Overview of the debate package

Emmanuel Rousseaux, Marion Deville and Gilbert Ritschard

2014.12.11

Contents

Introduction

The `debate` package offers facilities to analyse (political) debate in a semi-automatized way. This package offers a great flexibility of debate analysis, isolating relational aspects from the substancial ones and keeping them together in a solely graphic. It is especially designed to:

Code text From a .pdf file, you can extract member of parliament's interventions and code them based on textual markers. Dictionary can be composed and text exported as database or matrix. This allows one one hand to extract content analysis in a semi-automatic way for the substancial aspect of the discussion. On the other hand it offers tools to extract matrix of interaction for relational aspects of the discussion.

Graphically render a summary of the debate Based on the `spnet` package that offers facilities for rendering networks data on maps, the `debate` package is able to render a summary of the content and relational dynamics on the debate room. The debate rooms can be drafted from preset maps (examples) or designed by your own from a chekerboard basis where you can modify size, number and positions of boxes (`room.create.grid`). As the `spnet` package provides a lot of tools for changing color of boxes, labels, legend, adding symbols and draw networks, your summary of the debate is highly customizable.

Illustration

The `debate.example.basic` function provides a working example involving basic functionnalities of the `debate` package. Don't hesitate to look at its code and take what you need.

```
# net1 <- debate.example.basic()
# plot(net1)
```

User guide

Extracting content and relational from a debate transcript

Extracting text from a PDF file The first step is to convert a .pdf file into a text object. If your file is not in .pdf format, print it as .pdf with a software like `CUTEpdf Writer`, free of access and performant. If you need only a part of a text, print the .pdf into another one with only the pages concerned by the coding.

```
#txt <- pdftotext("the-pdf-file-to-convert.pdf")
```

To clean the text you can use the fonction `txt.prepare`

```
#txt <- txt.prepare(txt)
```

```
#txt <- gsub("nnnn","", txt)
```

Sometime the text extracted present many durty stuff and it could be useful to take a picture of the text before to extract it. In this case, use the function (. . . ..) in order to make this extra step and to have a cleaner corpus. The text is generated automatically into a .txt file.

Definition of stopping criteria The first step of a `debate` analysis consists to identify speech act one from the other by intervenant. The stopping criteria has to be define at the begining and at the end of the intervention. In offical parliamentary debate transcripts, each intervention is introduce by the name of the member of parliament. That defines the `pattern.start`. In some transcripts, each intervention concludes by a brief intervention of the presiding officer, which can be used as `pattern.stop`. If there is no textual marker which can be used as a stop (as double newline for exemple (does it work?)), you have to add stops manualy into the .txt file. For example, you can introduce the word `stop` at the end of each speech act, as a common `pattern.stop` criteria for all speakers.

The script to define the stopping criterias is the following:

```
stakeholders <- list(
  list(
    name = c("Greenwood"),
    pattern.start = c("\nMr Greenwood"),
    pattern.stop = c("\nthe presiding officer")
    ),
  list(
    name = c("de Gaulle"),
    pattern.start = c("\nCharles de Gaulle"),
    pattern.stop = c("\nLe président")
    )
)
```

`pattern.start` and `pattern.stop` are to be define for each stakeholder you want to analyse the discourse. The text outside of defined sections would not be taken into account for the analysis. Each stakeholder is labelized with the fonction `name` which will be reported into the database.

This step offers the possibility to add covariates in the database by intervenant for further analysis, and also to apprehend the debate in its temporal progress.

Setting a dictionary The further step consists to define a dictionnary to code the content of the debate. The script proceeds like in a manual content analysis (hyperlink for some references). It is a semi-automatic coding in the sens that you have to define categories, key word and formulations by yourself. You can define as much categories as you want, and as textual marks by categories as necessary.

The syntax is similar to the previous of stopping criteria. Each category has to be define by `name` and the keywords and formulations `match` are listed below.

```
dic <- list(
  list(
    name = "Energy",
    match = c("nuclear", "wind turbine", "Hydraulic dams", "hydaulic dams")
    ),
  list(
    name = "Ecosystem",
    match = c("biodiversity", "nature", "biosphere", "animals")
    )
)
```

Negative occurence can be signalized in a special code (how to do it?)

The step to define a good dictionnary is crucial. It needs some time to compose a grid draining as much possible aspects of the debate. To improuve the quality of analysis, you can try different methods of counting: keywords and short expression, only keywords, one code of each category by speech act, as much codes of the same category as occurences by speech act, etc.

Extracting content data You can extract content data by different ways of counting. For each count, you create a different object, which can be summarized as a variable.

For a total count of content in a corpus:

```
#content.overall <. txt.extract.dictionnary(txt, dic)
```

For a count by stakeholders:

```
#content.individual <- txt.extract.contributors(
#txt=txt,
#contrib=stakeholders,
#dictionary = dic
#)
```

To summarize:

```
#summarize(content.individual, count.by="intervention")

#summarize(content.individual, count.by="occurence")
```

`count.by="intervention"` reports a maximum of one code of each category by intervention. `count.by="occurences"` will report all occurences without limit by intervention. It is better to have tested both methods in order to reinforce robustness of the analysis.

Extracting relational data Relational data is extracted from references to other participants or collective actors during the intervention. You can also code external actors, depending of the analysis you want to show up. The syntax follows the same way that the content dictionary.

```
nodes <- list(
  list(
    name = "Obama",
    match = c("Obama")
  ),
  list(
    name = "Greens",
    match = c("Greens", "Ecologist parti", "ecologists", "Ecologists", "ecologist parti" )
  )
)
```

Be carreful, especially for collective actors, to identify every forms of markers for a same actor. The syntaxe is case-sensitive, you have to report the form with and without capital into the syntaxe.

By default, references are coded as neutre references, and exported as matrix.

You can define marker to characterize the network: approval or disapproval.

```
marker <- list(
  list(
    name = "Approval",
    match = c("I agree with", "As said by")
  ),
  list(
```

```
    name = "Disapproval",
    match = c("I don't agree with")
  )
)
```

But in regard to the large variety of textual markers, better to do it manually.

Creating the assembly map

The `spnet` package supports maps provided as a `SpatialPolygons` object.

```
room.u.0 <- room.create.u()
plot(room.u.0)
```

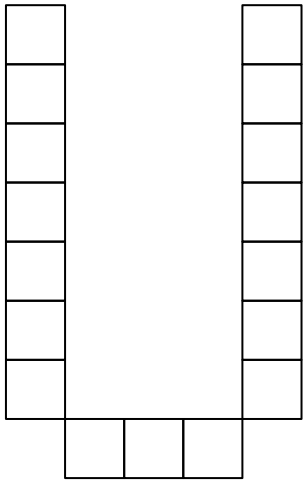

Figure 1: plot of chunk unnamed-chunk-13

```
room.u.1 <- room.create.u(c(9,4,9))
plot(room.u.1)
```

```
room.u.2 <- room.create.u(c(9,4,9), orientation = 'left')
plot(room.u.2)
```

```
room.u.4 <- room.create.u(c(9,4,9), orientation = 'bottom')
plot(room.u.4)
```

U-shape rooms

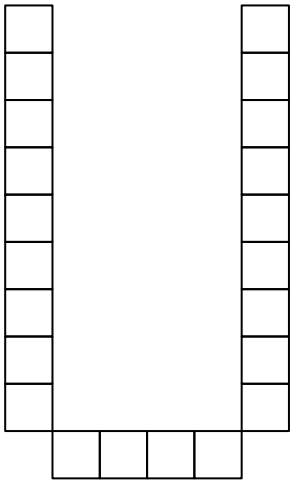

Figure 2: plot of chunk unnamed-chunk-14

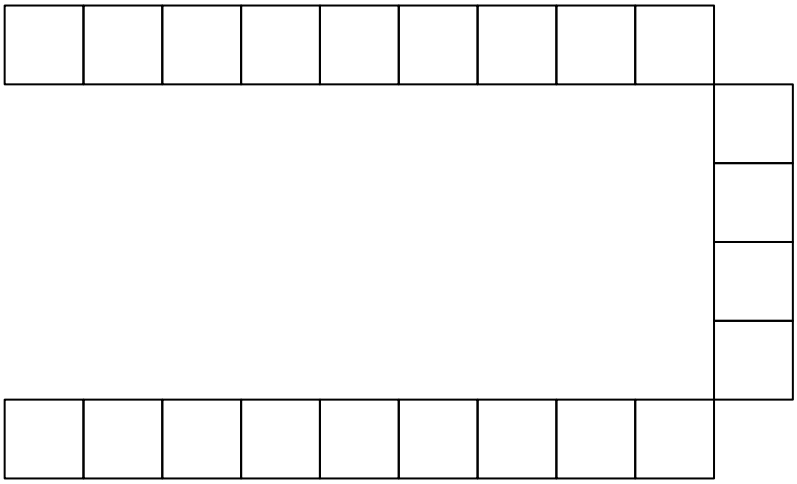

Figure 3: plot of chunk unnamed-chunk-15

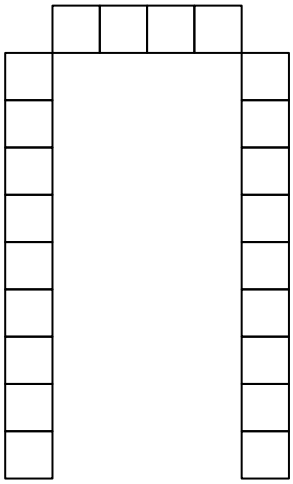

Figure 4: plot of chunk unnamed-chunk-16

Designing your own rooms The easiest way to create a room to represent a debate is with the `room.create.grid` function. Here is an example of use:

```
col <- 5
row <- 6
m <- matrix(rep(-1, col*row), nrow = row)
m[1,2:4] <- 0
m[3,c(1,5)] <- 0
m[4,c(1,5)] <- 0
m[5,c(1,5)] <- 0
m[6,c(1,5)] <- 0
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   -1    0    0    0   -1
## [2,]   -1   -1   -1   -1   -1
## [3,]    0   -1   -1   -1    0
## [4,]    0   -1   -1   -1    0
## [5,]    0   -1   -1   -1    0
## [6,]    0   -1   -1   -1    0
```

The -1 value corresponds to an empty grid, 0 to a grid. By default, grid are squared. To change proportion, you can modify it with functions `seat.width` and `seat.height`.

```
room1 <- room.create.grid(m, seat.width=2, seat.height=1)
spnet.map.plot.position(room1)
```

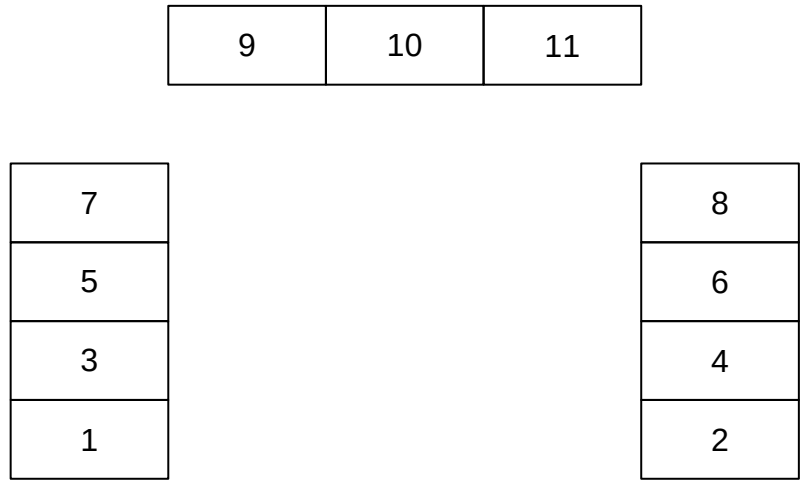

Figure 5: A simple room with table in invered 'U' form

If you want to shift a row or a column, you can do it into a `dimnames` object.

```
col <- 5
row <- 6
m2 <- matrix(rep(-1, col*row), nrow = row)
m2[1,2:3] <- 0
m2[3,c(1,4)] <- 0
```

```
m2[4,c(1,5)] <- 0
m2[5,c(1,5)] <- 0
m2[6,c(1,4)] <- 0
m2
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   -1    0    0   -1   -1
## [2,]   -1   -1   -1   -1   -1
## [3,]    0   -1   -1    0   -1
## [4,]    0   -1   -1   -1    0
## [5,]    0   -1   -1   -1    0
## [6,]    0   -1   -1    0   -1
```

```
dimnames(m2) <- list(
 c('O', 'E', 'O', 'E', 'E', 'O'),
 rep('E', 5)
)
```

Lines 1, 3 and 6 have been shifted of 0.5 grid on the right. To do it you have to defined as odd lines 'O' in the `dimnames` object. Same operation can be done with columns.

```
room2 <- room.create.grid(m2, seat.width=2, seat.height=1)
spnet.map.plot.position(room2)
```

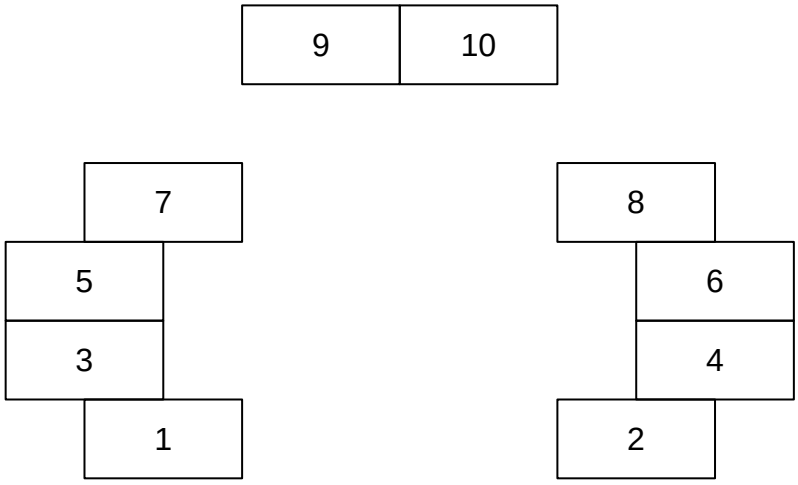

Figure 6: A simple room with table in invered 'U' form

For complete graph options (spnet vignette)

For an example of analysis (science slam example)