An Introduction to **RSNL**

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

December 22, 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** relies on S4 classes and generic functions to present programmers with a simple, uniform, and extensible interface to natural language processing in R. Specifically, it provides a suite of methods for common natural language tasks (e.g. tokenization, stemming, part-of-speech tagging, and topic modeling) and a collection of extensible S4 objects (e.g. tokenizers, stemmers, taggers, and topic model objects) to use when carrying out these operations. Furthermore, **RSNL**'s toolset allows the user to keep track of relationships between chunks of text—and decompositions and summaries of those bits of text—throughout the analysis process, using an *object-view* model to provide multiple, concurrent, representations of an underlying collection of texts. This *object-view* approach facilitates exploratory data analysis while simultaneously providing a powerful sub-structure upon which to build high-level NLP analysis and visualization software. Finally, while most publicly available NLP toolkits are designed to meet the needs of natural language research, **RSNL** is intended to facilitate NLP-based analyses by applied researchers, particularly those in the social and behavioral sciences, and is tuned to their problems and their datasets.

This document provides a short, example-based, introduction to **RSNL** that demonstrates how to prepare a 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 Preparing a Dataset for Analysis

RSNL builds upon the text-processing middle-ware layer provided by the **tm** package to provide a multitude of text processing and modeling tools. **RSNL** relies on **tm**[1] to handle text input-output, data management and storage. **tm** provides S4 types—the `TextDocument` and `Corpus` classes—that allow users to represent, examine, and manipulate individual chunks of text (typically individual documents) and large corpora of related documents. Additionally, these basic data types sport fields for maintaining detailed metadata about the underlying text. Furthermore, **tm** includes a wide array of I/O tools, including readers and writers for various document formats, back-end database support, and functions that allow users to apply transformations and filters to

an individual document or corpus with relative ease. Finally, **tm** provides a simple object, the `TermDocMatrix`, for representing corpora in terms of document-level word frequencies.

RSNL works intimately with **tm** to prepare a text collection for higher level analysis; **RSNL** users will typically adopt the following work-flow when preparing a set of documents for analysis:

1. The user will use **tm**'s text I/O tools to read a collection of documents into R and create a `Corpus` object.

2. The user will construct an **RSNL** `PCorpus` object from the newly minted `Corpus` to best take advantage of **RSNL**'s collection of tools.

3. The user will transform and filter the data as necessary to prepare it for higher-level analysis; during this step, one will make use of **tm**'s mapping and filtering functions to perform tasks that destructively modify the text collection, while relying on **RSNL**'s collection of tokenizers, stemmers, transforms, and filters to create a variety of non-destructive *views* of the underlying dataset.

The rest of this section explains this process in detail.

2.1 Working with Corpus Objects

Throughout this document we will work with an example dataset containing a selection of 475 short speeches given by legislators and bureaucrats during debates on legislation in the European Parliament (EP). The speeches are a non-random sample from a larger dataset of EP debates [2]. The dataset has a hierarchical structure and each speaker delivered her speech as part of a debate dedicated to a particular piece of legislation. Specifically, each speech fits into one of the first 29 debates on Codecision[1] legislation conducted by the Parliament in its 6th (current) term. These data ship with **RSNL** as a series of XML documents which contain both the raw text of the speeches and a variety of meta-data.

Figure 1 displays one of the XML documents in the example dataset. Each file is composed of four main sections. The first section contains the bill's title, unique EP code, date of debate, position on the day's debate schedule, the last name of the member of the EP (MEP) responsible for reporting on the bill, a number indicating one of eight policy issue areas assigned to the bill by EU bureaucrats, a dummy variable indicating whether or not the reading of the bill survived a vote on the legislation as a whole, and a binary indicator of whether or not that final vote was conducted by public roll call. The second section provides speaker-specific information including the speaker's name (or title, if the speaker is acting in a purely institutional role), the country and parliamentary party group the speaker represents (if applicable), and a series of dummy variables describing the speaker's institutional role vis-a-vis the given piece of legislation. Finally, the third section of the xml file contains a single indicator identifying the chronological position of the speech in the debate while the fourth section sports the raw text of the speech itself.[2]

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. Therefore, our first step in any analysis of the EP speeches will be to construct a **tm**-style reader and use it to build a `Corpus` object from our XML files on disk.

[1]The European Parliament considers legislation under a variety of legislative procedures. Codecision is generally considered the most important of these procedures although the particulars of EP protocol are of little consequence to the current example.

[2]The funny character sequences as the beginning and end of the text section are common in this dataset. We will remove these in the course of preparing the text for analysis.

```
<?xml version="1.0"?>
<DEBATE-SPEECH>
  <!-- 1. Bill-specific meta-data -->
  <BILL>
    <TITLE> Packaging and packaging waste</TITLE>
    <CODE>A6-0027/2004</CODE>
    <DATE>2004-11-17</DATE>
    <ITEM>5</ITEM>
    <RAPPORTEUR>CORBEY</RAPPORTEUR>
    <ISSUEAREA>3</ISSUEAREA>
    <PASSED>1</PASSED>
    <RCV>0</RCV>
  </BILL>

  <!-- 2. Speaker-specific meta-data -->
  <SPEAKER>
    <NAME>President</NAME>
    <COUNTRY></COUNTRY>
    <GROUP></GROUP>
    <STATUS>
      <ISPRESIDENT>1</ISPRESIDENT>
      <ISCOUNCIL>0</ISCOUNCIL>
      <ISCOMMISSION>0</ISCOMMISSION>
      <ISOTHERBUREAUCRAT>0</ISOTHERBUREAUCRAT>
      <ISRAPPORTEUR>0</ISRAPPORTEUR>
      <ISCOMMITTEEREP>0</ISCOMMITTEEREP>
      <ISAUTHOR>0</ISAUTHOR>
      <ISONBEHALFOFGROUP>0</ISONBEHALFOFGROUP>
    </STATUS>
  </SPEAKER>

  <!-- 3. Speaker ordering -->
  <SPEAKER-NUMBER>1</SPEAKER-NUMBER>

  <!-- 4. The text -->
  <TEXT>
    ÃĆÂãÃĆÂã The next item is the report ( A6-0027/2004 ) by Mrs Corbey
    on the draft European Parliament and Council Directive amending
    Directive 94/62/EC on packaging and packaging waste. ÃĆÂã
  </TEXT>
</DEBATE-SPEECH>
```

Figure 1: An example EP speech in XML format.

```
> library(RSNL)
> # Define a custom reader, see tm docs for details
> readSpeeches <- FunctionGenerator(function(...) {
+   function (elem, load, language, id) {
+     # 1. Get the xml nodes organized
+     tree <- xmlTreeParse(elem$content, asText = TRUE)
+     root <- xmlRoot(tree)
+     bill <- root[["BILL"]]          # bill-specific data
+     speaker <- root[["SPEAKER"]]    # speaker-specific data
+     status <- speaker[["STATUS"]]   # institutional context dummies
+
+     # 2. Create the default meta-data fields
+     title <- paste(xmlValue(bill[["TITLE"]]), "-",
+                    xmlValue(speaker[["NAME"]]), sep="")
+
+     dateTimeStamp <- as.POSIXct(xmlValue(bill[["DATE"]]), tz="CET")
+
+     id <- paste(xmlValue(bill[["CODE"]]),
+                 xmlValue(root[["SPEAKER-NUMBER"]]), sep=".")
+
+     content <- xmlValue(root[["TEXT"]])
+
+     # 3. Construct the document
+     doc<-new("PlainTextDocument", .Data = content, Cached = TRUE, URI =
+         elem$uri, Author = "European Parliament",
+         DateTimeStamp = dateTimeStamp,
+         Origin = "http://www.europarl.europa.eu", Heading = title,
+         Language = language, ID= id)
+
+     # 4. Add some custom metadata
+     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
+   }
+ })
```

The `readSpeeches()` function in the code snippet above uses the **xml** package[3] to convert a

single EP speech in XML format into a `PlainTextDocument` suitable for inclusion in a `Corpus`. The first chunk of the function sets up convenience handles to the XML nodes representing the bill-specific meta-data, speaker-specific information, and the dummy variables related to the institutional responsibilities of the speaker. Section two extracts a variety of specific fields from the XML document and uses them to fill in the `PlainTextDocument`'s default fields; specifically, it generates a title by concatenating the bill's title to the speaker's name, converts the date of the debate into a timestamp, generates a document id by concatenating the bill code to the speaker's chronological position in the debate, and extracts the content of the speech itself. Next, section three constructs the `PlainTextDocument` object using the variables extracted in the first two portions of the function. Finally, the fourth and last chunk of code appends a variety of meta-data to the `PlainTextDocument`, using **tm**'s `meta()` method. With this reader in hand, we can create a connection to the EP dataset included with **RSNL** and pass the connection and reader to the `Corpus` constructor provided by **tm**:

```
> debates.source <- system.file("samples/en/ep-debates", package="RSNL")
> tm.debates <- Corpus(DirSource(debates.source), readerControl =
+                        list(reader = readSpeeches, language="EN"))
```

2.1.1 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`.

This variance in `Corpus` passing semantics is simply a matter of practicality. When a `Corpus` maintains its data in memory it is constrained by R's default behavior to make copies of the text it represents whenever copied or passed to a function. On the other hand, a `Corpus` object only needs to copy its database handle when storing its text on disk; under these circumstances the **tm** developers are able to conserve processor cycles and disk space by leaving the data in one place while allowing the user to manipulate references to the underlying data in R itself. Yet, while logical, these implementation-dependent differences in `Corpus` passing semantics are potentially confusing to end users and, in the case of in-memory storage, inefficient.

2.1.2 `PCorpus` and Reference Objects

RSNL provides a wrapper class for `Corpus` objects, `PCorpus`, that unifies corpus semantics. `PCorpus` objects behave exactly like `Corpus` objects[3] except that they use *pass-by-reference* semantics regardless of their underlying storage model. Furthermore, they provide under-the-hood tools that

[3] 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.

facilitate the data-view model employed by **RSNL** which we describe in more detail below. Thus, the second step in any **RSNL** analysis is to convert a corpus represented by **tm**'s `Corpus` type into a `PCorpus`. Doing this is exceedingly simple; one need only pass an existing `Corpus` to the `PCorpus` constructor:

```
> 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)              # 1
> debates.copy.ref <- debates.copy            # 2
> debates.copy[[1]] <- debates.copy[[2]]      # 3
> debates[[1]] == debates.copy[[1]]           # 4

[1] FALSE

> debates.copy.ref[[1]] == debates.copy[[1]]  # 5

[1] TRUE
```

It is worth walking through the above example, step by step, to make the distinction between a reference and a pure copy absolutely clear. In step one we take the `PCorpus debates`, which stores its text data in memory, and make a pure copy of it called `debates.copy`. At this point we have two full copies of the corpus in memory. In step two we make a reference to `debates.copy` called `debates.copy.ref`. This step does not make an additional copy of the underlying corpus and `debates.copy` and `debates.copy.ref` behave simply as two different names for the same underlying dataset. In step three we take advantage of the subset operator for `PCorpus` objects, which allows us to access a single document within a corpus, to overwrite the first document in `debates.copy` with the second document in the same corpus. Steps four and five demonstrate how this modification is reflected in the three `PCorpus` objects and highlight the fact that `debates` and `debates.copy` reference distinct underlying datasets while `debates.copy` and `debates.copy.ref` reference a single, shared, corpus.

2.2 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 (e.g. run). 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,

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.

2.2.1 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 (we will take care of those funny first and last tokens when we get to filtering):

```
> 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:

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

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

In some cases, one might wish to obtain a view of a single document within a corpus. In this case, a user may employ the `index` argument to `tokenize()` to select an individual document from within the corpus. For example, this code creates a view of the first document in `debates`:

```
> (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, both of which are subtypes of the TokenizedView, and more generically, View virtual classes. All View objects provide a representation of a given PCorpus object. Each view maintains a *reference* to a PCorpus and encapsulates a policy for representing that corpus. For example, a TokenizedCorpusView of debates contains a reference to debates and information about the Tokenizer object[4] used to break the text in debates into individual tokens. Similarly, a TokenizedDocumentView references a single document within a PCorpus while maintaining a policy for representing that document as a sequence of tokens.[5] Furthermore, Views use a lazy approach and perform no actual computation until absolutely necessary. This means that, when you use tokenize() to create a view of a given corpus, the method performs no actual tokenization, but rather constructs a View that is committed to representing the corpus in a particular (tokenized) way. The tokenization only occurs when the user invokes further methods on the TokenizedView, as we discuss below.

We can examine our TokenizedViews with a variety of methods. For example, given the two views we just constructed, 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"
```

[4] We describe Tokenizer objects in more detail below.

[5] 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.

```
[34] "packaging"  "and"        "packaging"
[37] "waste"      "."          "ÃĆÂã"
```

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    ÃĆÂã
 1121  1094
```

or generate a list of unique tokens:

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

Each of these operations requires the view to invoke its policy—the `Tokenizer` it encapsulates—on the `PCorpus` it refers to. The first code snippet requires only that the view tokenize the first document in the corpus, but the latter two examples require the view to tokenize the entire corpus. If you perform these actions in order you will notice that the interpreter spends substantially more time generating the frequency table than it does generating the unique tokens. This is because the view stores the results of previous computations for later use and need not re-tokenize the corpus when generating the list of unique terms.[6]

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

[1] "TmTokenizer"

> TmTokenizer <- function ()                                          # 2
+   new("TmTokenizer", fun = function (x)
+     unlist(strsplit(gsub("[^[:alnum:]]+", " ", x    ), " ", fixed = TRUE)))
> setMethod("tokenize", signature(object="character",                 # 3
+                                 tokenizer="TmTokenizer",
+                                 index="missing"),
+   function (object, tokenizer, index) tokenizer@fun(object))

[1] "tokenize"
```

In the first step (1) we define a new S4 class, `TmTokenizer`, that extends the base `Tokenizer` class and includes a slot for a tokenizing function. Next (2) we create a constructor function for `TmTokenizer`s that takes no arguments and returns a `TmTokenizer` object with a set tokenizing function that uses R's regular expression tools to remove all of the non-alpha-numeric characters from a character string and converts the string into tokens by splitting the string

[6] One can adjust a view's storage behavior using the `keepComputed()` and `keepDocumentViews()` methods.

at spaces. Finally, (3) we implement a specialization of the `tokenize()` method for the signature `signature(object="character", tokenizer="TmTokenizer", index="missing")` so that, when one passes a `character` object and a `TmTokenizer` to `tokenize()`, it takes the given string and applies the set tokenizing function to that string, returning a sequence of tokens. In general, when implementing a new `Tokenizer` called, say, `MyTokenizer`, one need only implement the specialization of `tokenize()` for the signature `signature(object="character", tokenizer="MyTokenizer", index="missing")`; **RSNL** provides the rest of the method specializations necessary to make the `Tokenizer` work with `PCorpus` and `View` objects.

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):[7]

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

Note that all three tokenizers we've used thus far generate slightly different representations of the underlying text. The `tokenize()` method takes a `Tokenizer` as its second argument. Thus, the three calls below call `tokenize()` with the default tokenizer, a `TmTokenizer`, and using our custom `RegexTokenizer`, respectively:

```
> 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())

 [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"
```

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

```
[25] "94"          "62"          "EC"
[28] "on"          "packaging"   "and"
[31] "packaging"   "waste"       "ÃĆ"
```

```
> tokenize(debates[[1]], tokenizer=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.

2.3 Filters and Transforms

As we mentioned at the beginning of section 2.2, 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, something that can save many key-strokes—or executions of the `source()` function—when one performs an exploratory analysis on a dataset.

2.3.1 Token Transforms and Filters

As demonstrated briefly above, on can use **tm**'s `tmMap()` method to modify the text contained within a `PCorpus` object. Nonetheless, this approach modifies the corpus directly and, as we previously argued, it may often be useful to work with numerous modified representations of the text without rendering the corpus unreadable. Therefore, we'll use **RSNL**'s filtering and transformation methods to take care of the more destructive tasks we need to perform to get the dataset ready for analysis and generate filtered and transformed `TokenizedCorpusView`s of the underlying corpus to get these jobs done:

```
> debates.tok <- filterTokens(debates.tok)              #1
> debates.tok <- filterTokens(debates.tok,              #2
+   PunctTokenFilter())
> debates.tok <- filterTokens(debates.tok,              #3
+   FunctionalTokenFilter(function (x) nchar(x) > 3))
> debates.tok <- transformTokens(debates.tok,           #4
+   RegexTokenTransform("^[0-9]+$", "NUMBER"))
> debates.tok <- stem(debates.tok)                      #5
> debates.tok <- filterTokens(debates.tok,              #6
+   RegexTokenFilter("^[^a-zA-Z]+$", negate=TRUE))
> debates.tmp <- debates.tok  # Save for bigrams example
> debates.tok <- filterTokens(debates.tok,              #7
+   TokenDocFreqFilter(debates.tok, .05, .95))
> debates.tok

A tokenized corpus view with 41725 total tokens and 468 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
    report parliament   committe    support
       359        355        314        306
      time    protect      europ      peopl
       276        268        264        260
    energi     safeti      thank particular
       259        252        252        249
```

These operations take advantage of **RSNL**'s `filterTokens()`, `transformTokens()`, and `stem()` methods to (1) filter out common English words,[8] (2) remove all-punctuation tokens, (3) filter tokens shorter than three characters in length, (4) convert all-numeric tokens to the catch-all token "NUMBER", (5) reduce the tokens to common roots, (6) remove all remaining tokens—such as the garbled symbols leading off most of the speeches in the dataset—comprised completely of non-alphabetic characters, and (7) filter out tokens that occur in less than five or more than 95 percent of the documents.

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

Some of these transformations and filter operations are less self-explanatory than others. Most notably, take step (3), which makes use of the flexible `FunctionalTokenFilter` type to eliminate especially short tokens for the view. The `FunctionalTokenFilter` constructor takes a function as its first argument. This function provides the filter with a policy for manipulating a sequence of tokens; specifically it is expected to take a single argument—a vector of tokens—and return a `logical` vector of the same length, indicating which tokens to retain in the filtered view. In (3), the function returns a vector of `logical` values, with only those slots corresponding to tokens with more than three characters set to `TRUE`.

Note that, because the views use lazy updates, we perform virtually no computation until requesting a printed representation of the view[9] 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 tokenized view. Visualizing a document from the view side-by-side with a simple tokenized version of the original demonstrates the massive difference between the two representations:

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

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

> 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"
```

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"
```

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

```
 [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.[10] 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 of consecutive words. We can do this using a `FunctionalTokenTransform` object:

```
> bigTrans <- FunctionalTokenTransform(                          # 1
+   function (x) {                                               # 1a
+     index <- lapply(1:length(x), function (x) seq(x, x+1))
+     sapply(index, function (y) paste(x[y], collapse=" "))
+   })
> (debates.bigram <- transformTokens(debates.tmp, bigTrans))     # 2

A tokenized corpus view with 62146 total tokens and 47149 unique tokens

> debates.bigram <- filterTokens(debates.bigram,                 # 3
+   TokenDocFreqFilter(debates.bigram, .01, .95))
> tokens(debates.bigram[[1]])

[1] "item report"         "report NUMBER"
[3] "NUMBER NUMBER"       "european parliament"
[5] "parliament council"  "council direct"
[7] "packag wast"

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

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

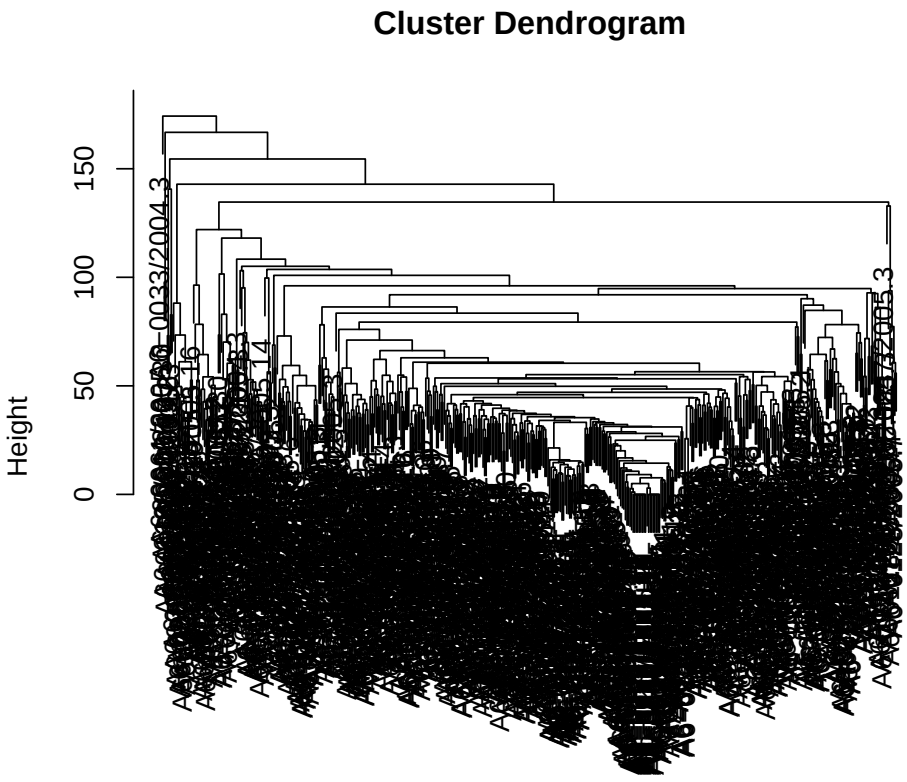
```
      NUMBER NUMBER      european union
                234                 188
     ladi gentlemen      commiss propos
                133                 120
       drive licenc european parliament
                103                  88
       madam presid   presid commission
                 85                  84
      public health         food safeti
                 65                  62
      natura NUMBER         presid ladi
                 60                  57
      energi effici       tran european
                 54                  54
 parliament council    thank rapporteur
                 53                  51
   committe environ      environ public
                 49                  48
       health food    financi perspect
                 47                  45
```

Here, we (1) construct a custom `TokenTransform` object of the type `FunctionalTokenTransform` to perform the transformation and then (2) apply it to the `TokenizedCorpusView` with the `transformTokens()` method. Finally, (3) we filter out especially common and uncommon bigrams (bigrams are sparser than unigrams, so we relax our floor somewhat), as we did with the unigram data, and visualize aspects of the resulting view. `FunctionalTokenTransform` is a versatile class that encapsulates an arbitrary function representing a given token transformation rule in an object that behaves in a manner expected by the `transformTokens()` method.[11] The `FunctionalTokenTransform` constructor takes a single-argument function as its first argument and this function should take a vector of tokens and return a transformed token vector. In the case of this example, our function (1a) iterates through every pair of tokens in the passed-in vector and returns a vector of concatenated pairs.

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)
```

[11] Remember, the `FunctionalTokenFilter` type serves an analogous role in token filtering with `filterTokens()`.



In the preceding chunk of code we use **RSNL**'s `documentTokenMarix()` method to extract an matrix of weighted—using the `weightTfIdf` method provided by **tm**—document-word frequencies (rows are documents while columns represent tokens) and invoke the `dist()` function in the **stats** package to generate a distance matrix suitable for hierarchical clustering methods. `documentTokenMatrix()` returns a `TermDocMatrix` object as defined by the **tm** package.

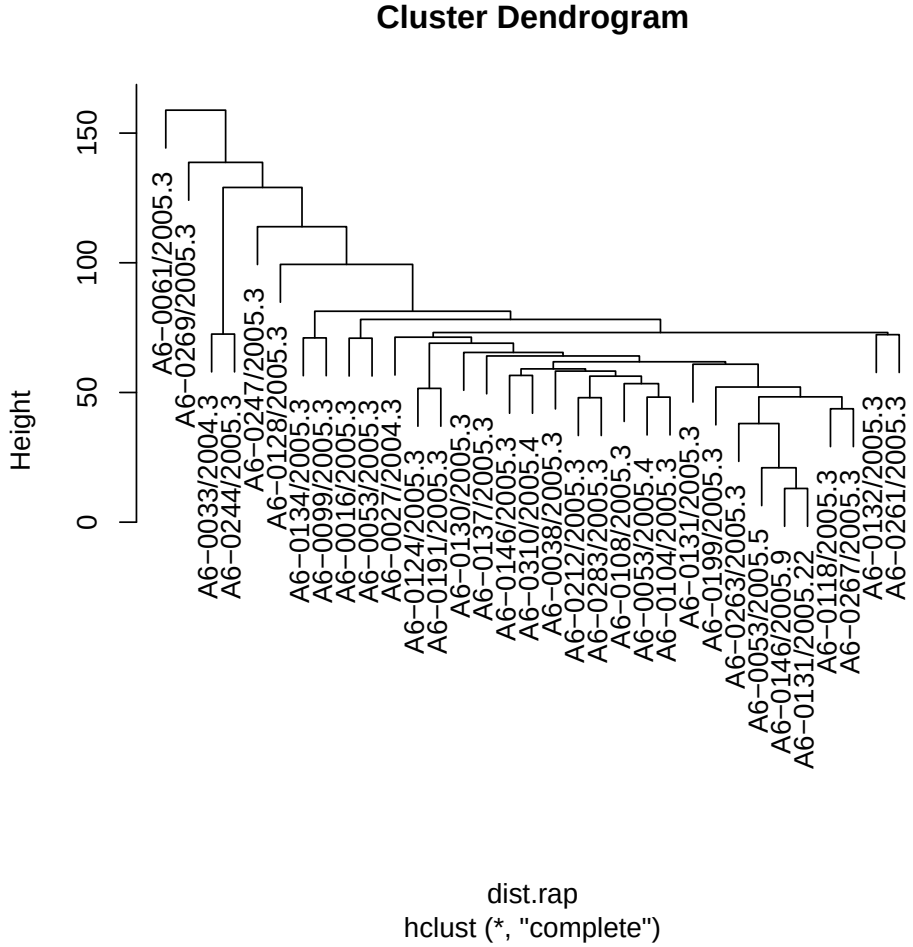
2.3.2 Document Filters

The graph we just generated is, for lack of a better word, ugly. But we can take advantage of the hierarchical nature of the dataset, and the meta-data encoded in our `PCorpus` object, to generate a more readable plot. 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` using **RSNL**'s `filterDocuments()` method.[12] 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

[12] See **tm**'s `tmFilter()` method for an analogous that directly modifies the corpus.

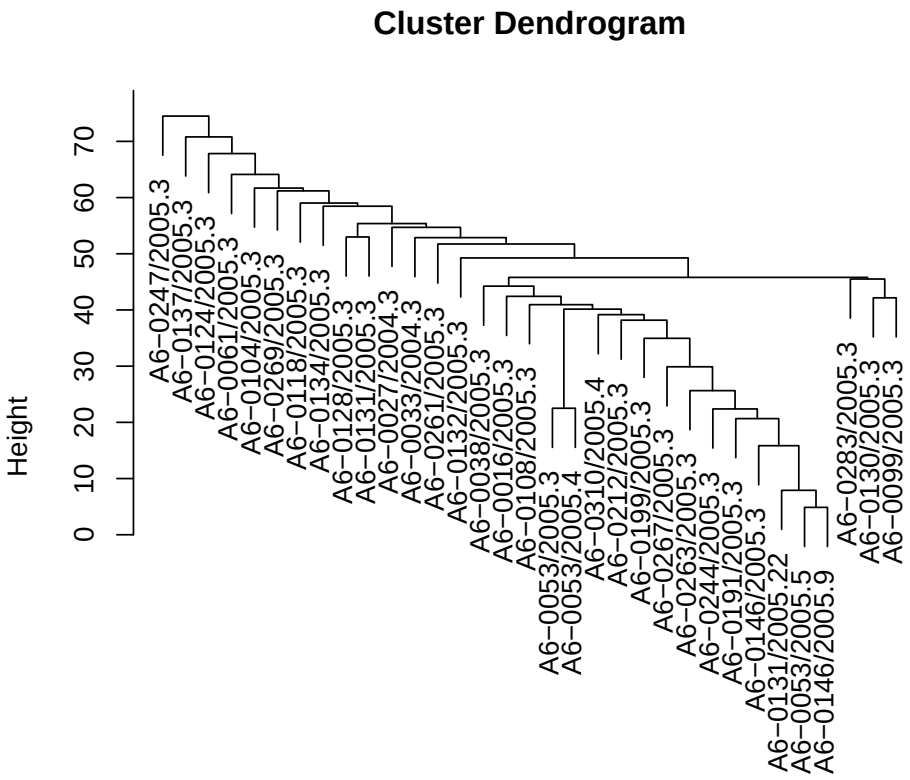
technique again, using only the rapporteurs' speeches, in hopes of generating a representation of the data that can tell us something about the similarity of the topics under debate.

```
> 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)
```



Additionally, we might also try visualizing the rapporteur's speeches from a bigram-based perspective:

```
> debates.rap.bigram <- filterDocuments(debates.bigram,
+   FunctionalDocumentFilter(function(x) rap == 1))
> dist.rap.bigram <- dist(documentTokenMatrix(debates.rap.bigram, weightTfIdf))
> clust.rap.bigram <- hclust(dist.rap.bigram)
> plot(clust.rap.bigram)
```



References

[1] Ingo Feinerer, Kurt Hornik, David Meyer. 2008. "Text Mining Infrastructure in R." *Journal of Statistical Software* 25 (5).

[2] Daniel Pemstein. 2008. "Predicting Roll Calls with Legislative Text." Presented at *The 25th Annual Summer Meeting of the Society for Political Methodology.*

[3] Duncan Temple Lang. 2008. "XML: Tools for parsing and generating XML within R and S-Plus." `http://cran.r-project.org/web/packages/XML/index.html`.