Analyzing Trump Tweets with LSH

Thomas Asikis

10/19/2019

Introduction

So in this example we are going to use Local Sensitive Hashing to analyze similarity of Trump's tweets over the years. Our goal is to determine whether his tweets showcase similar word patterns over the years. To reach this goal we are going to rely on document similarity and associate "tweets" with high similarity over the years. In NLP terms, we are going to perform a document similarity task. After doing so, we are going to use a visuallization to evaluate our results. This will be an exploratory analysis, so we will not setup any hypothesis or statistical tests beforehand. As Trump's twitter account has been very active in the last years, it would be very difficult and require a lot of resources to do this analysis efficiently with other techniques, such as KNN over some "tweet vectors". Furthermore the construction of such vectors relying on learning techniques would also introduce additional training times on top of the nearest neighborh search task. ##

Data Preparation

In this section we will check how to load the data from various sources and also pre-process it for our NLP tasks.

Data loading

Since this will be our first technical example, we are going to check 2 possible ways of loading data: (i) remote dataset loading from dataverse and (ii) local dataset loading from a file in the same folder as the current script.

Dataverse

Dataverse contains a huge collection of research datasets that can be downloaded and stored locally. Each corpus dataset contains text data and a lot of extra metadata. The corpora files can also be downloaded locally via the browser. This corpus is titled "Twitter Tweets for Donald J. Trump (@realdonaldtrump) Version 1.2" and is found in the url:

https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi%3A10.7910%2FDVN%2FKJEBIL

To load the corpus from code, we need to know its doi (digital object identifier), which is a permanent unique id for the dataset. The doi can be found on the above website under the "Metadata" tab. The code for loading looks like:

```
library(dataverse)
trump_doi <- 'doi:10.7910/DVN/KJEBIL'
trump_tweets_ds <- dataverse::get_dataset(trump_doi)
```

In general there might be many files per dataset, so please check the dataset object and the website to get the one you are interested into. In the current example, there is only one file that we are interested into:

```
filename<-trump_tweets_ds$files[1]$label
f <- dataverse::get_file(filename, trump_doi)
```

Once the file is loaded in the dataverse code we need to store it locally. Therefore, we create a temporal file and we write the results in there. Be carefult to use the right extension, so that it is clear in your code what file parser to use!! The downloaded file is written in your disk, as it may too big to fit in your computer memory. If you want, you can also write the file into a known location to load it at a later stage and avoid re-downloading.

```
tmp <- tempfile(fileext = ".ndjson")
writeBin(as.vector(f) , tmp )
# load data in memory
all_data<-  ndjson::stream_in(tmp)
```

For now we load all the dataset in memory. If this is not possible, please look for information around "streaming large files in R" or "lazy loading big files in R".

Loading local files

Now we are going to load the data from a local file. After visiting the dataverse website, we can download the file locally and store it under the folder `downloads/`, which is in the same folder as the current script. To load the file, we need to tell RStudio or our script where to look for this file. For that reason we set the working directory as the folder that contains our script. The we load the data:

```
#rstudio
#source.folder <- setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
# for R scripts
#setwd(getSrcDirectory()[1])

all_data <- ndjson::stream_in('datasets/realdonaldtrump.ndjson')
```

Preprocessing

Now that all of our data is loaded we will take some preprocessing texts.

Data selection

In the current analysis we are interested only on text of the tweets and the date of their creation. As NLP tasks tend to be memory hungry, we remove any anused variables after loading the data we are interested in. TO reduce compilation times and also showcase this example live, we randomly sample 5000 tweets.

```
working_data <- all_data[sample(1:nrow(all_data),5000), c('text', 'created_at')]
rm(all_data)
rm(f)
```

```
## Warning in rm(f): object 'f' not found
```

```
rm(tmp)
```

```
## Warning in rm(tmp): object 'tmp' not found
```

```
rm(trump_tweets_ds)
```

URL removal and regex

Now that the data are chosen, we need to do some preprocessing on text. To determine preprocessing steps let's have a look at some tweets:

```
library(magrittr)
working_data %>% head(4)
```

```
##
## 8057                                                                                                .@JerryLawle
## 27574   I don't think Ted Cruz can even run for President until he can assure Republican voters that
## 36895     .@MikeDeWine will be a great Governor for the People of Ohio. He is an outstanding man who
## 11444 "@abimlebt: Eric Schneiderman trying to get himself an image as a strong man is suing @realDona
##                            created_at
## 8057  Wed Apr 10 02:09:39 +0000 2013
## 27574 Tue Jan 19 01:57:26 +0000 2016
## 36895 Tue Oct 30 17:37:49 +0000 2018
## 11444 Sun Aug 25 05:56:24 +0000 2013
```

After many checks we see that the tweets include urls and punctuation. These elements are not of our main interest and we would like to remove them to reduce noise during our NLP task processing. Usualy this can be done via existing libraries, but let's take the chance to introduce regular expressions. Very often we need to search for words or phrases in documents. This is reffered to as "searching for literals". Now we could also be very interested to search for patterns. For example, when searching for a url we may know that it most likely starts with `https://` or `http://` and then any combination of characters may follow except for a space. So it would be convenient to be able to say to R, that if you find anything that fits the above description, please remove it from the scripts. And with regular expressions this can be done by running the code below:

```
working_data$text <- gsub("http.*","",  working_data$text)
working_data$text <- gsub("https.*","", working_data$text)
working_data$text <- gsub('[[:punct:] ]+',' ',working_data$text)
```

So what we did in the first line is to say, find all strings between spaces that start with `http` and then are followed by any non-space symbol/character (in regex lingo this is `.`) repeated any number of times `*`. This simple yet powerfule representation can make searching patterns much more efficient. There are many special symbols and combinations of symbols when doing regular expressions. A nice explanation on the meaning of special regex symbols can be found at:

http://www.endmemo.com/program/R/gsub.php

A last remark on regular expressions, they might be very versalite tools but they are difficult to master and they require a lot of experience. Use them with care, but always check their outcome extensively, as they might not work as expected in all situations. Implementing a `for` loop and some `if` checks over words or characters might have much better results. Using R packages is still the best option if available.

Playing with dates

Now if we look at the date field of our data, we can notice that the date format might be difficult to parse.

```
working_data$created_at[1]
```

```
## [1] "Wed Apr 10 02:09:39 +0000 2013"
```

For that reason we use on more regular expression to replace the strange pattern `+0000` with a regex that is translated as: use the plus sign as literal and not a pattern detection character and then check if it is followed by at least one digit

`d+`. If this happens, replace it with an empty string.

```
working_data$created_at = sub('\\+\\d+','',working_data$created_at)
```

Now that our dates have a more recognizable format, we will try to parse them using a formatter. Date formats may differ and it is good to know how to describe their patterns when parsing them. This can be done by placing date pattern symbols in a string and providing that string to the date parser as follows:

```
format = "%a %b %d %H:%M:%S  %Y"
as.Date(working_data$created_at[1], format=format)
```

```
## [1] "2013-04-10"
```

Helpful information on date formats and how to describe their patterns is found in:

https://www.r-bloggers.com/date-formats-in-r/

Since checking the similarity between Trump's tweets individually may be cumbersome (in worst case we would have to go over 25 million comparisons), we would like to check this similarity per year. Now let's parse the dates of our dataset and extract the year of each tweet:

```
working_data$date = as.Date(working_data$created_at, format = format)
working_data$group_year = format(as.Date(working_data$date), "%Y")
```

Text normalization

As tweets are small texts and start with a capital usually, converting everything to lowercase moght help us with treating same words as different, due to different case. Still, we expect that the president of the United States to be using a lot of acronyms and especially in the limited character setting of twitter. To overcome this, we use tokenization that does lowercase transformation to most words except from some predicted acronyms.

```
working_data %<>%
  dplyr::mutate(text= quanteda::char_tolower(text, keep_acronyms =TRUE))
```

The cleaning processes above may have reduced some tweets to empty strings. For example this could happen when a tweet only contains a url. To avoid any errors with processing empty tweets we remove them.

```
working_data <- working_data[!(is.na(working_data$text) | working_data$text=="" ), ]
```

Nearest Neighbors

Now that our data is prepared, we can dive into find similar tweets. To do so we create a list of texts that is compatible with the library `textreuse` for locality sensitive hashing. We use the integer index of a tweet in the matrix as the corresponding document id:

```
text_list <- as.character(working_data$text)
names(text_list) <- rownames(working_data)
working_data$doc_id = rownames(working_data)
```

Locality Sensitive Hashing

We can provide the above list of texts to the `textreuse` function to create a corpus. Each text is being hashed using a minfunction that returns 200 minhashes.

```
lsh_corpus = textreuse::TextReuseCorpus(text = text_list,
                                        minhash_func =  textreuse::minhash_generator(n = 200))
```

Tweets that had a too small number of characters and/or words are not minhassed and skipped. Now that our minhashes are calculated, we can proceed and create our buckets, that will approximately capture similarity:

```
lsh_buckets <- textreuse::lsh(lsh_corpus, bands=100)
```

The bucket creation process may take some time. For the whole dataset it takes several minutes. Still, the most expensive process is to find the canditate nearest neighbors for each tweet. To do so we create a function that queries our buckets with a tweed id and then returns a list of all canditates. To use this list for further processing we melt it as \texttt{(search_id, canditate_id), (search_id1, canditate_id2) ...}. If no canditates are found, we return nothing.

```
library(pbapply)
get_edges <- function(doc_id){
  canditates <- textreuse::lsh_query(lsh_buckets, doc_id)$b
  edges <- lapply(canditates, function(cand_id){
    return(c(doc_id, cand_id))
  })
  if(length(edges) > 0){
    return(edges)
  }
}
canditates_match <- pblapply(working_data$doc_id, get_edges)
```

The matching process for this corpus takes around 5 minutes. In more optimized implementation, it might take even less. For the whole corpus, the process is estimated to take around 4 hours. For a brute force nearest neighbors algorithm, over 80 million distance calculations would be needed.

Network Creation

Now we will do some restructuring on the neighbor data to create a list of edges between tweets that are considered canditate neighbors. We could fine tune this step more by applying an exact knn method over all the canditates, but for the sake of simplicity, we won't. The resulting data will later be used for plotting.

```
# break outer list to make a matrix
canditates_match<-unlist(canditates_match, recursive=FALSE)
# bind pairs of tweets by row making a 2 column matrix
canditates_match <- do.call("rbind",canditates_match)
# rename some columns for clarity
colnames(canditates_match) <- c('tweet_1', "tweet_2")
# make a dataframe
canditates_match <- as.data.frame(canditates_match)
# treat tweet ids as strings
canditates_match$tweet_1 <- as.character(canditates_match$tweet_1)
canditates_match$tweet_2 <- as.character(canditates_match$tweet_2)
```

Now we can map each tweet back to the year it was posted while preserving the matches. This is extremely usefull as we can move from similar tweets to similar tweets between years.

```
canditates_match$from_year <- canditates_match %>%
  dplyr::pull(tweet_1) %>%
  plyr::mapvalues(., as.vector(working_data$doc_id), as.vector(working_data$group_year))

canditates_match$to_year <-canditates_match %>%
  dplyr::pull(tweet_2) %>%
  plyr::mapvalues(., as.vector(working_data$doc_id), as.vector(working_data$group_year))
```

Now as not all documents are matched to canditates we may get a warning message, Now we want to know the total number of edges from one year to another and also all the edges per year (know also as degree). Years with a high count of edges between them are expected to have more similar tweets between them according to lsh. Years with high degree, are expected to have higher influence to tweets of other years.

```
# unit column to enable group by sum behave as count number of pairs
canditates_match$count = 1
# number of edges per pair of years
edge_weights <- aggregate(canditates_match$count,
                          by=list(canditates_match$from_year,
                                  canditates_match$to_year),
                          FUN=sum)
# number of edges per year
node_degrees <- aggregate(canditates_match$count,
                          by=list(canditates_match$from_year),
                          FUN=sum)
```

Now the edges are symmetric as jaccard distance is symmetric. In practise this means that we see edges of \texttt{year_from, year_to} and \texttt{year_to, year_from}. This would lead to double counting edges in our plots so we need to keep only one combination per pair of years, and also conside years with no matches at all. To do so:

```
# calculate unique pair theoretically and convert to dataframe
library(arrangements)
unique_edges <- arrangements::combinations(as.character(2009:2019), k=2)
unique_edges <- as.data.frame(unique_edges)
# right join on unique pairs to keep only matching edges.
edge_weights <- merge(x = edge_weights,
                      y = unique_edges,
                      by.x = c('Group.1', 'Group.2'),
                      by.y = c('V1', 'V2'),
                      all.y = TRUE)

# order edge weights so that they are sorted in graphs.
edge_weights <- edge_weights[order(edge_weights$Group.1, edge_weights$Group.2), ]
# set non existing edges to 0.
edge_weights[is.na(edge_weights)] <- 0
# renaming columns for clarity
edge_weights %<>%
  dplyr::rename(from_year = Group.1 , to_year = Group.2, counts = x)
node_degrees %<>%
  dplyr::rename(year = Group.1, counts = x)
```

Let's see how influential each year was:

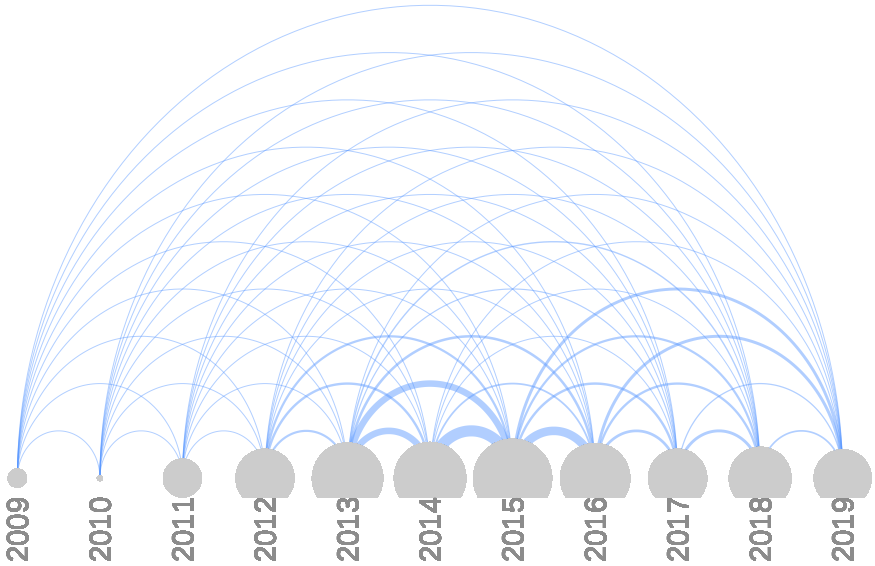
```
knitr::kable(t(node_degrees))
```

year	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
counts	30	12	117	470	1112	1206	1910	1043	476	613	441

Plotting

After constructing the graph, we can now proceed to make an arc diagram. In this diagram, years are represented as nodes and the width of the edges between them indicate the number of similar tweets. The size of each node indicates the node similarity to other years.

```
library(arcdiagram)
node_sizes <- node_degrees$counts
arcdiagram::arcplot(as.matrix(edge_weights[, c('from_year', 'to_year')]),
        ordering = node_degrees$year,
        cex.nodes = as.vector(log(node_sizes/15)+0.5),
        lwd.arcs = 0.015 * edge_weights$counts)
```



Now looking on the above plot, what do you observe?

Can you connect the result to some of your experience/theory?

In general, what observation you have on the process?