Overview of the debate package

Emmanuel Rousseaux, Marion Deville and Gilbert Ritschard

2015.01.23

Contents

Introduction

The `debate` package offers tools to analyse (political) debate in a semi-automatized way. This package offers a great flexibility to isolate relational aspects from the substancial ones by keeping them together in a solely graph. It is especially designed to:

Code text From a .pdf file, you can extract members of parliament's interventions and code them based on textual markers. A dictionary can be composed and a text can be as a database. This allows on the 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 rendering 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 based on a grid 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 drawing networks, your summary of the debate is highly customizable.

Illustration

The `debate.example.basic` function provides a working example which involves basic functionnalities of the `debate` package. Do not hesitate to look at its code and take what you need.

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

User guide

Extracting content and relations from a debate transcript

Extracting text from a PDF file We assume the text you want to extract is contained in a PDF file. If the the text you want to extract is not in PDF format, convert it in PDF with a software like *CUTEpdf Writer* (free of charge). If you only need specific pages of the PDF file, print it (with the PDF virtual printer) into another one with only the pages to code.

It is not so easy to extract text from a PDF file. The `debate.pdftotext` command works well in a lot of cases and is quite efficient. The function provides two different modes of extraction. The standard mode is based on the `pdftotext` utility. Thus, the `pdftotext` utility first have to be installed on your system. On most Linux distributions, the command is included as part of the `poppler-utils` package. On Windows, the command is included as part of the `Xpdf` software. Assuming the `pdftotext` utility is installed on your system, you can convert your PDF file as follows

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

The text is generated into a .txt file.

Some PDF file can embed complex structures as for example several layers, fonts or forms. Although the PDF is correctly rendered by the PDF viewer, the conversion to text partially fails and a lot of text may be lost or corrupted. To avoid this, a strategy can be to convert the PDF file into a image and then pass it through an Optical Character Recognition (OCR) software. This second option generally achieve better results, but is also more time consuming. The `tesseract` option allows to run this strategy using the tesseract OCR. For using this strategy, you first need to install the tesseract OCR. On most linux distributions the software is provided as part of the `tesseract-ocr` package. You may also need to install specific language utilities, for example the `tesseract-ocr-fra` package for enabling French support. Windows users can download the software from the tesseract website. You also need to install the `convert` utility. This utility, part of the `imagemagick` software, is used in backend for converting PDF files into images. On most linux distribution the command is included as part of the `imagemagick` package. Windows users can download `imagemagick` from its official website. Assuming the OCR mode requirement are satisfied on your system, you can convert your PDF file as follows

```
#txt <- debate.pdftotext(path.to.pdf, backend = "tesseract", lang = "fra")
```

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

```
#txt <- debate.txt.clean(txt)
```

Definition of stopping criteria The first step of a `debate` analysis consists in identifying speech acts one from the other by protagonist. The stopping criteria has to be defined at the begining and at the end of the intervention. In offical parliamentary debate transcripts, each intervention is introduced by the name of the member of parliament. That defines the `pattern.start`. In some transcripts, each intervention concludes with a brief intervention of the presiding officer, which can be used as `pattern.stop`. If no textual marker can be used as a stop (as double newline `\n\n` for exemple), you have to add stops manually 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("de Gaulle"),
    pattern.start = c("\nCharles de Gaulle"),
    pattern.stop = c("\nLe président")
    ),
    list(
    name = c("Greenwood"),
    pattern.start = c("\nMr Greenwood"),
    pattern.stop = c("\n\n")
    )
)
```

`pattern.start` and `pattern.stop` have to be defined for each stakeholder if you want to analyse the discourse. The text outside of the 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 into the database by protagonist for further analysis and also to apprehend the debate in its temporal progress.

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

Each category has to be defined by `name`. The keywords and formulations `match` are listed below. The syntax is the following:

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

Negative occurences can be signalized in a extra-line at the end of each category. Each match will remove 1 unit from the total of the category.

```
#list(
#    name = "Energy",
#    match = c("nuclear", "wind turbine", "Hydraulic dams", "hydaulic dams")
#    nomatch = c('nuclear bomb')
#    )
```

The step to define a good dictionnary is crucial. It takes some time to create a grid which includes as many aspects of the debate as possible. To improve 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 many 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 can create a different object, which can be summarized as a dataframe.

For a total count of content in a corpus:

```
#content.overall <-  debate.content.extract.dictionary(txt, dic)
```

For a count by stakeholders:

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

For a count by category:

```
#content.occurences <- debate.txt.count.matches(
#  txt=txt,
#  contrib=names,
#  dictionary = dic
#  )
```

Extracting relatios data The relations 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 highlight. The syntax follows the same way that the content dictionary.

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

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

By default references are coded as neutral references and exported as matrix.

You can define markers to characterize the network: “approval” or “disapproval” (not already available).

```r
marker <- list(
  list(
    name = "Approval",
    match = c("I agree with", "As said by")
  ),
  list(
    name = "Disapproval",
    match = c("I don't agree with")
  )
)
```

The function `debate.relational.extract` allows to extract dataframes with the score of approval, disapproval and neutral reference for each member of parliament defined in precedent script. Afterwhat the dataframe is transformed into matrix and ready to plot on a room.

Creating the assembly map

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

U-shape rooms A simple room with a table in an U-shape is designed as follows

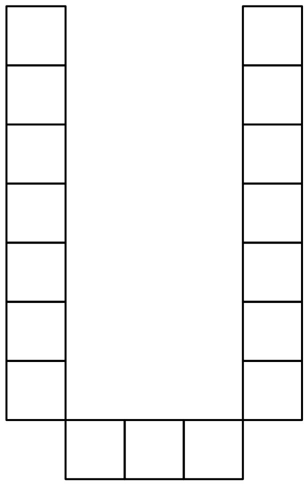
```r
room.u.0 <- room.create.u()
plot(room.u.0)
```

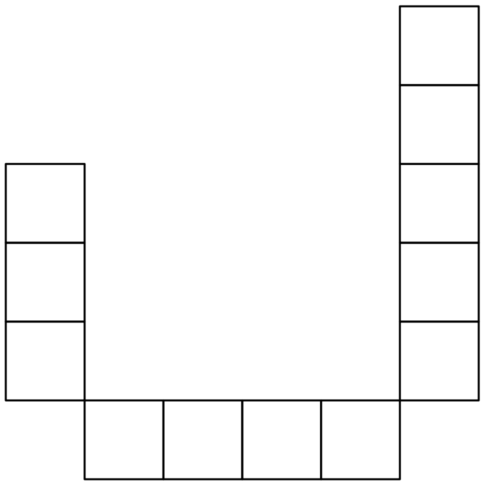
You can specify the number of seats in each part of the U

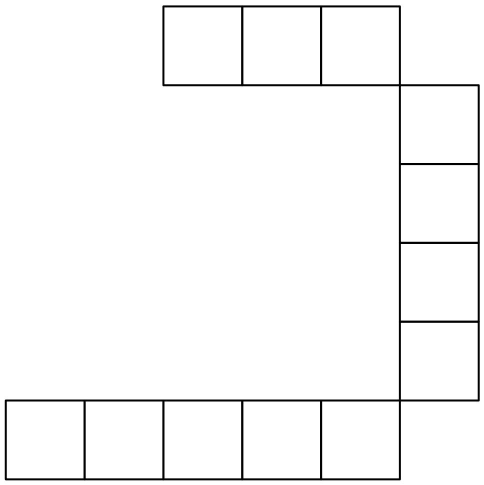
```r
room.u.1 <- room.create.u(c(3,4,5))
plot(room.u.1)
```

You can also change the orientation of the table

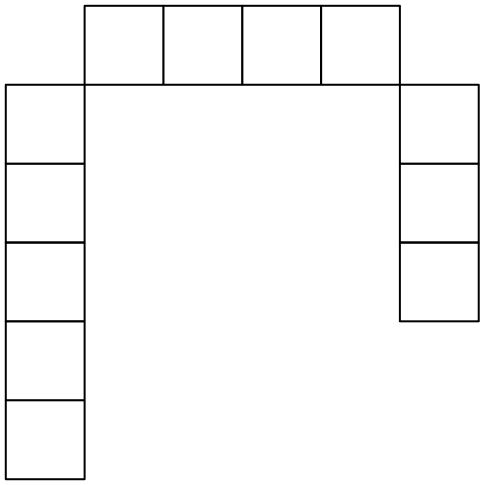
```r
room.u.2 <- room.create.u(c(3,4,5), orientation = 'left')
plot(room.u.2)
```







```
room.u.4 <- room.create.u(c(3,4,5), orientation = 'bottom')
plot(room.u.4)
```



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:

```
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
```

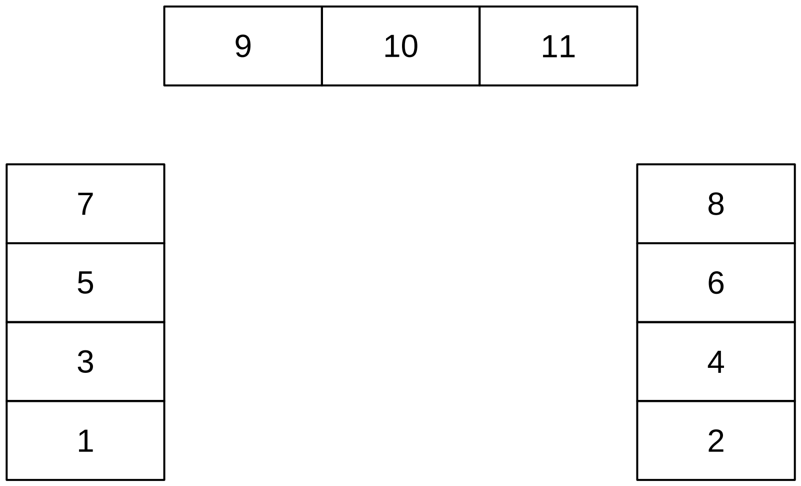
```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   -1    0    0    0   -1
## [2,]   -1   -1   -1   -1   -1
## [3,]    0   -1   -1   -1    0
## [4,]    0   -1   -1   -1    0
## [5,]    0   -1   -1   -1    0
## [6,]    0   -1   -1   -1    0
```

The -1 value corresponds to an empty grid, 0 to a grid. By default, grids are squared. To change the 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)
```

If you want to shift a row or a column, you can do it with the `dimnames` function.

```
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
```

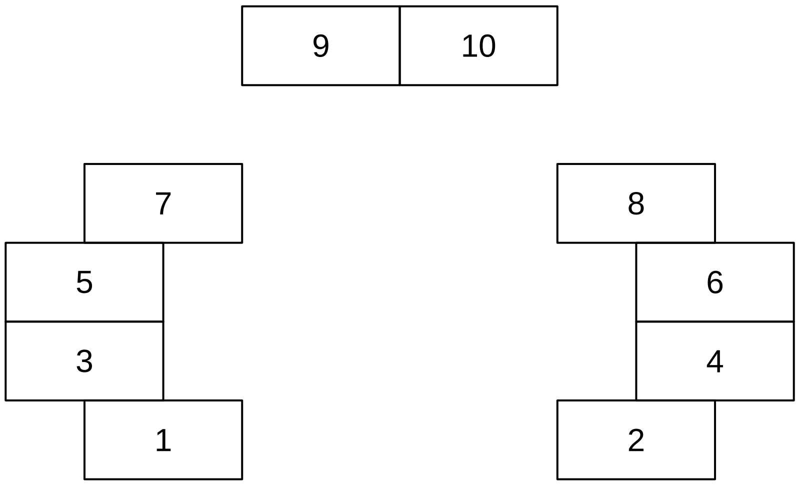
```
dimnames(m2) <- list(
 c('O', 'E', 'O', 'E', 'E', 'O'),
 rep('E', 5)
)
```

Lines 1, 3 and 6 have been shifted by 0.5 on the grid to the the right. To do it you have to define them as odd lines 'O' with `dimnames`. The same operation can be done with columns.

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

For complete graph options: spnet

For an example of analysis: example


9
10
7
8
5
6
3
4
1
2