# An Introduction to **RSNL**

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

August 31, 2009

## 1 Introduction

**RSNL** (R Statistics with Natural Language) provides R programmers with access to an aRSeNaL of natural language processing (NLP) tools 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 `DocumentTermMatrix`, 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** `RSNLCorpus` 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 [3]. 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.

**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.

```xml
<?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>
    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.
  </TEXT>
</DEBATE-SPEECH>
```


Figure 1: An example EP speech in XML format.

```
> library(RSNL)
> library(XML)
> # 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,
+         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 snipped above is a `FunctionGenerator`; that is, rather than returning a standard object when called, `readSpeeches()` returns a function. Specifically, each invocation of `readSpeeches()` returns a function that takes information about a single element (that is, document) read from a source—the element itself, loading info which we ignore here, the document language, and a document id—and returns a `PlainTextDocument` suitable for inclusion in a `Corpus`. In this case, we use the **xml** package[5] to parse each EP speech in XML format and populate the fields and metadata slots of each resulting `PlainTextDocument`.[2] The first chunk of the function returned by `readSpeeches()` 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

[2] The `FunctionGenerator()` function call in the above code acts only to set the type (i.e. `FunctionGenerator`) of the resulting function-generating object. Remember that, in R, functions are objects and can carry type information. For type safety, **tm**'s `Corpus` constructor expects all readers to be `FunctionGenerator` objects. In fact, the code `FunctionGenerator(function (...)    function () )` is equivalent to `new("FunctionGenerator", .Data = function (...)    function () )`.

allowing the user to manipulate references to the underlying data in R itself. Yet, while logical, these implementation-dependent differences in `Corpus` passing semantics may risk some potential confusion for users and, in the case of in-memory storage, inefficiency.

### 2.1.2 `RSNLCorpus` and Reference Objects

**RSNL** provides a wrapper class for `Corpus` objects, `RSNLCorpus`, that unifies corpus semantics. `RSNLCorpus` 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 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 `RSNLCorpus`. Doing this is exceedingly simple; one need only pass an existing `Corpus` to the `RSNLCorpus` constructor:

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

`RSNLCorpus` 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 `RSNLCorpus 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 `RSNLCorpus` 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 `RSNLCorpus` 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.

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

## 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 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:

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

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

On the other hand, given a `RSNLCorpus`, `tokenize()` will generate a view of that corpus:

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

```
A tokenized corpus view with 179959 total tokens and 9265 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.[4] 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 31 total tokens and 26 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 `RSNLCorpus` object. Each view maintains a *reference* to a `RSNLCorpus` 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[5] used to break the text in `debates` into individual tokens. Similarly, a `TokenizedDocumentView` references a single document within a `RSNLCorpus` while maintaining a policy for representing that document as a sequence of tokens.[6] 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] "The"           "next"          "item"
 [4] "is"            "the"           "report"
 [7] "-LRB-"         "A6-0027"       "\\/"
[10] "2004"          "-RRB-"         "by"
[13] "Mrs"           "Corbey"        "on"
[16] "the"           "draft"         "European"
[19] "Parliament"    "and"           "Council"
[22] "Directive"     "amending"      "Directive"
[25] "94\\/62\\/EC" "on"            "packaging"
[28] "and"           "packaging"     "waste"
[31] "."
```

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

[4] Note that a view of a document maintains a policy for representing the document (in this case a tokenization policy) while keeping track of what corpus the document comes from. Thus, applying `tokenize` to a corpus and using the `index` argument is quite distinct from applying tokenize directly to a document within a corpus, which simply generates a vector of tokens. We discuss views in greater detail below.

[5] We describe `Tokenizer` objects in more detail below.

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

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

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
10727  9467  6253  5710  5489  4823  3261  3045  2832
    a   for     I    on    be  this    we    it   are
 2752  2383  1871  1665  1603  1573  1374  1195  1188
  not  have
 1157  1121
```

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 `RSNLCorpus` 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.[7]

By default, `tokenize()` uses a tokenizer that breaks up the text according to Penn Treebank conventions, 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))
```

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

```
[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 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 `RSNLCorpus` 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—Java 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):[8]

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

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

 [1] "The"         "next"        "item"
 [4] "is"          "the"         "report"
 [7] "A6"          "0027"        "2004"
```

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

```
[10] "by"         "Mrs"        "Corbey"
[13] "on"         "the"        "draft"
[16] "European"   "Parliament" "and"
[19] "Council"    "Directive"  "amending"
[22] "Directive"  "94"         "62"
[25] "EC"         "on"         "packaging"
[28] "and"        "packaging"  "waste"
```

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

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

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 179959 total tokens and 9265 unique tokens
```

```
> tmMap(debates, tmTolower)
```

```
A corpus with 475 text documents
```

```
> debates.tok
```

```
A tokenized corpus view with 179958 total tokens and 8535 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, one can use **tm**'s `tmMap()` method to modify the text contained within a `RSNLCorpus` 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 `TokenizedCorpusViews` 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 41262 total tokens and 462 unique tokens

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

  european     propos    commiss    program
       786        713        674        506
    direct      amend     presid     report
       497        463        434        359
parliament     NUMBER   committe    support
       355        348        313        306
      time    protect      europ      peopl
       273        267        257        255
    energi     safeti      thank particular
       254        252        252        249
```

These operations take advantage of **RSNL**'s `filterTokens()`, `transformTokens()`, and `stem()` methods to (1) filter out common English words,[9] (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 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.

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

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

especially short tokens from 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[10] 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"     "-LRB-"
 [4] "NUMBER"     "-RRB-"      "draft"
 [7] "european"   "parliament" "council"
[10] "direct"     "amend"      "direct"
[13] "packag"     "packag"
```

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

 [1] "the"           "next"          "item"
 [4] "is"            "the"           "report"
 [7] "-LRB-"         "a6-0027"       "\\/"
[10] "2004"          "-RRB-"         "by"
[13] "mrs"           "corbey"        "on"
[16] "the"           "draft"         "european"
[19] "parliament"    "and"           "council"
[22] "directive"     "amending"      "directive"
[25] "94\\/62\\/ec" "on"            "packaging"
[28] "and"           "packaging"     "waste"
[31] "."
```

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 `RSNLCorpus` and `TokenizedView` objects—in which case they return `TokenizedView` objects of the appropriate type.

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

 [1] "the"           "next"          "item"
 [4] "is"            "the"           "report"
 [7] "-LRB-"         "a6-0027"       "\\/"
[10] "2004"          "-RRB-"         "by"
[13] "mrs"           "corbey"        "on"
[16] "the"           "draft"         "european"
```

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

```
[19] "parliament"   "and"          "council"
[22] "direct"       "amend"        "direct"
[25] "94\\/62\\/ec" "on"           "packag"
[28] "and"          "packag"       "wast"
[31] "."
```

```
> stem(debates)

A tokenized corpus view with 179958 total tokens and 5483 unique tokens

> stem(debates, tokenizer=RegexTokenizer(), index=1)

A tokenized document view of A6-0027/2004.1 with 37 total tokens and 29 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.[11] 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 62280 total tokens and 47459 unique tokens

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

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

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

     european union      ladi gentlemen
                187                 133
     commiss propos        drive licenc
                120                 100
european parliament         -LRB- -RRB-
```

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

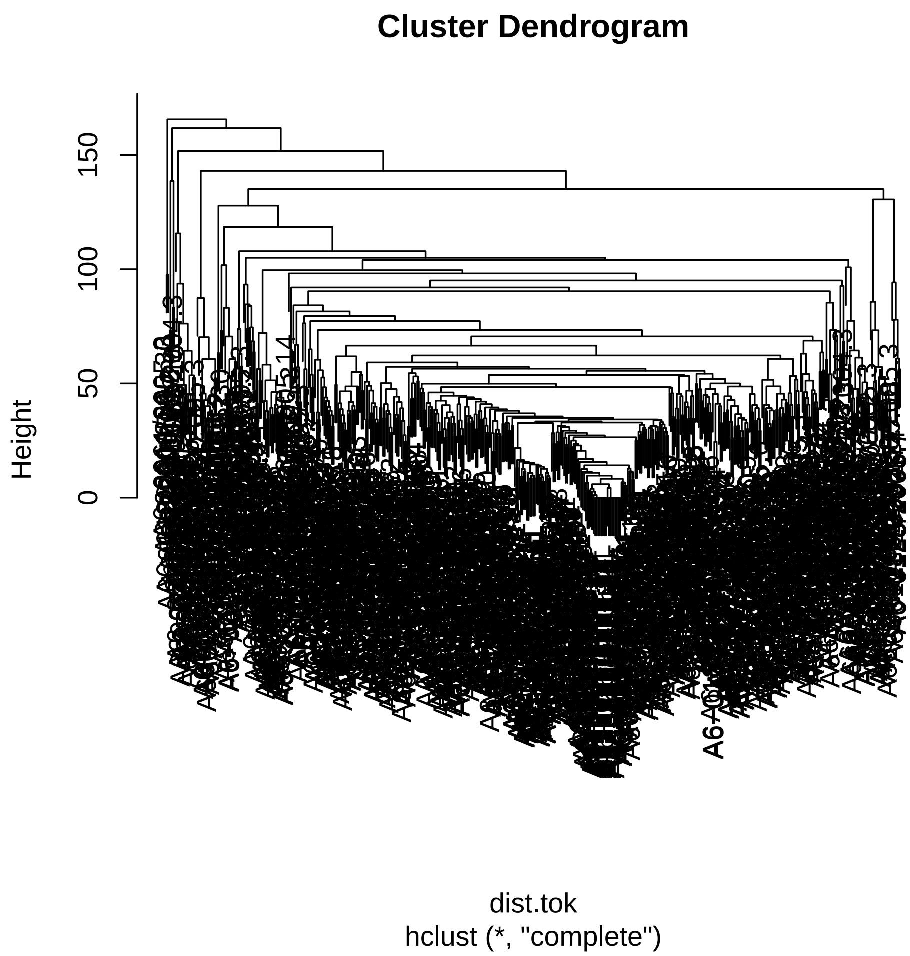
```
                 88                    86
       madam presid   presid commission
                 85                    84
      public health           food safeti
                 65                    62
      natura NUMBER          NUMBER -RRB-
                 60                    58
        presid ladi  parliament council
                 57                    53
           -RRB- NA    thank rapporteur
                 51                    51
  committe environ       environ public
                 49                    48
      energi effici           health food
                 47                    47
```

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.[12] 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 `DocumentTermMatrix` 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)
```

[12] 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 `DocumentTermMatrix` object as defined by the **tm** package.

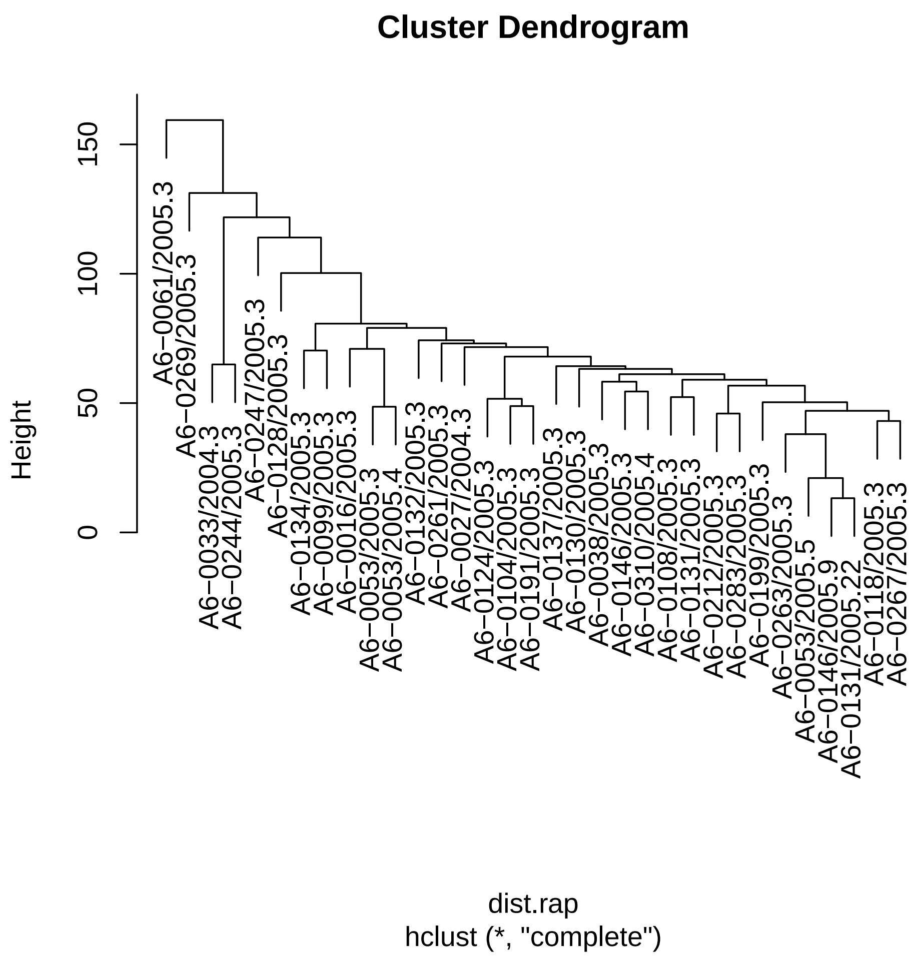
### 2.3.2 Document Filters

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 `RSNLCorpus` 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.[13] 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

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

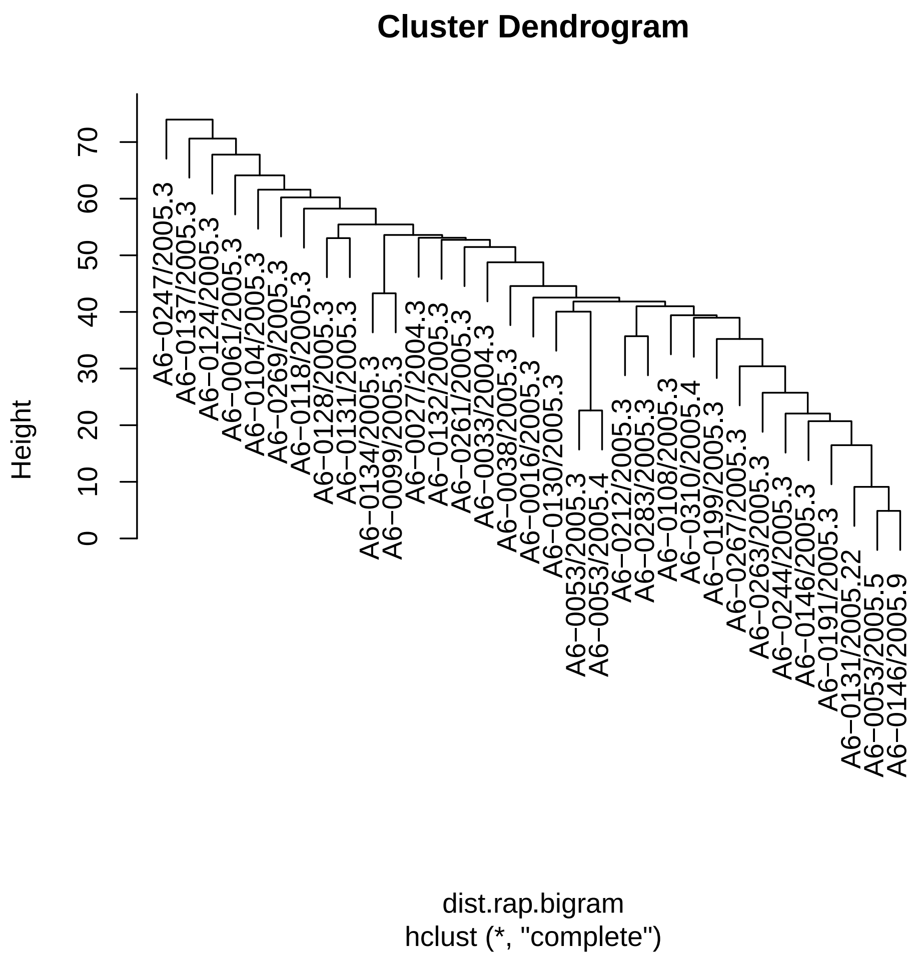
technique again, using only the rapporteurs' speeches, in hopes of generating a representation of the data that can tell us something about the similarity of the topics under debate.

```
> 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)
```




## 2.4 Tagging and Tagged Views

So far we have dealt with simple tokenized representations of documents and corpora. **RSNL** also provides tools for annotating tokenized text with labels, or tags. Tags provide information about individual tokens in a view; one can use tags to annotate tokens with part-of-speech information, entity type (e.g. person, place, or thing), or any other piece of information the user might have about individual tokens. Tags allow users to incorporate semantic information into their analyses and otherwise enhance the raw text with pertinent token-level metadata. In this section we demonstrate how **RSNL** represents text collections in terms of sequences of tagged tokens, using `TaggedView` objects. The `TaggedView` types extend the `TokenizedView` classes; thus a `TaggedCorpusView` is a type of `TokenizedCorpusView` and a `TaggedDocumentView` is a sort of`TokenizedCorpusView`. Therefore, the methods we provide to interrogate tokenized views—such as `documentTokenMatrix()`, `freq()`, `freqTable()`, `has()`, and `unique()`—all operate on tagged views, although the output of these methods may differ across types. Furthermore, tagged views provide a number of new methods that allow the user to effectively take advantage of the token label information represented by these views.

Because taggers often take advantage of case to do their work, we'll copy over our all lowercase `debates` object before proceeding.

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

### 2.4.1 Applying a Tagger

Tagging is similar to tokenization in practice and **RSNL** provides a method, `tag()` that can generate tagged tokens from a variety of data types. Tags are applied to tokens and **RSNL**'s taggers are designed to apply tags to objects representing token vectors. Given a `character` object, `tag()` will treat the object as a vector of tags if it contains more than one element; on the other hand, if the input contains only one element (or is a `TextDocument` object), `tag()` will treat the input as raw text and apply a tokenizer to it before tagging:

```
> tag(debates[[1]]) # Uses the default MaxentTagger and PTBTokenizer

 [1] "The/DT"           "next/JJ"
 [3] "item/NN"          "is/VBZ"
 [5] "the/DT"           "report/NN"
 [7] "-LRB-/-LRB-"      "A6-0027/NNP"
 [9] "\\//VBD"          "2004/CD"
[11] "-RRB-/-RRB-"      "by/IN"
[13] "Mrs/NNP"          "Corbey/NNP"
[15] "on/IN"            "the/DT"
[17] "draft/NN"         "European/JJ"
[19] "Parliament/NNP"   "and/CC"
[21] "Council/NNP"      "Directive/NNP"
[23] "amending/VBG"     "Directive/NNP"
[25] "94\\/62\\/EC/NNP" "on/IN"
[27] "packaging/NN"     "and/CC"
[29] "packaging/VBG"    "waste/NN"
[31] "./."

> tag(c("George Washington lived at Mount Vernon."),
+     tokenizer=RegexTokenizer("\\s+", FALSE))

[1] "George/NNP"     "Washington/NNP" "lived/VBD"
[4] "at/IN"          "Mount/NNP"      "Vernon./NNP"

> tag(c("George Washington lived at Mount Vernon."), MaxentTagger("english"))

[1] "George/NNP"     "Washington/NNP" "lived/VBD"
[4] "at/IN"          "Mount/NNP"      "Vernon/NNP"
[7] "./."

> tag(c("George", "Washington", "lived", "at", "Mount", "Vernon", "."),
+       NamedEntityTagger())

[1] "George/PERSON"     "Washington/PERSON"
[3] "lived/O"           "at/O"
[5] "Mount/LOCATION"    "Vernon/LOCATION"
[7] "./O"
```

The preceding code segment demonstrates the two types of taggers that currently ship with **RSNL**. The first is a Maximum Entropy based part-of-speech (POS) tagger developed by the Stanford

Natural Language Processing Group, providing pre-trained models for Arabic, Chinese, English, and German text [6]. The English tagger uses tag codes from the Penn Treebank. The appendix to this document contains a list of tags and corresponding parts of speech; Penn's tagging conventions are described in detail in the Treebank's part-of-speech tagging guide [4].

`MaxentTagger`s are designed to operate on text that is tokenized according to Penn Treebank conventions and thus should generally always be fed token vectors and views generated with a `PTBTokenizer`. Nonetheless, it is possible to specify a custom tokenizer when invoking `tag()`, as the second line of the above example shows. The second type of tagger is a Named Entity Recognizer (NER), also developed by the Stanford NLP Group [2], which assigns PERSON, LOCATION, and ORGANIZATION labels to English text.

When applied to a `character` vector `tag()` returns a vector of type `Tagged`. `Tagged` is a simple extension of the base `character` type that associates tags with tokens. When printed to the screen `Tagged` elements are displayed in token/tag format, but, internally, operations on `Tagged` objects only see tokens unless they request tags explicitly. One can extract tags with `label()` and convert a `Tagged` object to a `character` vector of the form token/tag with the `flatten()` method:

```
> (tagged <- tag(c("George Washington lived at Mount Vernon.")))

[1] "George/NNP"     "Washington/NNP" "lived/VBD"
[4] "at/IN"          "Mount/NNP"      "Vernon/NNP"
[7] "./."

> label(tagged)

[1] "NNP" "NNP" "VBD" "IN"  "NNP" "NNP" "."

> flatten(tagged)

[1] "George/NNP"     "Washington/NNP" "lived/VBD"
[4] "at/IN"          "Mount/NNP"      "Vernon/NNP"
[7] "./."
```

Of course, one can also tag a `RSNLCorpus`, yielding a view of the corpus, or generate a tagged view of a single document:

```
> (debates.pos <- tag(debates))

A tagged corpus view with 179959 total tagged tokens,
 10489 unique tagged tokens, 9265 unique tokens, and 43 unique tags

> (debates.pos.1 <- tag(debates, index=1))

A tagged document view of A6-0027/2004.1 with 31 total tagged tokens,
 27 unique tagged tokens, 26 unique tokens, and 13 unique tags
```

It is also possible to generate a tagged view from a tokenized view, inheriting the view's transforms and filters[14] in the process:

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

A tagged corpus view with 179959 total tagged tokens,
 10220 unique tagged tokens, 6194 unique tokens, and 43 unique tags
```

[14] As long as they are tag-safe. See Section 2.4.3 for details.

### 2.4.2 Working with Tagged Views

As with `TokenizedView` object, we can examine out `TaggedView`s with a variety of methods. First of all, we can examine the tokens in a document just as we would in a standard `TokenizedView`, except now the method returns an object of type `Tagged`:

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

 [1] "The/DT"            "next/JJ"
 [3] "item/NN"           "is/VBZ"
 [5] "the/DT"            "report/NN"
 [7] "-LRB-/-LRB-"       "A6-0027/NNP"
 [9] "\\//VBD"           "2004/CD"
[11] "-RRB-/-RRB-"       "by/IN"
[13] "Mrs/NNP"           "Corbey/NNP"
[15] "on/IN"             "the/DT"
[17] "draft/NN"          "European/JJ"
[19] "Parliament/NNP"    "and/CC"
[21] "Council/NNP"       "Directive/NNP"
[23] "amending/VBG"      "Directive/NNP"
[25] "94\\/62\\/EC/NNP" "on/IN"
[27] "packaging/NN"      "and/CC"
[29] "packaging/VBG"     "waste/NN"
[31] "./."

> tokens(debates.pos.1)

 [1] "The/DT"            "next/JJ"
 [3] "item/NN"           "is/VBZ"
 [5] "the/DT"            "report/NN"
 [7] "-LRB-/-LRB-"       "A6-0027/NNP"
 [9] "\\//VBD"           "2004/CD"
[11] "-RRB-/-RRB-"       "by/IN"
[13] "Mrs/NNP"           "Corbey/NNP"
[15] "on/IN"             "the/DT"
[17] "draft/NN"          "European/JJ"
[19] "Parliament/NNP"    "and/CC"
[21] "Council/NNP"       "Directive/NNP"
[23] "amending/VBG"      "Directive/NNP"
[25] "94\\/62\\/EC/NNP" "on/IN"
[27] "packaging/NN"      "and/CC"
[29] "packaging/VBG"     "waste/NN"
[31] "./."
```

We can also take a look at the most common words in the corpus, although `freqTable()` now returns tagged-token frequencies by default. Nonetheless, we can use the optional `what` argument to see both tag and token frequencies:

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

```
 the/DT     ,/,      ./.   of/IN   to/TO  and/CC
  10727    9467    6253    5710    5489    4823
  in/IN  is/VBZ    a/DT  for/IN that/IN   I/PRP
   3255    2832    2752    2383    2254    1871
  on/IN   be/VB this/DT  we/PRP  it/PRP are/VBP
   1658    1603    1573    1374    1195    1188
 not/RB will/MD
   1157     988
```

```
> sort(freqTable(debates.pos, what="Tokens"), dec=T)[1:20]
```

```
  the     ,     .    of    to   and    in  that    is
10727  9467  6253  5710  5489  4823  3261  3045  2832
    a   for     I    on    be  this    we    it   are
 2752  2383  1871  1665  1603  1573  1374  1195  1188
  not  have
 1157  1121
```

```
> sort(freqTable(debates.pos, what="Tags"), dec=T)[1:20]
```

```
   NN    IN    DT    JJ   NNS     ,   NNP    VB    RB
23689 22584 19178 12589 10248  9467  8757  7809  7804
  PRP     .    CC    TO   VBZ   VBP   VBN    MD   VBG
 6860  6386  5756  5515  4934  4458  4408  3501  2854
   CD  PRP$
 2607  1701
```

Similarly, we can generate lists of unique tagged tokens, tokens, and tags

```
> u.tagtok <- unique(debates.pos)
> u.tok <- unique(debates.pos, what="Tokens")
> (u.tag <- unique(debates.pos, what="Tags"))
```

```
 [1] "DT"    "JJ"    "NN"    "VBZ"   "-LRB-" "NNP"
 [7] "VBD"   "CD"    "-RRB-" "IN"    "CC"    "VBG"
[13] "."     ","     "NNS"   "VBN"   "NNPS"  "TO"
[19] ":"     "PRP$"  "VB"    "RB"    "MD"    "JJR"
[25] "RBR"   "RP"    "PRP"   "VBP"   "WDT"   "WRB"
[31] "JJS"   "POS"   "RBS"   "EX"    "WP"    "PDT"
[37] "``"    "''"    "WP$"   "FW"    "UH"    "LS"
[43] "SYM"
```

or generate document-token/tag/tagged-token matrices:

```
> documentTokenMatrix(debates.pos)
```

```
A document-term matrix (475 documents, 9265 terms)

Non-/sparse entries: 86652/4314223
Sparsity           : 98%
Maximal term length: 26
Weighting          : term frequency (tf)
```

```
> documentTagMatrix(debates.pos)

A document-term matrix (475 documents, 43 terms)

Non-/sparse entries: 12493/7932
Sparsity           : 39%
Maximal term length: 5
Weighting          : term frequency (tf)

> documentTaggedTokenMatrix(debates.pos)

A document-term matrix (475 documents, 10489 terms)

Non-/sparse entries: 88542/4893733
Sparsity           : 98%
Maximal term length: 29
Weighting          : term frequency (tf)
```

### 2.4.3 Transforming and Filtering Tagged Views

One can apply transforms and filters to tagged views, just as one may with tokenized views. Indeed, because a tagged view is a type of tokenized view, one can apply a large variety of of token transforms and filters directly to tagged views. For example, we can remove stopwords from a tagged view just as we might from a tokenized view:

```
> (debates.pos <- filterTokens(debates.pos, StopFilter()))

A tagged corpus view with 83312 total tagged tokens,
 9743 unique tagged tokens, 8696 unique tokens, and 33 unique tags
```

Furthermore, when one generates a tagged view from a tokenized view, the tagged view inherits the former's transforms:

```
> debates.tok <- tokenize(debates)
> debates.tok <- transformTokens(debates.tok,
+   RegexTokenTransform("^[0-9]+$", "NUMBER"))
> debates.tok <- filterDocuments(debates.tok, # 20% sample
+   FunctionalDocumentFilter(function(x) runif(length(x)) < .8))
> (debates.pos <- tag(debates.tok))

A tagged corpus view with 146457 total tagged tokens,
 9357 unique tagged tokens, 8279 unique tokens, and 42 unique tags
```

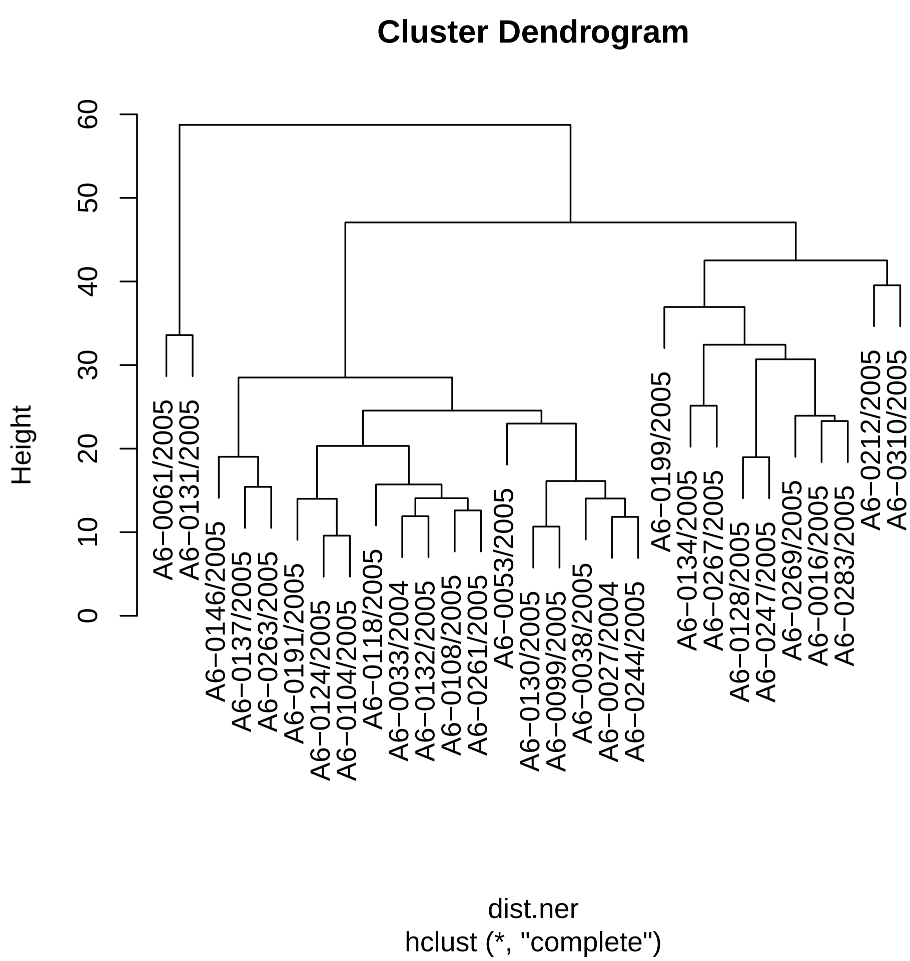
**RSNL** also provides tools that make tag-based filtering especially convenient. For example, we can restrict a view to noun forms as follows:

```
> filterTokens(debates.pos, RegexTagFilter("^NN"))

A tagged corpus view with 35967 total tagged tokens,
 4207 unique tagged tokens, 4138 unique tokens, and 4 unique tags
```

Along a similar vein, we can use the NER tagger to extend our cluster analysis from the previous section. This time we'll cluster debates purely in terms of organizations mentioned, collapsing word counts across speeches in the same debate:

```
> # Tag and filter
> debates.ner <- tag(debates, NamedEntityTagger())
> debates.ner <- filterTokens(debates.ner,
+   FunctionalTagFilter(function (x) x == "ORGANIZATION"))
> # Create a debate-token matrix
> docTM <- as.matrix(documentTokenMatrix(debates.ner))
> codes <- sapply(debates, meta, tag="CODE")
> ucodes <- unique(codes)
> debateTM <- t(sapply(ucodes,
+   function (ucode) apply(docTM[codes==ucode, ], 2, sum)))
> # Calculate distances and plot
> dist.ner <- dist(debateTM)
> clust.ner <- hclust(dist.ner)
> plot(clust.ner)
```




It is important to realize that one does face some limitations when applying token transforms to tagged views. First of all, it is important to note that **RSNL** tokenizes and tags text prior to applying token filters and transforms when constructing tagged views. This is generally the behavior that the user wants—both the POS and NER taggers are designed to operate on raw text and may perform poorly on transformed, and especially filtered, input—but it is important to keep in mind that transforms and filters are post-tag operations, even for tagged views that are constructed from filtered and transformed tokenized views. Furthermore, while all document and token filters are safe for use with tagged views the same does not hold true for token transforms more generally. The `TokenTransform` interface requires only that a transform provide a specialization of the `transformTokens` method that takes a `character` vector as its first argument and returns a transformed `character` vector. Therefore, arbitrary transforms may discard tag information when dealing with the contents of tagged views. To be tag-safe a transform must return an object of type `Tagged` when a `Tagged` vector is passed as the first argument to `transformTokens`. For example, we can build a tag-safe transform for converting tokens to upper case like so:

```
> ucTransform <- FunctionalTokenTransform(
+   function (x)
+     if (is(x, "Tagged"))
+       Tagged(toupper(x), label(x))
+     else
+       toupper(x), tagSafe=TRUE)
> sort(freqTable(transformTokens(debates.pos, ucTransform)), dec=T)[1:20]
```

```
  THE/DT       ,/,       ./.     OF/IN     TO/TO
    9582      7864      5171      4666      4574
  AND/CC     IN/IN    IS/VBZ      A/DT    FOR/IN
    4023      2946      2318      2299      2036
 THAT/IN   THIS/DT     I/PRP NUMBER/CD    WE/PRP
    1861      1576      1545      1541      1434
   ON/IN     BE/VB    IT/PRP   ARE/VBP    NOT/RB
    1420      1323      1308       963       940
```

Note that we indicate that `ucTransform` is a tag-safe object by passing the argument `tagSafe=TRUE` to the `FunctionalTokenTransform` constructor. This causes the constructor to generate a type of `FunctionalTokenTransform` object that extends the `TagSafe` class, indicating that it provides a tag-safe interface.[15] Furthermore, most of **RSNL**'s pre-built transform types—including `RegexTokenTransform`, `SnowballStemmer`, and `TolowerTokenTransform`—are tag-safe and implement the `TagSafe` interface.

[15] Note that passing `tagSafe=TRUE` to the constructor does not guarantee tag-safety, but rather provides an indicator that the transform conforms to the `TagSafe` interface; that is, that it provides a specialization of `transformTokens` that returns an object of type `Tagged` when a `Tagged` vector is provided as its first argument. Other code may test transforms for tag-safety prior to applying them by seeing if they extend the `TagSafe` class.

# 3 Appendix: Penn Treebank Part of Speech Codes

| | |
|---|---|
| CC | Coordinating conjunction |
| CD | Cardinal number |
| DT | Determiner |
| EX | Existential there |
| FW | Foreign word |
| IN | Preposition or subordinating conjunction |
| JJ | Adjective |
| JJR | Adjective, comparative |
| JJS | Adjective, superlative |
| LS | List item marker |
| MD | Modal |
| NN | Noun, singular or mass |
| NNS | Noun, plural |
| NNP | Proper noun, singular |
| NNPS | Proper noun, plural |
| PDT | Predeterminer |
| POS | Possessive ending |
| PRP | Personal pronoun |
| PRP$ | Possessive pronoun |
| RB | Adverb |
| RBR | Adverb, comparative |
| RBS | Adverb, superlative |
| RP | Particle |
| SYM | Symbol |
| TO | to |
| UH | Interjection |
| VB | Verb, base form |
| VBD | Verb, past tense |
| VBG | Verb, gerund or present participle |
| VBN | Verb, past participle |
| VBZ | Verb, 3rd person singular present |
| WDT | Wh-determiner |
| WP | Wh-pronoun |
| WP$ | Possessive wh-pronoun |
| WRB | Wh-adverb |

# References

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

[2] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. "Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling." Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005), pp. 363-370. http://nlp.stanford.edu/ manning/papers/gibbscrf3.pdf

[3] Daniel Pemstein. 2009. "Predicting Roll Calls with Legislative Text." Presented at *The 67th Annual National Conference of the Midwest Political Science Association*.

[4] Beatrice Santorini. 1990. "Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision, 2nd Printing)" `ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz`.

[5] 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`.

[6] Kristina Toutanova and Christopher D. Manning. 2000. "Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger." In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63-70.