Extensions
How to Handle Custom File Formats

Ingo Feinerer

January 11, 2014

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 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. Internally a source is represented as a `list` with the class attribute `Source` and has the following components:

DefaultReader a reader function suitable for processing the elements delivered by the source,

Encoding a character specifying the encoding of the elements delivered by the source,

Length a numeric denoting the number of elements delivered by the source (set to `NA` if unknown),

Names a character vector giving element names,

Position a numeric indicating the position in the source, and

Vectorized a logical indicating the ability for parallel element access.

Custom sources are required to inherit from this virtual base class `Source` and typically extend this internal list structure with additional named components necessary for representing and storing information for document access. The function `Source()` is a constructor with the signature `Source(defaultReader, encoding, length, names, position, vectorized, class)` and should be used when creating custom sources as it populates the corresponding list components and ensures the correct base class.

Each source must provide implementations for following interface functions:

eoi() returns `TRUE` if the end of input of the source is reached,

getElem() fetches the element at the current position,

pGetElem() if `Vectorized` is set, 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 connection; `NA` if not applicable or unavailable).

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

```
> VecSource <- function(x) {
+     s <- Source(length = length(x), names = names(x), class = "VectorSource")
+     s$Content <- as.character(x)
+     s
+ }
```

which overrides a few defaults (see `?Source` for defaults) and stores the vector in the `Content` list component. The functions `stepNext()` and `eoi()` have reasonable default methods already for the `Source` base class (basically just incrementing the `Position` component and comparing the current position with the number of available elements as claimed by `Length`, respectively), so we only need custom methods for element access:

```
> getElem.VectorSource <-
+ function(x) list(content = x$Content[x$Position], uri = NA)
> pGetElem.VectorSource <-
+ function(x) lapply(x$Content, function(y) list(content = y, uri = NA))
```

Readers

Readers are functions for extracting textual content and meta data 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 text's language, and

id a unique identification string for the returned text document.

The element 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 (see `?FunctionGenerator` for details). 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 able to access the additional arguments via lexical scoping. The corpus constructor `Corpus()` will check the 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 `?readTabular`.

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 General Purpose Readers

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. Now you want to use your information to build a corpus which is recognized by **tm**.

This can be done in **tm** by generating a custom reader function for tabular data structures via `readTabular()` that can be configured via a so-called *mapping*. A mapping describes how your information in the tabular data structure is mapped to the individual attributes of text documents in **tm**.

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

```
> df <- data.frame(contents = 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 `contents`, `title`, `authors`, and `topics` to the relevant entries of a text document. `readTabular()` always returns a `PlainTextDocument`, so possible attributes are:

```
> names(attributes(PlainTextDocument()))

[1] "Author"        "DateTimeStamp" "Description"   "Heading"
[5] "ID"            "Language"      "LocalMetaData" "Origin"
[9] "class"
```

Except `class` which contains inheritance information and which should not be modified, all other attributes can be used to access and set predefined meta data information for a text document. Additional user-defined meta data values can be stored via `LocalMetaData`. An entry `Content` in the mapping will be matched to fill the actual content of the text document.

So for our data frame we define a possible mapping as follows:

```
> m <- list(Content = "contents", Heading = "title",
+           Author = "authors", Topic = "topics")
```

Now we can construct a customized reader by passing over the previously defined mapping:

```
> myReader <- readTabular(mapping = m)
```

Finally we can apply our reader function at any place where **tm** expects a reader. E.g., we can construct a corpus out of the data frame:

```
> (corpus <- Corpus(DataframeSource(df), readerControl = list(reader = myReader)))

A corpus with 3 text documents
```

As we see the information is mapped as we want to the individual attributes of each document:

```
> corpus[[1]]

content 1

> meta(corpus[[1]])

Available meta data pairs are:
  Author       : author 1
  DateTimeStamp: 2014-01-11 15:39:30
  Description  :
  Heading      : title 1
  ID           : 1
  Language     : en
  Origin       :
User-defined local meta data pairs are:
$Topic
[1] "topic 1"
```

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 **XML** 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 four arguments:

x either a character identifying a file or a connection,

parser a function accepting an XML tree (as delivered by `xmlTreeParse()` in package **XML**) as input and returning a list of XML elements (each list element will then be delivered to the reader as such a self-contained block),

reader a reader function capable of turning XML elements as returned by the parser into a subclass of `TextDocument`,

encoding a character giving the encoding of `x`.

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

```
> mySource <- function(x, encoding = "UTF-8")
+     XMLSource(x, function(tree) XML::xmlChildren(XML::xmlRoot(tree)), myXMLReader, encoding)
```

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", "/document/writer"),
+                 Content = list("node", "/document/description"),
+                 DateTimeStamp = list("function",
+                                      function(x) as.POSIXlt(Sys.time(), tz = "GMT")),
+                 Description = list("attribute", "/document/@short"),
+                 Heading = list("node", "/document/caption"),
+                 ID = list("function", function(x) tempfile()),
+                 Origin = list("unevaluated", "My private bibliography"),
+                 Type = list("node", "/document/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 (of course 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 meta datum of the text document as specified by the named list entry. Valid names include `Content` to access the document's content, any valid attribute name (`Author`, `DateTimeStamp`, `Description`, `Heading`, `ID`, and `Origin` in above example specification), and characters (`Type` in above specification) which are mapped to so-called `LocalMetaData` 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 expression `spec` extracts information out of an XML node (as seen for `Author`, `Content`, `Heading`, and `Type` in our example specification).

`type = "attribute", spec = "XPathExpression"` the XPath expression `spec` extracts information from an attribute of an XML node (like `Description` in our example).

`type = "function", spec = function(tree) ...` The function `spec` is called, passing over a tree representation (as delivered by `xmlInternalTreeParse()` from package **XML**) of the read in XML document as first argument (as seen for `DateTimeStamp` and `ID`). As you notice in our example nobody forces us to actually use the passed over tree, 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 <- Corpus(mySource(custom.xml))
```

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

```
> corpus[[1]]

A story about an invisible man.

> meta(corpus[[1]])

Available meta data pairs are:
  Author       : Ano Nymous
  DateTimeStamp: 2014-01-11 15:39:30
  Description  : invisible man
  Heading      : The Invisible Man
  ID           : /tmp/Rtmpgv7tPt/file53531352f2f8
  Language     : en
  Origin       : My private bibliography
User-defined local meta data pairs are:
$Type
[1] "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`.