Ingo Feinerer

A text mining framework in R and its applications

Thesis

Dissertation

A Text Mining Framework in R and Its Applications

Zur Erlangung des akademischen Grades
eines Doktors der Sozial- und Wirtschaftswissenschaften
an der Wirtschaftsuniversität Wien

1. Beurteiler

Univ.-Prof. Dr. Kurt Hornik
Department of Statistics and Mathematics
Wirtschaftsuniversität Wien

2. Beurteiler

Univ.-Prof. Dr. Dr.h.c. Hans Robert Hansen
Institute for Management Information Systems
Wirtschaftsuniversität Wien

von

Ingo Feinerer

Wien, August 2008

Text Mining hat sich zu einer etablierten Disziplin sowohl in der Forschung als auch in der Industrie entwickelt. Dennoch fehlt es vielen Text Mining Umgebungen an einfacher Erweiterbarkeit und sie bieten nur wenig Unterstützung zur Interaktion mit statistischen Rechenumgebungen.

Aus dieser Motivation heraus wird in dieser Dissertation eine Text Mining Infrastruktur für die statistische Rechenumgebung R vorgestellt, die fortgeschrittene Methoden zur Manipulation von Textkorpora und deren Metadaten, zu Verarbeitungsschritten, zum Arbeiten mit Dokumenten, und zum Datenexport bietet. Es wird dargelegt, wie etablierte Text Mining Techniken mit der vorgestellten Infrastruktur durchgeführt werden können. Dazu wird auch gezeigt wie man diese Techniken einsetzt um gängige Aufgaben des Text Mining durchzuführen.

Der zweite Teil dieser Arbeit ist realistischen Anwendungen aus diversen Themenbereichen unter Nutzung der präsentierten Text Mining Infrastruktur gewidmet. Die erste Anwendung zeigt die Durchführung einer anspruchsvollen Analyse auf Daten einer E-Mail Verteilerliste. Die zweite Anwendung offenbart das Potential von Text Mining Methoden für den elektronischen Handel vorallem im sogenannten Business-to-Consumer Bereich. Das dritte Beispiel beschäftigt sich mit den Vorteilen von Text Mining auf juristischen Dokumenten. Zum Schluss wird eine Anwendung zur Identifikation von Autoren gezeigt, unter Verwendung der im englischsprachigen Raum recht bekannten Geschichten über den Zauberer von Oz.

Dissertation

A Text Mining Framework in R and Its Applications

Ingo Feinerer

Department of Statistics and Mathematics
Vienna University of Economics and Business Administration

Advisors: Kurt Hornik, Hans Robert Hansen, David Meyer

Vienna, August 2008

Text mining has become an established discipline both in research as in business intelligence. However, many existing text mining toolkits lack easy extensibility and provide only poor support for interacting with statistical computing environments.

Therefore we propose a text mining framework for the statistical computing environment R which provides intelligent methods for corpora handling, meta data management, preprocessing, operations on documents, and data export. We present how well established text mining techniques can be applied in our framework and show how common text mining tasks can be performed utilizing our infrastructure.

The second part in this thesis is dedicated to a set of realistic applications using our framework. The first application deals with the implementation of a sophisticated mailing list analysis, whereas the second example identifies the potential of text mining methods for business to consumer electronic commerce. The third application shows the benefits of text mining for law documents. Finally we present an application which deals with authorship attribution on the famous Wizard of Oz book series.

Acknowledgments

First of all, I would like to thank my family for their support during the last years. Only their continuous backing and appreciation of my time consuming academic activities made it possible to conduct the research grounding this thesis.

Next, I am very grateful to my advisor Kurt Hornik. His detailed feedback and suggestions always helped me to significantly improve the papers and software I wrote during the last years. As a side product of Kurt's scientific excellence I could learn a lot from him on writing papers, setting up a sound methodology for experiments, and on presenting results. Additionally I had the opportunity to learn a lot on developing high quality open source software due to Kurt's function as member in the core team of the statistical computing environment R, a major, actively developed, and very successful open source project.

Equally I appreciate all the assistance David Meyer gave me during the last years. He invested a lot of time reviewing and discussing my scientific work and came up with a lot of smart ideas. He helped me to analyze and describe topics from different perspectives, e.g., covering both technical and social aspects in an investigation. Additionally I must thank David for his administrative help, irrelevant whether I needed an account to a high-performance cluster or I needed a time slot for a presentation in a research seminar.

Finally I would like to thank my various scientific collaborators: Christian Buchta is among the first persons to contact when optimizing R software, Fridolin Wild has a broad linguistic background highly useful for text mining, and Alexandros Karatzoglou is an expert on string kernels and related methods.

Contents

1 Introduction **13**

2 Text Mining Overview **15**
2.1 What is Text Mining? . . . 15
2.1.1 Automated Processing of Text . . . 15
2.1.2 A Knowledge-Intensive Process . . . 16
2.1.3 Use of Text Collections to Discover New Facts . . . 17
2.1.4 Analyzing Unstructured Information . . . 17
2.1.5 Intelligent Text Processing . . . 18
2.2 Text Mining and Its Applications . . . 18
2.2.1 Text Clustering, Text Classification, and Information Retrieval . . . 19
2.2.2 Text Mining and Natural Language Processing . . . 20
2.2.3 Text Mining for Biomedical Research . . . 20
2.2.4 Text Mining in Stylometry . . . 22
2.2.5 Web Mining . . . 22
2.3 Text Mining Frameworks . . . 23

I Text Mining Framework **26**

3 Conceptual Process and Framework **27**
3.1 Framework Overview . . . 28

4 Data Structures and Algorithms **30**
4.1 Data Structures . . . 31
4.1.1 Text Document Collections . . . 32
4.1.2 Text Documents . . . 33
4.1.3 Text Repositories . . . 34
4.1.4 Term-Document Matrices . . . 34
4.1.5 Sources . . . 35
4.2 Algorithms . . . 36
4.3 Extensions . . . 45

5 Preprocessing **48**
5.1 Data Import . . . 48
5.2 Stemming . . . 49

5.3 Whitespace Elimination and Lower Case Conversion . . . 50
5.4 Stopword Removal . . . 50
5.5 Synonyms . . . 51
5.6 Part of Speech Tagging . . . 52

6 Basic Analysis Techniques **53**
6.1 Count-Based Evaluation . . . 53
6.2 Simple Text Clustering . . . 54
6.2.1 Hierarchical Clustering . . . 55
6.2.2 k-means Clustering . . . 56
6.3 Simple Text Classification . . . 57
6.3.1 k-nearest Neighbor Classification . . . 58
6.3.2 Support Vector Machine Classification . . . 59
6.4 Text Clustering with String Kernels . . . 60

II Applications **63**

7 Analysis of the R-devel 2006 mailing list **64**
7.1 Data . . . 64
7.2 Finding Most Active Authors, Threads, and Cliques . . . 65
7.3 Identifying Relevant Terms . . . 67

8 Text Mining for B2C E-Commerce **71**
8.1 Product . . . 72
8.1.1 Preparation . . . 72
8.1.2 Frequent Terms . . . 73
8.1.3 Associations . . . 75
8.1.4 Groups . . . 78
8.1.5 Non-Explicit Grouping . . . 80
8.2 Promotion . . . 81
8.2.1 Competitive Products . . . 81
8.2.2 Churn Analysis . . . 82

9 Law Mining **84**
9.1 Introduction . . . 84
9.2 Administrative Supreme Court Jurisdictions . . . 84
9.2.1 Data . . . 84
9.2.2 Data Preparation . . . 85
9.3 Investigations . . . 85
9.3.1 Grouping the Jurisdiction Documents into Tax Classes . . . 85
9.3.2 Classification of Jurisdictions According to Federal Fiscal Code Regulations . . . 87
9.3.3 Deriving the Senate Size . . . 88

9.4 Summary . . . 90

10 Wizard of Oz Stylometry **91**
10.1 Data . . . 91
10.2 Extracting Frequent Terms . . . 92
10.3 Principal Component Analysis . . . 93

11 Conclusion **96**

III Appendix **97**

A Framework Classes Minutiae **98**

B tm Manual **103**
acq . . . 103
appendElem-methods . . . 104
appendMeta-methods . . . 105
asPlain-methods . . . 106
c-methods . . . 106
colnames-methods . . . 107
stemCompletion . . . 108
convertMboxEml . . . 108
convertRCV1Plain . . . 109
convertReut21578XMLPlain . . . 110
crude . . . 111
CSVSource-class . . . 112
CSVSource . . . 112
Dictionary-class . . . 113
Dictionary . . . 114
dimnames-methods . . . 114
dim-methods . . . 115
DirSource-class . . . 115
DirSource . . . 116
dissimilarity-methods . . . 117
DublinCore-methods . . . 117
eoi-methods . . . 118
findAssocs-methods . . . 118
findFreqTerms-methods . . . 119
FunctionGenerator-class . . . 119
FunctionGenerator . . . 120
getElem-methods . . . 121
getFilters . . . 121
getReaders . . . 122

getSources . . . 122
getTransformations . . . 123
GmaneSource-class . . . 123
GmaneSource . . . 124
%IN%-methods . . . 125
inspect-methods . . . 125
length-methods . . . 126
loadDoc-methods . . . 126
makeChunks . . . 127
materialize . . . 128
MetaDataNode-class . . . 129
meta-methods . . . 129
ncol-methods . . . 130
NewsgroupDocument-class . . . 131
nrow-methods . . . 132
PlainTextDocument-class . . . 132
plot . . . 133
preprocessReut21578XML . . . 134
prescindMeta-methods . . . 135
RCV1Document-class . . . 136
readDOC . . . 136
readGmane . . . 137
readHTML . . . 138
readNewsgroup . . . 139
readPDF . . . 141
readPlain . . . 142
readRCV1 . . . 143
readReut21578XML . . . 144
removeCitation-methods . . . 145
removeMeta-methods . . . 146
removeMultipart-methods . . . 146
removeNumbers-methods . . . 147
removePunctuation-methods . . . 147
removeSignature-methods . . . 148
removeSparseTerms-methods . . . 149
removeWords-methods . . . 149
replacePatterns-methods . . . 150
Reuters21578Document-class . . . 150
ReutersSource-class . . . 151
ReutersSource . . . 151
rownames-methods . . . 152
searchFullText-methods . . . 153
sFilter . . . 153
show-methods . . . 154

Source-class . . . 155
stemDoc-methods . . . 155
stepNext-methods . . . 156
stopwords . . . 156
stripWhitespace-methods . . . 157
StructuredTextDocument-class . . . 157
[-methods . . . 158
summary-methods . . . 159
TermDocMatrix-class . . . 159
TermDocMatrix . . . 160
termFreq . . . 161
Corpus-class . . . 162
Corpus . . . 163
TextDocument-class . . . 165
TextRepository-class . . . 166
TextRepository . . . 167
tmFilter-methods . . . 168
tmIndex-methods . . . 168
tmIntersect-methods . . . 169
tmMap-methods . . . 170
tmTolower-methods . . . 171
tmUpdate-methods . . . 171
VectorSource-class . . . 172
VectorSource . . . 173
weightBin . . . 173
WeightFunction-class . . . 174
WeightFunction . . . 175
weightLogical . . . 175
weightTfIdf . . . 176
weightTf . . . 177
writeCorpus-methods . . . 177
XMLTextDocument-class . . . 178

List of Figures

2.1 A process model for text mining. 16

4.1 Conceptual layers and packages. 30
4.2 UML class diagram of the **tm** package. 31
4.3 UML class diagram for `Source`s. 36
4.4 Concatenation of two text document collections with `c()`. 39
4.5 Generic transform and filter operations on a text document collection. . . 43

6.1 Visualization of the correlations within a term-document matrix. 55
6.2 Dendrogram for hierarchical clustering. The labels show the original group names. 57

8.1 Visualization of the correlations for a selection of terms within the term-document matrix `cleanTDM`. 77

9.1 Plot of the contingency table between the keyword based clustering results and the expert rating. 87
9.2 Agreement plot of the contingency table between the senate size reported by text mining heuristics and the senate size reported by humans. 89

10.1 Principal component plot for five Oz books using 500 line chunks. 94
10.2 Principal component plot for five Oz books using 100 line chunks. 94

List of Tables

2.1 Overview of text mining products and available features. A feature is marked as implemented (denoted as ✓) if the official feature description of each product explicitly lists it. 24

4.1 Available readers in the **tm** package. 33
4.2 Transformations shipped with **tm**. 44

9.1 Rand index and Rand index corrected for agreement by chance of the contingency tables between k-means results, for $k \in \{3, 4, 5, 6\}$, and expert ratings for *tf* and *tf-idf* weightings. 86
9.2 Rand index and Rand index corrected for agreement by chance of the contingency tables between SVM classification results and expert ratings for documents under federal fiscal code regulations. 88
9.3 Number of jurisdictions ordered by senate size obtained by fully automated text mining heuristics. The percentage is compared to the percentage identified by humans. 88

1 Introduction

During the last decade text mining has become a widely used discipline utilizing statistical and machine learning methods. It has gained big interest both in academic research as in business intelligence applications. There is an enormous amount of textual data available in machine readable format which can be easily accessed via the Internet or databases. This ranges from scientific articles, abstracts and books to memos, letters, online forums, mailing lists, blogs, and other communication media delivering sensible information.

Text mining is a highly interdisciplinary research field utilizing techniques from computer science, linguistics, and statistics. For the latter R (R Development Core Team, 2008) is one of the leading and most versatile computing environments offering a broad range of statistical methods. However, until recently, R has lacked an explicit framework for text mining purposes. We tackle this situation by presenting a text mining infrastructure for R. This allows R users to work efficiently with texts and corresponding meta data and transform the texts into structured representations where existing R methods can be applied, e.g., for clustering or classification.

The presented infrastructure around the **tm** (Feinerer, 2008b) software offers capabilities as found in well known commercial and open source text mining projects. The **tm** package offers functionality for managing text documents, abstracts the process of document manipulation and eases the usage of heterogeneous text formats in R. The package has integrated database backend support to minimize memory demands. An advanced meta data management is implemented for collections of text documents to alleviate the usage of large and with meta data enriched document sets. With the package ships native support for handling the Reuters 21578 data set, Gmane Rss feeds, e-mails, and several classic file formats (e.g. plain text, Csv text, or Pdfs). The data structures and algorithms can be extended to fit custom demands, since the package is designed in a modular way to enable easy integration of new file formats, readers, transformations and filter operations. **tm** provides easy access to preprocessing and manipulation mechanisms such as whitespace removal, stemming, or conversion between file formats. Further a generic filter architecture is available in order to filter documents for certain criteria, or perform full text search. The package supports the export from document collections to term-document matrices, and string kernels can be easily constructed from text documents.

All this functionality allows the user to perform realistic text mining investigations utilizing the broad range of excellent statistical methods already provided by R, e.g., in the application fields stylometry, e-commerce, or law.

This thesis is organized as follows. Chapter 2 gives a short introduction and overview to text mining and to its applications for document clustering, document classification, information retrieval, natural language processing, stylometry, biomedicine, and web mining. Following chapters 3, 4, 5, 6, and 7 are mainly a compilation of a recent journal paper (Feinerer et al., 2008): Chapters 3, 4, and 5 present the developed text mining infrastructure for R centered around the software package **tm**. Chapter 3 elaborates, on a conceptual level, important ideas and tasks a text mining framework should be able to deal with. Chapter 4 presents the main structure of our framework, its algorithms, and ways to extend the text mining framework for custom demands. Chapter 5 describes preprocessing mechanisms, like data import, stemming, stopword removal and synonym detection. Chapter 6 shows how to conduct typical text mining tasks within our framework, like count-based evaluation methods, text clustering with term-document matrices, text classification, and text clustering with string kernels. Chapter 7 presents an application of **tm** by analyzing the R-devel 2006 mailing list. Chapter 8 shows an application of text mining for business to consumer electronic commerce. Chapter 9 is an application of **tm** to investigate Austrian supreme administrative court jurisdictions concerning dues and taxes. It is mainly based on a recently published article by Feinerer and Hornik (2008). Chapter 10 shows an application for stylometry and authorship attribution on the Wizard of Oz data set. Chapter 11 concludes. Finally Appendix A describes the internal data structures of **tm** whereas Appendix B gives a very detailed and technical description of **tm**'s methods.

2 Text Mining Overview

This chapter gives a short introduction and overview to text mining. We start with a general part where we present different definitions of text mining found in the literature and investigate what makes text mining special compared to classical data mining approaches. We identify application areas specific to text mining with focus on recent developments. We highlight existing research in text mining for document clustering, document classification, natural language processing, stylometry, biomedicine, and web mining. Finally we compare text mining frameworks with a focus on business intelligence applications and motivate the creation of our own text mining infrastructure presented in later chapters.

2.1 What is Text Mining?

2.1.1 Automated Processing of Text

Miller (2005) describes text mining as "the automated or partially automated processing of text". He characterizes text mining with a process model explaining components and interacting steps specific to texts. Figure 2.1 depicts such a process model.

At the beginning there is the raw text input denoted as *text corpus* representing a collection of text documents, like memos, reports, or publications. Grammatical parsing and preprocessing steps transform the unstructured text corpus into a semi-structured format denoted as a *text database*. Depending on the input material this process step may be highly complex involving file format conversions or sophisticated meta-data handling, or a simple task as just reading in the texts, i.e., redundantizing the parsing step, and collapsing the notions text corpus and text database.

Subsequently a structured representation is created by computing a *term-document matrix* from either the text corpus or the text database. The term-document matrix is a bag-of-words mechanism containing term frequencies for all documents in the corpus. This common data structure forms the basis for further text mining analyses, like

- text classification, i.e., assign a priori known labels to text documents,
- syntax analysis, i.e., analyzing the syntactic structure of texts,
- relationship identification, i.e., finding connections and similarities between distinct subsets of documents in the corpus,
- information extraction and retrieval, and

Figure 2.1: A process model for text mining.

- document summarization, i.e., extracting relevant and representative keywords, phrases, and sentences from texts.

2.1.2 A Knowledge-Intensive Process

According to Feldman and Sanger (2007) "text mining can be broadly defined as a knowledge-intensive process in which a user interacts with a document collection over time by using a suite of analysis tools". They emphasize that preprocessing is a major step in text mining compared to data mining since it involves significant processing steps for transforming a text into a structured format suitable for later analysis. A process model similar to Miller (2005) is suggested but with extensions on *refinement* techniques and *browsing functionality*.

Postprocessing In this context refinement (or *postprocessing*) means pruning, ordering, and generalization techniques used to improve or refine the results obtained from previous analysis steps (which they call *core text mining* operations). Core text mining includes pattern and knowledge discovery, deterministic and probabilistic information extraction, trend analysis, term frequency counting, clustering and link analysis. Depending on formal definitions some of these topics are found both in the core and refinement techniques' vocabulary.

Browsing functionality Browsing functionality subsumes filters, query and search interpreters, and visualizations tools like graph drawing. This includes tools for viewing or modifying clustering results, interactive feedback on visualizations (e.g., the graphical effects when a term is removed as an association to other concepts), backends to text search tools, and many more. Summarizing this functionality provides a main interaction point for the user with the text mining framework and process in behind.

Trend Analysis A rather new aspect in the text mining literature discussed by Feldman and Sanger (2007) is trend analysis in text mining. This view stems from the fact that text document collections are typically evolving over time. E.g., a news collection grows with each news story provided by an information broker, or may change by restricting the

time frame for the publication date of relevant news articles. Until recently document collections have been seen as rather static entities with no need in revolving analysis techniques. Evolving corpora demand for incremental techniques to be applied on regular intervals modifying and updating existing analysis results. Trend detection is now an active research field both in web and text mining, so e.g., Wang et al. (Berry, 2003, pages 173–183) show trend and behavior detection from web queries, or a survey by Kontostathis et al. (Berry, 2003, pages 185–224), discusses emerging trend detection in textual data mining. In detail they present several systems capable of textual trend detection, e.g., TimeMines, ThemeRiver, SPSS LexiQuest, or ClearForest, and explain some aspects of their theoretical background.

2.1.3 Use of Text Collections to Discover New Facts

A different view with a strong focus on undiscovered information within texts is described by Hearst (1999) where he suggests a metaphor for text mining: "the use of large online text collections to discover new facts and trends about the world itself". As a result he clearly distinguishes between various aspects normally subsumed under the concept of text mining depending on its novelty: *information retrieval* as a non-novel method for extracting information out of texts, whereas *real text data mining* means finding novel information patterns. The following overview shows how these two concepts can be classified within classical data and text mining approaches.

Non-Textual Data demands techniques from standard data mining where the majority of non-novel information is typically obtained via database queries.

Textual Data analysis takes methods from computational linguistics for novel text data mining and computer science techniques for information retrieval.

Novel text mining includes exploratory data analysis, i.e., the discovery of a priori unknown facts derived from texts, and hypothesis generation. The latter has successfully been applied in biology and medicine, e.g., investigations and medical hypothesis generation of causes for migraine headaches by mining abstracts and texts of related biomedical literature (Swanson and Smalheiser, 1994, 1997).

2.1.4 Analyzing Unstructured Information

Weiss et al. (2004) present a comprehensive overview and introduction to text mining. They discuss whether and how text is different from classical input in data mining, and why text is important. This results in a detailed explanation of necessary preprocessing steps due to the nature of unstructured text, and the importance of texts in daily (private and business) communication is highlighted.

According to Weiss et al. (2004) techniques for the transformation of unstructured text into structured formats (e.g., numerical vectors) include:

- document standardization, i.e., bringing different file formats to a common denominator like XML,

- tokenization, i.e., parsing the text into separate entities, like words or punctuation marks,
- lemmatization, also denoted as stemming,
- sentence boundary detection,
- part-of-speech tagging (POS), i.e., marking tokens with linguistic tags representing their grammatical speech,
- word sense disambiguation using dictionaries or catalogs if POS is not able to determine the exact meaning of a word,
- phrase and named entity recognition,
- parsing, e.g., in linguistics this means building a full parse tree or a directed acyclic graph for a sentence, and finally
- vector and feature generation, i.e., creating a term-document matrix from the in previous steps preprocessed tokens.

2.1.5 Intelligent Text Processing

Another recent overview on computational linguistics and intelligent text processing is presented by Gelbukh (2004). Topics include linguistics formalisms (syntax, grammar, and semantic frameworks), parsing, lexical analysis, named entity and topic recognition (Newman et al., 2006), word sense disambiguation, bilingual texts, machine translation, natural language generation, human-computer interaction, speech recognition, indexing, information retrieval, question answering, sentence retrieval, browsing, filtering, and information extraction.

The huge amount of contributions underlines the cross-disciplinary research in text mining with connections to various related fields, like statistics, computer science, or linguistics.

2.2 Text Mining and Its Applications

As described by Sirmakessis (2004) text mining applications cover a very broad range of interesting topics. Topics include document processing and visualization, web mining utilizing the content, structure and usage of texts in the web, business relevant text mining examples in linguistics, document clustering especially for web documents and automatic knowledge extraction. In addition they address multilingual corpora, industry aspects in text mining—e.g., with the tools in the NEMIS framework, the TEMIS toolkit, or Cogito, a linguistic platform—, and validation issues for text mining via bootstrapping and Monte Carlo methods for unsupervised methods.

The next subsections will give an overview of the most common text mining application fields within the last decades, discussing the relevant literature.

2.2.1 Text Clustering, Text Classification, and Information Retrieval

Since the early beginnings in the 1960s of what we would call today text mining (Weiss et al., 2004), mainly in the field of computational linguistics for textual analysis (Patófi, 1969; Walker, 1969), three major areas have continuously evolved as the basis of modern text mining: *text clustering* (also denoted as *document clustering*), *text classification* (also denoted as *document classification*), and *information retrieval*, all three originally coming from the data mining community.

Text Clustering Text clustering (Zhao and Karypis, 2005b,a; Boley, 1998; Boley et al., 1999; Steinbach et al., 2000), typically uses standard clustering techniques from statistics, like k-means or hierarchical clustering (Willett, 1988), based on a transformation of the text into a structured format, like a term-document matrix, subject to a wide variety of weighting schemes (Salton and Buckley, 1988) and dissimilarities measures (Zhao and Karypis, 2004; Strehl et al., 2000) (e.g., the common Cosine measure). Typical applications in document clustering include grouping news articles or information service documents (Steinbach et al., 2000). A special challenge for text clustering compared to classical clustering are large document collections (Tjhi and Chen, 2007) since the underlying data structures, e.g., term-document matrices, tend to grow significantly with bigger corpora. A possible solution are dimension reduction methods for text classification as presented by Howland and Park (Berry, 2003, pages 3–23) or intelligent feature selection for document clustering as shown by Dhillon et al. (Berry, 2003, pages 73–100).

Text Classification Text classification, also known as *text categorization*, is primarily combined and enhanced with methods from machine learning (Sebastiani, 2002), especially with support vector machines (Joachims, 1998). Additionally basic statistical classification techniques are employed, like k-nearest neighbor classification, Bayes classifiers, or decision trees. Compared to classical classification tasks texts possess specific inherent structures—the latent semantic structure within texts—and thus need the consideration of specific discriminative features (Torkkola, 2004) during the classification process. Traditionally text categorization methods are used in, e.g., e-mail filters and automatic labeling of documents in business libraries (Miller, 2005). Newer approaches utilize so-called string kernels (Lodhi et al., 2002) instead of term-document matrices so that the word order in documents is preserved.

Information Extraction and Retrieval Information retrieval deals with extracting information out of texts, a technique commonly used by modern Internet search engines. With the advent of the World Wide Web, support for information retrieval tasks (carried out by, e.g., search engines and web robots) has quickly become an issue. Here, a possibly unstructured user query is first transformed into a structured format, which is then matched against texts coming from a data base. To build the latter, again, the challenge is to normalize unstructured input data to fulfill the repositories' requirements on information quality and structure,

which often involves grammatical parsing. A collection of approaches for information extraction and retrieval in texts is given by Berry (2003) via a compilation of extended conference papers of the Second SIAM International Conference on Data Mining in 2002. It starts with a discussion on vector space models for search and cluster mining by Kobayashi and Aono (Berry, 2003, pages 103–122), unveils an approach for discovering relevant topics in low syntactic quality texts, i.e., text with typos, syntax and grammar misusage, or abbreviations by Castellanos (Berry, 2003, pages 123–157). Next, Cornelson et al. (Berry, 2003, pages 159–169) present families of retrieval algorithms utilizing meta learning.

2.2.2 Text Mining and Natural Language Processing

Text mining typically aims to extract or generate new information from textual information but does not necessarily need to understand the text itself. Instead, compared to text mining, natural language processing (Manning and Schütze, 1999; Jurafsky and Martin, 2000) tries to obtain a thorough impression on the language structure within texts, i.e., a form of real language processing (Lewis and Jones, 1996). This allows a deeper analysis of sentence structures, grammar (Wallis and Nelson, 2001), morphology, et cetera, and thus better retrieves the latent semantic structure inherent to texts. Techniques in natural language processing include:

- deep parsing utilizing tree structures for retrieving semantic information,
- n-grams and higher-order data structures for representing phrases or sentence structures,
- word sense disambiguation using dictionaries, thesauri, and statistical learning methods, and
- part-of-speech tagging utilizing context-free grammars and hidden Markov models.

In addition natural language processing uses methods known from information retrieval and classical text mining, like text clustering and text categorization.

Among newer methods is latent semantic analysis (Landauer et al., 1998; Deerwester et al., 1990) which tries to preserve and extract the latent structure in texts via singular value decompositions of the term-document matrix.

2.2.3 Text Mining for Biomedical Research

During the last decade text mining has encountered a steady increase in the application to biomedical research documents. The main causal factors for this trend are the sheer amount of daily growing medical literature, the availability of a comprehensive and well organized literature database system for biomedical research literature—the medical literature analysis and retrieval system MEDLINE (e.g., freely accessable via the search engine PUBMED at `http://www.pubmed.gov`)—and the efforts to adapt existing text mining techniques to work for biomedical texts.

Cohen and Hersh (2005) give a survey of current work in biomedical text mining. They identify the most common research text mining techniques used in this context. These are named entity recognition for identifying biomedical terms (Tanabe and Wilbur, 2002), like disease or gene names, text classification, e.g., by using support vector machines trained from MedLine abstracts for protein classification (Donaldson et al., 2003) and synonym and abbreviation extraction (Schwartz and Hearst, 2003). Then there are relationship extraction, e.g., identifying a biochemical association between drugs, using natural language processing techniques, hypothesis generation (Srinivasan, 2003; Swanson and Smalheiser, 1994, 1997), and integration frameworks to unify text mining approaches in the biomedical field.

A similar overview is presented by de Bruijn and Martin (2002) explaining the mining process of biomedical literature. They distinguish between four general text mining tasks widely used in mining biomedical literature: text categorization, named entity tagging, fact extraction, and collection-wide analysis. All these task are subsumed under the notion *automated reading*. Although most of these techniques are employed to use abstracts instead of full texts (e.g., MedLine abstracts) their accuracy has already been quite promising. The inclining usage of the full document content helps to improve the results but poses new challenges for automated reading: abstracts are rather standardized so this helps classical natural language processing techniques.

The four presented components of automated reading are considered as a modular process working similar to a human reader:

1. Text categorization deals with the classification of documents so that a selection of relevant material is possible.

2. Next, named entity tagging highlights relevant terms in the selected documents, e.g., protein and gene names.

3. Based on this information fact extraction deals with generating elaborate patterns out of existing tags and the existing structure.

4. Finally collection-wide analysis links and creates connections between patterns in different documents, i.e., a collection-wide view and information extraction is possible.

Further overview material to mining in the biomedical context is given by Shatkay and Feldman (2003) for mining the biomedical literature in the genomic era, by Ananiadou and Mcnaught (2005) for text mining for biology and biomedicine, and by Krallinger et al. (2005) for text mining approaches in molecular biology and biomedicine.

Biomedical text mining as a very active research field in recent years led to the development of a broad set of tools specialized in text mining in this context, e.g., the Bioconductor (Gentleman et al., 2004, 2005) project for R which is an open source software for the analysis and comprehension of genomic data. A recent list of biomedical text mining tools can be found at `http://arrowsmith.psych.uic.edu/arrowsmith_uic/tools.html`[1].

[1] Accessed 22 May 2008

2.2.4 Text Mining in Stylometry

During the last years another interesting application field of text mining techniques has been stylometry research. Textual stylometry deals with identifying the linguistic style of text documents. Typical research topics are the authorship identification problem, i.e., who wrote a specific text passage, or linguistic forensic tests. The advance of text mining techniques and computing power has led to a steady rise in usage of text mining for stylometry.

Classical textual stylometry (Holmes and Kardos, 2003) mainly deals with historical documents subject to unclear author-document correspondence. Examples are poems of *Shakespeare* (Holmes, 1998), books of the *Wizard of Oz* (Binongo, 2003), or the *Federalist Papers* (Tweedie et al., 1996). Typical text mining techniques for suchlike stylometry include

- Bayesian literal style analysis (Girón et al., 2005),
- neural network (Tweedie et al., 1996) and pattern recognition methods (Matthews and Merriam, 1993), and
- techniques known from text classification dominated by approaches with support vector machines (Burges, 1998).

In the context of linguistic forensics the main motivations for author identification are criminalistics and law enforcement, e.g., texts like threat letters or plagiarized documents authored by a suspected person need to be classified. de Vel et al. (2001) present some techniques for linguistic forensics, especially for mining the content of e-mails for author identification. In detail they use structural characteristics and linguistic patterns (like grammar, pronunciation, or spelling) in the text for clustering, whereas they use support vector machines trained with a set of filtered training documents for classification.

2.2.5 Web Mining

The availability of large text databases in combination with easy access makes the *World Wide Web* a very attractive source for text mining approaches. The resulting discipline is commonly denoted as *web mining* (Kosala and Blockeel, 2000; Cooley et al., 1997), i.e., text mining with material extracted from the web.

Latest developments in document exchange have brought up valuable concepts for automatic handling of texts. The semantic web (Berners-Lee et al., 2001; Berendt et al., 2002) propagates standardized formats for document exchange to enable agents to perform semantic operations on them. This is implemented by providing metadata and by annotating the text with tags. One key format is RDF (Manola and Miller, 2004) which can be handled in R with the Bioconductor project. This development offers great flexibility in document exchange. But with the growing popularity of semi-structured data formats and XML (Mignet et al., 2003) based formats (e.g., RDF/XML as a common representation for RDF) tools need to be able to handle XML documents and metadata.

Web mining is a highly interdisciplinary research field: on the one side it uses the existing mechanisms from classical text mining to operate on the textual data structures, on the other side the heterogeneous data formats on the web call for intelligent data processing mechanisms. The latter can be found in the research of database theory and information retrieval. In addition, web mining is broadly used in combination with other data mining techniques, e.g., web mining is used to enable automatic personalization based on web usage (Mobasher et al., 2000), or in search engines for learning rankings of documents from search engine logs of user behavior (Radlinski and Joachims, 2007). Further, web mining provides the basis for building mining applications in natural language processing (Knight, 1999) in the Internet, e.g., for automatic translation.

2.3 Text Mining Frameworks

The benefit of text mining comes with the large amount of valuable information latent in texts which is not available in classical structured data formats for various reasons: text has always been the default way of storing information for hundreds of years, and mainly time, personal and cost constraints prohibit us from bringing texts into well structured formats (like data frames or tables).

Statistical contexts for text mining applications in research and business intelligence include business applications making use of unstructured texts (Kuechler, 2007; Fan et al., 2006), latent semantic analysis techniques in bioinformatics (Dong et al., 2006), the usage of statistical methods for automatically investigating jurisdictions (Feinerer and Hornik, 2008), plagiarism detection in universities and publishing houses, computer assisted cross-language information retrieval (Li and Shawe-Taylor, 2007) or adaptive spam filters learning via statistical inference. Further common scenarios are help desk inquiries (Sakurai and Suyama, 2005), measuring customer preferences by analyzing qualitative interviews (Feinerer and Wild, 2007), automatic grading (Wu and Chen, 2005), fraud detection by investigating notification of claims, or parsing social network sites for specific patterns such as ideas for new products.

Nowadays almost every major statistical computing product offers text mining capabilities, and many well-known data mining products provide solutions for text mining tasks. According to a recent review on text mining products in statistics (Davi et al., 2005) these capabilities and features include:

Preprocess: data preparation, importing, cleaning and general preprocessing,

Associate: association analysis, that is finding associations for a given term based on counting co-occurrence frequencies,

Cluster: clustering of similar documents into the same groups,

Summarize: summarization of important concepts in a text. Typically these are high-frequency terms,

Categorize: classification of texts into predefined categories, and

Product	Preprocess	Associate	Cluster	Summarize	Categorize	API
Commercial						
ClearForest	✓	✓	✓	✓		
Copernic Sum.	✓			✓		
dtSearch	✓	✓		✓		
Insightful Infact	✓	✓	✓	✓	✓	✓
Inxight	✓	✓	✓	✓	✓	✓
SPSS Clementine	✓	✓	✓	✓	✓	
SAS Text Miner	✓	✓	✓	✓	✓	
TEMIS	✓	✓	✓	✓	✓	
WordStat	✓	✓	✓	✓	✓	
Open Source						
GATE	✓	✓	✓	✓	✓	✓
RapidMiner	✓	✓	✓	✓	✓	✓
Weka/KEA	✓	✓	✓	✓	✓	✓
R/**tm**	✓	✓	✓	✓	✓	✓

Table 2.1: Overview of text mining products and available features. A feature is marked as implemented (denoted as ✓) if the official feature description of each product explicitly lists it.

API: availability of application programming interfaces to extend the program with plug-ins.

Table 2.1 gives an overview over the most-used commercial products (Piatetsky-Shapiro, 2005) for text mining, selected open source text mining tool kits, and features. Commercial products include ClearForest (http://clearforest.com/), a text-driven business intelligence solution, Copernic Summarizer (http://www.copernic.com/), a summarizing software extracting key concepts and relevant sentences, dtSearch (http://www.dtsearch.com/), a document search tool, Insightful Infact (http://www.insightful.com/), a search and analysis text mining tool, Inxight (http://www.businessobjects.com), an integrated suite of tools for search, extraction, and analysis of text, SPSS Clementine (http://www.spss.com/), a data and text mining workbench, SAS Text Miner (http://www.sas.com/), a suite of tools for knowledge discovery and knowledge extraction in texts, TEMIS (http://www.temis.com/), a tool set for text extraction, text clustering, and text categorization, and WordStat (http://www.provalisresearch.com), a product for computer assisted text analysis.

From Table 2.1 we see that most commercial tools lack easy-to-use API integration and provide a relatively monolithic structure regarding extensibility since their source code is not freely available.

Among well known open source data mining tools offering text mining functionality is the **Weka** (Witten and Frank, 2005) suite (now part of the **Pentaho** project), a collection of machine learning algorithms for data mining tasks also offering classification

and clustering techniques with extension projects for text mining, like **KEA** (Witten et al., 1999, 2005) for keyword extraction. It provides good API support and has a wide user base. Then there is **GATE** (Cunningham et al., 2002), an established text mining framework with architecture for language processing, information extraction, ontology management and machine learning algorithms. Other tools are **RapidMiner** (formerly **Yale** (Mierswa et al., 2006)), a system for knowledge discovery and data mining, and **Pimiento** (Adeva and Calvo, 2006), a basic Java framework for text mining. However, many existing open-source products tend to offer rather specialized solutions in the text mining context, such as **Shogun** (Sonnenburg et al., 2006), a toolbox for string kernels, or the **Bow** toolkit (McCallum, 1996), a C library useful for statistical text analysis, language modeling and information retrieval. In R the (now deprecated) extension package **ttda** (Mueller, 2006) provides some methods for textual data analysis.

The next chapters present a text mining framework for the open source statistical computing environment R centered around the new extension package **tm**. This open source package, with a focus on extensibility based on generic functions and object-oriented inheritance, provides the basic infrastructure necessary to organize, transform, and analyze textual data. R has proven over the years to be one of the most versatile statistical computing environments available, and offers a battery of both standard and state-of-the-art methodology. However, the scope of these methods was often limited to "classical", structured input data formats (such as data frames in R). The **tm** package provides a framework that allows researchers and practitioners to apply a multitude of existing methods to text data structures as well. In addition, advanced text mining methods beyond the scope of most today's commercial products, like string kernels or latent semantic analysis, can be made available via extension packages, such as **kernlab** (Karatzoglou et al., 2004, 2006) or **lsa** (Wild, 2005), or via interfaces to established open source toolkits from the data/text mining field like **Weka** or **OpenNLP** (Bierner et al., 2007) from the natural language processing community. So **tm** provides a framework for flexible integration of premier statistical methods from R, interfaces to well known open source text mining infrastructure and methods, and has a sophisticated modularized extension mechanism for text mining purposes.

Part I

Text Mining Framework

3 Conceptual Process and Framework

A text mining analysis involves several challenging process steps mainly influenced by the fact that texts, from a computer perspective, are rather unstructured collections of words. A text mining analyst typically starts with a set of highly heterogeneous input texts. So the first step is to import these texts into one's favorite computing environment, in our case R. Simultaneously it is important to organize and structure the texts to be able to access them in a uniform manner. Once the texts are organized in a repository, the second step is tidying up the texts, including preprocessing the texts to obtain a convenient representation for later analysis. This step might involve text reformatting (e.g., whitespace removal), stopword removal, or stemming procedures. Third, the analyst must be able to transform the preprocessed texts into structured formats to be actually computed with. For "classical" text mining tasks, this normally implies the creation of a so-called term-document matrix, probably the most common format to represent texts for computation. Now the analyst can work and compute on texts with standard techniques from statistics and data mining, like clustering or classification methods.

This rather typical process model highlights important steps that call for support by a text mining infrastructure: A text mining framework must offer functionality for managing text documents, should abstract the process of document manipulation and ease the usage of heterogeneous text formats. Thus there is a need for a conceptual entity similar to a database holding and managing text documents in a generic way: we call this entity a *text document collection* or *corpus*.

Since text documents are present in different file formats and in different locations, like a compressed file on the Internet or a locally stored text file with additional annotations, there has to be an encapsulating mechanism providing standardized interfaces to access the document data. We subsume this functionality in so-called *source*s.

Besides the actual textual data many modern file formats provide features to annotate text documents (e.g., XML with special tags), i.e., there is *metadata* available which further describes and enriches the textual content and might offer valuable insights into the document structure or additional concepts. Also, additional metadata is likely to be created during an analysis. Therefore the framework must be able to alleviate metadata usage in a convenient way, both on a document level (e.g., short summaries or descriptions of selected documents) and on a collection level (e.g., collection-wide classification tags).

Alongside the data infrastructure for text documents the framework must provide tools and algorithms to efficiently work with the documents. That means the framework has to have functionality to perform common tasks, like whitespace removal, stemming or stopword deletion. We denote such functions operating on text document collections

as *transformations*. Another important concept is *filtering* which basically involves applying predicate functions on collections to extract patterns of interest. A surprisingly challenging operation is the one of *joining* text document collections. Merging sets of documents is straightforward, but merging metadata intelligently needs a more sophisticated handling, since storing metadata from different sources in successive steps necessarily results in a hierarchical, tree-like structure. The challenge is to keep these joins and subsequent look-up operations efficient for large document collections.

Realistic scenarios in text mining use at least several hundred text documents ranging up to several hundred thousands of documents. This means a compact storage of the documents in a document collection is relevant for appropriate RAM usage—a simple approach would hold all documents in memory once read in and bring down even fully RAM equipped systems shortly with document collections of several thousands text documents. However, simple database orientated mechanisms can already circumvent this situation, e.g., by holding only pointers or hashtables in memory instead of full documents.

Text mining typically involves doing computations on texts to gain interesting information. The most common approach is to create a so-called *term-document matrix* holding frequencies of distinct terms for each document. Another approach is to compute directly on character sequences as is done by string kernel methods. Thus the framework must allow export mechanisms for term-document matrices and provide interfaces to access the document corpora as plain character sequences.

3.1 Framework Overview

Basically, the framework and infrastructure supplied by the **tm** R extension package aims at implementing the conceptual framework presented above. The next chapters will introduce the data structures and algorithms provided. Then there are several other extension packages which provide interfaces to existing text mining or natural language processing tool kits which integrate well with the **tm** package, and are also freely available at *The Comprehensive R Archive Network* (CRAN, http://cran.r-project.org/):

openNLP An interface to **OpenNLP** (Bierner et al., 2007) (which is available at http://opennlp.sourceforge.net/), a collection of natural language processing tools including a sentence detector, tokenizer, part-of-speech-tagger, shallow and full syntactic parser, and named-entity detector, using the Maxent Java package for training and using maximum entropy models.

openNLPmodels English and Spanish training models for **OpenNLP**. These models are provided so that the **openNLP** package can be used out of the box, at least for English and Spanish language users.

RKEA An interface to **KEA** (Witten et al., 1999, 2005) (available at http://www.nzdl.org/Kea/) which provides an algorithm for extracting keyphrases from text documents. It can be either used for free indexing or for indexing with a controlled vocabulary.

tm This package offers functionality for managing text documents, abstracts the process of document manipulation and eases the usage of heterogeneous text formats in R. The package has integrated database backend support to minimize memory demands. An advanced meta data management is implemented for collections of text documents to alleviate the usage of large and with meta data enriched document sets. With the package ships native support for handling the Reuters 21578 data set, Gmane Rss feeds, e-mails, and several classic file formats (e.g. plain text, Csv text, or Pdfs). The data structures and algorithms can be extended to fit custom demands, since the package is designed in a modular way to enable easy integration of new file formats, readers, transformations and filter operations. **tm** provides easy access to preprocessing and manipulation mechanisms such as whitespace removal, stemming, or conversion between file formats. Further a generic filter architecture is available in order to filter documents for certain criteria, or perform full text search. The package supports the export from document collections to term-document matrices, and string kernels can be easily constructed from text documents.

wordnet An interface to WordNet (Fellbaum, 1998) (available at http://wordnet.princeton.edu/) using the Jawbone Java Api to WordNet. WordNet is an on-line lexical reference system developed by the Cognitive Science Laboratory at Princeton University. Its design is inspired by current psycholinguistic theories of human lexical memory. English nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing one underlying lexical concept. Different relations link the synonym sets.

Furthermore there is a task view dedicated to text mining and natural language processing. There is the Cran *Natural Language Processing Task View* at http://cran.r-project.org/web/views/NaturalLanguageProcessing.html which contains a list of R packages useful for natural language processing. It covers a broad range of topics like lexical databases, natural language processing, string kernels, and text mining. In addition it lists corresponding literature and provides links to related task views, e.g., on clustering or machine learning, and other related software.

4 Data Structures and Algorithms

In this section we explain both the data structures underlying our text mining framework and the algorithmic background for working with these data structures. We motivate the general structure and show how to extend the framework for custom purposes.

Commercial text mining products (Davi et al., 2005) are typically built in monolithic structures regarding extensibility. This is inherent as their source code is normally not available. Also, quite often interfaces are not disclosed and open standards hardly supported. The result is that the set of predefined operations is limited, and it is hard (or expensive) to write plug-ins.

Therefore we decided to tackle this problem by implementing a framework for accessing text data structures in R. We concentrated on a middle ware consisting of several text mining classes that provide access to various texts. On top of this basic layer we have a virtual application layer, where methods operate without explicitly knowing the details of internal text data structures. The text mining classes are written as abstract and generic as possible, so it is easy to add new methods on the application layer level. The framework uses the S4 (Chambers, 1998) class system to capture an object oriented design. This design seems best capable of encapsulating several classes with internal data structures and offers typed methods to the application layer.

This modular structure enables **tm** to integrate existing functionality from other text mining tool kits. E.g., we interface with the **Weka** and **OpenNLP** tool kits, via **RWeka** (Hornik et al., 2007) (and **Snowball** (Hornik, 2007b) for its stemmers) and **openNLP** (Feinerer, 2008a), respectively. In detail **Weka** gives us stemming and tokenization methods, whereas **OpenNLP** offers amongst others tokenization, sentence detection, and part of speech tagging (Bill, 1995). We can plug in this functionality at various points in **tm**'s infrastructure, e.g., for preprocessing via transformation methods (see Chapter 5), for generating term-document matrices (see Section 4.1.4), or for custom functions when extending **tm**'s methods (see Section 4.3).

Figure 4.1 shows both the conceptual layers of our text mining infrastructure and typical packages arranged in them. The system environment is made up of the R core

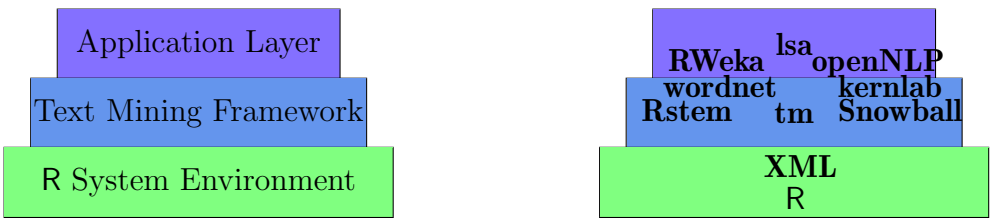

Figure 4.1: Conceptual layers and packages.

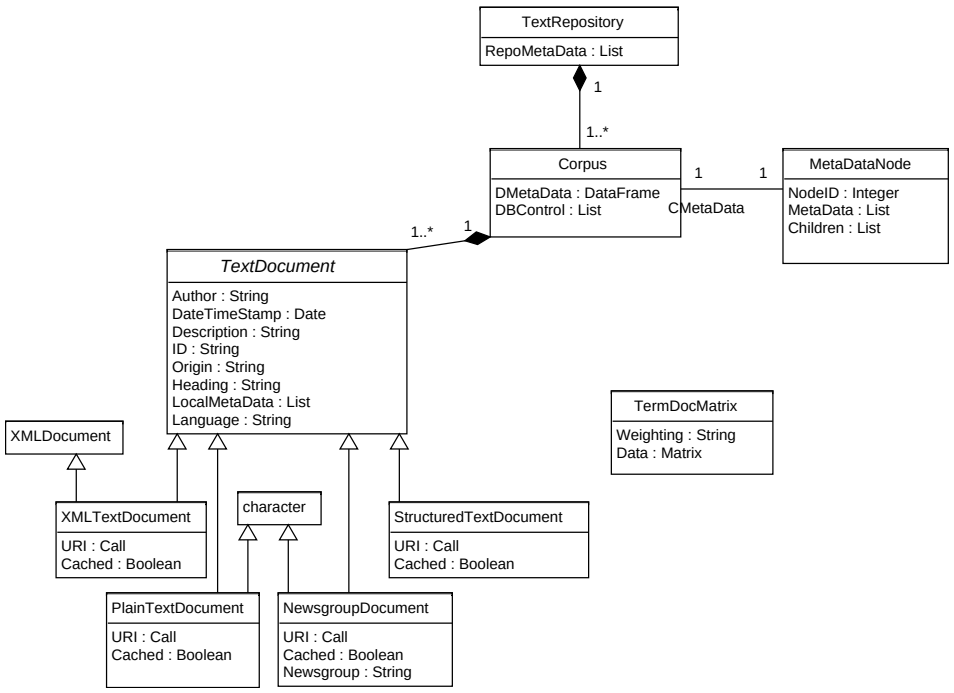

Figure 4.2: UML class diagram of the **tm** package.

and the **XML** (Temple Lang, 2006) package for handling XML documents internally, the text mining framework consists of our new **tm** package with some help of **Rstem** (Temple Lang, 2004) or **Snowball** for stemming, whereas some packages provide both infrastructure and applications, like **wordnet** (Feinerer, 2008c), **kernlab** with its string kernels, or the **RWeka** and **openNLP** interfaces. A typical application might be **lsa** which can use our middleware: the key data structure for latent semantic analysis (Landauer et al., 1998; Deerwester et al., 1990) (LSA) is a term-document matrix which can be easily exported from our **tm** framework. As default **lsa** provides its own (rather simple) routines for generating term-document matrices, so one can either use **lsa** natively or enhance it with **tm** for handling complex input formats, preprocessing, and text manipulations, e.g., as used by Feinerer and Wild (2007).

4.1 Data Structures

We start by explaining the data structures: The basic framework classes and their interactions are depicted in Figure 4.2 as a UML class diagram (Fowler, 2003) with implementation independent UML datatypes. In this section we give an overview how the classes interoperate and work whereas an in-depth description is found in the Appendix A to be used as detailed reference.

4.1.1 Text Document Collections

The main structure for managing documents in **tm** is a so-called text document collection, also denoted as corpus in linguistics (`Corpus`). It represents a collection of text documents and can be interpreted as a database for texts. Its elements are `TextDocuments` holding the actual text corpora and local metadata. The text document collection has two slots for storing global metadata and one slot for database support.

We can distinguish two types of metadata, namely *Document Metadata* and *Collection Metadata*. Document metadata (`DMetaData`) is for information specific to text documents but with an own entity, like classification results (it holds both the classifications for each documents but in addition global information like the number of classification levels). Collection metadata (`CMetaData`) is for global metadata on the collection level not necessarily related to single text documents, like the creation date of the collection (which is independent from the documents within the collection).

The database slot (`DBControl`) controls whether the collection uses a database backend for storing its information, i.e., the documents and the metadata. If activated, package **tm** tries to hold as few bits in memory as possible. The main advantage is to be able to work with very large text collections, a shortcoming might be slower access performance (since we need to load information from the disk on demand). Also note that activated database support introduces persistent object semantics since changes are written to the disk which other objects (pointers) might be using.

Objects of class `Corpus` can be manually created by

```
> new("Corpus", .Data = ..., DMetaData = ..., CMetaData = ...,
+     DBControl = ...)
```

where `.Data` has to be the list of text documents, and the other arguments have to be the document metadata, collection metadata and database control parameters. Typically, however, we use the `Corpus` constructor to generate the right parameters given following arguments:

`object:` a `Source` object which abstracts the input location.

`readerControl:` a list with the three named components `reader`, `language`, and `load`, giving a reader capable of reading in elements delivered from the document source, a string giving the Iso language code (typically in Iso 639 or Iso 3166 format, e.g., `en_US` for American English), and a Boolean flag indicating whether the user wants to load documents immediately into memory or only when actually accessed (we denote this feature as *load on demand*).

The **tm** package ships with several readers (use `getReaders()` to list available readers) described in Table 4.1.

`dbControl:` a list with the three named components `useDb`, `dbName` and `dbType` setting the respective `DBControl` values (whether database support should be activated, the file name to the database, and the database type).

Reader	Description
`readPlain()`	Read in files as plain text ignoring metadata
`readRCV1()`	Read in files in Reuters Corpus Volume 1 XML format
`readReut21578XML()`	Read in files in Reuters 21578 XML format
`readGmane()`	Read in Gmane RSS feeds
`readNewsgroup()`	Read in newsgroup postings in UCI KDD archive format
`readPDF()`	Read in PDF documents
`readDOC()`	Read in MS Word documents
`readHTML()`	Read in simply structured HTML documents

Table 4.1: Available readers in the **tm** package.

An example of a constructor call might be

```
> Corpus(object = ...,
+            readerControl = list(reader = object@DefaultReader,
+                                 language = "en_US",
+                                 load = FALSE),
+            dbControl = list(useDb = TRUE,
+                              dbName = "texts.db",
+                              dbType = "DB1"))
```

where `object` denotes a valid instance of class `Source`. We will cover sources in more detail later.

4.1.2 Text Documents

The next core class is a text document (`TextDocument`), the basic unit managed by a text document collection. It is an abstract class, i.e., we must derive specific document classes to obtain document types we actually use in daily text mining. Basic slots are `Author` holding the text creators, `DateTimeStamp` for the creation date, `Description` for short explanations or comments, `ID` for a unique identification string, `Origin` denoting the document source (like the news agency or publisher), `Heading` for the document title, `Language` for the document language, and `LocalMetaData` for any additional metadata.

The main rationale is to extend this class as needed for specific purposes. This offers great flexibility as we can handle any input format internally but provide a generic interface to other classes. The following four classes are derived classes implementing documents for common file formats and come with the package: `XMLTextDocument` for XML documents, `PlainTextDocument` for simple texts, `NewsgroupDocument` for newsgroup postings and e-mails, and `StructuredTextDocument` for more structured documents (e.g., with explicitly marked paragraphs, etc.).

Text documents can be created manually, e.g., via

```
> new("PlainTextDocument", .Data = "Some text.", URI = uri, Cached = TRUE,
+     Author = "Mr. Nobody", DateTimeStamp = Sys.time(),
```

```
+     Description = "Example", ID = "ID1", Origin = "Custom",
+     Heading = "Ex. 1", Language = "en_US")
```

setting all arguments for initializing the class (`uri` is a shortcut for a reference to the input, e.g., a call to a file on disk). In most cases text documents are returned by reader functions, so there is no need for manual construction.

4.1.3 Text Repositories

The next class from our framework is a so-called text repository which can be used to keep track of text document collections. The class `TextRepository` is conceptualized for storing representations of the same text document collection. This allows to backtrack transformations on text documents and access the original input data if desired or necessary. The dynamic slot `RepoMetaData` can help to save the history of a text document collection, e.g., all transformations with a time stamp in form of tag-value pair metadata.

We construct a text repository by calling

```
> new("TextRepository",
+     .Data = list(Col1, Col2), RepoMetaData = list(created = "now"))
```

where `Col1` and `Col2` are text document collections.

4.1.4 Term-Document Matrices

Finally we have a class for term-document matrices (Berry, 2003; Shawe-Taylor and Cristianini, 2004), probably the most common way of representing texts for further computation. It can be exported from a `Corpus` and is used as a bag-of-words mechanism which means that the order of tokens is irrelevant. This approach results in a matrix with document `ID`s as rows and terms as columns. The matrix elements are term frequencies.

For example, consider the two documents with `ID`s 1 and 2 and their contents `text mining is fun` and `a text is a sequence of words`, respectively. Then the term-document matrix is

	a	fun	is	mining	of	sequence	text	words
1	0	1	1	1	0	0	1	0
2	2	0	1	0	1	1	1	1

`TermDocMatrix` provides such a term-document matrix for a given `Corpus` element. It has the slot `Data` of the formal class `Matrix` from package **Matrix** (Bates and Maechler, 2007) to hold the frequencies in compressed sparse matrix format.

Instead of using the term frequency (`weightTf`) directly, one can use different weightings. The slot `Weighting` of a `TermDocMatrix` provides this facility by calling a weighting function on the matrix elements. Available weighting schemes include the *binary frequency* (`weightBin`) method which eliminates multiple entries, or the *inverse document frequency* (`weightTfIdf`) weighting giving more importance to discriminative compared

to irrelevant terms. Users can apply their own weighting schemes by passing over custom weighting functions to `Weighting`.

Again, we can manually construct a term-document matrix, e.g., via

```
> new("TermDocMatrix", Data = tdm, Weighting = weightTf)
```

where `tdm` denotes a sparse `Matrix`.

Typically, we will use the `TermDocMatrix` constructor instead for creating a term-document matrix from a text document collection. The constructor provides a sophisticated modular structure for generating such a matrix from documents: you can plug in modules for each processing step specified via a `control` argument. E.g., we could use an n-gram tokenizer (`NGramTokenizer`) from the **Weka** toolkit (via **RWeka**) to tokenize into phrases instead of single words

```
> TermDocMatrix(col, control = list(tokenize = NGramTokenizer))
```

or a tokenizer from the **OpenNLP** toolkit (via **openNLP**'s `tokenize` function)

```
> TermDocMatrix(col, control = list(tokenize = tokenize))
```

where `col` denotes a text collection. Instead of using a classical tokenizer we could be interested in phrases or whole sentences, so we take advantage of the sentence detection algorithms offered by **openNLP**.

```
> TermDocMatrix(col, control = list(tokenize = sentDetect))
```

Similarly, we can use external modules for all other processing steps (mainly via internal calls to `termFreq` which generates a term frequency vector from a text document and gives an extensive list of available control options), like stemming (e.g., the **Weka** stemmers via the **Snowball** package), stopword removal (e.g., via custom stopword lists), or user supplied dictionaries (a method to restrict the generated terms in the term-document matrix).

This modularization allows synergy gains between available established toolkits (like **Weka** or **OpenNLP**) and allows **tm** to utilize available functionality.

4.1.5 Sources

The **tm** package uses the concept of a so-called *source* to encapsulate and abstract the document input process. This allows to work with standardized interfaces within the package without knowing the internal structures of input document formats. It is easy to add support for new file formats by inheriting from the `Source` base class and implementing the interface methods.

Figure 4.3 shows a UML diagram with implementation independent UML data types for the `Source` base class and existing inherited classes.

A source is a `VIRTUAL` class (i.e., it cannot be instantiated, only classes may be derived from it) and abstracts the input location and serves as the base class for creating inherited classes for specialized file formats. It has four slots, namely `LoDSupport` indicating

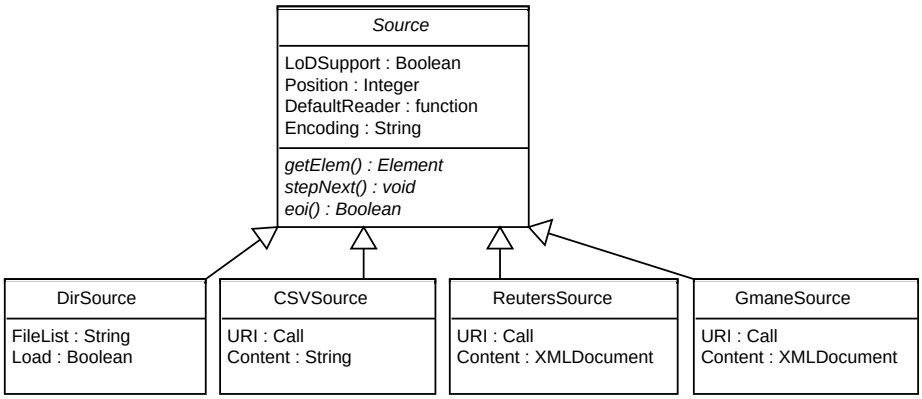

Figure 4.3: UML class diagram for `Source`s.

load on demand support, `Position` holding status information for internal navigation, `DefaultReader` for a default reader function, and `Encoding` for the encoding to be used by internal R routines for accessing texts via the source (defaults to UTF-8 for all sources).

The following classes are specific source implementations for common purposes: The class `DirSource` for directories with text documents, `CSVSource` for documents stored in CSV files, `ReutersSource` for special Reuters file formats, and `GmaneSource` for so-called RSS feeds as delivered by Gmane (Ingebrigtsen, 2007).

A directory source can manually be created by calling

```
> new("DirSource", LoDSupport = TRUE, FileList = dir(), Position = 0,
+     DefaultReader = readPlain, Encoding = "latin1")
```

where `readPlain()` is a predefined reader function in **tm**. Again, we provide wrapper functions for the various sources.

4.2 Algorithms

Next, we present the algorithmic side of our framework. We start with the creation of a text document collection holding some plain texts in Latin language from Ovid's *ars amatoria* (Naso, 2007). Since the documents reside in a separate directory we use the `DirSource` and ask for immediate loading into memory. The elements in the collection are of class `PlainTextDocument` since we use the default reader which reads in the documents as plain text:

```
> txt <- system.file("texts", "txt", package = "tm")
> (ovid <- Corpus(DirSource(txt),
+                 readerControl = list(reader = readPlain,
+                                      language = "la",
+                                      load = TRUE)))
```

```
A text document collection with 5 text documents
```

Alternatively we could activate database support such that only relevant information is kept in memory:

```
> Corpus(DirSource(txt),
+            readerControl = list(reader = readPlain,
+                                 language = "la", load = TRUE),
+            dbControl = list(useDb = TRUE,
+                             dbName = "/home/user/oviddb",
+                             dbType = "DB1"))
```

The loading and unloading of text documents and metadata of the text document collection is transparent to the user, i.e., fully automatic. Manipulations affecting R text document collections are written out to the database, i.e., we obtain persistent object semantics in contrast to R's common semantics.

We have implemented both accessor and set functions for the slots in our classes such that slot information can easily be accessed and modified, e.g.,

```
> ID(ovid[[1]])

[1] "1"
```

gives the `ID` slot attribute of the first `ovid` document. With e.g.,

```
> Author(ovid[[1]]) <- "Publius Ovidius Naso"
```

we modify the `Author` slot information.

To see all available metadata for a text document, use `meta()`, e.g.,

```
> meta(ovid[[1]])

Available meta data pairs are:
  Author       : Publius Ovidius Naso
  Cached       : TRUE
  DateTimeStamp: 2008-05-23 10:33:24
  Description  :
  Heading      :
  ID           : 1
  Language     : la
  Origin       :
  URI          : file /home/feinerer/lib/R/library/tm/texts/txt/ovid_1.txt UTF-8
Dynamic local meta data pairs are:
list()
```

Further we have implemented following operators and functions for text document collections:

`[` The subset operator allows to specify a range of text documents and automatically ensures that a valid text collection is returned. Further the `DMetaData` data frame is automatically subsetted to the specific range of documents.

```
> ovid[1:3]

A text document collection with 3 text documents
```

`[[` accesses a single text document in the collection. A special `show()` method for plain text documents pretty prints the output.

```
> ovid[[1]]

 [1]     Si quis in hoc artem populo non novit amandi,
 [2]          hoc legat et lecto carmine doctus amet.
 [3]     arte citae veloque rates remoque moventur,
 [4]          arte leves currus: arte regendus amor.
 [5]
 [6]     curribus Automedon lentisque erat aptus habenis,
 [7]          Tiphys in Haemonia puppe magister erat:
 [8]     me Venus artificem tenero praefecit Amori;
 [9]          Tiphys et Automedon dicar Amoris ego.
[10]     ille quidem ferus est et qui mihi saepe repugnet:
[11]
[12]          sed puer est, aetas mollis et apta regi.
[13]     Phillyrides puerum cithara perfecit Achillem,
[14]          atque animos placida contudit arte feros.
[15]     qui totiens socios, totiens exterruit hostes,
[16]          creditur annosum pertimuisse senem.
```

`c()` Concatenates several text collections to a single one.

```
> c(ovid[1:2], ovid[3:4])

A text document collection with 4 text documents
```

The metadata of both text document collections is merged, i.e., a new root node is created in the `CMetaData` tree holding the concatenated collections as children, and the `DMetaData` data frames are merged. Column names existing in one frame but not the other are filled up with `NA` values. The whole process of joining the metadata is depicted in Figure 4.4. Note that concatenation of text document collections with activated database backends is not supported since it might involve the generation of a new database (as a collection has to have exactly one database) and massive copying of database values.

`length()` Returns the number of text documents in the collection.

```
> length(ovid)

[1] 5
```

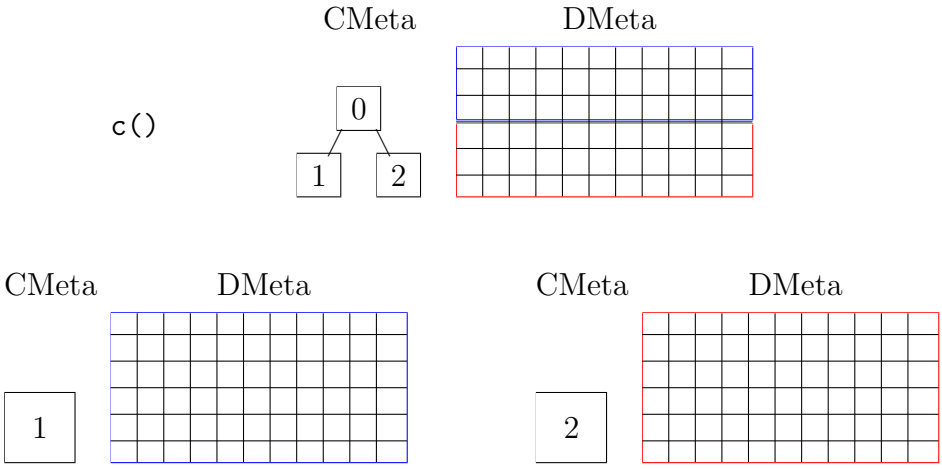

Figure 4.4: Concatenation of two text document collections with `c()`.

`show()` A custom print method. Instead of printing all text documents (consider a text collection could consist of several thousand documents, similar to a database), only a short summarizing message is printed.

`summary()` A more detailed message, summarizing the text document collection. Available metadata is listed.

```
> summary(ovid)

A text document collection with 5 text documents

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator
Available variables in the data frame are:
  MetaID
```

`inspect()` This function allows to actually see the structure which is hidden by `show()` and `summary()` methods. Thus all documents and metadata are printed, e.g., `inspect(ovid)`.

`tmUpdate()` takes as argument a text document collection, a source with load on demand support and a `readerControl` as found in the `Corpus` constructor. The source is checked for new files which do not already exist in the document collection. Identified new files are parsed and added to the existing document collection, i.e., the collection is updated, and loaded into memory if demanded.

```
> tmUpdate(ovid, DirSource(txt))

A text document collection with 5 text documents
```

Text documents and metadata can be added to text document collections with `appendElem()` and `appendMeta()`, respectively. As already described earlier the text document collection has two types of metadata: one is the metadata on the document collection level (`cmeta`), the other is the metadata related to the individual documents (e.g., clusterings) (`dmeta`) with an own entity in form of a data frame.

```
> ovid <- appendMeta(ovid,
+                    cmeta = list(test = c(1,2,3)),
+                    dmeta = list(clust = c(1,1,2,2,2)))
> summary(ovid)

A text document collection with 5 text documents

The metadata consists of 3 tag-value pairs and a data frame
Available tags are:
  create_date creator test
Available variables in the data frame are:
  MetaID clust

> CMetaData(ovid)

An object of class "MetaDataNode"
Slot "NodeID":
[1] 0

Slot "MetaData":
$create_date
[1] "2008-05-23 10:33:24 CEST"

$creator
   LOGNAME
"feinerer"

$test
[1] 1 2 3


Slot "children":
list()

> DMetaData(ovid)

  MetaID clust
1      0     1
2      0     1
```

```
3      0     2
4      0     2
5      0     2
```

For the method `appendElem()`, which adds the `data` object of class `TextDocument` to the data segment of the text document collection `ovid`, it is possible to give a column of values in the data frame for the added data element.

```
> (ovid <- appendElem(ovid, data = ovid[[1]], list(clust = 1)))

A text document collection with 6 text documents
```

The methods `appendElem()`, `appendMeta()` and `removeMeta()` also exist for the class `TextRepository`, which is typically constructed by passing a initial text document collection, e.g.,

```
> (repo <- TextRepository(ovid))

A text repository with 1 text document collection
```

The argument syntax for adding data and metadata is identical to the arguments used for text collections (since the functions are generic) but now we add data (i.e., in this case whole text document collections) and metadata to a text repository. Since text repositories' metadata only may contain repository specific metadata, the argument `dmeta` of `appendMeta()` is ignored and `cmeta` must be used to pass over repository metadata.

```
> repo <- appendElem(repo, ovid, list(modified = date()))
> repo <- appendMeta(repo, list(moremeta = 5:10))
> summary(repo)

A text repository with 2 text document collections

The repository metadata consists of 3 tag-value pairs
Available tags are:
created modified moremeta

> RepoMetaData(repo)

$created
[1] "2008-05-23 10:33:25 CEST"

$modified
[1] "Fri May 23 10:33:25 2008"

$moremeta
[1]  5  6  7  8  9 10
```

The method `removeMeta()` is implemented both for text document collections and text repositories. In the first case it can be used to delete metadata from the `CMetaData` and `DMetaData` slots, in the second case it removes metadata from `RepoMetaData`. The function has the same signature as `appendMeta()`.

In addition there is the method `meta()` as a simplified uniform mechanism to access metadata. It provides accessor and set methods for text collections, text repositories and text documents (as already shown for a document from the `ovid` corpus at the beginning of this section). Especially for text collections it is a simplification since it provides a uniform way to edit `DMetaData` and `CMetaData` (type `corpus`), e.g.,

```
> meta(ovid, type = "corpus", "foo") <- "bar"
> meta(ovid, type = "corpus")

An object of class "MetaDataNode"
Slot "NodeID":
[1] 0

Slot "MetaData":
$create_date
[1] "2008-05-23 10:33:24 CEST"

$creator
   LOGNAME
"feinerer"

$test
[1] 1 2 3

$foo
[1] "bar"


Slot "children":
list()

> meta(ovid, "someTag") <- 6:11
> meta(ovid)

  MetaID clust someTag
1      0     1       6
2      0     1       7
3      0     2       8
4      0     2       9
5      0     2      10
6      0     1      11
```

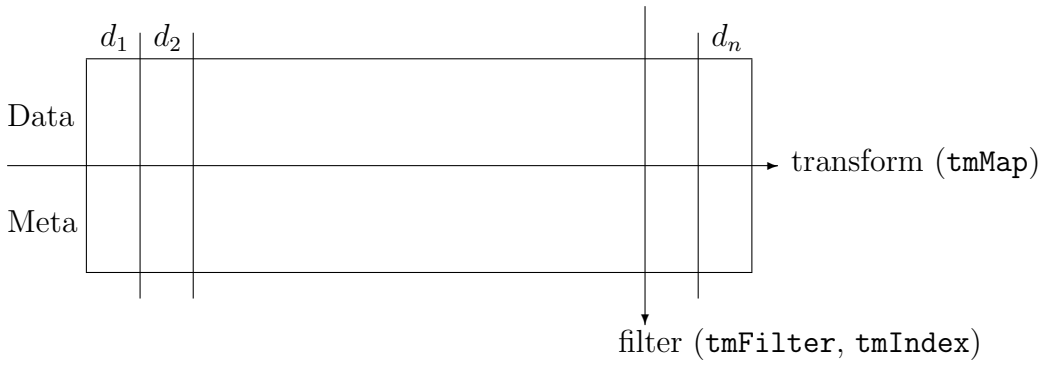

Figure 4.5: Generic transform and filter operations on a text document collection.

In addition we provide a generic interface to operate on text document collections, i.e., transform and filter operations. This is of great importance in order to provide a high-level concept for often used operations on text document collections. The abstraction avoids the user to take care of internal representations but offers clearly defined, implementation independent, operations.

Transformations operate on each text document in a text document collection by applying a function to them. Thus we obtain another representation of the whole text document collection. *Filter* operations instead allow to identify subsets of the text document collection. Such a subset is defined by a function applied to each text document resulting in a Boolean answer. Hence formally the filter function is just a predicate function. This way we can easily identify documents with common characteristics. Figure 4.5 visualizes this process of transformations and filters. It shows a text document collection with text documents d_1, d_2, ..., d_n consisting of corpus data (Data) and the document specific metadata data frame (Meta).

Transformations are done via the `tmMap()` function which applies a function `FUN` to all elements of the collection. Basically, all transformations work on single text documents and `tmMap()` just applies them to all documents in a document collection. E.g.,

```
> tmMap(ovid, FUN = tmTolower)

A text document collection with 6 text documents
```

applies `tmTolower()` to each text document in the `ovid` collection and returns the modified collection. Optional parameters `...` are passed directly to the function `FUN` if given to `tmMap()` allowing detailed arguments for more complex transformations. Further the document specific metadata data frame is passed to the function as argument `DMetaData` to enable transformations based on information gained by metadata investigation. Table 4.2 gives an overview over available transformations (use `getTransformations()` to list available transformations) shipped with **tm**.

Filters (use `getFilters()` to list available filters) are performed via the `tmIndex()` and `tmFilter()` functions. Both function have the same internal behavior except that

Transformation	Description
asPlain()	Converts the document to a plain text document
loadDoc()	Triggers load on demand
removeCitation()	Removes citations from e-mails
removeMultipart()	Removes non-text from multipart e-mails
removeNumbers()	Removes numbers
removePunctuation()	Removes punctuation marks
removeSignature()	Removes signatures from e-mails
removeWords()	Removes stopwords
replaceWords()	Replaces a set of words with a given phrase
stemDoc()	Stems the text document
stripWhitespace()	Removes extra whitespace
tmTolower()	Conversion to lower case letters

Table 4.2: Transformations shipped with **tm**.

tmIndex() returns Boolean values whereas tmFilter() returns the corresponding documents in a new Corpus. Both functions take as input a text document collection, a function FUN, a flag doclevel indicating whether FUN is applied to the collection itself or to each document separately (default), and optional parameters ... to be passed to FUN. As in the case with transformations the document specific metadata data frame is passed to FUN as argument DMetaData. E.g., there is a full text search filter searchFullText() available which accepts regular expressions and is applied on the document level:

```
> tmFilter(ovid, FUN = searchFullText, "Venus")

A text document collection with 2 text documents
```

Any valid predicate function can be used as custom filter function but for most cases the filter sFilter() does its job: it integrates a minimal query language to filter metadata. Statements in this query language are statements as used for subsetting data frames, i.e, a statement s is of format "tag1 == 'expr1' & tag2 == 'expr2' & ...". Tags in s represent data frame metadata variables. Variables only available at the document level are shifted up to the data frame if necessary. Note that the metadata tags for the slots Author, DateTimeStamp, Description, ID, Origin, Language and Heading of a text document are author, datetimestamp, description, identifier, origin, language and heading, respectively, to avoid name conflicts. For example, the following statement filters out those documents having an ID equal to 2:

```
> tmIndex(ovid, FUN = sFilter, "identifier == '2'")

[1] FALSE  TRUE FALSE FALSE FALSE FALSE
```

As you see the query is applied to the metadata data frame (the document local ID metadata is shifted up to the metadata data frame automatically since it appears in the statement) thus an investigation on document level is not necessary.

4.3 Extensions

The presented framework classes already build the foundation for typical text mining tasks but we emphasize available extensibility mechanisms. This allows the user to customize classes for specific demands. In the following, we sketch an example (only showing the main elements and function signatures).

Suppose we want to work with an RSS newsgroup feed as delivered by Gmane (Ingebrigtsen, 2007) and analyze it in R. Since we already have a class for handling newsgroup mails as found in the Newsgroup data set from the UCI KDD archive (Hettich and Bay, 1999) we will reuse it as it provides everything we need for this example. At first, we derive a new source class for our RSS feeds:

```
> setClass("GmaneSource",
+          representation(URI = "ANY", Content = "list"),
+          contains = c("Source"))
```

which inherits from the `Source` class and provides slots as for the existing `ReutersSource` class, i.e., `URI` for holding a reference to the input (e.g., a call to a file on disk) and `Content` to hold the XML tree of the RSS feed.

Next we can set up the constructor for the class `GmaneSource`:

```
> setMethod("GmaneSource",
+           signature(object = "ANY"),
+           function(object, encoding = "UTF-8") {
+               ## ---code chunk---
+               new("GmaneSource", LoDSupport = FALSE, URI = object,
+                   Content = content, Position = 0, Encoding = encoding)
+           })
```

where `--code chunk--` is a symbolic anonymous shorthand for reading in the RSS file, parsing it, e.g., with methods provided in the **XML** package, and filling the `content` variable with it.

Next we need to implement the three interface methods a source must provide:

```
> setMethod("stepNext",
+           signature(object = "GmaneSource"),
+           function(object) {
+               object@Position <- object@Position + 1
+               object
+           })
```

simply updates the position counter for using the next item in the XML tree,

```
> setMethod("getElem",
+           signature(object = "GmaneSource"),
+           function(object) {
```

```
+                 ## ---code chunk---
+                 list(content = content, uri = object@URI)
+             })
```

returns a list with the element's content at the active position (which is extracted in `--code chunk--`) and the corresponding unique resource identifier, and

```
> setMethod("eoi",
+             signature(object = "GmaneSource"),
+             function(object) {
+                 length(object@Content) <= object@Position
+             })
```

indicates the end of the XML tree.

Finally we write a custom reader function which extracts the relevant information out of RSS feeds:

```
> readGmane <- FunctionGenerator(function(...) {
+     function(elem, load, language, id) {
+         ## ---code chunk---
+         new("NewsgroupDocument", .Data = content, URI = elem$uri,
+             Cached = TRUE, Author = author, DateTimeStamp = datetimestamp,
+             Description = "", ID = id, Origin = origin, Heading = heading,
+             Language = language, Newsgroup = newsgroup)
+     }
+ })
```

The function shows how a custom `FunctionGenerator` can be implemented which returns the reader as return value. The reader itself extracts relevant information via XPath expressions in the function body's `--code chunk--` and returns a `NewsgroupDocument` as desired.

The full implementation comes with the **tm** package such that we can use the source and reader to access Gmane RSS feeds:

```
> rss <- system.file("texts", "gmane.comp.lang.r.gr.rdf", package = "tm")
> Corpus(GmaneSource(rss), readerControl = list(reader = readGmane,
+     language = "en_US", load = TRUE))

A text document collection with 21 text documents
```

Since we now have a grasp about necessary steps to extend the framework we want to show how easy it is to produce realistic readers by giving an actual implementation for a highly desired feature in the R community: a PDF reader. The reader expects the two command line tools `pdftotext` and `pdfinfo` installed to work properly (both programs are freely available for common operating systems, e.g., via the **poppler** or **xpdf** tool suites).

```
> readPDF <- FunctionGenerator(function(...) {
+   function(elem, load, language, id) {
+     ## get metadata
+     meta <- system(paste("pdfinfo", as.character(elem$uri[2])),
+                    intern = TRUE)
+
+     ## extract and store main information, e.g.:
+     heading <- gsub("Title:[[:space:]]*", "",
+                     grep("Title:", meta, value = TRUE))
+
+     ## [... similar for other metadata ...]
+
+     ## extract text from PDF using the external pdftotext utility:
+     corpus <- paste(system(paste("pdftotext", as.character(elem$uri[2]), "-"),
+                            intern = TRUE),
+                     sep = "\n", collapse = "")
+
+     ## create new text document object:
+     new("PlainTextDocument", .Data = corpus, URI = elem$uri, Cached = TRUE,
+         Author = author, DateTimeStamp = datetimestamp,
+         Description = description, ID = id, Origin = origin,
+         Heading = heading, Language = language)
+     }
+ })
```

Basically we use `pdfinfo` to extract the metadata, search the relevant tags for filling metadata slots, and use `pdftotext` for acquiring the text corpus.

We have seen extensions for classes, sources and readers. But we can also write custom transformation and filter functions. E.g., a custom generic transform function could look like

```
> setGeneric("myTransform", function(object, ...) standardGeneric("myTransform"))
> setMethod("myTransform", signature(object = "PlainTextDocument"),
+     function(object, ..., DMetaData) {
+         Content(object) <- doSomeThing(object, DMetaData)
+         return(object)
+     })
```

where we change the text corpus (i.e., the actual text) based on `doSomeThing`'s result and return the document again. In case of a filter function we would return a Boolean value.

Summarizing, this section showed that own fully functional classes, sources, readers, transformations and filters can be contributed simply by giving implementations for interface definitions.

Based on the presented framework and its algorithms the following sections will show how to use **tm** to ease text mining tasks in R.

5 Preprocessing

Input texts in their native raw format can be an issue when analyzing these with text mining methods since they might contain many unimportant stopwords (like `and` or `the`) or might be formatted inconveniently. Therefore preprocessing, i.e., applying methods for cleaning up and structuring the input text for further analysis, is a core component in practical text mining studies. In this section we will discuss how to perform typical preprocessing steps in the **tm** package.

5.1 Data Import

One very popular data set in text mining research is the Reuters-21578 data set (Lewis, 1997). It now contains over 20000 stories (the original version contained 21578 documents) from the Reuters news agency with metadata on topics, authors and locations. It was compiled by David Lewis in 1987, is publicly available and is still one of the most widely used data sets in recent text mining articles (see, e.g., Lodhi et al., 2002).

The original Reuters-21578 XML data set consists of a set of XML files with about 1000 articles per XML file. In order to enable load on demand the method `preprocessReut21578XML()` can be used to split the articles into separate files such that each article is stored in its own XML file. Reuters examples in the **tm** package and Reuters data sets used in this paper have already been preprocessed with this function.

Documents in the Reuters XML format can easily be read in with existing parsing functions

```
> reut21578XMLgz <- system.file("texts", "reut21578.xml.gz", package = "tm")
> (Reuters <- Corpus(ReutersSource(gzfile(reut21578XMLgz)),
+                    readerControl = list(reader = readReut21578XML,
+                                         language = "en_US",
+                                         load = TRUE)))

A text document collection with 10 text documents
```

Note that connections can be passed over to `ReutersSource`, e.g., we can compress our files on disk to save space without losing functionality. The package further supports Reuters Corpus Volume 1 (Lewis et al., 2004)—the successor of the Reuters-21578 data set—which can be similarly accessed via predefined readers (`readRCV1()`).

The default encoding used by sources is always assumed to be UTF-8. Anyway, one can manually set the encoding via the `encoding` parameter (e.g., `DirSource("texts/",`

encoding = "latin1")) or by creating a connection with an alternative encoding which is passed over to the source.

Since the documents are in XML format and we prefer to get rid of the XML tree and use the plain text instead we transform our collection with the predefined generic asPlain():

```
> tmMap(Reuters, asPlain)

A text document collection with 10 text documents
```

We then extract two subsets of the full Reuters-21578 data set by filtering out those with topics acq and crude. Since the Topics are stored in the LocalMetaData slot by readReut21578XML() the filtering can be easily accomplished e.g., via

```
> tmFilter(crude, FUN = sFilter, "Topics == 'crude'")
```

resulting in 50 articles of topic acq and 20 articles of topic crude. For further use as simple examples we provide these subsets in the package as separate data sets:

```
> data("acq")
> data("crude")
```

5.2 Stemming

Stemming is the process of erasing word suffixes to retrieve their radicals. It is a common technique used in text mining research, as it reduces complexity without any severe loss of information for typical applications (especially for bag-of-words).

One of the best known stemming algorithm goes back to Porter (1997) describing an algorithm that removes common morphological and inflectional endings from English words. The R **Rstem** and **Snowball** (encapsulating stemmers provided by **Weka**) packages implement such stemming capabilities and can be used in combination with our **tm** infrastructure. The main stemming function is wordStem(), which internally calls the Porter stemming algorithm, and can be used with several languages, like English, German or Russian (see e.g., **Rstem**'s getStemLanguages() for installed language extensions). A small wrapper in form of a transformation function handles internally the character vector conversions so that it can be directly applied to a text document. For example, given the corpus of the 10th acq document:

```
> acq[[10]]

[1] "Gulf Applied Technologies Inc said it sold its subsidiaries engaged in"
[2] "pipeline and terminal operations for 12.2 mln dlrs.  The company said"
[3] "the sale is subject to certain post closing adjustments, which it did"
[4] "not explain.  Reuter"
```

the same corpus after applying the stemming transformation reads:

```
> stemDoc(acq[[10]])

[1] "Gulf Appli Technolog Inc said it sold it subsidiari engag in pipelin"
[2] "and termin oper for 12.2 mln dlrs.  The compani said the sale is"
[3] "subject to certain post close adjustments, which it did not explain."
[4] "Reuter"
```

The result is the document where for each word the Porter stemming algorithm has been applied, that is we receive each word's stem with its suffixes removed.

This stemming feature transformation in **tm** is typically activated when creating a term-document matrix, but is also often used directly on the text documents before exporting them, e.g.,

```
> tmMap(acq, stemDoc)

A text document collection with 50 text documents
```

5.3 Whitespace Elimination and Lower Case Conversion

Another two common preprocessing steps are the removal of white space and the conversion to lower case. For both tasks **tm** provides transformations (and thus can be used with `tmMap()`)

```
> stripWhitespace(acq[[10]])

[1] "Gulf Applied Technologies Inc said it sold its subsidiaries engaged in"
[2] "pipeline and terminal operations for 12.2 mln dlrs. The company said"
[3] "the sale is subject to certain post closing adjustments, which it did"
[4] "not explain. Reuter"

> tmTolower(acq[[10]])

[1] "gulf applied technologies inc said it sold its subsidiaries engaged in"
[2] "pipeline and terminal operations for 12.2 mln dlrs.  the company said"
[3] "the sale is subject to certain post closing adjustments, which it did"
[4] "not explain.  reuter"
```

which are wrappers for simple `gsub` and `tolower` statements.

5.4 Stopword Removal

A further preprocessing technique is the removal of stopwords.

Stopwords are words that are so common in a language that their information value is almost zero, in other words their entropy is very low. Therefore it is usual to remove them before further analysis. At first we set up a tiny list of stopwords:

```
> mystopwords <- c("and", "for", "in", "is", "it", "not", "the", "to")
```

Stopword removal has also been wrapped as a transformation for convenience:

```
> removeWords(acq[[10]], mystopwords)
```

```
[1] "Gulf Applied Technologies Inc said sold its subsidiaries engaged"
[2] "pipeline terminal operations 12.2 mln dlrs.  The company said sale"
[3] "subject certain post closing adjustments, which did explain.  Reuter"
```

A whole collection can be transformed by using:

```
> tmMap(acq, removeWords, mystopwords)
```

For real application one would typically use a purpose tailored a language specific stopword list. The package **tm** ships with a list of Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Russian, Spanish, and Swedish stopwords, available via

```
> stopwords(language = ...)
```

For stopword selection one can either provide the full language name in lower case (e.g., `german`) or its ISO 639 code (e.g., `de` or even `de_AT`) to the argument `language`. Further, automatic stopword removal is available for creating term-document matrices, given a list of stopwords.

5.5 Synonyms

In many cases it is of advantage to know synonyms for a given term, as one might identify distinct words with the same meaning. This can be seen as a kind of semantic analysis on a very low level.

The well known WordNet database (Fellbaum, 1998), a lexical reference system, is used for many purposes in linguistics. It is a database that holds definitions and semantic relations between words for over 100,000 English terms. It distinguishes between nouns, verbs, adjectives and adverbs and relates concepts in so-called synonym sets. Those sets describe relations, like hypernyms, hyponyms, holonyms, meronyms, troponyms and synonyms. A word may occur in several synsets which means that it has several meanings. Polysemy counts relate synsets with the word's commonness in language use so that specific meanings can be identified.

One feature we actually use is that given a word, WordNet returns all synonyms in its database for it. For example we could ask the WordNet database via the **wordnet** package for all synonyms of the word `company`. At first we have to load the package and get a handle to the WordNet database, called dictionary:

```
> library("wordnet")
```

If the package has found a working WordNet installation we can proceed with

```
> synonyms("company")
```

```
[1] "caller"         "companionship"  "company"        "fellowship"
[5] "party"          "ship's company" "society"        "troupe"
```

giving us the synonyms.

Once we have the synonyms for a word a common approach is to replace all synonyms by a single word. This can be done via the `replaceWords()` transformation

```
> replaceWords(acq[[10]], synonyms(dict, "company"), by = "company")
```

and for the whole collection, using `tmMap()`:

```
> tmMap(acq, replaceWords, synonyms(dict, "company"), by = "company")
```

5.6 Part of Speech Tagging

In computational linguistics a common task is tagging words with their part of speech for further analysis. Via an interface with the **openNLP** package to the **OpenNLP** tool kit **tm** integrates part of speech tagging functionality based on maximum entropy machine learned models. **openNLP** ships transformations wrapping **OpenNLP**'s internal Java system calls for our convenience, e.g.,

```
> library("openNLP")
> tagPOS(acq[[10]])
```

```
[1] "Gulf/NNP Applied/NNP Technologies/NNPS Inc/NNP said/VBD it/PRP sold/VBD"
[2] "its/PRP$ subsidiaries/NNS engaged/VBN in/IN pipeline/NN and/CC"
[3] "terminal/NN operations/NNS for/IN 12.2/CD mln/NN dlrs./, The/DT"
[4] "company/NN said/VBD the/DT sale/NN is/VBZ subject/JJ to/TO certain/JJ"
[5] "post/NN closing/NN adjustments,/NN which/WDT it/PRP did/VBD not/RB"
[6] "explain./NN Reuter/NNP"
```

shows the tagged words using a set of predefined tags identifying nouns, verbs, adjectives, adverbs, et cetera depending on their context in the text. The tags are Penn Treebank tags (Mitchell et al., 1993), so e.g., `NNP` stands for *proper noun, singular*, or e.g., `VBD` stands for *verb, past tense*.

6 Basic Analysis Techniques

Now that we have learned in previous chapters about existing data structures and algorithms in our framework we want to actually perform analyses utilizing the presented infrastructure. So in this chapter we present common text mining techniques on texts, that is analysis based on counting frequencies, clustering and classification of texts, and show how they can be performed in our framework.

6.1 Count-Based Evaluation

One of the simplest analysis methods in text mining is based on count-based evaluation. This means that those terms with the highest occurrence frequencies in a text are rated important. In spite of its simplicity this approach is widely used in text mining (Davi et al., 2005) as it can be interpreted nicely and is computationally inexpensive.

At first we create a term-document matrix for the `crude` data set, where rows correspond to documents `ID`s and columns to terms. A matrix element contains the frequency of a specific term in a document. English stopwords are removed from the matrix (it suffices to pass over `TRUE` to `stopwords` since the function looks up the language in each text document and loads the right stopwords automagically)

```
> crudeTDM <- TermDocMatrix(crude, control = list(stopwords = TRUE))
```

Then we use a function on term-document matrices that returns terms that occur at least `freq` times. For example we might choose those terms from our `crude` term-document matrix which occur at least 10 times

```
> (crudeTDMHighFreq <- findFreqTerms(crudeTDM, 10, Inf))

[1] "oil"    "opec"   "kuwait"
```

Conceptually, we interpret a term as important according to a simple counting of frequencies. As we see the results can be interpreted directly and seem to be reasonable in the context of texts on crude oil (like `opec` or `kuwait`). We can also apply this function to see an excerpt (here the first 10 rows) of the whole (sparse compressed) term-document matrix, i.e., we also get the frequencies of the high occurrence terms for each document:

```
> Data(crudeTDM)[1:10, crudeTDMHighFreq]

10 x 3 sparse Matrix of class "dgCMatrix"
    oil opec kuwait
```

```
127   5    .       .
144  12   15       .
191   2    .       .
194   1    .       .
211   1    .       .
236   7    8      10
237   4    1       .
242   3    2       1
246   5    2       .
248   9    6       3
```

Another approach available in common text mining tools is finding associations for a given term, which is a further form of count-based evaluation methods. This is especially interesting when analyzing a text for a specific purpose, e.g., a business person could extract associations of the term "oil" from the Reuters articles.

Technically we can realize this in R by computing correlations between terms. We have prepared a function `findAssocs()` which computes all associations for a given `term` and `corlimit`, that is the minimal correlation for being identified as valid associations. The example finds all associations for the term "oil" with at least 0.85 correlation in the term-document matrix:

```
> findAssocs(crudeTDM, "oil", 0.85)

 oil opec
1.00 0.87
```

Internally we compute the correlations between all terms in the term-document matrix and filter those out higher than the correlation threshold.

Figure 6.1 shows a plot of the term-document matrix `crudeTDM` which visualizes the correlations over 0.5 between frequent (co-occurring at least 6 times) terms.

Conceptually, those terms with high correlation to the given term `oil` can be interpreted as its valid associations. From the example we can see that `oil` is highly associated with `opec`, which is quite reasonable. As associations are based on the concept of similarities between objects, other similarity measures could be used. We use correlations between terms, but theoretically we could use any well defined similarity function (confer to the discussion on the `dissimilarity()` function in the next section) for comparing terms and identifying similar ones. Thus the similarity measures may change but the idea of interpreting similar objects as associations is general.

6.2 Simple Text Clustering

In this section we will discuss classical clustering algorithms applied to text documents. For this we combine our known `acq` and `crude` data sets to a single working set `ws` in order to use it as input for several simple clustering methods

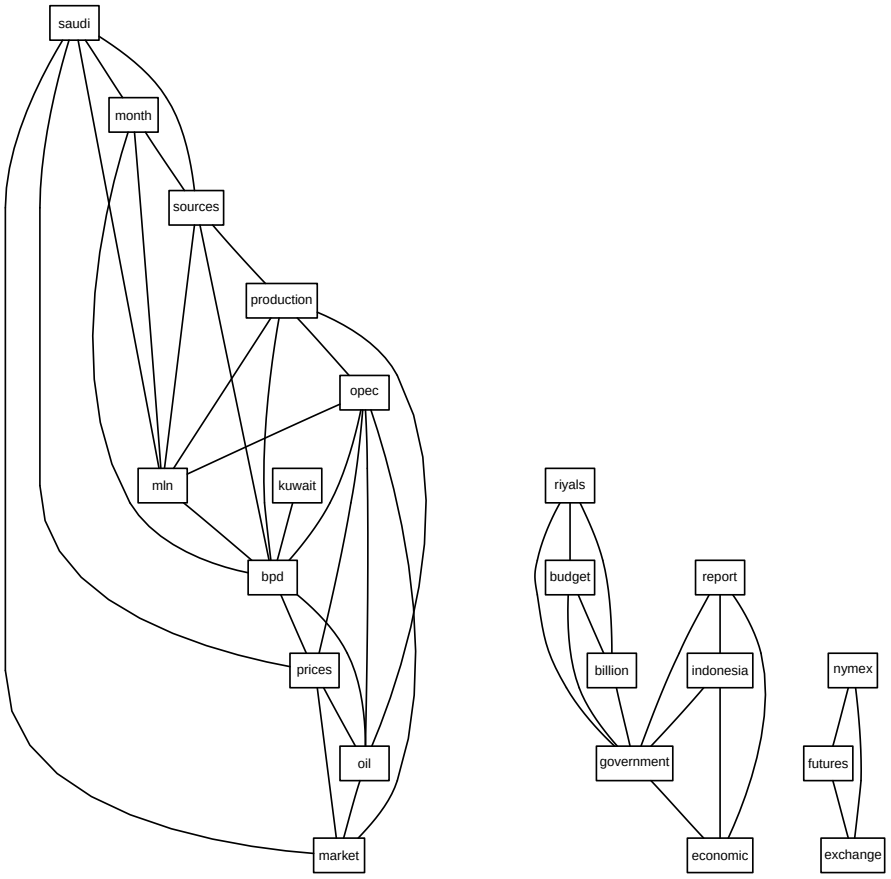

Figure 6.1: Visualization of the correlations within a term-document matrix.

```
> ws <- c(acq, crude)
> summary(ws)

A text document collection with 70 text documents

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  merge_date merger
Available variables in the data frame are:
  MetaID
```

6.2.1 Hierarchical Clustering

Here we show hierarchical clustering (Johnson, 1967; Hartigan, 1975; Anderberg, 1973; Hartigan, 1972) with text documents. Clearly, the choice of the distance measure significantly influences the outcome of hierarchical clustering algorithms. Common similarity

measures in text mining are Metric Distances, Cosine Measure, Pearson Correlation and Extended Jaccard Similarity (Strehl et al., 2000). We use the similarity measures offered by `dist` from package **proxy** (Meyer and Buchta, 2007) in our **tm** package with a generic custom distance function `dissimilarity()` for term-document matrices. So we could easily use as distance measure the Cosine for our `crude` term-document matrix

```
> dissimilarity(crudeTDM, method = "cosine")
```

Our dissimilarity function for text documents takes as input two text documents. Internally this is done by a reduction to two rows in a term-document matrix and applying our custom distance function. For example we could compute the Cosine dissimilarity between the first and the second document from our `crude` collection

```
> dissimilarity(crude[[1]], crude[[2]], "cosine")
```

```
          127
144 0.4425716
```

In the following example we create a term-document matrix from our working set of 70 news articles (`Data()` accesses the slot holding the actual sparse matrix)

```
> wsTDM <- Data(TermDocMatrix(ws))
```

and use the Euclidean distance metric as distance measure for hierarchical clustering with Ward's minimum variance method of our 50 `acq` and 20 `crude` documents:

```
> wsHClust <- hclust(dist(wsTDM), method = "ward")
```

Figure 6.2 visualizes the hierarchical clustering in a dendrogram. It shows two bigger conglomerations of `crude` aggregations (one left, one at right side). In the middle we can find a big `acq` aggregation.

6.2.2 k-means Clustering

We show how to use a classical k-means algorithm (Hartigan and Wong, 1979; MacQueen, 1967), where we use the term-document matrix representation of our news articles to provide valid input to existing methods in R. We perform a classical linear k-means clustering with $k = 2$ (we know that only two clusters is a reasonable value because we concatenated our working set of the two topic sets `acq` and `crude`)

```
> wsKMeans <- kmeans(wsTDM, 2)
```

and present the results in form of a confusion matrix. We use as input both the clustering result and the original clustering according to the Reuters topics. As we know the working set consists of 50 `acq` and 20 `crude` documents

```
> wsReutersCluster <- c(rep("acq", 50), rep("crude", 20))
```

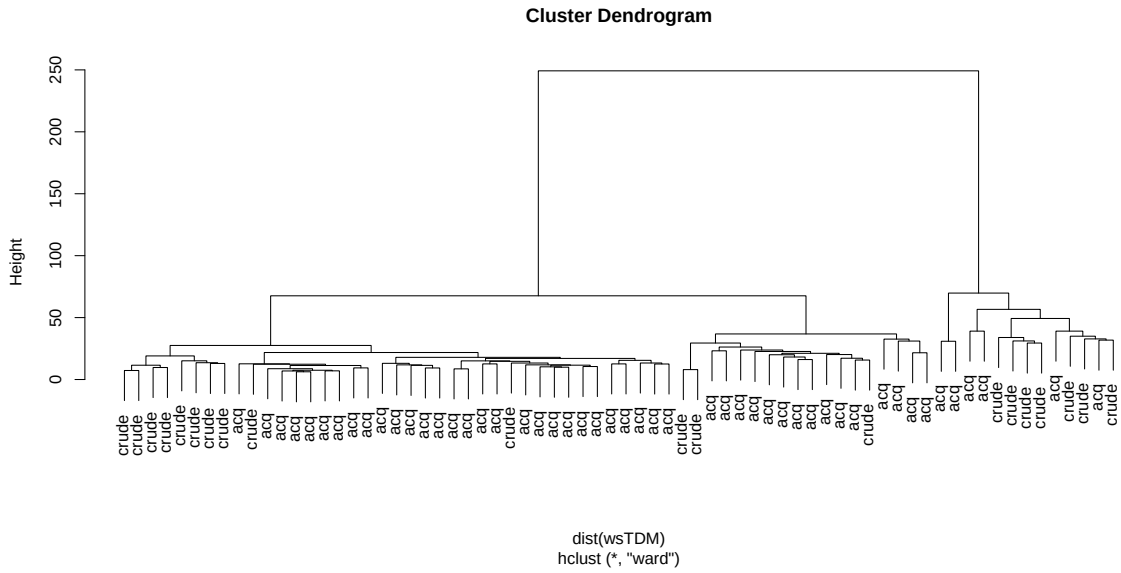

Figure 6.2: Dendrogram for hierarchical clustering. The labels show the original group names.

Using the function `cl_agreement()` from package **clue** (Hornik, 2005, 2007a), we can compute the maximal co-classification rate, i.e., the maximal rate of objects with the same class ids in both clusterings—the 2-means clustering and the topic clustering with the Reuters `acq` and `crude` topics—after arbitrarily permuting the ids:

```
> cl_agreement(wsKMeans, as.cl_partition(wsReutersCluster), "diag")

Cross-agreements using maximal co-classification rate:
     [,1]
[1,]  0.7
```

which means that the k-means clustering results can recover about 70 percent of the human clustering.

For a real-world example on text clustering for the **tm** package with several hundreds of documents confer to Karatzoglou and Feinerer (2007) who illustrate that text clustering with a decent amount of documents works reasonably well.

6.3 Simple Text Classification

In contrast to clustering, where groups are unknown at the beginning, classification tries to put specific documents into groups known in advance. Nevertheless the same basic means can be used as in clustering, like bag-of-words representation as a way to formalize unstructured text. Typical real-world examples are spam classification of e-mails or classifying news articles into topics. In the following, we give two examples: first, a very simple classifier (k-nearest neighbor), and then a more advanced method (Support Vector Machines).

6.3.1 k-nearest Neighbor Classification

Similar to our examples in the previous section we will reuse the term-document matrix representation, as we can easily access already existing methods for classification. A possible classification procedure is k-nearest neighbor classification implemented in the **class** (Venables and Ripley, 2002) package. The following example shows a 1-nearest neighbor classification in a spam detection scenario. We use the Spambase database from the Uci Machine Learning Repository (Asuncion and Newman, 2007) which consists of 4601 instances representing `spam` and `nonspam` e-mails. Technically this data set is a term-document matrix with a limited set of terms (in fact 57 terms with their frequency in each e-mail document). Thus we can easily bring text documents into this format by projecting our term-document matrices onto their 57 terms. We start with a training set with about 75 percent of the spam data set resulting in about 1360 `spam` and 2092 `nonspam` documents

```
> train <- rbind(spam[1:1360, ], spam[1814:3905, ])
```

and tag them as factors according to our know topics (the last column in this data set holds the `type`, i.e., `spam` or `nonspam`):

```
> trainCl <- train[, "type"]
```

In the same way we take the remaining 25 percent of the data set as fictive test sets

```
> test <- rbind(spam[1361:1813, ], spam[3906:4601, ])
```

and store their original classification

```
> trueCl <- test[, "type"]
```

Note that the training and test sets were chosen arbitrarily, but fixed for reproducibility. Finally we start the 1-nearest neighbor classification (deleting the original classification from column 58, which represents the `type`):

```
> knnCl <- knn(train[, -58], test[, -58], trainCl)
```

and obtain the following confusion matrix

```
> (nnTable <- table("1-NN" = knnCl, Reuters = trueCl))
```

```
         Reuters
1-NN      nonspam spam
  nonspam     502  139
  spam        194  314
```

As we see the results are already quite promising—the cross-agreement is

```
> sum(diag(nnTable))/nrow(test)
```

```
[1] 0.7101828
```

6.3.2 Support Vector Machine Classification

Another typical, but more sophisticated, classification method are support vector machines (Cristianini and Shawe-Taylor, 2000).

The following example shows an SVM classification based on methods from the **kernlab** package. We used the same training and test documents. Based on the training data and its classification we train a support vector machine:

```
> ksvmTrain <- ksvm(type ~ ., data = train)

Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

Then we classify the test set based on the created SVM (again, we omit the original classification from column 58 which represents the `type`)

```
> svmCl <- predict(ksvmTrain, test[, -58])
```

which yields the following confusion matrix

```
> (svmTable <- table(SVM = svmCl, Reuters = trueCl))

         Reuters
SVM       nonspam spam
  nonspam     635   80
  spam         61  373
```

with following cross-agreement:

```
> sum(diag(svmTable))/nrow(test)

[1] 0.8772846
```

The results have improved over those in the last section (compare the improved cross-agreement) and prove the viability of support vector machines for classification.

Though, we use a realistic data set and gain rather good results, this approach is not competitive with available contemporary spam detection methods. The main reason is that spam detection nowadays encapsulates techniques far beyond analysing the corpus itself. Methods encompass mail format detection (e.g., HTML or ASCII text), black- (spam) and whitelists (ham), known spam IP addresses, distributed learning systems (several mail servers communicating their classifications), attachment analysis (like type and size), and social network analysis (web of trust approach).

With the same approach as shown before, it is easy to perform classifications on any other form of text, like classifying news articles into predefined topics, using any available classifier as suggested by the task at hand.

6.4 Text Clustering with String Kernels

This section covers string kernels, which are methods dealing with text directly, and not anymore with an intermediate representation like term-document matrices.

Kernel-based clustering methods, like kernel k-means, use an implicit mapping of the input data into a high dimensional feature space defined by a kernel function k

$$k(x, y) = \langle \Phi(x), \Phi(y) \rangle \ ,$$

with the projection $\Phi \colon X \to H$ from the input domain X to the feature space H. In other words this is a function returning the inner product $\langle \Phi(x), \Phi(y) \rangle$ between the images of two data points x and y in the feature space. All computational tasks can be performed in the feature space if one can find a formulation so that the data points only appear inside inner products. This is often referred to as the "kernel trick" (Schölkopf and Smola, 2002) and is computationally much simpler than explicitly projecting x and y into the feature space H. The main advantage is that the kernel computation is by far less computationally expensive than operating directly in the feature space. This allows one to work with high-dimensional spaces, including natural texts, typically consisting of several thousand term dimensions.

String kernels (Lodhi et al., 2002; Shawe-Taylor and Cristianini, 2004; Watkins, 2000; Herbrich, 2002) are defined as a similarity measure between two sequences of characters x and y. The generic form of string kernels is given by the equation

$$k(x, y) = \sum_{s \sqsubseteq x, t \sqsubseteq y} \lambda_s \delta_{s,t} = \sum_{s \in \Sigma^*} \mathrm{num}_s(x) \mathrm{num}_s(y) \lambda_s \ ,$$

where Σ^* represents the set of all strings, $\mathrm{num}_s(x)$ denotes the number of occurrences of s in x and λ_s is a weight or decay factor which can be chosen to be fixed for all substrings or can be set to a different value for each substring. This generic representation includes a large number of special cases, e.g., setting $\lambda_s \neq 0$ only for substrings that start and end with a white space character gives the "bag of words" kernel (Joachims, 2002). Special cases are $\lambda_s = 0$ for all $|s| > n$, that is comparing all substrings of length less that n, often called full string kernel. The case $\lambda_s = 0$ for all $|s| \neq n$ is often referred to as string kernel.

A further variation is the string subsequence kernel

$$\begin{aligned} k_n(s, t) &= \sum_{u \in \Sigma^n} \langle \phi_u(s), \phi_u(t) \rangle \\ &= \sum_{u \in \Sigma^n} \sum_{\mathbf{i} : u = s[\mathbf{i}]} \lambda^{l(\mathbf{i})} \sum_{\mathbf{j} : u = t[\mathbf{j}]} \lambda^{l(\mathbf{j})} \\ &= \sum_{u \in \Sigma^n} \sum_{\mathbf{i} : u = s[\mathbf{i}]} \sum_{\mathbf{j} : u = t[\mathbf{j}]} \lambda^{l(\mathbf{i}) + l(\mathbf{j})} \ , \end{aligned}$$

where k_n is the subsequence kernel function for strings up to the length n, s and t denote two strings from Σ^n, the set of all finite strings of length n, and $|s|$ denotes

the length of s. u is a subsequence of s, if there exist indices $\mathbf{i} = (i_1, \ldots, i_{|u|})$, with $1 \leq i_1 < \cdots < i_{|u|} \leq |s|$, such that $u_j = s_{i_j}$, for $j = 1, \ldots, |u|$, or $u = s[\mathbf{i}]$ for short. $\lambda \leq 1$ is a decay factor.

A very nice property is that one can find a recursive formulation of the above kernel

$$
\begin{aligned}
k'_0(s,t) &= 1, \text{ for all } s,t, \\
k'_i(s,t) &= 0, \text{ if } \min(|s|,|t|) < i, \\
k_i(s,t) &= 0, \text{ if } \min(|s|,|t|) < i, \\
k'_i(sx,t) &= \lambda k'_i(s,t) + \sum_{j:t_j=x} k'_{i-1}(s, t[1:(j-1)])\lambda^{|t|-j+2} \text{ , with } i = 1, \ldots, n-1, \\
k_n(sx,t) &= k_n(s,t) + \sum_{j:t_j=x} k'_{n-1}(s, t[1:(j-1)])\lambda^2 \text{ ,}
\end{aligned}
$$

which can be used for dynamic programming aspects to speed up computation significantly.

Further improvements for string kernel algorithms are specialized formulations using suffix trees (Vishwanathan and Smola, 2004) and suffix arrays (Teo and Vishwanathan, 2006).

Several string kernels with above explained optimizations (dynamic programming) have been implemented in the **kernlab** package (Karatzoglou et al., 2004, 2006) and been used by Karatzoglou and Feinerer (2007). The interaction between **tm** and **kernlab** is easy and fully functional, as the string kernel clustering constructors can directly use the base classes from the **tm** classes. This proves that the S4 extension mechanism can be used effectively by passing only necessary information to external methods (i.e., the string kernel clustering constructors in this context) and still handle detailed meta information internally (i.e., the native text mining classes).

The following examples show an application of spectral clustering (Ng et al., 2002; Dhillon et al., 2005), which is a non-linear clustering technique using string kernels. We create a string kernel for it

```
> stringkern <- stringdot(type = "string")
```

and perform a spectral clustering with the string kernel on the working set. We specify that the working set should be clustered into 2 different sets (simulating our two original topics). One can see the clustering result with the string kernel

```
> stringCl <- specc(ws, 2, kernel = stringkern)

String kernel function.  Type =  string
 Hyperparameters : sub-sequence/string length =  4  lambda =  0.5
 Normalized
```

compared to the original Reuters clusters

```
> table("String Kernel" = stringCl, Reuters = wsReutersCluster)
```

```
             Reuters
String Kernel acq crude
            1  46     1
            2   4    19
```

This method yields the best results (the cross-agreement is 0.93) as we almost find the identical clustering as the Reuters employees did manually. This well performing method has been investigated by Karatzoglou and Feinerer (2007) and seems to be a generally viable method for text clustering.

Part II

Applications

7 Analysis of the R-devel 2006 mailing list

This chapter shows the application of the **tm** package to perform an analysis of the R-devel mailing list (https://stat.ethz.ch/pipermail/r-devel/) newsgroup postings from 2006. We will both show to utilize metadata and the text corpora themselves. For the first we analyze author and topic relations whereas for the second we concentrate on investigating the e-mail contents and discriminative terms (e.g., match the e-mail subjects the actual content).

7.1 Data

The mailing list archive provides downloadable versions in gzipped mbox format. We downloaded the twelve archives from January until December 2006, unzipped them and concatenated them to a single mbox file `2006.txt` for convenience. The mbox file holds 4583 postings with a file size of about 12 Megabyte.

We start by converting the single mbox file to eml format, i.e., every newsgroup posting is stored in a single file in the directory `2006/`.

```
> convertMboxEml("2006.txt", "2006/")
```

Next, we construct a text document collection holding the newsgroup postings, using the default reader shipped for newsgroups (`readNewsgroup()`), and setting the language to American English. For the majority of postings this assumption is reasonable but we plan automatic language detection (Sibun and Reynar, 1996) for future releases, e.g., by using n-grams (Cavnar and Trenkle, 1994). So you can either provide a string (e.g., `en_US`) or a function returning a character vector (a function determining the language) to the `language` parameter. Next, we want the the postings immediately loaded into memory (`load = TRUE`)

```
> rdevel <- Corpus(DirSource("2006/"),
+                  readerControl = list(reader = readNewsgroup,
+                                       language = "en_US",
+                                       load = TRUE))
```

We convert the newsgroup documents to plain text documents since we have no need for specific slots of class `NewsgroupDocument` (like the `Newsgroup` slot, as we only have R-devel here) in this analysis.

```
> rdevel <- tmMap(rdevel, asPlain)
```

White space is removed and a conversion to lower case is performed.

```
> rdevel <- tmMap(rdevel, stripWhitespace)
> rdevel <- tmMap(rdevel, tmTolower)
```

After preprocessing we have

```
> summary(rdevel)

A text document collection with 4583 text documents

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator
Available variables in the data frame are:
  MetaID
```

We create a term-document matrix, activate stemming and remove stopwords.

```
> tdm <- TermDocMatrix(rdevel, list(stemming = TRUE, stopwords = TRUE))
```

The computation takes about 7 minutes on a 3.4 GHz processor resulting in a $4,583 \times 29,265$ dimensioned matrix. A dense matrix would require about 4 Gigabyte RAM ($4,583 \cdot 29,265 \cdot 32/1,024^3$), the sparse compressed S4 matrix instead requires only 6 Megabyte RAM as reported by `object.size()`. The reason is the extremely sparse internal structure since most combinations of documents and terms are zero. Besides using sparse matrices another common approach for handling the memory problem is to reduce the number of terms dramatically. This can be done via tabulating against a given dictionary (e.g., by using the `dictionary` argument of `TermDocMatrix()`). In addition a well defined dictionary helps in identifying discriminative terms tailored for specific analysis contexts.

7.2 Finding Most Active Authors, Threads, and Cliques

Let us start finding out the three most active authors. We extract the author information from the text document collection, and convert it to a correctly sized character vector (since some author tags may contain more than one line):

```
> authors <- lapply(rdevel, Author)
> authors <- sapply(authors, paste, collapse = " ")
```

Thus, the sorted contingency table gives us the author names and the number of their postings (under the assumption that authors use consistently the same e-mail addresses):

```
> sort(table(authors), decreasing = TRUE)[1:3]
```

```
authors
   ripley at stats.ox.ac.uk (Prof Brian Ripley)
                                                483
       murdoch at stats.uwo.ca (Duncan Murdoch)
                                                305
ggrothendieck at gmail.com (Gabor Grothendieck)
                                                190
```

Next, we identify the three most active threads, i.e., those topics with most postings and replies. Similarly, we extract the thread name from the text document collection:

```
> headings <- lapply(rdevel, Heading)
> headings <- sapply(headings, paste, collapse = " ")
```

The sorted contingency table shows the biggest topics' names and the amount of postings

```
> (bigTopicsTable <- sort(table(headings), decreasing = TRUE)[1:3])
> bigTopics <- names(bigTopicsTable)

headings
                                  [Rd] 'CanMakeUseOf' field
                                                         46
[Rd] how to determine if a function's result is invisible
                                                         33
                             [Rd] attributes of environments
                                                         24
```

Since we know the most active threads, we might be interested in cliques communicating in these three threads. For the first topic "[Rd] 'CanMakeUseOf' field" we have

```
> topicCol <- rdevel[headings == bigTopics[1]]
> unique(sapply(topicCol, Author))

[1] "sfalcon at fhcrc.org (Seth Falcon)"
[2] "murdoch at stats.uwo.ca (Duncan Murdoch)"
[3] "pgilbert at bank-banque-canada.ca (Paul Gilbert)"
[4] "friedrich.leisch at stat.uni-muenchen.de (friedrich.leisch at"
[5] "stat.uni-muenchen.de)"
[6] "maechler at stat.math.ethz.ch (Martin Maechler)"
[7] "Kurt.Hornik at wu-wien.ac.at (Kurt Hornik)"
```

whereas for the second topic "[Rd] how to determine if a function's result is invisible" we obtain

```
> topicCol <- rdevel[headings == bigTopics[2]]
> unique(sapply(topicCol, Author))
```

```
[1] "ggrothendieck at gmail.com (Gabor Grothendieck)"
[2] "MSchwartz at mn.rr.com (Marc Schwartz)"
[3] "deepayan.sarkar at gmail.com (Deepayan Sarkar)"
[4] "murdoch at stats.uwo.ca (Duncan Murdoch)"
[5] "phgrosjean at sciviews.org (Philippe Grosjean)"
[6] "bill at insightful.com (Bill Dunlap)"
[7] "jfox at mcmaster.ca (John Fox)"
[8] "luke at stat.uiowa.edu (Luke Tierney)"
[9] "mtmorgan at fhcrc.org (Martin Morgan)"
```

7.3 Identifying Relevant Terms

R-devel describes its focus on proposals of new functionality, pre-testing of new versions, and bug reports. Let us find out how many postings deal with bug reports in that sense that `bug` appears in the message body (but e.g., not `debug`, note the regular expression).

```
> (bugCol <- tmFilter(rdevel,
+                     FUN = searchFullText, "[^[:alpha:]]+bug[^[:alpha:]]+"))

A text document collection with 796 text documents
```

The most active authors in that context are

```
> bugColAuthors <- lapply(bugCol, Author)
> bugColAuthors <- sapply(bugColAuthors, paste, collapse = " ")
> sort(table(bugColAuthors), decreasing = TRUE)[1:3]

bugColAuthors
ripley at stats.ox.ac.uk (Prof Brian Ripley)
                                          88
    murdoch at stats.uwo.ca (Duncan Murdoch)
                                          66
p.dalgaard at biostat.ku.dk (Peter Dalgaard)
                                          48
```

In the context of this analysis we consider some discriminative terms known a priori, e.g., above mentioned term `bug`, but in general we are interested in a representative set of terms from our texts. The challenge is to identify such representative terms: one approach is to consider medium frequent terms, since low frequent terms only occur in a few texts, whereas highly frequent terms have similar properties as stopwords (since they occur almost everywhere). The frequency range differs for each application but for our example we take values around 30, since smaller values for this corpus tend to be already negligible due to the large number of documents. On the other side bigger values tend to be too common in most of the newsgroup postings. In detail, the

function `findFreqTerms` finds terms in the frequency range given as parameters (30–31). The `grep` statement just removes terms with numbers in it which do not make sense in this context.

```
> f <- findFreqTerms(tdm, 30, 31)
> sort(f[-grep("[0-9]", f)])
```

```
 [1] "andrew"    "ani"       "binomi"    "breakpoint"  "brob"
 [6] "cach"      "char"      "check"     "coil"        "const"
[11] "distanc"   "document"  "env"       "error"       "famili"
[16] "function"  "gcc"       "gengc"     "giochannel"  "gkeyfil"
[21] "glm"       "goodrich"  "home"      "int"         "kevin"
[26] "link"      "method"    "name"      "node"        "packag"
[31] "param"     "pas"       "prefix"    "probit"      "rossb"
[36] "saint"     "sctest"    "suggest"   "thunk"       "tobia"
[41] "tripack"   "tube"      "uuidp"     "warn"
```

Some terms tend to give us hints on the content of our documents, others seem to be rather alien. Therefore we decide to revise the document corpora in the hope to get better results: at first we take only the incremental part of each e-mail (i.e., we drop referenced text passages) to get rid of side effects by referenced documents which we analyze anyway, second we try to remove authors' e-mail signatures. For this purpose we create a transformation to remove citations and signatures.

```
> setGeneric("removeCitationSignature",
+            function(object, ...) standardGeneric("removeCitationSignature"))
```

```
[1] "removeCitationSignature"
```

```
> setMethod("removeCitationSignature",
+           signature(object = "PlainTextDocument"),
+           function(object, ...) {
+               c <- Content(object)
+
+               ## Remove citations starting with '>'
+               citations <- grep("^[[:blank:]]*>.*", c)
+               if (length(citations) > 0)
+                   c <- c[-citations]
+
+               ## Remove signatures starting with '-- '
+               signatureStart <- grep("^-- $", c)
+               if (length(signatureStart) > 0)
+                   c <- c[-(signatureStart:length(c))]
+
+               Content(object) <- c
+               return(object)
+           })
```

```
[1] "removeCitationSignature"
```

Next, we apply the transformation to our text document collection

```
> rdevelInc <- tmMap(rdevel, removeCitationSignature)
```

and create a term-document matrix from the collection holding only the incremental parts of the original newsgroup postings.

```
> tdmInc <- TermDocMatrix(rdevelInc, list(stemming = TRUE, stopwords = TRUE))
```

Now we repeat our attempt to find frequent terms.

```
> f <- findFreqTerms(tdmInc, 30, 31)
> sort(f[-grep("[0-9]", f)])
```

```
 [1] "ani"       "binomi"  "breakpoint"  "cach"     "char"
 [6] "check"     "coil"    "const"       "davidb"   "dosa"
[11] "download"  "duncan"  "env"         "error"    "famili"
[16] "gcc"       "gengc"   "giochannel"  "gkeyfil"  "glm"
[21] "home"      "int"     "link"        "node"     "param"
[26] "pas"       "probit"  "saint"       "tama"     "thunk"
[31] "tube"      "uuidp"
```

We see the output is smaller, especially there are terms removed that have originally occurred only because they have been referenced several times. Nevertheless, for significant improvements a separation between pure text, program code, signatures and references is necessary, e.g., by XML metadata tags identifying different constructs.

Another approach for identifying relevant terms are the subject headers of e-mails which are normally created manually by humans. As a result subject descriptions might not be very accurate, especially for longer threads. For the R-devel mailing list we can investigate whether subjects match the contents.

```
> subjectCounts <- 0
> for (r in rdevelInc) {
+     ## Get single characters from subject
+     h <- unlist(strsplit(Heading(r), " "))
+
+     ## Count unique matches of subject strings within document
+     len <- length(unique(unlist(lapply(h, grep, r, fixed = TRUE))))
+
+     ## Update counter
+     subjectCounts <- c(subjectCounts, len)
+ }
> summary(subjectCounts)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.000   1.000   3.000   6.053   7.000 515.000
```

We see, the mean is about 6, i.e., on average terms from the subject occur six (different) times in the document corpus. The maximum value of 515 can be explained with very short subjects or single letters (e.g., only `R`) which are found in abnormally many words.

Summarizing we see that text mining can be a highly complex task and might need a lot of manual interaction where a convenient framework like **tm** is very helpful.

8 Text Mining for B2C E-Commerce

During the last decade the amount of electronic commerce (*e-commerce*) has undergone a considerable rise in terms of generated revenue, market penetration, and technology. E-commerce extensively uses internet technology to substitute and extend classical means of consumer attraction, consumer communication and market making. This combination contributes to the growing interest of text mining in the e-commerce field: most documents in the internet still lack coherent information about its meta structure (i.e., there is still a long way towards the *semantic web*) and individual consumer communication is still predominated by e-mail requests. So we end up with a huge amount of text documents containing business relevant information, but far too many to be investigated and analyzed manually. This is especially true in the *business to consumer* (*B2C*) context since communication is less formalized and standardized compared to business to business models with a strong interest of companies for information integration.

The main factors influencing e-commerce (Hansen and Neumann, 2005) can be divided into strategic and operational means. We will mainly discuss text mining for the latter since it has greater potential for automatic text investigation. Instead strategic decisions need deeper investigation as they base for long-term business operation.

As known from classical marketing science four main key areas can be listed for the e-commerce sector:

- product and its information technology (IT) support,
- price and its IT support,
- place and its IT support,
- promotion and its IT support.

These are commonly denoted as the *four Ps* or the *marketing mix* (McCarthy, 1960; Kotler and Keller, 2008). We will discuss how text mining can help to cover these main aspects in a company's operational business.

We identify great potential for text mining assistance especially in two key areas: *product* and *promotion* with their IT support. Thus, we will concentrate our discussions on these two areas. Text mining approaches within this context provide aid for gathering and semi-automatically analyzing valuable information in the fields:

- product development,
- product improvement,
- quality assurance,

- consumer feedback, and
- advertising.

The following sections will show how text mining can assist in these areas. We will both discuss some theoretical background, and give examples for demonstration.

8.1 Product

The notion product and its IT support subsumes various operational fields in the e-commerce context: product choice, product differentiation, product individualization, product development, recommender systems, customer service, and many more.

Especially product development and customer service are typically confronted with a lot of consumer texts dealing with customers' opinions, requests, complaints, ideas and questions. In daily e-commerce business these texts originate from user e-mails, forum postings, blogs, and web forms. So there is great potential for text mining to extract valuable information semi-automatically.

For demonstration we will perform an electronic commerce analysis by investigating a user forum for Plone (http://plone.org/). Plone is an open source content management system, i.e., you can manage web sites, their content and their integration via a sophisticated system using databases and modularized plug-ins. Plone was chosen as its forum postings cover diverse consumer and developer e-mails dealing with various aspects of this product, like requests, announcements, or questions. Second, it is freely publicly available via the Gmane mailing list archive and search engine. The purpose of this analysis is to show how to actually extract information to be used for product development and customer service. In detail, we are interested in very frequent terms, since they often point us to topics relevant for a lot of consumers. Next, associations to given terms are rewarding as they can connect emotions with products. This gives us the chance to investigate consumers' associations for features, problems, or competitive products. Then we present another approach useful for dividing the material into distinct groups so that manual investigation is easier (e.g., in e-mails dealing with problems or feature requests).

8.1.1 Preparation

We downloaded an archive of the forum via Gmane. Technically Gmane delivers a mbox file (`gmane.comp.web.zope.plone.user`) which we convert to single eml files such that each posting is in its own file. This allows us easier handling.

```
> convertMboxEml("gmane.comp.web.zope.plone.user", "plone/")
```

We create a collection of plain text documents from the `plone` directory so that we can work with the postings in R.

```
> plone <- Corpus(DirSource("plone/"),
+                 readerControl = list(reader = readNewsgroup))
> plone <- tmMap(plone, asPlain)
```

The constructed collection has 999 documents holding the forum postings.

```
> summary(plone)

A text document collection with 999 text documents

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator
Available variables in the data frame are:
  MetaID
```

For computational reasons (many following analysis methods need well-formed matrices for operation) we create a term-document matrix, using only terms which occur at least two times in a document. We skip terms with numbers in them (since numbers, like years or quantities, are not of interest for us at the moment), stem the terms to remove their suffixes, and utilize an English stopword list.

```
> params <- list(minDocFreq = 2,
+                removeNumbers = TRUE,
+                stemming = TRUE,
+                stopwords = TRUE)

> tdm <- TermDocMatrix(plone, control = params)
```

8.1.2 Frequent Terms

Our first attempt is to find relevant terms. Relevant always depends on the context but in general frequent terms have the potential to be rather important. Nevertheless, very frequent terms are typically stopwords with a low information entropy. Therefore we chose terms occurring between 20 and 25 times, a value working well for this example.

```
> sort(findFreqTerms(tdm, 20, 25))

 [1] "app"           "blank"         "call"          "cmfformcontrol"
 [5] "com"           "compon"        "content"       "context"
 [9] "control"       "div"           "event"         "famili"
[13] "file"          "five"          "folder"        "font"
[17] "gmane"         "head"          "href"          "http"
[21] "hwl"           "jami"          "ldap"          "ldapuserfold"
[25] "line"          "list"          "michael"       "modul"
[29] "nbsp"          "net"           "normalizearg"  "onclick"
```

```
[33] "openextlink"    "openid"         "org"            "packag"
[37] "pagetempl"      "pars"           "plone"          "portlet"
[41] "product"        "program"        "properti"       "public"
[45] "python"         "qseoptim"       "quot"           "return"
[49] "role"           "site"           "skin"           "sprint"
[53] "style"          "talinterpret"   "target"         "textindexng"
[57] "top"            "traceback"      "user"           "verdana"
[61] "wichert"        "window"         "zodb"
```

We see that the output is intermixed with rather cumbersome terms like `gmane`, `verdana`, or `nbsp` which typically occur in HTML encoded postings. To improve the corpus quality we can choose various approaches. We can try to extract the text parts of multipart messages to avoid non plain text passages, we can remove citations to avoid duplicates, and we can delete signatures containing personal information not related to the actual content from the forum postings. All three functions are already provided by the **tm** package, via the `removeMultipart()`, `removeCitation()`, and `removeSignature()` functions. Additionally we specify further non-standard patterns for signature recognition for `removeSignature()` with regular pattern expressions. E.g., `^-{10,}` means that each line beginning with at least ten `-` characters is treated as a signature mark. Similarly we define this behavior for `_` and `*` characters.

```
> plone <- tmMap(plone, removeMultipart)
> plone <- tmMap(plone, removeCitation)
> plone <- tmMap(plone, removeSignature,
+                c("^_{10,}", "^-{10,}", "^[*]{10,}"))
```

Technically we apply the three transformations on our text document collection and recreate a term-document matrix with the same parameters as before from the cleaned up collection.

```
> cleanTDM <- TermDocMatrix(plone, control = params)
```

Before continuing our investigations we create two small functions to encapsulate procedures we will use quite often:

```
> sft <- function(corpus, matrix, low, high)
+     stemCompletion(corpus,
+                    sort(findFreqTerms(matrix, low, high)))
> sa <- function(corpus, matrix, term, cor)
+     stemCompletion(corpus,
+                    names(findAssocs(matrix, term, cor)))
```

The function `sft` searches for frequent terms in a term-document matrix under given range, sorts the output and performs heuristic stem completion. The function `sa` search for associations for a given term in a term-document matrix, utilizing stem completion in a corpus. We use stem completion because stemming cut most terms to its word

stem making human interpretation a bit harder than necessary. Therefore we developed a heuristics for stem completion (`stemCompletion()`). In detail it works by searching the collection for possible completions and takes an appropriate one according to the heuristics. E.g., the application of the completion function to a subset of our problem yields

```
> stemCompletion(plone, c("compon", "modul"))

     compon       modul
"component"    "module"
```

making it easier for human interpretation.

Again we redo our attempt to find relevant terms,

```
> sft(plone, cleanTDM, 20, 25)

 [1] "app"           "component"     "event"         "file"
 [5] "http"          "line"          "module"        "normalizeargs"
 [9] "openid"        "portlet"       "python"        "traceback"
[13] "zodb"          "zope"
```

but this time a lot of obviously alien terms are removed. Still, the interpretation is rather hard since we miss the context for further guided inspection. Manual inspection of the forum postings (e.g., with `inspect(tmFilter(plone, "http"))`) shows that the terms `http`, `file`, or `app` are representative for postings dealing with the configuration of content management applications. The terms `event` and `traceback` identify postings related to problems with specific components of Plone.

8.1.3 Associations

Next, we want to find associations for interesting topics, e.g., the associations for `plone`, `info`, and `feature` (after stemming `featur`). This approach adds a predefined context to found terms. Compared to our first approach (with `findFreqTerms()`) this technique helps to reduce manual investigation.

We start with searching the associations for the term `plone`:

```
> sa(plone, cleanTDM, "plone", 0.7)

 [1] "plone"      "opt"        "file"       "python"     "lib"
 [6] "provide"    "zeocluster" "client1"    "traceback"  "info"
[11] "component"
```

The results are not surprising, since Plone is written in the programming language Python. Further the forum deals with user postings on problems (e.g., with tracebacks) or information (info) requests on components.

```
> sa(plone, cleanTDM, "info", 0.9)

 [1] "action"                    "cls"
 [3] "conform"                   "declarations"
 [5] "deprecated"                "dispatchObjectMovedEvent"
 [7] "factorytool"               "favorite"
 [9] "genericsetup"              "info"
[11] "large"                     "mailhost"
[13] "memberdata"                "normalizeargs"
[15] "notification"              "objectevent"
[17] "output"                    "passwordreset"
[19] "pickle"                    "sequence"
[21] "setstate"                  "setuphandlers"
[23] "sitemanager"               "specific"
[25] "subscription"              "toolset"
[27] "unpickler"                 "warn"
[29] "subscribers"               "component"
[31] "serialize"                 "interface"
[33] "textindexng"               "init"
[35] "provide"                   "load"
[37] "type"                      "import"
[39] "connect"                   "plonepas"
[41] "couldn"                    "traceback"
```

For the word `info` the results are key terms users were searching information for. These terms might be very interesting to be dealt in Frequently Asked Questions for further reference for other users. Such an action can both improve the documentation quality since the information is useful for a broad scope of users, and can focus the effort of the consumer response team on new questions.

```
> sa(plone, cleanTDM, "featur", 0.6)

[1] "feature"     "carlos"      "popoll"      "tweaked"     "plonepopoll"
[6] "poll"
```

Finally the associations for `feature` are rather generic. We notice there are polls for new features or users are searching for tweaks for some missing features. Figure 8.1 visualizes the correlations within the term-document matrix `cleanTDM` for this situation. So for identifying actual features, a key issue in business intelligence for e-commerce, we need to perform further steps.

We start by filtering out only those documents from our collection of postings containing the term `feature`:

```
> (features <- tmFilter(plone, "feature"))

A text document collection with 20 text documents
```

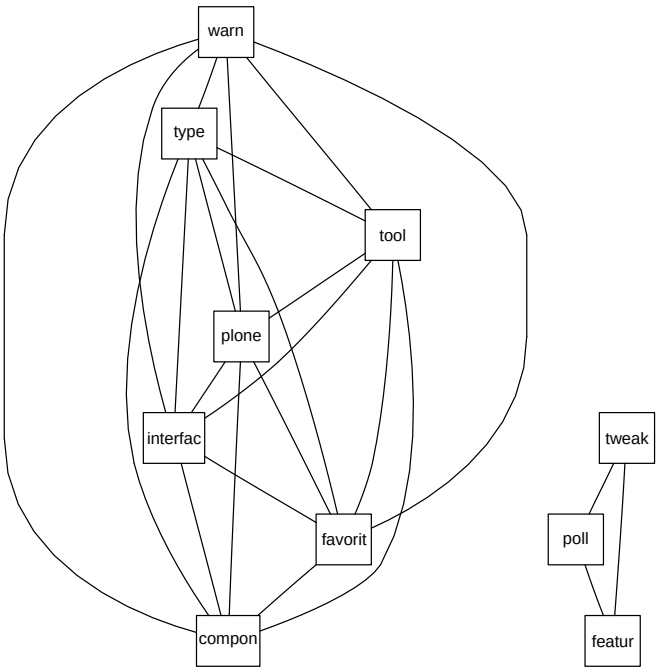

Figure 8.1: Visualization of the correlations for a selection of terms within the term-document matrix `cleanTDM`.

The resulting collection contains twenty articles. This small numbers allows the analyst to perform a manual deep investigation of the material. As we investigate the postings we see that most are feature announcements or requests, i.e., highly relevant information from a business perspective. E.g., the first posting (we show only the first few lines which mainly underline the availability of a new product)

```
> features[[1]][1:12]
```

```
 [1] Hi all,
 [2]
 [3]  EngageMedia are very excited to announce the release of Plumi, a free
 [4]  software video sharing platform. Plumi enables you to create a
 [5]  sophisticated video sharing and community site out-of-the-box. In a
 [6]  web landscape where almost all video sharing sites keep their
 [7]  distribution platform under lock and key Plumi is one contribution to
 [8]  creating a truly democratic media.
 [9]  http://plumi.org | http://engagemedia.org
[10]
[11]  Plumi is based on the popular Plone Content Management System <http://
[12]  plone.org> and adds a wide array of functionality including:
```

presents a new platform based on `plone`. So we might be interested what `plumi` can

actually do: we can either read manually the posting or we can try to find correlated terms to `plumi`. For the latter we need a term-document matrix

```
> featuresTDM <- TermDocMatrix(features, control = params)
```

such that we can apply it to our function for finding associations based on correlations within the term-document matrix:

```
> sa(plone, featuresTDM, "plumi", 0.7)
```

```
 [1] "add"           "announce"      "automatically" "based"
 [5] "box"           "dave"          "demo"          "download"
 [9] "engagemedia"   "feedback"      "flash"         "free"
[13] "function"      "included"      "join"          "licensed"
[17] "listinfo"      "media"         "metadata"      "platform"
[21] "playback"      "plumi"         "search"        "share"
[25] "transmission"  "upload"        "video"         "vodcast"
[29] "workspace"     "list"          "system"        "http"
[33] "custom"        "org"           "file"          "create"
```

As we find out the result matches the new concepts introduced by `plumi` and described in the posting announcing the new product.

8.1.4 Groups

A complementary approach in our e-commerce text mining analysis is clustering. We choose to use three clusters not knowing whether this helps us in identifying interesting groups in the postings. The idea is that we hope to find relations to previous three mentioned groups: problems, features, and other documents. We use a k-means clustering algorithm with $k = 3$ representing the three desired clusters. Then we print the size of each of the clusters (`km$size`):

```
> km <- kmeans(Data(cleanTDM), 3)
> km$size
```

```
[1]   1 989   9
```

Manual investigation shows that clusters 1 and 3 are very technical mails, connected by code parts and special terms occurring in code and texts. Thus the clustering process is not as satisfying as we hoped. Anyway we can remove the found technical clusters since they are not really relevant in a business intelligence context. Contrary, for a technical user support forum these very technical postings may be of special interest. Technical mails are characterized by their length since they contain longer code sections. As we see

```
> summary(sapply(plone, length))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    9.00   14.00   19.65   22.00  448.00
```

the mean posting length is about 20 lines. So we investigate postings longer than 80 lines to identify postings containing code sections.

```
> len <- sapply(plone, length) > 80
> plone[len]
```

```
A text document collection with 19 text documents
```

Manual investigation shows that the vast majority of filtered out postings in fact contain code which we remove for further experiments with the remaining text collection:

```
> ploneShort <- plone[!len]
```

We create a term-document matrix from the shortened collection `ploneShort`.

```
> csTDM <- TermDocMatrix(ploneShort, control = params)
```

and cluster the newly created term-document matrix `csTDM`.

```
> km <- kmeans(Data(csTDM), 3)
> km$size
```

```
[1]  10  95 875
```

Cluster 1 contains some remaining technical e-mails which are shorter than 80 lines and thus were not identified in our previous approaches. Cluster 3 contains more than 800 documents grouping potentially many concepts together. Cluster 2 is especially interesting since it contains only about 90 documents which seem to cover the concept of feature announcements after a first manual investigation. So let us test the number of postings mentioning the term `feature` in this second cluster:

```
> tmFilter(ploneShort[km$cluster == 2], "feature")
```

```
A text document collection with 6 text documents
```

The cluster 2 collection contains half as much documents dealing with `feature` as the collection from the third cluster. Nonetheless, when you compare the cluster sizes (relation 1:10) this shows that the percentage of feature documents is far higher than in any other cluster (6 feature documents in cluster 2 of size 95 versus 13 feature documents in cluster 3 of size 875). For other terms (e.g., `problem`), we find a relation as one would rather expect (i.e., documents are equally distributed in clusters according to their size). This highlights that cluster 2 contains the concept of feature announcements.

8.1.5 Non-Explicit Grouping

Based on the results from the previous paragraph we are interested in finding postings dealing with features *without* explicitly mentioning the term `feature` in them. We denote this as *non-explicit grouping*. A way to do this is to perform a classification. At first we have to manually label training documents dealing with feature related content. Due to the tedious work of the manual labeling procedure we restrict our experiments to 10 percent of our original data set, i.e., we only use about 100 documents. The first 20 documents are our training data set with the manual labels:

```
> trainClassification <-
+     factor(c("other", "other", "feature", "feature", "feature",
+              "other", "feature", "other", "other", "feature",
+              "other", "other", "feature", "other", "other",
+              "feature", "other", "feature", "feature", "other"))
```

However we skip those documents explicitly mentioning the term `feature` as we already know them.

```
> featureIndex <- tmIndex(plone, "feature")
```

Hence we obtain as training data set the first twenty remaining documents:

```
> train <- as.matrix(cleanTDM[!featureIndex][1:20, ])
```

whereas the test data set are 80 other non-related postings:

```
> test <- cleanTDM[!featureIndex, ][21:100, ]
> testCollection <- plone[!featureIndex][21:100]
```

Note that the training and test documents were chosen arbitrarily but fixed for reproducibility.

We use a classification technique via support vector machines from the **kernlab** package.

```
> ksvmTrain <- ksvm(train, trainClassification)
```

```
Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

Now we can predict the test documents:

```
> svmCl <- predict(ksvmTrain, test)
> testCollection[svmCl == "feature"]
```

```
A text document collection with 52 text documents
```

We obtain about 50 documents. Inspecting the documents shows that many of these as `feature` classified documents are business relevant since they in fact discuss the product's wish behavior and elements that should be changed or improved. The main point is that we found feature connected postings without explicitly having the term `feature` in them, i.e., not explicitly mentioned terms. Nevertheless, this procedure heavily relies on correct labels for the training set such that the learning procedure (e.g., the Support Vector Machine) can extract the relevant terms for discrimination.

8.2 Promotion

The area promotion and its IT support includes a broad set of different areas in the e-commerce context. Amongst others these are advertising in and for consumer information systems, direct marketing, news letters, call centers, and virtual communities.

Within these fields text mining can help to deal with textual material obtained via replies to news letters, transcriptions of call center interviews, and analyses of virtual communities (like forums). Since we have already used a data set containing forum postings (the Plone data set) in the previous section we will continue to use it for further example presentation.

8.2.1 Competitive Products

An interesting question for promotion has always been the connection between a company's own advertising and marketing efforts and those of its competitors. One approach addressing this issue is finding forum postings mentioning competitive products. Then a marketing specialist can analyze the subset of relevant postings to see what problems occur and why consumers use competitive products. This helps in improving the company's own advertising and marketing strategy.

So let us begin with finding out whether competitive products are discussed within the forum.

```
> (otherCMS <- tolower(readLines("Data/otherCMS.txt")))
```

```
 [1] "alfresco"       "lenya"          "apache2triad"   "b2evolution"
 [5] "blog:cms"       "blosxom"        "bricolage"      "cmsimple"
 [9] "cyclone3"       "daisy"          "dokuwiki"       "dotnetnuke"
[13] "drupal"         "e107"           "ez"             "fedora"
[17] "japs"           "joomla"         "knowledgetree"  "lyceum"
[21] "mambo"          "mediawiki"      "midgard"        "nucleus"
[25] "nuxeo"          "openacs"        "opencms"        "phpcms"
[29] "php-fusion"     "php-nuke"       "phpwcms"        "phpwebsite"
[33] "phpwiki"        "pmwiki"         "postnuke"       "quick.cms.lite"
[37] "scoop"          "silverstripe"   "siteframe"      "slash"
[41] "spip"           "textpattern"    "tikiwiki"       "twiki"
[45] "typo"           "typo3"          "webgui"         "wordpress"
[49] "xoops"
```

is a list of competitive products listed from Wikipedia (http://en.wikipedia.org/wiki/Comparison_of_content_management_systems; Accessed 17 May 2008) where we removed Plone and Zope (since Plone is based on it). Now we can intersect the product names with the forum postings to find documents dealing with competitors. Technically we use the `tmIntersect` function for this task:

```
> rivals <- tmIndex(plone, FUN = tmIntersect, otherCMS)
> plone[rivals]
```

```
A text document collection with 14 text documents
```

We see the result are only fourteen mails. This is rather good since this significantly increases the chances in real-word analyses that the postings are really read by qualified marketing analysts. Instead they would have to deal with all postings wasting valuable time since they cannot focus on important data.

So for Plone's creators these fourteen mails are valuable since they discuss competitive products compared to Plone, e.g., the second article deals with several competitive products under specific technical restrictions (e.g., no HTML experience):

```
> plone[rivals][[2]]

[1] Hello.  I'm new to using Plone and recently came into a non-profit job
[2] that is looking to overhaul it's communications infrastructure.  We are
[3] trying to find a new database and were recommended CiviCRM, but most of
[4] the staff does not have HTML experience.  However, we were also told
[5] that if we used a CMS like Plone it would be very easy to use CiviCRM.
[6] That said, does anyone know if the two programs (CiviCRM and Plone) are
[7] compatible? Or are we going to have to switch over to another program
[8] such as Drupla or Joomla (neither of which we currently use)?
```

This approach allows us to find missing features and yields opinions why other products can do some things better. This can be the basis for improving a company's own advertising, marketing strategy, and products.

8.2.2 Churn Analysis

In the previous paragraph we saw that we can find problems, questions, and announcements related to competitive products. In case of problems another vital issue for a company is to find (potential) migrators which had problems with the company's products. This allows to identify negative user experience and avoid further migration.

We filter out those documents containing the term `problem`

```
> problems <- tmFilter(plone, "problem")
```

and create a term-document matrices from it:

```
> problemsTDM <- TermDocMatrix(problems, control = params)
```

so we have a subset of our original data containing problem related text material.

Now let us investigate certain associations within this problem/migration collection.

```
> sa(plone, problemsTDM, "plone", 0.3)

 [1] "plone"   "zope"    "site"    "org"     "title"   "running" "based"
 [8] "gmane"   "public"  "type"    "irc"     "mailto"  "portal"  "regards"
[15] "spliter" "version" "tried"   "look"    "content" "python"  "mysql"
[22] "skins"
```

The associations found for `plone` seem to make sense, most of them match the main issues/nouns after manual investigation of the `problem` text collection, like problems with MySQL, HTML issues, or customizing a portal's look with skins. This means, we have found sensible information via a completely automated procedure, enabling better detection of negative user sentiments. As a consequence these terms should be the starting point for product improvements in further releases. Another issue might be to improve the support infrastructure so that existing problems in these areas can be easier solved.

9 Law Mining

9.1 Introduction

A thorough discussion and investigation of existing jurisdictions is a fundamental activity of law experts since convictions provide insight into the interpretation of legal statutes by supreme courts. On the other hand, text mining has become an effective tool for analyzing text documents in automated ways. Conceptually, clustering and classification of jurisdictions as well as identifying patterns in law corpora are of key interest since they aid law experts in their analyses. E.g., clustering of primary and secondary law documents as well as actual law firm data has been investigated by Conrad et al. (2005). Schweighofer (1999) has conducted research on automatic text analysis of international law.

In this chapter we use text mining methods to investigate Austrian supreme administrative court jurisdictions concerning dues and taxes. The data is described in Section 9.2.1 and analyzed in Section 9.3. Results of applying clustering and classification techniques are compared to those found by tax law experts. We also propose a method for automatic feature extraction (e.g., of the senate size) from Austrian supreme court jurisdictions. Section 9.4 concludes.

9.2 Administrative Supreme Court Jurisdictions

9.2.1 Data

The data set for our text mining investigations consists of 994 text documents. Each document contains a jurisdiction of the Austrian supreme administrative court (Verwaltungsgerichtshof, VwGH) in German language. Documents were obtained through the legal information system (Rechtsinformationssystem, RIS; `http://ris.bka.gv.at/`) coordinated by the Austrian Federal Chancellery. Unfortunately, documents delivered through the RIS interface are HTML documents oriented for browser viewing and possess no explicit metadata describing additional jurisdiction details (e.g., the senate with its judges or the date of decision). The data set corresponds to a subset of about 1000 documents of material used for the research project "Analyse der abgabenrechtlichen Rechtsprechung des Verwaltungsgerichtshofes" supported by a grant from the Jubiläumsfonds of the Austrian National Bank (Oesterreichische Nationalbank, OeNB), see Hornik et al. (2006). Based on the work of Achatz et al. (1987) who analyzed tax law jurisdictions in the 1980s this project investigates whether and how results and trends found by Achatz et al. compare to jurisdictions between 2000 and 2004, giving insight into

legal norm changes and their effects and unveiling information on the quality of executive and juristic authorities. In the course of the project, jurisdictions especially related to dues (e.g., on a federal or communal level) and taxes (e.g., income, value-added or corporate taxes) were classified by human tax law experts. These classifications will be employed for validating the results of our text mining analyses.

9.2.2 Data Preparation

We use the open source software environment R for statistical computing and graphics, in combination with the R text mining package **tm** to conduct our text mining experiments. R provides premier methods for clustering and classification whereas **tm** provides a sophisticated framework for text mining applications, offering functionality for managing text documents, abstracting the process of document manipulation and easing the usage of heterogeneous text formats.

Technically, the jurisdiction documents in HTML format were downloaded through the RIS interface. To work with this inhomogeneous set of malformed HTML documents, HTML tags and unnecessary white space were removed resulting in plain text documents. We wrote a custom parsing function to handle the automatic import into **tm**'s infrastructure and extract basic document metadata (like the file number).

9.3 Investigations

9.3.1 Grouping the Jurisdiction Documents into Tax Classes

When working with larger collections of documents it is useful to group these into clusters in order to provide homogeneous document sets for further investigation by experts specialized on relevant topics. Thus, we investigate different methods known in the text mining literature and compare their results with the results found by law experts.

k-means Clustering We start with the well known k-means clustering method on term-document matrices. Let $\mathrm{tf}_{t,d}$ be the frequency of term t in document d, m the number of documents, and df_t is the number of documents containing the term t. Term-document matrices M with respective entries $\omega_{t,d}$ are obtained by suitably weighting the term-document frequencies. The most popular weighting schemes are *Term Frequency* (*tf*), where $\omega_{t,d} = \mathrm{tf}_{t,d}$, and *Term Frequency Inverse Document Frequency* (*tf-idf*), with $\omega_{t,d} = \mathrm{tf}_{t,d} \log_2(m/\mathrm{df}_t)$, which reduces the impact of irrelevant terms and highlights discriminative ones by normalizing each matrix element under consideration of the number of all documents. We use both weightings in our tests. In addition, text corpora were stemmed before computing term-document matrices via the **Rstem** (Temple Lang, 2004) and **Snowball** (Hornik, 2007b) R packages which provide the Snowball stemming (Porter, 1980) algorithm.

Domain experts typically suggest a basic partition of the documents into three classes (income tax, value-added tax, and other dues). Thus, we investigated the extent to

Table 9.1: Rand index and Rand index corrected for agreement by chance of the contingency tables between k-means results, for $k \in \{3, 4, 5, 6\}$, and expert ratings for *tf* and *tf-idf* weightings.

	Rand		cRand	
k	*tf*	*tf-idf*	*tf*	*tf-idf*
3	0.48	0.49	0.03	0.03
4	0.51	0.52	0.03	0.03
5	0.54	0.53	0.02	0.02
6	0.55	0.56	0.02	0.03
Average	0.52	0.52	0.02	0.03

which this partition is obtained by automatic classification. We used our data set of about 1000 documents and performed k-means clustering, for $k \in \{2, \ldots, 10\}$. The best results were in the range between $k = 3$ and $k = 6$ when considering the improvement of the within-cluster sum of squares. These results are shown in Table 9.1. For each k, we compute the agreement between the k-means results based on the term-document matrices with either *tf* or *tf-idf* weighting and the expert rating into the basic classes, using both the Rand index (Rand) and the Rand index corrected for agreement by chance (cRand). Row "Average" shows the average agreement over the four ks. Results are almost identical for the two weightings employed. Agreements are rather low, indicating that the "basic structure" can not easily be captured by straightforward term-document frequency classification.

We note that clustering of collections of large documents like law corpora presents formidable computational challenges due to the dimensionality of the term-document matrices involved: even after stopword removal and stemming, our about 1000 documents contained about 36000 different terms, resulting in (very sparse) matrices with about 36 million entries. Computations took only a few minutes in our cases. Larger datasets as found in law firms will require specialised procedures for clustering high-dimensional data.

Keyword Based Clustering Based on the special content of our jurisdiction dataset and the results from k-means clustering we developed a clustering method which we call *keyword based clustering*. It is inspired by simulating the behaviour of tax law students preprocessing the documents for law experts. Typically the preprocessors skim over the text looking for discriminative terms (i.e., keywords). Basically, our method works in the same way: we have set up specific keywords describing each cluster (e.g., "income" or "income tax" for the income tax cluster) and analyse each document on the similarity with the set of keywords.

Figure 9.1 shows a mosaic plot for the contingency table of cross-classifications of keyword based clustering and expert ratings. The size of the diagonal cells (visualizing the proportion of concordant classifications) indicates that the keyword based clustering methods works considerably better than the k-means approaches, with a Rand index of

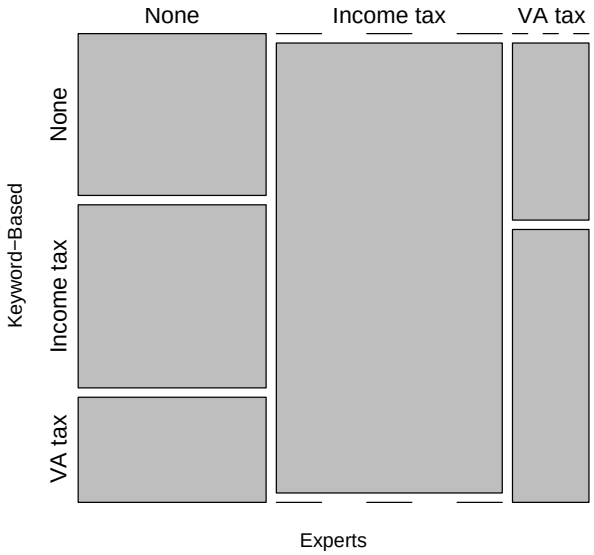

Figure 9.1: Plot of the contingency table between the keyword based clustering results and the expert rating.

0.66 and a corrected Rand index of 0.32. In particular, the expert "income tax" class is recovered perfectly.

9.3.2 Classification of Jurisdictions According to Federal Fiscal Code Regulations

A further rewarding task for automated processing is the classification of jurisdictions into documents dealing and into documents not dealing with Austrian federal fiscal code regulations (Bundesabgabenordnung, BAO).

Due to the promising results obtained with string kernels in text classification and text clustering (Lodhi et al., 2002; Karatzoglou and Feinerer, 2007) we performed a "C-svc" classification with support vector machines using a full string kernel, i.e., using

$$k(x, y) = \sum_{s \in \Sigma^*} \lambda_s \cdot \nu_s(x) \cdot \nu_s(y)$$

as the kernel function $k(x, y)$ for two character sequences x and y. We set the decay factor $\lambda_s = 0$ for all strings $|s| > n$, where n denotes the document lengths, to instantiate a so-called full string kernel (full string kernels are computationally much better natured). The symbol Σ^* is the set of all strings (under the Kleene closure), and $\nu_s(x)$ denotes the number of occurrences of s in x.

Table 9.2: Rand index and Rand index corrected for agreement by chance of the contingency tables between SVM classification results and expert ratings for documents under federal fiscal code regulations.

	tf	*tf-idf*
Rand	0.59	0.61
cRand	0.18	0.21

Table 9.3: Number of jurisdictions ordered by senate size obtained by fully automated text mining heuristics. The percentage is compared to the percentage identified by humans.

Senate size	0	3	5	9
Documents	0	255	739	0
Percentage	0.000	25.654	74.346	0.000
Human Percentage	2.116	27.306	70.551	0.027

For this task we used the **kernlab** (Karatzoglou et al., 2006, 2004) R package which supports string kernels and SVM enabled classification methods. We used the first 200 documents of our data set as training set and the next 50 documents as test set. We compared the 50 received classifications with the expert ratings which indicate whether a document deals with the BAO by constructing a contingency table (confusion matrix). We received a Rand index of 0.49. After correcting for agreement by chance the Rand index floats around at 0. We measured a very long running time (almost one day for the training of the SVM, and about 15 minutes prediction time per document on a 2.6 GHz machine with 2 GByte RAM).

Therefore we decided to use the classical term-document matrix approach in addition to string kernels. We performed the same set of tests with *tf* and *tf-idf* weighting, where we used the first 200 rows (i.e, entries in the matrix representing documents) as training set, the next 50 rows as test set.

Table 9.2 presents the results for classifications obtained with both *tf* and *tf-idf* weightings. We see that the results are far better than the results obtained by employing string kernels.

These results are very promising, and indicate the great potential of employing support vector machines for the classification of text documents obtained from jurisdictions in case term-document matrices are employed for representing the text documents.

9.3.3 Deriving the Senate Size

Jurisdictions of the Austrian supreme administrative court are obtained in so-called senates which can have 3, 5, or 9 members, with size indicative of the "difficulty" of the legal case to be decided. (It is also possible that no senate is formed.) An automated derivation of the senate size from jurisdiction documents would be highly useful, as it would allow to identify structural patterns both over time and across areas. Although

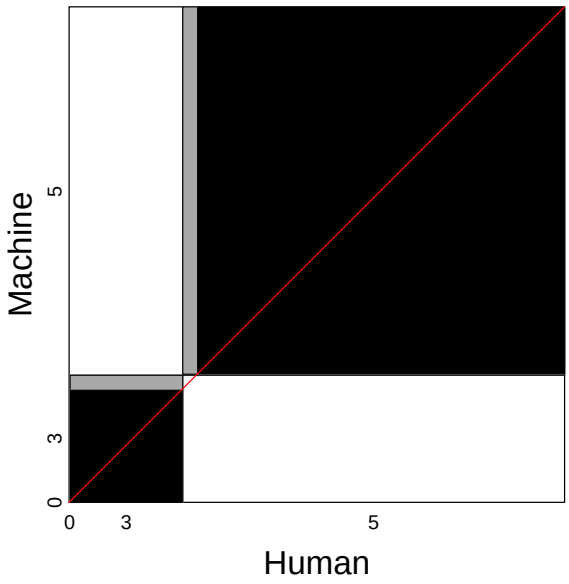

Figure 9.2: Agreement plot of the contingency table between the senate size reported by text mining heuristics and the senate size reported by humans.

the formulations describing the senate members are quite standardized it is rather hard and time-consuming for a human to extract the senate size from hundreds of documents because a human must read the text thoroughly to differ between senate members and auxiliary personnel (e.g., a recording clerk). Thus, a fully automated extraction would be very useful.

Since most documents contain standardized phrases regarding senate members (e.g., "The administrative court represented by president Dr. X and the judges Dr. Y and Dr. Z ... decided ...") we developed an extraction heuristic based on widely used phrases in the documents to extract the senate members. In detail, we investigate punctuation marks and copula phrases to derive the senate size. Table 9.3 summarizes the results for our data set by giving the total number of documents for senate sizes of zero (i.e., documents where no senate was formed, e.g., due to dismissal for want of form), three, five, or nine members. The table also shows the percentages and compares these to the aggregated percentages of the full data set, i.e., $n > 1000$, found by humans. Figure 9.2 visualizes the results from the contingency table between machine and human results in form of an agreement plot, where the observed and expected diagonal elements are represented by superposed black and white rectangles, respectively. The plot indicates that the extraction heuristic works very well. This is supported by the very high Rand index of 0.94 and by the corrected Rand index of 0.86.

Further improvements could be achieved by saving identified names of judges in order to identify them again in other documents. Of course, ideally information such as

senate size would be provided as metadata by the legal information system, perhaps even determined automatically by text mining methods for "most" documents (with a per-document measure of the need for verification by humans).

9.4 Summary

In this chapter we have presented approaches to use text mining methods on (supreme administrative) court jurisdictions. We performed k-means clustering and introduced keyword based clustering which works well for text corpora with well defined formulations as found in tax law related jurisdictions. We saw that the clustering works well enough to be used as a reasonable grouping for further investigation by law experts. Second, we investigated the classification of documents according to their relation to federal fiscal code regulations. We used both string kernels and term-document matrices with *tf* and *tf-idf* weighting as input for support vector machine based classification techniques. The experiments unveiled that employing term-document matrices yields both superior performance as well as fast running time. Finally, we considered a situation typical in working with specialized text corpora, i.e., we were looking for a specific property in each text corpus. In detail we derived the senate size of each jurisdiction by analyzing relevant text phrases considering punctuation marks, copulas and regular expressions. Our results show that text mining methods can clearly aid legal experts to process and analyze their law document corpora, offering both considerable savings in time and cost as well as the possibility to conduct investigations barely possible without the availability of these methods.

10 Wizard of Oz Stylometry

10.1 Data

The Wizard of Oz book series has been among the most popular children's novels in the last century. The first book was created and written by Lyman Frank Baum, published in 1900. A series of Oz books followed until Baum died in 1919. After his death Ruth Plumly Thompson continued the story of Oz books, but there was a longstanding doubt which was the first Oz book written by Thompson. Especially the authorship of the 15th book of Oz—*The Royal Book of Oz*—has been longly disputed amongst literature experts. Today it is commonly attributed to Thompson as her first Oz book, supported by several statistical stylometric analyses within the last decade.

Based on some techniques shown in the work of Binongo (2003) we will investigate a subset of the Oz book series for authorship attribution. Our data set consists of five books of Oz, three attributed to Lyman Frank Baum, and two attributed to Ruth Plumly Thompson:

The Wonderful Wizard of Oz is the first Oz book written by Baum and was published in 1900.

The Marvelous Land of Oz is the second Oz book by Baum published in 1904.

Ozma of Oz was published in 1907. It is authored by Baum and forms the third book in the Oz series.

The Royal Book of Oz is nowadays attributed to Thompson, but its authorship has been disputed for decades. It was published in 1921 and is considered as the 15th book in the series of Oz novels.

Ozoplaning with the Wizard of Oz was written by Thompson, is the 33rd book of Oz, and was published in 1939.

Most books of Oz, including the five books we use as corpus for our analysis, can be freely downloaded at the Gutenberg Project website (`http://www.gutenberg.org/`) or at the Wonderful Wizard of Oz website (`http://thewizardofoz.info/`).

The main data structure in **tm** is a *corpus* consisting of *text documents* and additional meta data. So-called *reader* functions can be used to read in texts from various *sources*, like text files on a hard disk. E.g., following code creates the corpus `oz` by reading in the text files from the directory 'OzBooks/', utilizing a standard reader for plain texts.

```
> oz <- Corpus(DirSource("OzBooks/"))
```

The directory 'OzBooks/' contains five text files in plain text format holding the five above described Oz books. So the resulting corpus has five elements representing them:

```
> oz
A text document collection with 5 text documents
```

Since our input are plain texts without meta data annotations we might want to add the meta information manually. Technically we use the `meta()` function to operate on the meta data structures of the individual text documents. In our case the meta data is stored locally in explicit slots of the text documents (`type = "local"`):

```
> meta(oz, tag = "Author", type = "local") <-
     c(rep("Lyman Frank Baum", 3),
       rep("Ruth Plumly Thompson", 2))
>  meta(oz, "Heading", "local") <-
     c("The Wonderful Wizard of Oz",
       "The Marvelous Land of Oz",
       "Ozma of Oz",
       "The Royal Book of Oz",
       "Ozoplaning with the Wizard of Oz")
```

E.g. for our first book in our corpus this yields

```
> meta(oz[[1]])

Available meta data pairs are:
  Author        : Lyman Frank Baum
  Cached        : TRUE
  DateTimeStamp: 2008-04-21 16:23:52
  ID            : 1
  Heading       : The Wonderful Wizard of Oz
  Language      : en_US
  URI           : file
    OzBooks//01_TheWonderfulWizardOfOz.txt UTF-8
```

10.2 Extracting Frequent Terms

The first step in our analysis will be the extraction of the most frequent terms, similarly to a procedure by Binongo (2003). At first we create a so-called *term-document matrix*, that is a representation of the corpus in form of a matrix with documents as rows and terms as columns. The matrix elements are frequencies counting how often a term occurs in a document.

```
> ozMatBaum <- TermDocMatrix(oz[1:3])
> ozMatRoyal <- TermDocMatrix(oz[4])
> ozMatThompson <- TermDocMatrix(oz[5])
```

After creating three term-document matrices for the three cases we want to distinguish—first, books by Baum, second, the Royal Book of Oz, and third, a book by Thompson—we can use the function `findFreqTerms(mat, low, high)` to compute the terms in the matrix `mat` occurring at least `low` times and at most `high` times (defaults to `Inf`).

```
> baum <- findFreqTerms(ozMatBaum, 70)
> royal <- findFreqTerms(ozMatRoyal, 50)
> thomp <- findFreqTerms(ozMatThompson, 40)
```

The lower bounds have been chosen in a way that we obtain about 100 terms from each term-document matrix. A simple test of intersection within the 100 most common words shows that the Royal Book of Oz has more matches with the Thompson book than with the Baum books, being a first indication for Thompson's authorship.

```
> length(intersect(thomp, royal))

[1] 73

> length(intersect(baum, royal))

[1] 65
```

10.3 Principal Component Analysis

The next step applies a Principal Component Analysis (PCA), where we use the **kernlab** (Karatzoglou et al., 2004) package for computation and visualization. Instead of using the whole books we create equally sized chunks to consider stylometric fluctuations within single books. In detail, the function `makeChunks(corpus, chunksize)` takes a corpus and the chunk size as input and returns a new corpus containing the chunks. Then we can compute a term-document matrix for a corpus holding chunks of 500 lines length. Note that we use a *binary weighting* of the matrix elements, i.e., multiple term occurrences are only counted once.

```
> ozMat <- TermDocMatrix(makeChunks(oz, 500),
              list(weighting = weightBin))
```

This matrix is the input for the Kernel PCA, where we use a standard Gaussian kernel, and request two feature dimensions for better visualization.

```
> k <- kpca(as.matrix(ozMat), features = 2)
> plot(rotated(k),
       col = c(rep("black", 10), rep("red", 14),
               rep("blue", 10),
               rep("yellow", 6), rep("green", 4)),
       pty = "s",
       xlab = "1st Principal Component",
       ylab = "2nd Principal Component")
```

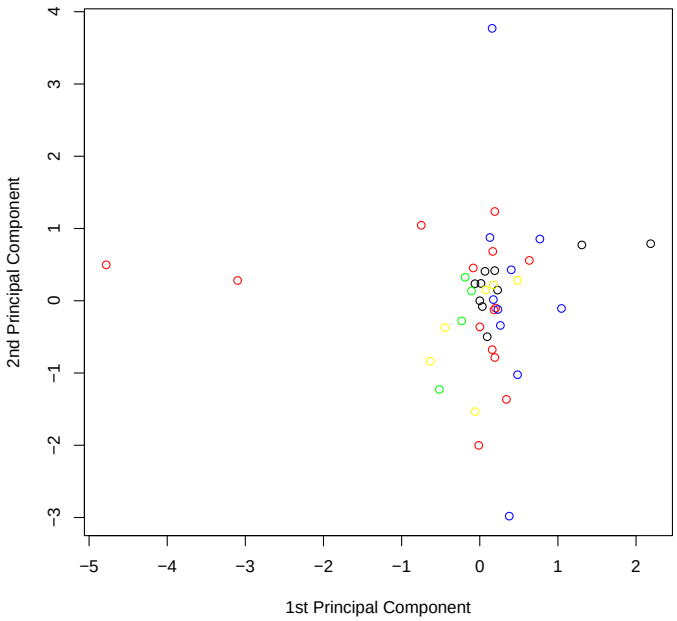

Figure 10.1: Principal component plot for five Oz books using 500 line chunks.

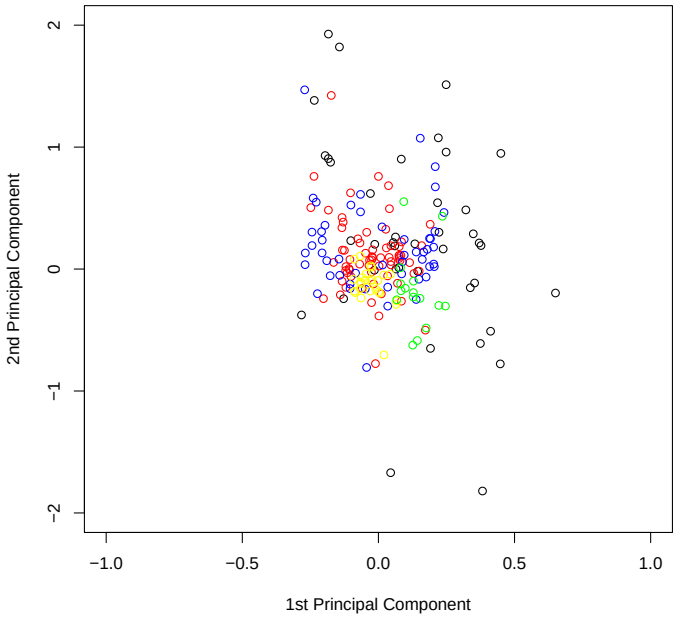

Figure 10.2: Principal component plot for five Oz books using 100 line chunks.

Figure 10.1 shows the results. Circles in black are chunks from the first book of Oz by Baum, red circles denote chunks from the second book by Baum, and blue circles correspond to chunks of the third book by Baum. Yellow circles depict chunks from the long disputed 15th book (by Thompson), whereas green circles correspond to the 33rd book of Oz by Thompson. The results show that there is a visual correspondence between the 15th and the 33rd book (yellow and green), this means that both books tend to be authored by Thompson. Anyway, the results also unveil that there are parts matching with books from Baum. Similarly, we can redo this plot using chunks of 100 lines. This time we use a weighting scheme called *term frequency inverse document frequency*, a weighting very common in text mining and information retrieval.

```
> ozMat <- TermDocMatrix(makeChunks(oz, 100),
              list(weighting = weightTfIdf))
```

Figure 10.2 shows the corresponding plot using the same colors as before. Again we see that there is a correspondence between the 15th book and the 33rd book by Thompson (yellow and green), but also matching parts with books by Baum.

11 Conclusion

In this thesis we introduced a new framework for text mining applications in R via the **tm** software package designed for applications both in research as in business intelligence applications.

We started with an overview of existing text mining literature, investigated approaches seen in various application fields, like stylometry or biomedicine, and reviewed existing text mining infrastructures and tools. After motivating the creation of our own text mining infrastructure we extensively described the concepts behind our framework, explaining data structures and the algorithms operating on them: **tm** can handle corpora from heterogeneous sources, makes it easy to interact with them and their corresponding meta data. The framework provides predefined mechanisms for preprocessing, like stemming, part-of-speech tagging or stopword removal. Further, the infrastructure is designed in a modular way to enable easy extensibility. Additionally **tm** provides both methods to export corpora into classical data structures, like term-document matrices, as to support the straightforward integration of newer methods like string kernels.

The second part of this thesis is dedicated to realistic applications of our framework. The first example was an analysis of a mailing list with aspects of a social network analysis. The second application dealt with text mining for business to consumer electronic commerce showing potential for incorporating text mining for product innovation, churn analysis, or consumer feedback investigation. The third example shows how text mining methods can be implemented when working with law documents. Finally we showed an application for authorship attribution on the well known Wizard of Oz book series. The results are promising for all shown examples and show that our framework can be used for interesting exploration studies.

In the end, the appendix gives a detailed reference for the **tm** software. Both an in-depth description of the internal data structures as an extensive explanation of **tm**'s methods and classes is given.

Part III

Appendix

A Framework Classes Minutiae

`Corpus` represents a collection of text documents, also denoted as corpus in linguistics, and can be interpreted as a database for texts. Technically it extends the formal class `list` and holds elements of class `TextDocument`. It contains two slots holding metadata and one slot for database support:

`DMetaData` representing *Document Metadata* is a data frame storing metadata attributes for the text documents in the collection. Conceptually, `DMetaData` is designed for document metadata regarded as an own entity, e.g., clustering or classification results since they might contain information on the number of available clusters or the classification technique. Document metadata best suited for single text documents should be stored directly with the text document (see `LocalMetaData` of `TextDocument` as described later). However, metadata local to text documents may be copied to the data frame for technical reasons, e.g., better performance for metadata queries, with the `prescindMeta()` command. In this case the user is responsible for holding the metadata consistent between the data frame and the locally stored text documents' metadata.

The data frame has a row for each document and possesses at least the column `MetaID` which associates each data frame row with its originating document collection, e.g., `MetaID` is automatically updated if several document collections are merged via the overloaded `c()` concatenation function for document collections. This allows to split away merged collections under full metadata recovery. metadata can be added via `appendMeta()` and deleted via the `removeMeta()` commands.

`CMetaData` representing *Collection Metadata* is of class `MetaDataNode` modeling a binary tree holding metadata specific for text document collections. Technically, a `MetaDataNode` has the three slots `NodeID` holding a unique identification number, `MetaData` holding the actual metadata itself, and `children` of class `MetaDataNode` building up the binary tree. Typically the root node would hold information like the creator or the creation date of the collection. The tree is automatically updated when merging a set of document collections via the `c()` function. `appendMeta()` and `removeMeta()` can also be used for adding or removing the collection metadata.

`DBControl` holds information to control database support. We use the package **filehash** (Peng, 2006) to source out text documents and the metadata data frame. Only references (i.e., keys identifying objects in the database) are kept in memory. On access objects are automatically loaded into memory

and unloaded after use. This allows to keep track of several thousand text documents. Formally, `DBControl` holds a list with three components:

`useDb` of class `logical` indicates whether the database support should be activated,

`dbName` of class `character` holds the filename holding the database, and

`dbType` of class `character` is one of the database types supported by **filehash**.

The `Corpus` constructor takes following arguments:

`object:` a `Source` object which abstracts the input location.

`readerControl:` a list with three components:

`reader` which constructs a text document from a single element delivered by a source. A reader must have the argument signature `(elem, load, language, id)`. The first argument is the element provided from the source, the second indicates the wish for immediate loading the document into memory (lazy loading), the third holds the texts' language, and the fourth is a unique identification string.

`language` describing the text documents' language (typically in ISO 639 or ISO 3166 format, e.g., `en_US`).

`load` signalizes whether the user wants to load the documents immediately into memory. Loading on demand (i.e., lazy loading) is possible if the `Source` object supports it. Typically this flag is passed over to the parsing function to activate the right bits in the reader for load on demand.

`...:` formally if the passed over `reader` object is of class `FunctionGenerator`, it is assumed to be a function generating a reader. This way custom readers taking various parameters (specified in `...`) can be built, which in fact must produce a valid reader signature but can access additional parameters via lexical scoping (i.e., by the including environment).

`dbControl:` a list with the three components `useDb`, `dbName` and `dbType` setting the respective `DBControl` values.

The next core class is a text document, i.e., the basic unit managed by a text document collection:

`TextDocument:` The `VIRTUAL` class `TextDocument` represents and encapsulates a generic text document in an abstract way. It serves as the base class for inherited classes and provides several slots for metadata:

`Author` of class `character` can be used to store information on the creator or authors of the document.

`DateTimeStamp` of class `POSIXct` holds the creation date of the document.

`Description` of class `character` may contain additional explanations on the text, like the text history or comments from reviewers, additional authors, et cetera.

`ID` of class `character` must be a unique identification string, as it is used as the main reference mechanism in exported classes, like `TermDocMatrix`.

`Origin` of class `character` provides further notes on its source, like its news agency or the information broker.

`Heading` of class `character` is typically a one-liner describing the main rationale of the document, often the title of the article, if available.

`Language` of class `character` holds the text document's language.

`LocalMetaData` of class `list` is designed to hold a list of metadata in tag-value pair format. It allows to dynamically store extra information tailored to the application range. The local metadata is conceived to hold metadata specific to single text documents besides the existing metadata slots.

The main rationale is to extend this class as needed for specific purposes. This offers great flexibility as we can handle any input format internally but provide a generic interface to other classes. The following four classes are derived classes implementing documents for common file formats:

`XMLTextDocument` inherits from `TextDocument` and `XMLDocument` (i.e., `list`), which is implemented in the **XML** package. It offers all capabilities to work with XML documents, like operations on the XML tree. It has the two slots, where

`URI` of class `ANY` holds a call (we denote it as *unique resource identifier*) which returns the document corpus if evaluated. This is necessary to implement load on demand.

`Cached` of class `logical` indicates whether the document corpus has already been loaded into memory.

`PlainTextDocument` inherits from `TextDocument` and `character`. It is the default class if no special input format is necessary. It provides the two slots `URI` and `Cached` with the same functionality as `XMLTextDocument`.

`NewsgroupDocument` inherits from `TextDocument` and `character`. It is designed to contain newsgroup postings, i.e., e-mails. Besides the basic `URI` and `Cached` slots it holds the newsgroup name of each posting in the `Newsgroup` slot.

`StructuredTextDocument:` It can be used to hold documents with sections or some structure, e.g., a `list` of paragraphs.

Another core class in our framework are term-document matrices.

`Term-Document Matrices` There is a class for term-document matrices (Berry, 2003; Shawe-Taylor and Cristianini, 2004), probably the most common way of representing texts for further computation. It can be exported from a `Corpus` and is used as a bag-of-words mechanism which means that the order of tokens is irrelevant. This approach results in a matrix with document `ID`s as rows and terms as columns. The matrix elements are term frequencies.

`TermDocMatrix` provides such a term-document matrix for a given `Corpus` element. It has the slot `Data` of the formal class `Matrix` to hold the frequencies in compressed sparse matrix format.

Instead of using the term frequency directly, one can use different weightings. `Weighting` provides this facility by calling a weighting function on the matrix elements. Available weighting schemes (let $\omega_{t,d}$ denote the weighting of term t in document d) for a given matrix M are:

- *Binary Logical* weighting (`weightLogical`) eliminates multiple frequencies and replaces them by a Boolean value, i.e.,

$$\omega_{t,d} = \begin{cases} \texttt{FALSE} & \text{if } \mathrm{tf}_{t,d} < \gamma \\ \texttt{TRUE} & \text{if } \mathrm{tf}_{t,d} \geq \gamma \end{cases} ,$$

  where γ denotes a cutoff value, typically $\gamma = 1$.
- *Binary Frequency* (`weightBin`) eliminates multiple frequencies in M, hence

$$\omega_{t,d} = \begin{cases} 0 & \text{if } \mathrm{tf}_{t,d} < \gamma \\ 1 & \text{if } \mathrm{tf}_{t,d} \geq \gamma \end{cases} ,$$

  where $\mathrm{tf}_{t,d}$ is the frequency of term t in document d and γ is again a cutoff. In other words all matrix elements are now dichotomous, reducing the frequency dimension.
- *Term Frequency* (`weightTf`) is just the identity function $\mathcal{I}$

$$\omega_{t,d} = \mathcal{I} ,$$

  as the matrix elements are term frequencies by construction.
- *Term Frequency Inverse Document Frequency* weighting (`weightTfIdf`) reduces the impact of irrelevant terms and highlights discriminative ones by normalizing each matrix element under consideration of the number of all documents, hence

$$\omega_{t,d} = \mathrm{tf}_{t,d} \cdot \log_2 \frac{m}{\mathrm{df}_t} ,$$

  where m denotes the number of rows, i.e., the number of documents, $\mathrm{tf}_{t,d}$ is the frequency of term t in document d, and df_t is the number of documents containing the term t.

The user can plug in any weighting function capable of handling sparse matrices.

Sources provide a way to abstract the input process:

`Source` is a `VIRTUAL` class and abstracts the input location and serves as the base class for creating inherited classes for specialized file formats. It has three slots,

`LoDSupport` of class `logical` indicates whether *load on demand* is supported, i.e., whether the source is capable of loading the text corpus into memory on any later request,

`Position` of class `numeric` stores the current position in the source, e.g., an index (or pointer address) to the position of the current active file,

`DefaultReader` of class `function` holds a default reader function capable of reading in objects delivered by the source, and

`Encoding` of class `character` contains the encoding to be used by internal R routines for accessing texts via the source (defaults to UTF-8 for all sources).

The following classes are specific source implementations for common purposes:

`DirSource` is designed to be used with a directory of files and has the slot `FileList` of class `character` to hold the full filenames (including path) for the files in the directory. Load on demand is supported since the files are assumed to stay in the directory and can be loaded on request.

`CSVSource` is to be used for a single CSV file where each line is interpreted as a text document. Load on demand is not supported since the whole single file would need to be traversed when accessing single lines of the file. It has the two slots `URI` of class `ANY` for holding a call and `Content` of class `list` to hold the list of character vectors (i.e., lines of the file).

`ReutersSource` should be used for handling the various Reuters file formats (e.g., the Reuters21578 collection (Lewis, 1997)) if stored in a single file (if stored separately simply use `DirSource`). Therefore load on demand is not supported. It has the slot `URI` of class `ANY` to hold a call and the `Content` of class `list` to hold the parsed XML tree.

`GmaneSource` can be used to access Gmane (Ingebrigtsen, 2007) RSS feeds.

Each source class must implement the following interface methods in order to comply with the **tm** package definitions:

`getElem()` must return the element at the current position and a URI for possible later access in form of a named list `list(content = ..., uri = ...)`.

`stepNext()` must update the position such that a subsequent `getElem()` call returns the next element in the source.

`eoi()` must indicate whether further documents can be delivered by the source, e.g., a typical end of file result if the file end has been reached.

Typically the `Position` slot of class `Source` is sufficient for storing relevant house keeping information to implement the interface methods but the user is free to use any means as long as the derived source fulfills all interface definitions.

B tm Manual

Title Text Mining Package

Version 0.3-1.7

Date 2008-07-08

Author Ingo Feinerer

Depends R (>= 2.7.0), filehash, Matrix, methods, Snowball, XML

Imports proxy

Suggests Rgraphviz, Rstem (>= 0.3-1)

Description A framework for text mining applications within R.

License GPL-2

`acq`	*50 Exemplary News Articles from the Reuters-21578 XML Data Set of Topic acq*

Description

This dataset holds 50 news articles with additional meta information from the Reuters-21578 XML data set. All documents belong to the topic `acq` dealing with corporate acquisitions.

Usage

```
data("acq")
```

Format

A text document collection of 50 text documents.

Source

Reuters-21578 Text Categorization Collection Distribution 1.0 (XML format).

References

Lewis, David (1997) *Reuters-21578 Text Categorization Collection Distribution 1.0.* http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html

Examples

```
data("acq")
summary(acq)
```

`appendElem-methods` *Methods for Function appendElem in Package 'tm'*

Description

Methods for function `appendElem` in package **tm**.

Methods

object = "Corpus", data = "TextDocument", meta = NULL Returns a text document collection where the `data` text document is added to the data slot of the `object` text document collection. Optionally meta data (`DMetaData`) can be added with `meta` to the document collection.

object = "TextRepository", data = "Corpus", meta = NULL Returns a text repository where the `data` text document collection is added to the data slot of the `object` text repository. Optionally meta data can be added with `meta` to the repository.

See Also

`DMetaData` `RepoMetaData`

Examples

```
data("acq")
data("crude")
summary(crude)
tdcl <- appendMeta(crude,
                   dmeta = list("number" = 1:20,
                                "letter" = rep(letters[1:10],2)))
summary(tdcl)
DMetaData(tdcl)
tdcl <- appendElem(tdcl, acq[[1]], c(21, letters[1]))
```

```
summary(tdcl)
DMetaData(tdcl)
```

---

appendMeta-methods

Methods for Function appendMeta in Package 'tm'

Description

Methods for function `appendMeta` in package **tm**.

Methods

object = "Corpus", cmeta = NULL, dmeta = NULL Returns a text document collection where the `cmeta` is added to the `CMetaData` and `dmeta` is added to the `DMetaData` of corpus `object`.

object = "TextRepository", cmeta = NULL, dmeta = NULL Returns a text repository where `cmeta` is added to the metadata slot of the `object` text repository.

See Also

`DMetaData` `CMetaData` `RepoMetaData`

Examples

```
data("crude")
summary(crude)
CMetaData(crude)
DMetaData(crude)
tdcl <- appendMeta(crude, cmeta = list("changed" = date()),
                          dmeta = list("number" = 1:20))
summary(tdcl)
CMetaData(tdcl)
DMetaData(tdcl)
```

`asPlain-methods` *Methods for Function asPlain in Package 'tm'*

Description

Methods for function `asPlain` in package **tm**.

Methods

object = "NewsgroupDocument" Converts `object` to a plain text document.

object = "PlainTextDocument" Returns `object`.

object = "XMLTextDocument", FUN Converts `object` to a `PlainTextDocument` by applying `FUN` on the XML root node and returns the new object.

object = "Reuters21578Document" Converts `object` to a `PlainTextDocument`.

object = "RCV1Document" Converts `object` to a `PlainTextDocument`.

object = "StructuredTextDocument" Converts `object` to a plain text document.

See Also

`convertReut21578XMLPlain` `convertRCV1Plain`

Use `getTransformations` to list available transformation (mapping) functions.

Examples

```
reut21578 <- system.file("texts", "reut21578", package = "tm")
reut21578TDC <- Corpus(DirSource(reut21578),
  readerControl = list(reader = readReut21578XML, language = "en_US", load = TRUE))
reut21578TDC[[1]]
asPlain(reut21578TDC[[1]])
```

`c-methods` *Methods for Function c in Package 'tm'*

Description

Methods for generic function `c` in package **tm**.

Details

Note that meta data from single items (corpora or documents) is preserved during concatenation and intelligently merged into the newly created corpus. Although we use a sophisticated merging strategy (by using a binary tree for corpus specific meta data and by joining document level specific meta data in data frames) you should check the newly created meta data for consistency when merging corpora with (partly) identical meta data. However, in most cases the meta data merging strategy will produce perfectly combined and arranged meta data structures.

Methods

x = "Corpus" Concatenates several text document collections to a single one.

x = "TextDocument" Concatenates several text documents to a single text document collection.

Examples

```
data("acq")
data("crude")
summary(c(acq,crude))
summary(c(acq[[30]],crude[[10]]))
```

`colnames-methods` *Methods for Function colnames in Package 'tm'*

Description

Methods for function `colnames` in package **tm**.

Methods

x = "TermDocMatrix" Retrieve the column names of the term-document matrix.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
colnames(tdm)
```

stemCompletion *Complete Stems*

Description

Heuristically complete stemmed words.

Usage

```
stemCompletion(object, words, type = c("prevalent", "first"))
```

Arguments

`object`	a `Corpus` to be searched for possible completions.
`words`	a `character` vector of stems to be completed.
`type`	a `character` naming the heuristics to be used. `prevalent` is default and takes the most frequent match as completion, whereas `first` takes the first found completion.

Value

A character vector with completed `words`.

Author(s)

Ingo Feinerer

Examples

```
data("crude")
stemCompletion(crude, c("compan", "entit", "suppl"))
```

convertMboxEml *Convert E-Mails From mbox Format To eml Format*

Description

Convert e-mails from mbox (i.e., several mails in a single box) format to eml (i.e., every mail in a single file) format.

Usage

```
convertMboxEml(mbox, EmlDir)
```

Arguments

mbox	a character or connection describing the mbox location.
EmlDir	a character describing the output directory.

Value

No explicit return value. As a side product the directory `EmlDir` contains the e-mails.

Author(s)

Ingo Feinerer

Examples

```
mbox <- system.file("texts", "gmane.comp.lang.r.general.mbox", package = "tm")
## Not run: convertMboxEml(mbox, "emldir/")
mbox.gz <- gzfile(system.file("texts","gmane.comp.lang.r.general.mbox.gz",
                              package = "tm"))
## Not run: convertMboxEml(mbox.gz, "emldir/")
```

---

convertRCV1Plain *Transform a RCV1 Document to a Plain Text Document*

Description

Transform a Reuters Corpus Volume 1 XML document to a plain text document.

Usage

```
convertRCV1Plain(node, ...)
```

Arguments

node	A `RCV1Document` representing a well-formed RCV1 XML file.
...	Arguments passed over by calling functions.

Value

A `PlainTextDocument` representing `node`.

Author(s)

Ingo Feinerer

See Also

`asPlain`

Examples

```
rcv1 <- system.file("texts", "rcv1", package = "tm")
rcv1TDC <- Corpus(DirSource(rcv1), readerControl = list(reader = readRCV1))
rcv1TDC[[1]]
asPlain(rcv1TDC[[1]], convertRCV1Plain)
```

`convertReut21578XMLPlain` *Transform a Reuters21578 XML Document to a Plain Text Document*

Description

Transform a Reuters21578 XML document to a plain text document.

Usage

```
convertReut21578XMLPlain(node, ...)
```

Arguments

`node`	an XML node representing a <REUTERS></REUTERS> element from a well-formed Reuters-21578 XML file.
`...`	Arguments passed over by calling functions.

Value

A `PlainTextDocument` representing `node`.

Author(s)

Ingo Feinerer

See Also

`asPlain`

Examples

```
reut21578 <- system.file("texts", "reut21578", package = "tm")
reut21578TDC <- Corpus(DirSource(reut21578),
  readerControl = list(reader = readReut21578XML, language = "en_US", load = TRUE))
reut21578TDC[[1]]
asPlain(reut21578TDC[[1]], convertReut21578XMLPlain)
```

`crude` *20 Exemplary News Articles from the Reuters-21578 XML Data Set of Topic crude*

Description

This data set holds 20 news articles with additional meta information from the Reuters-21578 XML data set. All documents belong to the topic `crude` dealing with crude oil.

Usage

```
data("crude")
```

Format

A text document collection of 20 text documents.

Source

Reuters-21578 Text Categorization Collection Distribution 1.0 (XML format).

References

Lewis, David (1997) *Reuters-21578 Text Categorization Collection Distribution 1.0.* http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html

Examples

```
data("crude")
summary(crude)
```

CSVSource-class *Source for Comma Separated Files*

Description

A class representing a comma separated file, where each line stands for a text document.

Objects from the Class

Objects can be created by calls of the form `new("CSVSource", ...)`.

Slots

`DefaultReader:` Object of class `function` describing a default reader

`Encoding:` Object of class `character` holding the encoding of the texts delivered by the source.

`URI:` Object of class `character` describing the connection call to be made for accessing the document physically.

`Content:` Object of class `character` holding the CSV file corpus.

`LoDSupport:` Object of class `logical` indicating Load on Demand Support.

`Position:` Object of class `numeric` giving the position in the source.

Extends

Class `Source`, directly.

Author(s)

Ingo Feinerer

CSVSource *Comma Seperated Value Source*

Description

Constructs a source for a comma separated values file.

Usage

```
## S4 method for signature 'character':
CSVSource(object, encoding = "UTF-8")
```

Arguments

`object`	either a character identifying the file or a call which results in a connection.
`encoding`	a character giving the encoding of the file.

Value

An S4 object of class `CSVSource` which extends the class `Source` representing a comma separated values file.

Author(s)

Ingo Feinerer

`Dictionary-class` *Dictionary*

Description

A class representing a dictionary, i.e., a character vector of terms.

Objects from the Class

Objects can be created by calls of the form `new("Dictionary", ...)`.

Author(s)

Ingo Feinerer

See Also

`Dictionary`

`Dictionary` *Dictionary*

Description

Constructs a dictionary from a character vector or a term-document matrix.

Usage

```
Dictionary(object)
```

Arguments

`object`	A character vector or a term-document matrix holding the terms for the dictionary.

Value

An S4 object of class `Dictionary` which extends the class `character` representing a dictionary.

Author(s)

Ingo Feinerer

Examples

```
Dictionary(c("some", "tokens"))
data(crude)
Dictionary(TermDocMatrix(crude))
```

`dimnames-methods` *Methods for Function dimnames in Package 'tm'*

Description

Methods for function `dimnames` in package **tm**.

Methods

x = "TermDocMatrix" Retrieve the dimnames of the term-document matrix.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
dimnames(tdm)
```

`dim-methods` *Methods for Function dim in Package 'tm'*

Description

Methods for function `dim` in package **tm**.

Methods

x = "TermDocMatrix" Retrieve the dimension of the term-document matrix.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
dim(tdm)
```

`DirSource-class` *Source for Directories*

Description

A class representing a directory. Each file in this directory is considered to be a document.

Objects from the Class

Objects can be created by calls of the form `new("DirSource", ...)`.

Slots

`DefaultReader:` Object of class `function` describing a default reader.

`Encoding:` Object of class `character` holding the encoding of the texts delivered by the source.

`FileList:` Object of class `character` giving the file list in the directory.

`LoDSupport:` Object of class `logical` indicating Load on Demand Support.

`Position:` Object of class `numeric` giving the position in the source.

Extends

Class `Source`, directly.

Author(s)

Ingo Feinerer

`DirSource` *Directory Source*

Description

Constructs a directory source.

Usage

```
## S4 method for signature 'character':
DirSource(directory, encoding = "UTF-8", recursive = FALSE)
```

Arguments

`directory`	a directory.
`encoding`	a character giving the encoding of the files in the directory.
`recursive`	a logical value indicating whether the directory should be traversed recursively.

Value

An S4 object of class `DirSource` which extends the class `Source` representing a directory.

Author(s)

Ingo Feinerer

`dissimilarity-methods`
Methods for Function dissimilarity in Package 'tm'

Description

Methods for function `dissimilarity` in package **tm**.

Methods

x = "TermDocMatrix", y = "ANY", method = "character" Computes the dissimilarity between the documents in `x` with the distance measure `method`. Any method accepted by `dist` from package **proxy** can be passed over.

Formally the dissimilarity is returned in form of a `dist` object.

x = "TextDocument", y = "TextDocument", method = "character" Computes the dissimilarity between `x` and `y` with the distance measure `method`. Any method accepted by `dist` from package **proxy** can be passed over.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
dissimilarity(tdm, method = "cosine")
dissimilarity(crude[[1]], crude[[2]], method = "eJaccard")
```

`DublinCore-methods`
Methods for Function DublinCore in Package 'tm'

Description

Methods for function `DublinCore` in package **tm**, which provide a bijective mapping between Simple Dublin Core meta data and **tm** meta data structures.

Methods

object = "TextDocument", tag = NULL Returns or sets the Simple Dublin Core meta datum named `tag` for `object`. `tag` must be a valid Simple Dublin Core element name (i.e, Title, Creator, Subject, Description, Publisher, Contributor, Date, Type, Format, Identifier, Source, Language, Relation, Coverage, or Rights) or `NULL`. For the latter all Dublin Core meta data are printed.

Examples

```
data("crude")
DublinCore(crude[[1]])
meta(crude[[1]])
DublinCore(crude[[1]], tag = "Creator") <- "Ano Nymous"
DublinCore(crude[[1]], tag = "Format") <- "XML"
DublinCore(crude[[1]])
meta(crude[[1]])
```

---

eoi-methods *Methods for Function eoi in Package 'tm'*

Description

Methods for function `eoi` in package **tm**..

Methods

object = "DirSource" returns `TRUE` if the last item has been reached.

object = "CSVSource" returns `TRUE` if the last item has been reached.

object = "ReutersSource" returns `TRUE` if the last item has been reached.

object = "GmaneSource" returns `TRUE` if the last item has been reached.

object = "VectorSource" returns `TRUE` if the last item has been reached.

---

findAssocs-methods *Methods for Function findAssocs in Package 'tm'*

Description

Methods for function `findAssocs` in package **tm**.

Methods

object = "TermDocMatrix", term = "character", corlimit Finds those terms from `object` which correlate with `term` more than `corlimit`.

object = "matrix", term = "character", corlimit Finds those terms from the correlation matrix `object` which correlate with `term` more than `corlimit`.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
findAssocs(tdm, "oil", 0.7)
```

`findFreqTerms-methods`
Methods for Function findFreqTerms in Package 'tm'

Description

Methods for function `findFreqTerms` in package **tm**.

Methods

object = "TermDocMatrix", lowfreq = 0, highfreq = Inf Return those terms from `object` which occur more or equal often than `lowfreq` times and less or equal often than `highfreq` times. This method works for all numeric weightings but is probably most meaningful for the standard term frequency (`tf`) weighting of `object`.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
findFreqTerms(tdm, 2, 3)
```

`FunctionGenerator-class`
Function Generator

Description

A class representing a function generator.

Objects from the Class

Objects can be created by calls of the form `new("FunctionGenerator", ...)`.

Extends

Class `function`, directly.

Author(s)

Ingo Feinerer

`FunctionGenerator` *Function Generator Constructor*

Description

Constructs a function generator object.

Usage

```
## S4 method for signature 'function':
FunctionGenerator(object)
```

Arguments

`object`	a generator function which takes some input and constructs and returns a new function based on that input information.

Value

An S4 object of class `FunctionGenerator` which extends the class `function` representing a function generator.

Author(s)

Ingo Feinerer

See Also

Many reader functions are function generators, e.g., `readPlain`.

Examples

```
funGen <- FunctionGenerator(function(y, ...) {
if (is(y, "integer")) function(x) x+1 else function(x) x-1
})
funGen
funGen(3L)
funGen("a")
```

`getElem-methods` *Methods for Function getElem in Package 'tm'*

Description

Methods for function `getElem` in package **tm**.

Methods

object = "DirSource" Returns the element at current position from the source.
object = "CSVSource" Returns the element at current position from the source.
object = "ReutersSource" Returns the element at current position from the source.
object = "GmaneSource" Returns the element at current position from the source.
object = "VectorSource" Returns the element at current position from the source.

`getFilters` *Get Available Filters*

Description

Get available filters.

Usage

```
getFilters()
```

Value

A character vector with available filters.

Author(s)

Ingo Feinerer

Examples

```
getFilters()
```

`getReaders` *Get Available Readers*

Description

Get available readers.

Usage

```
getReaders()
```

Value

A character vector with available readers.

Author(s)

Ingo Feinerer

Examples

```
getReaders()
```

`getSources` *Get Available Sources*

Description

Get available sources.

Usage

```
getSources()
```

Value

A character vector with available sources.

Author(s)

Ingo Feinerer

Examples

```
getSources()
```

`getTransformations` *Get Available Transformations*

Description

Get available transformations (mappings).

Usage

```
getTransformations()
```

Value

A character vector with available transformations.

Author(s)

Ingo Feinerer

Examples

```
getTransformations()
```

`GmaneSource-class` *Source for Gmane Feeds*

Description

A class representing a Gmane RSS feed.

Objects from the Class

Objects can be created by calls of the form `new("GmaneSource", ...)`.

Slots

`DefaultReader:` Object of class `function` describing a default reader.

`Encoding:` Object of class `character` holding the encoding of the texts delivered by the source.

`URI:` Object of class `character` describing the connection call to be made for accessing the document physically.

`Content:` Object of class `character` holding the RSS feed.

`LoDSupport:` Object of class `logical` indicating Load on Demand Support.

`Position:` Object of class `numeric` giving the position in the source.

Extends

Class `Source`, directly.

Author(s)

Ingo Feinerer

`GmaneSource` *Gmane Source*

Description

Constructs a source for a Gmane mailing list RSS feed.

Usage

```
## S4 method for signature 'character':
GmaneSource(object, encoding = "UTF-8")
```

Arguments

`object`	either a character identifying the file or a call which results in a connection.
`encoding`	a character giving the encoding of the file.

Value

An S4 object of class `GmaneSource` which extends the class `Source` representing a Gmane mailing list RSS feed.

Author(s)

Ingo Feinerer

`%IN%-methods` *Methods for Function %IN% in Package 'tm'*

Description

Methods for function `%IN%` in package **tm**.

Methods

x = "TextDocument", y = "Corpus" Returns `TRUE` if `x` occurs in `y`.

Examples

```
data("crude")
crude[[1]] %IN% crude
```

`inspect-methods` *Methods for Function inspect in Package 'tm'*

Description

Methods for function `inspect` in package **tm**.

Methods

object = "Corpus" Displays detailed information for `object`, i.e., it prints all text documents and metadata.

Examples

```
data("crude")
inspect(crude[1:3])
```

`length-methods` *Methods for Function length in Package 'tm'*

Description

Methods for function `length` in package **tm**.

Methods

x = "Corpus" Returns the number of text documents in `x`.

x = "TextRepository" Returns the number of text document collections in `x`.

Examples

```
data("crude")
length(crude)
length(TextRepository(crude))
```

`loadDoc-methods` *Methods for Function loadDoc in Package 'tm'*

Description

Methods for function `loadDoc` in package **tm**.

Methods

object = "PlainTextDocument" Loads the text corpus of `object` from disk into memory and return the new object.

object = "XMLTextDocument" Loads the text corpus of `object` from disk into memory and return the new object.

object = "NewsgroupDocument" Loads the text corpus of `object` from disk into memory and return the new object.

object = "StructuredTextDocument" Loads the text corpus of `object` from disk into memory and return the new object.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

`makeChunks` *Split a Corpus into Chunks*

Description

Split a corpus into equally sized chunks conserving document boundaries.

Usage

```
makeChunks(corpus, chunksize)
```

Arguments

`corpus`	The corpus to be split into chunks.
`chunksize`	The chunk size.

Value

A corpus consisting of the chunks. Note that corpus meta data is not passed on to the newly created chunk corpus.

Author(s)

Ingo Feinerer

Examples

```
txt <- system.file("texts", "txt", package = "tm")
ovid <- Corpus(DirSource(txt))
sapply(ovid, length)
ovidChunks <- makeChunks(ovid, 5)
sapply(ovidChunks, length)
```

`materialize` *Materialize Lazy Mappings*

Description

The function `tmMap` supports so-called lazy mappings, that are mappings which are delayed until the documents' content is accessed. This function triggers the evaluation, i.e., it materializes the documents.

Usage

```
materialize(corpus, range = seq_along(corpus))
```

Arguments

`corpus`	A document collection with lazy mappings.
`range`	The indices of documents to be materialized.

Value

A corpus with materialized, i.e., all mappings computed and applied, documents for the requested range.

Author(s)

Ingo Feinerer

See Also

`tmMap`

Examples

```
data("crude")
x <- tmMap(crude, stemDoc, lazy = TRUE)
x <- materialize(x)
```

`MetaDataNode-class` *Metadata Node*

Description

A class representing a node in the metadata tree of a text document collection.

Objects from the Class

Objects can be created by calls of the form `new("MetaDataNode",...)`.

Slots

`NodeID`: Object of class `numeric` containing a unique identification number for the node.

`MetaData`: Object of class `list` containing meta data at node level.

`children`: Object of class `list` containing either zero, one or two `MetaDataNode`s resulting in a binary tree.

Author(s)

Ingo Feinerer

`meta-methods` *Methods for Function meta in Package 'tm'*

Description

Methods for function `meta` in package **tm**.

Methods

object = "TextDocument", tag = NULL If no `tag` is given, this method pretty prints all `object`'s meta data. If `tag` is provided its value in the meta data is returned.

object = "Corpus", tag = NULL, type = "indexed" This method investigates the `type` argument. `type` must be either `indexed` (default), `local`, or `corpus`. Former is a shortcut for accessing document level meta data (`DMetaData`) stored at the collection level (because it forms an own entity, or for performance reasons, i.e., a form of indexing, hence the name indexed), `local` accesses the meta data local to each text document (i.e., meta data in text documents' S4 slots), and `corpus` is a shortcut for collection (corpus) specific meta data (`CMetaData`). Depending whether a tag is set or not, all or only the meta data identified by the tag is displayed or modified.

object = "TextRepository", tag = NULL If no `tag` is given, this method pretty prints all `object`'s meta data. If `tag` is provided its value in the meta data is returned.

Examples

```
data("crude")
meta(crude[[1]])
meta(crude[[1]], tag = "Topics")
meta(crude[[1]], tag = "Comment") <- "A short comment."
meta(crude[[1]])
meta(crude)
meta(crude, type = "corpus")
meta(crude, "labels") <- 21:40
meta(crude)
```

`ncol-methods` *Methods for Function ncol in Package 'tm'*

Description

Methods for function `ncol` in package **tm**.

Methods

x = "TermDocMatrix" Retrieve the number of columns of the term-document matrix.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
ncol(tdm)
```

NewsgroupDocument-class

Newsgroup Text Document

Description

A class representing a newsgroup document with additional information. The newsgroup documents must be formatted according to the Newsgroup dataset from the UCI KDD archive.

Objects from the Class

Objects can be created by calls of the form `new("NewsgroupDocument", ...)`.

Slots

`Newsgroup`: Object of class `character` containing the newsgroups where the document has been posted.

`URI`: Object of class `character` containing the path and filename holding the data physically on disk.

`Cached`: Object of class `logical` containing the status whether the file was already loaded into memory.

Extends

Class `character` and `TextDocument`, directly.

Methods

Content `signature(object = "NewsgroupDocument")`: Returns the text corpus, i.e., the actual character data slot.

Content`<-` `signature(object = "NewsgroupDocument")`: Sets the text corpus, i.e., the actual character data slot.

URI `signature(object = "NewsgroupDocument")`: Returns the filename on disk.

Cached `signature(object = "NewsgroupDocument")`: Returns status information for loading on demand.

Cached`<-` `signature(object = "NewsgroupDocument")`: Sets status information for loading on demand.

Author(s)

Ingo Feinerer

References

http://kdd.ics.uci.edu/databases/20newsgroups/20newsgroups.html

---

`nrow-methods` *Methods for Function nrow in Package 'tm'*

Description

Methods for function `nrow` in package **tm**.

Methods

x = "TermDocMatrix" Retrieve the number of rows of the term-document matrix.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
nrow(tdm)
```

`PlainTextDocument-class`
Plain Text Document

Description

A class representing a plain text document with additional information.

Objects from the Class

Objects can be created by calls of the form `new("PlainTextDocument", ...)`.

Slots

`URI:` Object of class `character` containing the path and filename holding the data physically on disk.

`Cached:` Object of class `logical` containing the status whether the file was already loaded into memory.

Extends

Class `character` and `TextDocument`, directly.

Methods

Content `signature(object = "PlainTextDocument")`: Returns the text corpus, i.e., the actual character data slot.

Content<- `signature(object = "PlainTextDocument")`: Sets the text corpus, i.e., the actual character data slot.

URI `signature(object = "PlainTextDocument")`: Returns the filename on disk.

Cached `signature(object = "PlainTextDocument")`: Returns status information for loading on demand.

Cached<- `signature(object = "PlainTextDocument")`: Sets status information for loading on demand.

Author(s)

Ingo Feinerer

`plot` *Visualize a Term-Document Matrix*

Description

Visualize correlations between terms of a term-document matrix.

Usage

```
## S3 method for class 'TermDocMatrix':
plot(x,
     terms = sample(colnames(x), 20),
     corThreshold = 0.7,
     weighting = FALSE,
     attrs = list(graph = list(rankdir = "BT"),
                  node = list(shape = "rectangle",
                              fixedsize = FALSE)),
     ...)
```

Arguments

`x`	an R object inheriting from S4 class `TermDocMatrix`.
`terms`	Terms to be plotted. Defaults to 20 randomly chosen terms of the term-document matrix.
`corThreshold`	Do not plot correlations below this threshold. Defaults to `0.7`.

`weighting`	Define whether the line width corresponds to the correlation.
`attrs`	Argument passed to the plot method for class `graphNEL`.
`...`	Other arguments passed to the `graphNEL` plot method.

Details

Visualization requires that package **Rgraphviz** is available.

Examples

```
if (require("Rgraphviz")) {
  data(crude)
  tdm <- TermDocMatrix(crude)
  plot(tdm, terms = colnames(tdm)[1:20])
  set.seed(1234)
  plot(tdm, corThreshold = 0.2, weighting = TRUE)
}
```

`preprocessReut21578XML`

Preprocess the Reuters21578 XML archive.

Description

Preprocess the Reuters21578 XML archive by correcting invalid UTF-8 encodings and copying each text document into a separate file.

Usage

```
preprocessReut21578XML(ReutersDir, ReutersOapfDir, fixEnc = TRUE)
```

Arguments

`ReutersDir`	a character describing the input directory.
`ReutersOapfDir`	a character describing the output directory.
`fixEnc`	a logical value indicating whether a invalid UTF-8 encoding in the Reuters21578 XML dataset should be corrected.

Value

No explicit return value. As a side product the directory `ReutersOapfDir` contains the corrected dataset.

Author(s)

Ingo Feinerer

References

Lewis, David (1997) *Reuters-21578 Text Categorization Collection Distribution 1.0*. http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html

Luz, Saturnino *XML-encoded version of Reuters-21578*. http://modnlp.berlios.de/reuters21578.html

---

`prescindMeta-methods`

Methods for Function prescindMetadata in Package 'tm'

Description

Methods for function `prescindMeta` in package **tm**.

Value

A `Corpus` constructed from `object` with shifted up `meta` data.

Methods

object = "Corpus", meta = "character" Copies the `meta` data from document slots or (local) document meta data of all documents in `object` to the (global) `DMetaData` data frame of `object` and returns the whole collection.

See Also

`DMetaData`

Examples

```
data("crude")
DMetaData(crude)
DMetaData(prescindMeta(crude, c("ID", "Heading")))
```

`RCV1Document-class` *RCV1 Text Document*

Description

A class representing a RCV1 XML text document with additional information. The XML document itself is represented by inheriting from `XMLTextDocument`.

Objects from the Class

Objects can be created by calls of the form `new("RCV1Document", ...)`.

Extends

Class `XMLTextDocument`, directly.

Author(s)

Ingo Feinerer

`readDOC` *Read In a MS Word Document*

Description

Returns a function which reads in a Microsoft Word document extracting its text.

Usage

```
readDOC(...)
```

Arguments

`...`	Arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in a text document) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Note that this MS Word reader needs the tool `antiword` installed and accessable on your system.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `PlainTextDocument` representing the text in `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

`readGmane` *Read In A Newsgroup Document*

Description

Returns a function which reads in a RSS feed as returned by Gmane for mailing lists.

Usage

```
readGmane(...)
```

Arguments

`...`	arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in an RSS feed) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `NewsgroupDocument` representing `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

`readHTML` *Read In a Simple HTML Document*

Description

Returns a function which reads in a simple HTML document extracting both its text and its metadata. The reader uses `h1` headings as structure information whereas text and tags between headings are considered as textual information. Meta data is extracted from `meta` tags in the HTML head.

Usage

```
readHTML(...)
```

Arguments

`...`	arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in a text document) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `StructuredTextDocument` representing `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

Examples

```
html <- system.file("texts", "html", package = "tm")
## Not run: (Corpus(DirSource(html),
  readerControl = list(reader = readHTML, load = TRUE)))
```

`readNewsgroup` *Read In a Newsgroup Document*

Description

Returns a function which reads in a newsgroup document as found in the UCI KDD newsgroup data set.

Usage

```
readNewsgroup(...)
```

Arguments

`...`	arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in a newsgroup document) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `NewsgroupDocument` representing `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

`readPDF` *Read In a PDF Document*

Description

Returns a function which reads in a portable document format (PDF) document extracting both its text and its meta data.

Usage

```
readPDF(...)
```

Arguments

`...`	Arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in a text document) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Note that this PDF reader needs both the tools `pdftotext` and `pdfinfo` installed and accessable on your system.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `PlainTextDocument` representing the text and meta data in `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

`readPlain` *Read In a Text Document*

Description

Returns a function which reads in a text document without knowledge about its internal structure and possible available metadata.

Usage

```
readPlain(...)
```

Arguments

`...`	arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in a text document) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `PlainTextDocument` representing `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

Examples

```
docs <- c("This is a text.", "This another one.")
vs <- VectorSource(docs)
elem <- getElem(stepNext(vs))
(result <- readPlain()(elem, TRUE, "en", "id1"))
meta(result)
```

`readRCV1` *Read In a Reuters Corpus Volume 1 Document*

Description

Returns a function which reads in a Reuters Corpus Volume 1 XML document.

Usage

```
readRCV1(...)
```

Arguments

...	Arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in a Reuters Corpus Volume 1 XML document) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `XMLTextDocument` representing `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

`readReut21578XML` *Read In a Reuters21578 XML Document*

Description

Returns a function which reads in a Reuters21578 XML document.

Usage

```
readReut21578XML(...)
```

Arguments

`...`	Arguments for the generator function.

Details

Formally this function is a function generator, i.e., it returns a function (which reads in a Reuters21578 XML document) with a well-defined signature, but can access passed over arguments via lexical scoping. This is especially useful for reader functions for complex data structures which need a lot of configuration options.

Value

A `function` with the signature `elem, language, load, id`:

`elem`	A `list` with the two named elements `content` and `uri`. The first element must hold the document to be read in, the second element must hold a call to extract this document. The call is evaluated upon a request for load on demand.
`language`	A `character` vector giving the text's language.
`load`	A `logical` value indicating whether the document corpus should be immediately loaded into memory.
`id`	A `character` vector representing a unique identification string for the returned text document.

The function returns a `XMLTextDocument` representing `content`.

Author(s)

Ingo Feinerer

See Also

Use `getReaders` to list available reader functions.

`removeCitation-methods`

Methods for Function removeCitation in Package 'tm'

Description

Methods for function `removeCitation` in package **tm**.

Methods

object = "PlainTextDocument" Remove citations from e-mail messages beginning with > and return the object.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

`removeMeta-methods`
Methods for Function removeMeta in Package 'tm'

Description

Methods for function `removeMeta` in package **tm**.

Methods

object = "Corpus", cname = NULL, dname = NULL Returns a text document collection where the `cname` is removed from the `CMetaData` and `dname` is removed from the `DMetaData` of corpus `object`.

object = "TextRepository", cname = NULL, dname = NULL Returns a text repository where `cname` is removed from the meta data slot of the `object` text repository.

See Also

`DMetaData` `CMetaData` `RepoMetaData`

Examples

```
data("crude")
CMetaData(crude)
DMetaData(crude)
tdcl <- appendMeta(crude, cmeta = list(created = date()),
                          dmeta = list(numbers = 1:20))
CMetaData(tdcl)
DMetaData(tdcl)
tdcl <- removeMeta(crude, cname = "created", dname = "numbers")
CMetaData(tdcl)
DMetaData(tdcl)
```

`removeMultipart-methods`
Methods for Function removeMultipart in Package 'tm'

Description

Methods for function `removeMultipart` in package **tm**.

Methods

object = "PlainTextDocument" Extracts only the text/plain parts out of multipart e-mail messages and return the object.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

`removeNumbers-methods`
Methods for Function removeNumbers in Package 'tm'

Description

Methods for function `removeNumbers` in package **tm**.

Methods

object = "PlainTextDocument", ... Strips any numbers from `object` and return the object.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

Examples

```
data("crude")
crude[[1]]
removeNumbers(crude[[1]])
```

`removePunctuation-methods`
Methods for Function removePunctuation in Package 'tm'

Description

Methods for function `removePunctuation` in package **tm**.

Methods

object = "PlainTextDocument" Removes all punctuation marks from `object` and return the object.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

Examples

```
data("crude")
crude[[1]]
removePunctuation(crude[[1]])
```

`removeSignature-methods`
Methods for Function removeSignature in Package 'tm'

Description

Methods for function `removeSignature` in package **tm**.

Methods

object = "PlainTextDocument" Removes signature lines from an e-mail message and return the object. Additional signature identification marks can be provided via the optional `marks` parameter in form of regular expression patterns.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

Examples

```
newsgroup <- system.file("texts", "newsgroup", package = "tm")
news <- Corpus(DirSource(newsgroup),
  readerControl = list(reader = readNewsgroup, language = "en_US", load = TRUE))
asPlain(news[[4]])
removeSignature(asPlain(news[[4]]))
asPlain(news[[5]])
removeSignature(asPlain(news[[5]]), marks = "^[+]-*[+]$")
```

`removeSparseTerms-methods`
Methods for Function removeSparseTerms in Package 'tm'

Description

Methods for function `removeSparseTerms` in package **tm**.

Methods

object = "TermDocMatrix", sparse = "numeric" Return `object` where those columns (i.e., terms) from `Data(object)` are removed which have at least a `sparse` percentage of empty (i.e., terms occurring 0 times in a document) elements. I.e., the resulting matrix contains only columns with a sparse factor of less than `sparse`.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
removeSparseTerms(tdm, 0.2)
```

`removeWords-methods`
Methods for Function removeWords in Package 'tm'

Description

Methods for function `removeWords` in package **tm**.

Methods

object = "PlainTextDocument", words = "character" Removes all words which occur in `words` from `object` and return the object.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

Examples

```
data("crude")
crude[[1]]
removeWords(crude[[1]], stopwords("english"))
```

`replacePatterns-methods`
Methods for Function replacePatterns in Package 'tm'

Description

Methods for function `replacePatterns` in package **tm**.

Methods

object = "PlainTextDocument", patterns = "character", by = "character" Replaces all `patterns` (in POSIX 1003.2 extended regular expressions format, as used by `gsub`) which occur in `object` with `by` and return the object.

See Also

Use `getTransformations` to list available transformation (mapping) functions.

Examples

```
data("crude")
crude[[1]]
replacePatterns(crude[[1]], "[[:space:]]((C|c)orp|copany)[[:space:]]", " company ")
```

`Reuters21578Document-class`
Reuters21578 Text Document

Description

A class representing a Reuters21578 XML text document with additional information. The XML document itself is represented by inheriting from `XMLTextDocument`.

Objects from the Class

Objects can be created by calls of the form `new("Reuters21578Document", ...)`.

Extends

Class `XMLTextDocument`, directly.

Author(s)

Ingo Feinerer

`ReutersSource-class` *Source for Reuters Files*

Description

A class representing either a Reuters21578 or RCV1 source, where several documents are found in a single source (i.e., typically a file).

Objects from the Class

Objects can be created by calls of the form `new("ReutersSource", ...)`.

Slots

`DefaultReader:` Object of class `function` describing a default reader.

`Encoding:` Object of class `character` holding the encoding of the texts delivered by the source.

`URI:` Object of class `character` describing the connection call to be made for accessing the document physically.

`Content:` Object of class `list` holding the Reuters XML source corpus.

`LoDSupport:` Object of class `logical` indicating Load on Demand Support.

`Position:` Object of class `numeric` giving the position in the source.

Extends

Class `Source`, directly.

Author(s)

Ingo Feinerer

`ReutersSource` *Reuters Source*

Description

Constructs a source for a Reuters XML document.

Usage

```
## S4 method for signature 'character':
ReutersSource(object, encoding = "UTF-8")
```

Arguments

`object`	Either a character identifying the file or a call which results in a connection.
`encoding`	A character giving the encoding of the file.

Value

An S4 object of class `ReutersSource` which extends the class `Source` representing a Reuters XML document.

Author(s)

Ingo Feinerer

`rownames-methods` *Methods for Function rownames in Package 'tm'*

Description

Methods for function `rownames` in package **tm**.

Methods

x = "TermDocMatrix" Retrieve the row names of the term-document matrix `x`.

Examples

```
data("crude")
tdm <- TermDocMatrix(crude)
rownames(tdm)
```

searchFullText-methods

Methods for Function searchFullText in Package 'tm'

Description

Methods for function `searchFullText` in package **tm**.

Methods

object = "PlainTextDocument", pattern = "character" Return `TRUE` if the regular expression `pattern` matches in `object`'s text.

See Also

Use `getFilters` to list available filter functions.

Examples

```
data("crude")
searchFullText(crude[[1]], "co[m]?pany")
```

---

sFilter *Statement Filter*

Description

Filter meta data by special statements.

Usage

```
sFilter(object, s, ...)
```

Arguments

`object`	A `Corpus` to be the filter applied to.
`s`	A statement of format `"tag1 == 'expr1' & tag2 == 'expr2' & ..."`.
`...`	Arguments passed over by calling functions.

Details

The statement `s` models a simple query language. It consists of an expression as passed over to a data frame for subsetting. Tags in `s` represent meta data variables. Variables only available at document level are shifted up to the data frame if necessary. Note that the meta data tags for the slots `Author`, `DateTimeStamp`, `Description`, `ID`, `Origin` and `Heading` are `author`, `datetimestamp`, `description`, `identifier`, `origin` and `heading`, respectively, to avoid name conflicts.

Value

A logical vector to represent the subset of the `DMetaData` (extended for shifted up variables) data frame as specified by the statement.

Author(s)

Ingo Feinerer

See Also

Use `getFilters` to list available filter functions.

Examples

```
## Load example dataset
data("crude")

## Returning TRUE for document with ID 127 and heading DIAMOND
sFilter(crude, "identifier == '127' &
                heading == 'DIAMOND SHAMROCK (DIA) CUTS CRUDE PRICES'")
```

`show-methods` *Methods for Function show in Package 'tm'*

Description

Methods for generic function `show` in package **tm**.

Methods

object = "PlainTextDocument" Prints the text of a plain text document without any slots.

object = "Corpus" Prints a short descriptive overview of the corpus.

object = "TextRepository" Prints a short descriptive overview of the repository.

Examples

```
data("crude")
show(crude[[1]])
```

`Source-class` *Source Manager*

Description

A class representing a source, e.g., a directory, a connection, et cetera.

Objects from the Class

Objects can be created by calls of the form `new("Source", ...)`.

Slots

`DefaultReader:` Object of class `function` describing a default reader.

`LoDSupport:` Object of class `logical` indicating where this source supports load on demand.

`Position:` Object of class `numeric` indicating the position in the source.

`Encoding:` Object of class `character` holding the encoding of the texts delivered by the source.

Author(s)

Ingo Feinerer

`stemDoc-methods` *Methods for Function stemDoc in Package 'tm'*

Description

Methods for function `stemDoc` in package **tm**.

Methods

object = "PlainTextDocument" Stems the corpus from `object` with Porter's stemming algorithm.

Examples

```
data("crude")
crude[[1]]
stemDoc(crude[[1]])
```

`stepNext-methods` *Methods for Function stepNext in Package 'tm'*

Description

Methods for function `stepNext` in package **tm**..

Methods

object = "DirSource" Jumps to the next element in the source.
object = "CSVSource" Jumps to the next element in the source.
object = "ReutersSource" Jumps to the next element in the source.
object = "GmaneSource" Jumps to the next element in the source.
object = "VectorSource" Jumps to the next element in the source.

`stopwords` *Multilingual Stopwords*

Description

Return stopwords in different languages.

Usage

```
stopwords(language = "en")
```

Arguments

`language`	A `character` giving the desired language.

Details

At the moment supported languages are `danish`, `dutch`, `english`, `finnish`, `french`, `german`, `hungarian`, `italian`, `norwegian`, `portuguese`, `russian`, `spanish`, and `swedish`. Alternatively, their ISO 639-1 code may be used.

Value

A character vector containing the requested stopwords.

Examples

```
stopwords("english")
```

`stripWhitespace-methods`
Methods for Function stripWhitespace in Package 'tm'

Description

Methods for function `stripWhitespace` in package **tm**.

Methods

object = "PlainTextDocument", ... Strips unnecessary whitespace from `object` and return the object.

Examples

```
data("crude")
crude[[1]]
stripWhitespace(crude[[1]])
```

`StructuredTextDocument-class`
Structured Text Document

Description

A class representing a structured text with additional information.

Objects from the Class

Objects can be created by calls of the form `new("StructuredTextDocument", ...)`.

Slots

`URI:` Object of class `character` containing the path and filename holding the data physically on disk.

`Cached:` Object of class `logical` containing the status whether the file was already loaded into memory.

Extends

Class `list` and `TextDocument`, directly.

Methods

Content `signature(object = "StructuredTextDocument")`: Returns the text corpus, i.e., the actual character data slot.

Content<- `signature(object = "StructuredTextDocument")`: Sets the text corpus, i.e., the actual character data slot.

URI `signature(object = "StructuredTextDocument")`: Returns the filename on disk.

Cached `signature(object = "StructuredTextDocument")`: Returns status information for loading on demand.

Cached<- `signature(object = "StructuredTextDocument")`: Sets status information for loading on demand.

Author(s)

Ingo Feinerer

`[-methods` *Methods for Subset Functions in Package 'tm'*

Description

Methods for subset functions `[`, `[<-`, `[[` and `[[<-` in package **tm**.

Arguments

`x = "Corpus"`
: The text document collection to be subsetted or modified.

`x = "TermDocMatrix"`
: The term-document matrix to be subsetted.

Examples

```
data("crude")
summary(crude[10:15])
crude[[1]]
```

`summary-methods` *Methods for Function summary in Package 'tm'*

Description

Methods for function `summary` in package **tm**.

Methods

object = "Corpus" Prints a description including information on available meta data tags.

object = "TextRepository" Prints a description including information on available meta data tags.

Examples

```
data("crude")
summary(crude)
```

`TermDocMatrix-class`

Term-Document Matrix

Description

A class representing a sparse term-document matrix. A column represents a term, and a row the ID of a text document, respectively.

Objects from the Class

Objects can be created by calls of the form `new("TermDocMatrix",...)` or by calling the function `TermDocMatrix`.

Slots

`Data`: Object of class `Matrix` holding the sparse matrix.

`Weighting`: Object of class `character` containing the mode which was applied on the matrix. Possible are term frequency `"term frequency"`, term frequency-inverse document frequency `"term frequency - inverse document frequency"`, binary `"binary"` and logical `"logical"`.

Methods

Data `signature(object = "TermDocMatrix")`: Return the sparse matrix.

Data<- `signature(object = "TermDocMatrix")`: Sets the sparse matrix.

Weighting `signature(object = "TermDocMatrix")`: Returns the weighting mode.

Weighting<- `signature(object = "TermDocMatrix")`: Sets the weighting mode.

Author(s)

Ingo Feinerer

See Also

`TermDocMatrix`

`TermDocMatrix` *Term-Document Matrix*

Description

Constructs a term-document matrix.

Usage

```
TermDocMatrix(object, control = list())
```

Arguments

`object`	a text document collection
`control`	a named list of control options. The component `weighting` must be a weighting function capable of handling a `dgCMatrix`. It defaults to `weightTf` for term frequency weighting. All other options are delegated internally to a `termFreq` call.

Value

An S4 object of class `TermDocMatrix` containing a sparse term-document matrix. The following slots contain useful information:

`Data`	The sparse `Matrix`.
`Weighting`	The weighting mode applied to the term-document matrix.

Author(s)

Ingo Feinerer

See Also

The documentation to `termFreq` gives an extensive list of possible options.

Available weighting functions shipped with this package are `weightTf`, `weightTfIdf`, `weightBin` and `weightLogical`.

Examples

```
data("crude")
(tdm <- TermDocMatrix(crude, list(weighting = weightTfIdf, stopwords = TRUE)))
```

`termFreq` *Term Frequency Vector*

Description

Generate a term frequency vector from a text document.

Usage

```
termFreq(doc, control = list())
```

Arguments

`doc`	An object inheriting from `TextDocument`.
`control`	A list of control options. Possible settings are • `tolower`: A function converting characters to lower case. Defaults to `base::tolower`. • `tokenize`: A function tokenizing documents to single tokens. Defaults to `function(x) unlist(strsplit(gsub("[^[:alnum:]]+", " ", x), " ", fixed = TRUE)`.

- `removeNumbers`: A logical value indicating whether numbers should be removed from `doc`. Defaults to `FALSE`.
- `stemming`: A Boolean value indicating whether tokens should be stemmed. Defaults to `FALSE`.
- `stopwords`: Either a Boolean value indicating stopword removal using default language specific stopword lists shipped with this package or a character vector holding custom stopwords. Defaults to `FALSE`.
- `dictionary`: A character vector to be tabulated against. No other terms will be listed in the result. Defaults to no action (i.e., all terms are considered).
- `minDocFreq`: An integer value. Words that appear less often in `doc` than this number are discarded. Defaults to `1` (i.e., every token will be used).
- `minWordLength`: An integer value. Words smaller than this number are discarded. Defaults to length `3`.

Value

A named integer vector with term frequencies as values and tokens as names.

Examples

```
data("crude")
termFreq(crude[[1]])
termFreq(crude[[1]], control = list(stemming = TRUE, minWordLength = 4))
```

`Corpus-class` *Corpus*

Description

A class representing a collection of text documents (denoted as corpus in linguistics).

Objects from the Class

Objects can be created by calls of the form `new("Corpus",...)` or by calling the function `Corpus`.

Slots

`CMetaData`: Object of class `MetaDataNode` containing the document collection (corpus) specific meta data for the collection in form of tag-value pairs and information about children in form of a binary tree. This information is useful for reconstructing meta data after e.g. merging document collections.

`DMetaData`: Object of class `data.frame` containing the document specific meta data for the collection. This dataframe typically encompasses clustering or classification results which basically are metadata for documents but form an own entity (e.g., with its name, the value range, etc.).

`DBControl`: Object of class `list` with three named components: `useDb` indicates whether database support is activated, `dbName` holds the path to the database storage, and `dbType` stores the database type.

Extends

Class `list`, directly.

Methods

CMetaData `signature(object = "Corpus")`: Returns the corpus specific meta-data in form of a tag-value paired list.

DMetaData `signature(object = "Corpus")`: Returns the document specific meta-data in form of a data frame.

DBControl `signature(object = "Corpus")`: Returns the database configuration settings.

Author(s)

Ingo Feinerer

See Also

`MetaDataNode-class` `Corpus`

`Corpus` *Corpus*

Description

Constructs a text document collection (corpus).

Usage

```
## S4 method for signature 'Source':
Corpus(object, readerControl = list(reader = object@DefaultReader,
language = "en_US", load = TRUE), dbControl = list(useDb = FALSE, dbName = "",
dbType = "DB1"), ...)
```

Arguments

`object`	A `Source` object.
`readerControl`	A list with the named components `reader` representing a reading function capable of handling the file format found in `object`, `language` giving the text's language (preferably in ISO 639-1 format), and `load` being a logical value indicating whether the text corpus of documents should be loaded immediately into memory (`load = TRUE`) or loaded when necessary (`load = FALSE`). This allows to minimize memory demands for large document collections. If `object` does not support load on demand the text corpus is automatically loaded, i.e., this argument is overruled.
`dbControl`	A list with the named components `useDb` indicating that database support should be activated, `dbName` giving the filename holding the sourced out objects (i.e., the database), and `dbType` holding a valid database type as supported by package **filehash**. Under activated database support the `tm` package tries to keep as few as possible resources in memory under usage of the database.
`...`	Optional arguments for the `reader`.

Value

An S4 object of class `Corpus` which extends the class `list` containing a collection of text documents.

Author(s)

Ingo Feinerer

Examples

```
txt <- system.file("texts", "txt", package = "tm")
## Not run:
(Corpus(DirSource(txt), readerControl = list(reader
= readPlain, language = "en_US", load = TRUE), dbControl = list(useDb =
TRUE, dbName = "oviddb", dbType = "DB1")))
## End(Not run)
```

```
reut21578 <- system.file("texts", "reut21578", package = "tm")
Corpus(DirSource(reut21578), list(reader = readReut21578XML, load = FALSE))
```

`TextDocument-class`
Text Document

Description

A class representing a text document with additional information.

Objects from the Class

Objects can be created by calls of the form `new("TextDocument", ...)`.

Slots

`Author`: Object of class `character` containing the author names.

`DateTimeStamp`: Object of class `character` containing the date and time when the document was written.

`Description`: Object of class `character` containing additional text information.

`ID`: Object of class `integer` containing an identifier.

`Origin`: Object of class `character` containing information on the source and origin of the text.

`Heading`: Object of class `character` containing the title or a short heading.

`Language`: Object of class `character` containing the language of the text.

`LocalMetaData`: Object of class `list` containing the local meta data in form of tag-value pairs.

Methods

Author `signature(object = "TextDocument")`: Returns the author names.

Author`<-` `signature(object = "TextDocument")`: Sets the author names.

DateTimeStamp `signature(object = "TextDocument")`: Returns the date and time when the document was written.

DateTimeStamp`<-` `signature(object = "TextDocument")`: Sets the date and time when the document was written.

Description `signature(object = "TextDocument")`: Returns additional text information.

Description`<-` `signature(object = "TextDocument")`: Sets additional text information.

ID `signature(object = "TextDocument")`: Returns the identifier.

ID<- `signature(object = "TextDocument")`: Sets the identifier.

Origin `signature(object = "TextDocument")`: Returns information on the source and origin of the text.

Origin<- `signature(object = "TextDocument")`: Sets information on the source and origin of the text.

Heading `signature(object = "TextDocument")`: Returns the title or a short heading.

Heading<- `signature(object = "TextDocument")`: Sets the title or a short heading.

Language `signature(object = "TextDocument")`: Returns the text language.

Language<- `signature(object = "TextDocument")`: Sets the text language.

LocalMetaData `signature(object = "TextDocument")`: Returns the local meta data in form of a tag-value paired list.

Author(s)

Ingo Feinerer

See Also

`DublinCore`

`TextRepository-class`

Text Repository

Description

A class representing a repository of text document collections.

Objects from the Class

Objects can be created by calls of the form `new("TextRepository",...)` or by calling the function `TextRepository`.

Slots

`RepoMetaData`**:** Object of class `list` containing meta data for the text document collections in form of tag-value pairs.

Extends

Class `list`, directly.

Methods

RepoMetaData `signature(object = "TextRepository")`: returns meta data in form of a tag-value paired list.

Author(s)

Ingo Feinerer

See Also

`TextRepository`

`TextRepository` *Text Repository*

Description

Constructs a text repository.

Usage

```
## S4 method for signature 'Corpus':
TextRepository(object, meta = list(created = Sys.time()))
```

Arguments

`object`	A directory containing the documents.
`meta`	An initial list of tag-value pairs for the meta data.

Value

An S4 object of class `TextRepository` which extends the class `list` containing text document collections.

Author(s)

Ingo Feinerer

Examples

```
data("crude")
repo <- TextRepository(crude)
summary(repo)
RepoMetaData(repo)
```

`tmFilter-methods` *Methods for Function tmFilter in Package 'tm'*

Description

Methods for function `tmFilter` in package **tm**.

Methods

object = "Corpus", FUN = searchFullText, doclevel = TRUE If `doclevel` is set, the method applies `FUN` onto each element of `object` and returns the filtered `object`; otherwise `FUN` is applied to `object` itself. If `FUN` has an attribute `doclevel` its value will be automatically used. Note that meta data (`DMetaData`) is automatically passed to `FUN`. The function `FUN` must return a logical value.

See Also

`sFilter` `tmIndex` `getFilters`

Examples

```
data("crude")
attr(searchFullText, "doclevel")
tmFilter(crude, FUN = searchFullText, "company")
```

`tmIndex-methods` *Methods for Function tmIndex in Package 'tm'*

Description

Methods for function `tmIndex` in package **tm**.

Methods

object = "Corpus", FUN = searchFullText, doclevel = TRUE If `doclevel` is set, the method applies `FUN` onto each element of `object` and returns the indices; otherwise `FUN` is applied to `object` itself. If `FUN` has an attribute `doclevel` its value will be automatically used. Note that meta data (`DMetaData`) is automatically passed to `FUN`. The function `FUN` must return a Boolean value.

See Also

`sFilter` `tmFilter` `getFilters`

Examples

```
data("crude")
attr(searchFullText, "doclevel")
tmIndex(crude, FUN = searchFullText, "company")
```

`tmIntersect-methods`

Methods for Function tmIntersect in Package 'tm'

Description

Methods for function `tmIntersect` in package **tm**.

Methods

object = "PlainTextDocument", words = "character" Return `TRUE` if `words` appear in `object`'s text.

See Also

Use `getFilters` to list available filter functions.

Examples

```
data("crude")
tmIntersect(crude[[1]], c("crude", "oil"))
tmIntersect(crude[[1]], "acquisition")
```

`tmMap-methods` *Methods for Function tmMap in Package 'tm'*

Description

Methods for function `tmMap` in package **tm**.

Methods

object = "Corpus", FUN, ..., lazy = FALSE Applies `FUN` onto each element of `object`. Note that document specific metadata (i.e., `DMetaData(object)` is automatically passed to `FUN` as argument `DMetaData`. `FUN` must be a function which returns a `TextDocument`. If `lazy` is set, so-called lazy mapping is activated, i.e., mappings are delayed until the documents' content is accessed. Lazy mapping is useful when working with large corpora but only a few documents are accessed, as it avoids applying the mapping to all elements in the document collection.

Note

Please be aware that lazy transformations are an experimental feature and change R's standard evaluation semantics.

See Also

See `getTransformations` for available transformations shipped with **tm**. See `materialize` for manually materializing lazy transformations.

Examples

```
data("crude")
tmMap(crude, stemDoc)

headings <- function(object, ...) {
new("PlainTextDocument", Heading(object), Cached = TRUE, DateTimeStamp =
Sys.time(), ID = ID(object), Language = Language(object))
}
inspect(tmMap(crude, headings))
```

tmTolower-methods *Methods for Function tmTolower in Package 'tm'*

Description

Methods for function `tmTolower` in package **tm**.

Methods

object = "PlainTextDocument", ... Converts all words from `object` to lower case and return the object.

Examples

```
data("crude")
crude[[1]]
tmTolower(crude[[1]])
```

tmUpdate-methods *Methods for Function tmUpdate in Package 'tm'*

Description

Methods for function `tmUpdate` in package **tm**.

Methods

object = "Corpus", origin = "DirSource", readerControl Checks whether new documents are available in `origin`. If yes, `object` is updated, i.e., it is augmented with the newly available items. `readerControl` specifies arguments to read in new elements provided by `origin`.

VectorSource-class

Source for Vectors

Description

A class representing a vector where each component is interpreted as a document.

Objects from the Class

Objects can be created by calls of the form `new("VectorSource", ...)`.

Slots

`DefaultReader:` Object of class `function` describing a default reader.

`Encoding:` Object of class `character` holding the encoding of the texts delivered by the source.

`Content:` Object of class `vector` holding the actual texts.

`LoDSupport:` Object of class `logical` indicating Load on Demand Support.

`Position:` Object of class `numeric` giving the position in the source.

Extends

Class `Source`, directly.

Author(s)

Ingo Feinerer

See Also

`VectorSource`

VectorSource *Gmane Source*

Description

Constructs a source for a vector.

Usage

```
## S4 method for signature 'vector':
VectorSource(object, encoding = "UTF-8")
```

Arguments

object	A vector holding the texts.
encoding	A character giving the encoding of the file.

Value

An S4 object of class `VectorSource` which extends the class `Source` representing a vector interpreting each element as a document.

Author(s)

Ingo Feinerer

Examples

```
docs <- c("This is a text.", "This another one.")
(vs <- VectorSource(docs))
inspect(Corpus(vs))
```

weightBin *Weight Binary*

Description

Binary weight a term-document matrix.

Usage

```
weightBin(m)
```

Arguments

`m`	A `dgCMatrix` in term frequency format.

Details

Formally this function is a `WeightingFunction` with the additional slot `Name`.

Value

The weighted `dgCMatrix`.

Author(s)

Ingo Feinerer

`WeightFunction-class`

Weighting Function

Description

A class representing a weighting function for a term-document matrix.

Objects from the Class

Objects can be created by calls of the form `new("WeightFunction", ...)`.

Slots

`.Data` Object of class `function` holding the weighting function.

`Name` Object of class `character` containing a name representation for the weighting function.

Extends

Class `function`, directly.

Author(s)

Ingo Feinerer

WeightFunction *Weighting Function Constructor*

Description

Constructs a weighting function for term-document matrices.

Usage

```
WeightFunction(object, name)
```

Arguments

`object`	A function which takes a `dgCMatrix` term-document matrix with term frequencies as input, weights the elements, and returns the weighted matrix.
`name`	A character naming the weighting function

Value

An S4 object of class `WeightFunction` which extends the class `function` representing a weighting function.

Author(s)

Ingo Feinerer

Examples

```
weightCutBin <- WeightFunction(function(m, cutoff) as(m > cutoff, "dgCMatrix"),
                               "binary with cutoff")
```

weightLogical *Weight Logical*

Description

Weight a term-document matrix such that positive term frequencies map to Boolean `TRUE` values.

Usage

```
weightLogical(m)
```

Arguments

`m`	A `dgCMatrix` in term frequency format.

Details

Formally this function is a `WeightingFunction` with the additional slot `Name`.

Value

The weighted `dgCMatrix`.

Author(s)

Ingo Feinerer

`weightTfIdf` *Weight By Term Frequency Inverse Document Frequency*

Description

Weight a term-document matrix by term frequency - inverse document frequency.

Usage

```
weightTfIdf(m)
```

Arguments

`m`	A `dgCMatrix` in term frequency format.

Details

Formally this function is a `WeightingFunction` with the additional slot `Name`.

Value

The weighted `dgCMatrix`.

Author(s)

Ingo Feinerer

`weightTf` *Weight By Term Frequency*

Description

Weight a term-document matrix by term frequency.

Usage

```
weightTf(m)
```

Arguments

`m`	A `dgCMatrix` in term frequency format.

Details

Formally this function is a `WeightingFunction` with the additional slot `Name`.

This function acts as the identity function since the input matrix is already in term frequency format.

Value

The weighted `dgCMatrix`.

Author(s)

Ingo Feinerer

`writeCorpus-methods`
Methods for Function writeCorpus in Package 'tm'

Description

Methods for function `writeCorpus` in Package **tm**

Methods

object = "Corpus" Writes the corpus `object` containing text documents to disk into `path` using `filenames`. In case no manual filenames are provided, filenames are automatically generated by using `object`'s documents' ID strings.

Examples

```
data("crude")
## Not run: writeCorpus(crude, path = ".",
  filenames = paste(seq_along(crude), ".txt", sep = ""))
```

`XMLTextDocument-class`
Text document

Description

A class representing an XML text document with additional information. The XML document itself is represented by an `XMLDocument` from the **XML** package.

Objects from the Class

Objects can be created by calls of the form `new("XMLTextDocument", ...)`.

Slots

`URI`: Object of class `character` containing the path and filename holding the data physically on disk.

`Cached`: Object of class `logical` containing the status whether the file was already loaded into memory.

Extends

Class `list` and `TextDocument`, directly.

Methods

Content `signature(object = "XMLTextDocument")`: Returns the text corpus, i.e., the actual XMLDocument in the data slot.

Content<- `signature(object = "XMLTextDocument")`: Sets the text corpus, i.e., the actual XMLDocument in data slot.

URI `signature(object = "XMLTextDocument")`: Returns the filename on disk.

Cached `signature(object = "XMLTextDocument")`: Returns status information for loading on demand.

Cached<- `signature(object = "XMLTextDocument")`: Sets status information for loading on demand.

Author(s)

Ingo Feinerer

Bibliography

Achatz, M., Kamper, K., and Ruppe, H. (1987). *Die Rechtssprechung des VwGH in Abgabensachen*. Orac Verlag.

Adeva, J. J. G. and Calvo, R. (2006). Mining text with pimiento. *IEEE Internet Computing*, 10(4):27–35.

Ananiadou, S. and Mcnaught, J. (2005). *Text Mining for Biology And Biomedicine*. Artech House, Inc., Norwood, MA, USA.

Anderberg, M. (1973). *Cluster Analysis for Applications*. Academic Press, New York.

Asuncion, A. and Newman, D. (2007). UCI machine learning repository.

Bates, D. and Maechler, M. (2007). **Matrix***: A Matrix Package for R*. R package version 0.999375-2.

Berendt, B., Hotho, A., and Stumme, G. (2002). Towards semantic web mining. In *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin / Heidelberg.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, pages 34–43.

Berry, M., editor (2003). *Survey of Text Mining: Clustering, Classification, and Retrieval*. Springer.

Bierner, G., Baldridge, J., and Morton, T. (2007). **OpenNLP**: A collection of natural language processing tools.

Bill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565.

Binongo, J. N. G. (2003). Who wrote the 15th book of Oz? An application of multivariate analysis to authorship attribution. *Chance*, 16(2):9–17.

Boley, D. (1998). Hierarchical taxonomies using divise partitioning. Technical Report 98-012, University of Minnesota.

Boley, D., Gini, M., Gross, R., Han, E.-H., Karypis, G., Kumar, V., Mobasher, B., Moore, J., and Hastings, K. (1999). Partitioning-based clustering for web document categorization. *Decision Support Systems*, 27(3):329–341.

Burges, C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167.

Cavnar, W. and Trenkle, J. (1994). N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*.

Chambers, J. (1998). *Programming with Data*. Springer-Verlag, New York.

Cohen, A. and Hersh, W. (2005). A survey of current work in biomedical text mining. *Briefings in Bioinformatics*, 6(1):57–71.

Conrad, J., Al-Kofahi, K., Zhao, Y., and Karypis, G. (2005). Effective document clustering for large heterogeneous law firm collections. In *10th International Conference on Artificial Intelligence and Law (ICAIL)*, pages 177–187.

Cooley, R., Mobasher, B., and Srivastava, J. (1997). Web mining: Information and pattern discovery on the world wide web. In *9th International Conference on Tools with Artificial Intelligence*, pages 558–567.

Cristianini, N. and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines (and Other Kernel-based Learning Methods)*. Cambridge University Press.

Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). Gate: A framework and graphical development environment for robust nlp tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, Philadelphia.

Davi, A., Haughton, D., Nasr, N., Shah, G., Skaletsky, M., and Spack, R. (2005). A review of two text-mining packages: SAS **TextMining** and **WordStat**. *The American Statistician*, 59(1):89–103.

de Bruijn, B. and Martin, J. (2002). Getting to the (c)ore of knowledge: mining biomedical literature. *International Journal of Medical Informatics*, 67:7–18.

de Vel, O., Anderson, A., Corney, M., and Mohay, G. (2001). Mining e-mail content for author identification forensics. *SIGMOD Record*, 30(4):55–64.

Deerwester, S., Dumais, S., Furnas, G., Landauer, T., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

Dhillon, I., Guan, Y., and Kulis, B. (2005). A unified view of kernel k-means, spectral clustering and graph partitioning. Technical report, University of Texas at Austin.

Donaldson, I., Martin, J., de Bruijn, B., Wolting, C., Lay, V., Tuekam, B., Zhang, S., Baskin, B., Bader, G. D., Michalickova, K., Pawson, T., and Hogue, C. W. (2003). PreBIND and Textomy – mining the biomedical literature for protein-protein interactions using a support vector machine. *BMC Bioinformatics*, 4(11).

Dong, Q.-W., Wang, X.-L., and Lin, L. (2006). Application of latent semantic analysis to protein remote homology detection. *Bioinformatics*, 22(3):285–290.

Fan, W., Wallace, L., Rich, S., and Zhang, Z. (2006). Tapping the power of text mining. *Communications of the ACM*, 49(9):76–82.

Feinerer, I. (2008a). **openNLP***:* **OpenNLP** *Interface*. R package version 0.1.

Feinerer, I. (2008b). **tm***: Text Mining Package*. R package version 0.3-1.

Feinerer, I. (2008c). **wordnet***: WordNet Interface*. R package version 0.1.

Feinerer, I. and Hornik, K. (2008). Text mining of supreme administrative court jurisdictions. In Preisach, C., Burkhardt, H., Schmidt-Thieme, L., and Decker, R., editors, *Data Analysis, Machine Learning, and Applications (Proceedings of the 31st Annual Conference of the Gesellschaft für Klassifikation e.V., Albert-Ludwigs-Universität Freiburg, March 7–9, 2007)*, Studies in Classification, Data Analysis, and Knowledge Organization. Springer-Verlag.

Feinerer, I., Hornik, K., and Meyer, D. (2008). Text mining infrastructure in R. *Journal of Statistical Software*, 25(5):1–54.

Feinerer, I. and Wild, F. (2007). Automated coding of qualitative interviews with latent semantic analysis. In Mayr, H. and Karagiannis, D., editors, *Proceedings of the 6th International Conference on Information Systems Technology and its Applications, May 23–25, 2007, Kharkiv, Ukraine*, volume 107 of *Lecture Notes in Informatics*, pages 66–77, Bonn, Germany. Gesellschaft für Informatik e.V.

Feldman, R. and Sanger, J. (2007). *The Text Mining Handbook*. Cambridge University Press.

Fellbaum, C., editor (1998). *WordNet: An Electronic Lexical Database*. Bradford Books.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley Professional, third edition.

Gelbukh, A., editor (2004). *Computational Linguistics and Intelligent Text Processing*, volume 2945 of *Lecture Notes in Computer Science*. Springer.

Gentleman, R., Carey, V., Huber, W., Irizarry, R., and Dodoit, S., editors (2005). *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*. Springer-Verlag.

Gentleman, R. C., Carey, V. J., Bates, D. M., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., Hornik, K., Hothorn, T., Huber, W., Iacus, S., Irizarry, R., Leisch, F., Li, C., Maechler, M., Rossini, A. J., Sawitzki, G., Smith, C., Smyth, G., Tierney, L., Yang, J. Y., and Zhang, J. (2004). Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5(10):R80.1–16.

Girón, J., Ginebra, J., and Riba, A. (2005). Bayesian analysis of a multinomial sequence and homogeneity of literary style. *The American Statistician*, 59(1):19–30.

Hansen, H. R. and Neumann, G. (2005). *Wirtschaftsinformatik 1 (Grundlagen und Anwendungen)*. Lucius & Lucius, UTB, 9th edition.

Hartigan, J. (1972). Direct clustering of a data matrix. *Journal of the American Statistical Association*, 67(337):123–129.

Hartigan, J. (1975). *Clustering Algorithms*. John Wiley & Sons, Inc., New York.

Hartigan, J. A. and Wong, M. A. (1979). Algorithm AS 136: A K-means clustering algorithm (AS R39: 81V30 p355-356). *Applied Statistics*, 28:100–108.

Hearst, M. (1999). Untangling Text Data Mining. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 3–10, Morristown, NJ, USA. Association for Computational Linguistics.

Herbrich, R. (2002). *Learning Kernel Classifiers Theory and Algorithms*. Adaptive Computation and Machine Learning. The MIT Press.

Hettich, S. and Bay, S. (1999). The UCI KDD archive.

Holmes, D. (1998). The evolution of stylometry in humanities scholarship. *Literary and Linguistic Computing*, 13(3):111–117.

Holmes, D. and Kardos, J. (2003). Who was the author? An introduction to stylometry. *Chance*, 16(2):5–8.

Hornik, K. (2005). A CLUE for CLUster Ensembles. *Journal of Statistical Software*, 14(12).

Hornik, K. (2007a). **clue***: Cluster Ensembles*. R package version 0.3-17.

Hornik, K. (2007b). **Snowball***: Snowball Stemmers*. R package version 0.0-1.

Hornik, K., Lang, M., Nagel, H., and Mamut, M. (2006). *Die Rechtsprechungspraxis des VwGH in Abgabensachen 2000–2004*. Linde Verlag.

Hornik, K., Zeileis, A., Hothorn, T., and Buchta, C. (2007). **RWeka***: An R Interface to Weka*. R package version 0.3-9.

Ingebrigtsen, L. M. (2007). Gmane: A mailing list archive.

Joachims, T. (1998). Text categorization with support vector machines: learning with many relevant features. In Nédellec, C. and Rouveirol, C., editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398 in Lecture Notes in Computer Science, pages 137–142, Chemnitz, DE. Springer Verlag, Heidelberg, DE.

Joachims, T. (2002). *Learning to Classify Text Using Support Vector Machines: Methods, Theory, and Algorithms*. The Kluwer International Series In Engineerig And Computer Science. Kluwer Academic Publishers, Boston.

Johnson, S. (1967). Hierarchical clustering schemes. *Psychometrika*, 2:241–254.

Jurafsky, D. and Martin, J. (2000). *SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall.

Karatzoglou, A. and Feinerer, I. (2007). Text clustering with string kernels in R. In Decker, R. and Lenz, H.-J., editors, *Advances in Data Analysis (Proceedings of the 30th Annual Conference of the Gesellschaft für Klassifikation e.V., Freie Universität Berlin, March 8–10, 2006)*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 91–98. Springer-Verlag.

Karatzoglou, A., Smola, A., Hornik, K., and Zeileis, A. (2004). **kernlab** - an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9).

Karatzoglou, A., Smola, A., Hornik, K., and Zeileis, A. (2006). **kernlab***: Kernel Methods Lab*. R package version 0.9-5.

Knight, K. (1999). Mining online text. *Communications of the ACM*, 42(11):58–61.

Kosala, R. and Blockeel, H. (2000). Web mining research: a survey. *ACM SIGKDD Explorations Newsletter*, 2(1):1–15.

Kotler, P. and Keller, K. (2008). *Marketing Management*. Pearson, 13th edition.

Krallinger, M., Erhardt, R. A.-A., and Valencia, A. (2005). Text-mining approaches in molecular biology and biomedicine. *Drug Discovery Today*, 10(6).

Kuechler, W. L. (2007). Business applications of unstructured text. *Communications of the ACM*, 50(10):86–93.

Landauer, T., Foltz, P., and Laham, D. (1998). An introduction to latent semantic analysis. *Discourse Processes*, 25:259–284.

Lewis, D. (1997). Reuters-21578 text categorization collection distribution 1.0.

Lewis, D. and Jones, K. S. (1996). Natural language processing for information retrieval. *Communications of the ACM*, 39(1):92–101.

Lewis, D., Yang, Y., Rose, T., and Li, F. (2004). Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361–397.

Li, Y. and Shawe-Taylor, J. (2007). Using kcca for japanese-english cross-language information retrieval and classification. *Journal of Intelligent Information Systems*.

Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., and Watkins, C. (2002). Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley. University of California Press.

Manning, C. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA.

Manola, F. and Miller, E. (2004). *RDF Primer*. World Wide Web Consortium.

Matthews, R. and Merriam, T. (1993). Neural computation in stylometry I: An application to the works of Shakespeare and Fletcher. *Literary and Linguistic Computing*, 8(4):203–209.

McCallum, A. K. (1996). **Bow**: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/~mccallum/bow/.

McCarthy, E. J. (1960). *Basic Marketing: A Managerial Approach*. Homewood, IL: Richard D. Irwin.

Meyer, D. and Buchta, C. (2007). **proxy***: Distance and Similarity Measures*. R package version 0.2.

Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., and Euler, T. (2006). Yale: Rapid prototyping for complex data mining tasks. In *KDD '06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–940, New York, NY, USA. ACM Press.

Mignet, L., Barbosa, D., and Veltri, P. (2003). The XML web: a first study. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 500–510, New York, NY, USA. ACM.

Miller, T. W. (2005). *Data and Text Mining*. Pearson Education International.

Mitchell, M., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330.

Mobasher, B., Cooley, R., and Srivastava, J. (2000). Automatic personalization based on web usage mining. *Communications of the ACM*, 43(8):142–151.

Mueller, J.-P. (2006). **ttda**: Tools for textual data analysis. R package version 0.1.1.

Naso, P. O. (2007). Gutenberg project.

Newman, D., Chemudugunta, C., Smyth, P., and Steyvers, M. (2006). Analyzing entities and topics in news articles using statistical topic models. In Mehrotra, S., Zeng, D. D., Chen, H., Thuraisingham, B. M., and Wang, F.-Y., editors, *Intelligence and Security Informatics, IEEE International Conference on Intelligence and Security Informatics, ISI 2006, San Diego, CA, USA, May 23–24, 2006, Proceedings*, volume 3975 of *Lecture Notes in Computer Science*, pages 93–104. Springer.

Ng, A., Jordan, M., and Weiss, Y. (2002). On spectral clustering: Analysis and an algorithm. In Dietterich, T., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems*, volume 14.

Patófi, J. (1969). On the problems of co-textual analysis of texts. In *Proceedings of the 1969 Conference on Computational Linguistics*, pages 1–45, Morristown, NJ, USA. Association for Computational Linguistics.

Peng, R. D. (2006). Interacting with data using the filehash package. *R News*, 6(4):19–24.

Piatetsky-Shapiro, G. (2005). Poll on text mining tools used in 2004. Checked on 2006-09-17.

Porter, M. (1980). An algorithm for suffix stripping. *Program*, 3:130–137.

Porter, M. (1997). An algorithm for suffix stripping. *Readings in Information Retrieval*, pages 313–316. Reprint.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Radlinski, F. and Joachims, T. (2007). Active exploration for learning rankings from clickthrough data. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 570–579, New York, NY, USA. ACM.

Sakurai, S. and Suyama, A. (2005). An e-mail analysis method based on text mining techniques. *Applied Soft Computing*, 6(1):62–71.

Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523.

Schölkopf, B. and Smola, A. (2002). *Learning with Kernels*. MIT Press.

Schwartz, A. and Hearst, M. (2003). A simple algorithm for identifying abbreviation definitions in biomedical text. In *Proceedings of the 8th Pacific Symposium on Biocomputing*, pages 451–462.

Schweighofer, E. (1999). *Legal Knowledge Representation, Automatic Text Analysis in Public International and European Law*, volume 7. Kluwer Law International, Law and Electronic Commerce.

Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47.

Shatkay, H. and Feldman, R. (2003). Mining the biomedical literature in the genomic era: An overview. *Journal of Computational Biology*, 10(6):821–855.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Sibun, P. and Reynar, J. (1996). Language identification: Examining the issues. In *5th Symposium on Document Analysis and Information Retrieval*, pages 125–135, Las Vegas.

Sirmakessis, S. (2004). *Text Mining and Its Applications: Results of the NEMIS Launch Conference (Studies in Fuzziness and Soft Computing, V. 138)*. Springer.

Sonnenburg, S., Raetsch, G., Schaefer, C., and Schoelkopf, B. (2006). Large scale multiple kernel learning. *Journal of Machine Learning Research*, 7:1531–1565.

Srinivasan, P. (2003). Text mining: Generating hypotheses from MEDLINE. *Journal of the American Society for Information Science and Technology*, 55(5):396–413.

Steinbach, M., Karypis, G., and Kumar, V. (2000). A comparison of document clustering techniques. In *KDD Workshop on Text Mining*.

Strehl, A., Ghosh, J., and Mooney, R. J. (2000). Impact of similarity measures on web-page clustering. In *Proc. AAAI Workshop on AI for Web Search (AAAI 2000), Austin*, pages 58–64. AAAI/MIT Press.

Swanson, D. R. and Smalheiser, N. R. (1994). Assessing a gap in the biomedical literature: Magnesium deficiency and neurologic disease. *Neuroscience Research Communications*, 15:1–9.

Swanson, D. R. and Smalheiser, N. R. (1997). An interactive system for finding complementary literatures: a stimulus to scientific discovery. *Artificial Intelligence*, 91(2):183–203.

Tanabe, L. and Wilbur, W. J. (2002). Tagging gene and protein names in biomedical text. *Bioinformatics*, 18(8):1124–1132.

Temple Lang, D. (2004). **Rstem***: Interface to Snowball Implementation of Porter's Word Stemming Algorithm*. R package version 0.2-0.

Temple Lang, D. (2006). **XML***: Tools for Parsing and Generating XML within R and S-Plus*. R package version 0.99-8.

Teo, C. and Vishwanathan, S. (2006). Fast and space efficient string kernels using suffix arrays. In *Proceedings of the 23rd International Conference on Machine Learning*.

Tjhi, W.-C. and Chen, L. (2007). Possibilistic fuzzy co-clustering of large document collections. *Pattern Recognition*, 40(12):3452–3466.

Torkkola, K. (2004). Discriminative features for text document classification. *Pattern Analysis & Applications*, 6(4):301–308.

Tweedie, F. J., Singh, S., and Holmes, D. I. (1996). Neural network applications in stylometry: The federalist papers. *Computers and the Humanities*, 30(1).

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer-Verlag, New York, fourth edition. ISBN 0-387-95457-0.

Vishwanathan, S. and Smola, A. (2004). Fast kernels for string and tree matching. In Tsuda, K., Schölkopf, B., and Vert, J., editors, *Kernels and Bioinformatics*. MIT Press, Cambridge, MA.

Walker, D. (1969). Computational linguistic techniques in an on-line system for textual analysis. In *Proceedings of the 1969 Conference on Computational Linguistics*, pages 1–3, Morristown, NJ, USA. Association for Computational Linguistics.

Wallis, S. and Nelson, G. (2001). Knowledge discovery in grammatically analysed corpora. *Data Mining and Knowledge Discovery*, 5(4):305–335.

Watkins, C. (2000). Dynamic alignment kernels. In Smola, A., Bartlett, P., Schölkopf, B., and Schuurmans, D., editors, *Advances in Large Margin Classifiers*, pages 39–50, Cambridge, MA. MIT Press.

Weiss, S., Indurkhya, N., Zhang, T., and Damerau, F. (2004). *Text Mining: Predictive Methods for Analyzing Unstructured Information*. Springer.

Wild, F. (2005). **lsa***: Latent Semantic Analysis*. R package version 0.57.

Willett, P. (1988). Recent trends in hierarchic document clustering: a critical review. *Information Processing and Management*, 24(5):577–597.

Witten, I. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, second edition.

Witten, I. H., Paynter, G. W., Frank, E., Gutwin, C., and Nevill-Manning, C. G. (1999). KEA: Practical automatic keyphrase extraction. In *DL '99: Proceedings of the fourth ACM conference on Digital libraries*, pages 254–255, New York, NY, USA. ACM.

Witten, I. H., Paynter, G. W., Frank, E., Gutwin, C., and Nevill-Manning, C. G. (2005). KEA: Practical automatic keyphrase extraction. In Theng, Y.-L. and Foo, S., editors, *Design and Usability of Digital Libraries: Case Studies in the Asia Pacific*, pages 129–152. Information Science Publishing, London.

Wu, Y.-F. and Chen, X. (2005). elearning assessment through textual analysis of class discussions. In *Fifth IEEE International Conference on Advanced Learning Technologies*, pages 388–390.

Zhao, Y. and Karypis, G. (2004). Empirical and theoretical comparisons of selected criterion functions for document clustering. *Machine Learning*, 55(3):311–331.

Zhao, Y. and Karypis, G. (2005a). Hierarchical clustering algorithms for document datasets. *Data Mining and Knowledge Discovery*, 10(2):141–168.

Zhao, Y. and Karypis, G. (2005b). Topic-driven clustering for document datasets. In *SIAM International Data Mining Conference*.