Readme for future data collectors

EUSpeech Project

Nicolai Berk*

June 2020

*You can contact me via nicolai.berk@gmail.com.

Hello!

You have been provided with the important task to update our EUSpeech dataset. This brief guide will explain to you how to use the existing python code to collect links and speeches. The Readme will show code like this:

```
print("Hello world!")
```

And output like this:

```
Hello world!
```

The guide is divided into three main sections: the link collection, the speech collection, and the further processing of the data. Each section will present the functions that were designed for this specific task and explain how to use them. **The explanations assume that you already installed python on your computer and have a basic understanding of it.** All code can be found in this Dropbox folder: https://github.com/nicolaiberk/EUSpeech/tree/master/EUSpeech Future Collections.

The execution of the functions described in this guide presuppose the installation of the following packages for python:

- lxml
- requests
- time
- csv
- re
- sys
- selenium
- langdetect

Contents

1 Link Collection

The collection of speeches is divided into two parts: the collection of all links to these speeches and the collection of the speeches themselves through these links.

1.1 Website structures and xpaths

In order to collect the necessary links, we need to specify a website where the links can be collected from. For the purpose of this example, we will use the Dutch government's website to collect Mark Rutte's speeches in the English language, see here: https://www.government.nl/government/members-of-cabinet/mark-rutte/documents?type=Speech&page=1. When you open the page, you will see that it contains a number of documents about and from Dutch Prime Minister Mark Rutte and has been filtered to contain only speeches. If you inspect the link, you can see that the page number is specified there ('page=1') - this will be useful for the collection later. Right-click on one of the item headers and click 'inspect element' in the menu popping up. As a result, a menu similar to that on the right-hand side of figure 1 should unfold (it might pop up at the bottom in your case).

This menu might have several tabs. If not already selected, click on the 'Inspector' tab. The tab shows you the html structure of the document, with elements (or nodes) nested in other elements (or parent nodes) on the website. Return to the item header on the website, right-click > 'inspect element' (again). The side bar should now highlight the corresponding html code for this element in the side bar. In the example, this is an 'h3' element, a type of header. Once you click the little arrow next to it, it will show the text. This header is nested in the 'a' element/node, which also contains a link and a date. We want to collect these elements in order to collect all speeches subsequently.

To identify these elements, we will need what are called 'xpaths'. These paths will tell our scraper function where to find the elements we need. An xpath uses the nested structure of nodes/elements to select one or several elements. There are two main ways to get the right xpaths to select our elements:

- **The easy way**: Right click on the highlighted code in the side bar (not the text it contains), select 'Copy' > 'XPath'. Paste the copied xpath into a text editor. You should get something like this: '/html/body/div[1]/main/div/div[1]/ div[4]/a[1]/h3'. You can see that the header ('h3') is nested in the first 'a' element, which is nested in the fourth 'div' element, and so on.

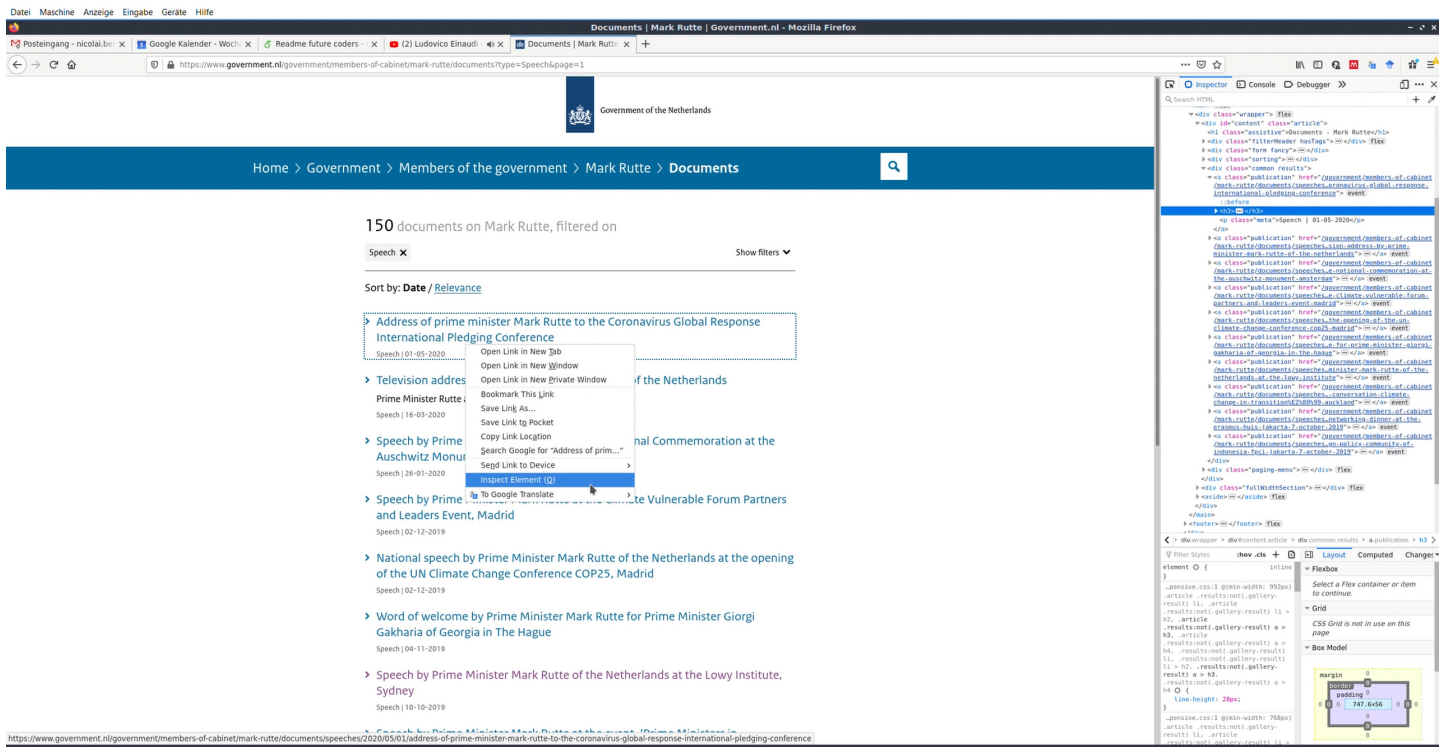

Figure 1: The structure of a website.

If you right-click the header of the second element and copy its xpath, you will see that it is largely identical, but this time the header is nested in the second 'a' element, 'a[2]'. Pay attention to the changing specifications across these elements. In xpaths, you can specify that all elements of a certain type should be collected using the '*' operator. For example, '/html/body/div[1]/ main/div/div[1]/div[4]/a[*]/h3' will match h3 elements that are nested in any a element at the location '/html/body/div[1]/main/div/div[1]'.

- **The better way**: Although the easy way might often be faster, it might also match elements which we are not interested in, or miss elements that reside at a different location. To circumvent this, one can use classes and ids. In our example page, the html code of the 'a' element where our header is nested specifies that it belongs to the class 'publication'. XPaths can be specified to match only elements of a certain class. For example, '/html/body/div[1]/main/div/div[1]/ div[4]/a[@class="publication"]/h3' matches all 'h3' elements within 'a' elements with the class 'publication' at the location '/html/body/-div[1]/main/div/div[1]/ div[4]'.

  However, this does not solve the issue that we might want to match elements of the same class at other locations. This can be solved with the '//' and '*' selectors. '//' selects nodes in the document that match the selection no matter where they are, while '*' is a so-called wildcard that matches any element[1]. The xpath to select any h3 element within an element of the class 'publication' on the website is thus '//*[@class="publication"]/h3'.

Which way to choose will largely depend on the structure of the website, but I advise to use the latter where possible. This approach is more structured and allows subsequent people reading your code to understand what you are trying to do more easily. To check whether you match the correct element(s), I advise you to use the following code:

```
1 import requests
2 from xml import html
3
4 fetchlink = 'https://www.government.nl/government/members-of-cabinet/mark-
      rutte/documents?type=Speech&page=1'
5 xpath = '//*[@class="publication"]/h3'
6
7 req = requests.get(fetchlink)
8 tree = html.fromstring(req.text)
```

[1] see https://www.w3schools.com/xml/xpath_syntax.asp for more information

```

len(tree.xpath(xpath)) # check the number of matched elements
tree.xpath(xpath)[0].text # check the text in the first element
```

```
10
'\n                Address of prime minister Mark Rutte to the Coronavirus
    Global Response International Pledging Conference'
```

This method allows you to simply check whether you matched the expected number of items and whether the matched items look like you would expect them to. Don't worry about leading/tailing strings, linebreaks ('\n'), or tabs and other alignment characters ('\t', '\r'), as the function will get rid of them for you.

1.2 The linkScraper function

1.2.1 Collecting from a single website

Time to collect the actual links. In the Github folder, you can find a file called 'functions.py'. This file contains several functions (surprise!). To collect links from simple websites as the Dutch government's, we can use the function 'linkScraper'. The function will take a number of arguments, such as a url and several xpaths, and collect the required data. Consider the example code (from the file '01_single_example.py') below. The code imports the linkScraper function from the 'functions' module [line 1] and defines a path where to dump the csv of collected links ('linkdir') [3:4]. Then, the function is called with several arguments: a filename is defined, the directory is specified, a 'sender' is specified. The url from which to scrape is specified, as well as the xpaths to find links, titles, and dates. Note that you only have to specify the elements containing the texts or dates (not the dates or texts themselves). The function will assume that the links are specified in the 'href' argument of the element, while titles and dates are provided as texts. Additionally, a regular expression is supplied to extract the date from the scraped text string[2]. Lastly, the date format is provided. Often, website links miss the first part (see the 'href' argument in the element [a] on the government's website). The 'linkbase' argument supplies a string to which to append the collected link. A full overvew of the arguments can be found in table 1.

```
from Collectors import linkScraper

import os
linkdir=os.getcwd()+'/CompleteLinks/'

```

[2]Find out how to specify regular expressions here: https://regexr.com/.

```
#%% NL (english)
linkScraper(file      = 'example_links',
            path      = linkdir,
            sender    = "M. Rutte",
            url       = "https://www.government.nl/government/members-of-
                cabinet/mark-rutte/documents?type=Speech&page=1",
            linkbase  = "https://www.government.nl",
            xpathLinks = '//*[@class="common results"]/a',
            xpathTitles = '//*[@class="publication"]/h3',
            xpathDates = '//*[@class="meta"]',
            regexDates = ' *Speech \| (.*)',
            strToDates = "%d-%m-%Y")
```

When you run this code in a python console, you should get the following output:

```
Fetching links M. Rutte...

Finished collecting 10 links for M. Rutte
```

The function successfully collected 10 links! Go to 'CompleteLinks/example_links.csv' to assess the written output. As you can see, the function wrote a csv with four columns, namely the date, the speaker, the title, and the link where to find the speech.

1.2.2 Collecting from multiple websites

However, collecting website by website might be very tedious if you want to collect hundreds or even thousands of links. In order to simplify this, the function also takes a different input for the `url`-argument. Open the file '02_multi_example.py' in a text editor:

```
from Collectors import linkScraper

import os
linkdir=os.getcwd()+'/CompleteLinks/'

#%% NL (english)
linkScraper(file      = 'example_links',
            path      = linkdir,
            sender    = "M. Rutte",
            url       = "https://www.government.nl/government/members-of-
                cabinet/mark-rutte/documents?type=Speech&page={}",
            linkbase  = "https://www.government.nl",
            xpathLinks = '//*[@class="common results"]/a',
            xpathTitles = '//*[@class="publication"]/h3',
            xpathDates = '//*[@class="meta"]',
            regexDates = ' *Speech \| (.*)',
            strToDates = "%d-%m-%Y",
```

```
            maxpage    = 15,
            mindate    = "03/06/2019",
            country    = "The Netherlands")
```

You can see that the input is largely identical, but there are some differences. The url has changed slightly - instead of `page=1`, it now includes curled brackets. This allows the function to format the url, in order to fetch links from every formatted page up to the `maxpage`. Hence, this function collects all elements from page 1, 2, ... up to page 15.[3] Additionally, the arguments `mindate` and `country` are specified. The first defines a minimum date where to stop scraping, the latter adds a column containing the specified string to the csv output. The output in the console should look like the following:

```
Fetching links M. Rutte...
              [====                      ]  2/15
       Reached 03/06/2019 (min), stopping process...

Finished collecting 14 links for M. Rutte
```

The first part is a progress bar, indicating the page currently scraped, up to the `maxpage`. It then stops, having reached the `mindate` and shows that it collected 14 links[4]. Check the csv output in the 'CompleteLinks' folder (note that your old file has been overwritten, you can control this behaviour via the `mode` argument). You can see that an additional column displays the `country` argument now.

[3]The skips can be specified with the `npage` argument. For further details, consider table 1

[4]Note that the precise number can differ as the website changes

1.2.3 Arguments

argument	format	description
`file`	string	name of output file
`path`	string	output path
`sender`	string	speaker name
`url`	string	url that links should be collected from, input for format()
`linkbase`	string	linkbase that directed links should be appended to
`xpathLinks`	string	xpath to elements containing 'href' argument to scrape
`xpathTitles`	string	xpath to elements containing title text
`xpathDates`	string	xpath to elements containing date text
`strToDates`	string, list	pattern for strptime (will try all of them if list)
`regexDates`	string	regular expression to find date (matches first group)
`maxpage`	integer	max number of pages that the scraper should
`mindate`	string	earliest date to be scraped, formatted in %d/%m/%Y
`maxdate`	string	earliest date to be scraped, formatted in %d/%m/%Y
`language`	string	language of date format, supports german and italian.
`npage`	integer	skip in between pages, default = 1
`start`	integer	first page number, if undefined start = 1
`country`	string	add a column with country name, left out if not provided
`mode`	string	'w' for overwriting existing csvs, 'a' for adding to them

Table 1: Arguments to the linkScraper function. All arguments below the line are optional.

1.3 The seleniumScraper function

1.3.1 Function

Some websites might not conform to the structure either of having all links ready to scrape on one page or a formattable url. Instead they might load additional elements as you scroll or make you click a button to load additional information. Go on https://www.elysee.fr/toutes-les-actualites?categories[ ]=discours to see an example. The page contains thirty elements, but there are no buttons to the next page. Instead, at the bottom of the page, a button says 'Afficher plus d'articles' ('show more articles'). Once you click it, more elements will be displayed, but the url remains unaffected. Pages like this can be scraped using selenium.

The 'selenium'-package allows you to scroll down or click on a website automatically to collect elements using formatted xpaths instead of urls. The function used here makes certain assumptions that might not correspond to every webpage, and sometimes you might not be able to use the function. However, for most cases, the function should make your life easier by guiding your efforts. For all other applications, I recommend you make yourself familiar with selenium to write your own code[5].

Before you can use selenium (and hence the function), make sure you have installed the Firefox webbrowser, as well as the geckodriver, which should also be in your PATH[6]. You can find another example script in the Dropbox folder, titled '03_selenium_example.py'. Open the file in a text editor to see the code below:

```
from Collectors import seleniumScraper

seleniumScraper(file = 'example_links',
    path = 'CompleteLinks/',
    sender = 'E. Macron',
    url = 'https://www.elysee.fr/toutes-les-actualites?categories[]=
        discours',
    xpathLinks = '/html/body/main/article/section[{}]/div/div/a',
    xpathTitles = '/html/body/main/article/section[{}]/div/div/a/h2',
    xpathDates = '/html/body/main/article/section[{}]/div/div/p',
    strToDates = '%d %B %Y',
    regexDates='(.*)',
    mindate='01/03/2020',
    language="french",
    country='France',
    xpbutton = '//*[@id="main"]/section[2]/p/button',
    xpcookie = '//*[@id="tarteaucitronPersonalize"]',
```

[5] See https://selenium-python.readthedocs.io/

[6] See https://selenium-python.readthedocs.io/installation.html for instructions

```
    process = 'button')
```

As you can see, the arguments of this function are rather similar to the ones in the `linkScraper` function, however the xpaths for links, titles, and dates include curled brackets, similar to the url in the second example. There are three additional arguments defined: `xpbutton`, `xpcookie`, and `process`. The full list of arguments can be found in table 2. These additional arguments show the function where to accept cookies when opening the site (`xpcookie`), whether to scroll or click for more elements (`process`), and where to find the button to load more elements in case the `process`-argument equals 'button'. The function will open a browser window and search for elements using the provided xpaths. The function formats the xpaths, getting the title, link, and date for the first element in the first iteration, the second in the second, and so on. If an element cannot be found, the function will look once more before clicking/scrolling to load additional elements. If that doesn't do the trick, it will skip an element. The program will only stop if the `mindate` is reached, or if the function cannot find any new elements five times in a row.

In our specifications for the function, we told it to load more elements by clicking on a button in the `process` parameter. If you run the function in a python terminal (and you have installed all necessary packages and placed the geckodriver in your path), a browser window should open up. Don't interact with the window, as the function is controlling the browser now. You will see that the website will present a banner asking for consent to use cookies, which will quickly disappear as our function clicks to consent. Then, the website will scroll downwards as the function clicks to load more elements. The output in the console should look similar to the output shown below.

```
Collecting links.....

              Collecting element #21 using button process...
              Element 21 not found, trying again...
              Collecting element #21 using button process...
              Element 21 not found, trying again...
              Collecting element #21 using button process...
              Element 21 not found, trying again...
              Collecting element #41 using button process...
              Element 41 not found, trying again...
              Collecting element #41 using button process...
              [...]
              Collecting element #109 using button process...
       Reached 01/03/2020, (min) stopping process after collection of 107
           elements.
```

The function could not find element 21, which it tried three times, then it

clicked the button to load more elements (not visible in the console), which apparently resulted in the display of 20 additional items before running into the same issue again with element 41. After a few seconds, the function had reached the defined minimum date and stopped the collection. Similar to the `linkScraper` function, the output is written to a csv 'example_links', as defined in the `file`-argument.

1.3.2 Arguments

argument	format	description
`file`	string	name of output file
`path`	string	output path
`sender`	string	speaker name
`url`	string	url that links should be collected from, input for format()
`linkbase`	string	linkbase that directed links should be appended to
`xpathLinks`	string	formattable xpath to elements containing 'href' argument to scrape
`xpathTitles`	string	formattable xpath to elements containing title text
`xpathDates`	string	formattable xpath to elements containing date text
`strToDates`	string, list	pattern for strptime (will try all of them if list)
`xpbutton`	string	xpath to button loading additional elements, in case process='button'
`xpcookie`	string	xpath to button to accept cookies, if website asks
`regexDates`	string	regular expression to find date (matches first group)
`maxpage`	integer	max number of pages that the scraper should
`mindate`	string	earliest date to be scraped, formatted in %d/%m/%Y
`maxdate`	string	earliest date to be scraped, formatted in %d/%m/%Y
`language`	string	language of date format, supports german and italian.
`country`	string	add a column with country name, left out if not provided
`mode`	string	'w' for overwriting existing csvs, 'a' for adding to them
`process`	string	'scrolling' to scroll pages, 'button' for clicking to load additional elements

Table 2: Arguments to the seleniumScraper function. All arguments below the line are optional.

2 Speech Collection

Now, we can finally go ahead and collect the speeches from the links we collected earlier. We will only collect the speeches from the french links collected earlier to give you an example. To do this, we will use the `speechScraper`-function.

2.1 The speechScraper function

2.1.1 Function

Open the file '04_speechCollector_example.py' in a text editor (or see below). You can see that the function takes different arguments than the ones before. Most importantly, we have to define a filename `inputfile` and directory `linkdir` from where to collect the links to the websites containing the speeches. Next, the function is told where to write the speeches to in the `speechdir`-parameter. Several xpaths to match are provided in the `xpath`-argument, divided by '|'. A special argument 'test_re' is provided to match the title to regular expressions. Only if a match occurs, the site is actually scraped. As the site we collected the links from contains other events, too (like press releases), we can use this argument to select only those that contain a speech. An additional `regex`-parameter can be defined to match only certain parts of the text (matches the first group).

```
from Collectors import speechScraper
from datetime import date
import os
linkdir = os.getcwd()+'/CompleteLinks/'

today = date.today()

speechScraper(inputfile = 'example_links',
              linkdir = linkdir,
              speechdir = os.getcwd()+'/speeches/',
              xpath = '//div[@class="size3-2 reset-last-space ck-styled
                  "]//* | //div[@class ="card"]//*',
              test_re = '.*[Dd]eclaration.*|.*[Aa]dresse.*|.*[Mm]essage')
```

Once you run the function, you should see something like the output below. The function checks every title, and if it does not match the regular expressions mentioned in `test_re`, the row (and hence the corresponding link) is skipped (according to our output, this happened 83 times - we might go back and check if we excluded too many). Once it finds a match, the link is used to fetch the speech. Some sites might only contain videos and a

brief description. As the function skips very short speeches[7], these speeches are not written to the csv ("collection of 6 links failed"). Finally, if the title matches the provided regular expression, and the speech is long enough, it writes the date, speaker, title, url and the speechtext to a csv at the provided destination.

```
Start fetching speeches...
   [...]
   Fetching speech # 107 ... ( example_links)
Not a speech, skipping

Finished fetching 23 speeches,
collection of 6 links failed,
skipped 83 links.
```

If you open the csv-file that should now have emerged in the speeches folder (e.g. in a text editor), you will see that another column was added to the inputfile, containing the speechtext (some of these characters might look weird as the program does not recognise the UTF-8 encoding; don't worry about it for now). Look at the collected speeches. Did they collect the text you would have wanted based on the link? If not, adapt the xpath(s) or consider using the `regex` parameter. You might have to play around with these elements a bit before getting desirable results, as the websites' structure might change.

2.1.2 Arguments

argument	format	description
`inputfile`	string	name of input file
`linkdir`	string	input path
`speechdir`	string	output path
`xpath`	string	xpath to element(s) containing text to scrape
`regex`	string	regular expression applied to scraped text, matching first group
`test_re`	string	regular expression applied to title. If no match, link isn't scraped
`mode`	string	'w' for overwriting existing csvs, 'a' for adding to them
`min_len`	integer	min number of characters a speech must contain to be written to output
`timestamp`	bool	should the output csv have a timestamp?

Table 3: Arguments to the speechScraper function. All arguments below the line are optional.

[7] You can control this behaviour via the `min_len` parameter

3 Further processing

3.1 The language recognition function

Often, speeches might be held in languages other than the domestic one, or even several languages. This will affect the analysis of a given text. Luckily, there is a python package that can do the checking for us. Open the file '05_langdet_example.py' and look at the code below.

```
from functions import langdetectspeeches
import os
basedir = os.getcwd()

inputcsv=basedir+"/speeches/speeches"
outputcsv=basedir+"/speeches/speeches_LangDet"

langdetectspeeches(inputcsv,
                   outputcsv)
```

This simple function will take as input the table of speeches that has been collected and check for each speech which languages are present. It will then append several columns to that table, containing the most likely language(s) of the speech, the likelihood of that estimate, the number of languages, and the length of the speech in words as well as characters. If you run the example (after running all of the code explained above), the consoles should show the output below, and the speeches folder should contain a new csv-file called 'speeches_LangDet'. This is the final form of the speech data to be transformed to an .RDta-file.

```
Detecting Languages:
Done   [====================] | 19 of 19
```

3.1.1 Arguments

argument	format	description
`inputcsv`	string	name of input file without file ending
`outputcsv`	string	name of input file without file ending
`readHeader`	bool	input file with header? default=False
`mode`	string	'w' for overwriting existing csvs, 'a' for adding to them
`timestamp`	bool	should the output csv have a timestamp?

Table 4: Arguments to the speechScraper function. All arguments below the line are optional.

3.2 A note on character encodings

Computers store data in a binary format. Every letter must thus be encoded into a collection of 0's and 1's. Sadly, there is no common standard for the translation of characters (eg 'a') into binary code (eg '01100001'). As 95% of websites are encoded in UTF-8 [8], a unicode encoding which is able to display about just every character used in *any* language, the functions collect the text data in UTF-8 encoding. This is great, especially as it is highly compatible with UNIX-like operating systems such as Linux or MacOS. However, especially windows users might have a hard time with UTF-8 encodings, as the platform and many programs running on it offer only limited or no compatibility. Hence, when you import the data into RStudio or Excel, you might be shocked as the data looks like absolute gibberish. Don't panick, as the problem is likely the default character encoding of your program, not the data itself. Make sure to import the data as UTF-8 encoded, eg by using `readtext(file text_field = x, encoding = "UTF-8")` or using `Data >> From Text/CSV >> Import` in Excel and selecting '65001: Unicode (UTF-8)' in the import wizard[9].

4 Conclusion

You should now have a basic understanding of the functions and should be able to get started with the collection. I recommend to collect the data country by country and then bring it together in the end, to make it easier to find errors and write simpler code. Waste one or two thoughts on the appropriate folder structure *before* starting the collection. Then find a webpage you want to collect data from, define the relevant parameters of the function and collect the data you want. It will usually take a bit until the function is properly defined and then it might take some additional tweaking until you get the output you want - this is the usual process, so be patient. Hopefully the functions will be able to guide your efforts. They are most definitely not able to cover every application and you might have to look into the code yourself at some point. Don't panic then, Stackoverflow and Google are your friends. Good luck.

[8] https://w3techs.com/technologies/cross/character_encoding/ranking

[9] Use of excel is discouraged.