Introduction to the **tm** Package
Text Mining in R

Ingo Feinerer

November 11, 2008

Introduction

This vignette gives a short introduction to text mining in R utilizing the text mining framework provided by the **tm** package. We present methods for data import, corpus handling, preprocessing, meta data management, and creation of term-document matrices. Our focus is on the main aspects of getting started with text mining in R—an in-depth description of the text mining infrastructure offered by **tm** was published in the *Journal of Statistical Software* and is available at `http://www.jstatsoft.org/v25/i05`. An introductory article on text mining in R was published in *R News*, 8(2):19–22, October 2008, and can be obtained at `http://CRAN.R-project.org/doc/Rnews/`.

Data Import

The main structure for managing documents in **tm** is a so-called Corpus, representing a collection of text documents. A corpus can be created via its constructor `Corpus(object, readerControl, dbControl)`.

`object` must be a Source object which abstracts the input location. Available sources provided by **tm** are DirSource, VectorSource, CSVSource, GmaneSource and ReutersSource which handle a directory, a vector interpreting each component as document, CSV files, a RSS feed as delivered by the Gmane mailing list archive service, and a Reuters file containing several documents, respectively. Except DirSource, which is designed solely for directories on a file system, and VectorSource, which only accepts (character) vectors, all other implemented sources can take connections as input (a character string is interpreted as file path). `getSources()` lists available sources, and the user can create his own sources.

`readerControl` has to be a list with the named components `reader`, `language`, and `load`. The first component `reader` constructs a text document from elements delivered by a source. The **tm** package ships with several readers (`readPlain()` (default), `readRCV1()`, `readReut21578XML()`, `readGmane()`, `readNewsgroup()`, `readPDF()`, `readDOC()` and `readHTML()`). See `getReaders()` for an up-to-date list of available readers. Each source has a default reader which can be overridden. E.g., for `DirSource` the default just reads in the input files and interprets their content as text. The second component `language` sets the texts' language, the third component `load` can activate lazy document loading, i.e., whether documents should be immediately loaded into memory or not.

Finally `dbControl` has to be a list with the named components `useDb` indicating that database support should be activated, `dbName` giving the filename holding the sourced out objects (i.e., the database), and `dbType` holding a valid database type as supported by package **filehash**. Activated database support reduces the memory demand, however, access gets slower since each operation is limited by the hard disk's read and write capabilities.

So e.g., plain text files in the directory `txt` containing Latin (`la`) texts by the Roman poet *Ovid* can be read in with following code:

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

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

Another example could be mails from newsgroups (as found in the UCI KDD newsgroup data set):

```
> newsgroup <- system.file("texts", "newsgroup", package = "tm")
> Corpus(DirSource(newsgroup),
+        readerControl = list(reader = readNewsgroup,
+                             language = "en_US"))
```

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

For simple examples `VectorSource` is quite useful, as it can create a corpus from simple character vectors, e.g.:

```
> docs <- c("This is a text.", "This another one.")
> Corpus(VectorSource(docs))
```

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

Finally we create a corpus for some Reuters documents as example for later use:

```
> reut21578 <- system.file("texts", "reut21578", package = "tm")
> reuters <- Corpus(DirSource(reut21578),
+                   readerControl = list(reader = readReut21578XML))
```

Data Export

For the case you have created a text collection via manipulating other objects in R, thus do not have the texts already stored on a hard disk, and want to save the text documents to disk, you can simply use standard R routines for writing out plain text documents. E.g.,

```
> lapply(ovid,
+        function(x) writeLines(x, paste(ID(x), ".txt", sep = "")))
```

Alternatively there is the function `writeCorpus()` which encapsulates this functionality.

Inspecting Corpora

Custom `show()` and `summary()` methods are available, which hide the raw amount of information (consider a collection could consists of several thousand documents, like a database). `summary()` gives more details on meta data than `show()`, whereas the full content of text documents is displayed with `inspect()` on a collection.

```
> inspect(ovid[1:2])

A text document collection with 2 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

[[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.

[[2]]
 [1]     quas Hector sensurus erat, poscente magistro
 [2]          verberibus iussas praebuit ille manus.
 [3]     Aeacidae Chiron, ego sum praeceptor Amoris:
 [4]          saevus uterque puer, natus uterque dea.
 [5]     sed tamen et tauri cervix oneratur aratro,
 [6]
 [7]          frenaque magnanimi dente teruntur equi;
 [8]     et mihi cedet Amor, quamvis mea vulneret arcu
 [9]          pectora, iactatas excutiatque faces.
[10]     quo me fixit Amor, quo me violentius ussit,
[11]          hoc melior facti vulneris ultor ero:
[12]
[13]     non ego, Phoebe, datas a te mihi mentiar artes,
[14]          nec nos aëriae voce monemur avis,
```

```
[15]     nec mihi sunt visae Clio Cliusque sorores
[16]          servanti pecudes vallibus, Ascra, tuis:
[17]     usus opus movet hoc: vati parete perito;
```

Transformations

Once we have a text document collection we typically want to modify the documents in it, e.g., stemming, stopword removal, et cetera. In **tm**, all this functionality is subsumed into the concept of *transformation*s. Transformations are done via the `tmMap` function which applies a function 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.

Converting to Plain Text Documents

The text document collection `reuters` contains documents in XML format. We have no further use for the XML interna and just want to work with the text content. This can be done by converting the documents to plain text documents. It is done by the generic `asPlain()`.

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

Eliminating Extra Whitespace

Extra whitespace is eliminated by:

```
> reuters <- tmMap(reuters, stripWhitespace)
```

Convert to Lower Case

Conversion to lower case by:

```
> reuters <- tmMap(reuters, tmTolower)
```

Remove Stopwords

Removal of stopwords by:

```
> reuters <- tmMap(reuters, removeWords, stopwords("english"))
```

Stemming

Stemming is done by:

```
> tmMap(reuters, stemDoc)
```

```
A text document collection with 10 text documents
```

Filters

Often it is of special interest to filter out documents satisfying given properties. For this purpose the function `tmFilter` is designed. It is possible to write custom filter functions, but for most cases the default filter does its job: it integrates a minimal query language to filter meta data. Statements in this query language are statements as used for subsetting data frames.

E.g., the following statement filters out those documents having the string "`COMPUTER TERMINAL SYSTEMS <CPML> COMPLETES SALE`" as their heading and an `ID` equal to 10 (both are meta data slot variables of the text document).

```
> query <- "identifier == '10' &
+   heading == 'COMPUTER TERMINAL SYSTEMS <CPML> COMPLETES SALE'"
> tmFilter(reuters, query)

A text document collection with 0 text documents
```

There is also a full text search filter available which accepts regular expressions:

```
> tmFilter(reuters, FUN = searchFullText,
+          pattern = "partnership", doclevel = TRUE)

A text document collection with 1 text document
```

Meta Data Management

Meta data is used to annotate text documents or whole corpora with additional information. The easiest way to accomplish this with **tm** is to use the `meta()` function. A text document has a few predefined slots like `Author`, but can be extended with an arbitrary number of local meta data tags. Alternatively to `meta()` the function `DublinCore()` provides a full mapping between Simple Dublin Core meta data and **tm** meta data structures and can be similarly used to get and set meta data information for text documents, e.g.:

```
> DublinCore(crude[[1]], "Creator") <- "Ano Nymous"
> meta(crude[[1]])

Available meta data pairs are:
  Author       : Ano Nymous
  Cached       : TRUE
  DateTimeStamp: 1987-02-26 17:00:56
  Description  :
  Heading      : DIAMOND SHAMROCK (DIA) CUTS CRUDE PRICES
  ID           : 127
  Language     : en_US
  Origin       : Reuters-21578 XML
  URI          :
Dynamic local meta data pairs are:
$Topics
[1] "crude"
```

For corpora the story is a bit more difficult. Text document collections in **tm** have two types of meta data: one is the meta data on the document collection level (`corpus` level), the other is the meta data related to the individual documents (`indexed` level) in form of a data frame. The latter is often done for performance reasons (hence the named `indexed` for indexing) or because the meta data has an own entity but still relates directly to individual text documents, e.g., a classification result; the classifications directly relate to the documents, but the set of classification levels forms an own entity. Both cases can be handled with `meta()`:

```
> meta(crude, tag = "test", type = "corpus") <- "test meta"
> meta(crude, type = "corpus")

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

Slot "MetaData":
$create_date
[1] "2008-01-24 15:26:16 CET"

$creator
   LOGNAME
"feinerer"

$test
[1] "test meta"


Slot "children":
list()

> meta(crude, "foo") <- letters[1:20]
> meta(crude)

   MetaID foo
1       0   a
2       0   b
3       0   c
4       0   d
5       0   e
6       0   f
7       0   g
8       0   h
9       0   i
10      0   j
11      0   k
12      0   l
13      0   m
14      0   n
15      0   o
```

```
16       0   p
17       0   q
18       0   r
19       0   s
20       0   t
```

Standard Operators and Functions

Many standard operators and functions (`[`, `[<-`, `[[`, `[[<-`, `c()`, `length()`, `lapply()`, `sapply()`) are available for text document collections with semantics similar to standard R routines. E.g., `c()` concatenates two (or more) text document collections. Applied to several text documents it returns a text document collection. The meta data is automatically updated, if text document collections are concatenated (i.e., merged).

There is also a custom element-of operator—it checks whether a text document is already in a text document collection (meta data is not checked, only the corpus):

```
> reuters[[1]] %IN% reuters
```

```
[1] TRUE
```

```
> crude[[1]] %IN% reuters
```

```
[1] FALSE
```

Creating Term-Document Matrices

A common approach in text mining is to create a term-document matrix from a corpus. In the **tm** package the class TermDocMatrix handles sparse matrices for text document collections.

```
> tdm <- TermDocMatrix(reuters)
> Data(tdm)[1:5, 150:155]
```

```
5 x 6 sparse Matrix of class "dgCMatrix"
  weeks year york zone activities america
1     1    2    8    1          .       .
2     .    .    .    .          1       2
3     .    .    .    .          .       .
4     1    1    .    .          .       .
5     .    .    .    .          .       .
```

Operations on Term-Document Matrices

Besides the fact that on the `Data` part of this matrix a huge amount of R functions (like clustering, classifications, etc.) is possible, this package brings some shortcuts. Imagine we want to find those terms that occur at least five times, then we can use the `findFreqTerms()` function:

```
> findFreqTerms(tdm, 5)

 [1] "bags"        "cocoa"       "comissaria"  "crop"
 [5] "dec"         "dlrs"        "july"        "mln"
 [9] "sales"       "sept"        "smith"       "times"
[13] "total"       "york"        "analysts"    "bankamerica"
[17] "debt"        "price"       "stock"       "level"
[21] "apr"         "feb"         "mar"         "nil"
[25] "prev"        "computer"    "terminal"
```

Or we want to find associations (i.e., terms which correlate) with at least 0.97 correlation for the term `crop`, then we use `findAssocs()` (we only display ten arbitrary associations found):

```
> findAssocs(tdm, "crop", 0.97)[31:40]

       bahia         bean       booked       butter
        0.98         0.98         0.98         0.98
      buyers         cake     carnival certificates
        0.98         0.98         0.98         0.98
       cocoa   comissaria
        0.98         0.98
```

The function also accepts a matrix as first argument (which does not inherit from a term-document matrix). This matrix is then interpreted as a correlation matrix and directly used. With this approach different correlation measures can be employed.

Term-document matrices tend to get very big already for normal sized data sets. Therefore we provide a method to remove *sparse* terms, i.e., terms occurring only in very few documents. Normally, this reduces the matrix dramatically without losing significant relations inherent to the matrix:

```
> removeSparseTerms(tdm, 0.4)

An object of class "TermDocMatrix"
Slot "Data":
10 x 2 sparse Matrix of class "dgTMatrix"
    Terms
     dlrs reuter
  1    14      1
  2     .      1
  3     2      1
  4     3      1
  5     2      1
  6     .      1
  7     1      1
  8     2      1
  9     .      1
  10    4      1

Slot "Weighting":
[1] "term frequency" "tf"
```

This function call removes those terms which have at least a 40 percentage of sparse (i.e., terms occurring 0 times in a document) elements.

Dictionary

A dictionary is a (multi-)set of strings. It is often used to represent relevant terms in text mining. We provide a class Dictionary implementing such a dictionary concept. It can be created via the `Dictionary()` constructor, e.g.,

```
> (d <- Dictionary(c("dlrs", "crude", "oil")))

An object of class "Dictionary"
[1] "dlrs"  "crude" "oil"
```

and may be passed over to the `TermDocMatrix()` constructor. Then the created matrix is tabulated against the dictionary, i.e., only terms from the dictionary appear in the matrix. This allows to restrict the dimension of the matrix a priori and to focus on specific terms for distinct text mining contexts, e.g.,

```
> tdmD <- TermDocMatrix(reuters, list(dictionary = d))
> Data(tdmD)

10 x 3 sparse Matrix of class "dgCMatrix"
    Terms
     dlrs crude oil
  1    14     .   .
  2     .     .   3
  3     2     .   .
  4     3     .   .
  5     2     .   .
  6     .     .   2
  7     1     .   .
  8     2     .   1
  9     .     .   .
  10    4     .   .
```