8
Data from the Social Network

In this final chapter, I will briefly pay attention to the social network. I am sorry for using this term to attract your attention, as it might mean several things to you, such as online news, Twitter, Facebook, YouTube, and the like. The fact is that each of these services has a particular policy to grant access to the data. I will not deal with this variety, as there are interesting and detailed books to guide you on this process, such as Russell and Klassen (2019).

For the sake of simplicity, or oversimplicity, let me use Twitter data to discuss the visualizations in this case. Twitter similar steps to other to allow you access to the data they keep:

- Create a developer account. You will need to apply for this and get approved. This permission may take some time (minutes, or even days).
- Register an **APP**lication that will access the Twitter data.
- Get the credentials that allow your code to interact with Twitter.

The creation of the process is documented by Twitter[1], and the steps are the same if you are in **R** or **Python**. However, there is more than one library in those languages to carry out this interaction, so I had to decide which one to use in my code. In **R**, I have used **rtweet** (Kearney et al., 2020); and for **Python**, I have used **tweepy** (Hill and Roesslein, 2020).

Every time you want to get information from a particular social web application, you may need to do some research and decide which library is more reliable and active. You may need to be very careful at this point, as the policies to access data change from time to time in these services, and some examples you could find while surfing the web might not work any more. The libraries I have chosen to collect the data are currently active (as of mid-2020),

[1] https://cran.r-project.org/web/packages/rtweet/vignettes/auth.html

https://doi.org/10.1017/9781108625425.011 Published online by Cambridge University Press

but if something is not working by the time you read this book might be due to changes in the policies in Twitter, for this case.

8.1 Getting Access

If your account as developer was approved, then you can simply copy and paste the *credentials* of your application. When using `rtweet` in **R**, you can use this code to create a token (a file) in your computer that will always be read by the `rtweet` functions:

```
> library(rtweet)
> # your credential in quotations:
> api_key = "write__yours__here"
> api_secret_key = "write__yours__here"
> access_token = "write__yours__here"
> access_token_secret = "write__yours__here"
> # creating the token (this creates a file in your computer)
> token = create_token(app = "bookVisual",
+                      consumer_key = api_key,
+                      consumer_secret = api_secret_key,
+                      access_token = access_token,
+                      access_secret = access_token_secret,)
```

In **Python**, I decided a different strategy. I created a text file with my access information formatted as a dict, then I just read the file and use the information there:

```
import json
import tweepy

# your credentials as a dictionary in a text file:
keysAPI = json.load(open('keysAPI.txt','r'))
api_key = keysAPI['consumer_key']
api_secret_key = keysAPI['consumer_secret']
access_token = keysAPI['access_token']
access_token_secret = keysAPI['access_token_secret']

# authorizing your application:
auth = tweepy.OAuthHandler(api_key, api_secret_key)
auth.set_access_token(access_token, access_token_secret)

# some extra attributes
api=tweepy.API(auth,
               retry_count=3,
               timeout=600,
               wait_on_rate_limit=True,
               wait_on_rate_limit_notify=True,
               parser=tweepy.parsers.JSONParser())
```

The **Python** option has similar code, but it has a piece of code at the end specifying what to do when you reach the time limit Twitter gives you. Also, you tell your API to collect the data into a dict structure in the `parser` option (via `JSONParser()`). The library `rtweet` does work like this, the "waiting" capability is available only in some functions (you will need to create more code for "waiting"). Remember that `rtweet` will return a data frame, as dicts are not supported in **R**, while you will get a dictionary in **Python**.

I have previously taken the time to organise a data frame with the Twitter accounts of the presidents or heads of government/state of the Americas (North, Central, South and the Caribbean), take a look:

```
> link1="https://github.com/resourcesbookvisual/data"
> link2="/raw/master/PresidentsTwitter.xlsx"
> LINK=paste0(link1,link2)
> library(rio)
> twusers=import(LINK)
> # check first two columns
> head(twusers[,c(1,2)])
```

```
         twitter                  country
1 MartinVizcarraC                  Per\'{u}
2 realDonaldTrump United States of America
3   JustinTrudeau                   Canada
4        SkerritR                 Dominica
5     chansantokhi                 Surinam

6   DrKeithRowley        Trinidad y Tobago
```

I can also upload the file in **Python**:

```
import pandas as pd

link1="https://github.com/resourcesbookvisual/data"
link2="/raw/master/PresidentsTwitter.xlsx"
LINK=link1 + link2

twusers=pd.read_excel(LINK)
```

You may need a different list when you read this book, but the strategy will be the same.

8.2 Getting Texts

Using the data from `twusers`, let me get the *Tweets* from *Donald Trump*.

```
> library(rtweet)
> trumpTweets = get_timeline("realDonaldTrump", n = 2)
```

The function `get_timeline` will bring at most two hundred recent tweets. You will get a *tibble* which I will just turn into a basic data frame. You will get several columns, but I will just select some:

```
> selection=c("created_at",
+             "text",
+             "is_retweet",
+             "favorite_count",
+             "retweet_count")
> trumpDF=as.data.frame(trumpTweets[,selection])
```

The column `created_at` includes *date* and *time*; if you want to create separate fields for this column, you can consider doing that with the help of *lubridate* (Grolemund and Wickham, 2011). Notice that using `wday()` function as it is will assign Sunday as the number **1** and Saturday as number **7**:[2]

```
> library(lubridate)
> trumpDF$Date=date(trumpDF$created_at)
> trumpDF$Hour=hour(trumpDF$created_at)
> trumpDF$Day=wday(trumpDF$created_at)
> #saving the selected info:
> write.csv(trumpDF,"trumps.csv",row.names = F)
```

Python will load its `trumpTweets` as a dict, which follows a structure similar to a tweet. Let me turn my dict into a data frame, so I will get the the data frame `trumpDF` like I did in R[3]:

```
# get timeline
who='realDonaldTrump'
trumpTweets = api.user_timeline(screen_name = who,
                                count = 2,
                                tweet_mode="extended")
# create data frame
dates=[t['created_at'] for t in trumpTweets]
text=[t['full_text'] for t in trumpTweets]
likes=[t['favorite_count'] for t in trumpTweets]
rts=[t['retweet_count'] for t in trumpTweets]

trumpDF=pd.DataFrame({'created_at':dates,
                      'text':text,
                      'retweet_count':rts,
                      'favorite_count':likes})
```

Now, let me create the columns with date, hour and day of the week, as well as the columns flagging which text is a re tweet:

```
from datetime import datetime as dt

trumpDF['created_at']=pd.to_datetime(trumpDF['created_at'],
                                     infer_datetime_format=True)
trumpDF['Date']=[dt.date(d) for d in trumpDF['created_at']]
trumpDF['Day']=[dt.date(d).isoweekday() for d in trumpDF['created_at']]
trumpDF['Hour'] = [dt.time(d).hour for d in trumpDF['created_at']]

# a column for flagging a retweet.
trumpDF['is_retweet'] = trumpDF.text.str.startswith('RT')

# saving the file (commented just to avoid rewriting it)
#trumpDF.to_csv("trumps.csv",index=False)
```

[2] If you need Monday to be the *first* day of the week, just add `week_start = 1`.
[3] Notice that using `isoweekday` will assign Monday as number **1** and Sunday as number **7**.

As you have seen, I saved the data in the file `trumps.csv` (I did not do it in **Python** to avoid overwriting the file I created in **R**), I will not collect tweets again, and I will only use the ones in that file for the next section.

8.3 Visuals Based on Text

Twitter, like other social data services, collects messages. Let me open the file I saved on *GitHub* and keep the texts that are not retweets:

```
> link3="/raw/master/trumps.csv"
> trumpLink=paste0(link1,link3)
> allTweets=read.csv(trumpLink,stringsAsFactors = F)
> DTtweets=allTweets[allTweets$is_retweet==FALSE,] #no retweets
> row.names(DTtweets)=NULL
```

Let me do the same in **Python**:

```
#loading
link3="/raw/master/trumps.csv"
trumpLink=link1 + link3
allTweets=pd.read_csv(trumpLink)

#no retweets
DTtweets=allTweets[~allTweets.is_retweet]
DTtweets.reset_index(drop=True,inplace=True)
```

The column `text` has the tweet messages. These messages are not yet ready to be turned into a visual. This is one of them:

```
> DTtweets$text[49]
```

```
[1] "Big Stock Market Numbers!"
```

Twitter texts are full of characters that are not needed by the researcher; but researchers are the ones who decide what is relevant or not. In general, you need at this point to get rid of what you consider not relevant. Generally, you should remove characters using some regular expressions:

- Emoticons. People use emoticons in their Tweets. Unless you translate their meaning, these are not to be analyzed. Use this code to get rid of them:[4]

  ```
  > DTtweets$text=gsub("[^\x01-\x7F]", "", DTtweets$text)
  ```

  Some **R** users have some strategies to translate emoticons (Peterka-Bonetta, 2017, 2019). **Python** has a package that also offers translation of the emoticons (Kim, 2015).

[4] Unfortunately, I can guarantee this code will erase every emoji, as there are more complex emojis and similar ornamental symbols appearing every now and then.

- URLs. Generally, you do not need the URLs.

```
> DTtweets$text=gsub("http\\S+\\s*","",  DTtweets$text)
```

- Special characters. Pay attention to symbols like &, >, or <, which may need to be replaced or eliminated like this:

```
> DTtweets$text=gsub("&amp;", "and", DTtweets$text) #replaced
> DTtweets$text=gsub("&lt;|&gt;", "", DTtweets$text) #eliminated
```

- Optional elements. There are some optional cleaning techniques that might be needed:
  - Users. You can delete references to other Twitter users, if you believe they do not matter:

  ```
  > DTtweets$text=gsub("@\\w+", "", DTtweets$text)
  ```

  - Hashtags. You can delete the hashtags, if you believe they do not matter:

  ```
  > DTtweets$text=gsub("#\\w+", "", DTtweets$text)
  ```

The previous steps for preparing the text can be done in **Python** like this:

```
# just readability
dirtyVar=DTtweets.text

# cleaning steps
DTtweets.loc[:,['text']]=dirtyVar.str.replace('[^\x01-\x7F]','')
DTtweets.loc[:,['text']]=dirtyVar.str.replace('http\\S+\\s*','')
DTtweets.loc[:,['text']]=dirtyVar.str.replace('&amp;','and')
DTtweets.loc[:,['text']]=dirtyVar.str.replace('&lt;|&gt;','')

### optional steps
#DTtweets['text']=dirtyVar.str.replace('@\\w+','')
#DTtweets['text']=dirtyVar.str.replace('#\\w+','')
```

You may decide you need users or hashtags; if so, keep them. In this example, I decided not to remove them (the @ and # symbols will be removed in a later step, but not the text after them).

8.3.1 Text as Clouds

Clouds of words, or *wordclouds*, are a very common visual to inform you about the text present in your messages. They are not meant as a precise visual to explain what the conversation in Twitter is about, or any collection of texts from other social media, but as always, they complement other visuals. The main strength of wordclouds is their ease of use for describing (similar to a barplot). They do not require the audience to know math or statistics. However, clouds can be frustrating if you do not see what you expect; or confusing if no patterns appear, or the default configuration does not help reveal those patterns.

I need to follow a couple of steps before I produce my cloud for the Twitter text. Let me deal with word cases and punctuation. I need every word to be in upper or lower case, while getting rid of punctuation symbols. That is easily achieved with one function `unnest_tokens` from *tidytext* (Queiroz et al., 2020):[5]

```
> library(tidytext)
> library(magrittr)
> DTtweets_Words = DTtweets %>%
+                 unnest_tokens(output=EachWord, # for DTtweets_Words
+                               input=text,      # from DTtweets
+                               token="words") # for tokenization
> # result
> head(DTtweets_Words[,-c(1,2)],10)
```

```
    favorite_count retweet_count Hour Day       Date    EachWord
1            18714          5305   23   5 2020-08-13 donyoungak
1.1          18714          5305   23   5 2020-08-13     really
1.2          18714          5305   23   5 2020-08-13   produces
1.3          18714          5305   23   5 2020-08-13        for
1.4          18714          5305   23   5 2020-08-13     alaska
1.5          18714          5305   23   5 2020-08-13         he
1.6          18714          5305   23   5 2020-08-13         is
1.7          18714          5305   23   5 2020-08-13         an
1.8          18714          5305   23   5 2020-08-13 incredible

1.9          18714          5305   23   5 2020-08-13 congressman
```

As you see above, `DTtweets_Words` has the same columns as `DTtweets` but it has been greatly modified. The new column `EachWord` has one row for each word in the tweets, as I requested *tokenization* as words. Keep in mind that words can repeat as they are generated per tweet. Notice they all are in lower case and punctuation symbols have been deleted. Let me keep using regular expression, basic string functions, and *Pandas* in **Python**:

```
# punctuation
import string
PUNCs=string.punctuation # '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
DTtweets.loc[:,['text']]=dirtyVar.str.replace('['+PUNCs+']', '')

# to lower case
DTtweets.loc[:,['text']]=dirtyVar.str.lower()
```

Python has done the cleaning in the tweet column, now let me tokenize the tweets:

```
#tokenize into a list
DTwordsList = " ".join(DTtweets.text).split()
```

[5] Notice that I got rid of some special characters before using this function, as they will not be properly erased.

I have tokenized the whole text into a list, so **Python** has not modified `DTtweets` as **R** did. Notice that you can alter the order of these steps, and you might get different amounts of tokens. In **Python**, I got twelve elements less; that difference occurs when you replace punctuation symbols, for example **R** will split `9:00` into `9` and 00, while my **Python** code will simply turn `9:00` into `900`.

I am not finished yet. Another common step is removing the stop words. These are words that are not needed for common analysis, like pronouns, prepositions, interjections, etc. The library *tidytext*, recently loaded, has a file for that:

```
> # calling the file
> data(stop_words)
> # seeing some rows
> head(stop_words)
```

```
# A tibble: 6 x 2
  word      lexicon
  <chr>     <chr>
1 a         SMART
2 a's       SMART
3 able      SMART
4 about     SMART
5 above     SMART
6 according SMART
```

So, the logical step here is simply to keep the words in `DTtweets_Words` that are *not* in the *stop_words*.[6] The function `anti_join` from *dplyr* (Wickham et al., 2020a) will be helpful:

```
> library(dplyr)
> # The column 'word' from 'stop_words' will be compared
> # to the column 'EachWord' in 'DTtweets_Words'
> DTtweets_Words = DTtweets_Words %>%
+   anti_join(stop_words,
+             by = c("EachWord" = "word"))
```

The data frame `DTtweets_Words` has reduced its previous version, and I will use its updated column `EachWord` to produce a frequency table of the words it contains.

```
> FTtrump = DTtweets_Words %>%
+   dplyr::count(EachWord, sort = TRUE)
```

The object `FTtrump` is a data frame representing a frequency table, which you can use to make barplots or any other visual mentioned in Chapter 4:

[6] The *stop_words* from *tidytext* is a *data frame*, so you can add elements to it as needed, using *rbind* or similar.

```
> head(FTtrump)
```

```
         EachWord  n
1             bus 20
2          people 18
3           usdot 17
4  infrastructure 14
5         service 14
6         support 13
```

I already have a list in **Python** (`DTwordsList`) with the tokens I will use for my wordcloud. I just need to remove the stop words and create the frequency table of words. Let me do the first step using the *nltk* library (Bird et al., 2009):[7]

```
from nltk.corpus import stopwords
STOPS = stopwords.words('english')
DTwordsList=[word for word in DTwordsList if word not in STOPS]
```

The last step may create discrepancies, as the stop words need not be the same across libraries in **R** or **Python**. Now let me create the frequency table:

```
FTtrump={word:DTwordsList.count(word) for word in DTwordsList}
```

The frequency table in **Python** is a dictionary, where the key is the word token, and the item value is the frequency of that token.

Let me first use the frequency table `FTtrump` from **R** to produce the code for our cloud. Before that, make sure you have installed the library *wordcloud2* (Lang and Chien, 2018), this time directly from its *GitHub* repository:

```
> library(devtools) # needed for "install_github()"
> install_github("lchiffon/wordcloud2")
```

Now, the code for the word cloud:

```
> library(wordcloud2)
> # option for shape are:
> # cardioid,diamond,triangle-forward,triangle,pentagon or star.
>
> wc1=wordcloud2(data=FTtrump,
+               size=1,
+               minSize = 0,
+               fontFamily = 'Arial',
+               color='random-light',
+               backgroundColor = "white",
+               shape = 'circle')
```

The cloud `wc1` can be seen in Figure 8.1.

[7] The *stop.words* from *nltk* is a *list*, so you can add elements to it as needed using the *append* function or similar.

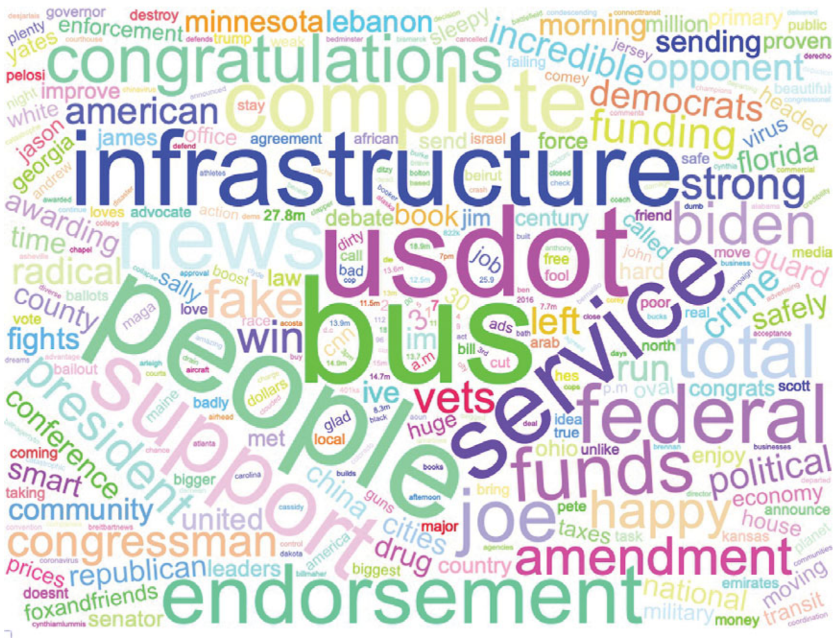

Figure 8.1 Wordcloud of all acceptable words from the last 200 tweets from Donald Trump's public account (from August 18 to August 14, 2020) collected using `get_timeline` from *rtweet*. (Kearney et al., 2020).

Let me use now the dictionary we created in **Python**, also named `FTtrump`, to create my cloud using the *wordcloud* library (Mueller, 2018):

```
import matplotlib.pyplot as plt
from wordcloud import WordCloud

wc1 = WordCloud(background_color='white')
wc1.generate_from_frequencies(frequencies=FTtrump)
plt.figure()
plt.imshow(wc1, interpolation="bilinear")
plt.axis("off")
plt.show()
```

Notice the use of `interpolation` in the code above. You might need this as the image you want to show may need to have its quality adjusted if scaled up or down.[8]

[8] In `https://matplotlib.org/3.3.1/gallery/images_contours_and_fields/interpolation_methods.html` you can find further details.

Most examples using the **Python** version do not compute the frequency table, and also make use of *wordcloud*'s own stop words.[9] This alternative version may look like this:

```
from wordcloud import STOPWORDS

wc1 = WordCloud(background_color='white',
                stopwords=STOPWORDS, #its own list
                collocations=False) # no bigrams
wc1.generate(" ".join(DTtweets.text))
plt.figure()
plt.imshow(wc1, interpolation="bilinear")
plt.axis("off")
plt.show()
```

In the previous code, I set the argument `collocations` as `False`; because by default the function `WordCloud` will try to detect and count couples of words (bigrams) (this argument is ignored when you use frequencies, as I did at first).

You need to decide if your wordclouds will be just a decorative piece, which in fact they can effectively be, or they actually help you to support decision-making. That requires you to play with the previous setting a little. A couple of things to try are subsetting `FTtrump` to avoid least common words, and/or choosing a different color combination. Let me prepare a code to keep only the words which have a frequency higher than four, and let me color the words using different levels of red.

```
> library(RColorBrewer)
> #subsetting
> FTsub=FTtrump[FTtrump$n>4,]
> colorQuant = length(unique(FTsub$n))
> #new colors
> newColors=brewer.pal(9,"Reds")
> palette = colorRampPalette(newColors)(colorQuant)[FTsub$n]
> #new version
> wc2=wordcloud2(FTsub, color=palette)
```

The cloud `wc2`, with less words and the *Reds* sequential color palette (Brewer, 2009), can be seen in Figure 8.2. Notice that I have chosen `9` for the color palette because that is the top value accepted.

I will offer two strategies to replicate 8.2 in **Python**. The first one will simply subset the dictionary with same condition (counts greater than 4), requesting the *Reds* color palette.

[9] The *STOPWORDS* from *wordcloud* is a *set*, so you can add elements to it as needed using the *update* function or similar.

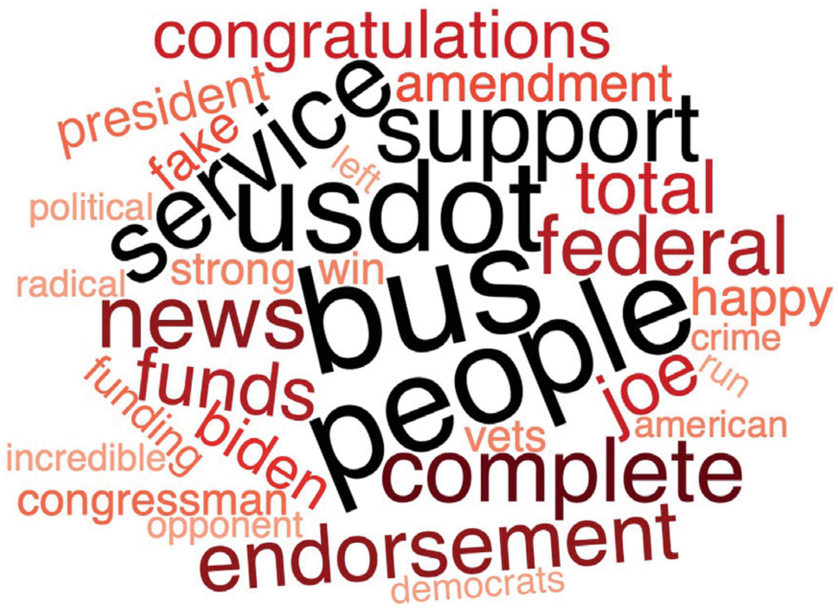

Figure 8.2 Wordcloud of words whose frequency is greater than four from the last 200 tweets from Donald Trump's public account (from August 18 to August 14, 2020) collected using `get_timeline` from *rtweet* (Kearney et al., 2020) Colors from *Reds* brewer palette (Brewer, 2009).

```
#subsetting
FTsub={k:v for k, v in FTtrump.items() if v>4}

#replotting
wc2 = WordCloud(background_color='white',
                colormap="Reds")
wc2.generate_from_frequencies(frequencies=FTsub)
plt.figure()
plt.imshow(wc2, interpolation="bilinear")
plt.axis("off")
plt.show()
```

The previous code will not give you what you might expect. The previous code will assign color randomly to the words, so it will not give you what you see in Figure 8.2. If you need to correlate color intensity with word frequency you need to use this function:

```
#recoloring function
def myColor_func(word, **kwargs):
     # key of max value
     kMax=max(FTsub.items(), key=operator.itemgetter(1))[0]
     # 0 for red /  120 for green / 240 for blue
     return "hsl(0, 100%%, %d%%)" % (5*FTsub[kMax]/FTsub[word])
```

As you see above, you need to create colors using an *hsl* format (hue, saturation, and lightness). The *hue* goes from *0* to *360*, where *0* is red, *120* is green, *240* is blue; *saturation* is a percentage value where *100 percent* is the full color, and any decrease will add some shade of gray (*0* will turn any color to gray); *lightness* is also a value from *0 percent* (black) to *100 percent* (white). In my function, you need to include a value for the *hue* (I chose *0* for red), the 100 percent saturation will keep the full *hue*, and the lightness will vary according to the frequency of the word (I wrote a simple function, you may try a different one). Let me use the function `myColor_func` in the `color_func` argument of the `WordCloud` function.

```
#replotting
wc3 = WordCloud(background_color='white',
                color_func=myColor_func)
wc3.generate_from_frequencies(frequencies=FTsub)
plt.figure()
plt.imshow(wc3, interpolation="bilinear")
plt.axis("off")
plt.show()
```

It requires some time to produce an informative wordcloud, I just recommend using it wisely, as different people may notice different patterns (which may not help while you are presenting).

8.3.2 Text as Network

A good complement to a wordcloud can be a network of words. The basic idea is that a network, or graph, connects words that appear one after the other. This adjacency is the focus of the visual exploration.

Let me use the `DTtweets` data frame I used when I started the previous subsection which was already cleaned. From that point, let's go to every clean text and get every *two* adjacent words, or *bigrams*:

```
> DTtweets_2g = DTtweets %>%
+                unest_tokens(output=pairWords,
+                             input=text,
+                             token = "ngrams", n = 2) # 2-grams
```

In the previous code, I used `unnest_tokens` again, but this time I requested bigrams instead of simple words. The bigrams are in the column `pairWords` in the `DTtweets_2g` data frame:

```
> head(DTtweets_2g$pairWords)
```

```
[1] "great meeting"    "meeting today"    "today with"       "with the"
[5] "the coronavirus"  "coronavirus task"
```

Let me get the bigrams in **Python**. I will only use the column `text` from the data frame:

```
#bigrams per twitter (per cell)
from nltk import bigrams

theBigrams=[bigrams(eachTW.split()) for eachTW in DTtweets.text]


# list of all bigrams
from itertools import chain

pairWords = list(chain(*theBigrams))
```

I followed a couple of steps to get something similar to `pairWords` as I did in **R**. First, using the `bigrams` function from *nltk*, I got the bigrams from every tweet. The function `bigrams` will turn the input into consecutive pairs; so you need to input the right structure. If you input a string, it will turn the string into pairs of characters, that is, `bigrams("Hi Jim")` will be: [('H', 'i'), ('i', ' '),(' ', 'J'), ('J', 'i'),('i', 'm')]).

You do not want the previous outcome, so I input a list of words into `bigrams`. I got the lists of words by tokenizing each tweet text using `split`. This process creates lists each with a *generator*[10] of tuples. After that, I needed to visit every element of `theBigrams`, and make a list containing all the tuples from those lists. Since I need a simple list for my word cloud (and not a list of tuples) I had to flatten the object `pairWords`.[11] I easily did that with the function `chain` from the library *itertools*.[12]

You have bigrams as two-word strings in **R**, and as a tuple in **Python**, so far. The next step is to get the frequency table of the bigrams. Let me first show you the steps in **R**:

- Split the column `pairWords` into different columns. I will use `separate` from *tidyr* (Wickham et al., 2020b) for splitting it into columns named `word1` and `word2`; then I use `select` from *dplyr* to keep the new columns:

```
> library(tidyr)
> DTtweets_2g_only=DTtweets_2g %>%
+                 separate(pairWords, #source column
+                          c("word1", "word2"), #new columns
+                          sep = " ") %>% # split by
```

10 This is a function from *nltk* that creates a generator, which actually can produce an iterator of tuples. It is a fine way to avoid populating the memory with all the tuples, which will only be created when requested.

11 Turning a list of tuples (or other structure) into a simple list is called *flattening*.

12 The function of `chain` is turning something like `list(chain([(1,2),(3,4)],[(5,6)]))` into `[(1, 2), (3, 4), (5, 6)]`. As in this case, `theBigrams` receive a list (several arguments as a list); you use the asterisk before its name.

```
+                  select(c("word1", "word2")) # keep these
>
```

Now, I have every pair of words in two columns in the `DTtweets_2g_only` data frame.

- Make the frequency table of bigrams:

```
> FT_DT_2g = DTtweets_2g_only %>%
+             dplyr::count(word1, word2, sort = TRUE)
> #take a look
> head(FT_DT_2g)
```

```
  word1   word2  n
1  <NA>    <NA> 22
2  will      be 19
3    of     the 18
4   for     the 14
5    in     the 13

6   bus service 12
```

- My data frame `FT_trump_2g` is almost ready, I just need to get rid of the missing values:

```
> FT_DT_2g=FT_DT_2g[complete.cases(FT_DT_2g),]
```

- Next, I should remove the "stop words":

```
> FT_DT_2g = FT_DT_2g %>%
+                  filter(!word1 %in% stop_words$word) %>%
+                  filter(!word2 %in% stop_words$word)
```

Let me create the frequency table `FT_DT_2g_dict` in **Python** (and turn it into a data frame):

```
# frequency of DTrump bigrams
from collections import Counter

FT_DT_2g_dict = Counter(pairWords) #generate counter

# Turn FT_DT_2g_dict  into dataframe, naming columns
FT_DT_2g = pd.DataFrame(FT_DT_2g_dict.most_common(),
                        columns=['theBigram', 'count'])
```

I used the function `Counter` from *collections* to create the frequency table (`FT_DT_2g_dict`) which returns a dictionary. This dict will have the bigram tuple as the key and its frequency as the value. When you have the dict produced by `Counter`, it allows you to apply the function `most_common`[13] to it, and you get a list as a result. *Pandas* easily turned that list into a data frame while giving names to the columns.

[13] I used this function without arguments to get all the pairs.

Python requires another step to turn the column `theBigram` (it has the tuples) into two columns (`word1` and `word2`). I will use those two new columns to filter the rows with stop words:

```
# Turn column of tuples into separate columns
FT_DT_2g['word1'], FT_DT_2g['word2'] = FT_DT_2g.theBigram.str

# Getting rid of stopwords:
FT_DT_2g=FT_DT_2g[~FT_DT_2g['word1'].isin(STOPS)]
FT_DT_2g=FT_DT_2g[~FT_DT_2g['word2'].isin(STOPS)]
```

I hope the bigrams in each row make some sense to you. Of course, you can not synthesize much just by looking the bigram texts. The proposal here is to show how all bigrams tell you something. However, if I use a wordcloud, you may see repeated words. Then, if I want to *connect* words that are part of a bigram, a suitable option is a graph. For that, let me first install and use *igraph* (Csardi and Nepusz, 2006):

```
> library(igraph)
> DT_2g_net=graph_from_data_frame(FT_DT_2g)
```

The function `graph_from_data_frame` just read the first two columns of the data frame `FT_DT_2g` and created the graph, translating the first column as the *source* (from) and the second as the *target* (to). Then, the object `DT_2g_net` is your graph where each *node* is a word, and words connected via a *link* indicate they are a bigram in the text. I can show you that using *ggraph* (Pedersen and RStudio, 2020), an extension for *ggplot*.

```
> library(ggraph)
> # graph layout: 'ggraph' will decide this time
> layout = ggraph(DT_2g_net) + theme_void()
> # draw nodes (words) in a position based on layout
> nodes= layout + geom_node_point()
> # draw links to connect nodes
> net1 = nodes + geom_edge_link()
> # customize some text in node
> net1= net1+ geom_node_text(aes(label = name),
+                                  vjust = 1,
+                                  hjust = 1,
+                                  size=2)
```

The graph `DT_2g_net` is now the visual object `net1`. You can see the plot in Figure 8.3. Notice that the variable `name` in the aesthetics of `geom_node_text` is referring to the node name attribute of the graph created by default. Also, notice that the position of nodes was automatically picked by `ggraph`.

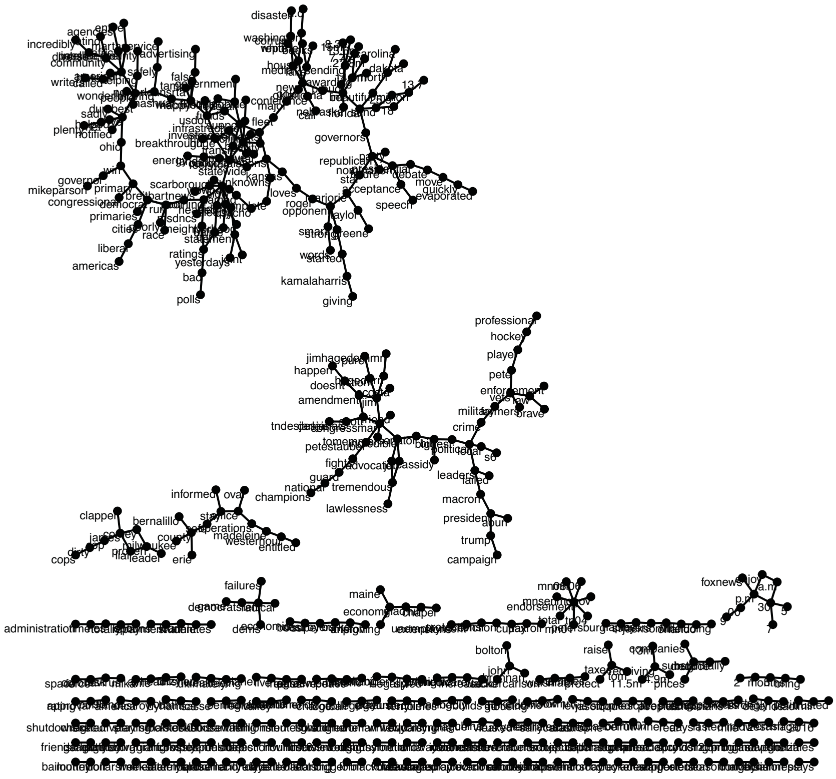

Figure 8.3 Graph of Bigrams

The graph represents all the bigrams from the last 200 tweets from Donald Trump's public account (from August 18 to August 14, 2020) collected using `get_timeline` from *rtweet* (Kearney et al., 2020). The position of the nodes was automatically picked by the function `ggraph` (*stress* algorithm (Gansner et al., 2005)).

Let me prepare my version of **Python** using *networkx* (Hagberg et al., 2020). *Networkx* can easily turn the `FT_DT_2g` data frame into a graph with its function `from_pandas_edgelist`:[14]

```
import networkx as nx

# from data frame to graph
DT_2g_net=nx.from_pandas_edgelist(df=FT_DT_2g,
                                  source='word1',
                                  target='word2')
# plotting graph (default layout)
nx.draw_networkx(DT_2g_net,
```

[14] The function `from_pandas_edgelist` requires that you indicate what columns are the source and the target, unless you had named the columns with those labels.

```
                 font_size=7,
                 edge_color='red',
                 node_color='yellow',
                 node_size=100,
                 alpha=0.9,
                 with_labels = True)
```

In the previous code, I created the *DT_2g_net* graph and used the function `draw_networkx` to create the plot. *Networkx* was not meant to be a package for beautiful plotting, it recommends using more specialized software for that, such as Cytoscape (Cytoscape Consortium, 2020) or Gephi (Bastian et al., 2009). You should not expect to get the same layout as shown in Figure 8.3, which was chosen by *igraph* in **R**; in this case, the default layout will be *spring* (Fruchterman and Reingold, 1991).

If you do not see clear labels, you can try moving them on the horizontal or vertical axis; for that, you need to use :

```
#setting size
fig, ax = plt.subplots(figsize=(10,10))

#saving layout positions
pos = nx.spring_layout(DT_2g_net)

# Plot networks
nx.draw_networkx(DT_2g_net,
                 pos,  #layout
                 edge_color='red',
                 node_color='yellow',
                 node_size=100,
                 with_labels = False,
                 ax=ax) # for matplotlib ax

# labels away from node
for word, freq in pos.items():
    x, y = freq[0]+.01, freq[1]+.01 # new pos values
    ax.text(x, y,#new positions
            s=word,#label
            horizontalalignment='center',
            fontsize=7,rotation=30)

plt.show()
```

Notice that in the previous code, I needed to save the actual positions of the nodes first, so that I could alter them later.

Let's come back to **R**. Figure 8.3 included all the data available; that may have not been the best choice. Let me create the graph `DT_2g_net3`, which will only include bigrams with a frequency higher than or equal to three:

```
> #subsetting
> FT_DT_2g3=FT_DT_2g[FT_DT_2g$n≥3,]
> DT_2g_net_sub=graph_from_data_frame(FT_DT_2g3)
> #new plot
> layout2 = ggraph(DT_2g_net_sub) + theme_void()
```

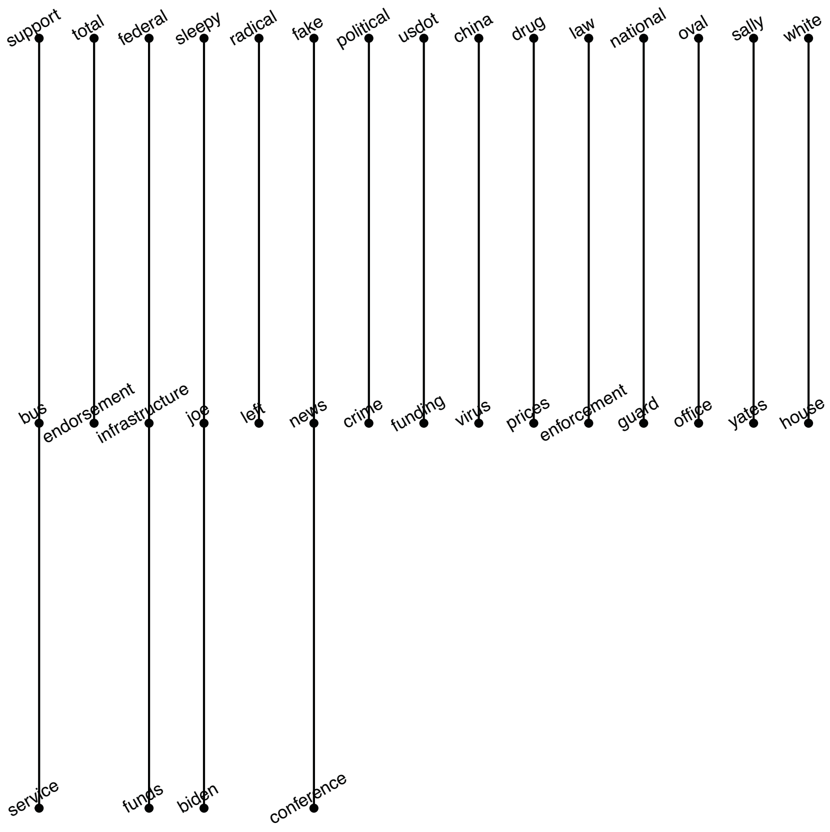

Figure 8.4 Graph of Bigrams with frequency greater than two

Bigrams from the last 200 tweets from Donald Trump's public account (from August 18 to August 14, 2020) collected using `get_timeline` from *rtweet* (Kearney et al., 2020). The position of the nodes is automatically picked by the function `ggraph`.

```
> nodes =layout2 +  geom_node_point()
> net2 = nodes +  geom_edge_link()
> net2 = net2 +  geom_node_text(aes(label = name),
+                              vjust = -0.5,
+                              hjust = 0.5,angle=30,
+                              size=3)
```

The visual object `net2` can be seen in Figure 8.4. Notice that the position of the nodes was again automatically picked by `ggraph`, here the *tree* layout was chosen (Pedersen, 2020).

You can redraw `net2` in **Python** using this code:

```
#subsetting
FT_DT_2g3=FT_DT_2g[FT_DT_2g['count']≥3]

DT_2g_net_sub=nx.from_pandas_edgelist(FT_DT_2g3,'word1','word2')

#plotting
fig, ax = plt.subplots(figsize=(6, 6))
pos = nx.spring_layout(DT_2g_net_sub)

# Plot networks
nx.draw_networkx(DT_2g_net_sub, pos,
                 edge_color='red',node_color='yellow',
                 node_size=100,with_labels = False,ax=ax)

# labels away from node
for word, freq in pos.items():
    x, y = freq[0]+.05, freq[1]+.03
    ax.text(x, y,s=word,horizontalalignment='center',
            fontsize=13,rotation=30)

plt.show()
```

I will now now focus on the relationships among network actors in this last section of the book.

8.4 Visuals Based on Actors

The previous section served as a quick introduction to a fascinating area of analysis: networks. A network is a familiar term used in science, engineering, humanities, and social sciences. If you have a network structure, composed of nodes (vertices) and links (edges) you can make several computations supported by the mathematics of graph theory, but the plotting of the network is a different story, with many algorithms competing to help you find some pattern in the complexity of networks (please read this discussion by Tarawaneh et al. (2012)). If you are interested in this topic, you should be aware that it is difficult to get a good layout when your nodes reach amounts pass the hundreds. However, let me guide you in the basic steps of organizing your network information and the steps required to get a plot.

8.4.1 Actors

Using the noun actor, instead of simply using "node" or "vertex", in social and policy research contexts may be a good idea. I think this can help you focus on the role of computational social science (Cioffi-Revilla, 2014), while allowing you to be a step away from introducing yourself into agent-based modeling (Railsback and Grimm, 2012), a topic I have not dealt with in this book.

The social media network is a great source of information about actors. In the data I presented at the beginning of this chapter, the presidents or heads of government/states of the Americas are the actors. Actors can have variables or attributes – in this case, the country they belong to. You can decide that the country is the actor, and while it can be more abstract you might have a good reason for that; however, that may change the kind of relationship that is suitable among the actors.

8.4.2 Relationships

Similarly, I prefer using the noun “relationship” instead of “edge” or “link”. As mentioned in the previous paragraph, the nature of the actor allows for some kinds of relationships. In general, they can be *directed* or *undirected*. A relationship is undirected when once it exists both actors have the same kind of connection; a great example of this is for our data on the Americas is *be neighbor of*, because if Bolivia is a neighbor of Brazil, Brazil can not avoid having the same relationship with Bolivia. On the other hand, when a relationship is directed, it is not always possible to have the same relationship reflected back. The directed relationship *supplies natural gas* may be one-directional; while the relationship *supplies workforce* may be two-directional.[15]

If the actors are people, like the political leaders we have, they might not have a neighborly relationship if they live in different countries (unless some have different places to live in). Since we have the Twitter accounts of each for these leaders, we could think of relationship *follower of*. This is an example of directed relationship which may not be a two-way one. This particular fact is very interesting in political terms, as it may reveal presidents whose tweets are of interest to some, or not. Of course, you need to be careful when you consider this as a general rule, as a political actor need not be a follower on Twitter to actually be following some other actor.

8.4.3 Making the Network

Making the network requires nodes and the connections among them. In Section 8.2, the data I collected using **R** and **Python** is not helpful in preparing a network, as it does not indicate who follows who. I need a different way to

[15] This is possible using the term workforce in a very general way, but digging further into it may make the relationships look more one-way.

get connections from Twitter, and only then I will use the functions to prepare a network, something similar to what I did with the bigrams in Section 8.3.2.

Finding Relationships

In **R**, the library *rtweet* has a function `lookup_friendships`.[16] It can tell you who follows who, given a pair of users. You can use the function like this:

```
> relationship= lookup_friendships(source='user1',target='user2')
```

If I input every possible pair[17] into that function, I can get the relationships that exist among them. You can use the function `combn` (from *utils*) to get all pairs:[18]

```
> pairs=combn(twusers$twitter,2,simplify = F)
```

Using both previous functions, I will verify the existing relationships, and I will keep the actual connections. I have organized those relationships into a data frame:

```
> link4="/raw/master/edgesAmericas.csv"
> linkEdges=paste0(link1,link4)
> relationships=read.csv(linkEdges,stringsAsFactors = F)
> head(relationships)
```

```
        source          target
1     alferdez     nayibbukele
2     alferdez   JustinTrudeau
3  JeanineAnez       IvanDuque
4  JeanineAnez realDonaldTrump
5 jairbolsonaro sebastianpinera

6 jairbolsonaro           Lenin
```

The data frame `relationships` has the edges showing who follows who; then I need a directed network:

```
> library(igraph)
> set.seed(123)
> net = graph_from_data_frame(d=relationships, #data frame
+                              vertices=twusers,#data frame
+                              directed=T)
```

16 Your alternative in *tweepy* is `show_friendship`.
17 If you do not have premium developer account, I recommend that you split these pairs into smaller chunks, both in **R** and **Python**, as the Twitter restrictions may cause your code execution to fail.
18 In **Python**, you can use the `combinations` function from the library *itertools*.

The function `graph_from_data_frame` used a couple of data frames, one with the *relationships* and one with the information about the *actors*. It is of course a directed network. Let me see what I have:

```
> summary(net)
```

```
IGRAPH fbfc7b1 DN-- 34 168 --
+ attr: name (v/c), country (v/c), president (v/c), region (v/c), sex
| | (v/c)
```

The summary presents several interesting facts. After a seven-character code (not relevant for any purpose), you see a **D** because the network is directed (the other option is *U*), and the **N** because you have vertex "names" (it is using the data frame). It is also telling you that you have 34 actors and there are 168 relationships. You see several attributes, all of them including **v/c**, which means that they are attributes of the actor (vertex) and the data type it holds, in this case a "character" (other options are **n**`umeric`, **l**`ogical`, and **x** for other)[19].

Let me create the same network using **Python**:

```
#get relationships data
link4="/raw/master/edgesAmericas.csv"
LINK=link1 + link4
relationships=pd.read_csv(LINK)

# build network
import networkx as nx
net=nx.from_pandas_edgelist(relationships,create_using=nx.DiGraph())

# make sure isolates are in the network
net.add_nodes_from(twusers.twitter)

#add attributes from data frame
### data frame as dictionary
attributes=twusers.set_index('twitter').to_dict('index')
### add attributes of nodes to network
nx.set_node_attributes(net, attributes)
```

The `net` object created using *networkx* was also created from the data frame `relationships`. The problem is that the data frame might not have some actors. The network from *igraph* in **R** included all the actor nodes because you also added the actors data frame. Then, I needed to use `add_nodes_from` to add the missing nodes. Once I had all the nodes, I added the attributes. Notice that the attributes should be input as a dictionary, that is why I used `to_dict` to convert the data frame into a dict where each row is a dict item. Since the user name in that data frame will be the key to

[19] For further details on these metadata, check the documentation of the function `print.igraph`.

each row, I needed to set the column `twitter`, where the user names are, as the index; let me show a portion of this dict:

```
In [8]: twusers.set_index('twitter').to_dict('index')
Out[8]:
{'MartinVizcarraC': {'country': 'Perú',
  'president': 'VIZCARRA, Martín',
  'region': 'SouthAmerica',
  'sex': 'male'},
 'realDonaldTrump': {'country': 'United States of America',
  'president': 'TRUMP, Donald',
  'region': 'NorthAmerica',
  'sex': 'male'},
 'JustinTrudeau': {'country': 'Canada',
  'president': 'TRUDEAU, Justin',
  'region': 'NorthAmerica',
  'sex': 'male'},
```

You do not have a function to summarize the `net` object from *networkx*, but you can find out about nodes and edges with other functions. If you use `net.nodes(data=True)`[20] you will get all the information available for every node; similarly, you can use `net.edges(data=True)`[21] to see what edges you have and their attributes.

8.4.4 Network Layout

Let me start with a basic plot using what we learned in Section 8.3.2.

```
> library(ggraph)
> layoutPresi = ggraph(net) + theme_void()
> nodesPresi= layoutPresi + geom_node_point()
> netPresi = nodesPresi + geom_edge_link()
> netPresi= netPresi+ geom_node_text(aes(label = name))
```

You can see a plot for the network `netPresi` in Figure 8.5. Remember that *ggraph* picks a layout automatically if you do not select one.[22]

Networkx has limited capabilities for drawing networks, and it recommends[23] exporting the network to be used for other more specialized programs, as mentioned before. If you decide to leave **Python** and use another type of drawing software, as recommended by itself, a good option is saving this *networkx* object as *graphml* file (Brandes et al., 2002):

```
nx.write_graphml_lxml(net, "presiAmericas.graphml")
```

20 In **R**, you can check each actor using `V(net)`.
21 In **R**, you can check each relationship using `E(net)`.
22 The library *ggraph* can use the layouts available in *igraph*.
23 https://networkx.github.io/documentation/stable/reference/drawing.html

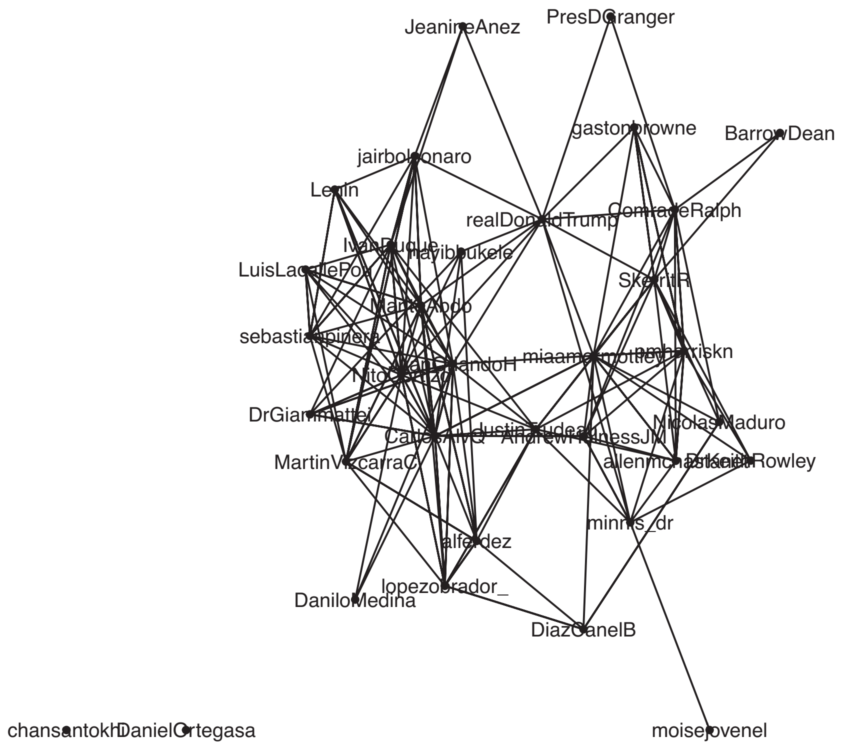

Figure 8.5 Network of Presidents of the Americas

The graph shows which president follows another president. The position of the nodes was automatically picked by the function `ggraph` (*stress* algorithm (Gansner et al., 2005)).

Let me continue in **Python**, and as you know, this is the most basic command:

```
nx.draw_networkx(net)
```

The previous plot could be improved if you manage[24] to use *graphviz* functionalities (Ellson et al., 2003). For that, you will need to install the libraries *pydot* (Carrera, 2018) and *graphviz* (Bank, 2020). This code will give you different results:

```
pos = nx.nx_pydot.graphviz_layout(net)
nx.draw_networkx(net,pos=pos)
```

[24] I have only tested this on Mac OS, so I will not use this capability again.

The result of the previous code will not look as nice as Figure 8.5. However, the default layout, *neato* (North, 2004),[25] will give a better result than the basic option. You can make some adjustments using *matplotlib* and some parameters in *networkx*:

```
import matplotlib.pyplot as plt

pos = nx.nx_pydot.graphviz_layout(net)
plt.figure(figsize=(8, 8))
plt.axis('off')
nx.draw_networkx(net,
                 pos=pos,
                 with_labels=True,
                 node_size=25,
                 edge_color='b')
plt.show()
```

Layouts depend on algorithms that decide the position of the nodes (as coordinates). They seek to position the node to avoid the overlapping of links while trying to reveal some structural pattern. There are several layout algorithms that network scientists use; some are available in **Python** and some in **R**.

8.4.5 Coloring Actors and Relationships

Once a layout has been chosen, you may want to use color to help see some patterns. You need to use some attribute for coloring the nodes. My original data frame had the attribute *region*, and it is part of the `net` object.

```
> nodesPresi2= layoutPresi + geom_node_point(aes(colour=region),
+                                            size=3)
> netPresi2= nodesPresi2 + geom_edge_link(color='grey90')
> netPresi2=netPresi2+ geom_node_text(aes(label = president),
+                                    size=3,
+                                    color="gray50",
+                                    repel=T)
> netPresi2=netPresi2 + scale_color_brewer(name="Region",
+                                        type = 'qual',
+                                        palette ="Set1")
```

You can see the visual for `netPresi2` in Figure 8.6. I have changed the color saturation by turning the black links into light gray. I have also shortened the actor names. Since the attribute region is an aesthetics, I used color as an aesthetics.

Drawing networks is pretty easy using *ggraph* if you are already familiar with *ggplot*. An alternative code in **Python** would need much more work. That

[25] Other options available are 'dot', 'twopi', 'fdp', 'sfdp', 'circo'.

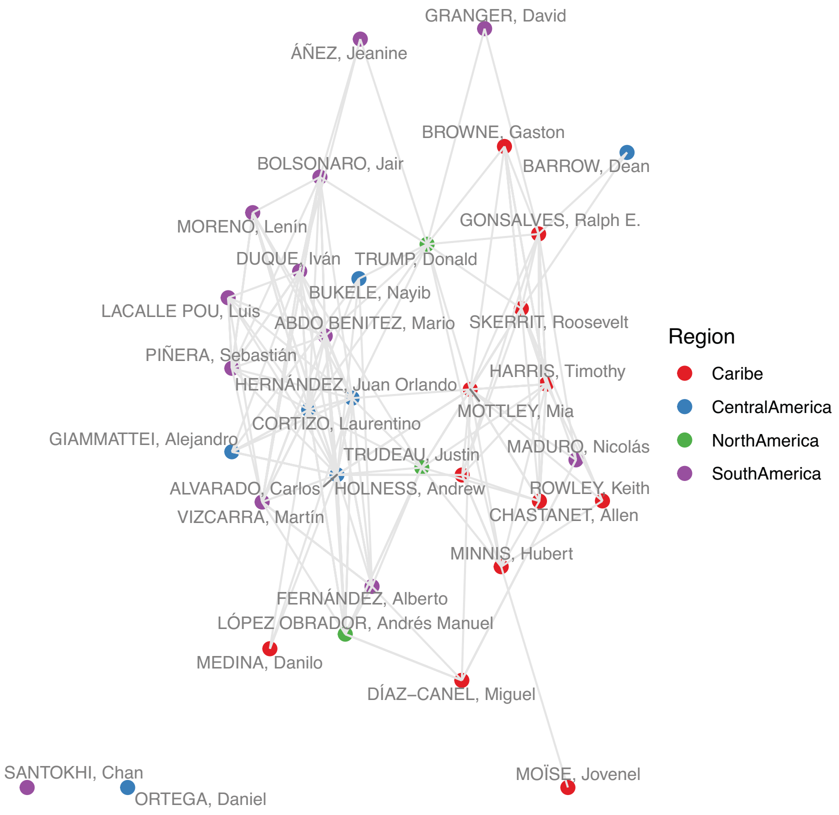

Figure 8.6 Network of Presidents of the Americas colored by region in the Americas

The graph shows which president follows another president. The position of the nodes was automatically picked by the function `ggraph` (*stress* algorithm (Gansner et al., 2005)).

amount of work requires you abandon the function `draw_networkx` and draw the nodes, the links, and the labels separately. Take a look:

```
from matplotlib.colors import rgb2hex
import matplotlib.pyplot as plt

plt.figure(figsize=(10,10))

#layout
pos=nx.spring_layout(net,k=1) #higher k gives more spread of nodes.
#prepare coloring by region
allValues=[n[1]['region'] for n in net.nodes(data=True)]
levels=pd.unique(allValues).tolist()
palette = plt.get_cmap("Set2") #palette

#drawing nodes - coloring nodes per attribute
```

```
for categoryChosen in levels:
    #presidents of the same region
    nodesChosen=[node[0] for node in net.nodes(data=True)
                 if node[1]['region'] in categoryChosen]
    #choosing color for these nodes
    colorChosen=rgb2hex(palette(levels.index(categoryChosen)))
    # draw chosen nodes
    nx.draw_networkx_nodes(net,pos=pos,node_size=100,
                           node_color=colorChosen,
                           nodelist=nodesChosen,
                           label=categoryChosen) # for legend!
#draw edges
nx.draw_networkx_edges(net,pos=pos,edge_color="silver")

#draw labels for nodes
#using President name (not Twitter username)
newLabels = {n[0]:n[1]['president'] for n in net.nodes(data=True)}
nx.draw_networkx_labels(net,pos=pos,font_size=8,
                        font_color='grey',
                        font_weight='bold',
                        labels=newLabels)

# requesting legend (needs "label" in nodes above)
plt.legend(markerscale=1, loc="best")
plt.show()
```

The code in **Python** required more steps than in **R**. Particularly, it needed a loop structure to code the nodes based on an attribute. Also, the code above required using the function `rgb2hex()` to get the right color code from the palette.[26] Notice also the use of `label` while drawing the nodes. That is not to label the nodes, but to create the legend. Also, notice that I changed the default labels using the dictionary `newLabels`. The keys of the dictionary are the node names and the values are the new label. If you just want to keep the node names, do not use the argument `labels` when drawing the labels.

8.4.6 Using Size Attributes

I will explore how I can use size in the network nodes. I do not have a quantitative attributes in my data, so I need to add some.

Centrality, a qualitative characteristic that tries to reveal the importance of network actors based on their relationships, has several quantitative ways to be computed, each highlighting a particular centrality concept.[27] Let me compute degree centrality with both of its variations for a directed network:

```
> # measure of being followed
> indeg = degree(net, mode="in",normalized = T)
> V(net)$indegree = indeg
> # measure of being a follower
> outdeg = degree(net, mode="out",normalized = T)
```

26 You need hexadecimal, but you get RGB from `plt.get_cmap()`.
27 The most common are *closeness*, *betweenness*, and *degree*.

```
> V(net)$outdegree = outdeg
> # reloading the network (new attributes)
> layoutPresi=ggraph(net) + theme_void()
>
```

Let me also update the *networkx* object:

```

nx.set_node_attributes(G=net,
                       values=nx.in_degree_centrality(net),
                       name='indegree')

nx.set_node_attributes(G=net,
                       values=nx.out_degree_centrality(net),
```

I will use that new attribute to vary the size of the nodes in **R**:

```
> sizeNodesIn=layoutPresi + geom_edge_link(color='grey90')
> sizeNodesIn=sizeNodesIn + geom_node_point(aes(size=(1+indegree)^10,
+                                               color=region)) +
+                           labs(size="In Degree") #legend title
> sizeNodesIn=sizeNodesIn + geom_node_text(aes(label = president),
+                                          color="gray50",
+                                          repel=T)
> sizeNodesIn=sizeNodesIn + scale_color_brewer(name="Region",
+                                              type = 'qual',
+                                              palette ="Set1")
```

The object `sizeNodesIn` is represented in Figure 8.7. Notice that in the size aesthetics of `geom_node_point` I added `1` to the actual value, since some nodes may have a zero value.

Notice that Figure 8.7 has two legends. The one for sizes may be confusing; besides the default legend name, the numeric values are not the real measurement values. Also, notice that, in contrast to Figure 8.6, I have plotted the links before the nodes, which prevents them covering the node.

Let me try **Python**, using the Kamada-Kawai algorithm for the layout (Kamada and Kawai, 1989). I would need several more lines of code to reproduce Figure 8.7. I will reuse some of the code I used to replicate Figure 8.6, adding a code inside the loop to get the `indegree` values which will alter the node sizes.

```
from matplotlib.colors import rgb2hex

plt.figure(figsize=(8,8))
pos=nx.kamada_kawai_layout(net) #layout
#prepare coloring by region
allValues=[n[1]['region'] for n in net.nodes(data=True)]
levels=pd.unique(allValues).tolist()
palette = plt.get_cmap("Set2")

for categoryChosen in levels:
    nodesChosen=[node[0] for node in net.nodes(data=True)
                  if node[1]['region'] == categoryChosen]
```

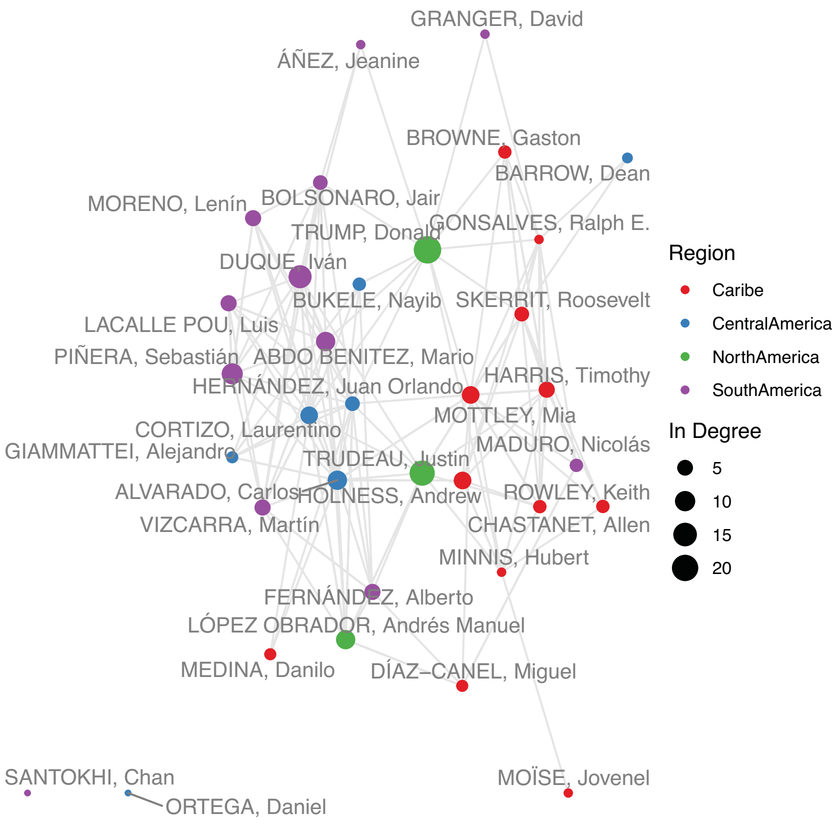

Figure 8.7 Network of Presidents of the Americas

Node sizes are based on in-degree centrality; nodes are colored by region in the Americas. The bigger the node, the more that president is followed. The position of the nodes was automatically picked by the function `ggraph` (*stress* algorithm (Gansner et al., 2005)).

```
    colorChosen=rgb2hex(palette(levels.index(categoryChosen)))
    #chosing sizes of the actors (in a list)
    sizesChosen=[(1+x[1]['indegree'])**20
                 for x in net.nodes(data=True)
                 if x[0] in list(nodesChosen)]
    # drawing the NODES
    nx.draw_networkx_nodes(net,pos=pos,
                           node_size=sizesChosen, #vector of sizes
                           node_color=colorChosen,
                           nodelist=nodesChosen,
                           label=categoryChosen)
#drawing edges and labels
nx.draw_networkx_edges(net,pos=pos,edge_color="silver")
nx.draw_networkx_labels(net,pos=pos,font_size=8,font_color='grey')

# customizing legend
```

```
MyLegend = plt.legend(loc="best")
for handle in MyLegend.legendHandles:
    handle.set_sizes([20.0])

plt.show()
```

Notice the customization of the legend in my previous **Python** code, almost at the end. I used `handle.set_sizes` to fix the size of the legend symbols; if you do not use that code, you will see that sizes will be proportional to the sizes in the plot.

The decision to vary the size of the nodes might not look as well as expected, as it might become difficult to decode variability effectively. So, let me use the text font size instead:

```
> nodesPresiIn=layoutPresi + geom_node_point(aes(color=region),
+                                           size=4)
> netPresiIn=nodesPresiIn + geom_edge_link(color='grey90')
> netPresiIn=netPresiIn + geom_node_text(aes(label=country,
+                                           size=(1+indegree)^10),
+                                       color="gray50",
+                                       repel=T)
> netPresiIn=netPresiIn + scale_color_brewer(name="Region",
+                                           type = 'qual',
+                                           palette ="Set1")
> netPresiIn=netPresiIn + guides(size=F) #NO legend for size
```

As we have a directed network, let me use also the column *outdegree* `netPresiOut`.

```
> netPresiOut=layoutPresi + geom_node_point(aes(color=region),
+                                          size=4)
> netPresiOut=netPresiOut + geom_edge_link(color='grey90')
> netPresiOut=netPresiOut + geom_node_text(aes(label = country,
+                                        size=(1+outdegree)^10),
+                                        color="gray50",
+                                        repel=T)
> netPresiOut=netPresiOut + scale_color_brewer(name="Region",
+                                                type = 'qual',
+                                                palette ="Set1")
> netPresiOut=netPresiOut + guides(size=F) #NO legend for size
```

You can see the object `netPresiOut` in Figure 8.9.

Let me use an alternative **Python** version of Figure 8.8:

```
from matplotlib.colors import rgb2hex
import matplotlib.pyplot as plt

plt.figure(figsize=(8,8))

#setting layout and palette
pos=nx.kamada_kawai_layout(net)
allValues=[n[1]['region'] for n in net.nodes(data=True)]
levels=pd.unique(allValues).tolist()
palette = plt.get_cmap("Set2")

#drawing nodes and edges
```

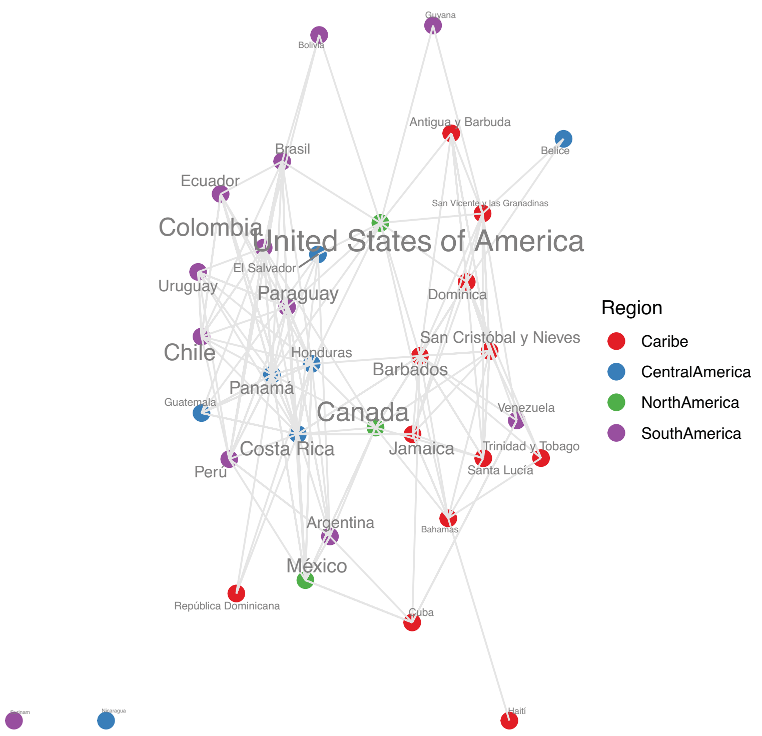

Figure 8.8 Network of Presidents of the Americas

Sizes are based on in-degree centrality; nodes are colored by region in the Americas. The bigger the text, the more that president is followed. The position of the nodes is based on *stress* algorithm (Gansner et al., 2005).

```
for categoryChosen in levels:
    nodesChosen=[node[0] for node in net.nodes(data=True)
                 if node[1]['region'] == categoryChosen]
    colorChosen=rgb2hex(palette(levels.index(categoryChosen)))
    nx.draw_networkx_nodes(net,pos=pos,node_size=100,
                           node_color=colorChosen,
                           nodelist=nodesChosen,
                           label=categoryChosen)
nx.draw_networkx_edges(net,pos=pos,edge_color="silver")

#drawing each node label - SIZE by indegree!!
for user in net.nodes():
    sizeLabel=net.nodes(data=True)[user]['indegree']+1
    nodeLabel=net.nodes(data=True)[user]['country']
    nx.draw_networkx_labels(net, pos=pos,
                            labels={user:nodeLabel},
                            font_size=sizeLabel**8) #varying sizes
# requesting legend
# marker in legend same size as actor node
plt.legend(markerscale=1, loc="best")
plt.show()
```

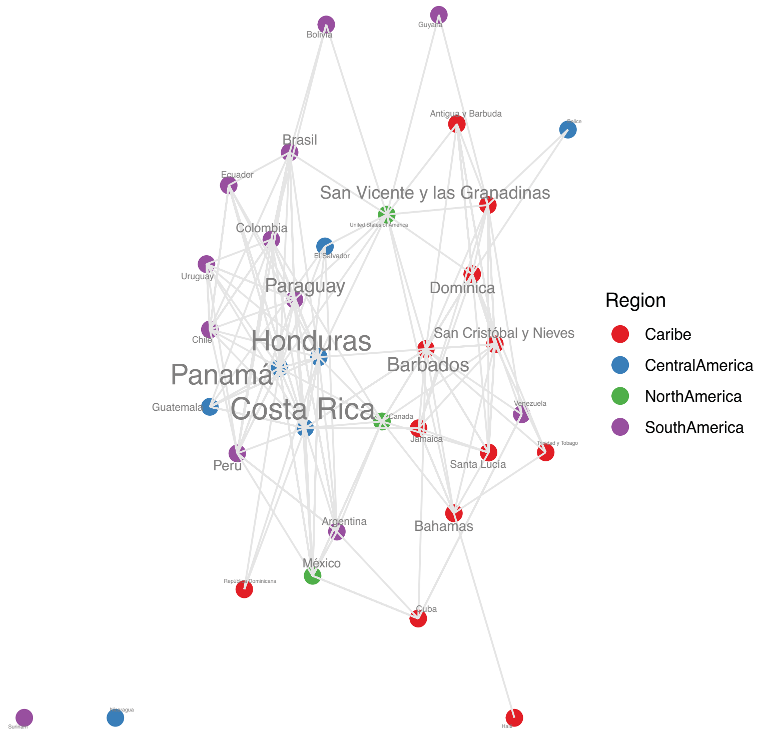

Figure 8.9 Network of Presidents of the Americas Sizes are based on out-degree centrality; nodes are colored by region in the Americas

The bigger the text, the more that president is a follower. The position of the nodes is based on *stress* algorithm (Gansner et al., 2005).

Similarly, an alternative **Python** version of Figure 8.9 follows:

```
from matplotlib.colors import rgb2hex
import matplotlib.pyplot as plt #513


plt.figure(figsize=(8,8))
#setting layout and palette
pos=nx.kamada_kawai_layout(net)
allValues=[n[1]['region'] for n in net.nodes(data=True)]
levels=pd.unique(allValues).tolist()
palette = plt.get_cmap("Set2")

#drawing nodes and edges
for categoryChosen in levels:
    nodesChosen=[node[0] for node in net.nodes(data=True)
                 if node[1]['region'] == categoryChosen]
    colorChosen=rgb2hex(palette(levels.index(categoryChosen)))
    nx.draw_networkx_nodes(net,pos=pos,node_size=100,
                           node_color=colorChosen,
                           nodelist=nodesChosen,
                           label=categoryChosen)
nx.draw_networkx_edges(net,pos=pos,edge_color="silver")
```

```
#drawing each node label - SIZE by outdegree!!
for user in net.nodes():
    sizeLabel=net.nodes(data=True)[user]['outdegree']+1
    nodeLabel=net.nodes(data=True)[user]['country']
    nx.draw_networkx_labels(net,pos=pos,
                            labels={user:nodeLabel},
                            font_size=sizeLabel**8)

plt.legend(markerscale=1, loc="best")
plt.show()
```

8.4.7 Highlighting Communities

The fact that actors belong to a particular neighbourhood does not mean that they are closely following or being followed by their neighbours. In networks, you speak of communities when you find that a set of actors is densely connected so as to differentiate itself from other set of actors, also densely connected. There are several algorithms for this, but their performance depends on how large the network is. If you are facing less than a thousand nodes, you may find that most algorithms can be chosen, but if you are speaking of thousands of nodes you need to carry out detailed research on this topic (see Fortunato (2010) and Yang et al. (2016)), and consider partnering with a network scientist.

Our data are very simple so it will be very easy to show you how this works. However, we have a directed network and most well-known algorithms can not work with directed edges. In some cases, the algorithm ignores the direction; in other cases it returns an error. Now, let me create an undirected version of my network:

```
> unet ← as.undirected(net, mode="mutual")
```

The object `unet` is an undirected network, but I have just kept the nodes that have a mutual relationship, using the `'mutual'` option in `mode`. At this point, I have several nodes that are totally disconnected from the rest. If I do not do anything else, each *isolate* node will be a community of its own. So, let me get rid of the isolates:

```
> unet=delete.vertices(graph=unet, #input
+                      v=which(degree(unet)==0))#what to remove
```

Let me do the same thing in **Python**:

```
# to indirected
unet=net.to_undirected(reciprocal=True)
# removing isolates
unet.remove_nodes_from(list(nx.isolates(unet)))
```

At this point, I am ready to find some communities in my network `unet`. Let me first use an algorithm based on *modularity maximization*; that is, modularity is an index that describes how well a set of nodes is connected, so those algorithms try optimize this index. Keep in mind that this family is not suitable for large networks (Clauset et al., 2004). In **R**, I can use the `cluster_fast_greedy` function to find the communities.

```
> modularityResult = cluster_fast_greedy(unet)
> # creating a new node attribute
> V(unet)$modCommunity = as.character(modularityResult$membership)
> # you get:
> V(unet)$modCommunity
```

```
 [1] "2" "1" "4" "3" "4" "3" "5" "6" "1" "3" "5" "1" "4" "4" "1" "2" "2"
     "6" "1"
[20] "4" "4" "2" "1" "2"
```

The new column `modCommunity` in the network `unet` tells you to what community a particular president belongs, so it is now very simple to prepare a plot:

```
> #layout
> layoutModu=ggraph(unet, layout="graphopt") + theme_void()
> #links
> moduLinks=layoutModu + geom_edge_link()
> #points
> moduNodes=moduLinks + geom_node_point(aes(colour=modCommunity),
+                                        size=5)
> #labels
> moduText=moduNodes+ geom_node_text(aes(label = name),
+                                   size=3,
+                                   color="black",
+                                   repel=T,
+                                   check_overlap = T)
> #final details
> moduVisual=moduText+scale_color_brewer(type = 'qual',
+                                   palette ="Set1")
> moduVisual=moduVisual + guides(color=FALSE)
```

The `moduVisual` object is drawn in Figure 8.10. This time, I have not allowed *ggraph* to choose the layout, I am explicitly requesting the use of the `graphopt` layout algorithm (Schmul, 2003).

Networkx has several algorithms to find communities, but I will use the external package *cdlib* (Rossetti, 2020) that works with *networkx* objects. Let me compute the communities following the same algorithm (modularity maximization):

```
# bring algorithm
from cdlib import algorithms

# Find the communities
modCommunity = algorithms.greedy_modularity(unet).communities
```

The result is saved in `modCommunity`, which I recovered after using `.communities`. If you check the contents of `modCommunity` you will see that the nodes have been organized into lists, like this:

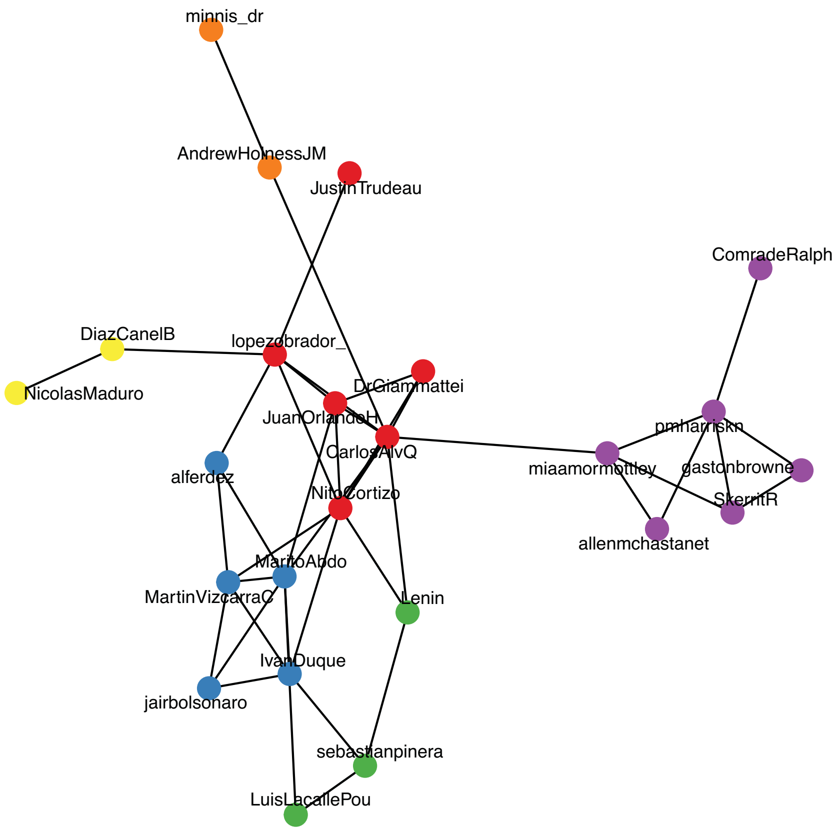

Figure 8.10 Network of Presidents of the Americas partitioned into communities
Color represents a particular community as detected following the Clauset et al. (2004) algorithm. The position of the nodes was computed using *graphopt* algorithm (Schmul, 2003).

```
[['ComradeRalph',
  'pmharriskn',
  'SkerritR',
  'miaamormottley',
  'gastonbrowne',
  'allenmchastanet'],
 ['JustinTrudeau',
  'CarlosAlvQ',
  'JuanOrlandoH',
  'lopezobrador_',
  'DrGiammattei',
  'NitoCortizo'],
 ['alferdez', 'MaritoAbdo', 'IvanDuque', 'MartinVizcarraC', 'jairbolsonaro'],
 ['LuisLacallePou', 'Lenin', 'sebastianpinera'],
 ['minnis_dr', 'AndrewHolnessJM'],
 ['DiazCanelB', 'NicolasMaduro']]
```

Once you know that the communities are identified, you can produce a result similar to Figure 8.10 like this:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import rgb2hex

palette = plt.get_cmap("Set1")

pos=nx.kamada_kawai_layout(unet)

# for each community
colorIndex=0
for community in modCommunity:
    # INSTEAD of rgb to hexadecimal: repeating list
    colorChosen=np.tile(palette(colorIndex), (len(community), 1))
    nx.draw_networkx_nodes(unet,pos,
                           nodelist=community, #nodes chosen
                           node_color=colorChosen)
    colorIndex+=1 #increase index

#edges and labels (default values)
nx.draw_networkx_edges(unet, pos)
nx.draw_networkx_labels(unet,pos)
plt.show()
```

Notice that I decided to show you a different approach to producing the colors. If you just use `palette(colorIndex)`, you will get a warning. The warning also recommends you to input a 2-D matrix. That is what the `np.tile()` function does. It takes the color in RGB and repeats it in a matrix with as many rows as there are nodes in the community. I will not use this again.

As mentioned before, not every algorithm is suitable for networks of high complexity. One very well known for large networks is the Louvain algorithm (Blondel et al., 2008). Let me show you how easy it is to use it in **R**:

```
> louvainResult = cluster_louvain(unet)
> # creating a new node attribute
> V(unet)$louvainCommunity = as.character(louvainResult$membership)
> # as before:
> layoutLouv=ggraph(unet, layout="graphopt") + theme_void()
> louvLinks=layoutLouv + geom_edge_link()
> louvNodes=louvLinks + geom_node_point(aes(colour=louvainCommunity),
+                                  size=5)
> louvLabels=louvNodes+ geom_node_text(aes(label = name),
+                                 size=3,
+                                 color="black",
+                                 repel=T,
+                                 check_overlap = T)
> louvVisual=louvLabels+scale_color_brewer(type = 'qual',
+                                    palette ="Set1")
> louvVisual=louvVisual + guides(color=FALSE)
>
```

The `louvVisual` object is drawn in Figure 8.11.

The **Python** alternative for detecting a community using the Louvain method follows:

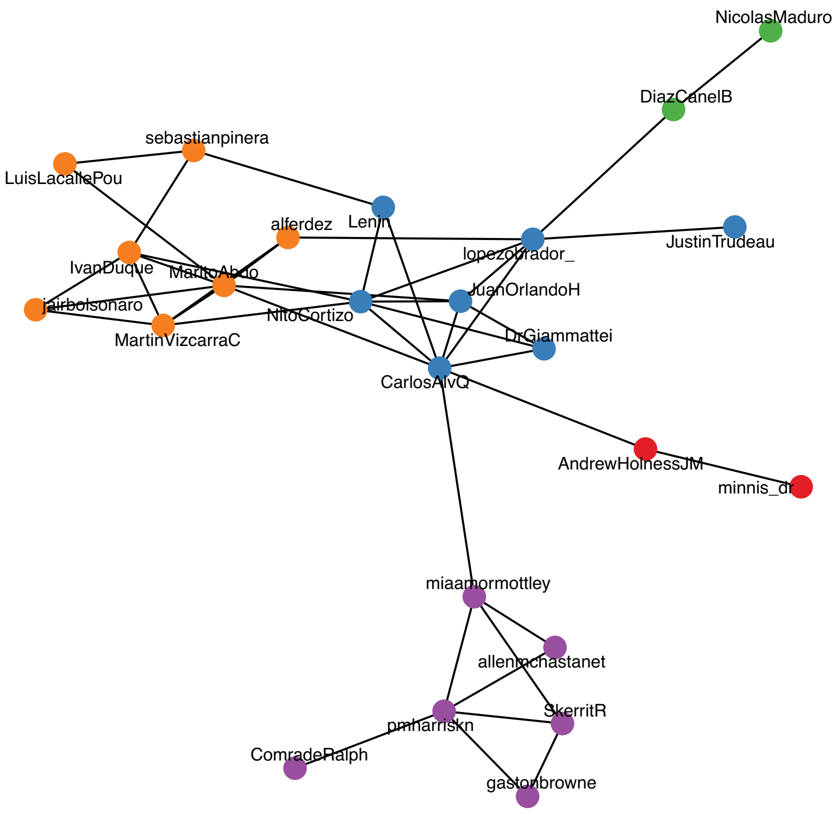

Figure 8.11 Network of Presidents of the Americas partitioned into communities

Color represents a particular community as detected following the Louvain algorithm (Blondel et al., 2008) The position of the nodes was computed using *graphopt* algorithm (Schmul, 2003).

```
louvainCommunity = algorithms.louvain(unet).communities

import matplotlib.pyplot as plt
from matplotlib.colors import rgb2hex

palette = plt.get_cmap("Set1")
pos=nx.kamada_kawai_layout(unet)

# for each community
colorIndex=0
for community in louvainCommunity:
    # from rgb to hexadecimal
    chosenColor=rgb2hex(palette(colorIndex))
    nx.draw_networkx_nodes(unet,pos,
                           nodelist=community, #nodes chosen
                           node_color=chosenColor)
    colorIndex+=1 #increase index

#edges and labels (default values)
nx.draw_networkx_edges(unet, pos)
nx.draw_networkx_labels(unet,pos)
plt.show()
```

Finally, let me show you an algorithm that works with *directed networks*. The algorithm is named *infomap* (Rosvall and Bergstrom, 2008) and it is also available in **R** and **Python**. Let me use my original network and show you the result in **R**:

```
> infomapResult = cluster_infomap(net)
> # creating a new node attribute
> V(net)$infmpCommunity = as.character(infomapResult$membership)
> layoutInfmp=ggraph(net, layout="graphopt") + theme_void()
> infmpLinks=layoutInfmp + geom_edge_link()
> infmpNodes=infmpLinks + geom_node_point(aes(colour=infmpCommunity),
+                                          size=5)
> infmpLabels=infmpNodes+ geom_node_text(aes(label = name),
+                                         size=3,
+                                         color="black",
+                                         repel=T,
+                                         check_overlap = T)
> infmpVisual=infmpLabels+scale_color_brewer(type = 'qual',
+                                             palette ="Set1")
> infmpVisual=infmpVisual + guides(color=FALSE)
>
```

The `infmpVisual` object is drawn in Figure 8.12.

The alternative in **Python** follows:

```
infmpCommunity = algorithms.infomap(net).communities

import matplotlib.pyplot as plt
from matplotlib.colors import rgb2hex

palette = plt.get_cmap("Set1")
pos=nx.kamada_kawai_layout(net)

# for each group
colorIndex=0
for community in infmpCommunity:
    # from rgb to hexadecimal
    chosenColor=rgb2hex(palette(colorIndex))
    nx.draw_networkx_nodes(net,pos,
                           nodelist=community, #nodes chosen
                           node_color=chosenColor)
    colorIndex+=1 #increase index

#edges and labels (default values)
nx.draw_networkx_edges(net, pos)
nx.draw_networkx_labels(net,pos)
plt.show()
```

There is so much more research on network science, but I think with these visual aids you can make a case whenever you have data suitable for this kind of work.

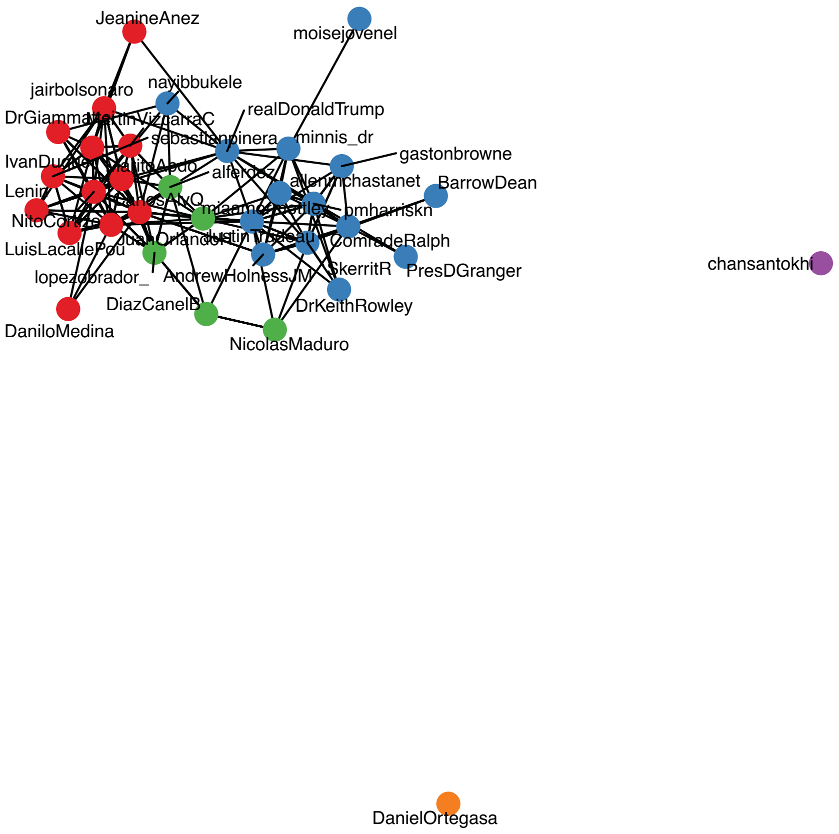

Figure 8.12 Network of Presidents of the Americas partitioned into communities
Color represents a particular community as detected following the Infomap algorithm (Rosvall and Bergstrom, 2008) The position of the nodes was computed using *graphopt* algorithm (Schmul, 2003).