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

Ingo Feinerer

July 1, 2009

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*~(Feinerer et~al., 2008). An introductory article on text mining in R was published in *R News*~(Feinerer, 2008).

Data Import

The main structure for managing documents in **tm** is a so-called Corpus, representing a collection of text documents. A corpus is an abstract concept (modelled via a virtual class), and there can exist several implementations in parallel. The default implementation is the so-called VCorpus (short for *Volatile Corpus*) which realizes a semantics as known from most R objects: corpora are R objects held fully in memory. Such a volatile corpus can be created via the constructor `Corpus(object, readerControl)`. Another implementation is the PCorpus which implements a *Permanent Corpus* semantics, i.e., the documents are physically located on a disk, and its changes are reflected to all R objects associated with it.

Within the corpus constructor, `object` must be a Source object which abstracts the input location. **tm** provides a set of predefined sources, e.g., DirSource, VectorSource, or DataframeSource, which handle a directory, a vector interpreting each component as document, data frame like structures (like CSV files), respectively. Except DirSource, which is designed solely for directories on a file system, and VectorSource, which only accepts (character) vectors, most 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.

The second argument `readerControl` of the corpus constructor has to be a list with the named components `reader` and `language`. The first component `reader` constructs a text document from elements delivered by a source. The **tm** package ships with several readers (e.g., `readPlain()`, `readGmane()`, `readNewsgroup()`, `readRCV1()`, `readReut21578XMLasPlain()`, `readPDF()`, `readDOC()`, ...). 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. Finally, the second component `language` sets the texts' language (preferably using ISO 639-2 codes).

In case of a permanent corpus, `dbControl` has to be a list with the named components `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 (`lat`) 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(language = "lat")))
```

```
A corpus 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 = "eng"))
```

```
A corpus 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 corpus 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 corpus 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 corpus could consist 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()`.

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

A corpus 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]]
    Si quis in hoc artem populo non novit amandi,
         hoc legat et lecto carmine doctus amet.
    arte citae veloque rates remoque moventur,
         arte leves currus: arte regendus amor.

    curribus Automedon lentisque erat aptus habenis,
         Tiphys in Haemonia puppe magister erat:
    me Venus artificem tenero praefecit Amori;
         Tiphys et Automedon dicar Amoris ego.
    ille quidem ferus est et qui mihi saepe repugnet:

         sed puer est, aetas mollis et apta regi.
    Phillyrides puerum cithara perfecit Achillem,
         atque animos placida contudit arte feros.
    qui totiens socios, totiens exterruit hostes,
         creditur annosum pertimuisse senem.

[[2]]
```

```
quas Hector sensurus erat, poscente magistro
     verberibus iussas praebuit ille manus.
Aeacidae Chiron, ego sum praeceptor Amoris:
     saevus uterque puer, natus uterque dea.
sed tamen et tauri cervix oneratur aratro,

     frenaque magnanimi dente teruntur equi;
et mihi cedet Amor, quamvis mea vulneret arcu
     pectora, iactatas excutiatque faces.
quo me fixit Amor, quo me violentius ussit,
     hoc melior facti vulneris ultor ero:

non ego, Phoebe, datas a te mihi mentiar artes,
     nec nos aëriae voce monemur avis,
nec mihi sunt visae Clio Cliusque sorores
     servanti pecudes vallibus, Ascra, tuis:
usus opus movet hoc: vati parete perito;
```

Transformations

Once we have a corpus 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 corpus. Basically, all transformations work on single text documents and `tmMap` just applies them to all documents in a corpus.

Converting to Plain Text Documents

The corpus `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)
```

Note that alternatively we could have read in the files with the `readReut21578XMLasPlain` reader which already returns a plain text document in the first place.

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 corpus 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 `sFilter` 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 an `ID` equal to 10 and the string "`COMPUTER TERMINAL SYSTEMS <CPML> COMPLETES SALE`" as their heading (both are meta data slot variables of the text document).

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

A corpus with 1 text document
```

There is also a full text search filter available (which is default when no explicit filter function `FUN` is specified) accepting regular expressions:

```
> tmFilter(reuters, pattern = "partnership")

A corpus 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
  DateTimeStamp: 1987-02-26 17:00:56
  Description  :
  Heading      : DIAMOND SHAMROCK (DIA) CUTS CRUDE PRICES
  ID           : 127
  Language     : eng
  Origin       : Reuters-21578 XML
User-defined local meta data pairs are:
$Topics
[1] "crude"
```

For corpora the story is a bit more difficult. Corpora in **tm** have two types of meta data: one is the meta data on the corpus level (`corpus`), the other is the meta data related to the individual documents (`indexed`) 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] "2009-06-29 06:14:55 GMT"

$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 corpora with semantics similar to standard R routines. E.g., `c()` concatenates two (or more) corpora. Applied to several text documents it returns a corpus. The meta data is automatically updated, if corpora are concatenated (i.e., merged).

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

```
> 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 classes TermDocumentMatrix and DocumentTermMatrix (depending on whether you want terms as rows and documents as columns, or vice versa) handle sparse matrices for corpora.

```
> dtm <- DocumentTermMatrix(reuters)
> inspect(dtm[1:5, 150:155])
```

```
A document-term matrix (5 documents, 6 terms)

Non-/sparse entries: 4/26
Sparsity           : 87%
Maximal term length: 12
Weighting          : term frequency (tf)

  exclusive exercisable expect expected expects experiencing
1         0           0      0        1       0            1
2         0           0      0        0       0            0
3         0           0      0        0       0            0
4         0           0      1        1       0            0
5         0           0      0        0       0            0
```

Operations on Term-Document Matrices

Besides the fact that on this matrix a huge amount of R functions (like clustering, classifications, etc.) can be applied, 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(dtm, 5)

 [1] "bags"         "cocoa"        "comissaria"  "crop"         "dec"
 [6] "dlrs"         "july"         "mln"          "sales"        "sept"
[11] "smith"        "times"        "york"         "analysts"     "bankamerica"
[16] "debt"         "stock"        "level"        "price"        "apr"
[21] "feb"          "mar"          "nil"          "prev"         "total"
[26] "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(dtm, "crop", 0.97)[31:40]

     drought          dry          end    estimated    estimates experiencing
        0.98         0.98         0.98         0.98         0.98         0.98
   exporters      farmers        final          fit
        0.98         0.98         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:

```
> inspect(removeSparseTerms(dtm, 0.4))

A document-term matrix (10 documents, 2 terms)

Non-/sparse entries: 17/3
Sparsity           : 15%
Maximal term length: 6
Weighting          : term frequency (tf)

   dlrs reuter
1    14      1
2     0      1
3     2      1
4     3      1
5     2      1
6     0      1
```

```
7      1      1
8      2      1
9      0      1
10     4      1
```

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 `DocumentTermMatrix()` constructor. Then the created matrix is tabulated against the dictionary, i.e., only terms from the dictionary appear in the matrix (terms not occurring in the document are skipped for performance reasons). This allows to restrict the dimension of the matrix a priori and to focus on specific terms for distinct text mining contexts, e.g.,

```
> inspect(DocumentTermMatrix(reuters, list(dictionary = d)))

A document-term matrix (10 documents, 2 terms)

Non-/sparse entries: 10/10
Sparsity           : 50%
Maximal term length: 4
Weighting          : term frequency (tf)

    Terms
Docs dlrs oil
  1    14   0
  2     0   3
  3     2   0
  4     3   0
  5     2   0
  6     0   2
  7     1   0
  8     2   1
  9     0   0
  10    4   0
```

References

I.~Feinerer. An introduction to text mining in R. *R News*, 8(2):19–22, Oct. 2008. URL `http://CRAN.R-project.org/doc/Rnews/`.

I.~Feinerer, K.~Hornik, and D.~Meyer. Text mining infrastructure in R. *Journal of Statistical Software*, 25(5): 1–54, March 2008. ISSN 1548-7660. URL `http://www.jstatsoft.org/v25/i05`.