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

Ingo Feinerer

July 27, 2022

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, metadata 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, 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. We denote this as volatile since once the R object is destroyed, the whole corpus is gone. Such a volatile corpus can be created via the constructor `VCorpus(x, readerControl)`. Another implementation is the PCorpus which implements a *Permanent Corpus* semantics, i.e., the documents are physically stored outside of R (e.g., in a database), corresponding R objects are basically only pointers to external structures, and changes to the underlying corpus are reflected to all R objects associated with it. Compared to the volatile corpus the corpus encapsulated by a permanent corpus object is not destroyed if the corresponding R object is released.

Within the corpus constructor, `x` 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, or 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 users can create their 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()`, `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, a third argument `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 <- VCorpus(DirSource(txt, encoding = "UTF-8"),
+                  readerControl = list(language = "lat")))

<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 5
```

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

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

<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2
```

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

```
> reut21578 <- system.file("texts", "crude", package = "tm")
> reuters <- VCorpus(DirSource(reut21578, mode = "binary"),
+                    readerControl = list(reader = readReut21578XMLasPlain))
```

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 `writeCorpus()`

```
> writeCorpus(ovid)
```

which writes a character representation of the documents in a corpus to multiple files on disk.

Inspecting Corpora

Custom `print()` methods are available which hide the raw amount of information (consider a corpus could consist of several thousand documents, like a database). `print()` gives a concise overview whereas more details are displayed with `inspect()`.

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

<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2

[[1]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 676

[[2]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 700
```

Individual documents can be accessed via `[[`, either via the position in the corpus, or via their identifier.

```
> meta(ovid[[2]], "id")

[1] "ovid_2.txt"

> identical(ovid[[2]], ovid[["ovid_2.txt"]])

[1] TRUE
```

A character representation of a document is available via `as.character()` which is also used when inspecting a document:

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

```
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 700

    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;
```

```
> lapply(ovid[1:2], as.character)

$ovid_1.txt
 [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."

$ovid_2.txt
 [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 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 a *transformation*. Transformations are done via the `tm_map()` function which applies (maps) a function to all elements of the corpus. Basically, all transformations work on single text documents and `tm_map()` just applies them to all documents in a corpus.

Eliminating Extra Whitespace

Extra whitespace is eliminated by:

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

Convert to Lower Case

Conversion to lower case by:

```
> reuters <- tm_map(reuters, content_transformer(tolower))
```

We can use arbitrary character processing functions as transformations as long as the function returns a text document. In this case we use `content_transformer()` which provides a convenience wrapper to access and set the content of a document. Consequently most text manipulation functions from base R can directly be used with this wrapper. This works for `tolower()` as used here but also e.g. for `gsub()` which comes quite handy for a broad range of text manipulation tasks.

Remove Stopwords

Removal of stopwords by:

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

Stemming

Stemming is done by:

```
> tm_map(reuters, stemDocument)
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 20
```

Filters

Often it is of special interest to filter out documents satisfying given properties. For this purpose the function `tm_filter` is designed. It is possible to write custom filter functions which get applied to each document in the corpus. Alternatively, we can create indices based on selections and subset the corpus with them. E.g., the following statement filters out those documents having an `ID` equal to `"237"` and the string `"INDONESIA SEEN AT CROSSROADS OVER ECONOMIC CHANGE"` as their heading.

```
> idx <- meta(reuters, "id") == '237' &
+   meta(reuters, "heading") == 'INDONESIA SEEN AT CROSSROADS OVER ECONOMIC CHANGE'
> reuters[idx]
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 1
```

Metadata Management

Metadata 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 attributes like `author` but can be extended with an arbitrary number of additional user-defined metadata tags. These additional metadata tags are individually attached to a single text document. From a corpus perspective these metadata attachments are locally stored together with each individual text document. Alternatively to `meta()` the function `DublinCore()` provides a full mapping between Simple Dublin Core metadata and **tm** metadata structures and can be similarly used to get and set metadata information for text documents, e.g.:

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

  author       : Ano Nymous
  datetimestamp: 1987-02-26 17:00:56
  description  :
  heading      : DIAMOND SHAMROCK (DIA) CUTS CRUDE PRICES
  id           : 127
  language     : en
  origin       : Reuters-21578 XML
  topics       : YES
  lewissplit   : TRAIN
  cgisplit     : TRAINING-SET
  oldid        : 5670
  places       : usa
  people       : character(0)
  orgs         : character(0)
  exchanges    : character(0)
```

For corpora the story is a bit more sophisticated. Corpora in **tm** have two types of metadata: one is the metadata on the corpus level (`corpus`), the other is the metadata 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 metadata 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")

$test
[1] "test meta"

attr(,"class")
[1] "CorpusMeta"

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

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

```
15   o
16   p
17   q
18   r
19   s
20   t
```

Standard Operators and Functions

Many standard operators and functions (`[`, `[<-`, `[[`, `[[<-`, `c()`, `lapply()`) 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 metadata is automatically updated, if corpora are concatenated (i.e., merged).

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) employ sparse matrices for corpora. Inspecting a term-document matrix displays a sample, whereas `as.matrix()` yields the full matrix in dense format (which can be very memory consuming for large matrices).

```
> dtm <- DocumentTermMatrix(reuters)
> inspect(dtm)

<<DocumentTermMatrix (documents: 20, terms: 1183)>>
Non-/sparse entries: 1908/21752
Sparsity           : 92%
Maximal term length: 17
Weighting          : term frequency (tf)
Sample             :
     Terms
Docs  crude dlrs last mln oil opec prices reuter said saudi
  144     0    0    1   4  11   10      3      1    9     0
  236     1    2    4   4   7    6      2      1    6     0
  237     0    1    3   1   3    1      0      1    0     0
  242     0    0    0   0   3    2      1      1    3     1
  246     0    0    2   0   4    1      0      1    4     0
  248     0    3    1   3   9    6      7      1    5     5
  273     5    2    7   9   5    5      4      1    5     7
  489     0    1    0   2   4    0      2      1    2     0
  502     0    1    0   2   4    0      2      1    2     0
  704     0    0    0   0   3    0      2      1    3     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] "15.8"          "abdul-aziz"    "ability"       "accord"
 [5] "agency"        "agreement"     "ali"           "also"
 [9] "analysts"      "arab"          "arabia"        "barrel."
[13] "barrels"       "billion"       "bpd"           "budget"
[17] "company"       "crude"         "daily"         "demand"
[21] "dlrs"          "economic"      "emergency"     "energy"
[25] "exchange"      "expected"      "exports"       "futures"
[29] "government"    "group"         "gulf"          "help"
```

```
[33] "hold"          "industry"      "international" "january"
[37] "kuwait"        "last"          "market"        "may"
[41] "meeting"       "minister"      "mln"           "month"
[45] "nazer"         "new"           "now"           "nymex"
[49] "official"      "oil"           "one"           "opec"
[53] "output"        "pct"           "petroleum"     "plans"
[57] "posted"        "present"       "price"         "prices"
[61] "prices,"       "prices."       "production"    "quota"
[65] "quoted"        "recent"        "report"        "research"
[69] "reserve"       "reuter"        "said"          "said."
[73] "saudi"         "sell"          "sheikh"        "sources"
[77] "study"         "traders"       "u.s."          "united"
[81] "west"          "will"          "world"
```

Or we want to find associations (i.e., terms which correlate) with at least 0.8 correlation for the term `opec`, then we use `findAssocs()`:

```
> findAssocs(dtm, "opec", 0.8)

$opec
  meeting emergency       oil      15.8  analysts    buyers      said   ability
     0.88      0.87      0.87      0.85      0.85      0.83      0.82      0.80
```

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))

<<DocumentTermMatrix (documents: 20, terms: 3)>>
Non-/sparse entries: 58/2
Sparsity           : 3%
Maximal term length: 6
Weighting          : term frequency (tf)
Sample             :
     Terms
Docs  oil reuter said
  127   5      1    1
  144  11      1    9
  236   7      1    6
  242   3      1    3
  246   4      1    4
  248   9      1    5
  273   5      1    5
  352   5      1    1
  489   4      1    2
  502   4      1    2
```

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 denote relevant terms in text mining. We represent a dictionary with a character vector which may be passed to the `DocumentTermMatrix()` constructor as a control argument. 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.,

```
> inspect(DocumentTermMatrix(reuters,
+                            list(dictionary = c("prices", "crude", "oil"))))
```

```
<<DocumentTermMatrix (documents: 20, terms: 3)>>
Non-/sparse entries: 41/19
Sparsity           : 32%
Maximal term length: 6
Weighting          : term frequency (tf)
Sample             :
     Terms
Docs  crude oil prices
  127     2   5      3
  144     0  11      3
  236     1   7      2
  248     0   9      7
  273     5   5      4
  352     0   5      4
  353     2   4      1
  489     0   4      2
  502     0   4      2
  543     2   2      2
```

Performance

Often you do not need all the generality, modularity and full range of features offered by **tm** as this sometimes comes at the price of performance.

SimpleCorpus provides a corpus which is optimized for the most common usage scenario: importing plain texts from files in a directory or directly from a vector in R, preprocessing and transforming the texts, and finally exporting them to a term-document matrix. The aim is to boost performance and minimize memory pressure. It loads all documents into memory, and is designed for medium-sized to large data sets.

However, it operates only under the following contraints:

- only `DirSource` and `VectorSource` are supported,
- no custom readers, i.e., each document is read in and stored as plain text (as a string, i.e., a character vector of length one),
- transformations applied via `tm_map` must be able to process strings and return strings,
- no lazy transformations in `tm_map`,
- no meta data for individual documents (i.e., no `"local"` in `meta()`).

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`.