Introduction to the tm Package
Text Mining in R
Ingo Feinerer
June 10, 2014

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:

```r
> txt <- system.file("texts", "txt", package = "tm")
> (ovid <- VCorpus(DirSource(txt, encoding = "UTF-8"),
+ readerControl = list(language = "lat")))
<<VCorpus (documents: 5, metadata (corpus/indexed): 0/0)>>
```

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))

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),
+ 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 the full content of text documents is displayed with `inspect()`.

> inspect(ovid[1:2])

[[1]]

<<PlainTextDocument (metadata: 7)>>

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, astas 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]]

<<PlainTextDocument (metadata: 7)>>

quas Hector sensurus erat, poscente magistro
    verberibus iussas praebuit ille manus.
Aeacidae Chiron, ego sum praeceptor Amoris:
    saevis 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:
Individual documents can be accessed via `[[`, either via the position in the corpus, or via their identifier.

```r
> meta(ovid[[2]], "id")
[1] "ovid_2.txt"
> identical(ovid[[2]], ovid[["ovid_2.txt"]])
[1] TRUE
```

### 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:

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

#### Convert to Lower Case

Conversion to lower case by:

```r
> 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:

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

#### Stemming

Stemming is done by:

```r
> tm_map(reuters, stemDocument)
<<VCorpus (documents: 20, metadata (corpus/indexed): 0/0)>>
```

#### 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 CROSROADS OVER ECONOMIC CHANGE” as their heading.
idx <- meta(reuters, "id") == '237' &
+ meta(reuters, "heading") == 'INDONESIA SEEN AT CROSSROADS OVER ECONOMIC CHANGE'

> reuters[idx]

<<VCorpus (documents: 1, metadata (corpus/indexed): 0/0)>>

**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.:

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

Metadata:

- 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():

```r
> 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
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.

```r
> dtm <- DocumentTermMatrix(reuters)
> inspect(dtm[5:10, 740:743])
```

Non-/sparse entries: 8/16
Sparsity : 67%
Maximal term length: 8
Weighting : term frequency (tf)

Terms

<table>
<thead>
<tr>
<th>Docs</th>
<th>one-week</th>
<th>opec</th>
<th>opec&quot;s</th>
<th>opec's</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>236</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>237</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>246</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>248</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

```r
> 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"
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()`:

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

```
  Term       Value
  ------------
    opec         1
      meeting  0.88
        emergency  0.87
          oil  0.87
            15.8  0.85
      analysts  0.85
         buyers  0.83
       said  0.82
       ability  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:

```r
> 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)
```

```
Terms
 Docs oil reuter said
 127  5  1  1
 144  1  1  9
 191  2  1  1
 194  1  1  1
 211  1  1  3
 236  7  1  6
 237  3  1  0
 242  3  1  3
 246  4  1  4
 248  9  1  5
 273  5  1  5
 349  3  1  1
 352  5  1  1
 353  4  1  1
 368  3  1  2
 489  4  1  2
 502  4  1  2
 543  2  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.,

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

Non-/sparse entries: 41/19
Sparsity : 32%
Maximal term length: 6
Weighting : term frequency (tf)

<table>
<thead>
<tr>
<th>Terms</th>
<th>Docs</th>
<th>crude</th>
<th>oil</th>
<th>prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>0</td>
<td>11</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>191</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>236</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>242</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>246</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>248</td>
<td>0</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>273</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>349</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>352</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>353</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>368</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>489</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>502</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>543</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>708</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

References
