**Introduction**

I started off this year by exploring a world that was unknown to me, the world of historical newspapers.  
I did not know that historical newspapers data was a thing, and have been thoroughly enjoying myself  
exploring the different datasets published by the National Library of Luxembourg. You can find  
the data [here](https://data.bnl.lu/data/historical-newspapers/).

In my [first blog post](https://www.brodrigues.co/blog/2019-01-04-newspapers/), I analyzed data from  
*L’indépendence Luxembourgeoise*. I focused on the ads, which were for the most part in the 4th and  
last page of the newspaper. I did so by extracting the data from the ALTO files. ALTO files contain  
the content of the newspapers, (basically, the words that make up the article). For this first  
exercise, I disregarded the METS files, for two reasons. First, I simply wanted to have something  
quick, and get used to the data. And second, I did not know about ALTO and METS files enough to  
truly make something out of them. The problem of disregarding the METS file is that I only had a big  
dump of words, and did not know which words came from which article, or ad in this case.

In the [second blog post](https://www.brodrigues.co/blog/2019-01-13-newspapers_mets_alto/)), I extracted  
data from the *L’Union* newspaper, this time by using the metadata from the METS files too. By  
combining the data from the ALTO files with the metadata from the METS files, I know which  
words came from which article, which would make further analysis much more interesting.

In the [third blog post](https://www.brodrigues.co/blog/2019-01-31-newspapers_shiny_app/) of this series,  
I built a Shiny app which makes it easy to explore the 10 years of publications of *L’Union*. In this  
blog post, I will explain in great detail how I created this app.

**Part 1: Getting the data ready for the Shiny app**

**Step 1: Extracting the needed data**

If you want to follow along with a dataset from a single publication, you can download the following archive on  
[dropbox](https://www.dropbox.com/s/56ttqetz4cirsja/1533660_newspaper_lunion_1860-11-14.zip?dl=0).  
Extract this archive, and you will find the data exactly as you would get it from the the big  
archive you can download from the website of the National Library of Luxembourg. However, to keep  
the size of the archive small, I removed the .pdf and .jpeg scans.

In the [second blog post](https://www.brodrigues.co/blog/2019-01-13-newspapers_mets_alto/)) I wrote  
some functions that made extracting the needed data from the files easy. However, after I wrote the  
article, I noticed that in some cases these functions were not working exactly as intended. I  
rewrote them a little bit to overcome these issues. You can find the code I used right below. I won’t  
explain it too much, because you can read the details in the previous blog post. However, should  
something be unclear, just drop me an email or a tweet!

Click if you want to see the code

# This functions will be used within the next functions to extract the relevant pieces

extractor <- function(string, regex, all = FALSE){

if(all) {

string %>%

str\_extract\_all(regex) %>%

flatten\_chr() %>%

str\_remove\_all("=|\\\"") %>%

#str\_extract\_all("[:alnum:]+|.|,|\\?|!", simplify = FALSE) %>%

map(paste, collapse = "") %>%

flatten\_chr()

} else {

string %>%

str\_extract(regex) %>%

str\_remove\_all("=|\\\"") %>%

#str\_extract\_all("[:alnum:]+|.|,|\\?|!", simplify = TRUE) %>%

paste(collapse = " ") %>%

tolower()

}

}

# This function extracts the data from the METS files, and returns a tibble:

extract\_mets <- function(article){

id <- article %>%

extractor("(?<=ID)(.\*?)(?=LABEL)")

label <- article %>%

extractor("(?<=LABEL)(.\*?)(?=TYPE)")

type <- article %>%

extractor("(?<=TYPE)(.\*?)(?=>)")

begins <- article %>%

extractor("(?<=BEGIN)(.\*?)(?=BETYPE)", all = TRUE)

tibble::tribble(~label, ~type, ~begins, ~id,

label, type, begins, id) %>%

unnest()

}

# This function extracts the data from the ALTO files, and also returns a tibble:

extract\_alto <- function(article){

begins <- article[1] %>%

extractor("(?<=^ID)(.\*?)(?=HPOS)", all = TRUE)

content <- article %>%

extractor("(?<=CONTENT)(.\*?)(?=WC)", all = TRUE)

tibble::tribble(~begins, ~content,

begins, content) %>%

unnest()

}

# This function takes the path to a page as an argument, and extracts the data from

# each article using the function defined above. It then writes a flat CSV to disk.

alto\_csv <- function(page\_path){

page <- read\_file(page\_path)

doc\_name <- str\_extract(page\_path, "(?<=/text/).\*")

alto\_articles <- page %>%

str\_split("TextBlock ") %>%

flatten\_chr()

alto\_df <- map\_df(alto\_articles, extract\_alto)

alto\_df <- alto\_df %>%

mutate(document = doc\_name)

write\_csv(alto\_df, paste0(page\_path, ".csv"))

}

# Same as above, but for the METS file:

mets\_csv <- function(page\_path){

page <- read\_file(page\_path)

doc\_name <- str\_extract(page\_path, "(?<=/).\*")

mets\_articles <- page %>%

str\_split("DMDID") %>%

flatten\_chr()

mets\_df <- map\_df(mets\_articles, extract\_mets)

mets\_df <- mets\_df %>%

mutate(document = doc\_name)

write\_csv(mets\_df, paste0(page\_path, ".csv"))

}

# Time to use the above defined functions. First, let's save the path of all the ALTO files

# into a list:

pages\_alto <- str\_match(list.files(path = "./", all.files = TRUE, recursive = TRUE), ".\*/text/.\*.xml") %>%

discard(is.na)

# I use the {furrr} library to do the extraction in parallel, using 8 cores:

library(furrr)

plan(multiprocess, workers = 8)

tic <- Sys.time()

future\_map(pages\_alto, alto\_csv)

toc <- Sys.time()

toc - tic

#Time difference of 18.64776 mins

# Same for the METS files:

pages\_mets <- str\_match(list.files(path = "./", all.files = TRUE, recursive = TRUE), ".\*mets.xml") %>%

discard(is.na)

library(furrr)

plan(multiprocess, workers = 8)

tic <- Sys.time()

future\_map(pages\_mets, mets\_csv)

toc <- Sys.time()

toc - tic

#Time difference of 18.64776 mins

If you want to try the above code for one ALTO and METS files, you can use the following lines  
(use the download link in the beginning of the blog post to get the required data):

Click if you want to see the code

mets <- read\_file("1533660\_newspaper\_lunion\_1860-11-14/1533660\_newspaper\_lunion\_1860-11-14-mets.xml")

mets\_articles2 <- mets %>%

str\_split("DMDID") %>%

flatten\_chr()

alto <- read\_file("1533660\_newspaper\_lunion\_1860-11-14/text/1860-11-14\_01-00001.xml")

alto\_articles <- alto %>%

str\_split("TextBlock ") %>%

flatten\_chr()

mets\_df2 <- mets\_articles2 %>%

map\_df(extract\_mets)

# Same exercice for ALTO

alto\_df <- alto\_articles %>%

map\_df(extract\_alto)

**Step 2: Joining the data and the metadata**

Now that I extracted the data from the ALTO files, and the metadata from the METS files, I still  
need to join both data sets and do some cleaning. What is the goal of joining these two sources?  
Remember, by doing this I will know which words come from which article, which will make things  
much easier later on. I explain how the code works as comments in the code block below:

Click if you want to see the code

library(tidyverse)

library(udpipe)

library(textrank)

library(tidytext)

# First, I need the path to each folder that contains the ALTO and METS files. Each newspaper

# data is inside its own folder, one folder per publication. Inside, there's `text` folder that

# contains the ALTO and METS files. This is also where I saved the .csv files from before.

pathdirs <- list.dirs(recursive = FALSE) %>%

str\_match(".\*lunion.\*") %>%

discard(is.na)

# The following function imports the METS and the ALTO csv files, joins them, and does some

# basic cleaning. I used a trick to detect German articles (even though L'Union is a French publication

# some articles are in German) and then remove them.

tidy\_papers <- function(path){

mets\_path <- paste0(path, "/", list.files(path, ".\*.xml.csv"))

mets\_csv <- data.table::fread(mets\_path)

alto\_path <- paste0(path, "/text/", list.files(paste0(path, "/text/"), ".\*.csv"))

alto\_csv <- map\_dfr(alto\_path, data.table::fread)

final <- full\_join(alto\_csv, mets\_csv, by = "begins") %>%

mutate(content = tolower(content)) %>%

mutate(content = if\_else(str\_detect(content, "hyppart1"), str\_extract\_all(content, "(?<=CONTENT\_).\*", simplify = TRUE), content)) %>%

mutate(content = if\_else(str\_detect(content, "hyppart2"), NA\_character\_, content)) %>%

# When words are separated by a hyphen and split over two lines, it looks like this in the data.

# ex SUBS\_TYPEHypPart1 SUBS\_CONTENTexceptée

# ceptée SUBS\_TYPEHypPart2 SUBS\_CONTENTexceptée

# Here, the word `exceptée` is split over two lines, so using a regular expression, I keep

# the string `exceptée`, which comes after the string `CONTENT`, from the first line and

# replace the second line by an NA\_character\_

mutate(content = if\_else(str\_detect(content, "superscript"), NA\_character\_, content)) %>%

mutate(content = if\_else(str\_detect(content, "subscript"), NA\_character\_, content)) %>%

filter(!is.na(content)) %>%

filter(type == "article") %>%

group\_by(id) %>%

nest %>%

# Below I create a list column with all the content of the article in a single string.

mutate(article\_text = map(data, ~paste(.$content, collapse = " "))) %>%

mutate(article\_text = as.character(article\_text)) %>%

# Detecting and removing german articles

mutate(german = str\_detect(article\_text, "wenn|wird|und")) %>%

filter(german == FALSE) %>%

select(-german) %>%

# Finally, creating the label of the article (the title), and removing things that are

# not articles, such as the daily feuilleton.

mutate(label = map(data, ~`[`(.$label, 1))) %>%

filter(!str\_detect(label, "embranchement|ligne|bourse|abonnés|feuilleton")) %>%

filter(label != "na")

# Save the data in the rds format, as it is not a flat file

saveRDS(final, paste0(path, "/", str\_sub(path, 11, -1), ".rds"))

}

# Here again, I do this in parallel

library(furrr)

plan(multiprocess, workers = 8)

future\_map(pathdirs, tidy\_papers)

This is how one of these files looks like, after passing through this function:

One line is one article. The first column is the id of the article, the second column contains  
a data frame, the text of the article and finally the title of the article.  
Let’s take a look at the content of the first element of the *data* column:

This is the result of the merger of the METS and ALTO csv files. The first column is the id of the  
article, the second column contains each individual word of the article, the *label* column the  
label, or title of the article.

**Step 3: Part-of-speech annotation**

Part-of-speech annotation is a technique with the aim of assigning to each word its part of speech.  
Basically, Pos annotation tells us whether a word is a verb, a noun, an adjective… This will  
be quite useful for the analysis. To perform Pos annotation, you need to install the {udpipe}  
package, and download the pre-trained model for the language you want to annotate, in my case French:

Click if you want to see the code

# Only run this once. This downloads the model for French

udpipe\_download\_model(language = "french")

# Load the model

udmodel\_french <- udpipe\_load\_model(file = 'french-gsd-ud-2.3-181115.udpipe')

# Save the path of the files to annotate in a list:

pathrds <- list.files(path = "./", all.files = TRUE, recursive = TRUE) %>%

str\_match(".\*.rds") %>%

discard(is.na)

annotate\_rds <- function(path, udmodel){

newspaper <- readRDS(path)

s <- udpipe\_annotate(udmodel, newspaper$article\_text, doc\_id = newspaper$label)

x <- data.frame(s)

saveRDS(x, str\_replace(path, ".rds", "\_annotated.rds"))

}

library(furrr)

plan(multiprocess, workers = 8)

tic <- Sys.time()

future\_map(pathrds, annotate\_rds, udmodel = udmodel\_french)

toc <- Sys.time()

toc - tic

And here is the result:

The *upos* column contains the tags. Now I know which words are nouns, verbs, adjectives, stopwords…  
Meaning that I can easily focus on the type of words that interest me. Plus, as an added benefit, I  
can focus on the lemma of the words. For example, the word *viennent*, is the  
[conjugated](https://en.wikipedia.org/wiki/French_conjugation) form of the verb *venir*. *venir* is  
thus the lemma of *viennent*. This means that I can focus my analysis on lemmata. This is useful,  
because if I compute the frequency of words, *viennent* would be different from *venir*, which is  
not really what we want.

**Step 4: tf-idf**

Just like what I did in my [first blog post](https://www.brodrigues.co/blog/2019-01-04-newspapers/),  
I compute the tf-idf of words. The difference, is that here the “document” is the article. This means  
that I will get the most frequent words inside each article, but who are at the same time rare  
in the other articles. Doing this ensures that I will only get very relevant words for each article.

In the lines below, I prepare the data to then make the plots. The files that are created using  
the code below are available in the following [Github link](https://github.com/b-rodrigues/newspapers_shinyapp/tree/master/tf_idf_data).

In the Shiny app, I read the data directly from the repo. This way, I can keep the app small in size.

Click if you want to see the code

path\_annotatedrds <- list.files(path = "./", all.files = TRUE, recursive = TRUE) %>% str\_match(".\*\_annotated.rds") %>%

discard(is.na)

prepare\_tf\_idf <- function(path){

annotated\_newspaper <- readRDS(path)

tf\_idf\_data <- annotated\_newspaper %>%

filter(upos %in% c("NOUN", "VERB", "ADJ", "PROPN")) %>%

filter(nchar(lemma) > 3) %>%

count(doc\_id, lemma) %>%

bind\_tf\_idf(lemma, doc\_id, n) %>%

arrange(desc(tf\_idf)) %>%

group\_by(doc\_id)

name\_tf\_idf\_data <- str\_split(path, "/", simplify = 1)[1] %>%

paste0("\_tf\_idf\_data.rds") %>%

str\_sub(start = 9, -1)

saveRDS(tf\_idf\_data, paste0("tf\_idf\_data/", name\_tf\_idf\_data))

}

library(furrr)

plan(multiprocess, workers = 8)

future\_map(path\_annotatedrds, prepare\_tf\_idf)

**Step 5: Summarizing articles by extracting the most relevant sentences, using {textrank}**

The last step in data preparation is to extract the most relevant sentences of each articles, using  
the {textrank} package. This packages implements the *PageRank* algorithm developed by Larry Page  
and Sergey Brin in 1995. This algorithm ranks pages by the number of links that point to the pages;  
the most popular and important pages are also the ones with more links to them. A similar approach  
is used by the implementation of {textrank}. The algorithm is explained in detail in the following  
[paper](https://web.eecs.umich.edu/~mihalcea/papers/mihalcea.emnlp04.pdf).

However, I cannot simply apply {textrank} to the annotated data frame as it is. Because I have  
several articles, I have to run the textrank\_sentences() function, which extracts the relevant  
sentences, article by article. For this I still need to transform the data set and also need to  
prepare the data in a way that makes it digestible by the function. I will not explain the code  
below line by line, since the documentation of the package is quite straightforward. However,  
keep in mind that I have to run the textrank\_sentences() function for each article, which explains  
that as some point I use the following:

group\_by(doc\_id) %>%

nest() %>%

which then makes it easy to work by article (*doc\_id* is the id of the articles). This part is  
definitely the most complex, so if you’re interested in the methodology described here, really  
take your time to understand this function. Let me know if I can clarify things!

Click if you want to see the code

library(textrank)

library(brotools)

path\_annotatedrds <- list.files(path = "./", all.files = TRUE, recursive = TRUE) %>% str\_match(".\*\_annotated.rds") %>%

discard(is.na)

prepare\_textrank <- function(path){

annotated\_newspaper <- readRDS(path)

# sentences summary

x\_text\_rank <- annotated\_newspaper %>%

group\_by(doc\_id) %>%

nest() %>%

mutate(textrank\_id = map(data, ~unique\_identifier(., c("paragraph\_id", "sentence\_id")))) %>%

mutate(cleaned = map2(.x = data, .y = textrank\_id, ~cbind(.x, "textrank\_id" = .y))) %>%

select(doc\_id, cleaned)

x\_text\_rank2 <- x\_text\_rank %>%

mutate(sentences = map(cleaned, ~select(., textrank\_id, sentence))) %>%

# one\_row() is a function from my own package, which eliminates duplicates rows

# from a data frame

mutate(sentences = map(sentences, ~one\_row(., c("textrank\_id", "sentence"))))

x\_terminology <- x\_text\_rank %>%

mutate(terminology = map(cleaned, ~filter(., upos %in% c("NOUN", "ADJ")))) %>%

mutate(terminology = map(terminology, ~select(., textrank\_id, "lemma"))) %>%

select(terminology)

x\_final <- bind\_cols(x\_text\_rank2, x\_terminology)

possibly\_textrank\_sentences <- possibly(textrank\_sentences, otherwise = NULL)

x\_final <- x\_final %>%

mutate(summary = map2(sentences, terminology, possibly\_textrank\_sentences)) %>%

select(doc\_id, summary)

name\_textrank\_data <- str\_split(path, "/", simplify = 1)[1] %>%

paste0("\_textrank\_data.rds") %>%

str\_sub(start = 9, -1)

saveRDS(x\_final, paste0("textrank\_data/", name\_textrank\_data))

}

library(furrr)

plan(multiprocess, workers = 8)

future\_map(path\_annotatedrds, prepare\_textrank)

You can download the annotated data sets from the following  
[link](https://github.com/b-rodrigues/newspapers_shinyapp/tree/master/textrank_data). This is how  
the data looks like:

Using the summary() function on an element of the *summary* column returns the 5 most relevant  
sentences as extracted by {textrank}.

**Part 2: Building the shiny app**

The most difficult parts are behind us! Building a dashboard is quite easy thanks to the {flexdashboard}  
package. You need to know Markdown and some Shiny, but it’s way easier than building a complete  
Shiny app. First of all, install the {fleshdashboard} package, and start from a template, or  
from [this list of layouts](https://rmarkdown.rstudio.com/flexdashboard/layouts.html).

I think that the only trick worth mentioning is that I put the data in a Github repo, and read it  
directly from the Shiny app. Users choose a date, which I save in a reactive variable. I then  
build the right url that points towards the right data set, and read it:

path\_tf\_idf <- reactive({

paste0("https://raw.githubusercontent.com/b-rodrigues/newspapers\_shinyapp/master/tf\_idf\_data/newspaper\_lunion\_", as.character(input$date2), "\_tf\_idf\_data.rds")

})

dfInput <- reactive({

read\_rds(url(path\_tf\_idf())) %>%

top\_n(as.numeric(input$tf\_df\_words), tf\_idf) %>%

mutate(word = reorder(lemma, tf\_idf))

})

Because I did all the computations beforehand, the app simply reads the data and creates the bar  
plots for the tf-idf data, or prints the sentences for the textrank data. To print the sentences  
correcly, I had to use some html tags, using the {htmltools} package. Below you can find the  
source code of the app:

Click if you want to see the code

---

title: "Exploring 10 years of daily publications of the Luxembourguish newspaper, \*L'Union\*"

output:

flexdashboard::flex\_dashboard:

theme: yeti

orientation: columns

vertical\_layout: fill

runtime: shiny

---

`` `{r setup, include=FALSE}

library(flexdashboard)

library(shiny)

library(tidyverse)

library(textrank)

library(tidytext)

library(udpipe)

library(plotly)

library(ggthemes)

`` `

Sidebar {.sidebar}

=====================================

`` `{r}

dateInput('date2',

label = paste('Select date'),

value = as.character(as.Date("1860-11-14")),

min = as.Date("1860-11-12"), max = as.Date("1869-12-31"),

format = "yyyy/mm/dd",

startview = 'year', language = 'en-GB', weekstart = 1

)

selectInput(inputId = "tf\_df\_words",

label = "Select number of unique words for tf-idf",

choices = seq(1:10),

selected = 5)

selectInput(inputId = "textrank\_n\_sentences",

label = "Select the number of sentences for the summary of the article",

choices = seq(1:20),

selected = 5)

`` `

\*The BnL has digitised over 800.000 pages of Luxembourg newspapers. From those, more than 700.000

pages have rich metadata using international XML standards such as METS and ALTO.

Multiple datasets are available for download. Each one is of different size and contains different

newspapers. All the digitised material can also be found on our search platform a-z.lu

(Make sure to filter by “eluxemburgensia”). All datasets contain XML (METS + ALTO), PDF, original

TIFF and PNG files for every newspaper issue.\*

Source: https://data.bnl.lu/data/historical-newspapers/

This Shiny app allows you to get summaries of the 10 years of daily issues of the "L'Union" newspaper.

In the first tab, a simple word frequency per article is shown, using the tf-idf method. In the

second tab, summary sentences have been extracted using the `{textrank}` package.

Word frequency per article

=====================================

Row

-----------------------------------------------------------------------

### Note: there might be days without any publication. In case of an error, select another date.

`` `{r}

path\_tf\_idf <- reactive({

paste0("https://raw.githubusercontent.com/b-rodrigues/newspapers\_shinyapp/master/tf\_idf\_data/newspaper\_lunion\_", as.character(input$date2), "\_tf\_idf\_data.rds")

})

dfInput <- reactive({

read\_rds(url(path\_tf\_idf())) %>%

top\_n(as.numeric(input$tf\_df\_words), tf\_idf) %>%

mutate(word = reorder(lemma, tf\_idf))

})

renderPlotly({

df\_tf\_idf <- dfInput()

p1 <- ggplot(df\_tf\_idf,

aes(word, tf\_idf)) +

geom\_col(show.legend = FALSE, fill = "#82518c") +

labs(x = NULL, y = "tf-doc\_idf") +

facet\_wrap(~doc\_id, ncol = 2, scales = "free") +

coord\_flip() +

theme\_dark()

ggplotly(p1)

})

`` `

Summary of articles {data-orientation=rows}

=====================================

Row

-----------------------------------------------------------------------

### The sentence in bold is the title of the article. You can show more sentences in the summary by using the input in the sidebar.

`` `{r}

print\_summary\_textrank <- function(doc\_id, summary, n\_sentences){

htmltools::HTML(paste0("**", doc\_id, "**"), paste("

", summary(summary, n\_sentences), sep = "", collapse = "  
"), "

")

}

path\_textrank <- reactive({

paste0("https://raw.githubusercontent.com/b-rodrigues/newspapers\_shinyapp/master/textrank\_data/newspaper\_lunion\_", as.character(input$date2), "\_textrank\_data.rds")

})

dfInput2 <- reactive({

read\_rds(url(path\_textrank()))

})

renderUI({

df\_textrank <- dfInput2()

df\_textrank <- df\_textrank %>%

mutate(to\_print = map2(doc\_id, summary, print\_summary\_textrank, n\_sentences = as.numeric(input$textrank\_n\_sentences)))

df\_textrank$to\_print

})

`` `

I host the app on Shinyapps.io, which is really easy to do from within Rstudio.

That was quite long, I’m not sure that anyone will read this blog post completely, but oh well.  
Better to put the code online, might help someone one day, that leave it to rot on my hard drive.