It has been given an introduction on how to query web APIs – specifically the Twitter API – and automated data extraction from websites (i.e. *web scraping*). I showed an example that combined both of these techniques for the goal of getting data about the Twitter activities of members of the current (19th) German Bundestag, which is the federal German parliament. The focus was especially on the question “who follows who” on Twitter. There was a thought it’s a nice little project showing how to use the Twitter API, do web scraping, combine the collected data and do some exploratory network analysis – all within the R environment. So It was decided to polish the code a little bit, and wrote two blog posts. The first part, i.e. this part, is all about getting the data.

**Data sources**

In order to explore the Twitter activities of members of the German Bundestag we need to know their Twitter account names (aka “Twitter handles”) if they use one for their work as representatives. The [Open Data Portal of the German Bundestag](https://www.bundestag.de/services/opendata) provides an XML file containing basic information about all representatives, i.e. deputies, since 1949 like day and place of birth, membership periods and roles in the Bundestag etc. Unfortunately, the dataset does not contain links to social media profiles (I would have been surprised if it did), although the individual [profile pages](https://www.bundestag.de/abgeordnete) do contain such links. We could implement a web scraper to extract these, but because scraping should always be the last resort, I first looked at another website.

The platform [Abgeordnetenwatch.de](https://www.abgeordnetenwatch.de/) (“parliament watch”) is an independent, non-partisan and non-profit site that allows citizens in Germany to publicly ask questions to their representatives in the federal parliament, the European parliament and state parliaments. Representatives have a profile on this website and may publish links to their social media sites there. Abgeordnetenwatch also provides a public [API](https://www.abgeordnetenwatch.de/api) (denoted as “beta” state) which allows to fetch data for all profiles of a specific parliament. This includes also some interesting variables like the rate of answers to citizens’ questions, but these are not of interest to us in this case.

Anyway, this API also doesn’t provide the social media links we’re interested in. So this means we will have to implement a webscraper either for the deputy profiles on the Bundestag website or on Abgeordnetenwatch. I decided to do it for the latter one, because 1) the profiles on the Bundestag website use a lot of JavaScript which makes it harder to implement a scraper than for the mostly static profile pages on Abgeordnetenwatch and 2) Abgeordnetenwatch doesn’t prohibit scraping their website according to their terms of use – which is something you should **always** check before trying to implement a web scraper. Still, we have to keep in mind that not all deputies may include a link to their Twitter page on their Abgeordnetenwatch profile. For example, SPD member Andrea Nahles has a [Twitter account,](https://twitter.com/AndreaNahlesSPD) but it is not listed on her [profile page](https://www.abgeordnetenwatch.de/profile/andrea-nahles). So this solution, as most web scraping solutions, isn’t perfect but it definitely saves a lot of time. If we wanted to make sure to get the Twitter pages of *all* deputies, we would later manually check only those which didn’t include such a link on their profile (and try to google their Twitter account).

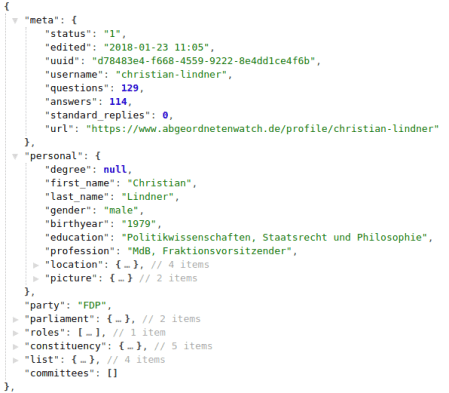
And by the way: yes, I would usually simply ask Abgeordnetenwatch if they, as an organization devoted to transparency, could provide the required data before writing a web scraper. But since I looked for a good example for a tutorial on web scraping, I deliberately decided against that.

**Getting the data about the representatives**

For the rest of the text, I will focus on some code parts and won’t provide full implementations.

First of all, we make use of the profile information of the members of the current Bundestag provided by the Abgeordnetenwatch API. With this data, we can iterate through the profiles and extract the social media links from each profile page directly, instead of implementing a web crawler that extracts the profile links from the website first.

Getting the profile data is as easy as downloading the respective JSON file at <https://www.abgeordnetenwatch.de/api/parliament/bundestag/deputies.json>. To understand the contents of this file, it’s good to explore it with a JSON formatter. We can see an example output below:

[](https://i2.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2019/07/abgwatch-profile-json.png)

This file can be parsed with fromJSON() from the jsonlite package. We’re interested in the URL to the profile pages and in the unique identifier for each deputy, because this identifier can later be user for combining the collected data. Both is inside the profiles$meta dataframe of the nested list that represents the JSON structure:

library(jsonlite)

library(dplyr)

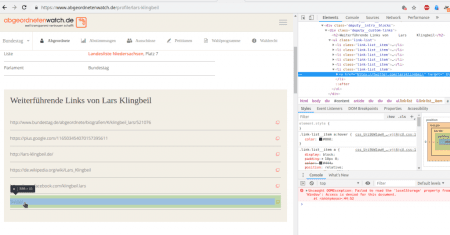
deputies <- fromJSON('data/deputies.json')

prof\_urls <- deputies$profiles$meta %>% select(uuid, url)

This will give us the identifier and profile URLs for all 718 members of the current Bundestag.

**Scraping the profile pages**

From each of these profile pages, we now need to extract the links to social media sites. To do so, we need to identify the elements in the HTML structure of the website which contain our information of interest. An example profile page with links to different social media sites in the “Weiterführende Links” (“further links”) section would be [this one](https://www.abgeordnetenwatch.de/profile/lars-klingbeil). We can use the “inspect” tool in Chrome or other browsers to have a look at the HTML structure (right click on a website element and select “Inspect” to activate this tool). This helps us to find a “path” through the HTML structure to our elements of interest:

[](https://i1.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2019/07/abgwatch-profile-inspect.png)

The crucial information for the “path” to the elements of interest, which is the links specified by an ... tag, is:

* a

container with class attribute "deputy\_\_custom-links"

* inside this, a list tag

with class attribute "link-list"

* inside this, list elements
* inside these, links tags from which we need to extract the href attribute (the URL the link points to)

The package [rvest](https://cran.r-project.org/web/packages/rvest/index.html) allows to extract the required data by filtering HTML elements according to the above “path” which can be translated into a CSS selector. Let’s use rvest for the above sample profile. At first we load the package and fetch the HTML from the profile page:

library(rvest)

html <- read\_html('https://www.abgeordnetenwatch.de/profile/lars-klingbeil')

Now, we filter the HTML by applying a CSS selector that points to the links (the elements) inside the “further links” section:

(links <- html\_nodes(html, 'div.deputy\_\_custom-links ul.link-list li a'))

## {xml\_nodeset (6)}

## [1]

From these links, we extract the value of the href attribute and voilà, we get the URLs to the (social media) links on this profile:

(urls <- html\_attr(links, 'href'))

## [1] "http://www.bundestag.de/abgeordnete/biografien/K/kli...

## [2] "https://plus.google.com/116503454070157395611"

## ...

## [6] "https://twitter.com/larsklingbeil"

What is left is to put this in a function that uses a profile URL as argument and apply this function to all profile URLs from the “deputies” dataset. However, we should also pay attention to the [robots.txt file](https://www.robotstxt.org/robotstxt.html) of Abgeordnetenwatch. It contains a [line](https://www.abgeordnetenwatch.de/robots.txt) specifying the minimum delay of 10 seconds of subsequent automated requests to the web server (Crawl-delay: 10). We should comply with this and only fetch and parse a profile page every 10 seconds by setting a delay with Sys.sleep(10) inside the profile scraper function (given below). This means the whole web scraping process will take about 2h for all 718 profiles.

Scrapper.R

|  |
| --- |
| library(jsonlite) |
|  | library(rvest) |
|  | library(dplyr) |
|  |  |
|  | # data from members of the 19th German Bundestag |
|  | # obtained from https://www.abgeordnetenwatch.de/api/parliament/bundestag/deputies.json |
|  | deputies <- fromJSON('data/deputies\_20190702.json') |
|  |  |
|  | sleep\_sec <- 10 # according to robots.txt |
|  |  |
|  | # get profile URLs |
|  | n\_profiles <- nrow(deputies$profiles) |
|  | print(paste('Num. profiles:', n\_profiles)) |
|  |  |
|  | prof\_urls <- deputies$profiles$meta %>% select(uuid, url) |
|  |  |
|  | #prof\_urls <- prof\_urls %>% head(10) |
|  |  |
|  | # function to fetch HTML from profile page and extract "further links" section |
|  | # ("Weiterführende Links von ...") on the page |
|  | fetch\_urls <- function(profile\_row) { |
|  | print(paste('fetching profile page at', profile\_row$url)) |
|  |  |
|  | # wait and fetch HTML |
|  | Sys.sleep(sleep\_sec) |
|  | html <- read\_html(profile\_row$url) |
|  |  |
|  | # extract links |
|  | links <- html\_nodes(html, 'div.deputy\_\_custom-links ul.link-list li a') |
|  | urls <- html\_attr(links, 'href') |
|  |  |
|  | if (length(urls) == 0) { |
|  | urls <- NA |
|  | } |
|  |  |
|  | # return data frame for this deputy which will be concatenated to a single data frame |
|  | # of all deputies |
|  | data.frame(profile\_row, custom\_links = urls, stringsAsFactors = FALSE) |
|  | } |
|  |  |
|  | # apply fetch\_urls to each profile |
|  | prof\_urls\_complete <- prof\_urls %>% rowwise() %>% do(fetch\_urls(.)) |
|  |  |
|  | # save result |
|  | write.csv(prof\_urls\_complete, 'data/deputies\_custom\_links\_20190702.csv', row.names = FALSE) |

**Extracting the Twitter handles**

Once this is done, we have a dataset that I will call dep\_links with the unique IDs of the deputies and for each of these one or more extracted URLs, which looks like this:

uuid custom\_links

d623efb5-7afc-... https://twitter.com/MartinSchulz

a96583f1-e2b0-... http://www.fabiodemasi.de

a96583f1-e2b0-... http://www.facebook.com/fabio.d.masi

a96583f1-e2b0-... https://www.youtube.com/channel/UCf\_...

a96583f1-e2b0-... https://twitter.com/FabioDeMasi

Not all of these URLs point to the respective Twitter profiles but to Facebook pages, personal websites or other sites a deputy wanted to share. Furthermore, we need to extract the Twitter handle from the URLs that point to Twitter profile pages, because we later need those handles when working with the Twitter API. So in the next step, we will have to do two things: 1) filter the dataset to find matches to Twitter URLs, e.g. https://twitter.com/exampleuser ; 2) extract the Twitter handle from these URLs, i.e. the part after the last slash which would be exampleuser here.

We can do both things at once with a *regular expression*. By this, we define a pattern to which an URL must match, and we can also extract parts of that pattern that were matched, e.g. the section after the last slash. Before we think of such a pattern, we need to think about how links to Twitter pages might look like. For example, https://twitter.com/exampleuser is a valid Twitter page, but https://www.twitter.com/exampleuser and http://twitter.com/exampleuser/ (note the http without *s* and the trailing slash) are too. Links to Twitter pages appear in all these forms and our pattern should detect all of them. We could write the pattern as http[s]://[www.]twitter.com/[/] where optional parts are given in [brackets] and the part we want to extract is . We can translate this into the following regular expression pattern:

pttrn\_twitter <- '^https?://(www\.)?twitter\.com/([A-Za-z0-9\_-]+)/?

Note how optional parts are denoted by a question mark after a character or a string sequence in parentheses. The part /([A-Za-z0-9\_-]+) describes the Twitter handle as a string made from a set of characters (all characters that are valid for a Twitter handle) after twitter.com/. We can later extract the value inside the parenthesis because it is denoted as a *regular expression group*. We can apply this regular expression to the custom\_links column in the dataset:

matches <- regexec(pttrn\_twitter, dep\_links$custom\_links)

matches is now a list of the same length as the custom\_links vector and contains information about whether the pattern matched and also where the regular expression groups matched. We can extract this information, using the regmatches() function. For example, the first link in the above dataset, https://twitter.com/MartinSchulz, clearly matches the pattern and we should get the values from the regular expression groups:

regmatches(dep\_links$custom\_links, matches)[[1]]

## [1] "https://twitter.com/MartinSchulz" "" "MartinSchulz"

Note how this returns a vector with three strings: The first is always the full matched string, the second is the value from first regular expression group (defined as (www\.) in the pattern – hence it is an empty string here because there is no “www.” prefix in the URL) and the third is the value from the second regular expression group, which was defined as the sub-string that represents the Twitter handle.

The second link in the dataset, http://www.fabiodemasi.de, doesn’t match the pattern, hence we get an empty result for that:

regmatches(dep\_links$custom\_links, matches)[[2]]

## character(0)

We can use sapply to apply the Twitter handle extraction to the full dataset. The full code is then:

pttrn\_twitter <- '^https?://(www\.)?twitter\.com/([A-Za-z0-9\_-]+)/?'

matches <- regexec(pttrn\_twitter, dep\_links$custom\_links)

# if there's a match, take component 3 (the Twitter handle)

dep\_links$twitter\_name <- sapply(

regmatches(dep\_links$custom\_links, matches),

function(s) { if (length(s) == 3) s[3] else NA })

All that is left is to join the resulting dataset with the original deputies dataset from Abgeordnetenwatch.de using the unique identifier (uuid) so that we have a dataset with information about all deputies and their Twitter handle if they had a link to their Twitter page on their Abgeordnetenwatch profile page. See the full code in twitter\_profiles.R.

Twitter\_Profiles.R

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | library(dplyr) |
|  | library(tibble) |
|  | library(jsonlite) |
|  |  |
|  | # ---- load and process social media links ---- |
|  |  |
|  | # load links obtained from scraping |
|  | dep\_links <- distinct(read.csv('data/deputies\_custom\_links\_20190702.csv', stringsAsFactors = FALSE)) |
|  |  |
|  | # extract Twitter handle from Twitter URLs |
|  | # matches the following strings where optional parts are given in [brackets]: |
|  | # http[s]://[www.]twitter.com/<NAME>[/] |
|  | pttrn\_twitter <- '^https?://(www\\.)?twitter\\.com/([A-Za-z0-9\_-]+)/?' |
|  | matches <- regexec(pttrn\_twitter, dep\_links$custom\_links) |
|  | dep\_links$twitter\_name <- sapply(regmatches(dep\_links$custom\_links, matches), |
|  | function(s) { if (length(s) == 3) s[3] else NA }) # if there's a match, take component 3 (the Twitter handle) |
|  |  |
|  | # have a look at a sample |
|  | sample\_n(dep\_links[c('custom\_links', 'twitter\_name')], 10) |
|  |  |
|  | # filter to get only the rows with extracted Twitter handles |
|  | dep\_twitter <- dep\_links[!is.na(dep\_links$twitter\_name), ] |
|  | head(dep\_twitter) |
|  |  |
|  | # corrections for wrong links to Twitter (affects two accounts) |
|  | # e.g. "https://twitter.com/twitter.com/MartinaRenner" |
|  | pttrn\_twitter\_correct <- '/([A-Za-z0-9\_-]+)/?$' |
|  | matches <- regexec(pttrn\_twitter\_correct, dep\_twitter[dep\_twitter$twitter\_name == 'twitter', 'custom\_links']) |
|  | dep\_twitter[dep\_twitter$twitter\_name == 'twitter', ]$twitter\_name <- sapply( |
|  | regmatches(dep\_twitter[dep\_twitter$twitter\_name == 'twitter', 'custom\_links'], matches), |
|  | function(s) { s[2] }) |
|  |  |
|  | # drop extremely short Twitter handles (invalid -- affects one account) |
|  | dep\_twitter <- dep\_twitter[nchar(dep\_twitter$twitter\_name) > 3, ] |
|  | print(paste('Anzahl Twitternamen:', nrow(dep\_twitter))) |
|  |  |
|  | # further clean up |
|  | dep\_twitter <- dep\_twitter %>% select(-custom\_links) %>% |
|  | mutate(twitter\_name = tolower(twitter\_name)) %>% # all to lowercase |
|  | filter(!is.na(twitter\_name) & twitter\_name != 'search') %>% |
|  | distinct() # remove duplicates |
|  |  |
|  |  |
|  | # ---- join with deputies data ---- |
|  |  |
|  | # load deputies data |
|  | deputies <- fromJSON('data/deputies\_20190702.json', flatten = TRUE) |
|  |  |
|  | # select important columns |
|  | dep <- as\_tibble(deputies$profiles) %>% select(starts\_with('meta'), starts\_with('personal'), 'party', |
|  | starts\_with('parliament')) |
|  |  |
|  | orig\_nrows <- nrow(dep) |
|  |  |
|  | # join via abgeordnetenwatch.de profile UUID |
|  | dep <- dep %>% left\_join(dep\_twitter %>% select(uuid, twitter\_name), by = c('meta.uuid' = 'uuid')) |
|  |  |
|  | stopifnot(orig\_nrows == nrow(dep)) |
|  |  |
|  | # have a glimpse |
|  | head(dep %>% select(uuid = meta.uuid, personal.first\_name, personal.last\_name, party, twitter\_name)) |
|  |  |
|  | # save result |
|  | write.csv(dep, 'data/deputies\_twitter\_20190702.csv', row.names = FALSE) |

**Fetching the list of “friends” from the Twitter API**

From the 718 members of the Bundestag, 396 had a link to their Twitter page on their Abgeordnetenwatch.de profile when I retrieved the data on July 2 2019. From these links we extracted the Twitter handles, which we can now use to get more information about these accounts by querying the Twitter API. We’re especially interested in the network of followers between deputies, i.e. which deputy follows which of her or his colleagues?

In Twitter’s API terminology, there is a distinction between *friends* and *followers*. The former is the set of Twitter users that a certain Twitter user actively *follows*, i.e. the users that appear in her or his “following” list. The latter is the set of Twitter users that follow a certain Twitter user, i.e. the users that appear in her or his “followers” list. You see that this describes the asymmetric nature of connections between users on Twitter: not everybody you follow automatically follows you back.

Since we want to know *who follows who*, we’re interested in the “friends” of each deputy. The package rtweet provides comfortable functions to access the Twitter API. Before using it, you will need to create a Twitter account (if you don’t have one already) and set up a Twitter app on apps.twitter.com. There you can get the four authentication tokens that you need to pass to the create\_token() function. The documentation for rtweet explains the whole procedure.

For our goal we can use the get\_friends() function which accepts a character vector of Twitter handles and will return a dataframe with the Twitter handles for which the request was made and the respective Twitter user IDs for her or his friends. Let’s try it out by fetching the friends user IDs for @WZB\_Berlin and @DIW\_Berlin. Make sure you’ve loaded rtweet before (library(rtweet)) and authenticated with create\_token().

get\_friends(c('WZB\_Berlin', 'DIW\_Berlin'))

## user user\_id

##

## 1 WZB\_Berlin 831148275038306304

## 2 WZB\_Berlin 45557114

## 3 WZB\_Berlin 2585450642

## ...

## 4 DIW\_Berlin 95394865

## 5 DIW\_Berlin 19709133

## 6 DIW\_Berlin 5776022

We can apply this function to the Twitter handles of all deputies. We should make sure to add the parameter retryonratelimit = TRUE to get\_friends(), though. The Twitter API employs *rate limiting*, i.e. it limits the amount of requests that can be made to the API per a certain time frame. See the [API documentation](https://developer.twitter.com/en/docs/basics/rate-limiting) for details. Setting retryonratelimit to TRUE makes sure that whenever we hit a rate limit while querying the API, rtweet will pause for a certain time and then try to continue.fetch\_friends.R contains the whole code for retrieving friends IDs.

Fetch\_friends.R

|  |
| --- |
| library(dplyr) |
|  | library(rtweet) |
|  |  |
|  | source('twitterkeys.R') # contains access tokens for Twitter API |
|  |  |
|  | # ---- load deputies data with their Twitter handle ---- |
|  |  |
|  | dep <- read.csv('data/deputies\_twitter\_20190702.csv', |
|  | stringsAsFactors = FALSE, |
|  | colClasses = c(personal.location.postal\_code = "character")) |
|  |  |
|  | twitternames <- filter(dep, !is.na(twitter\_name)) %>% pull(twitter\_name) %>% unique() |
|  | n\_twitternames <- length(twitternames) |
|  |  |
|  | # ---- fetch list of Twitter friends ---- |
|  |  |
|  | # get authentication token for Twitter API |
|  | token <- create\_token( |
|  | app = twitter\_app, |
|  | consumer\_key = consumer\_key, |
|  | consumer\_secret = consumer\_secret, |
|  | access\_token = access\_token, |
|  | access\_secret = access\_secret) |
|  |  |
|  | print(paste('will fetch list of friends for', n\_twitternames, 'users')) |
|  |  |
|  | # this will return a data frame consisting of column "user" (the deputy Twitter handle) and |
|  | # "user\_id", which is a Twitter user ID of a friend of the respective "user" |
|  | # we will need to use "lookup\_users()" in order to get information such as the Twitter name |
|  | # about the friends user IDs |
|  | friends <- get\_friends(twitternames, retryonratelimit = TRUE) |
|  | friends$fetch\_friends\_timestamp <- Sys.time() # add a timestamp for the collection time |
|  |  |
|  | head(friends) |
|  |  |
|  | # save to file |
|  | saveRDS(friends, 'data/deputies\_twitter\_friends\_tmp\_20190702.RDS') |

The Twitter user IDs of the friends of each deputy alone don’t tell us much. It would be nice to have some additional information for each of these IDs, especially the Twitter handle because this is the information that we need in order to later create a network of deputies on Twitter.

The dataset of friend user IDs per deputy Twitter handle, which I will call friends, contains more than 340,000 observations. Of course, there are overlaps of friend IDs between deputies (several deputies may follow the same Twitter user). So in order to reduce the amount of queries to the Twitter API, we should create a vector of unique friend IDs at first:

friends\_ids <- unique(friends$user\_id)

This leaves about 100,000 unique user IDs. The documentation says that it retrieves this data for up to 90,000 user IDs at once, so we could create two chunks of IDs and feed those to lookup\_users(). However, in practice I noticed that this doesn’t work and the queries will fail after a while because of rate limiting. I think that Twitter changed the rate limits for this API call, but the change is not reflected in the rtweet package. At the moment the [Twitter documentation](https://developer.twitter.com/en/docs/basics/rate-limits) says you can make 300 requests with 100 IDs each per 15 minute time frame. So this means we will have to divide the >100,000 user IDs in chunks of 100 each, make 300 subsequent requests with lookup\_users() until we wait for at least 15 minutes. Then we continue with this process until we fetched data for all user IDs. Whenever an error occurs (because of rate limiting, lost connection, etc.) we should re-try for a certain number of trials. This is implemented in lookup\_friends.R.

Lookup\_Friends.R

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

|  |
| --- |
| library(dplyr) |
|  | library(rtweet) |
|  |  |
|  | source('twitterkeys.R') |
|  |  |
|  | # ---- load the data with the friends user IDs ---- |
|  |  |
|  | friends <- readRDS('data/deputies\_twitter\_friends\_tmp\_20190702.RDS') |
|  |  |
|  | friends\_ids <- unique(friends$user\_id) |
|  | #friends\_ids <- friends\_ids[1:1000] # subset for testing |
|  | n\_friends\_ids <- length(friends\_ids) |
|  |  |
|  | # ---- look up information about each friend user ID ---- |
|  |  |
|  | n\_retries <- 5 # maximum number of \*subsequent\* retries when an API call failed |
|  | sleep\_sec <- 16 \* 60 # 15 minutes is the time window for rate limit reset; add a little time buffer of 1 min. |
|  | chunksize <- 100 # number of user IDs per request; the docs say 300 requests with 100 IDs each per 15 min. |
|  | n\_max\_requests <- 280 # max. number of requests within a 15 min. time frame; we stay a bit below the 300 requests threshold |
|  | chunk\_idx <- 0 # current chunk index |
|  | cur\_retry <- 0 # current number of retries; is reset to 0 once a successful API call was made |
|  | friendsdata <- tibble() # collected data |
|  |  |
|  | print('fetching data from Twitter API...') |
|  |  |
|  | # get authentication token for Twitter API |
|  | token <- create\_token( |
|  | app = twitter\_app, |
|  | consumer\_key = consumer\_key, |
|  | consumer\_secret = consumer\_secret, |
|  | access\_token = access\_token, |
|  | access\_secret = access\_secret) |
|  |  |
|  | request\_i <- 0 # current number of requests |
|  |  |
|  | # repeat API requests until all data was collected or too many retries happened due to request failures |
|  | while(TRUE) { |
|  | # get chunk of friends IDs |
|  | chunk\_start <- chunk\_idx \* chunksize + 1 |
|  | chunk\_end <- min(c((chunk\_idx + 1) \* chunksize, n\_friends\_ids)) |
|  | friends\_ids\_chunk <- friends\_ids[chunk\_start:chunk\_end] |
|  | print(sprintf('fetching data for friends IDs in range [%d, %d] (%d ids)', |
|  | chunk\_start, chunk\_end, length(friends\_ids\_chunk))) |
|  |  |
|  | # make an API request for user ID lookup |
|  | # if it successes, add the data to the "friendsdata" data frame set "success" to TRUE, |
|  | # else do not add data and set "success" to FALSE |
|  | success <- tryCatch({ |
|  | request\_i <- request\_i + 1 |
|  | friendsdata\_chunk <- lookup\_users(friends\_ids\_chunk) |
|  | friendsdata <- bind\_rows(friendsdata, friendsdata\_chunk) |
|  | TRUE |
|  | }, error = function(cond) { |
|  | FALSE |
|  | }) |
|  |  |
|  | if (success) { # on success |
|  | cur\_retry <- 0 # reset number of retries |
|  |  |
|  | # check if we collected data for all IDs |
|  | if (chunk\_start + chunksize >= n\_friends\_ids) { |
|  | break() |
|  | } |
|  | } else { # on failure |
|  | # increment the number of retries |
|  | cur\_retry <- cur\_retry + 1 |
|  |  |
|  | # check if number of retries reached maximum |
|  | if (cur\_retry >= n\_retries) { |
|  | print(sprintf('failed after %d retries', cur\_retry)) |
|  | break() |
|  | } |
|  |  |
|  | print(sprintf('will advance with retry %d', cur\_retry)) |
|  | } |
|  |  |
|  | if (request\_i %% n\_max\_requests == 0 || !success) { # wait after max. num. requests or when no success |
|  | print(sprintf('waiting for %d sec.', sleep\_sec)) |
|  | Sys.sleep(sleep\_sec) |
|  | } |
|  |  |
|  | if (success) { # if no success, retry with same chunk, otherwise increment chunk index |
|  | chunk\_idx <- chunk\_idx + 1 |
|  | } |
|  | } |
|  |  |
|  | # ---- process collected friends data ---- |
|  |  |
|  | n\_fetched <- sum(!is.na(friendsdata$screen\_name)) |
|  | print(sprintf('got data for %d out of %d unique friends accounts', n\_fetched, n\_friends\_ids)) |
|  |  |
|  | friendsdata$fetch\_friendsdata\_timestamp <- Sys.time() # add timestamp |
|  |  |
|  | # join friends user data by user ID |
|  | print('joining data...') |
|  | friendsfull <- left\_join(friends, friendsdata, by = 'user\_id') |
|  | n\_matched <- sum(!is.na(friendsfull$screen\_name)) |
|  | print(sprintf('matching successful for %d out of %d rows', n\_matched, nrow(friends))) |
|  |  |
|  | # save as RDS |
|  | saveRDS(friendsfull, 'data/deputies\_twitter\_friends\_full\_20190702.RDS') |
|  |  |
|  | # friendsfull\_csvfriendly <- select\_if(friendsfull, function(x) all(!is.list(x))) |
|  | # write.csv(friendsfull\_csvfriendly, 'data/deputies\_twitter\_friends.csv', row.names = FALSE) |

In the end, we have a dataframe that maps deputy Twitter handles to their friends, i.e. the accounts they follow, along with lots of additional information about the friend accounts. I’m showing a small sample of the more than 340,000 rows, including only the deputy Twitter handle user, the friend Twitter data screen\_name, name and location:

user screen\_name name location

martinschulz dosomething4eu Do something for Europe ""

martinschulz machiavellipod Machiavelli – Rap Köln

martinschulz AndreaNahlesSPD Andrea Nahles ""

...

cem\_oezdemir EuropeElects Europe Elects ""

cem\_oezdemir MirjamWenzel Mirjam Wenzel Frankfurt...

...

This is all the information that we need. We have the Twitter handles of the deputies that published them on their Abgeordnetenwatch.de profile, their friends (i.e. the accounts they follow) and all necessary information about these friends. We also have the data about deputies from Abgeordnetenwatch.de including the Twitter handle that we extracted. With this we can combine both datasets (the deputies data and the Twitter friends data).