the document) or zero (if the word is not in the document). This makes *non-negative matrix factorization* (NMF) a good method for these types of approximations.

This chapter discusses the general NMF method introduced by Lee and Seung [25] which we chose to look at in this thesis. The notation in this chapter is the same as Lee and Seung use in [25] for the sake of simplicity.

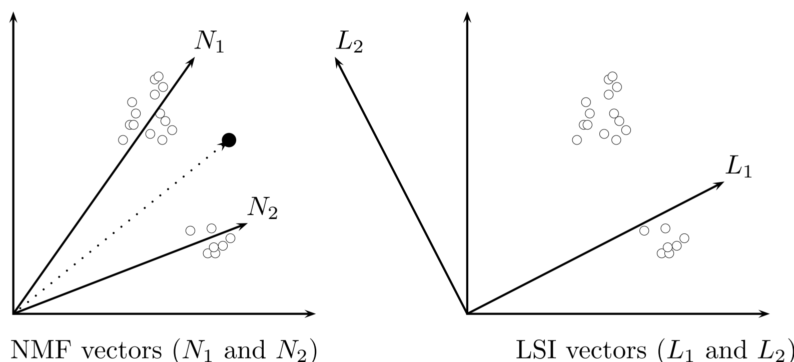

Figure 6.1: Difference between NMF and LSI (figure adopted from [45] with minor changes)

6.1 Standard NMF-Algorithm

According to [47], the sparseness of documents in the semantic space, the documents non-negative nature and the fact that the documents are topic-based, makes NMF a better choice than SVD methods. In general, NMF is an unsupervised learning algorithm which has been shown to outperform traditional vector space approaches such as LSI[45]. Experiments shown in [45] also indicate that NMF surpasses SVD and eigen-vector based methods in accuracy and reliable cluster derivation. Therefore, this method seems like a very logical choice for the clustering needed in this thesis. A general problem with LSI and eigenvector-space models is that the resulting eigenvectors do not directly describe the individual clusters.

Another drawback of LSI is that the resulting topic-vectors of the semantic space are required to be orthogonal. This is not the case with NMF. Looking at figure 6.1, it can be seen that while NMF would divide the documents into two specific clusters, LSI would put all the documents in the same cluster, since the angle between the two document clusters is less than 90 degrees.

This makes NMF more suited for overlapping clusters, since a document can easily contain more than one topic. Looking at figure 6.1, the solid dot between the clusters represents a document which contains both topics, i.e. belongs to both clusters. The document vector for that document is the dotted line in the figure. More precisely the document vector is equal to $\frac{1}{2}N_1 + \frac{1}{2}N_2$. This shows how NMF can handle documents with overlapping clusters (topics) and how the document vectors are an additive of the basis vectors (topic-vectors).

6.1.1 Initial problem

As mentioned above, the NMF method tries to find an approximation to the term-document matrix as shown in equation 6.1. NMF differs from other rank reduction methods by producing non-negative basis vectors for the semantic space. These basis vectors are also called *topic-vectors* and they describe the

vocabulary for each cluster[1]. NMF produces an overlapping clustering (part-based) where each document in the data set is represented as an additive combination of the topic-vectors.

So the basic idea is to factorize the term-document matrix into two matrices which yield a good approximation to the original matrix. The matrices **W** and **H** are not unique. They change after every iteration of the algorithm and the quality of the approximation depends greatly on the initialization of these matrices. That is, the better the initialization, the faster the algorithm will converge to an acceptable solution.

The quality measure of the approximation can be measured by calculating the Frobenius norm[2] of the difference between the original matrix and the approximated matrices. For a matrix **A**, the Frobenius norm is defined as:

$$\|A\|_F^2 = \sum_{i=1}^{m} \sum_{\mu=1}^{n} |a_{i\mu}|^2$$

This means that the NMF method seeks to minimize the objective function (cost function):

$$\|\mathbf{T} - \mathbf{WH}\|_F^2 = \sum_i \sum_\mu (T_{i\mu} - WH_{i\mu})^2, \text{ where } \mathbf{W}, \mathbf{H} \geq 0 \tag{6.2}$$

Hence, the quality of the approximation is measured by the value of the Frobenius norm. The closer the norm is to zero, the better the approximation.

6.1.2 Updating rules

In order to solve the problem in equation 6.2, Lee and Seung [24, 25] present a multiplicative update rule that they describe as a good compromise between speed and ease of implementation. These rules are defined as:

$$H_{a\mu} \leftarrow H_{a\mu} \frac{(W^T T)_{a\mu}}{(W^T W H)_{a\mu}} \tag{6.3}$$

$$W_{ia} \leftarrow W_{ia} \frac{(T H^T)_{ia}}{(H H^T W)_{ia}} \tag{6.4}$$

where a denotes the topic-vectors, i.e. $1 \leq a \leq r$. The update rule is used to update the approximated matrices between iteration without having to recalculate the whole approximation. This makes the algorithm faster and more efficient. The algorithm outline is as follows:

1. Initialize **W** and **H** with non-negative values
2. Iterate for each a, μ and i until convergence or after maximum l iterations
   (a) Update **H** using update rules
   (b) Update **W** using update rules

[1] Much like the centroids in Spherical k-means

[2] Also known as the Euclidian norm

The most common way of initializing the approximation matrices is simply to use random numbers. Other initialization methods will not be discussed here.

Using the above update rules, Shahnaz *et. al.* [37] state that complexity of the algorithm is $O(rmn)$ for r-clusters and a $m \times n$ term-document matrix.

6.2 Summary

The NMF algorithm discussed in this chapter is the multiplicative algorithm introduced by Lee and Seung in [24]. The algorithm is fairly simple to understand and implement. It has been used widely and seems to be very efficient.

Several other NMF algorithms have been proposed such as *Gradient Descent Algorithm*[24] or *Alternating Least Squares* [7]. These algorithms differ mainly in their update rules, but also in their objective functions. Yang *et. al.* [47] also introduce an algorithm, *Sparse Non-negative Matrix Factorization* where they utilize, and control the sparseness of the term-document matrix in order to improve cluster quality. These methods were all possible candidates in this thesis, but Lee and Seung's algorithm was chosen because of its simplicity and the good results it produces.

Chapter 7

Frequent term-based text clustering (FTC)

While searching for promising and appropriate clustering algorithms, we found an article describing an algorithm called *Frequent term-based text clustering* (FTC).

The algorithm looked very simple, easily implemented and had comparable results to other algorithms such as *Bisecting k-means*. The resulting clusters were of similar quality but the algorithm was said to be faster than the traditional algorithms. Last but not least, the algorithm not only clusters documents by their frequent term sets, it also returns a description of each clusters i.e. the terms that best describe the clusters - a good quality in order to understand the content of the clusters.

In this chapter we will briefly describe how the algorithm works and discuss the pros and cons of it. This entire chapter is based on [5] where the algorithm is described and evaluated. The author's notation is used for the sake of simplicity.

7.1 Definitions

Before describing the algorithm itself, some definitions are needed. First, let D be the set of all documents, i.e. $D = \{D_1, \ldots, D_n\}$ and T be the set of all terms occurring in these documents. Then each document can be represented by its terms, e.g. $D_j \subseteq T$. Further:

$$cov(S) = \{D_j \in D | S \subseteq D_j\}$$

where S is a set of frequent terms, and $cov(S)$ is the set of all documents containing all the terms of S.

The set of all frequent term sets in D is defined as $F = \{F_1, \ldots, F_n\}$. The cover of these frequent terms sets can be regarded as *cluster candidates*. A description of these cluster candidates would be the terms in the frequent term sets.

A clustering description for the document-set D, would be the set of clusters that satisfy the condition:

$$\bigcup_{i \in I} cov(F_i) = D$$

That is, together, all clusters must cover all documents in the database.

The algorithm tries to find a clustering with minimum overlap of the clusters. Ideally, each document can only belong to one cluster. This is not very likely so a measure for cluster overlap has to be defined. First, let f_j be the number of frequent term sets that contain document D_j:

$$f_j = |\{F_i \in R | F_i \subseteq D_j\}|$$

where R is the set of frequent terms sets not yet selected and $||$ is the cardinality of a set. Now the overlap of a cluster C_i is small, if the values of f_j are small. If each document only supports one cluster, i.e. $f_j = 1$ for all j, then $C_i = 0$ for all other cluster candidates. The *standard overlap* is defined as:

$$SO(C_i) = \frac{\sum_{D_j \in C_i} (f_j - 1)}{|C_i|}$$

Given a frequent term set of n-terms, any subset of that set is also a frequent term set, meaning that any document supporting the n-term set, will also support any subset of that set. The effect of this property is that a cluster candidate with many terms will have a much larger standard overlap than a candidate with few terms, thus favoring the smaller frequent term sets. Due to this shortcoming, another overlap is defined, based on entropy. Here, $p_j = \frac{1}{f_j}$ denotes the probability of document D_j belonging to one cluster candidate. Again, ideally, $p_j = 1$ if document D_j only belongs to one cluster candidate. Conversely, p_j becomes very small for large f_j values. Thus, the entropy overlap is defined as:

$$EO(C_i) = \sum_{D_j \in C_i} -\frac{1}{f_j} \ln \frac{1}{f_j}$$

The entropy overlap is 0 if all documents in the cluster do not support any other candidates, i.e. if $f_j = 1$ for all documents.

7.2 Algorithm

The algorithm is very dependent on the calculation of the frequent term sets. The basic outline of the algorithm is as follows:

1. Determine the frequent term sets
2. For each remaining frequent term set:
   (a) Calculate overlap for the set (standard or entropy)
   (b) Find best candidate term set based on minimum overlap

(c) Add the best candidate term set to already selected term sets

(d) Remove the best candidate term set from the remaining sets

(e) Remove all documents in $cov(Best)$ from D and from $cov(Remaining)$

3. Return the clustering and the cluster descriptions (the terms)

This greedy algorithm produces a non-overlapping clustering and a clustering description for the terms in each cluster.

The article also introduces a hierarchical version of the algorithm, but as the results for that algorithm are similar to the results of the flat version, it will not explained in any detail here.

7.3 Summary

At first glance this algorithm seemed like a very good choice for our clustering purposes. It is simple, easy to understand and gives a natural description of its clusters. However, we also found some problems with it.

First of all, the algorithm relies on an efficient way to determine the frequent term sets, but does not produce this algorithm. Implementing such an algorithm efficiently could make the whole implementation a lot more complex than intended, thus making the seemingly simple FTC implementation very complex.

Second, we could not find any articles on the Internet describing the results or experiences with the algorithm, making it hard to determine if it was suited for our purpose. Scalability is especially an issue in our case, since we are dealing with very large data sets (approx. 200.000 documents or more) and the article's results are based on much smaller data sets (no more than 9.000 documents). Furthermore, the number of clusters in the data sets used in the article are relatively small, ranging from 3 to 52 clusters. In our dataset, we presume that the number of clusters needed could be as many as several thousand clusters. This scalability and experience issue was of great concern to us.

Finally, trying out the example in the article, we could not produce the presented results. This severely undermined our faith in the algorithm and was the last and decisive factor in our choice not to implement it. The algorithm seems like an effective and fast way of clustering smaller data sets with fewer, and non-overlapping clusters. Whether it scales well to larger sets remains to be seen.

Chapter 8

Clustering discussion

In this chapter we will sum up the clustering discussion from previous chapters. First we will discuss our choices regarding the algorithms, e.g. which ones we have chosen and why. Then we will discuss the chosen algorithms weighed against each other, i.e. pros and cons of each algorithm along with the similarities between them. Finally, we will mention what our framework is capable of at this point and how we intend to use the elements we have discussed so far in this work.

8.1 Algorithm choices

In the previous chapters three different algorithms used for text-clustering were discussed. The FTC algorithm looked very promising, but due to the dimensionality of the used dataset, enough arguments supporting the use of this algorithm could not be found. Therefore, only the NMF and the Spherical k-means algorithms are considered.

The most obvious choice seems to be the Spherical k-means algorithm. This is, in part, due to the algorithms simplicity, but also because the algorithm has a fast convergence, i.e. takes few iterations to return a solution. Each iteration, in the dimensions used, is very time costly and therefore if the algorithm converges fast, it makes a great deal of difference in the calculation of the clusters. However, the seeding of the algorithm can cause some problems. Random initializations do not necessarily return the same results every time, in fact it is very unlikely they will. Therefore, a good starting point is vital for a good final solution.

Despite the initialization issue of the Spherical k-means, its use can still be justified, since NMF also relies on a good (random) initialization. However,

NMF has been shown to find better solutions but with more iterations. Still, NMF is not that complex with regards to implementation, so it is still a candidate algorithm if Spherical k-means fails to give the results needed.

8.2 Algorithm pros, cons and similarities

The NMF and Spherical k-means algorithms were selected in this work based on their qualities. However, both algorithms also have drawbacks. Even though the algorithms are structurally very different, there are some similarities between them.

8.2.1 NMF discussion

One of the best qualities of NMF are the resulting overlapping clusters. This type of clustering is more intuitively correct, i.e. closer to how humans would perform clustering by allowing documents to belong to more than one cluster. Furthermore, the NMF algorithm is fairly simple to implement and rather elegant in its updating between iterations. It has given good results when it comes to cluster quality and it returns a set of vectors that describe each cluster. These are all features that make NMF an attractive choice.

However, one of the main drawbacks of the algorithm is its slow convergence. The algorithm can require many iterations before finding a good solution. This depends on its initialization, which is usually done by random assignment and also the mathematics behind the matrix factorizations can be very computationally difficult, especially in very high dimensions. Computer limitations could become a problem when dealing with many clusters, but this of course depends on the available computer hardware.

8.2.2 Spherical k-means discussion

Like NMF, the Spherical k-means algorithm returns a set of centroid vectors which describe the individual clusters. The algorithm usually converges fast, i.e. only requires a few iterations to find a proper solution. The calculations for each iteration are simple mathematically, but grow linearly in accordance to the dimensions of the documents, the number of documents and the number of clusters to divide the data set into. As the number of documents and clusters grow, more similarity calculations are needed at each iteration. That is, each document has to be measured against more cluster centroids making the execution time longer. To try and reduce this effect, Elkan [16] uses the triangle inequality to reduce these similarity calculations in the standard *k-means* algorithm. His experiments show that on some datasets, the algorithm is up to 350 times faster than the standard algorithm[1]. The details of his experiments will not be presented here, but curious readers are referred to [16] for more details.

[1] Using similarity calculations as the time measure

The Spherical k-means algorithm has a very simple structure and is easily implemented because of its simple mathematics. However, due to its simplicity, the cluster quality suffers. NMF has given much better results with regards to cluster quality compared to Spherical k-means. This can of course be due to the non-overlapping structure of the algorithm. Like NMF, the algorithm also relies on proper initialization to find a good solution. Bad initialization can lead to more iterations and/or worse cluster quality.

8.2.3 Algorithm similarities

As described, the algorithms' structures are very different. However, there are some similar features between them. For instance, both algorithms rely heavily on the initial clustering in order to find a good solution. Both algorithms represent data in the same way, i.e. using high-dimensional term-document matrices although NMF works on a dimensionality reduced approximation to the original representation. The most important similarity between the algorithms lies in their cost functions.

For NMF the objective function (cost function) is defined as:

$$\begin{aligned} E_{NMF} &= \|\mathbf{T} - \mathbf{WH}\|^2 \\ &= \sum_{d=1}^{D}\sum_{i=1}^{I}(x_{di} - \sum_{k=1}^{K}\mathbf{W}_{dk}\mathbf{H}_{ki})^2 \\ &= x_{di}^2 + \widetilde{x}_{di}^2 - 2x_{di}\sum_{k=1}^{K}\mathbf{W}_{dk}\mathbf{H}_{ki} \end{aligned} \tag{8.1}$$

where D is the number of documents in the collection, K is the number of clusters and I is the number of elements in the vectors (the term list). Further,

$$\widetilde{x}_{di}^2 = \left(\sum_{k=1}^{K}\mathbf{W}_{dk}\mathbf{H}_{ki}\right)^2$$

If document vectors are normalized to have unit length, and assuming

$$x_{di}^2 \approx \widetilde{x}_{di}^2$$

then $x_{di}^2 + \widetilde{x}_{di}^2$ in equation 8.1 is merely a constant, leading to:

$$E_{NMF} \simeq 2 - 2\sum_{d=1}^{D}\sum_{i=1}^{I}\sum_{k=1}^{K}x_{di}\mathbf{W}_{dk}\mathbf{H}_{ki} \tag{8.2}$$

where clearly equation 8.2 should be minimized in order to get the least value of the cost function.

The quality function for Spherical k-means is defined in equation 5.7 on page 53. This function measures the quality of individual clusters and therefore only deals with documents that belong to the individual clusters. By maximizing the quality of each cluster, the function basically tries to minimize the difference

between the document vectors and the centroid vectors. Thus, the function can be rewritten as a cost function in the following way:

$$E_{SKM} = \sum_{k=1}^{K} \sum_{d \in \pi_k} (x_d - c_k)^2 \tag{8.3}$$

The above equation only deals with documents within a single cluster. To rectify that, the equation is multiplied with a d-by-k matrix $\mathbf{M}$ consisting of only zeros or ones where $\mathbf{M}_{dk} = 1$ means that document d belongs to cluster k and $\mathbf{M}_{dk} = 0$ means the document d does not belong to cluster k. Thus, the cost function for Spherical k-means can be expressed by::

$$\begin{aligned} E_{SKM} &= \sum_{k=1}^{K} \sum_{d}^{D} (x_d - c_k)^2 \mathbf{M}_{dk} \\ &= \sum_{k=1}^{K} \sum_{d}^{D} \sum_{i=1}^{I} (x_{di} - c_{ki})^2 \mathbf{M}_{dk} \\ &= x_{di}^2 + c_{ki}^2 - 2 \sum_{k=1}^{K} \sum_{d}^{D} \sum_{i=1}^{I} x_{di} c_{ki} \mathbf{M}_{dk} \end{aligned}$$

where K is the number of clusters, D is the set of all documents and I is the number of elements in the term vector. Also, since all vectors are normalized to unit length, $x_{di}^2 + c_{ki}^2 = 2$ the above equation becomes:

$$E_{SKM} = 2 - 2 \sum_{k=1}^{K} \sum_{d}^{D} \sum_{i=1}^{I} x_{di} c_{ki} \mathbf{M}_{dk} \tag{8.4}$$

Again, the goal is to minimize equation 8.4 to get the smallest cost value and thereby the best quality.

Comparing equations 8.2 and 8.4 shows that they are very similar. Given that the vectors in NMF are normalized to unit length, and the matrix $\mathbf{W}$ would only contain zeros and ones, the objective functions would virtually be the same since the vectors in $\mathbf{H}$ can be seen as centroids for the clusters much like the vectors c_k.
In the NMF cost function, the assumption that the two vectors x and $\widetilde{x}$ are equal will seldom be true. The cost functions are though very similar in many ways as the equations clearly show.

8.3 Current state of affairs

Using the clustering algorithms, the plan is to cluster the Wikipedia article training set, described in Part I, to get good clusters with strong contexts. We then want to index the downloaded data (also discussed in Part I) on top of the clusters, meaning that for each downloaded document we calculate which cluster the document is most similar to (using the cosine similarity measure). The document will then be added to the appropriate cluster in the cluster index,

and then indexed in the usual way. The idea is that when submitting a query to the search engine, it will find which clusters the documents belong to such that the user can select the most relevant cluster.

For example, lets say a query is submitted using the word *jaguar* and the user is looking for information about the animal. Then the search engine will find results in, lets say 3 clusters and present them unsorted with the clusters being, *animal*, *british car* and *Formula 1 team*. The user can then select the *animal* cluster, thus removing all the non-relevant results from the other clusters.

This is how we intend to utilize the clustering. However, in order to achieve this, we must implement the retrieval process of the search engine along with the ranking part of it. In the next chapter we test the selected clustering algorithms and discuss their results.

Chapter 9

Clustering tests

In the previous chapters we discussed various algorithms that can be used to cluster unstructured data. We have chosen to start with the Spherical k-means algorithm and see how it performs.

In this chapter we will discuss our implementation and test strategy for Spherical k-means clustering. We will also present the results of our tests and try to draw conclusions as to how clustering in our system is best achieved.

9.1 Data set

The original idea was to test the clustering on the full Wikipedia index as described in Part I. However, the full index of approximately 185.000 articles is very large (about 700MB) with more than 20.000 categories. It would require several GB of RAM to run clustering on this set, but also it would be difficult to manually inspect each cluster to assess the quality. Therefore, a smaller test set was created. The set was reduced to 49.748 articles by using the category *Musical groups* as the top level category with a total of 3.605 categories.

Before clustering the entire data set, Spherical k-means clustering was first run on a set with 10.000 Wikipedia articles on a Linux-grid engine. This grid has 32GB of RAM available thus making it the best place to run such memory demanding tests. The clustering of 10.000 Wikipedia articles took the grid engine 40 hours to cluster and used a staggering 2.83GB of RAM on the grid engine.

Based on these results the test data set was reduced even further. The data set was based on articles below the category *Blues* in Wikipedia. This data set contains 1283 articles divided into 93 categories with an index size of 9MB.

Using this set it was possible to run clustering in a reasonable amount of time on a 2.80GHz Dell computer with 512MB of RAM. Calculating clusters on the grid engine makes a huge difference in running time, but requires a lot of copying back and forth which is why it was desirable to be able to run the tests locally on ordinary PCs.

9.2 Test strategy

Using the blues data set, test scenarios have to be defined.

First of all, the implementation of the Spherical k-means algorithm needs to be tested. As discussed in chapter 5, the results of the algorithm depends mainly on two factors. Namely the number of clusters to divide the data set into and the initialization. Therefore, several initializations of the clusters are created and then the algorithm is run using the same number of clusters to see the effect of the initializations. The algorithm should also be to run on the blues set with various number of clusters to see how the number of clusters affect the final result. Furthermore, the algorithm performance compared to its baseline has to be investigated, i.e. measure the cluster quality if documents were assigned to clusters in a random fashion and the algorithm not run on the clusters. Therefore baseline clusterings are also created to measure against the algorithm results.

While implementing Spherical k-means, we got the idea of simply using Wikipedia's categories as clusters without any further calculations. For each category in Wikipedia, the list of articles belonging to that cluster was available and therefore it was decided to check the quality of these clusters and measure against the Spherical k-means algorithm. Note that there is a fundamental difference in the clusters provided by Wikipedia and the ones found in Spherical k-means. Wikipedia's structure allows overlapping categories, i.e. an article can belong to one or more clusters, while the Spherical k-means clusters have a flat non-overlapping structure. Therefore the clusters in Wikipedia are in general larger than the ones in Spherical k-means. Despite this difference, it is still interesting to see how these clusters looked compared to each other.

9.3 Quality measure

As described in chapter 5, finding the quality of a single Spherical k-means cluster is fairly simple. The quality of a cluster is simply the sum of the dot products between each document in the cluster and the cluster's centroids. This can also be seen mathematically in equation 5.7 on page 53. To properly represent different document lengths, each document is normalized to unit length (according to Spherical k-means) and weighted using the *Tf x Idf* weighting scheme described in chapter 4. This measure is implemented as described and will be applied to both the clusters from Spherical k-means and the clusters in Wikipedia with the centroid calculated first.

9.4 Test results

The tests are based on the described test strategy. The random initializations and baseline clusterings for Spherical k-means are done using MATLAB. A script was created that could write files with different numbers of clusters, overlapping and non-overlapping such that the algorithm could be initialized with these files. All the clusters are initialized to have approximately the same size but with random articles. This emulates the way Spherical k-means divides its initial document set into desired parts.

MATLAB was also used to generate overlapping baseline clusterings for Wikipedia. In order to keep the structure of Wikipedia the baselines were created with the same number of clusters and sizes as the actual Wikipedia clusters. This was done in order to get a more accurate baseline for the cluster structure of the blues category in Wikipedia. Using random sized clusters would not properly represent the structure of the blues set and would give results that did not compare to the actual Wikipedia categories.

9.4.1 Cluster quality and baseline measures

Using the test strategy, first the effect of different cluster sizes using the Spherical k-means algorithm was tested. Here ten different cluster sizes were. Wikipedia has 93 clusters for the blues set and therefore the algorithm was tested with both fewer and more clusters than Wikipedia. The 1283 articles were tested using the following number of clusters

$$c_s \in \{50, 60, 70, 80, 93, 100, 120, 140, 160, 180\}$$

For each test, the algorithm was randomly initialized as described above, but also a baseline quality was calculated for these random initializations without the algorithm being run. The results of these tests can be found in figure 9.1.

Figure 9.1 clearly shows how the cluster quality for both the algorithm and the baseline increases along with the number of clusters selected. Intuitively this makes sense since the quality measure is based on similarities whereas it is easier to find similarities in fewer documents than in many. It is also interesting to see that the difference between the baseline and the algorithm results is almost constant but the baseline quality is rather low. Using 93 clusters like Wikipedia, the algorithm has a cluster quality of 0.367072 but increases to 0.470877 when using 180 clusters. That is an increase in quality of about 28% which is quite impressive. The baseline quality for 180 clusters is 0.34083 which is close to the cluster quality found by the algorithm using 93 clusters.

After testing how different cluster sizes affect the results for Spherical k-means, the effect of random initialization of the algorithm using a constant number of clusters was tested. The same test was performed on Wikipedia's clustering to see how they compared. The test was performed as follows: 10 random initializations were created for both Wikipedia and Spherical k-means, the algorithm was run on the initializations and the baseline scores were calculated for both Wikipedia and the Spherical k-means. The initialization for

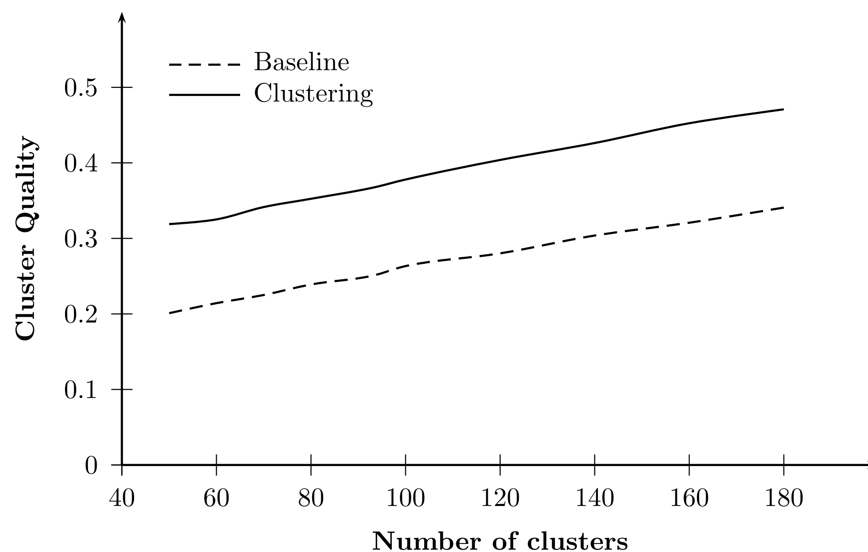

Figure 9.1: Spherical k-means baseline- and cluster-quality

Spherical k-means was non-overlapping whereas the Wikipedia initializations were overlapping.

Figure 9.2 shows the results of these tests. It is easy to see that Wikipedia has much higher quality for the clusters in the blues set than Spherical k-means can find. The algorithm does not even come close to the baseline quality of Wikipedia.

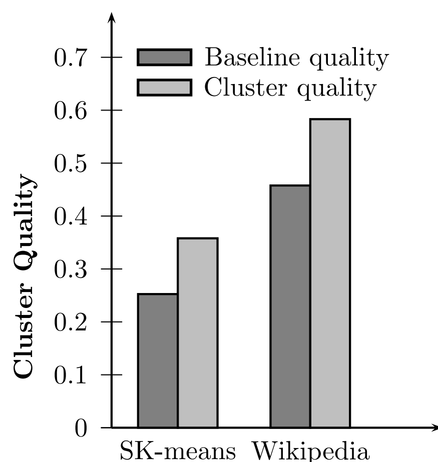

Figure 9.2: Wikipedia and Spherical k-means cluster quality

Table 9.1 lists the numerical results of the tests. Using Wikipedia's own clustering in unchanged form yields a quality of 0.5830990 with an average baseline

quality of 0.4577303 while Spherical k-means gives an average cluster quality of 0.3579345 with a baseline quality of 0.2525347. The big difference in quality probably lies in the fact that Wikipedia has some clusters with few articles and sometimes just one article, giving a quality of 1 for these clusters, whereas Spherical k-means has clusters with less variation in size, giving a more constant, but poorer quality in the clusters. Table 9.1 also shows that the minimum quality for Wikipedia is actually much worse than the minimum quality for Spherical k-means, again supporting the assumption of more constant quality of the algorithm.

	Wikipedia	**Spherical K-means**
Cluster min.	0.0510957	0.2080710
Cluster max.	1.0000000	0.7965290
Cluster avg.	0.5830990	0.3579345
Baseline min.	0.0452843	0.0853201
Baseline max.	1.0000000	0.3151620
Baseline avg.	0.4577303	0.2525347
Improvement	**27.39%**	**41.74%**

Table 9.1: Clustering statistics

9.5 Discussion

As shown in the previous section, dividing data into more clusters gives better cluster quality when using Spherical k-means. However, adding more clusters also has the drawback of requiring a lot of memory for large data sets and also has a negative effect on execution time. Calculating the clustering with many clusters takes a lot more time than using fewer clusters since each document needs to be measured against more centroids. On the other hand the cluster quality improves, so finding the best trade-off between number of clusters and cluster quality is essential when using Spherical k-means.

Our tests of the Spherical k-means indicate that initializing the algorithm using random cluster assignment results in relatively stable cluster qualities. The baseline qualities using random assignment also yield approximately the same cluster qualities for all tests. The same goes for the baseline tests for Wikipedia. Although higher than the baseline for Spherical k-means, the random initialization of Wikipedia clusters give almost the same cluster quality every time. The overall clustering quality of Wikipedia is also higher than the quality found using Spherical k-means which indicates that the human labeling of the Wikipedia articles is very good.

Despite the higher quality of Wikipedia's clustering, table 9.1 shows that the Spherical k-means algorithm improves its average cluster quality by an astonishing 41.74%, compared to its baseline, while Wikipedia 'only' has a 27.39% improvement compared to its baseline. This can be explained by looking at some

of the Wikipedia clusters, especially the clusters with very few articles. Randomly assigning these clusters will on average give a very good quality measure for these clusters, resulting in a much higher baseline value whereas the Spherical k-means clusters are all of equal size (approximately) with much lower average cluster quality. Therefore, Spherical k-means can improve the clusterings more compared to its baseline than Wikipedia. A randomly assigned Wikipedia cluster with one article will always have a quality of 1, regardless of what article gets assigned to it, thus making Wikipedia's baseline much higher than Spherical k-means baseline.

Table 9.1 also reveals more variation in the Wikipedia cluster quality compared to Spherical k-means. The minimum Wikipedia quality is 0.0510957 while it is 0.2080710 for Spherical k-means. The maximum for Wikipedia is 1, as mentioned before, compared to 0.7965290 for Spherical k-means. This large variation in Wikipedia cluster quality is due to the overlapping structure of the Wikipedia clusters, meaning that a document can easily be in more than one cluster, making the cluster larger on average. This hierarchical overlapping structure has clusters with many articles (> 200 for the blues set) but also clusters with only one article. Referring to figure 9.1, the cluster quality deteriorates as the clusters get bigger since it is difficult to get a high similarity value for many articles, thus resulting in a lower quality score for those clusters. This also applies to the Wikipedia clusters, i.e. that the larger clusters have very low quality values. Spherical k-means is not as affected by this since the algorithm has (on average) fewer articles per cluster, giving the worst clusters higher quality but at the same time giving the best clusters lower quality. This leads us to the following question:

> Does it make any sense to compare the cluster quality of Wikipedia and Spherical k-means?

The answer is not entirely clear, due to the different structures of the clusters. Wikipedia has a great advantage of having clusters with very few articles, but also suffers from the large clusters while Spherical k-means has more constant cluster sizes. The overlapping in the Wikipedia clusters probably does more damage than good, since the smaller categories in Wikipedia are more specific with higher quality. Figure 9.3 illustrates the overlapping structure. Here, the *Blues singers* cluster is very large and would have low quality, while the sub-clusters have more similar articles and thereby higher quality. If Spherical k-means was used on the data in figure 9.3, the clusters would be divided into equal parts (approximately). If one could be sure that all the articles in the large Wikipedia clusters are contained in one or more of the smaller categories (as shown in the figure), the largest categories could be dismissed, here *Blues singers*, and only use the smaller, more precise categories, thus giving a higher cluster quality for Wikipedia.

Essentially, the cluster quality measure is a mathematical way of measuring quality between the Wikipedia and Spherical k-means clusters. Examining the clusters manually would be far too time consuming and also relies on the evaluator's knowledge of the articles, therefore making manual quality measuring impossible. Although easy to use, it should be noted that the quality measure is merely a mathematical representation of similarity between documents based

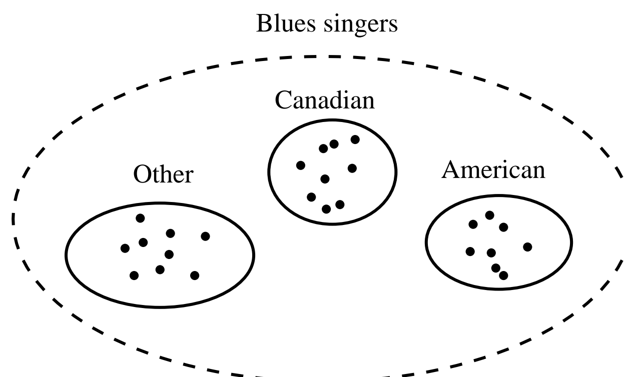

Figure 9.3: Wikipedia overlapping clusters

on the assumption that documents dealing with the same topic will make use of the same vocabulary. This is not always the case, since words can have many ambiguous forms (i.e. word polysemy) thereby making it possible to discuss the same topic using different words. In a recent article [12], Cucerzan from Microsoft Research introduces a framework that can be utilized to recognize disambiguations in the Wikipedia articles. Using this framework could help clustering texts which use different vocabularies for the same topics.

One could argue that the Wikipedia clusters are more precise, at least by human standards, since the users of Wikipedia have in fact manually labeled the articles, putting them in appropriate categories etc. Since the quality measure is a mathematical model of document similarity, the model falls short compared to human categorization. Humans simply would not categorize documents based on their specific vocabulary, but the general context. There is no way a mathematical model can imitate human behavior, but the quality measure does make sense both mathematically and intuitively. The question is whether the quality measure for non-overlapping clusters like the Spherical k-means clusters is suited for the overlapping structure of Wikipedia. Such research is beyond the scope of this work, but is worth mentioning as it is not entirely clear.

9.6 Summary

To sum up the tests and results, the only remaining question is whether to use Wikipedia's clusters or to rely on the Spherical k-means algorithm to provide the needed clusters. Besides the fact that Wikipedia has better cluster quality, by both human and mathematical standards, the overlapping structure in Wikipedia's clusters also makes it intuitively a better choice since humans could easily find that a document belonged to more than one category.

Nevertheless, Spherical k-means has great potential and perhaps, given enough clusters, time and memory, the algorithm could cluster the data-set with equal or better mathematical quality than the Wikipedia clusters, whereas no conclusion can be drawn as to how a human would feel about the resulting clusters.

Therefore we find it not only the easiest, but also the best way to simply use the clusters provided by Wikipedia. If at a later time we find that the Wikipedia clustering is not good enough, we have found the Spherical k-means algorithm reliable and very easy to use and would definitely try that as our clustering algorithm.

Part III

Retrieval

CHAPTER 10

Retrieval

With downloaded data preprocessed, indexed and clustered, the final element missing is document retrieval. The process of retrieving relevant documents from the *Zeeker Search Engine* index is divided into three different steps:

- Query Processing
- Document retrieval
- Ranking and presentation

In this chapter the implemented retrieval part will be discussed as well as some of the technical considerations found important and interesting during the implementation process. The following sections contain discussions of various retrieval related problems and what solutions were considered. Finally, the summary section at the end of the chapter sums up the implementation and the choices made.

10.1 Query Processing

Before a query can be submitted to the search engine, the query terms must be represented in the same way as the documents are represented in the index, i.e. using the Vector Space Model as described in section 4.1 on page 41. The resulting query vector will, in general, be a very sparse vector since queries are usually much shorter (often merely a term or two) than the documents they retrieve.

10.1.1 Vocabulary pruning

The vocabulary of these query vectors is not complicated due to their sparse nature. However, the query terms can still be submitted in any form, e.g. plural,

singular, upper-case etc. Previously it has been described how the vocabulary in the index is reduced using stop-word removal, stemming and Part-of-speech filtering. Obviously these techniques need to be applied to the query vectors since stop-words or unstemmed words in the query vectors will not match any terms in the index.

Although Part-of-speech filtering is used while indexing, it would not help in the query processing. This is due to the fact that POS-taggers can not find any semantic meaning within a few terms which most likely do not make up a correct sentence but only is a collection of search terms. Hence, the POS-filtering is omitted in the query processing while stop-words are removed and query terms are reduced to their stems.

10.1.2 Misspelled queries

One aspect of query processing is how the search engine deals with misspelled terms in the queries. Implementing this kind of features is not a simple task. If a term is misspelled in the query, the search engine would have to go through its vocabulary and find which terms are most similar to the misspelled term and present the most probable terms to the user. The user would then select the correct form of the terms before the query is submitted. With that in mind, and taking the number of unique terms in the index into account, this sort of feature requires quite a lot of research and consideration before it can be implemented properly.

However, some errors can be corrected without much effort. During query processing, the queries are parsed for unexpected symbols such that given a query like: "Er/ic Clapton", the search engine would identify the slash as a misplaced token and present the user with the alternative query - "Eric Clapton" . By parsing the query, some of the common misspellings can be reduced without having to calculate which term the misspelled terms are most similar to.

10.2 Query operators

Many well known search engines, such as Google and Yahoo!, make advanced search options available to their users. These options are used to submit more specialized queries. Most search engines make a broad search (by default) using the given query terms. It is then left to the users to make use of the advanced search options in order to force the engine to be more selective and strict in its evaluation of relevant documents thus giving the user a possibility of a more narrow search.

10.2.1 Search operators

The most common kind of search options is the use of special search operators. Common search operators include operators for exact matches of the query terms, Boolean AND and Boolean OR operators among others. Exact operators force the search engine to match the query terms in their exact form

and in the same order as they are submitted. A Boolean AND search tells the engine that all the query terms should be matched, but not necessarily in the submitted order. Using a Boolean OR means that any document containing any of the submitted terms is a match for the query.

Many other operators can also be found in modern search engines. Some make it possible to include or exclude words from resulting documents, limit a search to specific sites and many more. Suffice to say that users can make their queries quite precise if the right operators are used.

Search operators are not a necessity in a modern search engine, but given the enormous amount of data on the Internet, submitting a general query to a search engine is often not enough. Hence, some operators are made available in *Zeeker Search Engine* such that the users could have some control over their queries. Only the most common operators are implemented, i.e. Boolean AND, Boolean OR and exact match operator. The default query in the search engine is quite strict as it demands that all terms appear in the document where the terms should appear in the submitted order with at most one term between them. Query operators would therefore help users find more relevant documents if the strict default search returns no, few or too many results. The syntax for the operators will not be described here but can be found in the User Guide chapter in appendix A.

10.2.2 Category filtering

Thus far, much has been said about how documents are clustered in the index. Having achieved that, a description of how these clusters are presented to the search engine's users is in order.

The primary goal of the clustering is to provide a filtering mechanism on the retrieved documents. When the engine retrieves documents from the index it will calculate to what clusters these documents belong. With this information available, users are able to see which categories the results belong to. Furthermore, it is possible for users to select which clusters they want to use as filter. See appendix A for a screen-shot of a query example.

This kind of filtering could be implemented as search operators as described above, or as a list from which the users can select appropriate categories. However, presenting the users with the full list of categories could be a problem since there might be several thousand categories[1] thus making it a tiresome affair to find the appropriate filters to use. Therefore, the category filtering is implemented as a search operator. Users are also presented with links to the categories such that they can resubmit their queries with the category filtering enabled without knowing the exact syntax for the operators. The syntax for the category operator can be found in the User Guide in appendix A.

[1] There are over 3.000 categories for the *musical group* set

10.2.3 Query expansion

In chapter 1 the concept of query expansion was introduced. In short, query expansion is used to expand the submitted queries with additional query terms and/or reweighing the original query terms before submitting the query. The idea is that adding more terms to the queries will make the search engine more likely to find relevant results. Query expansion is especially effective when it comes to ambiguous query terms such as *jaguar*.

Despite the simple idea behind query expansion, it is in no way a trivial task. There are mainly three strategies that can be used to expand queries. These are *automatic*, *manual* or *user-assisted*.

Automatic expansion relies on the search engine itself to find what terms should be added to the query. Here clustering could be helpful since for each query term, the engine could calculate which clusters the terms belong to and get the most relevant terms for these clusters and then expand the query using these terms. However, this strategy will not always be feasible. Considering the term *jaguar* as an example, the engine might find that it belonged to two different clusters, one regarding the car and one regarding the animal. Popular terms would then be grabbed from these clusters and used to expand the query. The query would then contain terms used for both the animal and the car, thus making the query more difficult to match with a document in the index.

When using manual query expansion, users are presented with a list of terms to choose from in order to expand their queries. The term lists for the indexes can be very long, therefore making it impractical to display them all. Instead, user-defined expansion could be used alongside automatic expansions, such that the engine would grab terms from popular clusters and present to the user thereby making the user choose the appropriate terms and categories.

The final approach is to use user assisted query expansion. Relevance feedback, a form of user assisted query expansion, is when a user is presented with results to the original query in unchanged form and is then asked to tell the search engine what document is most relevant. The search engine then analyzes the document, grabs relevant terms from it, expands the original query with the grabbed terms and resubmits it. This way the user helps the search engine find the relevant document and categories.

Query expansion is without a doubt a useful tool when it comes to short queries, whereas expanding longer queries might be very difficult and could give worse results than the original queries. Relevance feedback is probably the best way to implement query expansion since the user indicates what is relevant and what is not. However, this kind of expansion means that the user has to submit two queries to find the wanted results. Clearly, retrieving relevant results first time around is preferable.

Although query expansion has great potential, it is not implemented in the first versions of the *Zeeker Search Engine*. As discussed above, great care must be taken when implementing query expansion, and with the limited time frame

for this project, query expansion was not implemented. It was also desirable to see how the search engine performed without any additional help from either users or additional terms.

10.3 Document retrieval

Given a processed list of query terms, the matching documents have to be retrieved from the index, ranked appropriately and presented to the user.

10.3.1 Retrieval method

With documents and queries represented as high-dimensional vectors using the vector space model, the matching documents are found using the *Cosine Similarity Measure* with documents normalized and term-weighting applied as described in section 4.1.1 on page 42.

The Cosine Similarity Measure is of course not the only way to retrieve documents from the index. Many sophisticated and more complicated algorithms have been developed to serve this purpose. However, starting with a simple method and see how it handled retrieval seemed logical. Furthermore, the simple retrieval method does not over-complicate the search engine structure.

10.3.2 Ranking

Since the vector space model and cosine similarity are used for retrieval, the ranking of the results becomes fairly simple. The ranking of documents is done by looking at the similarity measure between the query vector and document vectors. The higher the value, the higher the document will appear in the list of results.

In chapter 1, different ranking algorithms were introduced, such as PageRank and the HITS algorithm. These algorithms have produced very good results, but seem to be a bit too complicated to be incorporated in this work. The algorithms are primarily based on link analysis which is not used in the vector space model and the use of these was already dismissed in the introduction. Hence, simple document ranking based on the cosine similarity was the ranking method of choice.

The list of categories found also has to be ranked in a proper manner. Categories are ranked based on their category-score where the individual category-score is the sum of the cosine similarity scores for each document in the result list that is also a part of the category divided by the number of documents in the category. That is:

$$CS_i = \frac{\sum_{d \in D} SIM_d}{|CS_i|} \tag{10.1}$$

where CS_i is the category-score for cluster i, $|CS_i|$ is the number of documents in cluster i, SIM_d is the cosine similarity score for document d and D is the

union of documents in the result list and the documents in cluster i.
First it was considered to rank the categories according to popularity, i.e. the category with most document in the result list would be ranked first. However, this approach will always place the large categories at the top without any regard as to how good matches the documents were. Therefore, the cosine similarity score was added to the category score but had to normalized with the category size such that it did not again favor the large categories. This category ranking seems to rank the categories quite well.

10.3.3 Lack of results

As mentioned in the discussion on search operators, the default query in the *Zeeker Search Engine* is quite strict. Knowing that not all users are familiar with the use of extra operators in the queries, a secondary default query was implemented - i.e. relaxing the original query to retrieve more results. This query is submitted automatically when the results to a users default query returns no results.

10.4 Summary

When implementing the retrieval process of a search engine, months could be used to analyze the various problems and pitfalls. As described in this chapter, many things need to be taken into account and wrong decisions might lead to poor retrieval performance. The retrieval process is implemented such that user involvement is minimal, but at the same time satisfies the users with special demands. Hence, the implemented retrieval should be efficient, simple but still return acceptable results.

Query processing is done similar to document processing. Stop-words are removed and the search terms are reduced to their stems whereas POS-filtering is omitted due to the POS-taggers inability to deal with short sentences (queries). If search results are not satisfactory with the default search, search operators have been implemented to give the users more control over their queries. Three operators were implemented, namely *AND*, *OR*, and *EXACT*. Furthermore, a category filter is included in the query syntax, thus enabling users to search within a given category. The query syntax can be found in the *User Guide* in appendix A.

Even when restricting queries with search operators (or not restricting at all), results found in the index could be several thousand. *Zeeker Search Engine* only returns a maximum of 100 results to any query. The threshold is primarily based on prior search experiences, i.e. merely skimming the first tens of results thus strengthening our beliefs in the *few but relevant* mantra. In the introduction, the *few but relevant* mantra was introduced as one of the problems with search engines today - they return too many results.

Query expansion is not used in any way in *Zeeker Search Engine*, but it is one of the things that could be added at a later time. Great potential lies in

query expansion whereas great care must be taken when selecting which strategy to use. Query expansion can easily worsen the results if implemented or used in a wrong way.

Ranking of retrieved documents is done using the cosine similarity method. Categories are also ranked based on accumulated and normalized similarity scores as shown in equation 10.1.

Finally, the front-end of the search engine, and thus the actual retrieval and presentation, is implemented as a basic Web interface similar to many known search engines. The interface is clean, simple and easy to use. The User Guide in appendix A contains screen-shots and explanation of the various implemented features of the search engine. The implemented *Zeeker Search Engine* can be found online at:

http://studweb1.imm.dtu.dk

Chapter 11

Evaluating retrieval

> The astonishing growth of the Web propelled the rapid development of Web search engines. However, the evaluation of these search engines has not been keeping up with the pace of their development.
>
> - Liwen Vaughan, 2003 [39]

Measuring a search engine's retrieval in terms of performance and accuracy may seem like a trivial task for a user of the Internet, but trying to automate this process is in no way trivial. When discussing search engine measurements, usually *Precision, Recall and F-measure* come to mind. However, utilizing the user's feedback is of great importance and might be a more realistic way of measuring a search engine's quality.

In this chapter the F-measure will be introduced as well as how user feedback can be used to measure search engine accuracy.

11.1 Recall, Precision and F-measure

The most common way of measuring search engine performance is using the *F-measure* which is a mean, based on the trade-off between *precision* and *recall*. Precision is measurement for how many relevant documents are retrieved for a given query out of how many documents retrieved. In mathematical terms, precision is defined as:

$$P = \frac{D_{relevant\ retrieved}}{D_{retrieved\ total}} \tag{11.1}$$

Similarly, recall is the measurement for how many relevant documents are retrieved out of how many relevant documents there actually are. In mathematical terms, recall is defined as:

$$R = \frac{D_{relevant\ retrieved}}{D_{relevant\ \ total}} \tag{11.2}$$

The F-measure is then a measurement for the weighting of precision and recall. F-measure is defined as:

$$F_\alpha = (1+\alpha)\frac{P \cdot R}{\alpha P + R} \tag{11.3}$$

where the parameter α is the trade-off factor between precision and recall. For instance, if $\alpha = 2$ then recall is weighted twice as much as precision.

Using the F-measure is a fairly simple way to estimate search engine performance. Using the same datasets when measuring new search engines' performance gives a good picture of how they perform compared to each other. However, the drawback of using the same datasets is that search engines can be tuned to perform well on these specific queries and datasets, whereas they would not necessarily do so well on the Internet. This would give a very incorrect picture of the engines performance and accuracy. Furthermore, using these well known datasets, the queries and their results are known whereas the precise results are not known for the entire Internet. Hence, in order to use the F-measure properly, it requires datasets and queries that are well defined with known results.

11.2 User Feedback

The main problem, when it comes to measuring search engine performance, is that it requires a lot of human relevance judgment, which is quite a costly affair. Users' feedback can be very useful when trying to measure a search engine's performance. Given a list of queries, users can be asked to evaluate the results by, for example, going through the first 20 documents returned and evaluate the relevance of each one. This is of course a very slow and inefficient way of measuring performance, but might be the most precise way to evaluate the results. Since the users needed to represent all the different types of users on the Internet, and at the same time try every query possible, this might seem like an impossible approach. Therefore, this strategy is merely an estimate like the F-measure and no conclusion can be drawn from such an experiment. The users could also be biased toward a certain search engine or even toward certain web pages, thus skewing the results.

Researchers have tried to develop efficient methods that can automate the evaluation process. In [11], Can *et. al.* present an automatic method which they use to evaluate the performance of eight popular search engines. They also use human evaluation of the same engines for the same queries to see how well their method works. This method finds the same search engines to be in the top and bottom two places in the list, thus making it comparable to the human user feedback. However, the automatic method does not find as many relevant documents as the users did, but as mentioned before, correctly identifies the best and the worst among the eight engines tested.

In another study [39], Liwen Vaughan also used human user evaluation to rank a list of web pages. This ranking was then compared to the lists retrieved from three commercial search engines. New measures are presented in order

to measure the search engines' performance. These new measurements are described as follows:

- **[Quality of result ranking]** Described by the correlation between human and search engine ranking
- **[Ability to retrieve top ranked pages]** Top results for a query are taken from the search engines and merged into a single set. Human users then rank these results after relevance. The ability of a search engine to retrieve a relevant page is then measured as a percentage of how many pages were relevant in its top results.
- **[Stability over time (10-weeks)]** Stability of the number of pages retrieved and how many pages remain in the top results over a short period.

Liwen's research showed that these new measurements can distinguish search engine performance very well.

In a new study [2], Ali *et. al.* present an automated framework to measure engine performance. This framework takes advantage of user feedback, Cosine Similarity Measure, PageRank (as used by Google), Boolean Similarity Measure[1] as well as Rank Aggregation techniques to evaluate search engine performance. The framework presented submits a query to a search engine and presents it to a user but also stores the ranking. User feedback is then achieved through the user's click data, i.e. which links are followed, printed, bookmarked, saved etc. This click data is then analyzed and four new rankings are calculated based on the user's actions. These new rankings are calculated using Cosine Similarity Measure, Boolean Similarity Measure, PageRank and user feedback. The rankings are then aggregated into a new ranked list of results. Correlation is then calculated between the original list returned by the search engine and the new ranked list. The higher the correlation coefficient, the more effective the engine is. The four different measurements are used in order to avoid bias between the different structures of the search engines tested. Seven search engines were tested in the study with good results.

11.3 Summary

Since searching is without a doubt one of the most popular activities on the Internet and as the search engines get more complex, the measurements for how effective they are also have to keep up. Using the simple *F-measure* can be useful, but is very cost-inefficient if standard datasets are not used, since humans have to organize datasets and evaluate the relevance of each document with regards to the test queries. User feedback has been shown to give good results, but is also cost inefficient. A more automated approach is therefore desirable and the research within the field of search engine evaluation seems to be heading toward more automated evaluation methods.

[1] A simplified version of Li Danzig's [26] $S^{\otimes}$ measure is used to reduce computational effort.

Chapter 12

Testing retrieval

In this chapter, the testing of the implemented search engine (using the retrieval discussed in chapter 10) is described. The considered test strategies are discussed in the following section. Test scenarios and results will also be discussed. Finally, the testing is summed up and conclusions are drawn regarding the overall performance of the search engine.

12.1 Test strategies

In the previous chapter, two approaches commonly used to evaluate a search engine's performance and retrieval precision were discussed. Indeed the *F-measure* is simple and computationally easy, given datasets and queries that have been analyzed and where the desired outcome of each query is known. However, this is not the case with the datasets used in this work. The approximately 200.000 web documents are not labeled in any way, except for the clusters they are assigned to. The *Zeeker Search Engine* uses clusters to provide additional filtering of the result set and therefore measuring recall and precision on the original result set does not make any sense as the filtering is what makes *Zeeker Search Engine* different from other search engines. Therefore, the use of recall, precision and F-measure on known datasets was quickly dismissed.

Since the F-measure was of no use in this case, the only measure left was User Feedback. In the previous chapter, manual and automatic user feedback scenarios were described. The automatic scenario described by Ali *et. al.* in [2] seemed a bit excessive, and as the other strategies did not seem to fit the purpose either, it was decided to rely entirely on manual user feedback, i.e. have users test *Zeeker Search Engine* and give feedback about its performance and retrieval precision.

12.1.1 Selected test methods

Two methods were mainly used to test the engine. First of all, the retrieval part was tested using a trial-and-error approach, where the primary goal was to find errors in the retrieval logic and programming code. Trial-and-error was also used to see how the engine handled various potentially problematic queries[1]. These tests revealed some errors which were fixed before the search engine was put on-line for others to try out.

The second method used was manual user feedback. A questionnaire was constructed which was sent out to numerous people asking them to participate. Before creating the questionnaire, the information and answers valuable to the search engine's performance had to be defined. Based on general search behavior using search engines, e.g. Google, it was concluded that there were mainly two ways users use search engines, either for *question answering* (who is, what is etc.) or for *research* (what has been written about some topic). Therefore, the questionnaire should include questions that would give indications as to how well the search engine can be used for question answering and research respectively. To test the question answering part, users were asked to find answers to questions known to exist in the index, given minimal clues to go on. Researching was tested by asking the users to submit queries on their own and evaluate the relevance of the results returned by the search engine.

It was also considered asking users to evaluate results from predefined queries. This idea presented a couple of problems. First of all, users might not know anything about the chosen topic of the queries and would therefore be in no position to evaluate the relevance of the retrieved information. Furthermore, predefined queries known to give good results could also be selected, thus giving biased results making the questionnaire unreliable. Finally, this approach does not model the general search behavior mentioned above and therefore the use of predefined queries was entirely dismissed.

The devised questionnaire can be found in chapter B.2 in the appendix. The test results are presented and discussed in the next section.

12.2 Test discussion

The questionnaire was kept open for seven days where friends, family and everyone interested in participating was invited to participate. People were even encouraged to pass the invitation along to other people as well. After the seven day period, 24 people had anonymously answered the questions. Out of the 24 participants, there were 12 men and 12 women where 19 of these were between 21 and 30 years of age, 4 were between 31 and 40 years of age and 1 was above 50 years of age. The participants were based in Denmark, Iceland, France, USA and Sweden[2] - yet the exact nationalities are unknown. The vast majority (over 90%) of the participants rated their experience with search engines as interme-

[1] For example queries with many stop-words etc.

[2] Based on IP addresses

diate[3] or better. This yields a test group of users mainly in the age group of 21 to 30 years old, where most are well familiar with how search engines work and equally distributed among the two sexes.

With that in mind, the results of the questionnaire and the conclusions drawn from it will be presented.

12.2.1 Searching

Based on the described test strategy, the purpose was to test two different scenarios namely *question answering* and *research*. Participants were asked to find information known to exist in the index and to submit their own queries. In the following tables the key words in the table headers refer to the questions in the questionnaire where participants were asked to find information on these key words. The information retrieval tasks were:

1. Find a single from an album called *Ten*.
2. Find the real name of the artist which uses the stage name *The Edge*.
3. Find the name of the band behind the song *Lord of the Boards*.

The goal of these information retrieval tasks was to find out whether or not users were able to find useful information - this being the main functionality of a search engine. Table 12.1 shows how the 24 survey participants answered that question.

Answer	**Ten**	**The Edge**	**Lord of the Boards**	%
Yes	21	16	20	79.2
No	1	7	4	16.7
Don't know	2	1	0	4.2

Table 12.1: Did you find what we asked for?

Clearly the table shows that *Zeeker Search Engine* is capable of retrieving information when users are asked to find something known to exist in the index. Finding the real name of the artist behind the stage name *The Edge*[4] caused problems for several of the participants as seen in the table. The problem causing this was quickly located and lies in the handling of upper- and lower-case letters in the index. If searching for *the edge* the results are much more relevant than with the query *The Edge*. The answers to this question in the questionnaire resulted in a small but very serious bug fix.

Even though the participants seemed to be able to find the information required, knowing how difficult it was to find seemed important. People tend to try harder when participating in a survey than when trying out a new product at leisure. Table 12.2 shows how difficult the participants found the information retrieval tasks. Again it seems that the participants did not have problems

[3] Where Intermediate was defined as people well familiar with Google
[4] Guitarist David Howell Evans from the band U2

finding some of the information asked for, which is also confirmed by comment 1 in table 12.3. A number of users did however find it very difficult. Especially finding the band behind the song *Lord of the Boards*[5], where users had trouble finding the right answer. Again, the handling of uppercase and lowercase letters might be the source of this problem. If searching for *lord of the boards* or *Lord of the Boards* (as written in the questionnaire), Guano Apes (the correct answer) is number four in the list of results. However, if any of the stop-words in the query, i.e. *of* or *the* are written with capital letters, Guano Apes is not in the list of results. When users were faced with these problems, several of them requested more query operators (see comment 3, 6, 7 and 8 in table 12.3) as they believed the problem was a fundamental searching problem. Besides correcting the uppercase/lowercase problem, a future version of *Zeeker Search Engine* will also introduce more query operators to help users get the information they need. Future versions and extensions are discussed in chapter 14.

Answer	**Ten**	**The Edge**	**Lord of the Boards**	%
Very easy	7	8	2	27.0
Easy	5	3	7	23.8
Normal	8	3	6	27.0
Hard	1	2	5	12.7
Very Hard	0	3	3	9.5

Table 12.2: How difficult was it to find?

Zeeker Search Engine differs from other search engines on account of its categories. Obviously testing whether users found the categories practical and effective was necessary. Many found the categories a good additional tool when searching whereas a large percentage (37.5%) of the participants did not find them useful as shown in table 12.4 and supported by comments 6 and 7 in table 12.3. The main reason may be that the category filtering is not strict enough, meaning that too many web pages are clustered under categories they do not (strictly) belong to. This was also expressed by a participant (see comment 9 in table 12.3). This issue is not easy to rectify and is discussed further in chapter 14. The categories did however help some users as expressed in comments 4 and 5 in table 12.3.

In order to test the research capabilities of *Zeeker Search Engine*, participants were asked to submit queries of their own and then asked to rate the relevance of the results. Table 12.5 shows how the participants rated the relevance of the results to their own queries. More than 80% found the retrieved information relevant or better whereas only 4.2% found the retrieved information not relevant at all.

The tests on *Zeeker Search Engine*'s search capabilities have revealed that participants rate the ease of use, relevance of retrieved information and the ease of finding information very highly. At the same time, the survey also revealed a problem in the handling of upper- and lower-case letters which resulted in some poor ratings. *Zeeker Search Engine* already has future features planned that will hopefully make these numbers even better (see chapter 14).

[5] Song by Guano Apes

Id	Selected comments
1	Couldn't understand it at first, but then it was very easy.
2	Please include a spelling wiz to help the user.
3	I had some difficulties when searching for The Edge, did not get any results when writing with small caps and no relevant results when writing as shown in the survey...
4	Good work. Shouldn't be case sensitive though? Bugs aside, good for finding stuff within categories, i.e. 'I like rock, show me some bands.' For a specific search I'll rather google
5	I like the categories, they are very useful to guide the search in the right direction
6	The engine definitely needs the use of quoted expressions: I always use queries like "the beatles" "last single" op:AND if I want to find the last single issued by the Beatles. Furthermore I just couldn't find any use for the clusters - apparently they kept suggesting a partitioning of the results that I simply had no use for.
7	The operators arent as useable compared to google's, nor as usefull. I didn't get to use the categories a single time...
8	Had problems finding Lord of the Boards. The categories were just not useful there. An operator like SONG: could be very helpful in this case.
9	... I feel like the "filtering" possibilities are too broad. It would be great if you could somehow come up with more specific filtering for the users ...

Table 12.3: Selected comments

Answer	Ten	The Edge	Lord of the Boards	%
Very Useful	2	8	6	22.2
Useful	5	3	6	19.4
Somewhat Useful	7	4	4	20.1
Not Useful	10	9	8	37.5

Table 12.4: Did you find the categories useful?

Answer	Count	Percent
Very Relevant	5	20.8%
Quite Relevant	10	41.7%
Relevant	5	20.8%
Somewhat Relevant	3	12.5%
Not Relevant at all	1	4.2%
Total	24	100%

Table 12.5: How relevant were the results to your queries?

12.2.2 Overall evaluation

The participants' overall evaluation of *Zeeker Search Engine* is presented in tables 12.6 and 12.7. The majority of users found the performance between good and average whereas a future version will probably be able to increase performance and decrease the search engine's response time, i.e. the time it

takes the engine to respond to queries. The participants found the performance acceptable and adequate.

Answer	**Count**	**Percent**
Very Good	1	4%
Good	12	50%
Average	10	42%
Bad	1	4%
Very Bad	0	0.00%
Total	24	100%

Table 12.6: How do you rate our search engine's overall performance?

The overall performance statistics also reflect how likely a user is to use a topic-based search engine (like *Zeeker Search Engine*) in the future. In general, the participants are positive toward this kind of search engine (see Table 12.7) yet some (25%) find it unlikely or very unlikely to use such a search engine in the future. This is of course disheartening but the future plans and features for *Zeeker Search Engine* are believed to greatly improve the search engine thus hopefully lower the number of unsatisfied users.

Answer	**Count**	**Percent**
Very Likely	5	20.8%
Likely	10	41.7%
Unlikely	5	20.8%
Very Unlikely	1	4.2%
Don't Know	3	12.5%
Total	24	100%

Table 12.7: How likely are you to use this kind of search engine again?

12.3 Summary

Zeeker Search Engine has been tested by 24 individuals, equally distributed between men and women, primarily in the age group 21 to 30 years old. All participants have answered questions as to how hard it was to retrieve information, whether or not they found the requested information, how they would rate the results to their own queries and more.

The tests have revealed a few weaknesses in the search engine. When searching for information known to exist in the index, the participants found it easily. Yet some queries did not retrieve relevant results and an error in the search engine's logic was located due to this of lack results. The error has to do with how upper- and lower-case letters are handled in the index and it explains many of the negative feedbacks participants gave on the 'problematic' queries. Many participants requested more query operators to help in the search when no relevant information was retrieved as they seemingly believed that the lack of relevant results was due to poor retrieval techniques. Query operators will definitely help users in their search and more will certainly be implemented.

However, appropriate handling of upper- and lower-case letters will reduce the problems severely.

The questionnaire provided many inputs that gave a better understanding of how users rate, use and view the search engine. Query operators were in high demand and even a spelling wizard was requested thus indicating that the implemented error handling feature is not quite adequate.

Not only was it encouraging that the search engine retrieved relevant information, but also that many users found it likely that they would this kind of search engine again. All in all, the questionnaire provided invaluable feedback that will subsequently be used to improve *Zeeker Search Engine*.

Part IV

Implementation

Chapter 13

Implementation

In this chapter we introduce the developed applications and how these interact. The inner workings of the individual applications will not be explained in detail, only an overview of their basic capabilities and functionality will be presented. If the descriptions are not of interest to the reader, the *Zeeker Application section* can be skipped, whereas the chapter's summary gives a general overview of the *Zeeker Search Engine* work-flow.

13.1 Lemur and Indri

Implementing a search engine demands many different design considerations. It quickly became clear that implementing the entire system from scratch within the given time frame would be futile (although indeed desirable). There are far too many subtleties when working with data that can have ANY form. Furthermore, data needs to be stored and retrieved effectively and the many considerations regarding efficient storing etc. could be a thesis by itself. In order to concentrate on the problems we wanted to address, a framework was needed that could handle some of the more trivial tasks in the search engine.

Two frameworks were seriously considered: *The Apache Lucene project*[1], written in Java and ported to C#, and *The Lemur Project*[2], written in C++. Both frameworks looked promising and even though the Microsoft C# world is more familiar, Lemur was the framework of choice. Lemur is a large toolkit with some great qualities such as being written in C++ (which is faster than C#) and being developed as a joint venture between Carnegie Mellon University and University of Massachusetts.

[1]http://lucene.apache.org
[2]http://www.lemurproject.org

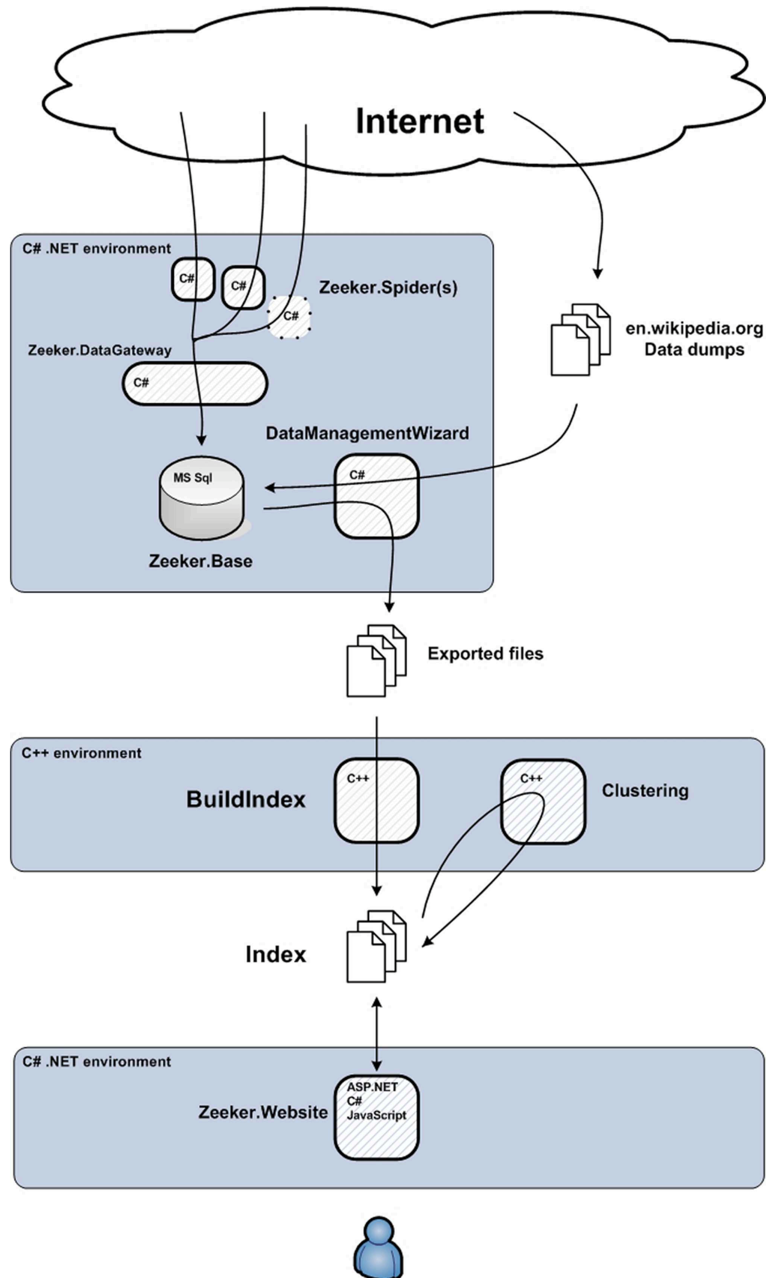

Figure 13.1: Zeeker Search Engine data and application flow

As mentioned, Lemur is written in C++ and is basically a framework for creating search engines. Lemur also adds an already implemented search engine called *Indri* to its portfolio. Since the goal was to develop the *Zeeker Search Engine*, the Lemur framework had to be extended and the Indri search engine was disregarded.

Lemur offers functionality for storing data efficiently in files on disk with a well designed and implemented work flow. Many of the already implemented building blocks in Lemur were extended with additional features and new building blocks were added to the framework. The changes turned out to be considerable and therefore the extended framework was renamed *LemurPlus*. The details of the extensions and improvements to the framework will not be discussed further but some of the improvements are mentioned in this chapter's summary.

13.2 Zeeker Applications

Here follows a brief description of various applications that together construct the *Zeeker Search Engine*. The intent is not to describe the inner workings of these applications, but merely give a concise overview of their capabilities and responsibilities.

The data flow and how the applications fit into the flow can be seen in figure 13.1.

13.2.1 *Zeeker.Spider* and *Zeeker.DataGateway*

It has already been pointed out that the web-spider, *Zeeker.Spider* and the web-service, *Zeeker.DataGateway* will not be discussed further as they were developed separately in a different project. Nevertheless, they have to be mentioned as they play a big part in the data flow of the *Zeeker Search Engine*.

The web spider is a basic, yet highly configurable and distributed web spider written in C#. Distributed refers to the fact that many web spiders can be run concurrently on several computers[3] at once thus increasing the download rate accordingly. The web spiders send the downloaded web resources to the *Zeeker.DataGateway* which in turn stores the data in *Zeeker.Base*.

13.2.2 *Zeeker.Base* (database)

Zeeker.Base is a MS SQL Server 2005 holding all the downloaded Html files and all data from Wikipedia and Wiktionary data dumps. The MS SQL database was chosen as it gives complete control of the downloaded data. When testing, it is very important to have full data control yet when migrating from test environment to a live environment, the importance of the database will be diminished as the downloaded data is moved to the file system to reduce overhead in data handling. The database creates an unnecessary overhead by storing a lot of meta information on the downloaded data and as a result, data sizes grow

[3] As many as the webservice can handle

at a fast rate in the database.

Zeeker.Base consists of more than 6 databases, more or less intertwined. The databases store data from:

- http://en.wikipedia.org – (enwiki)
- http://en.wiktionary.org – (enwiktionary)
- http://www.musicmoz.com – (musicmoz)
- Downloaded Html resources – (musicbase)
- Logging and exception handling – (datamanagement)
- Tests – (test)

These databases consist of over 50 tables, countless stored procedures and take up more than 45GB on disk. All this data is very useful when finding Wikipedia articles from specific categories, creating test sets, computing clusters etc. Wikipedia articles, downloaded Html and more can be joined in just about any way necessary, which gives many possibilities data wise.

13.2.3 *DataManagementWizard*

The *DataManagementWizard* is written in C# and helps manage data, control data flow, size and form in the overall process. *DataManagementWizard* is responsible for importing data from data dumps from wikipedia.org, wiktionary.org and in parts from musicmoz.com. *DataManagementWizard* is also responsible for exporting the data into appropriate formats.

The different import and export functionalities are described below.

Data import

The *DataManagementWizard* application can import any Wikipedia or Wiktionary data dump and store it in *Zeeker.Base*. This is done by more than 15 parsers that remove errors in the data dumps (which occur surprisingly often), normalize data by removing control characters and the like, calculate Wikipedia Ids between dumps (cross-reference) etc. The primary import capabilities are listed below.

- Import any Wikipedia data dumps
- Import any Wiktionary data dumps
- Import an Acronym list
- Import MusicMoz[4] band dump
- Import MusicMoz styles dump

[4] http://www.musicmoz.com/

Data export

Even though correct import is desirable and needed, the applications export capabilities is the primary purpose of the application. The exporting functions create the data sets that the rest of the indexing process depends on. The *DataManagementWizard* is capable of exporting Wikipedia articles from specific categories, external links and downloaded Html files, calculating Wikipedia clusters, exporting test sets and more. The data can be exported in formats such as text, Html and TREC. At the moment, the *DataManagementWizard* has over 30 export methods. The primary export capabilities are listed below.

- Export Wikipedia data (such as titles, category links, external links etc.)
- Export Wiktionary data (such as titles, acronyms, external links etc.)
- Wikipedia articles (from chosen category and other criteria)
- External links (from chosen category)
- Dump all downloaded Html web resources.
- Clusters (from chosen category)
- Test sets
- A few selected MusicMoz data dumps

13.2.4 *BuildIndex*

The *BuildIndex* application is written in C++ and is the backbone of the system. The application is responsible for creating every element of the indexing process, such as:

- Creating the appropriate parser (based on the files parsed)
- Creating the POS tagger and POS filter
- Creating the stopper
- Creating the stemmer
- Indexing the data file
  - Store tokens
  - Store token positions in file
  - Store tag extents (which tokens are in which tags)
  - Store other meta information on token (e.g. POS tag)

Needless to say, this application is very complex and it has several parameters that control how indexing is done. The indexing process is basically a set of building blocks on top of each other and the parameters simply inform the application of which building blocks are to be used when indexing. If all possible building blocks are chosen, the indexing chain will look like the one shown in figure 3.2 on page 38.

The work flow of the application is very flexible and makes it easy to create an index without stop-word removal or stemming of tokens, if needed.

Two parsers have been created, a Wikipedia parser and a Html parser. Most of the information on music on the Internet is available as Html (or as flash) which makes the creation of additional parsers a second priority. A new POS tagger has also been implemented and added to the Lemur framework as well as a POS tag filter. The POS tagger tags a term and the filter determines if the term is worth storing or not - based on the tag.

The indexer itself has been extended with additional meta information such as tags (both Wikipedia tags and Html tags), the position of tags, which terms are included in a tag and more.

Various parameters are supported by the indexer that determine how the indexing should be done. At present the following possibilities are available for:

- Include/exclude stop-words
- Include/exclude stemming
- Include/exclude POS tagging and filtering
- Index Wikipedia articles
- Index Html resources (from several files)
- Setting maximum memory usage (RAM)
- Creating a new or adding to an existing index

13.2.5 *Clustering*

The *Clustering* application is written in C++ and is responsible for clustering the external Html files and place them into the predefined Wikipedia categories. The application uses a file dump from the *DataManagementWizard* to get the Wikipedia clusters. Each Html document is measured against every cluster centroid and if the similarity is above a given threshold (here 0.25), the document is added to the cluster. The output from the application is a binary file with cluster representations for the Wikipedia clusters as well as the external Html documents. This binary file is then loaded into memory on the website and used to find which categories to display to the user.

The application can also be used to calculate the cluster qualities used for testing purposes. In fact, the application was used to calculate baseline and cluster quality scores for the Spherical k-means clusters and the Wikipedia categories shown in chapter 9.

13.2.6 *Zeeker.Website*

The *Zeeker.Website* can be found at:

http://studweb1.imm.dtu.dk

The web-interface is the front-end of the search engine on the Internet. The website is created using ASP.NET running on an Internet Information Service (IIS 6.0). Webpages and code-behind logic is implemented using C#, Html, Css and JavaScript. To search the index, the webpages use a C# wrapper giving access to the Lemur C++ search functions.

The more detailed functionality of the *Zeeker.Website* is discussed in chapter 10 and a User Guide can be found in appendix A.

13.2.7 Test programs

Various test applications have been implemented for different purposes such as monitoring data changes from other applications in the data flow, testing the implemented code, basic monitoring and administration of data etc. These applications have been written in C#, C++, MATLAB and Perl.

13.3 Summary

The complexity of creating a search engine is enormous and very time consuming. Even though data was restricted to Html web resources and Wikipedia articles only, the amount of data is still measured in tens of GB. Working on such large data sets demands a lot of the programming as the running time suddenly has a major importance on the project. As mentioned at the beginning of this chapter, The Lemur Project was chosen as an underlying framework and even when using Lemur, indexing several GB of data easily took several hours.

As in all complex software development projects, some of the encountered problems were not at all anticipated and some of the anticipated problems never came up. There was really no way of knowing how the applications would scale to the large amounts of data but as the index has grown, the scaling has not been a problem. All applications written in C++ have been tested thoroughly for memory leaks as these bugs tend to be very subtle and can strike at any time.

Good objective programming techniques have been used such as always testing on objects before using them etc. This approach has saved a lot of time when debugging as it is very hard to find the errors when several GB of data run through an application.

The large amount of data has been the primary concern. When applications work correctly on small data sets but suddenly crash on larger data sets, the problems often lie in other areas of the computer e.g. external disk drives, network connectivity, shortage of RAM etc. These problems are hard to deal with and even harder to find. However, the *BuildIndex* application which indexes more than 20 GB of Html and 2GB of Wikipedia articles is very stable.

The problems with finding and localizing errors have spawned several smaller test programs that monitor data flow, read and validate binary data to name a few. All of these test programs have been very helpful and will at a later time

be integrated into the different applications. Linux and MS Dos have many small applications that also have been used extensively to ease the programming burden. Here the choice of Lemur and C++ has made everything easier as the programming code is easily ported to other environments such as the Linux cluster available at the Institute of Informatics and Mathematical Modeling (IMM) at The Danish Technical University (DTU). Porting the code to the Linux cluster is ideal as the cluster has resources that can speed up indexing, clustering and various tests.

13.3.1 Data flow

The many applications created are shown in figure 13.1. The figure shows the overall flow of data from downloading from the Internet using *Zeeker.Spider* to the point where the data is available in the index and can be retrieved using *Zeeker.Website*.

Two primary data sources are used: the Internet and *http://en.wikipedia.org*. The data from the Internet is automatically downloaded by the *Zeeker.Spider* and stored in compressed form in *Zeeker.Base* with the help of a webservice called *Zeeker.DataGateway*. Wikipedia data dumps are downloaded manually and imported into the *Zeeker.Base* using *DataManagementWizard* application. This concludes the data collecting phase of the *Zeeker Search Engine*.

A MS SQL 2005 database is used as it has the possibility of extracting exactly the data needed when indexing and testing. *Zeeker.Base* and *DataManagementWizard* can calculate Wikipedia clusters (Wikipedia categories) and dump data in different formats.

All the different data dumps make it easier to build the parsers used in the *BuildIndex* application. No data is removed when it gets dumped from the database, instead, several thousand Html resources are accumulated and stored in one or more (very big) file(s). In this new format, usually a variant of TREC[5], meta data not available in the original downloaded data can also be added, such as an id pointing back to the database etc. *DataManagementWizard* supports over 30 different data dumps in various formats.

BuildIndex is the application responsible for indexing the data dumped by *DataManagementWizard*. BuildIndex is by far the most complex of all the applications making up the *Zeeker Search Engine*. The application creates parsers, POS tagger, stopper, stemmer and indexer. The BuildIndex data flow can be seen in figure 3.2. When all exported files have been processed by BuildIndex, a complete searchable index (not clustered) is available.

Html files are added to clusters using the application *Clustering*. The application calculates similarity, using cosine similarity measure, for each Html document and measures against the clusters. Html documents are added to the appropriate cluster if the similarity score is above a preset threshold. The application stores the clustering of the Wikipedia articles and the Html files in

[5]Text Retrieval Conference (TREC) http://trec.nist.gov/

a binary file which is loaded by the *Zeeker.Website*. When the index has been clustered, a searchable index with category filtering options is ready.

Zeeker.Website is a basic web interface created solely for the purpose of making the index public. The website searches the file based index and presents results as any other search engine on the web. Few but important search operators are implemented as described in chapter 10. The website is designed in a simple manner and should be intuitively easy to use.

The implemented applications and features have extended the Lemur framework. Features such as parsers, POS tagger, POS filter, the ability to store more data and meta data, a new clustering method, cluster quality calculation and much more have been added to the framework. There are not many objects or classes that have not been modified or even been completely rewritten.

Zeeker Search Engine is a very complex array of applications which together consist of more than 2600 files taking up more than 900 MB on disk. This is of course a very relative measure as some files are bigger than others, yet it gives a feeling of the complexity of *Zeeker Search Engine*.

Part V

Conclusion

Chapter 14

Future work

During the analysis and implementation of *Zeeker Search Engine*, many choices have been made as to how data is represented, indexed, filtered and retrieved. Building a search engine is more about making choices rather than an exact science. Even though the many choices have at times been a dilemma, they can also be seen as *parameters*, which later can be adjusted in order to improve the results. Some of the more dominant *parameters* available are: adjusting how vocabulary is reduced, changing the way documents are clustered and improving the retrieval method.

The choices made have revealed some minor issues and improvements that can be worked on to improve the search engine. This chapter summarizes these issues and improvements and proposed solutions are presented.

14.1 Known issues

First of all, the fairly conservative way of pruning the vocabulary has seemingly been too strict. Even though the idea was not to remove too many terms from the index, many vital terms seem to be missing regardless. Reading the literature on the subject (see introduction), researchers have removed large amounts of terms in indexes without reducing the quality of retrieval. We believe this is due to the indexes' vocabulary. For example, a vocabulary based on email correspondence will probably not be very diverse as people tend to use the same terms over and over again in emails and daily conversations. Therefore, most of the terms can be removed and only a few (distinct and important) terms left for indexing. The *Zeeker Search Engine* index is based on resources regarding musical groups, song titles, band names etc. Removing stop words from such resources could easily remove an entire title from a song or an entire band name. Pruning was found to be too aggressive as a search on *The Who* returned no results since both terms are stop words. Localizing the stop-word problem re-

vealed another minor, yet serious issue. Terms were found to be automatically down-cased *before* they were POS-tagged. This resulted in wrong POS categories for some terms, e.g. a band name like *The Who* became *the who* thus failing to identify the terms as nouns.

Cluster precision and cluster quality is another known issue. As described earlier, documents are added to the most similar clusters by measuring similarity between the document and cluster centroids using the Cosine Similarity Method. This method works to some extent, yet some web pages seem to be placed in too many clusters. The solution to this problem is not entirely clear. One solution could be to change the similarity threshold which would result in a more selective adding of documents to clusters although some documents might not be added to any cluster. Another solution is re-evaluate the entire clustering. Using cosine similarity works well with flat cluster structures but does not seem to work properly when the cluster structure is hierarchical like the Wikipedia clusters. The cosine similarity measure is more effective with smaller, more strict vocabularies whereas the larger and more general clusters in the Wikipedia hierarchy have many documents and thus a very general and rich vocabulary. Measuring document similarity against the centroids for these general vocabularies will never be very specific. The best solution for this problem is in no way clear and needs further analysis and testing in order to improve the cluster precision and quality.

14.2 Future features

Extensive work has been done on the programming code for the *Zeeker Search Engine*. Many of the choices made have been based on time issues. Some of the *nice to have* features have been put aside for the more important *must have* features. These nice to have features will hopefully give better and more precise results in the future as well as a more streamlined search engine.

One future feature is to utilize more of the available Wikipedia information. There is a lot of meta data that can be used when building the categories thus making it possible to display more detailed search results in info-box or similar grouping on the result page. For example, if the search engine knew that a category was a band, information about the band could be shown in a separate info-box on the web page along with the search results. This could also be applied for songs, artists, albums etc.

Storing and indexing various mark-ups such as titles, headings and anchor text in both Html and Wikipedia data is already implemented. This means that it is already known what tag a word on a web page is part of. Currently, all this meta information is not used in the retrieval. Html tags can say a lot about the information they are presenting e.g. if a word is part of a title tag, it is probably more important than a word in a table tag. This tag information could be used to weight the indexed terms differently thus retrieving more relevant documents.

Term positions are also stored in the index and will be used to generate small snippets of text in the result list as already known from other search engines.

These small extracts of a web page are very helpful when deciding on what results are most relevant.

Query expansion could also improve results dramatically - but as mentioned in previous chapters, it can also also worsen the results. Testing whether or not query expansion is helpful is a necessity before adding it to the retrieval process. Adding more query operators could give a more flexible retrieval by giving the users the ability to search for the already indexed meta data, e.g. searching only within title tags etc. A query operator giving the user a possibility to search for songs, bands, albums etc. would be extremely useful. If the meta information could reveal whether a web page contains information regarding a song, a band or an album, the results could be filtered in a much more precise manner. This is an issue that will be analyzed, tested and perhaps included in a future version of *Zeeker Search Engine*.

Not all future features have to do with retrieval. A lot of data is processed in order to build an index and the entire process of downloading data, calculating clusters, building the index etc. requires a lot of manual work. Therefore, automating this process as much as possible and at the same time make the applications more flexible would greatly reduce the amount of manual work needed to create the index and clusters. The added flexibility of the applications should also make the testing more automatic thus making it easier to see where improvements are due.

How the applications and choices perform when scaling the index to Tera byte sizes is still unknown and will be tested further. The Lemur framework has the ability to distribute smaller indexes over several servers which could be very useful in order to balance the search engine load. Using this support, smaller indexes, regarding specific topics, could be distributed over several servers such that each server only dealt with one topic but still was a part of a larger search engine network. A general search engine could submit queries to all the smaller search engines and present them to the user. The users could also access the individual smaller engines directly if needed. This approach would create an array of topic specific search engines making up a large scale general search engine.

14.3 Summary

Small issues have been located which we would like to improve. Some are of a more charismatic nature while others are more fundamental.

Of the charismatic nature we mention that the current web presentation does not deal with every possible exception. It is possible to get a (not so pretty) exception page if an unexpected error occurs. Furthermore, the result list sometimes includes duplicate entries and there are some minor variations in code execution depending on the web browser used. These are small problems that have no bearing on how the search engine performs - yet our vanity demands a fix.

The questionnaire and user tests revealed issues regarding the removal of

stop-words and also that the categories seem to be too broad, i.e. some documents are included in very many categories. Fortunately, the stop word removal issue is very easy to correct as stop words can simply be included when building the index. The documents belonging to many categories is a more complicated issue and will require further analysis and tests before it can be solved properly.

Zeeker Search Engine has many promising features and we have no doubt that it can be made even better by correcting the errors found and adding some of the future features described here such as using meta information in retrieval, implementing query expansion and adding more query operators.

CHAPTER 15

Conclusion

> Everything should be made as simple as possible, but no simpler.
>
> - Albert Einstein

During the past six months, we have analyzed many potential problems, read numerous articles and written countless lines of programming code. Boiling it all down to a decisive conclusion is very hard, yet we feel there are several points worth mentioning at this point.

Initial analysis and brainstorms made it clear that the work we were setting out to do could easily become very complex and thus difficult to finish within the given time frame. Therefore, everything had to be done as simply and effectively as possible and the above quote by Albert Einstein quickly became our mantra during the whole process. When faced with the choice between a simple or complex solution, we always chose the simplest solution possible yet no simpler. Throughout the thesis this dogma has been our guide and preferred way of solving problems.

In the problem description we stated that we wanted to try and build a topic-based search engine, i.e. using pre-calculated clusters to produce better search results. The pre-calculated clusters are a distinctive feature meant to differentiate *Zeeker Search Engine* from other search engines on the Internet. We realized that rethinking or improving current search engine techniques, with players like Google and Yahoo! out there, was a tremendous task. Furthermore, we had no illusions of creating a search engine better than these giants, nevertheless we wanted to build an engine with future potential. With a search engine using Wikipedia articles and a clustered index, the ambition was to create an engine that people found efficient, convenient and useful. Clustering search results has been done before in commercial search engines. However, these engines are usually meta search engines based on results from other search engines such as MSN Live Search, Yahoo!, Google etc. and have had very limited, if any, impact on

the Internet search market. Our vision was that using the information available from Wikipedia, the categories could be made strict and accurate enough, such that users would find them effective and reliable enough to provide a first-rate search result.

The first step toward a first-rate search is to retain full control of every step in the entire data flow. The data flow covers the downloading of data from the Internet, building the index, clustering the index, retrieve relevant information and present it to the user. With this kind of control over data, fine tuning every aspect of *Zeeker Search Engine* is possible in order to produce better results. Several individually implemented programs process the downloaded data into a format which is used to build the search engine's index. Downloaded web resources are clustered on top of Wikipedia's category structure using the Cosine Similarity Measure. The individual programs make it easier to improve and add additional information to the calculation of clusters and indexing.

With the topic-driven approach and ranking using the Cosine Similarity some of the weaknesses discussed in the introduction have been improved. As *Zeeker Search Engine* does not use link analysis or anchor text in any way, link farms and Google Bombs are not an issue. Cleverly added meta data is still a vulnerability since adding a list of keywords as hidden text on a web page will be treated like any other text by the search engine. However, if keywords for different categories are placed on a page, the page will be harder to cluster, thus possibly excluding it from any cluster. Hence, adding meta data in form of hidden keywords is still possible, but has to be done carefully in order for it to work. In order to eliminate the use of hidden text, the page's style sheet, script files and mark-up would have to be analyzed in order to exclude hidden keywords.

Although some of the initial problems we set out to solve have been solved, the process of creating *Zeeker Search Engine* has not always been smooth sailing. Several problems have been analyzed and most of them have been solved based on research literature on the subject. Despite our best efforts, some of the solutions we relied on have simply not been good enough. An example of such a problem is the handling of stop-words. The initial problem analysis showed that most articles read, supported the removal of stop-words (some even uncritically). We have however found that stop-word removal needs some analysis before the stop-words can safely be removed. Not all vocabularies can afford to remove stop-words without losing important semantic meaning. Working with a vocabulary regarding music, i.e. songs, bands and musicians, stop-words play a very important part as many song titles include several stop-words. In our opinion, stop-words should be included in the index if creating a general search engine (where hardware is not an issue). A full index is always preferable although stop-word removal should be analyzed for each scenario as it can be very useful. The vocabulary is the deciding factor in whether to include or remove the stop-words.

Clustering was another major problem which took some time and effort before a solution was found. The main problem with clustering is the dimensionality of data, i.e. the term-vectors. Many clustering algorithms have been

developed and used for text clustering some of which we studied and considered using. However, the problem with using these traditional clustering algorithms on our data set with 200.000 documents and over 1 million unique terms, the memory and CPU power needed to complete such a task would be enormous. Using the Spherical k-means algorithm to cluster "only" 10.000 documents took 40 hours and 2.83GB of RAM which we thought was excessive[1]. Therefore we had to come up with a computationally less expensive solution to this problem. Here we focused our attention toward the Wikipedia categories. All Wikipedia articles are categorized in a hierarchical structure which looked like a good solution to our clustering problem. Using smaller datasets, the quality of these clusters were measured against the cluster quality produced by the Spherical k-means algorithm. The Wikipedia cluster quality was found to be considerably better than the Spherical k-means clusters. Therefore, the Wikipedia clusters were used as basis clusters. This solution has worked quite well although some of the clusters seem to be of poor quality. As discussed in the previous chapter, using the Cosine Similarity Measure on a hierarchical overlapping clustering is perhaps not a good idea and will have to be reevaluated in a future version of *Zeeker Search Engine*.

Much of our efforts were focused on the index creation and especially on the document clustering. The retrieval part of the *Zeeker Search Engine* was from the beginning a low priority as we firmly believed that with a good index and clustering, retrieving the documents would not cause severe problems. Fortunately that assumption worked out well. User tests revealed that *Zeeker Search Engine* was fully capable of retrieving relevant documents from the index despite the stop-word and uppercase/lowercase errors discussed in chapter 14. In addition to fixing these errors, the retrieval also needs some added flexibility which is also discussed in chapter 14.

Adhering to Einstein's comment, the ranking of search results and found clusters is done in a very simplistic manner. Search results are ranked using Cosine Similarity scores whereas clusters are ranked using a sum of normalized Cosine Similarity scores (see equation 10.1). These simplistic ranking methods have produced good results but should perhaps be refined to produce even better results as the index scales to Tera-Byte sizes.

Concluding remarks

When we started this work, our main goal was to create a full-scale search engine with a clustered index using Wikipedia articles as a learning source. The search engine is now implemented in a general manner without being specially tuned or biased toward any topic. We have tried to make everything as simple as possible without cutting any corners. All elements in the search engine have been thoroughly analyzed and handled whereas nothing has been taken for granted. With an index consisting of 200.000 music related documents, the engine retrieves good results as the conducted user survey demonstrated. Users ranked the engine as being average to good in such categories as retrieval relevance,

[1] Although code optimizations might reduce these numbers

ease of use and performance. The engine also provides a filtering mechanism using the calculated clusters - a mechanism many users found useful.

The implementation process has taught us some valuable lessons. For instance when working with text data measured in GBs, great care must be taken in every line of the programming code as locating errors when processing such amounts of data is a very difficult and tiresome affair. Handling memory and reducing clock-cycles is crucial for the implementation's efficiency. We also used an incremental implementation strategy where we started with data processing and finished with the retrieval part. Nothing was started before the underlying elements were in place. We found this incremental process very useful when implementing the search engine from scratch as it helped us retain our perspective. Last but not least we learned that when working with natural language processing and information retrieval there are no absolutes - merely a matter of finding the best set of choices.

With all things considered, we are very satisfied with our overall results. Our efforts these past six months have resulted in a complete, topic-driven search engine where we have full control over each element and have many parameters that can be adjusted in order to fine-tune the results. Despite some minor errors, we feel that *Zeeker Search Engine* has great potential in the future - there is definitely more to come.

> You don't have to be first as long as you are the best
>
> \- *Zeeker Search Engine*

Part VI

Appendix

Appendix A

User Guide

Here we give a brief user guide of the *Zeeker Search Engine*.

The front-end of the search engine is a simple Web interface with one text box and a button. Queries are submitted by typing the query terms in the text box and pressing the button labeled "Zeek". This will transfer the user to the result page.

The result page contains the list of results, a list of categories found for the query, a text box and a button to submit new queries to the engine. The category links (shown to the left of the results) can be used to filter out any unwanted results thus reducing the result set.

A link to a page describing the search engine's query syntax can also be found on both the above mentioned pages.

The search engine can be found using the URL:

http://studweb1.imm.dtu.dk

A.1 Query syntax

The query syntax accepts three different search operators as well as a category filter in form of an operator. Here the syntax and usage of these operators is explained.

Search Operators

There are three different query operators available within the query syntax. These are: *Op:AND*, *Op:OR* and *Op:Exact*. Note that the operators are NOT

case-sensitive.

Op:AND

This operator is used for a Boolean AND query over all the terms.

Example: the query: *"Rolling Stones Op:AND"* will match both Rolling and Stones in the documents, but not necessarily in that order. A text like "...stones are rolling down the hill" would be a legal match for this type of query.

Op:OR

This operator is used for a Boolean OR query over all the terms.

Example: the query: *"Rolling Stones Op:OR"* will match any documents containing either of the terms or both.

Op:EXACT

This operator is used for an exact matching query. Only documents containing all the query terms in the exact order will be matched.

Example: the query: *"The Stones Op:EXACT"* would not match the text "...The Rolling Stones on tour in Europe...", but the text "... the stones are rolling down the hill...".

No operators

With no operators used, the search engine will match the query terms in the same order as they appear, though allowing a few terms between them.

Example: the query: *"The Stones"* would match the text "...The Rolling Stones on tour in Europe...".

If no results are found using the default configuration, the search engine will relax its initial query conditions and resubmit the query.

Category filter

The query syntax allows filtering by using predefined categories. When a query is submitted, some categories are shown to the left of the results which can be used as filters by clicking on them. However, the category filter can also be applied when the query is submitted. This is done using the **Category** operator. **Example:** the query: *"Green Day Op:EXACT Category:1213"* will return documents which match the query terms exactly as described above, but only documents within the category with ID 1213.

A.2 Screen-shots

Here are a couple of screen shots from the Web interface for the search engine. The first picture shows the front-end of the *Zeeker Search Engine*. The second screen shot shows the result page after the query *"green day Op:EXACT Category:1213"* is submitted. The categories found can be seen on the left side of the image under the label *Categories*.

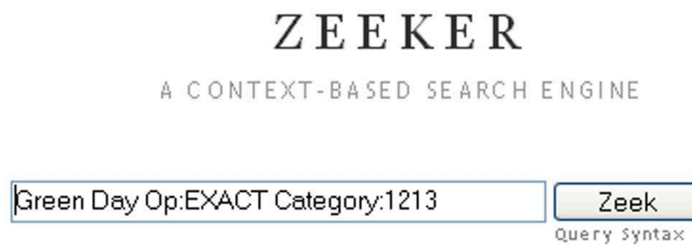

Figure A.1: Zeeker Search Engine front-end

ZEEKER
A CONTEXT-BASED SEARCH ENGINE

green day Op:EXACT Category:1213
Zeek
Query Syntax
Your search for **green day** in category **1213** gave **39** results

Categories:
Green Day
Green Day albums
Green River (band)
Brooke Fraser albums
Green Day songs
more clusters...

Search results:
1. Green Day
http://en.wikipedia.org/wiki/Green_Day
2. YouTube Broadcast Yourself
http://youtube.com/results?search_query=green
3. Living Green
http://www.daytondailynews.com/green/content/shared/green/
4. Green Day discography
http://en.wikipedia.org/wiki/Green_Day_discography
5. Baby Born Too Late from folklib
http://www.cdbaby.com/cd/tomgreen1/from/folklib
6. Rolling Stone Al Green Biography
http://www.rollingstone.com/artists/algreen/biography

Figure A.2: Zeeker Search Engine result page

Appendix B

Tests

B.1 Test indexes

Test index name	Documents	Terms	Unique terms	Size
Index full 1	10.000	21.986.858	439.251	218.618.442 bytes
Index full 2	20.000	42.900.993	669.801	408.716.538 bytes
Index full 3	30.000	57.966.761	820.996	545.597.989 bytes
Index full 4	40.000	72.791.232	970.141	679.959.869 bytes
Index full 5	50.000	83.147.031	1.059.147	773.057.305 bytes
Index full 6	60.000	91.769.505	1.130.588	849.595.692 bytes
Index full 7	70.000	97.761.261	1.160.028	900.232.367 bytes
Index full 8	80.000	104.818.520	1.199.333	960.230.599 bytes
Index full 9	90.000	112.268.383	1.251.858	1.025.530.981 bytes
Index full 10	100.000	124.340.972	1.352.497	1.134.314.931 bytes

Table B.1: Test indexes - full index

Test index name	Documents	Terms	Unique terms	Size
Index pos 1	10.000	14.796.227	421.490	193.076.729 bytes
Index pos 2	20.000	29.100.109	643.194	360.334.830 bytes
Index pos 3	30.000	39.296.047	788.386	480.282.588 bytes
Index pos 4	40.000	49.452.772	931.755	598.313.479 bytes
Index pos 5	50.000	56.714.905	1.017.308	680.328.401 bytes
Index pos 6	60.000	62.643.548	1.086.336	747.268.986 bytes
Index pos 7	70.000	66.676.093	1.114.693	790.710.844 bytes
Index pos 8	80.000	71.485.672	1.152.734	842.494.411 bytes
Index pos 9	90.000	76.521.488	1.203.316	899.048.137 bytes
Index pos 10	100.000	84.813.960	1.299.860	994.312.740 bytes

Table B.2: Test indexes - POS tagged

Test index name	Documents	Terms	Unique terms	Size
Index ss 1	10.000	14.052.498	376.552	182.733.812 bytes
Index ss 2	20.000	27.667.350	583.997	343.460.460 bytes
Index ss 3	30.000	37.369.091	719.862	458.870.342 bytes
Index ss 4	40.000	47.043.609	855.473	572.817.775 bytes
Index ss 5	50.000	53.987.641	936.677	652.245.302 bytes
Index ss 6	60.000	59.660.953	1.003.627	717.272.088 bytes
Index ss 7	70.000	63.503.354	1.031.724	759.616.007 bytes
Index ss 8	80.000	68.108.351	1.069.210	810.144.185 bytes
Index ss 9	90.000	72.910.153	1.118.145	865.026.795 bytes
Index ss 10	100.000	80.810.875	1.210.012	957.128.556 bytes

Table B.3: Test indexes - Stopped and stemmed

Test index name	Documents	Terms	Unique terms	Size
Index 1	10.000	13.637.170	362.849	179.379.310 bytes
Index 2	20.000	26.864.069	563.218	337.438.156 bytes
Index 3	30.000	36.275.764	694.303	450.899.062 bytes
Index 4	40.000	45.672.800	825.173	562.954.761 bytes
Index 5	50.000	52.429.810	903.519	641.110.741 bytes
Index 6	60.000	57.959.206	968.427	705.176.561 bytes
Index 7	70.000	61.721.191	995.556	746.994.911 bytes
Index 8	80.000	66.219.153	1.031.906	796.877.034 bytes
Index 9	90.000	70.896.397	1.079.210	850.932.403 bytes
Index 10	100.000	78.561.396	1.167.676	941.469.195 bytes

Table B.4: Test indexes - POS tagged, stopped and stemmed

B.2 Questionnaire

Hello,

We hereby invite you to participate in a survey to evaluate our search engine (Zeeker). This survey should not take much more than 15 minutes to complete.

In this survey we ask you to submit queries to the search engine and evaluate the results. We ask you to evaluate the usability of the engine, the performance and the results returned. Of course your participation is completely anonymous and voluntary.

Before continuing, we want to explain our main goals when designing the search engine. At present, the engine only deals with documents relating to music. It has the Wikipedia music articles in its index, along with documents downloaded from the Internet.

The main goal was to implement a search engine returning precise and most of all relevant documents. This is done using clustering, a technique to group relevant documents together. In the Web interface the clusters are presented as the Category list. For any given query, the search engine will return search results along with the categories it deems relevant for the query. These categories can be used as a filtering mechanism. When a category link is clicked, the search engine will filter its results and show only results from the clicked category. The goal of these categories is to easily find the relevant results without looking at many documents before finding the best ones. We ask you to bear the goals in mind when evaluating, i.e. have we reached our goals?

Your responses will be strictly confidential and only used for statistical purposes in our Masters Thesis. If you have any questions regarding this survey, you may contact us (Magnús and Søren) by email at maggi.sig@gmail.com

Thank you very much for your time and support. We hope you like it, because we do. Please start the survey now by clicking on the Continue button below.

Gender:
1. Male
2. Female

Age:
1. <20
2. 21-30
3. 31-40
4. 41-50
5. >50

What is your experience with search engines?
1. None (Never used a search engine)
2. Beginner (You know where and what Google is)
3. Intermediate (You are well familiar with Google)
4. Advanced (You know about search operators and such techniques)
5. Expert (You are familiar with PageRank, stop-words, stemming)

In the following questions, we ask you to find something we specify using our search engine. We will give you minimal information, which you can use in any way to find the results you find relevant. We encourage you to make use of both the category filter and the normal search results to find what you are looking for. Furthermore we have made three search operators available. An explanation of these can be found at http://studweb1.imm.dtu.dk/QuerySyntax.aspx

The questions require that you, in a seperate window, go to the search engine website at: http://studweb1.imm.dtu.dkNote that you will not be asked for the answer to any of the questions. The questions are just to see if you can find information we know exists in our index.

Should you by any chance know the answer to our questions, we ask you to still try and find the answer using the search engine.

Here we ask you to find a name of a single from an album called
Ten. The artist name and the name of the single will remain unknown.

Did you find what we asked for?
1. Yes
2. No
3. Dont know

How difficult was it to find?
(Skip this question if you did not find what we asked for)
1. Very easy
2. Easy
3. Normal
4. Difficult
5. Very difficult

Did you find the categories useful?
1. Very Useful
2. Useful
3. Somewhat Useful
4. Not Useful

We were looking for any song from the Ten album by Pearl Jam.
Did you find that?
1. Yes
2. No

Now we would like you to find the real name of the artist which uses the stage name The Edge. This artist has been a member of a very popular rock band for many years.

Did you find what we asked for?
1. Yes
2. No
3. Dont know

How difficult was it to find?
(Skip this question if you did not find what we asked for)
1. Very easy
2. Easy
3. Normal
4. Difficult
5. Very difficult

Did you find the categories useful?
1. Very Useful
2. Useful
3. Somewhat Useful
4. Not Useful

We were looking for the name David Howell Evans, a member of the band U2. Did you find that?
1. Yes
2. No

```
Finally we ask you to find the name of the band behind the
song Lord of the Boards. The band was active between 1994-2005.

Did you find what we asked for?
1. Yes
2. No
3. Dont know

How difficult was it to find?
(Skip this question if you did not find what we asked for)
1. Very easy
2. Easy
3. Normal
4. Difficult
5. Very difficult

Did you find the categories useful?
1. Very Useful
2. Useful
3. Somewhat Useful
4. Not Useful

We were looking for the band Guano Apes
Did you find that?
1. Yes
2. No
```

```
Now we ask you to find things on your own using the search
engine.  First we would like you to search for an artist or
a band of your choice, and then we ask you to search for a
song title or an album title. Dont be shy, put the engine
to the test.

Subsequently we ask you to evaluate the general results from
 these queries. The questions are based on how many relevant
 results you got. We also ask that you evaluate the usability
 of the engine, i.e. was it easy, hard or complex to use.

How relevant were the results to your queries?
1. Very Relevant
2. Quite Relevant
3. Relevant
4. Somewhat Relevant
5. Not Relevant at all
```

You are almost done. Just a few more questions regarding your overall satisfaction with the search engine.

How would you rate our search engines overall performance, i.e. taking speed, ease-of-use and relevance of results into account?
1. Very Good
2. Good
3. Avarage
4. Bad
5. Very Bad

How likely are you to use this kind of search engine again?
1. Very Likely
2. Likely
3. Unlikely
4. Very Unlikely
5. Dont know

Finally we ask for your comments, good or bad regarding our search engine. (Optional)

If you have any comments regarding this survey, feel free (Optional)

Appendix C

POS Tagging

POS tags	**10.000 articles**	**50.000 articles**	**100.000 articles**	**Average**
BE,BEDR,BEDZ,BEG, BEM,BEN,BER,BEZ	6.59%	6.65%	7.61%	6.95%
CC, CS	5.91%	5.81%	6.03%	5.92%
CD, OD	7.94%	8.76%	11.63%	9.44%
DT,PDT	9.80%	9.76%	10.32%	9.96%
FW	2.29%	2.47%	2.53%	2.43%
HV,HVD,HVG,HVN, HVZ	0.58%	0.57%	0.67%	0.61%
IN	5.49%	5.51%	5.89%	5.63%
JJ,JJR,JJS	12.97%	12.95%	13.25%	13.06%
NN,NNS,NP,NPS	33.58%	32.55%	27.36%	31.16%
PN,PP,PP$,PPX, WP,WP$	1.56%	1.57%	1.56%	1.56%
RB,RBR,RBS,RP	1.67%	1.61%	1.51%	1.60%
VB,VBD,VBG,VBN, VBZ	3.58%	3.56%	3.43%	3.52%
Remaining tags	2.80%	2.73%	2.75%	2.76%
???	5.24%	5.50%	5.46%	5.40%

Table C.1: Test on distribution of the tags in the Brown/Penn-style tagset

C.1 POS tagset

POS	Description
BE	be
BEDR	were
BEDZ	was
BEG	being
BEM	am
BEN	been
BER	are
BEZ	is
CC	conjunction, coordinating (and)
CD	number, cardinal (four)
CS	conjunction, subordinating (until)
DO	do
DOD	did
DOG	doing
DON	done
DOZ	does
DT	determiner, general (a, the, this, that)
EX	existential there
FW	foreign word (ante, de)
HV	have
HVD	had (past tense)
HVG	having
HVN	had (past participle)
HVZ	has
IN	preposition (on, of)
JJ	adjective, general (near)
JJR	adjective, comparative (nearer)
JJS	adjective, superlative (nearest)
MD	modal auxiliary (might, will)
NN	noun, common singular (action)
NNS	noun, common plural (actions)
NP	noun, proper singular (Thailand)
NPS	noun, proper plural (Americas, Atwells)
OD	number, ordinal (fourth)
PDT	determiner, pre- (all, both, half)
PN	pronoun, indefinite (anyone, nothing)

Table C.2: Variant of the Brown/Penn-style tagset - Part I

POS	**Description**
POS	possessive particle (’, ’s)
PP	pronoun, personal (I, he)
PP$	pronoun, possessive (my, his)
PPX	pronoun, reflexive (myself, himself)
RB	adverb, general (chronically, deep)
RBR	adverb, comparative (easier, sooner)
RBS	adverb, superlative (easiest, soonest)
RP	adverbial particle (back, up)
SYM	symbol or formula (US$500, R300)
TO	infinitive marker (to)
UH	interjection (aah, oh, yes, no)
VB	verb, base (believe)
VBD	verb, past tense (believed)
VBG	verb, -ing (believing)
VBN	verb, past participle (believed)
VBZ	verb, -s (believes)
WDT	det, wh- (what, which, whatever, whichever)
WP	pronoun, wh- (who, that)
WP$	pronoun, possessive wh- (whose)
WRB	adv, wh- (how, when, where, why)
XNOT	negative marker (not, n’t)
”	quotation mark
(	(
,	,
.	.
:	:
?	?
!	!
’	apostrophe
)	)
-	-
...	...
;	;
???	unclassified

Table C.3: Variant of the Brown/Penn-style tagset - Part II

Appendix D

Stopwords

Stopwords	(a-h)			
a	about	above	according	across
after	afterwards	again	against	albeit
all	almost	alone	along	already
also	although	always	am	among
amongst	an	and	another	any
anybody	anyhow	anyone	anything	anyway
anywhere	apart	are	around	as
at	av	be	became	because
become	becomes	becoming	been	before
beforehand	behind	being	below	beside
besides	between	beyond	both	but
by	can	cannot	canst	certain
cf	choose	contrariwise	cos	could
cu	day	do	does	doesn't
doing	dost	doth	double	down
dual	during	each	either	else
elsewhere	enough	et	etc	even
ever	every	everybody	everyone	everything
everywhere	except	excepted	excepting	exception
exclude	excluding	exclusive	far	farther
farthest	few	ff	first	for
formerly	forth	forward	from	front
further	furthermore	furthest	get	go
had	halves	hardly	has	hast
hath	have	he	hence	henceforth
her	here	hereabouts	hereafter	hereby
herein	hereto	hereupon	hers	herself
him	himself	hindmost	his	hither
hitherto	how	however	howsoever	

Table D.1: Stop word list (a-h)

Stopwords	(i-y)			
ie	if	in	inasmuch	inc
include	included	including	indeed	indoors
inside	insomuch	instead	into	inward
inwards	is	it	its	itself
just	kind	kg	km	last
latter	latterly	less	lest	let
like	little	ltd	many	may
maybe	me	meantime	meanwhile	might
moreover	most	mostly	more	mr
mrs	ms	much	must	my
myself	namely	need	neither	never
nevertheless	next	no	nobody	none
nonetheless	noone	nope	nor	not
nothing	notwithstanding	now	nowadays	nowhere
of	off	often	ok	on
once	one	only	onto	or
other	others	otherwise	ought	our
ours	ourselves	out	outside	over
own	per	perhaps	plenty	provide
quite	rather	really	round	
said	sake	same	sang	save
saw	see	seeing	seem	seemed
seeming	seems	seen	seldom	selves
sent	several	shalt	she	should
shown	sideways	since	slept	slew
slung	slunk	smote	so	some
somebody	somehow	someone	something	sometime
sometimes	somewhat	somewhere	spake	spat
spoke	spoken	sprang	sprung	stave
staves	still	such	supposing	than
that	the	thee	their	them
themselves	then	thence	thenceforth	there
thereabout	thereabouts	thereafter	thereby	therefore
therein	thereof	thereon	thereto	thereupon
these	they	this	those	thou
though	thrice	through	throughout	thru
thus	thy	thyself	till	to
together	too	toward	towards	ugh
unable	under	underneath	unless	unlike
until	up	upon	upward	upwards
us	use	used	using	very
via	vs	want	was	we
week	well	were	what	whatever
whatsoever	when	whence	whenever	whensoever
where	whereabouts	whereafter	whereas	whereat
whereby	wherefore	wherefrom	wherein	whereinto
whereof	whereon	wheresoever	whereto	whereunto
whereupon	wherever	wherewith	whether	whew
which	whichever	whichsoever	while	whilst
whither	who	whoa	whoever	whole
whom	whomever	whomsoever	whose	whosoever
why	will	wilt	with	within
without	worse	worst	would	wow
ye	yet	year	yippee	you
your	yours	yourself	yourselves	

Table D.2: Stop word list (i-y)

Index

L^2 Norm, 51
Spherical k-means, 51
    Cluster Quality, 53

Apache Lucene, 103
Applications
    DataManagementWizard, 106
    Zeeker.Base, 105
    Zeeker.Spider, 105
    Zeeker.DataGateway, 105
    Zeeker.Website, 108
    BuildIndex, 107
    Clustering, 108

Centroid, 53
Clustering, 12, 41
    Spherical k-means, 51
    Algorithm similarities, 67
    Algorithms, 13
    FTC, 61
    NMF, 57
    Non-Overlapping, 46
    Overlapping, 46
    Tests, 71
Cosine Similarity Measure, 17, 42

Document Processor, 4

F-measure, 89
FTC, 61

Google, 1
    Bomb, 2
    PageRank, 1, 17
    Wash, 2

Hierarchical clustering, 46

Index reduction, 33
Indexing, 9
    Full Inverted file, 11
    Inverted file, 9

Lemur, 103
    Indri, 103
Linearization, 8
Link Analysis, 16

Machine Learning, 47
    Supervised, 47
    Unsupervised, 47

NMF, 57
    Updating rules, 59

Open Directory Project, 28

Part-of-Speech, 6
    Tags, 139
Partitional clustering, 46
Precision, 89

Query
    Analysis, 15
    Expansion, 14
    Processing, 13
Query Expansion, 84
    Automatic, 84
    Manual, 84
    User-Assisted, 84
Query Operators, 82
    Category, 83
    Search, 82
Query Processing, 81
Query syntax, 125

Ranking, 15, 85
  Link Analysis, 16
    HITS, 17
    PageRank, 17
  Relevance Feedback, 18
  Vector Space Model, 17
Recall, 89
Relevance Feedback, 18
  Explicit, 18
  Implicit, 18
Retrieval, 81
  Evaluation
    F-measure, 89
    Precision, 89
    Recall, 89
    User Feedback, 90
  Method, 85
  Query Expansion, 84
  Query Operators, 82
  Query Processing, 81
  Ranking, 85
  Tests, 93
  Vocabulary pruning, 81

Search Engine Optimization, 1
Semantics
  Compositional, 4
  Lexical, 4
Spectral clustering, 46
Stemming, 7
  Lovins, 7
  Porter, 7
Stop Words, 6, 143

Term weighting, 7, 43
  Tf x Idf, 8, 43
Term-Document Matrix, 17, 41
Test sets, 29

User Feedback, 90
User Guide, 125

Vector Space Model, 17, 41
Vector space model, 41
Vocabulary Pruning, 81
Vocabulary pruning, 33

Wikipedia, 25
WordNet, 6

Bibliography

[1] Exact and approximation algorithms for clustering. *Algorithmica*, 33(2):201–26, 2002.

[2] R. Ali and M.M.S. Beg. A framework for evaluating web search systems. *WSEAS Transactions on Systems*, 6(2):257–264, 2007.

[3] Vamshi Ambati and Rohini U. Improving re-ranking of search results using collaborative filtering. *Lecture Notes in Computer Science*, 4182:205–216, 2006.

[4] Ricarco Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[5] Florian Beil, Martin Ester, and Xiaowei Xu. Frequent term-based text clustering. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 436–442, 2002.

[6] S.M. Beitzel, E.C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. *Proceedings of Sheffield SIGIR 2004. The Twenty-Seventh Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 321–8, 2004.

[7] M.W. Berry, V. Pauca, R.J. Plemmons, M. Browne, and A.N. Langville. Algorithms and applications for the approximate nonnegative matrix factorization. *Computational Statistics and Data Analysis (Elsevier - to appear)*, 2007.

[8] J. Bhogal, A. Macfarlane, and P. Smith. A review of ontology based query expansion. *Information Processing and Management*, 43(4):866–886, 2007.

[9] S. Birch. Statistical text modelling - towards modelling of matching problems. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2003.

[10] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[11] F. Can, R. Nuray, and A.B. Sevdik. Automatic performance evaluation of web search engines. *Information Processing and Management*, 40(3):495–514, 2004.

[12] S. Cucerzan. Large-scale named entity disambiguation based on Wikipedia data. In *Proceedings of EMNLP-CoNLL 2007*, pages 708–716, 2007.

[13] M. de Buenaga Rodriguez, J. M. Gomez Hidalgo, and B. Diaz Agudo. Using WordNet to Complement Training Information in Text Categorization. In *eprint arXiv:cmp-lg/9709007*, pages 9007–+, September 1997.

[14] Franca Debole and Fabrizio Sebastiani. Supervised term weighting for automated text categorization. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 784–788, New York, NY, USA, 2003. ACM Press.

[15] Inderjit S. Dhillon and Dharmendra S. Modha. Concept decompositions for large sparse text data using clustering. *Machine Learning*, 42(1/2):143–175, 2001.

[16] Charles Elkan. Using the triangle inequality to accelerate k-means. *Proceedings, Twentieth International Conference on Machine Learning*, 1:147–153, 2003.

[17] Christopher Fox. A stop list for general text. *SIGIR Forum*, 24(1-2):19–21, r 90.

[18] D. Hull and G. Grefenstette. A detailed analysis of english stemming algorithms. Technical report, Rank XEROX, 1996.

[19] R.R. Joshi and Y.A. Aslandogan. Concept-based web search using domain prediction and parallel query expansion. *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration (IEEE Cat. No.06EX1467)*, page 6 pp., 2007.

[20] S. Jung, J.L. Herlocker, and J. Webster. Click data as implicit relevance feedback in web search. *Information Processing and Management*, 43(3):791–807, 2007.

[21] A. Kehagias, V. Petridis, V. Kaburlasos, and P. Fragkou. A comparison of word- and sense-based text categorization using several classification algorithms, 2003.

[22] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[23] Amy N. Langville and Carl D. Meyer. *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2006.

[24] Daniel D. Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. In *NIPS*, pages 556–562, 2000.

[25] D.D. Lee and H.S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[26] Shih-Hao Li and P.B. Danzig. Boolean similarity measures for resource discovery. *IEEE Transactions on Knowledge and Data Engineering*, 9(6):863–876, 1997.

[27] Yanjun Li and Soon M. Chung. Parallel bisecting k-means with prediction clustering algorithm. *J. Supercomput.*, 39(1):19–37, 2007.

[28] Janet Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31, 1968.

[29] James B. MacQueen. Some methods for classification and analysis of multivariate observations. 1966.

[30] Rasmus Elsborg Madsen, Sigurdur Sigurdsson, Lars Kai Hansen, and Jan Larsen. Pruning the vocabulary for better context recognition.

[31] M. Porter. An algorithm for suffix stripping. 14(3):130–137, 1980.

[32] Yonggang Qiu and H.P. Frei. Concept based query expansion. *Proceedings of the Annual International ACM SIGIR Conference on Research and Development in Infofmation Retrieval*, pages 160–169, 1993.

[33] G. Salton and M.J. McGill. Introduction to modern information retrieval. 1983.

[34] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. Technical report, Ithaca, NY, USA, 1987.

[35] Sergio M. Savaresi and Daniel L. Boley. A comparative analysis on the bisecting k-means and the pddp clustering algorithms. *Intelligent Data Analysis*, 8(4):345–362, 2004.

[36] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.

[37] F. Shahnaz, M.W. Berry, V. Pauca, and R.J. Plemmons. Document clustering using nonnegative matrix factorization. *Information Processing and Management*, 42(2):373–386, 2006.

[38] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques, 2000.

[39] L. Vaughan. New measurements for search engine evaluation proposed and tested. *Information Processing and Management*, 40(4):677–691, 2004.

[40] Olga Vechtomova and Ying Wang. A study of the effect of term proximity on query expansion. *Journal of Information Science*, 32(4):324–333, 2006.

[41] Vicente Vidal and Daniel Jiménez. Parallel implementation of information retrieval clustering models. *Lecture Notes in Computer Science*, 3402:129–141, 2005.

[42] W.B.Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, 1992.

[43] Stefan Wild, James Curry, and Anne Dougherty. Improving non-negative matrix factorizations through structured initialization. *Pattern Recognition*, 37(11):2217–2232, 2004.

[44] Catherine Wolf, John Karat, Sherman R. Alpert, and Sameer Patil. Thats what i was looking for: Comparing user-rated relevance with search engine rankings. *Lecture Notes in Computer Science*, 3585:117–129, 2005.

[45] Wei Xu, Xin Liu, and Yihong Gong. Document clustering based on non-negative matrix factorization. *SIGIR Forum (ACM Special Interest Group on Information Retrieval)*, (SPEC. ISS.):267–273, 2003.

[46] Torsten Zesch and Iryna Gurevych. Analysis of the Wikipedia category graph for NLP applications. In *Proceedings of the Second Workshop on TextGraphs: Graph-Based Algorithms for Natural Language Processing*, pages 1–8, Rochester, NY, USA, 2007. Association for Computational Linguistics.

[47] Jing Zhao, Mao Ye, and C.F. Yang. Document clustering based on non-negative sparse matrix factorization. *Lecture Notes in Computer Science*, 3611:557–563, 2005.

[48] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.