An Introduction to **RSNL**

Tony Fader, Gary King, Daniel Pemstein, and Kevin Quinn

December 16, 2008

1 Introduction

RSNL (R Statistics with Natural Language) provides R programmers with access to an arsenal of natural language processing (NLP) tools that they can use without leaving the R environment. **RSNL** builds upon the text-processing middle-ware layer provided by the **tm** package, and provides a uniform interface to a multitude of text processing and modeling tools. **RSNL** relies on S4 classes and generic functions to present programmers with a simple, uniform, and extensible interface and provides tools that allow the user to keep track of relationships between chunks of text—and decompositions and summaries of those bits of text—throughout the analysis process. While most publicly available NLP toolkits are designed to meet the needs of natural language research, **RSNL** is intended to faciliate NLP-based analyses by applied researchers, particularly those in the social and behavioral sciences.

This document provides a short, example-based, introduction to **RSNL** that demonstrates how to prepare a **tm**-managed text collection to interact with **RSNL**'s toolset, how to pre-process the text using a combination of **tm**-based and **RSNL**-provided tools, how to extend and adapt these tools, and how to apply NLP tools to the text collection to produce multiple "views" of a text collection, or corpus. As development on **RSNL** progresses, this document will also demonstrate how to use **RSNL** to better keep track of corpus structure and meta-data, and explain the range of clustering and classification methods, visualization tools, discourse modeling and testing techniques, and topic modeling methods included in the **RSNL** toolkit.

2 Working with Corpus Objects

RSNL relies on **tm** to handle text input-output, data management and storage. **tm** represents collections of documents using the `Corpus` data structure which can read text from a disk or other `Source` using either a pre-defined or custom `reader` function.[1] For this example, we will construct a `Corpus` from a selection of 475 short speeches given by legislators and bureaucrats during debates on legislation in the European Parliament. These data ship with **RSNL** as a series of XML documents. The following code reads the collection from disk:

```
> library(RSNL)
> readSpeeches <- FunctionGenerator(function(...) {
+     function(elem, load, language, id) {
+         tree <- xmlTreeParse(elem$content,
+             asText = TRUE)
```

[1] Please see `http://cran.r-project.org/web/packages/tm/index.html` or `http://www.jstatsoft.org/v25/i05` for detailed **tm** documentation and examples.

```
+         root <- xmlRoot(tree)
+         bill <- root[["BILL"]]
+         speaker <- root[["SPEAKER"]]
+         status <- speaker[["STATUS"]]
+         title <- paste(xmlValue(bill[["TITLE"]]),
+             "-", xmlValue(speaker[["NAME"]]),
+             sep = "")
+         dateTimeStamp <- as.POSIXct(xmlValue(bill[["DATE"]]),
+             tz = "CET")
+         content <- xmlValue(root[["TEXT"]])
+         id <- paste(xmlValue(bill[["CODE"]]),
+             xmlValue(root[["SPEAKER-NUMBER"]]),
+             sep = ".")
+         doc <- new("PlainTextDocument", .Data = content,
+             Cached = TRUE, URI = elem$uri,
+             Author = "European Parliament",
+             DateTimeStamp = dateTimeStamp,
+             Origin = "http://www.europarl.europa.eu",
+             Heading = title, Language = language,
+             ID = id)
+         meta(doc, "SPEAKER-NUMBER") <- xmlValue(root[["SPEAKER-NUMBER"]])
+         for (name in c("CODE", "ISSUEAREA",
+             "PASSED", "RCV")) meta(doc, name) <- xmlValue(bill[[name]])
+         for (name in c("COUNTRY", "GROUP")) meta(doc,
+             name) <- xmlValue(speaker[[name]])
+         for (name in c("ISPRESIDENT", "ISCOUNCIL",
+             "ISCOMMISSION", "ISOTHERBUREAUCRAT",
+             "ISRAPPORTEUR", "ISCOMMITTEEREP",
+             "ISAUTHOR", "ISONBEHALFOFGROUP")) meta(doc,
+             name) <- xmlValue(status[[name]])
+         doc
+     }
+ })
> debates.source <- system.file("samples/en/ep-debates",
+     package = "RSNL")
> tm.debates <- Corpus(DirSource(debates.source),
+     readerControl = list(reader = readSpeeches,
+         language = "EN"))
```

2.1 Corpora and Passing Semantics

Currently, `Corpus` objects may store their internal data either directly in memory, or in a simple database format. When using in-memory storage, `Corpus` objects use R's standard pass-by-value semantics. This means that whenever a `Corpus` is passed to a function or method the interpreter makes a copy of the object and any changes to the object made within the function are not reflected in the original object. Furthermore, if we were to make a simple copy of `tm.debates`

```
> tm.debates.copy <- tm.debates
```

`tm.debates.copy` and `tm.debates` would represent distinct collections of text and changes to one object would have no effect on the other. On the other hand, when a `Corpus` stores its data on disk, it uses pass-by-reference semantics; in this case `Corpus` objects are essentially handles to an underlying data store and multiple copies of the handle all refer to the same set of data. Under these circumstances modifications to `tm.debates.copy` would be reflected in subsequent calls to `tm.debates`.

2.2 `PCorpus` and Reference Objects

RSNL provides a wrapper class for `Corpus` objects, `PCorpus`, that unifies corpus semantics. `PCorpus` objects behave exactly like `Corpus` objects[2] except that they use pass-by-reference semantics regardless of their underlying storage model. Furthermore, they provide under-the-hood tools that facilitate the data-view model employed by **RSNL** which we describe in more detail below. Thus, the first step in any **RSNL** analysis is to wrap an existing `Corpus`:

```
> debates <- PCorpus(tm.debates)
```

`PCorpus` objects are an example of a non-standard type of S4 object used throughout **RSNL**: `RObject` or reference objects. These objects pass one or more of their internal slots by reference when one makes a copy or passes the object to a function. It is possible to force the interpreter to make a pure copy of a `RObject` using the `clone()` method:

```
> debates.copy <- clone(debates)
> debates.copy.ref <- debates.copy
> debates.copy[[1]] <- debates.copy[[2]]
> debates[[1]] == debates.copy[[1]]

[1] FALSE

> debates.copy.ref[[1]] == debates.copy[[1]]

[1] TRUE
```

3 Tokenization and View Construction

Natural language models typically rely on patterns of tokens within the data. Tokens are often individual words, but can, in principle, represent the output of any procedure that splits a single piece of text into individual chunks. In this example, we'll take the traditional approach to tokenization and attempt to represent, or view, each document in the corpus as a series of individual words. To do so, we need to clean up the text a bit—transform the text to all lowercase, remove punctuation, numbers (which we'll represent using a single token), and common words—and do the actual tokenization. In this example, we'll also employ a common technique known as stemming, which reduces similar words with different suffixes (e.g. run, runner, running) to common roots. Finally, we'll ignore especially short words when modeling the text.

The above-mentioned procedures all do varying degrees of violence to the original text. In English, converting the documents to lowercase will have little impact on our ability to refer back to and understand the content of the original documents while performing an exploratory analysis,

[2] At the current stage of development this not quite true: we have not implemented the `c()` method for `PCorpus` objects, nor have we tested their compatibility with **tm**'s lazy mapping facilities.

but other operations, such as transforming or eliminating certain symbols or strings, tokenizing, and stemming, can render the original text unreadable. **RSNL** takes advantage of an "object-view" model to help overcome this issue. When performing basic text-cleaning operations, such as converting the text to lower case, the analyst will often wish to employ **tm**'s various filtering and mapping to tools to modify the `Corpus` itself. But when performing more destructive operations the analyst can benefit by creating "views" of the underlying `Corpus`, or its constituent documents, that encapsulate both a reference to the original text and a policy for transforming the text in some way. Of course, there is a trade-off here: iteratively modifying the text in place requires less storage space than the object-view approach and may be necessary with large datasets; on the other hand, the object-view model makes it far easier for the analyst to refer back to the underlying data and work with multiple representations (views) of the data at once, and will generally use fewer resources than maintaining a copy of the corpus for each desired representation of the data.

3.1 Working with Tokenized Views

We'll start by constructing a simple tokenized view of the corpus. **RSNL** provides a method, `tokenize()` that can generate tokens from a variety of data types. At the most basic level, given a `character` object (or child type such as **tm**'s `PlainTextDocument`), `tokenize()` will return a vector of tokens:

```
> tokenize(debates[[1]])

 [1] "ÃĆÂăÃĆÂă"       "The"         "next"
 [4] "item"        "is"          "the"
 [7] "report"      "("           "A6"
[10] "-"           "0027"        "/"
[13] "2004"        ")"           "by"
[16] "Mrs"         "Corbey"      "on"
[19] "the"         "draft"       "European"
[22] "Parliament" "and"         "Council"
[25] "Directive"  "amending"    "Directive"
[28] "94"          "/"           "62"
[31] "/"           "EC"          "on"
[34] "packaging"  "and"         "packaging"
[37] "waste"       "."           "ÃĆÂă"
```

On the other hand, given a `PCorpus`, `tokenize()` will generate a view of that corpus or of one of its documents:

```
> (debates.tok <- tokenize(debates))

A tokenized corpus view with 183842 total tokens and 9035 unique tokens

> (debates.tok.1 <- tokenize(debates, index = 1))

A tokenized document view of A6-0027/2004.1 with 39 total tokens and 32 unique tokens
```

`debates.tok` and `debates.tok.1` are examples of `View` objects; specifically, `debates.tok` is a `TokenizedCorpusView` and `debates.tok.1` is a `TokenizedDocumentView`.[3] We can examine these

[3]Note that views are only defined in reference to `PCorpus` objects. Therefore, you can not create a view to a document not contained in a corpus.

objects with a variety of methods. For example, given these two views, we can look at the tokens within the first document in the corpus in one of two ways:

```
> tokens(debates.tok[[1]])

 [1] "ÃĆÂăÃĆÂă"       "The"        "next"
 [4] "item"       "is"         "the"
 [7] "report"     "("          "A6"
[10] "-"          "0027"       "/"
[13] "2004"       ")"          "by"
[16] "Mrs"        "Corbey"     "on"
[19] "the"        "draft"      "European"
[22] "Parliament" "and"        "Council"
[25] "Directive"  "amending"   "Directive"
[28] "94"         "/"          "62"
[31] "/"          "EC"         "on"
[34] "packaging"  "and"        "packaging"
[37] "waste"      "."          "ÃĆÂă"

> tokens(debates.tok.1)

 [1] "ÃĆÂăÃĆÂă"       "The"        "next"
 [4] "item"       "is"         "the"
 [7] "report"     "("          "A6"
[10] "-"          "0027"       "/"
[13] "2004"       ")"          "by"
[16] "Mrs"        "Corbey"     "on"
[19] "the"        "draft"      "European"
[22] "Parliament" "and"        "Council"
[25] "Directive"  "amending"   "Directive"
[28] "94"         "/"          "62"
[31] "/"          "EC"         "on"
[34] "packaging"  "and"        "packaging"
[37] "waste"      "."          "ÃĆÂă"
```

Furthermore, we can easily see the most common words in the corpus

```
> sort(freqTable(debates.tok), dec = T)[1:20]

  the     ,     .    of    to   and    in  that    is
10729  9386  6266  5711  5498  4825  3269  3045  2832
    a   for     I    on    be  this    we    it   are
 2775  2384  1873  1669  1603  1573  1374  1195  1188
 have    ÃĆÂă
 1121  1094
```

or generate a list of unique tokens:

```
> u <- unique(debates.tok)
```

By default, `tokenize()` uses a simple regular-expression based tokenizer to break up the text, but the package provides a number of pre-defined `Tokenizer` types and users are free to extend this base class to define their own. For example, to emulate the tokenizing done within **tm**'s `termFreq()` function we might define a custom `Tokenizer` like so:

```
> setClass("TmTokenizer", representation("Tokenizer",
+     fun = "function"))

[1] "TmTokenizer"

> TmTokenizer <- function() new("TmTokenizer",
+     fun = function(x) unlist(strsplit(gsub("[^[:alnum:]]+",
+         " ", x), " ", fixed = TRUE)))
> setMethod("tokenize", signature(object = "character",
+     tokenizer = "TmTokenizer", index = "missing"),
+     function(object, tokenizer, index) tokenizer@fun(object))

[1] "tokenize"
```

While our `TmTokenizer` relies on R's regular expression engine to identify tokens, **RSNL**'s `RegexTokenizer` type allows users to define arbitrary regex-based tokenizers that match strings using the—often substantially faster—Python regular expression engine. For example, we might construct a very simple tokenizer that splits strings solely on whitespace using the following definition (the second argument tells the tokenizer to match the spaces in between tokens, rather than tokens):[4]

```
> space.breaker <- RegexTokenizer("\\s+", matchToken = FALSE)
```

Note that all three tokenizers we've used thus far generate slightly different representations of the underlying text.

```
> tokenize(debates[[1]])

 [1] "ÃĆÂăÃĆÂă"       "The"        "next"
 [4] "item"       "is"         "the"
 [7] "report"     "("          "A6"
[10] "-"          "0027"       "/"
[13] "2004"       ")"          "by"
[16] "Mrs"        "Corbey"     "on"
[19] "the"        "draft"      "European"
[22] "Parliament" "and"        "Council"
[25] "Directive"  "amending"   "Directive"
[28] "94"         "/"          "62"
[31] "/"          "EC"         "on"
[34] "packaging"  "and"        "packaging"
[37] "waste"      "."          "ÃĆÂă"

> tokenize(debates[[1]], TmTokenizer())
```

[4] Note the double-escaping of character classes.

```
 [1] "ÃĆ"          "ÃĆ"          "The"
 [4] "next"       "item"       "is"
 [7] "the"        "report"     "A6"
[10] "0027"       "2004"       "by"
[13] "Mrs"        "Corbey"     "on"
[16] "the"        "draft"      "European"
[19] "Parliament" "and"        "Council"
[22] "Directive"  "amending"   "Directive"
[25] "94"         "62"         "EC"
[28] "on"         "packaging"  "and"
[31] "packaging"  "waste"      "ÃĆ"
```

```
> tokenize(debates[[1]], space.breaker)
```

```
 [1] "ÃĆÂăÃĆÂă"         "The"          "next"
 [4] "item"         "is"           "the"
 [7] "report"       "("            "A6-0027/2004"
[10] ")"            "by"           "Mrs"
[13] "Corbey"       "on"           "the"
[16] "draft"        "European"     "Parliament"
[19] "and"          "Council"      "Directive"
[22] "amending"     "Directive"    "94/62/EC"
[25] "on"           "packaging"    "and"
[28] "packaging"    "waste."       "ÃĆÂă"
```

In what follows, we'll use the `TokenizedCorpusView` named `debates.tok` that we created with the default tokenizer.

3.2 Filters and Transforms

As we mentioned at the beginning of Section 3, we're going to need to transform our text in a number of ways to make it amenable to analysis. First of all, because it has little impact on the readability of the text, we'll start out by using **tm**'s `tmpMap()` function to convert our text to lower-case.

```
> debates.tok

A tokenized corpus view with 183842 total tokens and 9035 unique tokens

> tmMap(debates, tmTolower)

A text document collection with 475 text documents

> debates.tok

A tokenized corpus view with 183842 total tokens and 8299 unique tokens
```

This sequence of operations illustrates one nice perk of **RSNL**'s object-view model: auto-updating. As we previously noted, `View` objects maintain references to the objects that they view. One advantage of this approach is the ability to quickly examine an original document in light of something one finds in a view; another is that views can keep track of when anything changes in the underlying data structure and update to reflect modifications. This auto-updating ability saves users from the tedious task of redefining views after making changes to a corpus.

3.3 Token Transforms and Filters

Now let's perform some more destructive manipulations, using the view model to preserve the corpus:

```
> debates.tok <- filterTokens(debates.tok)
> debates.tok <- filterTokens(debates.tok,
+     PunctTokenFilter())
> debates.tok <- filterTokens(debates.tok,
+     FunctionalTokenFilter(function(x) nchar(x) >
+         3))
> debates.tok <- transformTokens(debates.tok,
+     RegexTokenTransform("^[0-9]+$", "NUMBER"))
> debates.tok <- stem(debates.tok)
> debates.tok <- filterTokens(debates.tok,
+     TokenDocFreqFilter(debates.tok, 0.05,
+         0.95))
> debates.tok

A tokenized corpus view with 42251 total tokens and 471 unique tokens

> sort(freqTable(debates.tok), dec = T)[1:20]

  european     propos    commiss     NUMBER
       847        713        674        544
  programm     direct      amend     presid
       521        497        463        447
        ÃćÂã     report parliament   committe
       437        359        355        314
   support       time    protect      europ
       306        276        268        264
     peopl     energi     safeti      thank
       260        259        252        252
```

These operations take advantage of the `filterTokens()`, `transformTokens()`, and `stem()` methods to filter out common English words,[5] remove all-punctuation tokens, filter tokens shorter than three characters in length, convert all-numeric tokens to the catch-all token "NUMBER", reduce the tokens to common roots, and filter out tokens that occur in less than five or more than 95 percent of the documents. Note that the views perform lazy updates and we perform virtually no computation until requesting a printed representation of the view[6] in the next to last line of the code snippet. As you can see, the resulting view provides a representation of the corpus with far fewer tokens than the original tokenzied view. Visualizing a document from the view side-by-side with its original form demonstrates the massive difference between the two representations:

```
> tokens(debates.tok[[1]])
```

[5]The default behavior of `filterTokens()` is to remove stop-words. To see a list of stopwords, type `stopwords("english")` at the R prompt.

[6]Note that printing a view to the screen is actually quite computationally costly because it calls `freqTable`, `unique`, and `documentTokenMatrix` under the hood.

```
 [1] "ÃćÂă"         "item"       "report"
 [4] "NUMBER"     "NUMBER"     "draft"
 [7] "european"   "parliament" "council"
[10] "direct"     "amend"      "direct"
[13] "packag"     "packag"
```

```
> document(debates.tok[[1]])
```

```
[1] ÃćÂăÃćÂă the next item is the report ( a6-0027/2004 ) by mrs corbey on the draft european p
```

Before moving on, note that `transformTokens()` and its brethren are, like `tokenize()`, capable of operating on inputs ranging from basic character strings—in which case they return a vector of tokens, appropriately filtered or transformed—to `PCorpus` and `TokenizedView` objects—in which case they return `TokenizedView` objects of the appropriate type.

```
> stem(debates[[1]])
```

```
 [1] "ÃćÂă"         "the"        "next"
 [4] "item"       "is"         "the"
 [7] "report"     "("          "a6"
[10] "-"          "0027"       "/"
[13] "2004"       ")"          "by"
[16] "mrs"        "corbey"     "on"
[19] "the"        "draft"      "european"
[22] "parliament" "and"        "council"
[25] "direct"     "amend"      "direct"
[28] "94"         "/"          "62"
[31] "/"          "ec"         "on"
[34] "packag"     "and"        "packag"
[37] "wast"       "."          "Ãć"
```

```
> stem(debates)
```

```
A tokenized corpus view with 183842 total tokens and 5185 unique tokens
```

```
> stem(debates, tokenizer = PunktWordTokenizer(),
+     index = 1)
```

```
A tokenized document view of A6-0027/2004.1 with 30 total tokens and 24 unique tokens
```

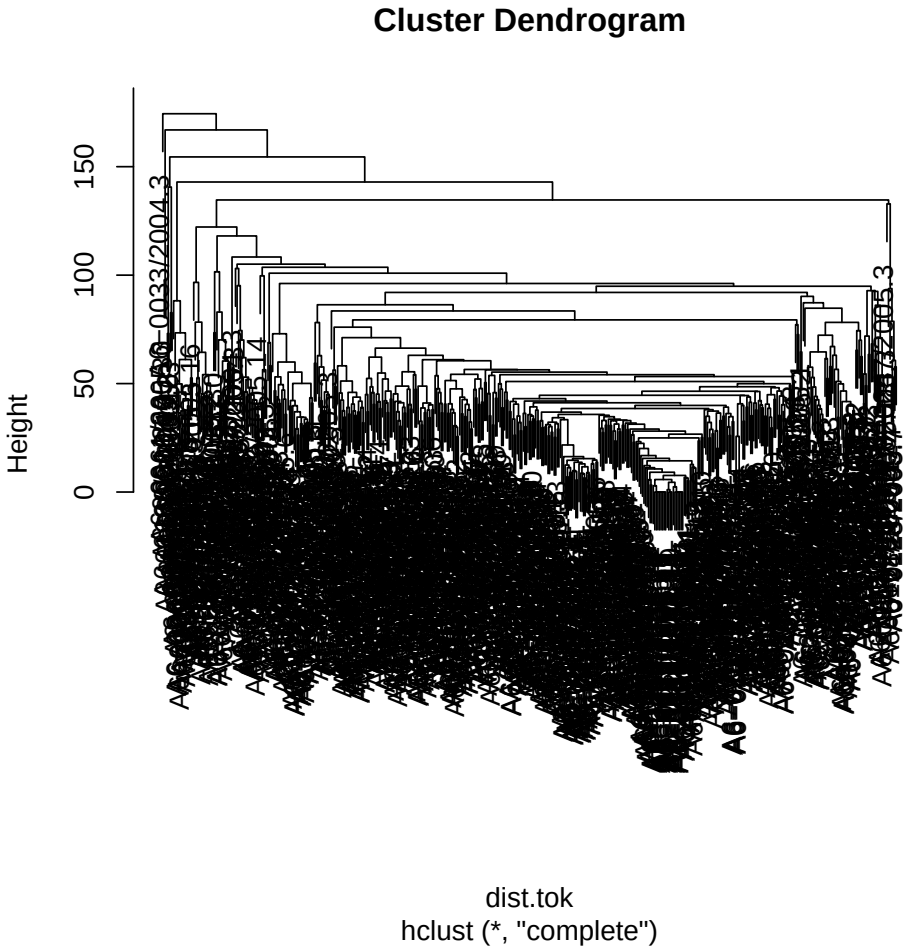
Our flexible transform and filter object model also makes it easy to construct non-standard views of the data. In the above example we use a `RegexTokenTransform` object to transform numbers to a single token and the flexible `FunctionalTokenFilter` type to eliminate short tokens. These objects are accompanied by a variety of other `TokenTransform` and `TokenFilter` object types, and the user may readily extend these base classes as needed.[7] As another example, while we might use the tokenized representation in `debates.tok` as the basis for a unigram-focused bag-of-words analysis of the data, we might also want to represent the document in terms of pairs consecutive words. We can do this using a `FunctionalTokenTransform` object:

[7] Most common extensions can be performed with appropriate sub-classing of the `FunctionalTokenTransform` and `FunctionalTokenFilter` types.

```
> bigTrans <- FunctionalTokenTransform(function(x) {
+     index <- lapply(1:length(x), function(x) seq(x,
+         x + 1))
+     sapply(index, function(y) paste(x[y],
+         collapse = " "))
+ })
> (debates.bigram <- transformTokens(debates.tok,
+     bigTrans))

A tokenized corpus view with 42251 total tokens and 26914 unique tokens

> tokens(debates.bigram[[1]])

 [1] "ÃćÂă item"             "item report"
 [3] "report NUMBER"        "NUMBER NUMBER"
 [5] "NUMBER draft"         "draft european"
 [7] "european parliament" "parliament council"
 [9] "council direct"       "direct amend"
[11] "amend direct"         "direct packag"
[13] "packag packag"        "packag NA"

> sort(freqTable(debates.bigram), dec = T)[1:20]

          ÃćÂă presid       NUMBER NUMBER
                 261                241
     european union      ladi gentlemen
                 189                133
     commiss propos        drive licenc
                 128                105
european parliament        madam presid
                  89                 86
  presid commission            ÃćÂă madam
                  85                 76
      public health         food safeti
                  65                 62
        presid ladi  parliament council
                  61                 58
      energi effici    thank rapporteur
                  55                 55
      tran european    committe environ
                  54                 49
        health food     commission ladi
                  49                 48
```

At this point we are ready to do some analysis. As a simple example, we might simply wish to visualize how similar our speeches are to one another. We can use an unsupervised clustering technique to visualize the data in this way. To do this we generate a document-token matrix from our tokenized view (as a `TermDocMatrix` object), calculate the euclidean distances between the rows of the matrix, cluster, and plot the result:

```
> dist.tok <- dist(documentTokenMatrix(debates.tok,
+     weightTfIdf))
> clust.tok <- hclust(dist.tok)
> plot(clust.tok)
```



3.4 Document Filters

So far we've restricted our transforms and filters to operations on individual tokens but we can also filter out particular documents from a `CorpusView`.[8] For example, the 475 speeches in our dataset come from a smaller set of debates on particular pieces of legislation. During the debate, the rapporteur—the member of the European Parliament responsible for guiding the legislation through parliament—almost always gives a short informational speech describing the bill. We can take advantage of the meta-data attached to our corpus to identify the rapporteur speeches in the dataset. Furthermore, we can use this information to generate a filtered view of the data and try our simple visualization technique again, using only the rapporteurs' speeches, in hopes of generating a representation of the data that can tell us something about the similarity of the topics under debate.

[8] As a rule, we leave transformation of entire documents to **tm**.

```
> rap <- sapply(debates, meta, tag = "ISRAPPORTEUR")
> debates.rap <- filterDocuments(debates.tok,
+     FunctionalDocumentFilter(function(x) rap ==
+         1))
> dist.rap <- dist(documentTokenMatrix(debates.rap,
+     weightTfIdf))
> clust.rap <- hclust(dist.rap)
> plot(clust.rap)
```