

Extensions

How to Handle Custom File Formats

Ingo Feinerer

December 10, 2025

Introduction

The possibility to handle custom file formats is a substantial feature in any modern text mining infrastructure. **tm** has been designed aware of this aspect from the beginning on, and has modular components which allow for extensions. A general explanation of **tm**'s extension mechanism is described by Feinerer et al. (2008, Sec. 3.3), with an updated description as follows.

Sources

A source abstracts input locations and provides uniform methods for access. Each source must provide implementations for following interface functions:

close() closes the source and returns it,
eoi() returns TRUE if the end of input of the source is reached,
getElem() fetches the element at the current position,
length() gives the number of elements,
open() opens the source and returns it,
reader() returns a default reader for processing elements,
pGetElem() (optional) retrieves all elements in parallel at once, and
stepNext() increases the position in the source to the next element.

Retrieved elements must be encapsulated in a list with the named components **content** holding the document and **uri** pointing to the origin of the document (e.g., a file path or a URL; NULL if not applicable or unavailable).

Custom sources are required to inherit from the virtual base class **Source** and typically do so by extending the functionality provided by the simple reference implementation **SimpleSource**.

E.g., a simple source which accepts an R vector as input could be defined as

```
> VecSource <- function(x)
+   SimpleSource(length = length(x), content = as.character(x),
+               class = "VecSource")
```

which overrides a few defaults (see **?SimpleSource** for defaults) and stores the vector in the **content** component. The functions **close()**, **eoi()**, **open()**, and **stepNext()** have reasonable default methods already for the **SimpleSource** class: the identity function for **open()** and **close()**, incrementing a position counter for **stepNext()**, and comparing the current position with the number of available elements as claimed by **length()** for **eoi()**, respectively. So we only need custom methods for element access:

```
> getElem.VecSource <-
+   function(x) list(content = x$content[x$position], uri = NULL)
> pGetElem.VecSource <-
+   function(x) lapply(x$content, function(y) list(content = y, uri = NULL))
```

Readers

Readers are functions for extracting textual content and metadata out of elements delivered by a source and for constructing a text document. Each reader must accept following arguments in its signature:

elem a list with the named components **content** and **uri** (as delivered by a source via `getElem()` or `pGetElem()`),
language a string giving the language, and
id a character giving a unique identifier for the created text document.

The element **elem** is typically provided by a source whereas the language and the identifier are normally provided by a corpus constructor (for the case that `elem$content` does not give information on these two essential items).

In case a reader expects configuration arguments we can use a function generator. A function generator is indicated by inheriting from class `FunctionGenerator` and `function`. It allows us to process additional arguments, store them in an environment, return a reader function with the well-defined signature described above, and still be able to access the additional arguments via lexical scoping. All corpus constructors in package `tm` check the reader function for being a function generator and if so apply it to yield the reader with the expected signature.

E.g., the reader function `readPlain()` is defined as

```
> readPlain <- function(elem, language, id)
+   PlainTextDocument(elem$content, id = id, language = language)
```

For examples on readers using the function generator please have a look at `?readPDF` or `?readPDF`.

However, for many cases, it is not necessary to define each detailed aspect of how to extend `tm`. Typical examples are XML files which are very common but can be rather easily handled via standard conforming XML parsers. The aim of the remainder in this document is to give an overview on how simpler, more user-friendly, forms of extension mechanisms can be applied in `tm`.

Custom Data Formats

A general situation is that you have gathered together some information into a tabular data structure (like a data frame or a list matrix) that suffices to describe documents in a corpus. However, you do not have a distinct file format because you extracted the information out of various resources, e.g., as delivered by `readtext()` in package `readtext`. Now you want to use your information to build a corpus which is recognized by `tm`.

We assume that your information is put together in a data frame. E.g., consider the following example:

```
> df <- data.frame(doc_id    = c("doc 1"      , "doc 2"      , "doc 3"      ),
+                     text       = c("content 1", "content 2", "content 3"),
+                     title     = c("title 1"   , "title 2"  , "title 3"  ),
+                     authors   = c("author 1" , "author 2" , "author 3" ),
+                     topics    = c("topic 1"  , "topic 2" , "topic 3" ),
+                     stringsAsFactors = FALSE)
```

We want to map the data frame rows to the relevant entries of a text document.

An entry `text` in the mapping will be matched to fill the actual content of the text document, `doc_id` will be used as document ID, all other fields will be used as metadata tags. So we can construct a corpus out of the data frame:

```
> (corpus <- Corpus(DataframeSource(df)))

<<SimpleCorpus>>
Metadata: corpus specific: 1, document level (indexed): 3
Content: documents: 3

> corpus[[1]]

<<PlainTextDocument>>
Metadata: 7
Content: chars: 9

> meta(corpus[[1]])
```

```

author      : character(0)
datetimestamp: 2025-12-10 12:27:19.7882430553436
description : character(0)
heading     : character(0)
id          : doc 1
language    : en
origin      : character(0)

```

Custom XML Sources

Many modern file formats already come in XML format which allows to extract information with any XML conforming parser, e.g., as implemented in R by the **xml2** package.

Now assume we have some custom XML format which we want to access with **tm**. Then a viable way is to create a custom XML source which can be configured with only a few commands. E.g., have a look at the following example:

```

> custom.xml <- system.file("texts", "custom.xml", package = "tm")
> print(readLines(custom.xml), quote = FALSE)

[1] <?xml version="1.0"?>
[2] <corpus>
[3]   <document short="invisible man">
[4]     <writer>Ano Nymous</writer>
[5]     <caption>The Invisible Man</caption>
[6]     <description>A story about an invisible man.</description>
[7]     <type>Science fiction</type>
[8]   </document>
[9]   <document short="(ne)scio">
[10]    <writer>Sokrates</writer>
[11]    <caption>Scio Nescio</caption>
[12]    <description>I know that I know nothing.</description>
[13]    <type>Classics</type>
[14]  </document>
[15] </corpus>

```

As you see there is a top-level tag stating that there is a corpus, and several document tags below. In fact, this structure is very common in XML files found in text mining applications (e.g., both the Reuters-21578 and the Reuters Corpus Volume 1 data sets follow this general scheme). In **tm** we expect a source to deliver self-contained blocks of information to a reader function, each block containing all information necessary such that the reader can construct a (subclass of a) **TextDocument** from it.

The **XMLSource()** function can now be used to construct a custom XML source. It has three arguments:

x a character giving a uniform resource identifier,

parser a function accepting an XML document (as delivered by **read_xml()** in package **xml2**) as input and returning a XML elements/nodes (each element/node will then be delivered to the reader as a self-contained block),

reader a reader function capable of turning XML elements/nodes as returned by the parser into a subclass of **TextDocument**.

E.g., a custom source which can cope with our custom XML format could be:

```

> mySource <- function(x)
+   XMLSource(x, parser = xml2::xml_children, reader = myXMLReader)

```

As you notice in this example we also provide a custom reader function (**myXMLReader**). See the next section for details.

Custom XML Readers

As we saw in the previous section we often need a custom reader function to extract information out of XML chunks (typically as delivered by some source). Fortunately, **tm** provides an easy way to define custom XML reader functions. All you need to do is to provide a so-called *specification*.

Let us start with an example which defines a reader function for the file format from the previous section:

```

> myXMLReader <- readXML(
+   spec = list(author = list("node", "writer"),
+               content = list("node", "description"),
+               datetimestamp = list("function",
+                                     function(x) as.POSIXlt(Sys.time(), tz = "GMT")),
+               description = list("node", "@short"),
+               heading = list("node", "caption"),
+               id = list("function", function(x) tempfile()),
+               origin = list("unevaluated", "My private bibliography"),
+               type = list("node", "type")),
+   doc = PlainTextDocument())

```

Formally, `readXML()` is the relevant function which constructs an reader. The customization is done via the first argument `spec`, the second provides an empty instance of the document which should be returned (augmented with the extracted information out of the XML chunks). The specification must consist of a named list of lists each containing two character vectors. The constructed reader will map each list entry to the content or a metadatum of the text document as specified by the named list entry. Valid names include `content` to access the document's content, and character strings which are mapped to metadata entries.

Each list entry must consist of two character vectors: the first describes the type of the second argument, and the second is the specification entry. Valid combinations are:

`type = "node", spec = "XPathExpression"` the XPath (1.0) expression `spec` extracts information out of an XML node (as seen for `author`, `content`, `description`, `heading`, and `type` in our example specification).

`type = "function", spec = function(doc) ...` The function `spec` is called, passing over the XML document (as delivered by `read_xml()` from package `xml2`) as first argument (as seen for `datetimestamp` and `id`). As you notice in our example nobody forces us to actually use the passed over document, instead we can do anything we want (e.g., create a unique character vector via `tempfile()` to have a unique identification string).

`type = "unevaluated", spec = "String"` the character vector `spec` is returned without modification (e.g., `origin` in our specification).

Now that we have all we need to cope with our custom file format, we can apply the source and reader function at any place in `tm` where a source or reader is expected, respectively. E.g.,

```
> corpus <- VCorpus(mySource(custom.xml))
```

constructs a corpus out of the information in our XML file:

```

> corpus[[1]]

<<PlainTextDocument>>
Metadata: 8
Content: chars: 31

> meta(corpus[[1]])

author      : Ano Nymous
datetimestamp: 2025-12-10 12:27:19.8029866218567
description  : invisible man
heading      : The Invisible Man
id          : /tmp/RtmpUDIn6/file2a82074c27e3b7
language     : en
origin       : My private bibliography
type         : Science fiction

```

References

- 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>.