Like many other fans of the show, I had great expectations for the eighth and last season of Game of Thrones (GoT) that premiered on 14 April 2019. Nonetheless, it provided a good testing ground for quantitative text analysis – by scouring and analysing tweets from the US in the course of the eighth and final GoT season, spanning the period between 7 April and 28 May 2019.

**Introduction**

In many regards, this post will be very different from previous entries. While the focus is the usual R-based statistical analysis, data collection is also discussed in depth and this in turn begs for basic Unix / macOS terminal commands. MS Windows users can refer to VirtualBox or Ubuntu installations. In summary, this post will demonstrate how to

* Schedule a Twitter API program to harvest up to 100,000 tweets on a daily basis;
* Process and aggregate harvested tweets;
* Explore the resulting dataset using geocoding, document-feature and feature co-occurrence matrices, wordclouds and time-resolved sentiment analysis.

**Kaggle**

The large size of the resulting Twitter dataset (714.5 MB), also unusual in this blog series and prohibitive for GitHub standards, had me resorting to Kaggle Datasets for hosting it. Kaggle not only literally offers a considerable storage capacity for both private and public datasets, but also both writing and reproducible execution of R and Python scripts, also called kernels or notebooks. Notably, execution is served with up to 16 GB RAM and 5 GB storage ATTOW.

[**You can find the dataset page here**](https://www.kaggle.com/monogenea/game-of-thrones-twitter) and download the files either manually or using the Kaggle API with the following command,

kaggle datasets download monogenea/game-of-thrones-twitter -p INSERT\_PATH

The Twitter dataset gotTwitter.csv shows up under *Data Sources* along with the code used for data collection. The code was split between the complementary scripts harvest.R and process.R that deal with tweet harvest and processing, respectively

**The cron scheduler**

Before turning to the R analysis I would like to introduce the cron scheduler. The cron is a convenient Unix scheduler that can assist with repeating tasks over regular time intervals, and a powerful tool for searching tweets using the Twitter API. To initialise a cron job you need to launch a text editor from the terminal by running crontab -e and add a line as shown below, before saving.

\* \* \* \* \* INSERT\_PATH\_INTERPRETER INSERT\_PATH\_SCRIPT

The first five arguments separated by whitespace specify minute, hour, day of month, month and day of the week, respectively. An asterisk in place of any of the five instructs the job to be executed every such instance. Say, if you want a job to be run every Monday at 17:30 you could use 30 17 \* \* 1. Taking another example, if every tenth day of the month and every minute between 6:00 and 7:00, \* 06 10 \* \*. One can additionally specify ranges (e.g. 00-14 in the first position for the first 15 minutes) or arrays (e.g. 01,05 in the second position for 1:00 and 5:00). This is testament to the versatility of the cron scheduler. Once the job is finished you can simply kill the scheduler by running crontab -r. If, however, you have additional unfinished jobs, enter the editor with crontab -e instead, delete the line above and save.

**Sharing is caring**

Much of this post will review, in finer detail, both harvest and processing steps as well as the analysis from the accompanying kernel. Due to the large volume of all harvested files combined and the restricted access to tweets from that period, data collection and analysis are decoupled. I will first describe the data collection so that you can familiarise yourself with the process and reutilise the two scripts. The R analysis, on the other hand, is based on the provided dataset and should be fully reproducible both locally and on Kaggle.

Take the utmost responsibility when handling demographic information. The present records captured from the Twitter API are in the [public domain](https://creativecommons.org/publicdomain/zero/1.0/) and licensed as such, sensitive to the extent they associate with usernames and geographical coordinates. Use these tools for the common good, and always aim at making your data and work visible and accessible.

**Let’s get started with R**

**Harvest**

The quality of data collection and downstream analyses is dictated by the scheduled daily tweet search, or as I call it, the harvest. The harvest was executed using the script harvest.R, which will next be broken down into three separate parts – the Twitter API, the optional Google Maps API and the actual tweet search. We will then define the cron job setup that brings all three together.

**Twitter API**

The standard Twitter API, which is free of charge, offers a seven-day endpoint to your tweet searches. This means you can only retrieve tweets that are at most seven-day old. If you are planning a search to trace back longer than that, the obvious alternative to [paid subscriptions](https://developer.twitter.com/en/docs/tweets/search/api-reference) is to repeatedly search tweets weekly or daily, using the standard option. Here is how.

To get started you first need a set of credentials from a [Twitter API account](https://developer.twitter.com/). You can use your regular Twitter account, if you have one. All it takes is creating an app, then extracting consumer key, consumer secret, access token and access secret. Handle these carefully and do not share them. You can then create a token with create\_token from rtweet by passing the app name and the four keys as shown below.

Only much later did I notice that a single authentication creates the hidden file .rtweet\_token.rds to be used in future executions via .Renviron. Eventually, repeated searches will write the same file with different number endings, creating unnecessary clutter. Because it makes no different in this process, you can prevent this from happening by switching off the set\_renv option from create\_token.

**Google Maps API (optional)**

The existence of geographical coordinates from users or devices can substantially empower studies based on social media content. If you want to geocode tweets you also need a [Maps JavaScript API key](https://developers.google.com/maps/documentation/javascript/tutorial) from the Google Maps Platform. This can be done for free after setting up an account, creating a project, setting up a billing account with your credit card details and generating a set of credentials you can then pass to rtweet. You should get about US$200 monthly free credit, which is plenty for searching tweets.

After setting up your Google Maps token, you can pass the key to lookup\_cords as apikey, immediately after the region, country or city you want to study. In the present case we are interested in the whole of the US, so we have lookup\_coords("usa", apikey = apiKey) as will be seen later.

**Tweet search**

The key player in the harvest process is the actual tweet search, which is singlehandedly managed by the function search\_tweets. Inside this function you can define keywords q, whether or not to use retryonratelimit, language lang, geocoding geocode, whether or not to include retweets include\_rts and search size n among other options. I used the simple query game of thrones with retry-on-rate-limit, English language, US geocoding, discarding retweets and limiting the search to 100,000 tweets. The resulting object newTweets is then to be written into a CSV file in a directory called tweets. I found it convenient calling Sys.date() to set the name of the individual CSV files to the corresponding day of harvesting, i.e. YYYY-MM-DD.csv.

**Cron job setup**

At this stage the tweet search harvest.R is fully set up and ready to be scheduled. The interpreter for executing R scripts is Rscript and the script we want to execute on a regular basis is harvest.R. As put forth in the Get-Started kernel, to reproduce my cron job you need to launch a text editor from the terminal by running crontab -e and add the following line before saving,

0 04 \* \* \* /Library/Frameworks/R.framework/Resources/Rscript ~/Documents/INSERT\_PATH/harvest.R

My cron job was instructed to execute the script harvest.R everyday at 4:00. You might also have noticed the very first line in harvest.R, carrying a #! prefix. This is a [shebang](https://en.wikipedia.org/wiki/Shebang_(Unix)), a special header that invokes an interpreter for executing the script. By using it we no longer need to specify the interpreter in the cron instance above. However, execution permissions must be first granted from the terminal, with e.g. the command chmod u+x harvest.R. Only then the following alternative to the above crontab will work,

0 04 \* \* \* ~/Documents/INSERT\_PATH/harvest.R

**Processing**

I ran through some issues at the start of the harvest, so the first effective batch from 17 April 2019 had to stretch back well over three days, when the first episode aired. To address the problem, I exceptionally set the search size to 300,000 tweets in the first day and to 100,000 in all successive days until the end of the harvest. Taking 100,000 per day was clearly more than needed, as reflected by large overlaps in successive files (data not shown). Despite the redundancy, it nevertheless assured capturing most activity including peaks coinciding with the air dates of all six episodes. By the end of the harvest all batches but the first weighted up to ~100 MB.



The process generated a large volume of data (4.02 GB), so it made little sense to keep the individual files. Therefore, I wrote the script processing.R that

***i)*** Lists all CSV files in the directory tweets;

***ii)***Defines a function mergeTweets to extract records from a donor table to a reference recipient table, using the column status\_id to identify unique tweets;

***iii)*** Iterates the function over all files listed in *i)* to populate a recipient table called allTweets;

***iv)*** Writes the resulting allTweets table as gotTwitter.csv with UTF-8 encoding.

Note that I make use of rtweet functions throughout, namely read\_twitter\_csv, do\_call\_rbind and write\_as\_csv. These functions are optimised for handling rtweet objects, they run considerably faster and are cross-compatible, compared to alternative methods. Also note the unflatten = T option in read\_twitter\_csv that prevents inadvertently writing the flat long-lat coordinates back to CSV. This was one of the issues I ran into, as the default mode will return a table whose coordinates can no longer be used with the maps package.

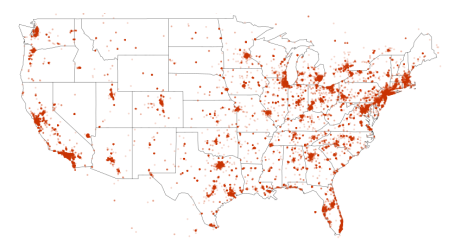
To my relief, the resulting gotTwitter.csv file carried tweets dated to between 7 April and 28 May 2019, therefore covering all six episode air dates and some more. It amassed an impressive total of 760,660 unique tweets.

**Get-Started analysis**

This analysis is largely based on the quanteda and maps packages and fully described in the [Get-Started kernel](https://www.kaggle.com/monogenea/quick-guide-to-game-of-thrones-s8-twitter). We will first load all required packages and read the CSV Twitter file. The tidyverse package and downstream dependencies work seamlessly with rtweet, maps and lubridate. The package reshape2 will be used later to convert timestamps from wide to long format. The Twitter dataset will then be read and unflattened using read\_twitter\_csv.

Before proceeding, you might want to convert the UTC timestamps under created\_at to an appropriate US timezone. In the kernel, created\_at is overwritten with the corresponding lubridate encoding via as\_datetime, and then converted to EDT (NY time) using with\_tz.

Next, we can look into the US-wide geographical distribution of the harvested tweets. It will first take overwriting of allTweets using lat\_long, which will simply add two new columns lat and lng carrying valid long-lat coordinates. Then we can create an instance of maps, which will require very large par margins in Kaggle kernels. To draw the US map with state borders we use map("state") with an appropriate lwd option to set line width. Now we can add the data points to the canvas by passing the long-lat coordinates.



Aside from having more GoT reactions on Twitter in more densely populated areas, the above figure clearly hints on the representativity of all 48 contiguous US states. My apologies to all Alaskans and Hawaiians, please modify the code above to visualise either state or the whole of the 50-state US. Interestingly, the dataset covers some activity outside of the US.

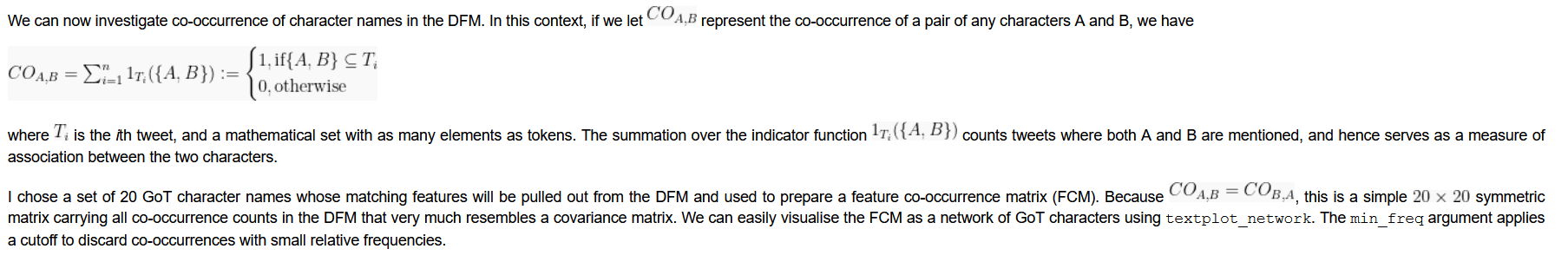
Let us move on to textual analysis and the underlying mathematical representation. We will clean up the tweet text by removing irrelevant elements, generate tokens from the resulting content and build a document-feature matrix (DFM). In rigour, tokens are vocables that comprise single or multiple words (i.e. *n*-grams) delineated by whitespace, and the number of occurrences per tweet shows up in the corresponding DFM column. Because tokenisation is exhaustive, DFMs tend to be extremely sparse.

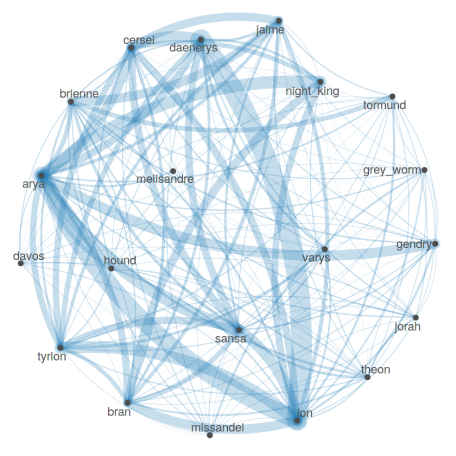
The proposed tokenisation process strips off Twitter-specific characters, separators, symbols, punctuation, URLs, hyphens and numbers. Then, it identifies all possible uni- and bigrams. The inclusion of bigrams is important to capture references to characters like Night King and Grey Worm. Then, after setting all alphabetical characters to lower-case and removing English stop-words (e.g. *the*, *and*, *or*, *by*) we create the DFM gotDfm, with a total of 2,025,121 features.

We can now investigate co-occurrence of character names in the DFM. In this context, if we let  represent the co-occurrence of a pair of any characters A and B, we have

where is the *i*th tweet, and a mathematical set with as many elements as tokens. The summation over the indicator function counts tweets where both A and B are mentioned, and hence serves as a measure of association between the two characters.

I chose a set of 20 GoT character names whose matching features will be pulled out from the DFM and used to prepare a feature co-occurrence matrix (FCM). Because , this is a simple symmetric matrix carrying all co-occurrence counts in the DFM that very much resembles a covariance matrix. We can easily visualise the FCM as a network of GoT characters using textplot\_network. The min\_freq argument applies a cutoff to discard co-occurrences with small relative frequencies.

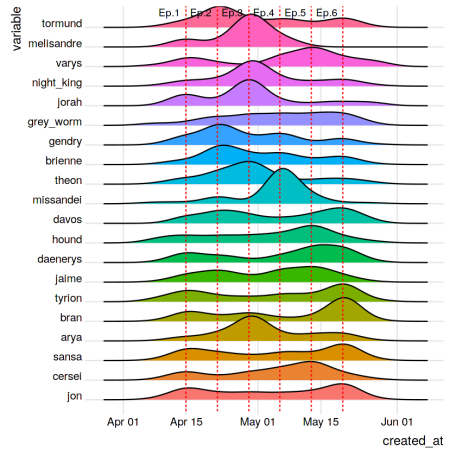




The undirected graph above represents my choice of 20 GoT characters using nodes, and the underlying co-occurrences by means of connecting edges, which are the larger the greater their relative frequency. Do these results make sense? For one, the strongest associations occur between lovers or enemies. In the first case we have couples like Daenerys and Jon, Jaime and Cersei, Arya and Gendry, Brienne and Jaime with a dash of Tormund. Although not popular overall, Grey Worm associated more closely to his beloved Missandei too. In the second case, we have Arya and the Night King, Cersei and Daenerys and Arya and Cersei. This is clearly a very crude way to characterise their relationships and one could argue this should be done on separate time intervals, as these same relationships can be assumed to evolve the show throughout.

Next we will consider the popularity dynamics of all 20 GoT characters to flesh out patterns based on their interventions in all six episodes. We will create the object popularity to carry binary values that ascertain whether individual tweets mention any of the characters, from the tkn list object. Then, we append the created\_at column from allTweets and use the melt function to expand created\_at over all 20 columns. Finally, entries where any of the 20 GoT characters is mentioned are selected and the results can be visualised.

Since character popularity can be expected to fluctuate over time, pinpointing the exact air dates of all six episodes can greatly improve our analysis. They all occurred in regular intervals of one week, at 21:00 EST starting 14 May, 2019. Again with the help of lubridate, we can encode this using ymd\_hms("2019-04-14 21:00:00", tz="EST") + dweeks(0:5). We can now use epAirTime to highlight the exact time of all six air dates. Finally, a bit of ggplot and ggridges magic will help plotting the distribution of character references over time.



Here too we can relate the results to key events in the show, specifically

* Brienne knighted by Jaime in Ep.2;
* The affair between Arya and Gendry in Ep.2;
* The demise of Melisandre, Jorah, Theon and Night King in Ep. 3;
* The beheading and execution of Missandei and Varys, respectively, in Ep.4;
* The demise of both the Hound and Cersei in Ep.5;
* The crowning of Bran in Ep.6.

Up to this point all materials and analysis we discussed are described and available from the [Kaggle dataset page](https://www.kaggle.com/monogenea/game-of-thrones-twitter). Time for a break?

**Since you are here**

To avoid making this post a mere recycling of what I uploaded to Kaggle a few months ago, I propose moving on to discuss Twitter bots, building wordclouds and conducting sentiment analysis.

**Dealing with Twitter bots**

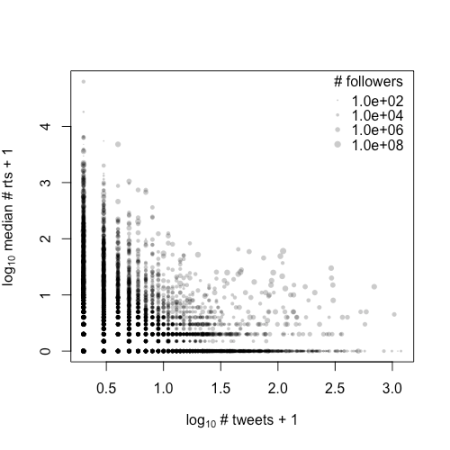
In working with Twitter data, one can argue that the inexpressive and pervasive nature of ads and news put out by bot accounts can severely bias analyses aimed at user sentiment, which we will use shortly. One strategy to identify and rule out bots is to simply summarise the number of tweets, as there should be a human limit to how many you can write in the period between 7 April and 28 May 2019. An appropriate tweet count limit can then be used, beyond which users are considered bots.

But before kicking off, I would like to bring out some intuition about tweeting behaviour. I would expect that users are generally less reactive to corporate than personal tweets (your thoughts?). How reactive users are about a certain tweet, for example, can be manifest in retweeting. Moreover, considering the point discussed above,  I would therefore expect users that tweet less to gather more retweets in average. The following piece will test this hypothesis in the present dataset, and investigate that association in relation to the number of followers.

The dataset can be partitioned based on the usernames available under allTweets$screen\_name, from which we can then derive

* num\_tweets, how often each user tweeted during this period;
* median\_rt, the median number or retweets from each user (median over mean, to avoid large fluctuations due to extreme cases of null or exaggerated retweeting);
* mean\_followers, the average number of followers in that period (average since of course it is dynamic).

We should also visualise the data in log-scale, as they are expectedly highly skewed. The code below uses a transformation with a unit offset.



It seems that indeed there is a negative relationship between the frequency of tweeting and the average number of retweets. Here, the plotting character size is proportional to the of the number of followers. In the original scale, we are talking about users with a number of followers ranging from zero to 77,581,90. This impressive number in particular comes from @TheEllenShow. Nonetheless, there is no clear association with neither median retweet or tweet counts, which suggests that both high- and low-profile users can be equally prolific and equally contribute to viral retweeting.

Before turning our attention to wordclouds, I propose we first remove potential bots. How likely is it for a human user to publish tweets about GoT in seven weeks? My approach removes all users that tweeted as many as  times during this period. You may find it to be too strict, so feel free to update the cutoff.

**Wordclouds**

Wordclouds are an effective way of summarising textual data by displaying the top frequent terms in a DFM, where word size is proportional to its relative frequency. To better understand the audience, the following wordcloud construction procedure will be focussed on tweets published right after each episode.

This will require cleaning the textual data more thoroughly than before. I mentioned UTF-8 previously, which is the encoding used on the text from our dataset. UTF-8 helps encoding special characters and symbols, such as those from non-latin alphabets, and can easily be decoded back. Since special characters and symbols can tell us little about opinion, we can easily remove them using a trick proposed. Then we also have Unicode for emojis, equally irrelevant. Under allTweets$text, emojis share a common encoding structure between arrows, e.g.  so we might be better off using a regex (i.e. regular expression) to replace all emoji encodings with single whitespaces while avoiding off-targets. In the code below, I propose using the regex <[A-Z+0-9]+>. You can read this pattern as something like *identify all occurrences of < followed by one or more capital letters, digits and +, followed by >*. Regex patterns are really useful and I might cover them in greater depth in a future post.

The tokenisation recipe above to remove symbols, punctuation and more can be re-used here. In preparing the subsequent DFM we will convert all letters to lower case and remove stop-words, as before, but also stem words. Stemming words, as the name suggests, clips the few last characters of a word and is effective in resolving the differences among singular, plural, and verbal forms of semantically related terms (e.g. *imagined*, *imagination* and *imagining* can be stemmed to *imagin*). Further down the line, irrelevant terms such as the show title and single-letter features can also be discarded from the DFM. Finally, we subset the resulting DFM for every two hours and four days after the airing of each of the six episodes. The six subsets will be use to construct six separate wordclouds with a word limit of 100. You might be warned about words not fitting the plot, worry not!

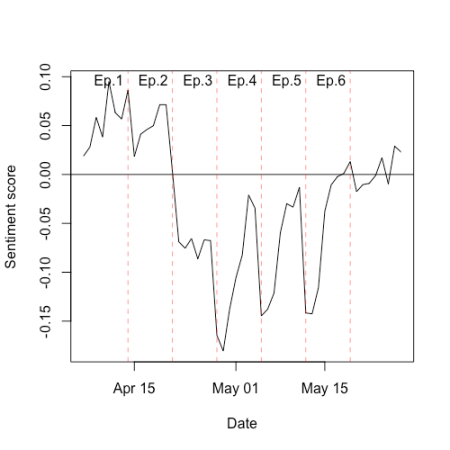
Move your mouse over the different panels to reveal the episode number and title. We can now inquire about why some of these frequent terms emerge:

* Ep.1 – *premier*, *first* and *one* certainly allude to this being the season premiere. Another interesting emerging term is *recap*, hinting on fans going back to previous seasons;
* Ep.2 – *night*, *week* and *battl* are clear references to the much anticipated battle of Winterfell that aired the week after (Ep.3, The Long Night); *arya* and *sex* likely refer to the sex scene with Gendry;
* Ep.3 – *night*, *battl* and *winterfel* as explained above. There are also some curious references to Avengers: Endgame, the film that premiered on 24 April 2019;
* Ep.4 – *end* and *final*, evidently; *bran* and *king* also emerge, as Bran is crowned the king of the six kingdoms. Some disappointment, as noticed before.

**Sentiment analysis**

I would like to conclude the post with sentiment analysis, i.e. determining the balance between positive and negative emotions over time, from matching tokens to a sentiment dictionary from quanteda. By framing the analysis against the six air dates we can make statements about the public opinion on the last GoT season.

Conducting sentiment analysis is deceptively simple. The tokens from the wordcloud exercise are initially matched against the dictionary data\_dictionary\_LSD2915 and processed just as before, to build a DFM. In contrast to previous DFMs, this instance carries counts of words associated with either positive or negative emotions. Alternative methods including the package SentimentAnalysis also list emotionally neutral words, thereby relying on a tertiary response. Finally, we pool counts from the same day accordingly and can now set to investigate the evolution of sentiment regarding the eighth season of GoT.



It is interesting to observe the dominant positive sentiment around the Ep.1 and Ep.2  that gives way to an uninterrupted period of negative sentiment between Ep.3 and Ep.5, which in turn evolves to a more neutral sentiment by Ep.6 and the end of the show.

While no statistical tests were used, I think it is fair to make some assumptions. There was a lot of excitement with the approaching premiere, as fans waited over one year for the final season. In that light, it would make sense to observe overall positive emotions leading to Ep.1. Then, the negative emotions that dominates from Ep.2 to Ep.5 are in fine agreement with the disappointment picked up from our wordclouds and indeed echoed in various media. Finally, with the approaching end of the season and the show by Ep.6, users presumably reviewed it as a whole, explaining perhaps the appeased negative reactions. Had we used a permutation test, e.g. shuffling the date order of the tweets and re-analysed the data, we could have drawn a confidence interval and determine the significance of the sentiment changes. Up for the challenge?

**Wrap-up**

This was a long journey, six months in the making, a very productive one . Various different aspects of Twitter data analysis were considered, including

* How to collect Twitter data, first by harvesting tweets using a cron job with daily access to the Twitter API, then by processing the harvested tweets;
* How to judiciously clean textual data, remove stop-words, stem words and manipulate date-time variables;
* How to build DFMs, FCMs and co-occurrence networks;
* How to analyse the popularity dynamics of individual GoT characters;
* How to distinguish human users and bots;
* How to build basic wordclouds using well defined time intervals;
* How to conduct a basic sentiment analysis.