Documentation: collecto.R

Table of Contents

The Script

The R script documented here has a modular structure. It is divided into 2 sections that handle loading the packages necessary for the process and exporting the aggregated data into usable formats in the end. The remaining sections handle one specific data source each (eg.: Twitter, Mastodon, Reddit). While the Package-Section is obviously necessary for the remaining sections (depending on which ones you actually want to use) as well as the Export-Section for actually using the data in other applications, scripts or by other people, you can cherry-pick between the Datasource-Sections. These can be used independently and in no particular order to another. Keep that in mind, if you only want to analyze *X*.

As a site-note, the script is written to keep the data collected as anonymous as possible, however because we deal with a rather small sample and because of the nature of social media, it is in most cases still possible to track down each specific user in the resulting data. As we only access public postings, it is safe to assume, that people want their posts to be seen anyways, so it is not as problematic as it may seem. Nevertheless, we should still treat the data with much care and do not leak meta-data if possible.

Packages

As of writing this script and its documentation, two platform specific and two general scraper-packages are being used:

- rtweet (Version 0.6.0)
- curl (Version 3.1)
- rjson (Version 0.2.15)
- RedditExtractoR (Version 2.0.2)

The rtweet package

rtweet has a rather extensive documentation as well as "in-R-manuals". Simply enter `??rtweet` into the R console or look up a specific function with `?function`, replacing `function` with its actual name. It is the successor of the previously used twittR package with a lot of improvements and fewer restrictions regarding the Twitter-API. rtweet has several useful functions to scrape Twitter-Data, most of which however apply to the Twitter account in use - which in our case is not necessary. The Twitter-Section uses only two functions, which will be discussed individually, later on:

```
create_token()  # authentication
search_tweets() # searching Twitter for a particular string
```

As a site-note; I had to install the httr-package - a dependency of rtweet - from the Repositories of my distribution of choice, as the one provided by CRAN would not compile for some reason. So if you run into a similar issue, look for something like `r-cran-httr` in your packagemanager.

The curl and rjson packages

The good thing about Mastodon is, that searches are not restricted to a single Mastodon-Instance or to Mastodon at all. If your Instance has enough outbound connections (so make sure you chose a very active and inter-communicative one), you are able to not only search Mastodon-Instances, but also GNUsocial, Pump.io and other compatible Social Media instances. Previously, we used a specialized package for scraping the Fediverse, simply called mastodon, however it proved to be unreliable, poorly documented and probably even unmaintained. Luckily, Mastodon as an opensource platform also has a really open API we can access with simple tools like curl and rjson. Specifically, we use following functions of the `curl` package:

```
curl_fetch_memory() # import HTTP headers from the API-page
parse_headers()     # extracts responses from the HTTP header
```

This will generate an output in a JSON format, which we can transform to a `list()` item with a function from the `rjson` package:

```
fromJSON() # transform JSON to a list() item
```

The RedditExtractoR package

RedditExtractoR has a rather extensive documentation but no general "in-R-manual". You can however look up a specific function within the package by entering `?function`into the R-promt, replacing `function` with its actual name. RedditExtractoR has several useful function to scrape Reddit-Posts or create fancy graphs from them. In our case, we only need two very basic functions that will be discussed in the Reddit-Section later on:

```
reddit_urls()         # searching Reddit for a particular string
reddit_content()      # scrape data of an indicidual post
```

You may have noticed, that there is no "authenticate" command within this package. As of now, the Reddit-API does not require authentication, as all posts are for general consumption anyways. This may or may not change in the future, so keep an eye on this.

Twitter

Authenticate

As the package in use here needs access to the Twitter-API, what we first need are the "Consumer Key", "Consumer Secret" and our "App Name", all of which you can order from apps.twitter.com. Of course, you need a Twitter-Account for this (staff may ask for the FSFE's Account).

The authentication can be done in two ways:

1. via manual input. The R-Console will prompt you to enter the credentials by typing them in. Going this route will exclude the option to run the script automatically.

2. via a plain text file with the saved credentials. This `.txt` file has a very specific structure which you have to follow. You can find an example file in the examples folder. Going this route can potentially be a security risk for the twitter-account in use, as the `.txt` file is stored in plain text. The problem can be mitigated if your harddrive is encrypted. *It may also be possible to implement decryption of a file via GNUPG with the* `system()` *command. However, this has not been implemented in this script (yet).*

The first line of the credential-file contains the *labels*. These have to be in the same order as the *credentials* themselves in the line below. The *labels* as well as the *credentials* are each separated by a single semi-colon `;`. Storing the credentials in plain text surely is not optimal, but the easiest way to get the information into our R-Session. This should not be too critical, if your disk is encrypted.

Next, we create the oauth token with `create_token()`.The oauth token can not only be used to scrape information from Twitter, it also grants write-access, so can be used to manipulate the affiliated Twitter-Account or interact with Twitter in any other way.

The function used to authenticate takes the consumer-key and consumer-secret as well as the name of the app, which you registered on the twitter-developer page before, as arguments, which in this script are stored in the `twitter_consumerkey twitter_consumerpri twitter_appname` variables:

```
twitter_token <- create_token(app = twitter_appname,
                              consumer_key = twitter_consumerkey,
                              consumer_secret = twitter_consumerpri)
```

Scraping Tweets

Once we have an oauth token, we can already start looking for desired tweets to collect. For this we use the `search_tweets()` function. All functions in the `rtweet` package access the token via environment variables. So make sure to create it before use and don't override it, afterwards. What arguments we need to forward to the function are:

- the string to search for, in this case `ilovefs`. This will not only include things like "ilovefs18", "ilovefs2018", "ILoveFS", etc but also hashtags like "#ilovefs"
- the maximum number of tweets to be aggregated. This number is only useful for search-terms that get a lot of coverage on twitter (eg.: trending hashtags). For our purpose, we can safely set it to a number that is much higher than the anticipated participation in the campaign, like `9999999999` so we get ALL tweets containing our specified string
- whether we want to include retweets in your data as well (we do not, in this case, so set it to `FALSE`)

We save the result of this command in the variable `twitter_tw`. The resulting code is:

```
twitter_tw <- search_tweets(q = "#ilovefs",
                            n = 9999,
                            include_rts = FALSE)
```

Stripping out data

Most of the resulting data can be extracted with simple assignment, only a few characteristics are organized within `list()` items in the `data.frame()`. The structure of the dataset is as follows:

```
twitter_tw
  |
  |- ...
  |- text = "Yay! #ilovefs", "Today is #ilovefs, celebrate!", ...
  |- ...
  |- screen_name = "user123", "fsfe", ...
  |- ...
  |- source = "Twidere", "Tweetdeck", ...
```

```
|- ...
|- favorite_count = 8, 11, ...
|- retweet_count = 2, 7, ...
|- ...
|- lang = "en", "en", ...
|- ...
|- created_at = "14-02-2018" 10:01 CET", "14-02-2018 10:01 CET", ...
|- ...
|- urls_expanded_url
|    |- NA, "https://ilovefs.org", ...
|    '- ...
|
|- media_expanded_url
|    |- NA, NA, ...
|    '- ...
|
'- ...
```

The inconvenience about the `list()` structure stems from that we need to use a for-loop in order to run through each `list()` item and extract its variables individually.

For the sake of keeping this short, this documentation only explains the extraction of a single argument, namely the Client used to post a Tweet and the extracting of one of the items in the `list()`, namely the media-URL. All other information are scraped in a very similar fashion.

For the media-URL, firstly we create a new, empty `vector()` item called `murl` with the `vector()` command. Usually you do not have to pre-define empty vectors in R, but it will be created automatically if you assign it a value, has we've done before multiple times. You only need to pre-define it, if you want to address a specific *location* in that vector, say skipping the first value and filling in the second. We do it like this here, as we want the resulting `vector()` item to have the same order as the original dataset. In theory, you could also use the combine command `c()`, however `vector()` gives you the option to pre-define the mode (numeric, character, factor, ...) as well as the length of the variable. The variable `twitter_number` that we create before that, simply contains the number of all tweets, so we know how long the `murl` vector has to be:

```
twitter_number <- length(twitter_tw$text)
...
murl <- vector(mode = "character", length = twitter_number)
```

The for-loop has to count up from 1 to as many tweets as we scraped. We already saved this value into the `twitter_number` variable. So if we scraped four Tweets, the for-loop has to count `1 2 3 4`:

```
for(i in c(1:twitter_number)){
  ...
}
```

Next, we simply assign the first value of the according `list()` item to our pre-defined vector. To be precise, we assign it to the current location in the vector. You could also check first, if this value exists in the first place, however the `retweet` package sets `NA`, meaning "Not Available", if that value is missing, which is fine for our purpose:

```
murl[i] <- twitter_tw$media_expanded_url[[i]][1]
```

All combined, this looks like this:

```
twitter_number <- length(twitter_tw$text)
clnt <- twitter_tw$source
...

murl <- vector(mode = "character", length = twitter_number)
```

```
for(i in 1:twitter_number){
  ...
  murl[i] <- twitter_tw$media_expanded_url[[i]][1]
}
```

Sometimes, as it is the case with `fdat` the "full date" variable, the extracted string contains things that we do not need or want. So we use regex to get rid of it.

If you are not familiar with regex, I highly recommend regexr.com to learn how to use it. It also contains a nifty cheat-sheet.

```
time <- sub(pattern = ".* ", x = fdat, replace = "")
time <- gsub(pattern = ":", x = time, replace = "")
date <- sub(pattern = " .*", x = fdat, replace = "")
date <- gsub(pattern = "-", x = date, replace = "")
```

This step is particular useful, if we want only tweets in our dataset, that were posted during a specific time-period. Using the `which` command, we can figure out the positions of each tweet in our dataset, which has been posted prior to or after a certain date (or time, if you wish). The `date` and `time` variables we created before are numeric values describing the date/time of the tweet as `YYYYMMDD` and `HHMMSS` respectively. As soon as we found out which positions fir the criteria (before February 10th and after February 16th, for example), we can eliminate all of these tweets from our dataset:

```
twitter_exclude <- which(as.numeric(date) > 20180216 | as.numeric(date) < 20180210)
date <- date[-twitter_exclude]
..
user <- user[-twitter_exclude]
```

Creating the finished dataset

After we scraped all desired tweets and extracted the relevant information from it, it makes sense to combine the individual variables to a dataset, which can be easily handled, exported and reused. It also makes sense to have relatively short variable-names within such dataset.

When combining these variables into a `data.frame()`, we first need to create a matrix from them, by *binding* these variables as columns of said matrix with the `cbind()` command. The result can be used by the `data.frame()` function to great such item. We label this dataset `twitter`, making it clear, what source of data we are dealing with:

```
twitter <- data.frame(cbind(date, time, fdat, retw, favs, text,
                      lang, murl, link, clnt, user))
```

Often during that process, all variables within the `data.frame()` item are transformed into `factor()` variables, which is not what we want for most of these. Usually, when working with variables within a `data.frame()` you have to prefix the variable with the name of the `data.frame` and a dollar-sign, meaning that you want to access that variable **within** that `data.frame()`. This would make the process of changing the mode quite tedious for each variable:

```
twitter$text <- as.numeric(as.character(twitter$text))
```

Instead, we can use the `within()` function, using the `twitter` dataset as one argument and the expression of what we want to do *within* this dataset as another:

```
twitter <- within(data = twitter, expr = {
           date <- as.character(date);
           time <- as.character(time);
                   fdat <- as.character(fdat);
                   retw <- as.character(retw);
                   favs <- as.character(favs);
```

```
        text <- as.character(text);
        link <- as.character(link);
                murl <- as.character(murl);
                lang <- as.character(lang);
                clnt <- as.character(clnt);
                user <- as.character(user);
    })
```

The expression `as.numeric(as.character(...))` in some of these assignments are due to the issues, when transforming `factor()` variables to `numeric()` variables directly, as mentioned before. First transforming them into a `character()` (string), which then can be transformed into a `numeric()` value without risks, is a little *hack*.

The dataset is now finished and contains every aspect we want to analyze later on. You can skip down to the Exporting-Section to read about how to export the data, so it can be used outside your current R-Session.

Fediverse

The Mastodon-API doesn't require authentication for public timelines or (hash-) tags. Since this is exactly the data we want to aggregate, authentication is not needed here.

Scraping Toots and Postings

Contrary to Twitter, Mastodon does not allow to search for a string contained in posts, however we can search for hashtags through the tag-timeline in the API. For this we have to construct an URL for the API-call (keep an eye on changes to the API and adapt accordingly) in the following form:

```
https://DOMAIN.OF.INSTANCE/api/v1/timeline/tag/SEARCHTERM
```

Additionally, you can add `?limit=40` at the end of the URL to raise the results from 20 to 40 posts. For the search term it makes sense to use our official hashtag for the "I Love Free Software" Campaign: *ilovefs*.

In R, you can easily construct this with the `paste0()` function (the `paste()` function will introduce spaces between the arguments, which we obviously do not want):

```
mastodon_instance <- "https://mastodon.social"
mastodon_hashtag <- "ilovefs"
mastodon_url <- paste0(mastodon_instance,
          "/api/v1/timelines/tag/",
                     mastodon_hashtag,
                     "?limit=40")
```

Next, we use the `curl_fetch_memory()` function to fetch the data from our mastodon instance. The result of this is raw data, not readable by humans. In order to translate this into a readable format, we use `rawToChar()` form the R base package. This readable format is actually JSON, which can be easily transformed into a `list()` item with the `fromJSON()` function. All three functions put together, we have something like this:

```
mastodon_reqres <- curl_fetch_memory(mastodon_url)
mastodon_rawjson <- rawToChar(mastodon_reqres$content)
toots <- fromJSON(mastodon_rawjson)
```

`toots` is our resulting `list()` item.

Another issue is, that the Mastodon-API currently caps at 40 toots. However, we want much more than only the last 40, so we need to make several API-calls, specifying the *"range"*. This is set with the `max_id=`

parameter within the URL. The "ID" is the unique identifier of each status/post. You can have several parameters in the query string by dividing them by the `&` character, which will look similar to this:

```
https://DOMAIN.OF.INSTANCE/api/v1/timeline/tag/SEARCHTERM/?limit=40&max_id=IDNUMBER
```

Luckily, we do not have to find out the ID manually. The header `link` of the API response is saved into the `mastodon_lheader` variable, it lists the "*next page*" of results, so we can simply grab this with the `parse_headers()` function from the `curl` package and use some regex to strip it out:

```
mastodon_lheader <- parse_headers(mastodon_reqres$headers)[11]
mastodon_next <- sub(x = mastodon_lheader, pattern = ".*link:\ <", replace = "")
mastodon_url <- sub(x = mastodon_next, pattern = ">;\ rel=\"next\".*", replace = "")
```

If you are not familiar with regex, I highly recommend regexr.com to learn how to use it. It also contains a nifty cheat-sheet.

If this returns a valid result (if the `toot` variable is set), we forward it to the extraction function called `mastodon.fetchdata()`, which is defined earlier in the script. This returns a `data.frame()` item, containing all relevant variables **of the current "page"**. If we continously bind them together in a for-loop, we finally receive multiple vectors of all toots ever posted with the (hash-) tag *#ilovefs*:

```
if(length(toots) > 0){
  tmp_mastodon_df <- mastodon.fetchdata(data = toots)
  datetime <- c(datetime, as.character(tmp_mastodon_df$tmp_datetime))
  lang <- c(lang, as.character(tmp_mastodon_df$tmp_lang))
  inst <- c(inst, as.character(tmp_mastodon_df$tmp_inst))
  link <- c(link, as.character(tmp_mastodon_df$tmp_link))
  text <- c(text, as.character(tmp_mastodon_df$tmp_text))
  reto <- c(reto, as.character(tmp_mastodon_df$tmp_reto))
  favs <- c(favs, as.character(tmp_mastodon_df$tmp_favs))
  murl <- c(murl, as.character(tmp_mastodon_df$tmp_murl))
  acct <- c(acct, as.character(tmp_mastodon_df$tmp_acct))
} else {
  break
}
```

As of writing this documentation, the cap of the for-loop is set to 9999999999, which most likely will never be reached. However, the loop will always stop, as soon as the `toot` variable doesn't contain meaningful content anymore (see the *break* command in the code above).

When extracted, some of the date has to be reformed, reformatted or changed in some way. We use regex for this as well. For the sake of simplicity, the example below only shows the cleaning of the `text` variable. Other variables are treated in a similar fashion:

```
text <- gsub(pattern = "<.*?>", x = text, replacement = "")
text <- gsub(pattern = "  ", x = text, replacement = "")
```

Additionally, posts that are too old have to be removed (usually, setting the oldest date to January 01 of the current year works fine, February may be fine as well). The format of the date should be `YYYYMMDD` and a `numeric()` value:

```
mastodon_exclude <- which(date < 20180101)
date <- date[-mastodon_exclude]
time <- time[-mastodon_exclude]
lang <- lang[-mastodon_exclude]
inst <- inst[-mastodon_exclude]
text <- text[-mastodon_exclude]
link <- link[-mastodon_exclude]
reto <- reto[-mastodon_exclude]
```

```
favs <- favs[-mastodon_exclude]
murl <- murl[-mastodon_exclude]
acct <- acct[-mastodon_exclude]
```

Extraction Function

The extraction function `mastodon.fetchdata()` has to be defined prior to running it (obviously), hence it is the first chunk of code in the fediverse-section of the script. As argument, it only takes the extracted data in a `list()` format (which we saved in the variable `toot`). For each post/toot in the `list()` item, the function will extract:

- date & time of the post
- language of the post (currently only differenciates between english/japanese)
- the instance of the poster/tooter
- the URL of the post
- the actual content/text of the post
- the number of boots/shares/retweets
- the number of favorites
- the URL of the attached image (NA, if no image is attached)
- the account of the poster (instance & username)

```
mastodon.fetchdata <- function(data){

...

for(i in 1:length(data)){

  #### Time and Date of Toot
  if(length(data[[i]]$created_at) > 0){
    tmp_datetime[i] <- data[[i]]$created_at
  } else {
  # insert empty value, if it does not exist
    tmp_datetime[i] <- NA
  }

  ...

}

return(data.frame(cbind(tmp_datetime,
           tmp_lang,
           tmp_inst,
           tmp_text,
           tmp_link,
           tmp_reto,
           tmp_favs,
           tmp_murl,
           tmp_acct)))
}
```

Creating the finished dataset

As before with the Twitter-data, we combine these newly created variables into a `data.frame()` item by first turning it into a matrix by binding these vectors as columns with `cbind()` and turning it into the finished

dataset called `mastodon` with `data.frame()`:

```
mastodon <- data.frame(cbind(date, time, lang, inst, text, link, reto, favs, murl, acct))
```

As this usually re-defines the variables as `factor()`, we will use `within()` again, to give them the correct mode:

```
mastodon <- within(data = mastodon, expr = {
             date <- as.numeric(as.character(date));
             time <- as.numeric(as.character(time));
             text <- as.character(text);
             link <- as.character(link);
             murl <- as.character(murl);
          })
```

The dataset can now be exported. Skip down to the Exporting-Section to learn how.

Reddit

RedditExtractoR (or actually Reddit) doesn't currently require you to authenticate. So you can get right into scraping!

Scraping Posts

There are multiple ways of searching for a certain string. Optionally, you can determine which Subreddit you want to search in. In most cases, it makes sense to search in all of them. To search for a particular string, we use the `reddit_urls()` function. It takes one mandatory and five optional arguments:

- the string we want to search for (I am not certain, whether this includes the content / text of the actual posts or only titles). In our case, **ilovefs** should work just fine, as this is the name of the campaign and probably what people will use in their posts
- the subreddits we want to search in. There is no real reason to limit this in the case of the ILoveFS-Campaign, but it may make sense in other cases. If not needed, this argument can be commented out with a `#`
- the minimum number of comments a post should have in order to be included. As we want all posts regardless of their popularity, we should set this to `0`
- how many pages of posts the result should include. Here applies the same as before: we want all posts, so we set this to a very high number like `99999`
- the sort order of the results. This doesn't really matter, as we try to scrape all posts containing our search string. You can most likely leave it out or set it to `new`
- the wait time between API-requests. The minimum (API limit) is 2 seconds, but if you want to be sure set it to a slightly higher level

The resultis saved to the variable `reddit_post_dirty`, where as the *dirty* stands for the fact that we haven't yet sorted out older posts than from this year's event:

```
reddit_post_dirty <- reddit_urls(search_terms = "ilovefs",
                                 #subreddit = "freesoftware linux opensource",
                                 cn_threshold = 0,
                                 page_threshold = 99999,
                                 sort_by = "new",
                                 wait_time = 5)
```

Stripping out data

The data from the `RedditExtractoR` package comes in an easily usable `data.frame()` output. Its structure is illustrated below:

```
reddit_post
  |
  |- date = "14-02-17", "13-02-17", ...
  |- num_comments = "23", "15", ...
  |- title = "Why I love Opensource #ilovefs", "Please participate in ILoveFS", ...
  |- subreddit = "opensource", "linux", ...
  '- URL = "https://www.reddit.com/r/opensource/comments/dhfiu/", ...
```

Firstly, we should exclude all postings from years before. For this, we simply trim the `date` variable (Format is "DD-MM-YY") within the `data.frame()` to only display the year and use those posts from the current year. We save the result in the `reddit_post` variable:

```
reddit_searchinyear <- 18
reddit_post_year <- gsub(x = reddit_post_dirty$date,
             pattern = "\\d.-\\d.-",
             replace = "")
reddit_post <- reddit_post_dirty[which(reddit_post_year == reddit_searchfromyear),]
```

To ease the handling of this process, the year we want to search in is assigned to the variable `reddit_searchinyear` in a "YY" format first (here: "18" for "2018"). We use `gsub()` to trim the date to just display the year and use `which()` to determine which post's year is equal to `reddit_searchinyear`.

Afterwards, we can use a single for-loop to extract all relevant variables. We simply create an empty `vector()` for each variable:

```
comt <- c()
subr <- c()
ttle <- c()
date <- c()
rurl <- c()
```

And fill the appropriate position on the vector with the corresponding value. We do this for each scraped post:

```
for(i in c(1:length(reddit_post$URL))){
  comt[i] <- reddit_post$num_comments[i]
  ttle[i] <- reddit_post$title[i]
  rurl[i] <- reddit_post$URL[i]
  date[i] <- gsub(x = reddit_post$date[i], pattern = "-", replace = "")
  subr[i] <- reddit_post$subreddit[i]
  ...
}
```

However, not all of the relevant data is contained in the `reddit_post` dataset. We need another function from the `RedditExtractoR` package, called `reddit_content()` which is able to also give us the score, text and linked-to website of the post. As an argument, this function only needs the URL of a post, which is contained in our previously mentioned `data.frame()`:

```
reddit_content <- reddit_content(URL = reddit_post$URL[1])
```

The resulting variable `reddit_content` is another `data.frame()` with a similar structure as the previously used `reddit_post`:

```
reddit_content
  |
```

```
|- ...
|- num_comments = "20"
|- ...
|- post_score = "15"
|- ...
|- post_text = "I really do love this software because..."
|- link = "https://cran.r-project.org"
'- ...
```

Since we need to do this for every single post, we can include this into our for-loop. Because we call the function with only one post-URL at a time, we can set the wait time between request to zero. However the for-loop will call the function multiple times regardless, so we should make sure it is actually waiting before doing so or else we will hit the API-timeout. We can do this with the `Sys.sleep()` function. Everything put together:

```
comt <- c()
subr <- c()
ptns <- c()
ttle <- c()
text <- c()
link <- c()
date <- c()
rurl <- c()
for(i in c(1:length(reddit_post$URL))){
  comt[i] <- reddit_post$num_comments[i]
  ttle[i] <- reddit_post$title[i]
  rurl[i] <- reddit_post$URL[i]
  date[i] <- gsub(x = reddit_post$date[i], pattern = "-", replace = "")
  subr[i] <- reddit_post$subreddit[i]
  Sys.sleep(2)
  reddit_content <- reddit_content(URL = reddit_post$URL[i], wait_time = 0)
  ptns[i] <- reddit_content$post_score
  text[i] <- reddit_content$post_text
  link[i] <- reddit_content$link
}
```

Creating the finished dataset

As we do not really need to *filter* something out (we have already done so with the dates before), we can directly bind our variables to a `data.frame()`. As with the other datasources (eg.: Twitter) we create a matrix with the `cbind()` fucntion, which can be tunred into the finished dataset with `data.frame()`, assigning it to the variable `reddit`:

```
reddit <- data.frame(cbind(date, rurl, link, text, ttle, ptns, subr, comt))
```

This usually re-defines every single variable within the dataset as `factor()`, so we use the `within()` function to change their mode:

```
reddit <- within(data = reddit, expr = {
          date <- as.numeric(as.character(date));
          rurl <- as.character(rurl);
          link <- as.character(link);
          text <- as.character(text);
          ttle <- as.character(ttle);
          ptns <- as.numeric(as.character(ptns));
          subr <- as.character(subr);
```

```
 comt <- as.numeric(as.character(comt));
})
```

The dataset can now be exported. Skip down to the Exporting-Section to learn how.

Exporting Datasets

There are several reasons to why we want to export our data:

1. to keep a backup / an archive. As we have seen in the Twitter-Section, the social media sites do not always enable us to collect a full back-log of what has been posted in the past. If we want to analyze our data at a later point of time or if we want to compare several point of times to another, it makes sense to have an archive and preferably a backup to prevent data loss
2. to use the data outside your current R-session. The variables only live for as long as your R-session is running. As soon as you close it, all is gone (except if you agree to save to an image, which actually does the very same, we are doing here). So it makes sense to export the data, which then can be imported and worked with later again.
3. to enable other use to analyze and work with the data. Obviously, this is an important one for us. We **do** want to share our results and the data we used for this so other people can learn and to make our anylsis transparent.

In order to fully enable anyone to use the data, whatever software he or she is using, we export in three common and easily readable formats: `.RData .csv .txt`. The later one is the simplest one and can be read by literally **any** text-editor. Each string in there is enclosed by quotes `"` and seperated with a single space in a table-layout. The `.csv` format is very similar, though the seperation is done with a symbol - in this case a colon `,`. This format is not only readable by all text-editors (because it is pure text), it can also be read by spreadsheet applications like libreoffice-calc. The disadvantage of both formats is, that they can only hold items with the same "labels", so we need to create multiple export-files for each data source. Also, when importing, you often have to redefine each vaiable's mode again.

Lastly, we also export as `.RData`, R's very own format. Since R is free software, I would suspect, that most statistics-software can read this format, but I do not actually know for a fact. However, it certainly is the easiest to work with in R, as you can include as many variables and datasets as you want and the modes of each variable stay in tact. `.RData` is a binary format and can not be read by text-editors or non-specialized software.

In order to have an easily navigatable archive, we should not only label the output-files with the source of the data, but also with the date when they were collected. For this, we first need the current time/date, which R provides with the `Sys.time()` function. We want to bring it in a format suitable for file names like "YYYY-MM-DD_HH-MM-SS", which we can do with `sub()` and `gsub()` respectively:

```
time_of_saving <- sub(x = Sys.time(), pattern = " CET", replace = "")
time_of_saving <- sub(x = time_of_saving, pattern = " ", replace = "_")
time_of_saving <- gsub(x = time_of_saving, pattern = ":", replace = "-")
```

Next, we model the save-path we want the data to be exported to, for which we can use `paste0()`. For example to save the `.RData` file, we want to export to the `data/` folder:

```
save_path <- paste0("./data/ilovefs-all_", time_of_saving, ".RData")
```

Note: using `paste()` instead of `paste0()` will create a space between each strings, which we do not want here.

We follow a similar approach for the individual `.txt` files, also adding the name of the source into the filename (as they will only hold one data source each). For example:

```
save_path_twitter_t <- paste0("./data/ilovefs-twitter_", time_of_saving, ".txt")
```

Lastly, we need to actually export the data, which we can do with:

```
save()        # for .RData
write.table() # for .txt
write.csv()   # for .csv
```

All three functions take the data as argument, as well as the previously defined file path. In the case of `save()` where we export multiple datasets, their names need to be collected in a `vector()` item with the `c()` function first:

```
save(list = c("twitter", "mastodon"), file = save_path)

write.table(mastodon, file = save_path_fed_t)

write.csv(twitter, file = save_path_twitter_c)
```

If this is done, we can safely close our R-Session, as we just archived all data for later use or for other people to join in!