Crawling

Christian Kahmann

9 10 2020

Basiert auf Vorarbeiten von Andreas Niekler und Gregor Wiedemann. Wiedemann, Gregor; Niekler, Andreas (2017): Hands-on: A five day text mining course for humanists and social scientists in R. Proceedings of the 1st Workshop on Teaching NLP for Digital Humanities (Teach4DH@GSCL 2017), Berlin.

This tutorial covers how to extract and process text data from web pages or other documents for later analysis. The automated download of HTML pages is called **Crawling**. The extraction of the textual data and/or metadata (for example, article date, headlines, author names, article text) from the HTML source code (or the DOM document object model of the website) is called **Scraping**. For these tasks, we use the package "rvest". In a third exercise, we will extract text data from various formats such as PDF, DOC, DOCX and TXT files with the "readtext" package.

1. Download a single web page and extract its content
2. Extract links from a overview page and extract articles
3. Extract text data from PDF and other formats on disk

Preparation

Create a new R script (File -> New File -> R Script) named "Tutorial_2.R". In this script you will enter and execute all commands. If you want to run the complete script in RStudio, you can use Ctrl-A to select the complete source code and execute with Ctrl-Return. If you want to execute only one line, you can simply press Ctrl-Return on the respective line. If you want to execute a block of several lines, select the block and press Ctrl-Return.

Tip: Copy individual sections of the source code directly into the console (2) and run it step by step. Get familiar with the function calls included in the Help function.

First, make sure your working directory is the data directory we provided for the exercises.

```
# important option for text analysis
options(stringsAsFactors = F)
# check working directory. It should be the destination folder of the extracted
# zip file. If necessary, use `setwd("your-tutorial-folder-path")` to change it.
getwd()
```

Prepare scraping of dynamic web pages

Modern websites often do not contain the full content displayed in the browser in their corresponding source files which are served by the webserver. Instead, the browser loads additional content dynamically via javascript code contained in the original source file. To be able to scrape such content, we rely on a headless browser "phantomJS" which renders a site for a given URL for us, before we start the actual scraping, i.e. the extraction of certain identifiable elements from the rendered site.

If not done yet, please install the `webdriver` package for R and install the phantomJS headless browser. This needs to be done only once.

```
install.packages("webdriver")
library(webdriver)
install_phantomjs()
```

Now we can start an instance of PhantomJS and create a new browser session that awaits to load URLs to render the corresponding websites.

```
require(webdriver)
```

```
## Loading required package: webdriver
```

```
pjs_instance <- run_phantomjs()
pjs_session <- Session$new(port = pjs_instance$port)
```

Crawl single webpage

In a first exercise, we will download a single web page from "The Guardian" and extract text together with relevant metadata such as the article date. Let's define the URL of the article of interest and load the *rvest* package, which provides very useful functions for web crawling and scraping.

```
url <- "https://www.theguardian.com/world/2017/jun/26/angela-merkel-and-donald-trump-head-for-clash-at-
require("rvest")
```

A convenient method to download and parse a webpage provides the function `read_html` which accepts a URL as a parameter. The function downloads the page and interprets the html source code as an HTML / XML object.

Dynamic web pages

To make sure that we get the dynamically rendered HTML content of the website, we pass the original source code dowloaded from the URL to our PhantomJS session first, and the use the rendered source.

NOTICE: In case the website does not fetch or alter the to-be-scraped content dynamically, you can omit the PhantomJS webdriver and just download the the static HTML source code to retrieve the information from there. In this case, replace the following block of code with a simple call of `html_document <- read_html(url)` where the `read_html()` function downloads the page source for you.

```
# load URL to phantomJS session
pjs_session$go(url)
# retrieve the rendered source code of the page
rendered_source <- pjs_session$getSource()
# parse the dynamically rendered source code
html_document <- read_html(rendered_source)
```

Scrape information from XHTML

HTML / XML objects are a structured representation of HTML / XML source code, which allows to extract single elements (headlines e.g. `<h1>`, paragraphs `<p>`, links `<a>`, ...), their attributes (e.g. `<a href="http://...">`) or text wrapped in between elements (e.g. `<p>my text...</p>`). Elements can be extracted in XML objects with XPATH-expressions.

XPATH (see https://en.wikipedia.org/wiki/XPath) is a query language to select elements in XML-tree structures. We use it to select the headline element from the HTML page. The following xpath expression

queries for first-order-headline elements `h1`, anywhere in the tree `//` which fulfill a certain condition `[...]`, namely that the `class` attribute of the `h1` element must contain the value `content__headline`.

The next expression uses R pipe operator %>%, which takes the input from the left side of the expression and passes it on to the function ion the right side as its first argument. The result of this function is either passed onto the next function, again via %>% or it is assigned to the variable, if it is the last operation in the pipe chain. Our pipe takes the `html_document` object, passes it to the html_node function, which extracts the first node fitting the given xpath expression. The resulting node object is passed to the `html_text` function which extracts the text wrapped in the `h1`-element.

```
title_xpath <- "//h1[contains(@class, 'content__headline')]"
title_text <- html_document %>%
  html_node(xpath = title_xpath) %>%
  html_text(trim = T)
```

Let's see, what the `title_text` contains:

```
cat(title_text)
```

```
## Angela Merkel and Donald Trump head for clash at G20 summit
```

Now we modify the xpath expressions, to extract the article info, the paragraphs of the body text and the article date. Note that there are multiple paragraphs in the article. To extract not only the first, but all paragraphs we utilize the `html_nodes` function and glue the resulting single text vectors of each paragraph together with the `paste0` function.

```
intro_xpath <- "//div[contains(@class, 'content__standfirst')]//p"
intro_text <- html_document %>%
  html_node(xpath = intro_xpath) %>%
  html_text(trim = T)

cat(intro_text)
```

```
## German chancellor plans to make climate change, free trade and mass migration key themes in Hamburg,
```

```
body_xpath <- "//div[contains(@class, 'content__article-body')]//p"
body_text <- html_document %>%
  html_nodes(xpath = body_xpath) %>%
  html_text(trim = T) %>%
  paste0(collapse = "\n")
```

```
cat(body_text)
```

```
## A clash between Angela Merkel and Donald Trump appears unavoidable after Germany signalled that it w
```

```
date_xpath <- "//time"
date_object <- html_document %>%
  html_node(xpath = date_xpath) %>%
  html_attr(name = "datetime") %>%
  as.Date()

cat(format(date_object, "%Y-%m-%d"))
```

```
## 2017-06-26
```

The variables `title_text`, `intro_text`, `body_text` and `date_object` now contain the raw data for any subsequent text processing.

Follow links

Usually, we do not want download a single document, but a series of documents. In our second exercise, we want to download all Guardian articles tagged with "Angela Merkel". Instead of a tag page, we could also be interested in downloading results of a site-search engine or any other link collection. The task is always two-fold: First, we download and parse the tag overview page to extract all links to articles of interest:

```
url <- "https://www.theguardian.com/world/angela-merkel"

# go to URL, download and render page
pjs_session$go(url)
rendered_source <- pjs_session$getSource()
# parse the source code into an XML object
html_document <- read_html(rendered_source)
```

Second, we download and scrape each individual article page. For this, we extract all `href`-attributes from `a`-elements fitting a certain CSS-class. To select the right contents via XPATH-selectors, you need to investigate the HTML-structure of your specific page. Modern browsers such as Firefox and Chrome support you in that task by a function called "Inspect Element" (or similar), available through a right-click on the page element.

```
links <- html_document %>%
  html_nodes(xpath = "//div[contains(@class, 'fc-item__container')]/a") %>%
  html_attr(name = "href")
```

Now, `links` contains a list of 20 hyperlinks to single articles tagged with Angela Merkel.

```
head(links, 3)
```

```
## [1] "https://www.theguardian.com/world/2020/nov/02/global-coronavirus-report-who-chief-tedros-in-isol
## [2] "https://www.theguardian.com/commentisfree/2020/oct/29/the-guardian-view-on-a-second-wave-of-covi
## [3] "https://www.theguardian.com/politics/2020/oct/29/the-real-brexit-battle-was-democracy-v-realpoli
```

But stop! There is not only one page of links to tagged articles. If you have a look on the page in your browser, the tag overview page has several more than 60 sub pages, accessible via a paging navigator at the bottom. By clicking on the second page, we see a different URL-structure, which now contains a link to a specific paging number. We can use that format to create links to all sub pages by combining the base URL with the page numbers.

```
page_numbers <- 1:3
base_url <- "https://www.theguardian.com/world/angela-merkel?page="
paging_urls <- paste0(base_url, page_numbers)

# View first 3 urls
head(paging_urls, 3)
```

```
## [1] "https://www.theguardian.com/world/angela-merkel?page=1"
## [2] "https://www.theguardian.com/world/angela-merkel?page=2"
## [3] "https://www.theguardian.com/world/angela-merkel?page=3"
```

Now we can iterate over all URLs of tag overview pages, to collect more/all links to articles tagged with Angela Merkel. We iterate with a for-loop over all URLs and append results from each single URL to a vector of all links.

```
all_links <- NULL
for (url in paging_urls) {
  # download and parse single overview page
  pjs_session$go(url)
```

```
  rendered_source <- pjs_session$getSource()
  html_document <- read_html(rendered_source)

  # extract links to articles
  links <- html_document %>%
    html_nodes(xpath = "//div[contains(@class, 'fc-item__container')]/a") %>%
    html_attr(name = "href")

  # append links to vector of all links
  all_links <- c(all_links, links)
}
```

An effective way of programming is to encapsulate repeatedly used code in a specific function. This function then can be called with specific parameters, process something and return a result. We use this here, to encapsulate the downloading and parsing of a Guardian article given a specific URL. The code is the same as in our exercise 1 above, only that we combine the extracted texts and metadata in a data.frame and wrap the entire process in a function-Block.

```
scrape_guardian_article <- function(url) {

  pjs_session$go(url)
  rendered_source <- pjs_session$getSource()
  html_document <- read_html(rendered_source)

  title_xpath <- "//h1[contains(@class, 'content__headline')]"
  title_text <- html_document %>%
    html_node(xpath = title_xpath) %>%
    html_text(trim = T)

  intro_xpath <- "//div[contains(@class, 'content__standfirst')]//p"
  intro_text <- html_document %>%
    html_node(xpath = intro_xpath) %>%
    html_text(trim = T)

  body_xpath <- "//div[contains(@class, 'content__article-body')]//p"
  body_text <- html_document %>%
    html_nodes(xpath = body_xpath) %>%
    html_text(trim = T) %>%
    paste0(collapse = "\n")

  date_xpath <- "//time"
  date_text <- html_document %>%
    html_node(xpath = date_xpath) %>%
    html_attr(name = "datetime") %>%
    as.Date()

  article <- data.frame(
    url = url,
    date = date_text,
    title = title_text,
    body = paste0(intro_text, "\n", body_text)
  )

  return(article)
```

```
}
```

Now we can use that function `scrape_guardian_article` in any other part of our script. For instance, we can loop over each of our collected links. We use a running variable i, taking values from 1 to `length(all_links)` to access the single links in `all_links` and write some progress output.

```
all_articles <- data.frame()
for (i in 1:length(all_links)) {
  cat("Downloading", i, "of", length(all_links), "URL:", all_links[i], "\n")
  article <- scrape_guardian_article(all_links[i])
  # Append current article data.frame to the data.frame of all articles
  all_articles <- rbind(all_articles, article)
}
```

```
## Downloading 1 of 60 URL: https://www.theguardian.com/world/2020/nov/02/global-coronavirus-report-who
## Downloading 2 of 60 URL: https://www.theguardian.com/commentisfree/2020/oct/29/the-guardian-view-on-a
## Downloading 3 of 60 URL: https://www.theguardian.com/politics/2020/oct/29/the-real-brexit-battle-was
```

```
# View first articles
head(all_articles, 3)

# Write articles to disk
write.csv2(all_articles, file = "data/guardian_merkel.csv")
```

The last command write the extracted articles to a CSV-file in the data directory for any later use.