Journal of Statistical Software

March 2008, Volume 25, Issue 5. *http://www.jstatsoft.org/*

Text Mining Infrastructure in R

Ingo Feinerer
Wirtschaftsuniversität Wien

Kurt Hornik
Wirtschaftsuniversität Wien

David Meyer
Wirtschaftsuniversität Wien

Abstract

During the last decade text mining has become a widely used discipline utilizing statistical and machine learning methods. We present the **tm** package which provides a framework for text mining applications within R. We give a survey on text mining facilities in R and explain how typical application tasks can be carried out using our framework. We present techniques for count-based analysis methods, text clustering, text classification and string kernels.

Keywords: text mining, R, count-based evaluation, text clustering, text classification, string kernels.

1. Introduction

Text mining encompasses a vast field of theoretical approaches and methods with one thing in common: text as input information. This allows various definitions, ranging from an extension of classical data mining to texts to more sophisticated formulations like "the use of large online text collections to discover new facts and trends about the world itself" (Hearst 1999). In general, text mining is an interdisciplinary field of activity amongst data mining, linguistics, computational statistics, and computer science. Standard techniques are text classification, text clustering, ontology and taxonomy creation, document summarization and latent corpus analysis. In addition a lot of techniques from related fields like information retrieval are commonly used.

Classical applications in text mining (Weiss *et al.* 2004) come from the data mining community, like document clustering (Zhao and Karypis 2005b,a; Boley 1998; Boley *et al.* 1999) and document classification (Sebastiani 2002). For both the idea is to transform the text into a structured format based on term frequencies and subsequently apply standard data mining techniques. Typical applications in document clustering include grouping news articles or information service documents (Steinbach *et al.* 2000), whereas text categorization methods are

used in, e.g., e-mail filters and automatic labeling of documents in business libraries (Miller 2005). Especially in the context of clustering, specific distance measures (Zhao and Karypis 2004; Strehl *et al.* 2000), like the Cosine, play an important role. 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.

During the last years, more innovative text mining methods have been used for analyses in various fields, e.g., in linguistic stylometry (Girón *et al.* 2005; Nilo and Binongo 2003; Holmes and Kardos 2003), where the probability that a specific author wrote a specific text is calculated by analyzing the author's writing style, or in search engines for learning rankings of documents from search engine logs of user behavior (Radlinski and Joachims 2007).

Latest developments in document exchange have brought up valuable concepts for automatic handling of texts. The semantic web (Berners-Lee *et al.* 2001) 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) where efforts to handle this format have already been made in R (R Development Core Team 2007) with the Bioconductor project (Gentleman *et al.* 2004, 2005). This development offers great flexibility in document exchange. But with the growing popularity of XML based formats (e.g., RDF/XML as a common representation for RDF) tools need to be able to handle XML documents and metadata.

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 contraints 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 latent semantic analysis techniques in bioinformatics (Dong *et al.* 2006), the usage of statistical methods for automatically investigating jurisdictions (Feinerer and Hornik 2007), 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 capabilites 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,

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

Table 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.

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

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

Table 1 gives an overview over the most-used commercial text mining products (Piatetsky-Shapiro 2005), selected open source text mining tool kits, and features. Commercial products include **Clearforest**, a text-driven business intelligence solution, Copernic **Summarizer**, a summarizing software extracting key concepts and relevant sentences, **dtSearch**, a document search tool, Insightful **Infact**, a search and analysis text mining tool, **Inxight**, an integrated suite of tools for search, extraction, and analysis of text, SPSS **Clementine**, a data and text mining workbench, SAS **Text Miner**, a suite of tools for knowledge discovery and knowledge extraction in texts, **TEMIS**, a tool set for text extraction, text clustering, and text categorization, and **WordStat**, a product for computer assisted text analysis.

From Table 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, 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.* 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. It is fully written in Java. Another 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 extension package **ttda** (Mueller 2006) provides some methods for textual data analysis.

We present a text mining framework for the open source statistical computing environment R centered around the new extension package **tm** (Feinerer 2007b). 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 languange 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.

This paper is organized as follows. Section 2 elaborates, on a conceptual level, important ideas and tasks a text mining framework should be able to deal with. Section 3 presents the main structure of our framework, its algorithms, and ways to extend the text mining framework for custom demands. Section 4 describes preprocessing mechanisms, like data import, stemming, stopword removal and synonym detection. Section 5 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. Section 6 presents an application of **tm** by analyzing the R-devel 2006 mailing list. Section 7 concludes. Finally Appendix A gives a very detailed and technical description of **tm** data structures.

2. 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 frequences 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 mech-

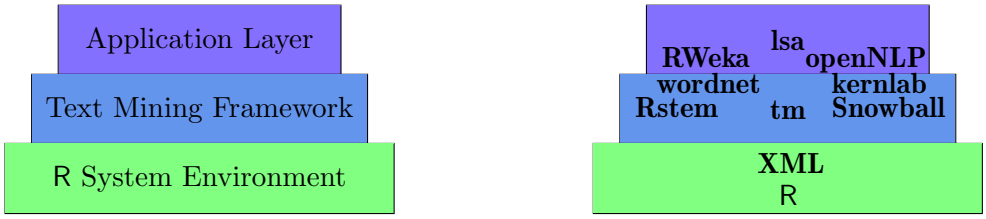

Figure 1: Conceptual layers and packages.

anisms for term-document matrices and provide interfaces to access the document corpora as plain character sequences.

Basically, the framework and infrastructure supplied by **tm** aims at implementing the conceptual framework presented above. The next section will introduce the data structures and algorithms provided.

3. 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 2007a), 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 Section 4), for generating term-document matrices (see Paragraph 3.1.4), or for custom functions when extending **tm**'s methods (see Section 3.3).

Figure 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 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

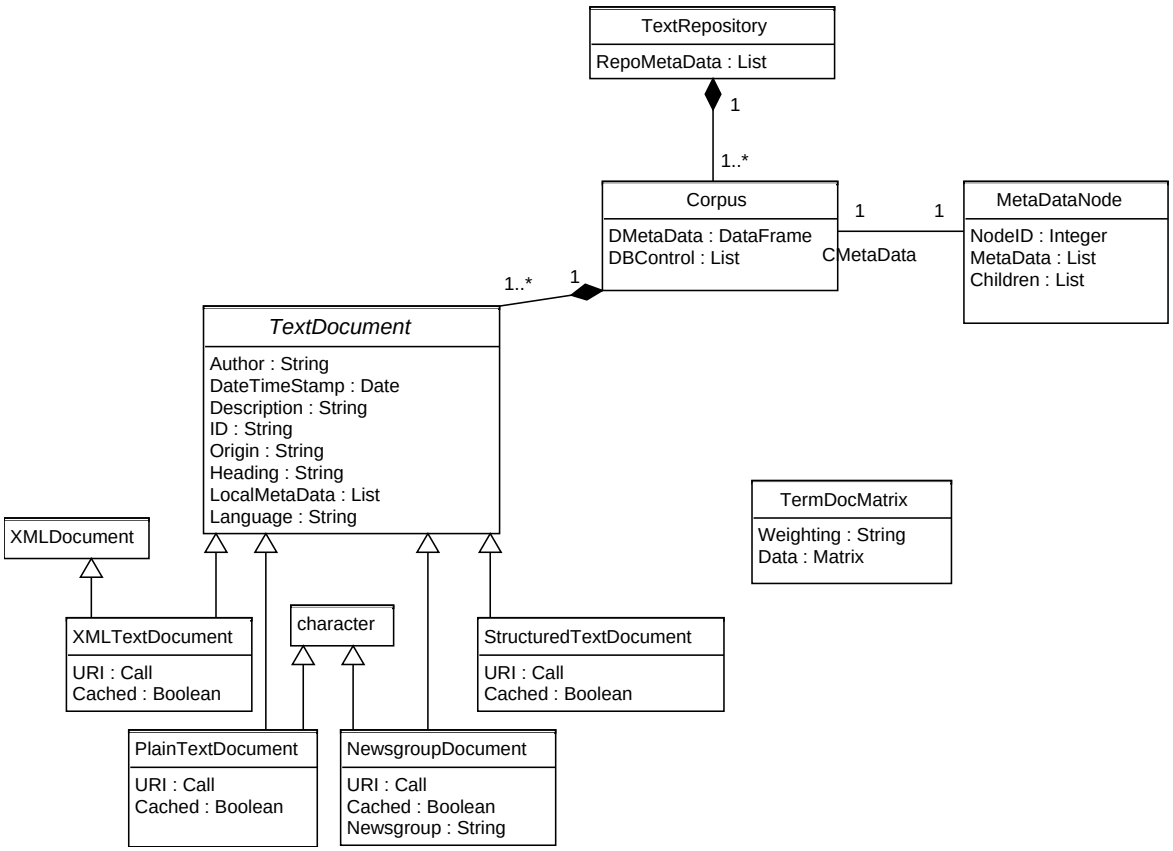

Figure 2: UML class diagram of the **tm** package.

stemming, whereas some packages provide both infrastructure and applications, like **wordnet** (Feinerer 2007c), **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 (LSA Landauer *et al.* 1998; Deerwester *et al.* 1990) 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).

3.1. Data structures

We start by explaining the data structures: The basic framework classes and their interactions are depicted in Figure 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.

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 `TextDocument`s 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

```
R> 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 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 2.

`dbControl` : a list with the three 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).

An example of a constructor call might be

```
R> Corpus(object = ...,
+             readerControl = list(reader = object@DefaultReader,
```

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 posting (e-mails) in UCI KDD archive format
`readPDF()`	Read in PDF documents
`readDOC()`	Read in MS Word documents
`readHTML()`	Read in simply structured HTML documents

Table 2: Available readers in the **tm** package.

```
+                                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.

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

```
R> 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.

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

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

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

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

```
R> 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

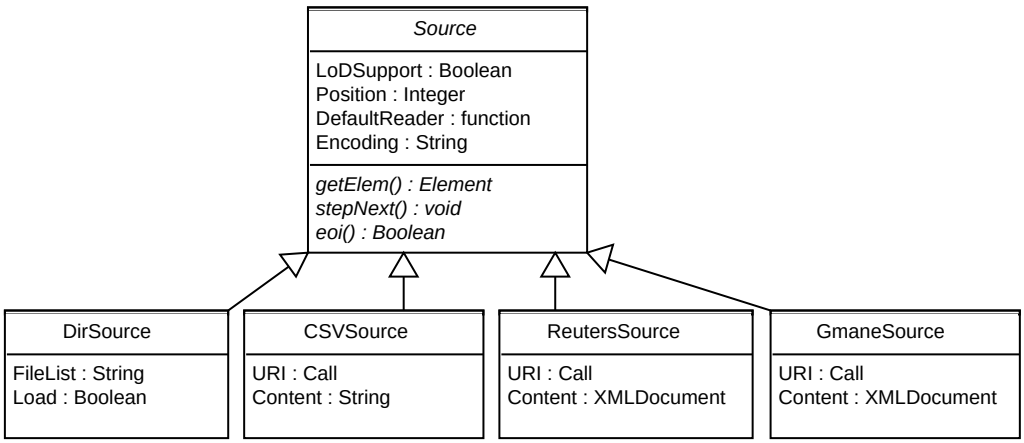

Figure 3: UML class diagram for `Source`s.

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

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

```
R> 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**.

```
R> 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.

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 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 *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: `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

```
R> 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.

3.2. 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:

```
R> txt <- system.file("texts", "txt", package = "tm")
R> (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:

```
R> 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.,

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

[1] "1"
```

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

```
R> 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.,

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

Available meta data pairs are:
  Author       : Publius Ovidius Naso
  Cached       : TRUE
  DateTimeStamp: 2008-03-16 14:49:58
  Description  :
  ID           : 1
  Heading      :
  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.

```
R> 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.

```
R> 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.

```
R> 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. 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.

```
R> length(ovid)

[1] 5
```

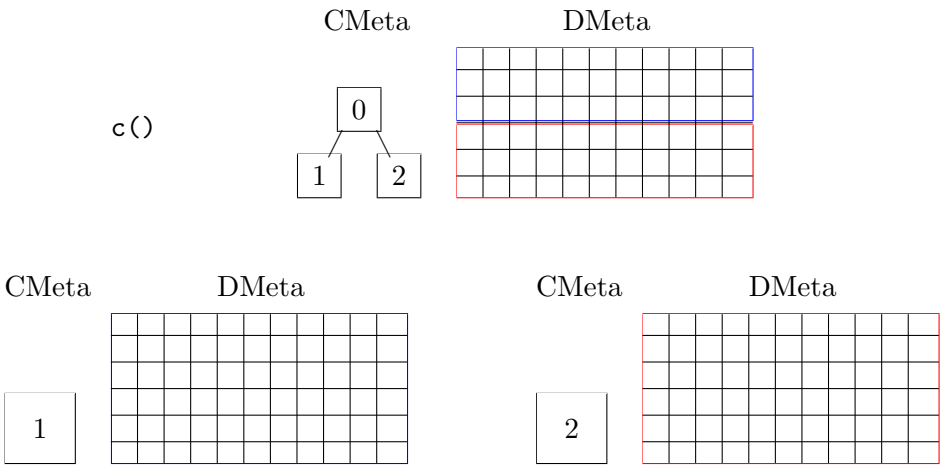

Figure 4: Concatenation of two text document collections with `c()`.

`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.

```
R> 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.

```
R> 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.

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

```
R> CMetaData(ovid)

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

Slot "MetaData":
$create_date
[1] "2008-03-16 14:49:58 CET"

$creator
   LOGNAME
"feinerer"

$test
[1] 1 2 3


Slot "children":
list()

R> 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.

```
R> (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.,

```
R> (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.

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

R> RepoMetaData(repo)

$created
[1] "2008-03-16 14:49:58 CET"

$modified
[1] "Sun Mar 16 14:49:58 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.,

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

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

Slot "MetaData":
$create_date
```

```
[1] "2008-03-16 14:49:58 CET"

$creator
   LOGNAME
"feinerer"

$test
[1] 1 2 3

$foo
[1] "bar"


Slot "children":
list()
```

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

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 5 visualizes this process of transformations and filters. It shows a text document collection with text documents $d_1, d_2, \ldots, d_n$ 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.,

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

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

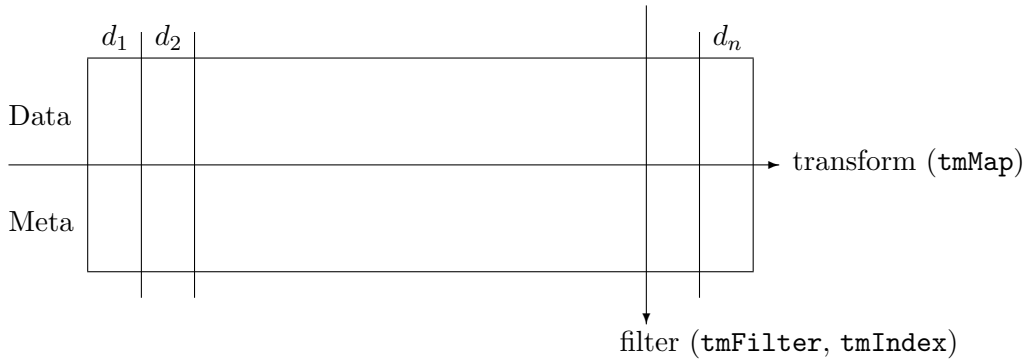

Figure 5: Generic transform and filter operations on a text document collection.

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 3: Transformations shipped with **tm**.

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 3 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 `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 (default) or to each document separately, 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:

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

```
A text document collection with 2 text documents
```

Any valid predicate function can be used as custom filter function but for most cases the default 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:

```
R> tmIndex(ovid, "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.

3.3. 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:

```
R> 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`:

```
R> 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:

```
R> 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,

```
R> 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

```
R> 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:

```
R> 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:

```
R> rss <- system.file("texts", "gmane.comp.lang.r.gr.rdf", package = "tm")
R> 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).

```
R> 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

```
R> setGeneric("myTransform", function(object, ...) standardGeneric("myTransform"))
R> 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.

4. 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.

4.1. 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

```
R> reut21578XMLgz <- system.file("texts", "reut21578.xml.gz", package = "tm")
R> (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()`:

```
R> 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

```
R> tmFilter(crude, "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:

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

4.2. 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:

```
R> 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:

```
R> 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.,

```
R> tmMap(acq, stemDoc)

A text document collection with 50 text documents
```

4.3. 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()`)

```
R> 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"

R> 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.

4.4. 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:

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

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

```
R> 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:

```
R> 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

```
R> 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.

4.5. 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:

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

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

```
R> 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

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

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

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

4.6. 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.,

```
R> library("openNLP")
R> 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*.

5. Applications

Here we present some typical applications on texts, that is analysis based on counting frequencies, clustering and classification of texts.

5.1. 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)

```
R> 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

```
R> (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:

```
R> 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:

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

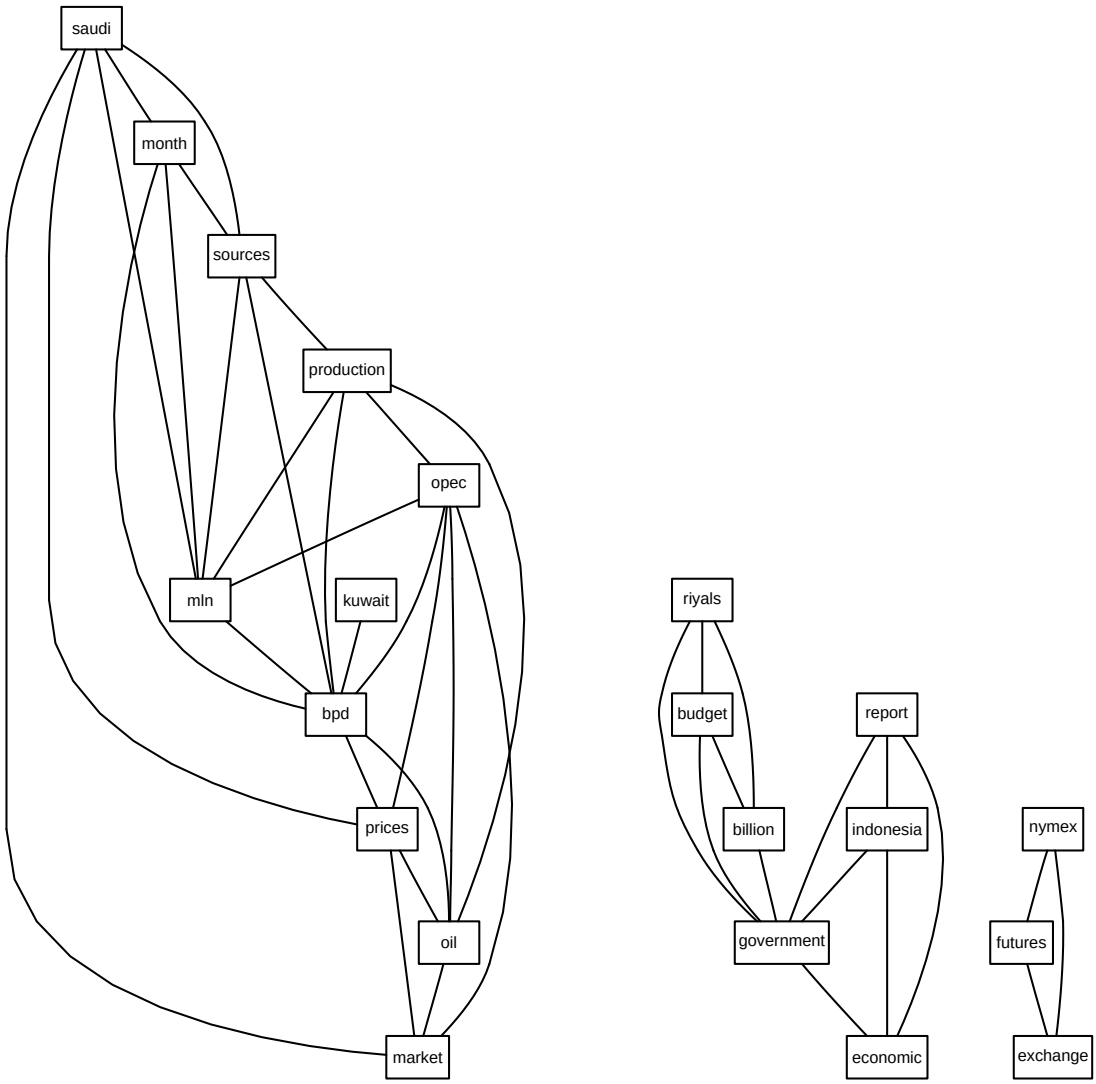

Figure 6: Visualization of the correlations within a term-document matrix.

```
 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 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.

5.2. 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

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

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

```
R> 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

```
R> 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)

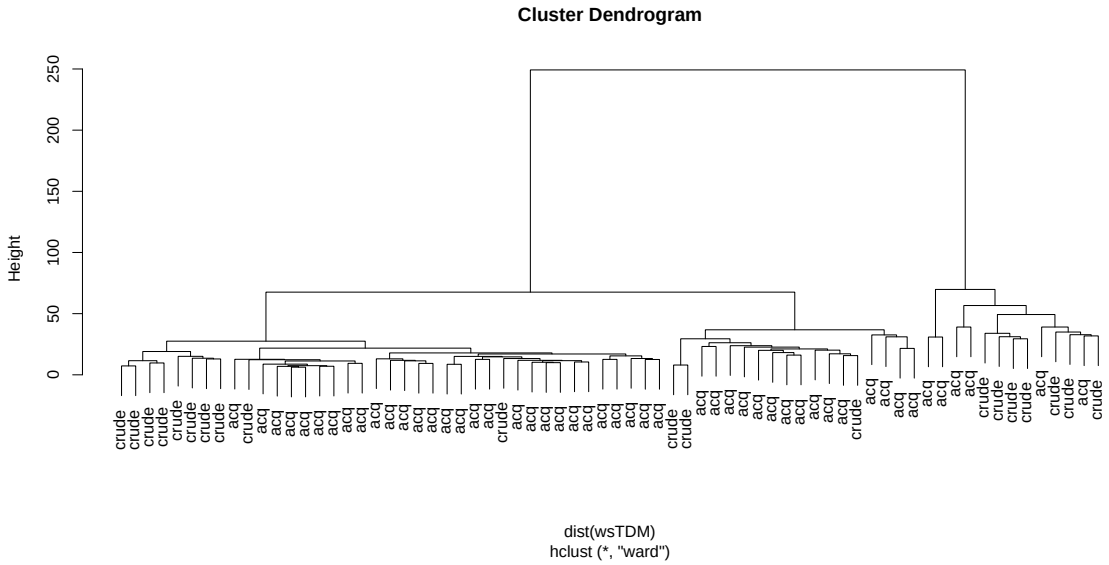

Figure 7: Dendrogram for hierarchical clustering. The labels show the original group names.

```
R> 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:

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

Figure 7 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.

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`)

```
R> 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

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

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:

```
R> 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.

5.3. 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).

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

```
R> 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`):

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

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

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

and store their original classification

```
R> 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`):

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

and obtain the following confusion matrix

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

```
         Reuters
1-NN      nonspam spam
  nonspam     503  138
  spam        193  315
```

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

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

```
[1] 0.7119234
```

Support vector machine classification

Another typical, 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:

```
R> 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`)

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

which yields the following confusion matrix

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

```
         Reuters
SVM       nonspam spam
  nonspam     634   79
  spam         62  374
```

with following cross-agreement:

```
R> 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 realisitic 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.

5.4. 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, 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, $\text{num}_s(x)$ 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

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

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

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

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 in 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

```
R> 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

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

String kernel function.  Type =  string
 Hyperparameters : sub-sequence/string length =  4  lambda =  1.1
 Normalized
```

compared to the original Reuters clusters

```
R> 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.

6. Analysis of the R-devel 2006 mailing list

This section 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).

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/`.

```
R> 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`)

```
R> 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.

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

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

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

After preprocessing we have

```
R> 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.

```
R> 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.

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):

```
R> authors <- lapply(rdevel, Author)
R> 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):

```
R> 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:

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

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

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

```
R> topicCol <- rdevel[headings == bigTopics[1]]
R> 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

```
R> topicCol <- rdevel[headings == bigTopics[2]]
R> 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)"
```

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).

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

A text document collection with 796 text documents
```

The most active authors in that context are

```
R> bugColAuthors <- lapply(bugCol, Author)
R> bugColAuthors <- sapply(bugColAuthors, paste, collapse = " ")
R> 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.

```
R> f <- findFreqTerms(tdm, 30, 31)
R> 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.

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

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

```
R> 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

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

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

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

Now we repeat our attempt to find frequent terms.

```
R> f <- findFreqTerms(tdmInc, 30, 31)
R> 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.

```
R> subjectCounts <- 0
R> 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)
+ }
R> 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.

7. Conclusion

We introduced a new framework for text mining applications in R via the **tm** package. It 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 metadata management is implemented for collections of text documents to alleviate the usage of large and with metadata enriched document sets. With the package ships native support for handling the Reuters-21578 data set, the Reuters Corpus Volume 1 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 (e.g., Reuters to plain text). 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 which are frequently used in the text mining literature. This allows the straight-forward integration of existing methods for classification and clustering.

tm already supports and covers a broad range of text mining methods, by using available technology in R but also by interfacing with other open source tool kits like **Weka** or **openNLP** offering further methods for tokenization, stemming, sentence detection, and part of speech tagging. Nevertheless there are still many areas open for further improvement, e.g., with methods rather common in linguistics, like latent semantic analysis. We are thinking of a better integration of **tm** with the **lsa** package. Another key technique to be dealt in the future will be the efficient handling of very large term-document matrices. In particular, we are working on memory-efficient clustering techniques in R to handle highly dimensional sparse matrices as found in larger text mining case studies. With the ongoing research efforts in analyzing large data sets and by using sparse data structures **tm** will be among the first to take advantage of new technology. Finally, we will keep adding reader functions and source classes for popular data formats.

Acknowledgments

We would like to thank Christian Buchta for his valuable feedback throughout the paper.

References

Adeva JJG, Calvo R (2006). "Mining Text with Pimiento." *IEEE Internet Computing*, **10**(4), 27–35. ISSN 1089-7801. doi:10.1109/MIC.2006.85.

Anderberg M (1973). *Cluster Analysis for Applications*. Academic Press, New York.

Asuncion A, Newman D (2007). "UCI Machine Learning Repository." URL http://www.ics.uci.edu/~mlearn/MLRepository.html.

Bates D, Maechler M (2007). ***Matrix**: A Matrix Package for R*. R package version 0.999375-2, URL http://CRAN.R-project.org/package=Matrix.

Berners-Lee T, Hendler J, Lassila O (2001). "The Semantic Web." *Scientific American*, pp. 34–43.

Berry M (ed.) (2003). *Survey of Text Mining: Clustering, Classification, and Retrieval*. Springer-Verlag. ISBN 0387955631.

Bierner G, Baldridge J, Morton T (2007). "**OpenNLP**: A Collection of Natural Language Processing Tools." URL http://opennlp.sourceforge.net/.

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.

Boley D (1998). "Hierarchical Taxonomies Using Divise Partitioning." *Technical Report 98-012*, University of Minnesota.

Boley D, Gini M, Gross R, Han EH, Karypis G, Kumar V, Mobasher B, Moore J, Hastings K (1999). "Partitioning-based Clustering for Web Document Categorization." *Decision Support Systems*, **27**(3), 329–341. ISSN 0167-9236. doi:10.1016/S0167-9236(99)00055-X.

Cavnar W, Trenkle J (1994). "*N*-Gram-based Text Categorization." In "Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval," pp. 161–175. Las Vegas.

Chambers J (1998). *Programming with Data*. Springer-Verlag, New York.

Cristianini N, Shawe-Taylor J (2000). *An Introduction to Support Vector Machines (and Other Kernel-based Learning Methods)*. Cambridge University Press. ISBN 0 521 78019 5.

Cunningham H, Maynard D, Bontcheva K, 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, Spack R (2005). "A Review of Two Text-Mining Packages: SAS **TextMining** and **WordStat**." *The American Statistician*, **59**(1), 89–103.

Deerwester S, Dumais S, Furnas G, Landauer T, Harshman R (1990). "Indexing by Latent Semantic Analysis." *Journal of the American Society for Information Science*, **41**(6), 391–407.

Dhillon I, Guan Y, Kulis B (2005). "A Unified View of Kernel *k*-Means, Spectral Clustering and Graph Partitioning." *Technical report*, University of Texas at Austin.

Dong QW, Wang XL, Lin L (2006). "Application of Latent Semantic Analysis to Protein Remote Homology Detection." *Bioinformatics*, **22**(3), 285–290. ISSN 1367-4803. doi:10.1093/bioinformatics/bti801.

Feinerer I (2007a). ***openNLP**: **OpenNLP** Interface*. R package version 0.1, URL http://CRAN.R-project.org/package=openNLP.

Feinerer I (2007b). ***tm**: Text Mining Package*. R package version 0.3, URL http://CRAN.R-project.org/package=tm.

Feinerer I (2007c). ***wordnet**: WordNet Interface*. R package version 0.1, URL http://CRAN.R-project.org/package=wordnet.

Feinerer I, Hornik K (2007). "Text Mining of Supreme Administrative Court Jurisdictions." In C Preisach, H Burkhardt, L Schmidt-Thieme, R Decker (eds.), "Data Analysis, Machine Learning, and Applications (Proceedings of the 31st Annual Conference of the Gesellschaft f ür Klassifikation e.V., March 7–9, 2007, Freiburg, Germany)," Studies in Classification, Data Analysis, and Knowledge Organization. Springer-Verlag.

Feinerer I, Wild F (2007). "Automated Coding of Qualitative Interviews with Latent Semantic Analysis." In H Mayr, D Karagiannis (eds.), "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*, pp. 66–77. Gesellschaft für Informatik e.V., Bonn, Germany.

Fellbaum C (ed.) (1998). *WordNet: An Electronic Lexical Database*. Bradford Books. ISBN 0-262-06197-X.

Fowler M (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley Professional, third edition. ISBN 0321193687.

Gentleman R, Carey V, Huber W, Irizarry R, Dodoit S (eds.) (2005). *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*. Springer-Verlag.

Gentleman RC, Carey VJ, Bates DM, 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 AJ, Sawitzki G, Smith C, Smyth G, Tierney L, Yang JY, Zhang J (2004). "Bioconductor: Open Software Development for Computational Biology and Bioinformatics." *Genome Biology*, **5**(10), R80.1–16. URL http://genomebiology.com/2004/5/10/R80.

Girón J, Ginebra J, Riba A (2005). "Bayesian Analysis of a Multinomial Sequence and Homogeneity of Literary Style." *The American Statistician*, **59**(1), 19–30.

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 JA, Wong MA (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," pp. 3–10. Association for Computational Linguistics, Morristown, NJ, USA. ISBN 1-55860-609-3.

Herbrich R (2002). *Learning Kernel Classifiers Theory and Algorithms*. Adaptive Computation and Machine Learning. The MIT Press.

Hettich S, Bay S (1999). "The UCI KDD Archive." URL http://kdd.ics.uci.edu/.

Holmes D, 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). URL http://www.jstatsoft.org/v14/i12/.

Hornik K (2007a). ***clue**: Cluster Ensembles*. R package version 0.3-17, URL http://CRAN.R-project.org/package=clue.

Hornik K (2007b). ***Snowball**: Snowball Stemmers*. R package version 0.0-1.

Hornik K, Zeileis A, Hothorn T, Buchta C (2007). ***RWeka**: An R Interface to Weka*. R package version 0.3-9, URL http://CRAN.R-project.org/package=RWeka.

Ingebrigtsen LM (2007). "Gmane: A Mailing List Archive." URL http://gmane.org/.

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.

Karatzoglou A, Feinerer I (2007). "Text Clustering with String Kernels in R." In R Decker, HJ Lenz (eds.), "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, pp. 91–98. Springer-Verlag.

Karatzoglou A, Smola A, Hornik K, Zeileis A (2004). "**kernlab** - An S4 Package for Kernel Methods in R." *Journal of Statistical Software*, **11**(9). URL http://www.jstatsoft.org/v11/i09/.

Karatzoglou A, Smola A, Hornik K, Zeileis A (2006). ***kernlab**: Kernel Methods Lab*. R package version 0.8-1, URL http://CRAN.R-project.org/package=kernlab.

Landauer T, Foltz P, 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." URL http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html.

Lewis D, Yang Y, Rose T, Li F (2004). "RCV1: A New Benchmark Collection for Text Categorization Research." *Journal of Machine Learning Research*, **5**, 361–397.

Li Y, Shawe-Taylor J (2007). "Using KCCA for Japanese-English Cross-Language Information Retrieval and Classification." *Journal of Intelligent Information Systems*. URL http://eprints.ecs.soton.ac.uk/10786/.

Lodhi H, Saunders C, Shawe-Taylor J, Cristianini N, 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, pp. 281–297. University of California Press, Berkeley.

Manola F, Miller E (2004). *RDF Primer*. World Wide Web Consortium. URL http://www.w3.org/TR/rdf-primer/.

McCallum AK (1996). "**Bow**: A Toolkit for Statistical Language Modeling, Text Retrieval, Classification and Clustering." http://www.cs.cmu.edu/~mccallum/bow/.

Meyer D, Buchta C (2007). ***proxy**: Distance and Similarity Measures*. R package version 0.2, URL http://CRAN.R-project.org/package=proxy.

Mierswa I, Wurst M, Klinkenberg R, Scholz M, 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," pp. 935–940. ACM Press, New York, NY, USA. ISBN 1-59593-339-5. doi:10.1145/1150402.1150531.

Miller TW (2005). *Data and Text Mining*. Pearson Education International.

Mitchell M, Santorini B, Marcinkiewicz MA (1993). "Building a Large Annotated Corpus of English: The Penn Treebank." *Computational Linguistics*, **19**(2), 313–330. URL ftp://ftp.cis.upenn.edu/pub/treebank/doc/cl93.ps.gz.

Mueller JP (2006). "**ttda**: Tools for Textual Data Analysis." R package version 0.1.1, URL http://wwwpeople.unil.ch/jean-pierre.mueller/.

Naso PO (2007). "Gutenberg Project." URL http://www.gutenberg.org/.

Ng A, Jordan M, Weiss Y (2002). "On Spectral Clustering: Analysis and an Algorithm." In T Dietterich, S Becker, Z Ghahramani (eds.), "Advances in Neural Information Processing Systems," volume 14.

Nilo J, Binongo G (2003). "Who Wrote the 15th Book of Oz? An Application of Multivariate Analysis to Authorship Attribution." *Chance*, **16**(2), 9–17.

Peng RD (2006). "Interacting with Data Using the Filehash Package." *R News*, **6**(4), 19–24. URL http://CRAN.R-project.org/doc/Rnews/.

Piatetsky-Shapiro G (2005). "Poll on Text Mining Tools Used in 2004." Checked on 2006-09-17, URL http://www.kdnuggets.com/polls/2005/text_mining_tools.htm.

Porter M (1997). "An Algorithm for Suffix Stripping." *Readings in Information Retrieval*, pp. 313–316. Reprint.

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Radlinski F, 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," pp. 570–579. ACM, New York, NY, USA. doi:10.1145/1281192.1281254.

Sakurai S, Suyama A (2005). "An E-mail Analysis Method Based on Text Mining Techniques." *Applied Soft Computing*, **6**(1), 62–71. doi:10.1016/j.asoc.2004.10.007.

Schölkopf B, Smola A (2002). *Learning with Kernels*. MIT Press.

Sebastiani F (2002). "Machine Learning in Automated Text Categorization." *ACM Computing Surveys*, **34**(1), 1–47. ISSN 0360-0300. doi:10.1145/505282.505283.

Shawe-Taylor J, Cristianini N (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press. ISBN 0 521 81397 2.

Sibun P, Reynar J (1996). "Language Identification: Examining the Issues." In "Proceedings of SDAIR-96, 5th Symposium on Document Analysis and Information Retrieval," pp. 125–135. Las Vegas.

Sonnenburg S, Raetsch G, Schaefer C, Schoelkopf B (2006). "Large Scale Multiple Kernel Learning." *Journal of Machine Learning Research*, **7**, 1531–1565.

Steinbach M, Karypis G, Kumar V (2000). "A Comparison of Document Clustering Techniques." In "KDD Workshop on Text Mining," URL http://www.cs.cmu.edu/~dunja/PapersWshKDD2000.html.

Strehl A, Ghosh J, Mooney RJ (2000). "Impact of Similarity Measures on Web-page Clustering." In "Proc. AAAI Workshop on AI for Web Search (AAAI 2000), Austin," pp. 58–64. AAAI/MIT Press. ISBN 1-57735-116-9. URL http://strehl.com/download/strehl-aaai00.pdf.

Temple Lang D (2004). ***Rstem**: Interface to Snowball Implementation of Porter's Word Stemming Algorithm*. R package version 0.2-0, URL http://www.omegahat.org/Rstem/.

Temple Lang D (2006). ***XML**: Tools for Parsing and Generating XML within R and S-Plus*. R package version 0.99-8, URL http://CRAN.R-project.org/package=XML.

Teo C, Vishwanathan S (2006). "Fast and Space Efficient String Kernels Using Suffix Arrays." In "Proceedings of the 23rd International Conference on Machine Learning," URL http://www.icml2006.org/icml_documents/camera-ready/117_Fast_and_Space_Effic.pdf.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Springer-Verlag, New York, fourth edition. ISBN 0-387-95457-0, URL http://www.stats.ox.ac.uk/pub/MASS4/.

Vishwanathan S, Smola A (2004). "Fast Kernels for String and Tree Matching." In K Tsuda, B Schölkopf, J Vert (eds.), "Kernels and Bioinformatics," MIT Press, Cambridge, MA.

Watkins C (2000). "Dynamic Alignment Kernels." In A Smola, P Bartlett, B Schölkopf, D Schuurmans (eds.), "Advances in Large Margin Classifiers," pp. 39–50. MIT Press, Cambridge, MA.

Weiss S, Indurkhya N, Zhang T, Damerau F (2004). *Text Mining: Predictive Methods for Analyzing Unstructured Information*. Springer-Verlag. ISBN 0387954333.

Wild F (2005). ***lsa**: Latent Semantic Analysis*. R package version 0.57, URL http://CRAN.R-project.org/package=lsa.

Witten I, Frank E (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, second edition.

Witten IH, Paynter GW, Frank E, Gutwin C, Nevill-Manning CG (2005). "Kea: Practical automatic keyphrase extraction." In YL Theng, S Foo (eds.), "Design and Usability of Digital Libraries: Case Studies in the Asia Pacific," pp. 129–152. Information Science Publishing, London.

Wu YF, Chen X (2005). "eLearning Assessment through Textual Analysis of Class Discussions." In "Fifth IEEE International Conference on Advanced Learning Technologies," pp. 388–390. doi:10.1109/ICALT.2005.132.

Zhao Y, Karypis G (2004). "Empirical and Theoretical Comparisons of Selected Criterion Functions for Document Clustering." *Machine Learning*, **55**(3), 311–331. ISSN 0885-6125. doi:10.1023/B:MACH.0000027785.44527.d6.

Zhao Y, Karypis G (2005a). "Hierarchical Clustering Algorithms for Document Datasets." *Data Mining and Knowledge Discovery*, **10**(2), 141–168.

Zhao Y, Karypis G (2005b). "Topic-driven Clustering for Document Datasets." In "Proceedings of the 2005 SIAM International Conference on Data Mining (SDM05)," pp. 358–369.

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.

`TermDocMatrix` 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 Reuters-21578 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.

Affiliation:

Ingo Feinerer
Department of Statistics and Mathematics
Wirtschaftsuniversität Wien
Augasse 2–6
A-1090 Wien, Austria
E-mail: h0125130@wu-wien.ac.at

Kurt Hornik
Department of Statistics and Mathematics
Wirtschaftsuniversität Wien
Augasse 2–6
A-1090 Wien, Austria
E-mail: Kurt.Hornik@wu-wien.ac.at
URL: http://statmath.wu-wien.ac.at/~hornik/

David Meyer
Institute for Management Information Systems
Wirtschaftsuniversität Wien
Augasse 2–6
A-1090 Wien, Austria
E-mail: David.Meyer@wu-wien.ac.at
URL: http://wi.wu-wien.ac.at/home/meyer/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 25, Issue 5 *Submitted:* 2007-09-05
March 2008 *Accepted:* 2008-02-10