# ACIT4420 final project: Website-Crawler

**By candidate: 103**

## Table of contents:

## 1. Introduction:

According to Wikipedia, web scraping, web harvesting, or web data extraction is data scraping used for extracting data from websites. The web scraping software may directly access the World Wide Web using the Hypertext Transfer Protocol or a web browser. While web scraping can be done manually by a software user, the term typically refers to automated processes implemented using a bot or web crawler. [1] There are many sources for web-scraping like for example social media as Twitter, Facebook, and WhatsApp, searching engine like Google, research work, wireless connected devices, and websites. In this project we are going to work on building a "web crawler" for mining data from websites because data mining and analyzing, especially from websites, is getting more and more important for in different sectors. Now we have tons of data on web which are left without analyzing and taking insights from.

## 2. Solution Design:

Generally speaking, we tried to commit to object oriented programming principles, especially compositionality. What It means by compositionality here is initializing an object one or more other classes, reusing some methods of a class inside others. In other words, our methods build on each other. Therefore, the order of calling the methods in the most cases is not so important, otherwise you will get an alarm message about which method you need to run before the current method. In addition, in the software solutions we design, we usually follow the confidentiality and encapsulation principles. We mean by them is preventing unauthorized access for data, instance variables, and primary methods. But we did not do it in this project to make it easy for you to test the different parts of the solution. We also combined the getters with the setters because for the same reason, readability. Our solution is like one main huge class called "web_Crawler", and one supportive calles "statistical_model". This class receives the parameters which are a website address, depth, and a wild regular expression. The type of the first parameter is String, the second is integer and the last is a String. In additional to the parameters, we have five instance variables. The first three of them are made to save the value of the arguments. The fourth one is to pull out the source code of the given website address, and the fifth one is a dictionary which saves the most frequent words with their frequency. This class has 13 different methods. As we mentioned earlier, they build on each other where the earlier ones are the most primitive and important. In addition to these 15 methods, we have main, an in-built function. We use it for testing. Regarding the supportive class called "statistical_model". We build it from scratch to do statistics on the words and their classes or pos-tags. It provides some basic functions of statistics. It will substitute the use of external libraries like stats. The following functions are among the ones we made median, standard deviation, quartiles, z scores, and coefficient of variance.  We found useful to make a function which detect outliers as well.

## 3. Models' description:

### 3.1. Beautiful Soup

It is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work [2]. We will pay your attention to that we are going to use version nr. 4, not 3.

### 3.2. Requests

Requests is a HTTP library for the Python programming language. The goal of the project is to make HTTP requests simpler and more human-friendly. The current version is 2.26.0. this library is released under the Apache License 2.0. It is one of the most popular Python libraries that is not included with Python. [3]

### 3.3. NLTK model

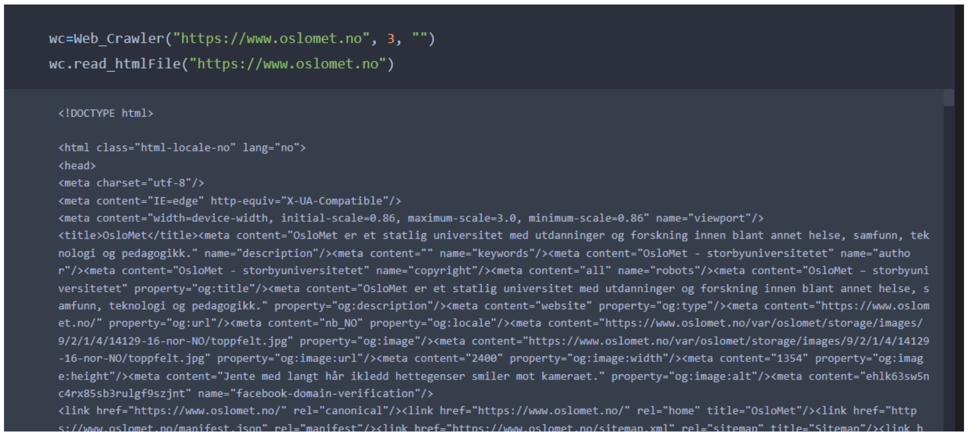
We will mainly use a model called Natural Language Toolkit. NLTK is a pre-trained statistical NLP model. [4] It has many useful corpuses and tools which can deal with natural languages. Some of the common tools the model offers are tagging, lemmatizing, and stemming. Pos-tagging is the most used one. According to Wikipedia, POS is a category of words or lexical items that have similar grammatical properties like nouns, verbs and so on. We will come back to it later in the implementation section. Note that the attached code is totally written by me, except one simple method borrowed from Stack Overflow-website [5].

## 4. Implementation and analysis

In this section, we are going to show the Implementation in detail. It is also natural to analyze and test our model in this section together with explaining the implementation.

### 4.1. Reading html-files:

We made a method which can deal with html-files. read_htmlFile() is the given name to it. It starts by initializing a HTTP-request and calling is method get () which receives a URL as a parameter. Then an object of BeautifulSoup gets initiated. It receives the answer from the HTTP request and parse it. Then return the html-file of the given URL. It is important to mention that some webpages-owners put restrictions on pulling their html-files. Anyway, there are some regulations which need to be take in consideration when crawling webpages. [6] The figure below shows a test of the method.



```
wc=Web_Crawler("https://www.oslomet.no", 3, "")
wc.read_htmlFile("https://www.oslomet.no")
```

```
<!DOCTYPE html>

<html class="html-locale-no" lang="no">
<head>
<meta charset="utf-8"/>
<meta content="IE=edge" http-equiv="X-UA-Compatible"/>
<meta content="width=device-width, initial-scale=0.86, maximum-scale=3.0, minimum-scale=0.86" name="viewport"/>
<title>OsloMet</title><meta content="OsloMet er et statlig universitet med utdanninger og forskning innen blant annet helse, samfunn, tek
nologi og pedagogikk." name="description"/><meta content="" name="keywords"/><meta content="OsloMet - storbyuniversitetet" name="autho
r"/><meta content="OsloMet - storbyuniversitetet" name="copyright"/><meta content="all" name="robots"/><meta content="OsloMet – storbyuni
versitetet" property="og:title"/><meta content="OsloMet er et statlig universitet med utdanninger og forskning innen blant annet helse, s
amfunn, teknologi og pedagogikk." property="og:description"/><meta content="website" property="og:type"/><meta content="https://www.oslom
et.no/" property="og:url"/><meta content="nb_NO" property="og:locale"/><meta content="https://www.oslomet.no/var/oslomet/storage/images/
9/2/1/4/14129-16-nor-NO/toppfelt.jpg" property="og:image"/><meta content="https://www.oslomet.no/var/oslomet/storage/images/9/2/1/4/14129
-16-nor-NO/toppfelt.jpg" property="og:image:url"/><meta content="2400" property="og:image:width"/><meta content="1354" property="og:imag
e:height"/><meta content="Jente med langt hår ikledd hettegenser smiler mot kameraet." property="og:image:alt"/><meta content="ehlk63sw5n
c4rx85sb3rulgf9szjnt" name="facebook-domain-verification"/>
<link href="https://www.oslomet.no/" rel="canonical"/><link href="https://www.oslomet.no/" rel="home" title="OsloMet"/><link href="http
s://www.oslomet.no/manifest.json" rel="manifest"/><link href="https://www.oslomet.no/sitemap.xml" rel="sitemap" title="Sitemap"/><link h
```

***Figure [1]***

## **4.2.** Links of subpages:

The method *extract_links()* captures all links which most probably point out to subpages of the main webpage, the given URL. Actually, there is no way to sperate between links, '*a*' tags, which belongs to subpages and which belong to external web-pages. But we found a common property about subpages. They are usually links inside html-lists, *'ul'* tags. These lists again consist of elements, '*li*' tags, which can work as containers for links of subpages, '*a*' tags.

An example:

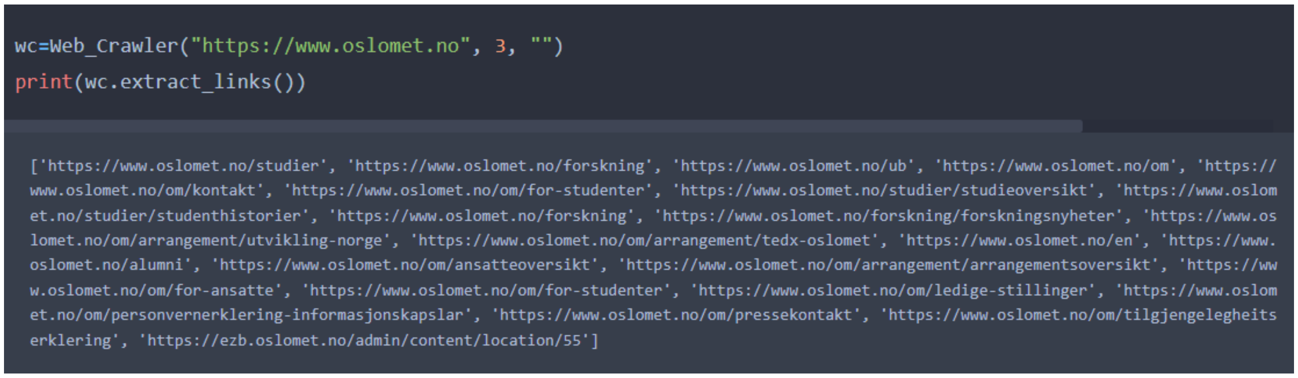
```
<ul>

<li>  <a href="https://xx.xx.xx/xx/">products</a>   </li>

<li>  <a href="https://xx.xx.xx/xx/">about us</a>   </li>

<li>  <a href="https://xx.xx.xx/xx/">contact us</a>   </li>

</ul>
```

Moving to talk about implementation, the method starts by pulling the source code and extracting all li-tags. Then again It extracts the 'a' tags which are inside these li-tags, and have href-attribute. In other words, we are not interested in li-tags which do not have a-tags, neither in a-tags which do not have href-attribute, inactive links. After that it pulls out the links as strings. One problem we met is that in some webpages, like Oslomet.no, the links of the subpages are not are not given as a full link address. Here it is an example about this case:

-The link address of the main webpage: https://www.oslomet.no/

-The link address of a subpage: <a href="/studier ">Study at OsloMet</a>

Therefore we need to concatenate the link of the main page with the not-fully-written links of subpages for further use, the depth. The way to capture the links which do not have a full adress is by checking if a link does not start with 'https', or if it starts by a back slash. The figure below shows a test of the method.



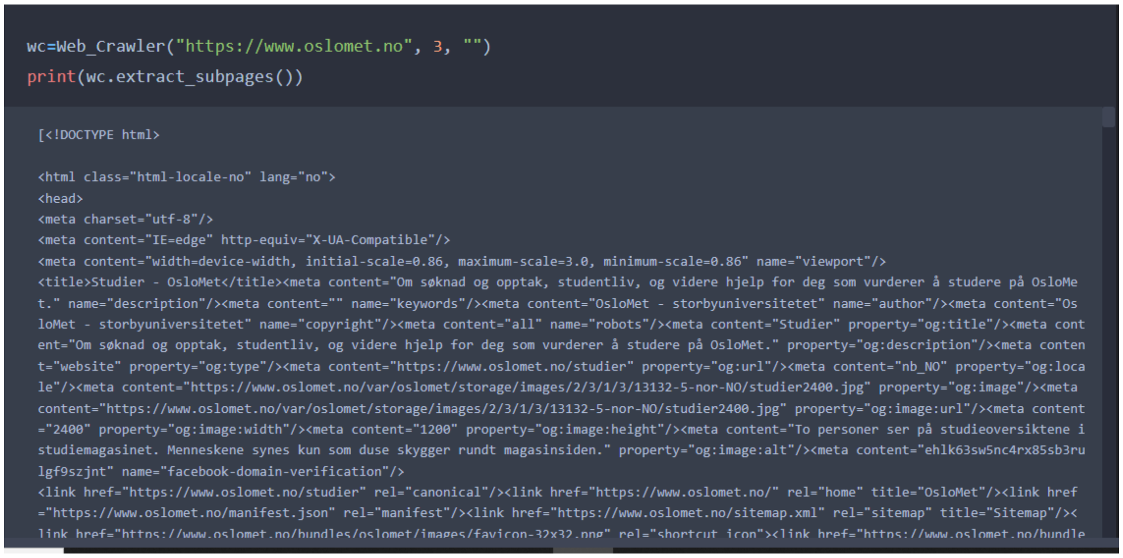
```
wc=Web_Crawler("https://www.oslomet.no", 3, "")
print(wc.extract_links())
```

```
['https://www.oslomet.no/studier', 'https://www.oslomet.no/forskning', 'https://www.oslomet.no/ub', 'https://www.oslomet.no/om', 'https://
www.oslomet.no/om/kontakt', 'https://www.oslomet.no/om/for-studenter', 'https://www.oslomet.no/studier/studieoversikt', 'https://www.oslom
et.no/studier/studenthistorier', 'https://www.oslomet.no/forskning', 'https://www.oslomet.no/forskning/forskningsnyheter', 'https://www.os
lomet.no/om/arrangement/utvikling-norge', 'https://www.oslomet.no/om/arrangement/tedx-oslomet', 'https://www.oslomet.no/en', 'https://www.
oslomet.no/alumni', 'https://www.oslomet.no/om/ansatteoversikt', 'https://www.oslomet.no/om/arrangement/arrangementsoversikt', 'https://ww
w.oslomet.no/om/for-ansatte', 'https://www.oslomet.no/om/for-studenter', 'https://www.oslomet.no/om/ledige-stillinger', 'https://www.oslom
et.no/om/personvernerklering-informasjonskapslar', 'https://www.oslomet.no/om/pressekontakt', 'https://www.oslomet.no/om/tilgjengelegheits
erklering', 'https://ezb.oslomet.no/admin/content/location/55']
```

***Figure [2]***

## **4.3.** Source code of subpages:

The method extract_subpages() takes no parameters, and return a list of the source code of all subpages, based on the given depth. For example, if the depth is 4, then it extracts 4 subpages if found. Let us talk about the implementation now. The method starts by calling the method which extracts the links of all possible subpages, *extract_links()*. Then it checks out that the depth is lower than the subpages. If so, then if it goes recursively over every link and pull out the source code, by using the method read_htmlFile(link), and add to subpages, a local list which takes care of the source code of subpages. Every element represents the source code of a subpage. The figure below shows a test of the method.



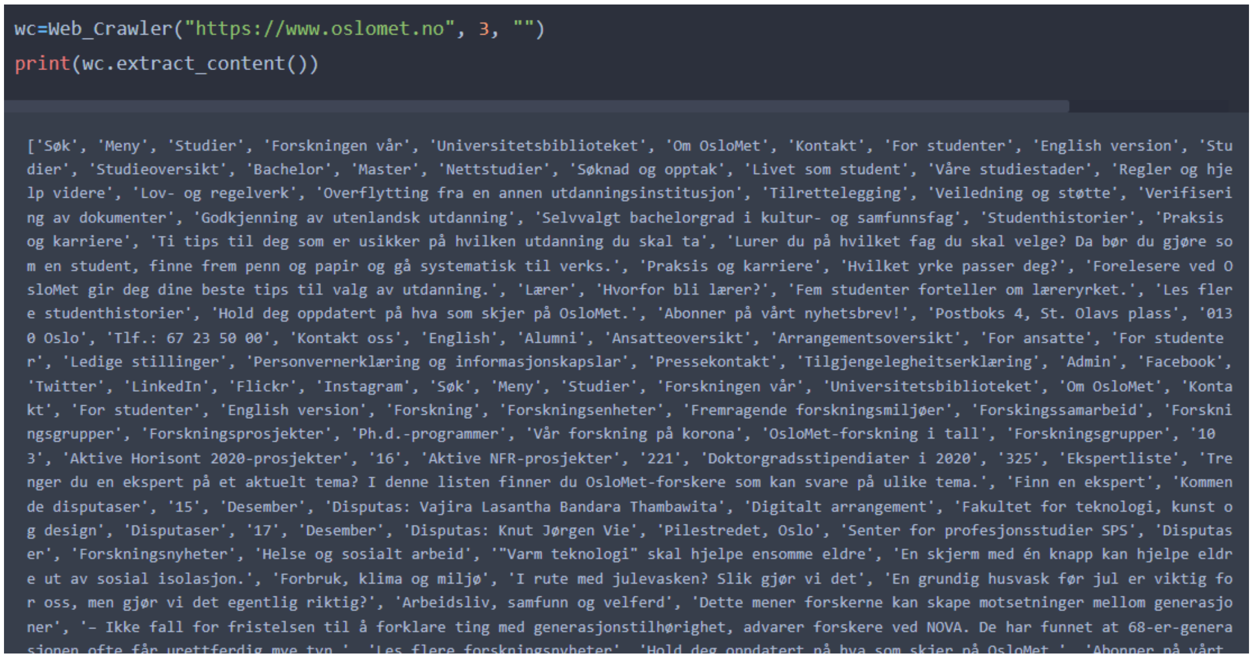
```
wc=Web_Crawler("https://www.oslomet.no", 3, "")
print(wc.extract_subpages())
```

```
[<!DOCTYPE html>

<html class="html-locale-no" lang="no">
<head>
<meta charset="utf-8"/>
<meta content="IE=edge" http-equiv="X-UA-Compatible"/>
<meta content="width=device-width, initial-scale=0.86, maximum-scale=3.0, minimum-scale=0.86" name="viewport"/>
<title>Studier - OsloMet</title><meta content="Om søknad og opptak, studentliv, og videre hjelp for deg som vurderer å studere på OsloMe
t." name="description"/><meta content="" name="keywords"/><meta content="OsloMet - storbyuniversitetet" name="author"/><meta content="Os
loMet - storbyuniversitetet" name="copyright"/><meta content="all" name="robots"/><meta content="Studier" property="og:title"/><meta cont
ent="Om søknad og opptak, studentliv, og videre hjelp for deg som vurderer å studere på OsloMet." property="og:description"/><meta conten
t="website" property="og:type"/><meta content="https://www.oslomet.no/studier" property="og:url"/><meta content="nb_NO" property="og:loca
le"/><meta content="https://www.oslomet.no/var/oslomet/storage/images/2/3/1/3/13132-5-nor-NO/studier2400.jpg" property="og:image"/><meta
content="https://www.oslomet.no/var/oslomet/storage/images/2/3/1/3/13132-5-nor-NO/studier2400.jpg" property="og:image:url"/><meta content
="2400" property="og:image:width"/><meta content="1200" property="og:image:height"/><meta content="To personer ser på studieoversiktene i
studiemagasinet. Menneskene synes kun som duse skygger rundt magasinsiden." property="og:image:alt"/><meta content="ehlk63sw5nc4rx85sb3ru
lgf9szjnt" name="facebook-domain-verification"/>
<link href="https://www.oslomet.no/studier" rel="canonical"/><link href="https://www.oslomet.no/" rel="home" title="OsloMet"/><link href
="https://www.oslomet.no/manifest.json" rel="manifest"/><link href="https://www.oslomet.no/sitemap.xml" rel="sitemap" title="Sitemap"/><
link href="https://www.oslomet.no/bundles/oslomet/images/favicon-32x32.png" rel="shortcut icon"><link href="https://www.oslomet.no/bundle
```

***Figure [3]***

## **4.4.** Extracting content:

The method *extract_content()* extracts all textual and numeric content of the main webpages and the subpages. In other words, it come by every single tag in every single page and takes out its content. In the end, it returns a list of strings. Let us move now to actual implementation. The method starts by initializing a local list called soups. This list includes the html-files of the main page, and other related subpages. then it loops over the strings inside body tag of those pages. as we know, body tag includes the other seen tags, while the head tag has usually several tags which usually are unseen for the users like meta-tags. By looping over *body* tags we will make sure that we capture all content of all tags, in additional to titles of sections. The figure below shows a test of the method.



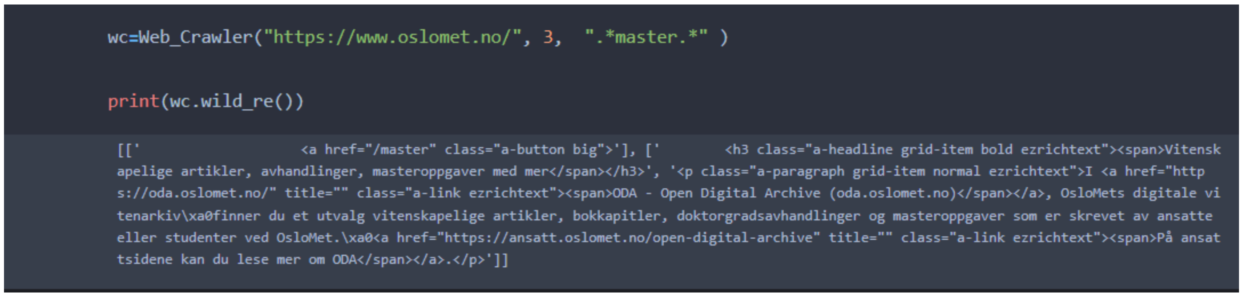
```
wc=Web_Crawler("https://www.oslomet.no", 3, "")
print(wc.extract_content())
```

```
['Søk', 'Meny', 'Studier', 'Forskningen vår', 'Universitetsbiblioteket', 'Om OsloMet', 'Kontakt', 'For studenter', 'English version', 'Stu
dier', 'Studieoversikt', 'Bachelor', 'Master', 'Nettstudier', 'Søknad og opptak', 'Livet som student', 'Våre studiestader', 'Regler og hje
lp videre', 'Lov- og regelverk', 'Overflytting fra en annen utdanningsinstitusjon', 'Tilrettelegging', 'Veiledning og støtte', 'Verifiseri
ng av dokumenter', 'Godkjenning av utenlandsk utdanning', 'Selvvalgt bachelorgrad i kultur- og samfunnsfag', 'Studenthistorier', 'Praksis
og karriere', 'Ti tips til deg som er usikker på hvilken utdanning du skal ta', 'Lurer du på hvilket fag du skal velge? Da bør du gjøre so
m en student, finne frem penn og papir og gå systematisk til verks.', 'Praksis og karriere', 'Hvilket yrke passer deg?', 'Forelesere ved O
sloMet gir deg dine beste tips til valg av utdanning.', 'Lærer', 'Hvorfor bli lærer?', 'Fem studenter forteller om læreryrket.', 'Les fler
e studenthistorier', 'Hold deg oppdatert på hva som skjer på OsloMet.', 'Abonner på vårt nyhetsbrev!', 'Postboks 4, St. Olavs plass', '013
0 Oslo', 'Tlf.: 67 23 50 00', 'Kontakt oss', 'English', 'Alumni', 'Ansatteoversikt', 'Arrangementsoversikt', 'For ansatte', 'For studente
r', 'Ledige stillinger', 'Personvernerklæring og informasjonskapslar', 'Pressekontakt', 'Tilgjengelegheitserklæring', 'Admin', 'Facebook',
'Twitter', 'LinkedIn', 'Flickr', 'Instagram', 'Søk', 'Meny', 'Studier', 'Forskningen vår', 'Universitetsbiblioteket', 'Om OsloMet', 'Konta
kt', 'For studenter', 'English version', 'Forskning', 'Forskningsenheter', 'Fremragende forskningsmiljøer', 'Forskingssamarbeid', 'Forskni
ngsgrupper', 'Forskningsprosjekter', 'Ph.d.-programmer', 'Vår forskning på korona', 'OsloMet-forskning i tall', 'Forskningsgrupper', '10
3', 'Aktive Horisont 2020-prosjekter', '16', 'Aktive NFR-prosjekter', '221', 'Doktorgradsstipendiater i 2020', '325', 'Ekspertliste', 'Tre
nger du en ekspert på et aktuelt tema? I denne listen finner du OsloMet-forskere som kan svare på ulike tema.', 'Finn en ekspert', 'Kommen
de disputaser', '15', 'Desember', 'Disputas: Vajira Lasantha Bandara Thambawita', 'Digitalt arrangement', 'Fakultet for teknologi, kunst o
g design', 'Disputaser', '17', 'Desember', 'Disputas: Knut Jørgen Vie', 'Pilestredet, Oslo', 'Senter for profesjonsstudier SPS', 'Disputas
er', 'Forskningsnyheter', 'Helse og sosialt arbeid', '"Varm teknologi" skal hjelpe ensomme eldre', 'En skjerm med én knapp kan hjelpe eldr
e ut av sosial isolasjon.', 'Forbruk, klima og miljø', 'I rute med julevasken? Slik gjør vi det', 'En grundig husvask før jul er viktig fo
r oss, men gjør vi det egentlig riktig?', 'Arbeidsliv, samfunn og velferd', 'Dette mener forskerne kan skape motsetninger mellom generasjo
ner', '- Ikke fall for fristelsen til å forklare ting med generasjonstilhørighet, advarer forskere ved NOVA. De har funnet at 68-er-genera
sjonen ofte får urettferdig mye tyn.', 'Les flere forskningsnyheter', 'Hold deg oppdatert på hva som skjer på OsloMet.', 'Abonner på vårt
```

***Figure [4]***

## 4.5. Wild regular expression:

The method wild_re() takes no parameters. It searches inside the source code of all webpages; main page is included. Then it finds the strings which matches the given regular expression. Let us move now to talk about the implementation. We first need a local list to take care of the matches if found. We need also to call the link addresses of subpages and the main one and put them in a list. After that our method goes recursively over every link, and make a http request, and pull the html file. Then it reads the file. The type of file is Bytes. this is why we need to decode the file. In other words, we need to convert the Bytes file to a String object to be able to do the matching. It goes on and on until all files are extracted before to return all the matches in a list. The figure below is a test of the method where we passed a regular expression to find the topics which are related to master on the main webpage of OsloMet.



```
wc=Web_Crawler("https://www.oslomet.no/", 3,  ".*master.*" )

print(wc.wild_re())
```

```
[['                    <a href="/master" class="a-button big">'], ['        <h3 class="a-headline grid-item bold ezrichtext"><span>Vitensk
apelige artikler, avhandlinger, masteroppgaver med mer</span></h3>', '<p class="a-paragraph grid-item normal ezrichtext">I <a href="http
s://oda.oslomet.no/" title="" class="a-link ezrichtext"><span>ODA - Open Digital Archive (oda.oslomet.no)</span></a>, OsloMets digitale vi
tenarkiv\xa0finner du et utvalg vitenskapelige artikler, bokkapitler, doktorgradsavhandlinger og masteroppgaver som er skrevet av ansatte
eller studenter ved OsloMet.\xa0<a href="https://ansatt.oslomet.no/open-digital-archive" title="" class="a-link ezrichtext"><span>På ansat
tsidene kan du lese mer om ODA</span></a>.</p>']]
```

***Figure [5]***

## **4.6.** extracting emails:

The method *extract_emails()* takes no parameters, and returns a list of the matched emails, as strings. Emails in html-files are represented in two different ways. It is either as a value of an href-attribute of a-tag as following:

<a href="mailto: xxxxx@xxxx.xxx">  .. </a>

Or as a plain text inside tags for example as the following:

<span>  xxxxx@xxxx.xxx  </span>

Now we will move to talk about the implementation. The method starts by initializing a local variable, a list of strings, called pages. This list has overview on the source code of all webpages, html-files, included the main page, which given as a parameter to the class. Observe here the two methods, *extract_subpages()* and *get_main().* Then it loops over all elements in the list. In other words, it goes recursively over webpages, html-files, and do the following steps:

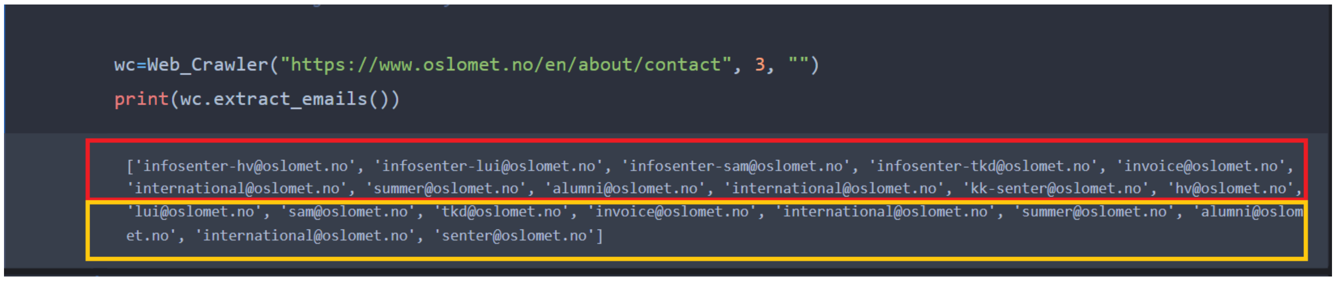
-Firstly, finding all the ‘a’ tags which the value of their href-atrribute starts with ‘mailto’

-Secondly extracting the emails, as strings, from those ‘a’ tags

-Lastly adding them all to *emails*, a local list, by using List method *extend()*

Now we move to say about the second part of the method where our method extracts the emails from the texts in the webpages. Firstly, it calls the method *extract_content()* which brings the texts of all tags in all web-pages, based on the given depth. Then it loops over every single text or string, and check if the texts match the regular expression. If so, it adds them to *emails*, a local variable list.

The figure below Shows a test of the method. The red rectangular stands for the emails which are clickable, links, in the webpages while the yellow one stands for emails which are not clickable, just plain texts.



```
wc=Web_Crawler("https://www.oslomet.no/en/about/contact", 3, "")
print(wc.extract_emails())
```

```
['infosenter-hv@oslomet.no', 'infosenter-lui@oslomet.no', 'infosenter-sam@oslomet.no', 'infosenter-tkd@oslomet.no', 'invoice@oslomet.no',
'international@oslomet.no', 'summer@oslomet.no', 'alumni@oslomet.no', 'international@oslomet.no', 'kk-senter@oslomet.no', 'hv@oslomet.no',
'lui@oslomet.no', 'sam@oslomet.no', 'tkd@oslomet.no', 'invoice@oslomet.no', 'international@oslomet.no', 'summer@oslomet.no', 'alumni@oslom
et.no', 'international@oslomet.no', 'senter@oslomet.no']
```

***Figure [6]***

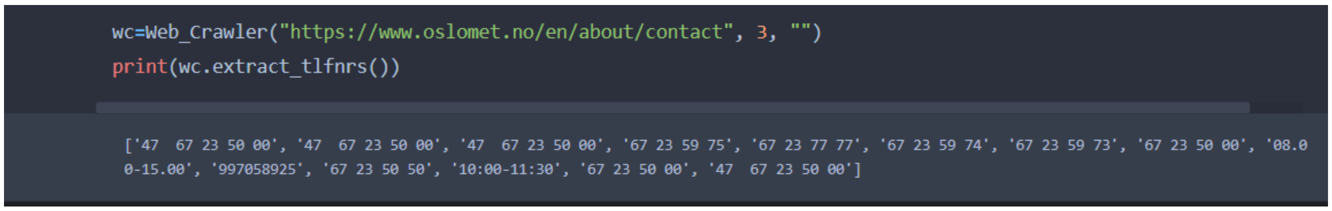
## **4.7.** Telephone numbers:

The method *extract_tlfnrs()* takes no parameters, and returns a list of phone numbers, as strings. It starts by calling the method *extract_content()* which extracts the texts of all tags in all web-pages, based on the given depth. Then it loops over every single text or string, and check if the text matches a regular expression. If so, it adds them to *tlfnumbers*, a local variable list. In this project, the focus on capturing Norwegians mobile phone numbers. Our regular expression matches any phone number which starts, or not, with +47, 0047 0r 47 and has 8 digits. These 8 digits can come in order or every pair of them can be separated by a space, dot or hyphen.

Here are some examples on valid telephone numbers:

+4753421067, 53421067, 0047 53 42 10 67, 53-42-10-67, 47 53.42.10.67

One major method we used is *re.findAll()* which returns a list of tuple(s). Every tuple consists of characters which together form a telephone number. This is why we need to join these chars by a String method called *join()*. It is also worthy to mention that our method can capture more than one phone number in a text if found. The figure below shows the captured phone numbers from the contact page of OsloMet:



```
wc=Web_Crawler("https://www.oslomet.no/en/about/contact", 3, "")
print(wc.extract_tlfnrs())
```

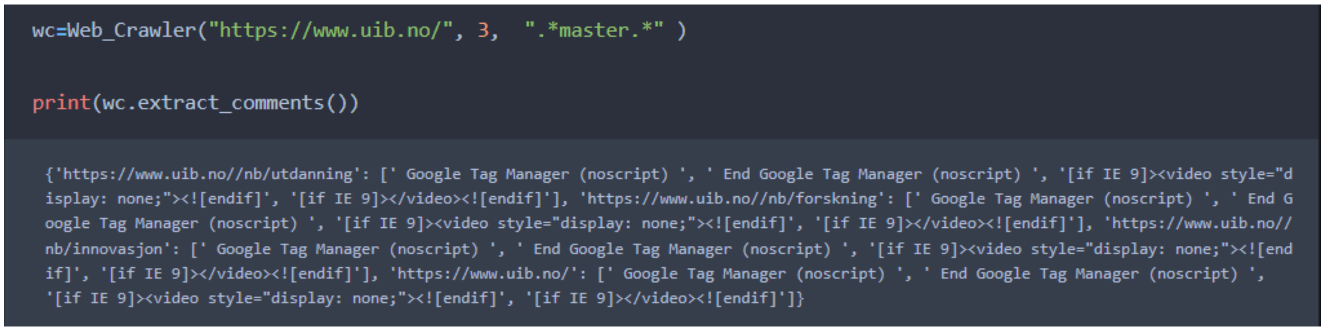
```
['47  67 23 50 00', '47  67 23 50 00', '47  67 23 50 00', '67 23 59 75', '67 23 77 77', '67 23 59 74', '67 23 59 73', '67 23 50 00', '08.0
0-15.00', '997058925', '67 23 50 50', '10:00-11:30', '67 23 50 00', '47  67 23 50 00']
```

***Figure[7]***

## **4.8.** Extracting comments:

The method *extract_comments()* takes no parameters and captures all comments, written in the source code of the pages, together with the link address of the file the comments found in. The method returns a dictionary where the keys are the link addresses of the web pages where the comments exist in, and the values are a list of the self-comments. The notation of comments in html look like as following: <! --write something-->. Now we can dive a little more in the implementation. The method initializes 3 variables. The first one is called links. We use it to keep track on the link addresses of the comments.

Observe than the method *extract_links()* extracts all links of possible subpages of the main one. But we need just a limited amount which the global variable decides for us. The second variable is of type dictionary which we will use to collect the comments there. The third variable is called pages. It fetches the source code of the subpages and the main one. We need it to extract the comments from. After initialization of the variables, we go recursively over every source code of the pages. First, we use *find_all()* from BeutifulSoup library to extract the object which is instances of the class Comment. Observe here we use a lambda function to loop over all parts/texts of the page to extract the comments. After that we add the link address of the current webpage with the matched comments as a new element in the dictionary. We repeat that until the loop is done. Then we return the dictionary. The figure below Shows a test of the method.



```
wc=Web_Crawler("https://www.uib.no/", 3,  ".*master.*" )

print(wc.extract_comments())
```

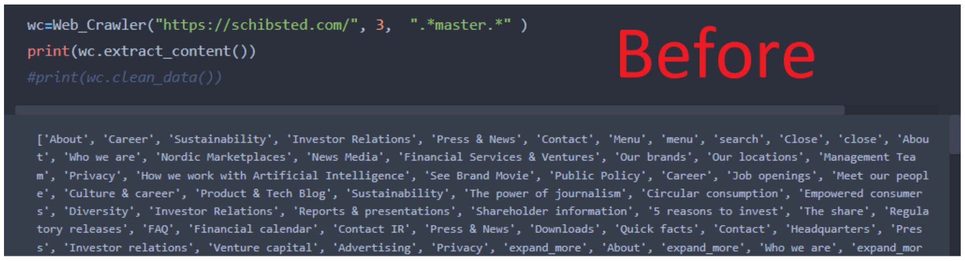
```
{'https://www.uib.no//nb/utdanning': [' Google Tag Manager (noscript) ', ' End Google Tag Manager (noscript) ', '[if IE 9]><video style="d
isplay: none;"><![endif]', '[if IE 9]></video><![endif]'], 'https://www.uib.no//nb/forskning': [' Google Tag Manager (noscript) ', ' End G
oogle Tag Manager (noscript) ', '[if IE 9]><video style="display: none;"><![endif]', '[if IE 9]></video><![endif]'], 'https://www.uib.no//
nb/innovasjon': [' Google Tag Manager (noscript) ', ' End Google Tag Manager (noscript) ', '[if IE 9]><video style="display: none;"><![end
if]', '[if IE 9]></video><![endif]'], 'https://www.uib.no/': [' Google Tag Manager (noscript) ', ' End Google Tag Manager (noscript) ',
'[if IE 9]><video style="display: none;"><![endif]', '[if IE 9]></video><![endif]']}
```

***Figure[8]***

## **4.9.** Word frequency

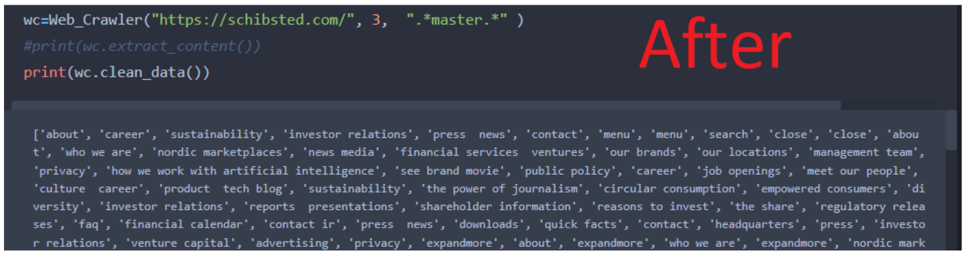
### **4.9.1.** Data cleaning:

Data cleaning is the most important and time-consuming process regarding finding word frequency. It starts by initializing 4 different variables. The first one a string which includes all special chars, the second one is a list of digits as string, the third one is a list that takes care of the cleaned data, and returns them at the end, and the last one is a list of the extracted content, or texts, from the webpages by calling *extract_content().* Then the method does an if-check to make sure that the list of content is not empty. After that it makes a one-line for-loop to exclude empty strings. Then it gets rid of special characters like punctuation marks and digits, by checking every single char in every single word in the extracted content. The next to last step is to convert all strings to lower case because when we will count the frequency of words, we do not want words that start with capital letters to be counted differently from the ones which are the same but has small letters. For example, the words "Hi" and "hi" will be counted as two different words computationally. This is something we want to avoid by make all words to lower case. Then it removes the spaces at the beginning and at the end of a string if found, by using *strip()*, before to add the string to the cleaned list and returns it. The figure below shows how the data looks like before and after cleaning.



```
wc=Web_Crawler("https://schibsted.com/", 3,  ".*master.*" )
print(wc.extract_content())
#print(wc.clean_data())
```

```
['About', 'Career', 'Sustainability', 'Investor Relations', 'Press & News', 'Contact', 'Menu', 'menu', 'search', 'Close', 'close', 'Abou
t', 'Who we are', 'Nordic Marketplaces', 'News Media', 'Financial Services & Ventures', 'Our brands', 'Our locations', 'Management Tea
m', 'Privacy', 'How we work with Artificial Intelligence', 'See Brand Movie', 'Public Policy', 'Career', 'Job openings', 'Meet our peopl
e', 'Culture & career', 'Product & Tech Blog', 'Sustainability', 'The power of journalism', 'Circular consumption', 'Empowered consumer
s', 'Diversity', 'Investor Relations', 'Reports & presentations', 'Shareholder information', '5 reasons to invest', 'The share', 'Regula
tory releases', 'FAQ', 'Financial calendar', 'Contact IR', 'Press & News', 'Downloads', 'Quick facts', 'Contact', 'Headquarters', 'Pres
s', 'Investor relations', 'Venture capital', 'Advertising', 'Privacy', 'expand_more', 'About', 'expand_more', 'Who we are', 'expand_mor
```



```
wc=Web_Crawler("https://schibsted.com/", 3,  ".*master.*" )
#print(wc.extract_content())
print(wc.clean_data())
```

```
['about', 'career', 'sustainability', 'investor relations', 'press  news', 'contact', 'menu', 'menu', 'search', 'close', 'close', 'abou
t', 'who we are', 'nordic marketplaces', 'news media', 'financial services  ventures', 'our brands', 'our locations', 'management team',
'privacy', 'how we work with artificial intelligence', 'see brand movie', 'public policy', 'career', 'job openings', 'meet our people',
'culture  career', 'product  tech blog', 'sustainability', 'the power of journalism', 'circular consumption', 'empowered consumers', 'di
versity', 'investor relations', 'reports  presentations', 'shareholder information', 'reasons to invest', 'the share', 'regulatory relea
ses', 'faq', 'financial calendar', 'contact ir', 'press  news', 'downloads', 'quick facts', 'contact', 'headquarters', 'press', 'investo
r relations', 'venture capital', 'advertising', 'privacy', 'expandmore', 'about', 'expandmore', 'who we are', 'expandmore', 'nordic mark
```

***Figure [9]***

### **4.9.2.** Tagging and Lemmatization

#### **a.** Why we need them

One of our goals we want to achieve in this project is finding the most frequent English words via filtering raw texts. One typical challenge of pc while dealing with natural language is inflections of words. Inflection can be defined as the process of word formation in which items are added to the base form of a word to express a grammatical function, and to mark such distinctions as tense, person, number, gender, mood, voice, and case. For example, the computer will recognize the following words: “go”, “goes”, “went” and “going”, as four different words, not as one word if we do not tell it explicitly via implementation of tagging and lemmatization on the words.

#### **b.** Definitions

According to www.nltk.og , tagging is the process of classifying words into their parts of speech and labeling them accordingly is known as POS-tagging, or simply tagging. Parts of speech are also known as word classes or lexical categories like nouns, verbs and so on. While lemmatization is defined as a morphological analysis of words or cutting the edges which words can stand without. In linguistics, these edges called affixes. In other words, it is the process of returning the word to its base or dictionary form, which is known as the lemma of a word. For example, the lemma of ‘found’, ‘finding’, and ‘finds’ is ‘find’.

### **c.** How they work

POS tagging is a supervised learning solution that uses features like the previous, the next word, and first letter capitalized or not etc. While the lemmmatizer, in help with Regular Expression, looks at the letters which a word starts or/and ends with, called in linguistics prefixes and suffixes, like 're', 'dis', 'ing', and 'ed', and removes them. In addition, it does inflection on words by searching in a map of the word-family to give the contextual base form of that word.

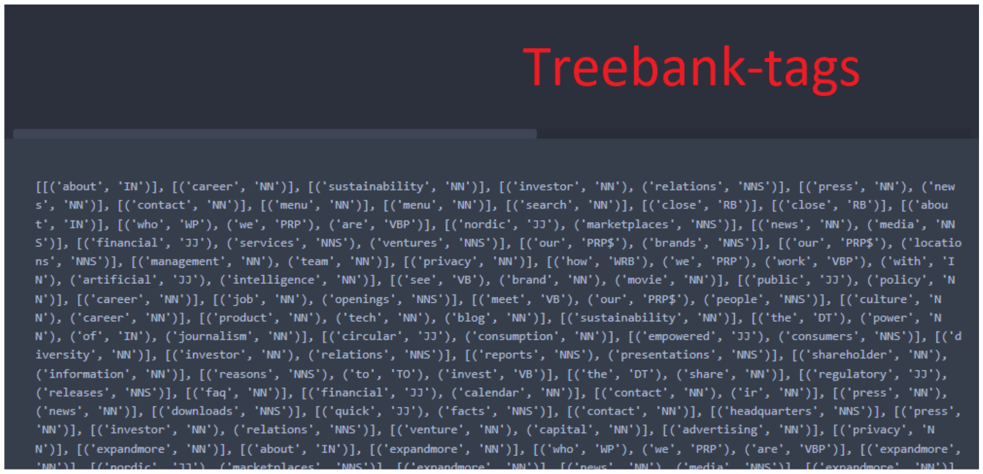
### **d.** Treebank- vs. WordNet-tagger

Treebank-tagger is a powerful tool where it has 36 tags or classes.[7] This gives a precise POS of a word. While WordNet-tagger is a little poor because it has just five POS-tags: 'n' stands for nouns, 'v' for verbs, 'j' for adjectives, and 's' and 'a' for adverbs. [8] WordNet has a good lemmatizer but Treebank does not. In this project we would like to take the benefits of the tagger of Treebank- and lemmatizer of WordNet. It is important to mention that WordNet-lemmatizer only understand POS given by WorNet-tagger. Therefore, we need to map between TreeBank- and WordNet-tags to be able to use Treebank-lemmatizer in combination with WordNet-tagger.

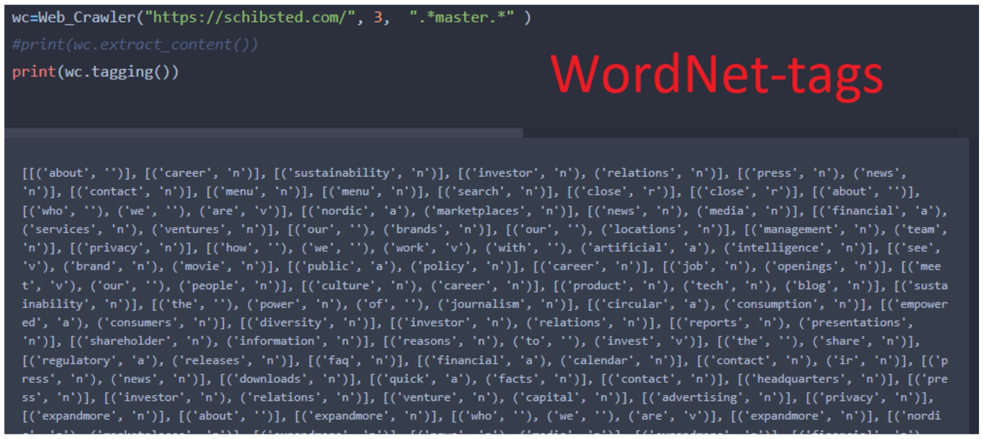
Let us jump now to talk about the implementation of the related methods:

1- tagging

The method ***tagging()*** takes no parameters and return a return a matrix where every list is an element which represent a string. The first list of the main list consists of a pair of a set where the first element of it is the word, and the second element is its part of speech tag. Here is an example of how the returned list looks like: [ [('go', 'verb')....], [...], ... ]. The method starts by initializing three variables; all of them are of type list. The first one, *taggedStrs_Treebank*, is for taking care of the tagged strings via Treebank tagger while the second one is for taking care of the tagged strings via WordNet- tagger, the simpler one which has just 5 parts of speech, after converting them from Treebank-tagger. In fact, this list is the returned list we talked about earlier in this paragraph. The last variable includes the raw data we need to be tagged. Then as the method continues reading the code, we will see an if-check which makes sure that the raw data list is not empty. After that the method loops over every string and check that the string is not an empty list. If so, it calls *nltk.pos_tag()* to gives every word in the string a pos-tag. After that It adds the list of the pairs, words, and tags, to *taggedStrs_Treebank*. Then it makes a temp list which takes care of the converted elements from Treebank- to WordNet-tagger by one-line for-loop. Then it appends these converted values to *to_wornetPosTags*. This heavy process goes on and on until all strings got looped over before returning the list of the lists where every element consists of pairs of words and their pos-tagged values. The figure below shows how the difference between Treebank-tagger and WordNet tagger.



```
[[('about', 'IN')], [('career', 'NN')], [('sustainability', 'NN')], [('investor', 'NN'), ('relations', 'NNS')], [('press', 'NN'), ('new
s', 'NN')], [('contact', 'NN')], [('menu', 'NN')], [('menu', 'NN')], [('search', 'NN')], [('close', 'RB')], [('close', 'RB')], [('abou
t', 'IN')], [('who', 'WP'), ('we', 'PRP'), ('are', 'VBP')], [('nordic', 'JJ'), ('marketplaces', 'NNS')], [('news', 'NN'), ('media', 'NN
S')], [('financial', 'JJ'), ('services', 'NNS'), ('ventures', 'NNS')], [('our', 'PRP$'), ('brands', 'NNS')], [('our', 'PRP$'), ('locatio
ns', 'NNS')], [('management', 'NN'), ('team', 'NN')], [('privacy', 'NN')], [('how', 'WRB'), ('we', 'PRP'), ('work', 'VBP'), ('with', 'I
N'), ('artificial', 'JJ'), ('intelligence', 'NN')], [('see', 'VB'), ('brand', 'NN'), ('movie', 'NN')], [('public', 'JJ'), ('policy', 'N
N')], [('career', 'NN')], [('job', 'NN'), ('openings', 'NNS')], [('meet', 'VB'), ('our', 'PRP$'), ('people', 'NNS')], [('culture', 'N
N'), ('career', 'NN')], [('product', 'NN'), ('tech', 'NN'), ('blog', 'NN')], [('sustainability', 'NN')], [('the', 'DT'), ('power', 'N
N'), ('of', 'IN'), ('journalism', 'NN')], [('circular', 'JJ'), ('consumption', 'NN')], [('empowered', 'JJ'), ('consumers', 'NNS')], [('d
iversity', 'NN')], [('investor', 'NN'), ('relations', 'NNS')], [('reports', 'NNS'), ('presentations', 'NNS')], [('shareholder', 'NN'),
('information', 'NN')], [('reasons', 'NNS'), ('to', 'TO'), ('invest', 'VB')], [('the', 'DT'), ('share', 'NN')], [('regulatory', 'JJ'),
('releases', 'NNS')], [('faq', 'NN')], [('financial', 'JJ'), ('calendar', 'NN')], [('contact', 'NN'), ('ir', 'NN')], [('press', 'NN'),
('news', 'NN')], [('downloads', 'NNS')], [('quick', 'JJ'), ('facts', 'NNS')], [('contact', 'NN')], [('headquarters', 'NNS')], [('press',
'NN')], [('investor', 'NN'), ('relations', 'NNS')], [('venture', 'NN'), ('capital', 'NN')], [('advertising', 'NN')], [('privacy', 'N
N')], [('expandmore', 'NN')], [('about', 'IN')], [('expandmore', 'NN')], [('who', 'WP'), ('we', 'PRP'), ('are', 'VBP')], [('expandmore',
```



```
wc=Web_Crawler("https://schibsted.com/", 3,  ".*master.*" )
#print(wc.extract_content())
print(wc.tagging())
```

```
[[('about', '')], [('career', 'n')], [('sustainability', 'n')], [('investor', 'n'), ('relations', 'n')], [('press', 'n'), ('news',
'n')], [('contact', 'n')], [('menu', 'n')], [('menu', 'n')], [('search', 'n')], [('close', 'r')], [('close', 'r')], [('about', '')],
[('who', ''), ('we', ''), ('are', 'v')], [('nordic', 'a'), ('marketplaces', 'n')], [('news', 'n'), ('media', 'n')], [('financial', 'a'),
('services', 'n'), ('ventures', 'n')], [('our', ''), ('brands', 'n')], [('our', ''), ('locations', 'n')], [('management', 'n'), ('team',
'n')], [('privacy', 'n')], [('how', ''), ('we', ''), ('work', 'v'), ('with', ''), ('artificial', 'a'), ('intelligence', 'n')], [('see',
'v'), ('brand', 'n'), ('movie', 'n')], [('public', 'a'), ('policy', 'n')], [('career', 'n')], [('job', 'n'), ('openings', 'n')], [('mee
t', 'v'), ('our', ''), ('people', 'n')], [('culture', 'n'), ('career', 'n')], [('product', 'n'), ('tech', 'n'), ('blog', 'n')], [('susta
inability', 'n')], [('the', ''), ('power', 'n'), ('of', ''), ('journalism', 'n')], [('circular', 'a'), ('consumption', 'n')], [('empower
ed', 'a'), ('consumers', 'n')], [('diversity', 'n')], [('investor', 'n'), ('relations', 'n')], [('reports', 'n'), ('presentations',
'n')], [('shareholder', 'n'), ('information', 'n')], [('reasons', 'n'), ('to', ''), ('invest', 'v')], [('the', ''), ('share', 'n')],
[('regulatory', 'a'), ('releases', 'n')], [('faq', 'n')], [('financial', 'a'), ('calendar', 'n')], [('contact', 'n'), ('ir', 'n')], [('p
ress', 'n'), ('news', 'n')], [('downloads', 'n')], [('quick', 'a'), ('facts', 'n')], [('contact', 'n')], [('headquarters', 'n')], [('pre
ss', 'n')], [('investor', 'n'), ('relations', 'n')], [('venture', 'n'), ('capital', 'n')], [('advertising', 'n')], [('privacy', 'n')],
[('expandmore', 'n')], [('about', '')], [('expandmore', 'n')], [('who', ''), ('we', ''), ('are', 'v')], [('expandmore', 'n')], [('nordi
```

***Figure [10]***

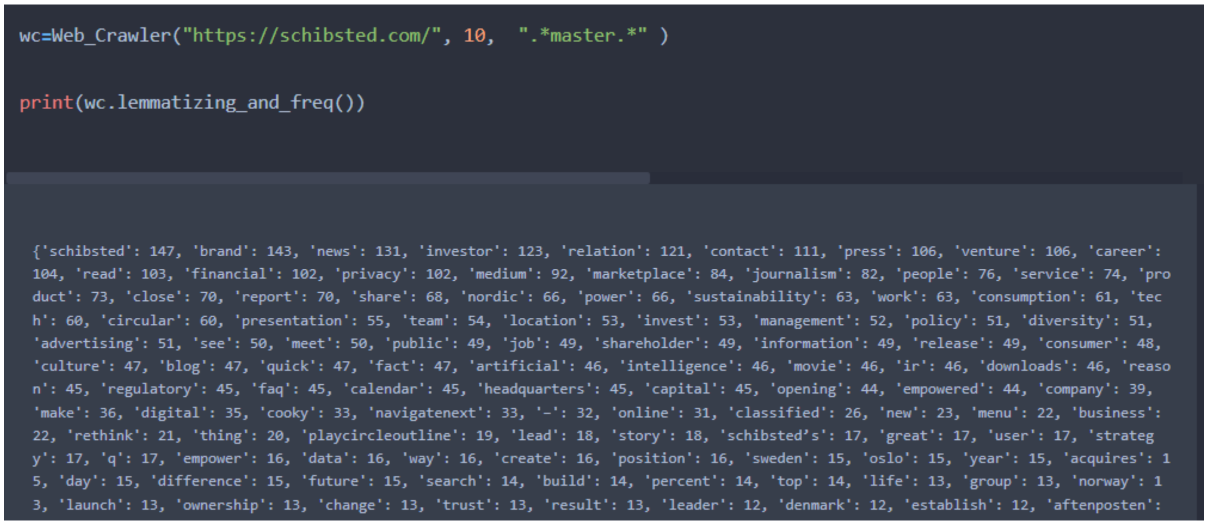
2- converting to WN-tags

*to_wordnet_tag()* is a private method which takes one parameter. This parameter is a list of Treebank-tags where every tag in it corresponds to a word in another list, which have overview on the targeted words. The main goal of this method is just to make an overlapping between BT tags and WordNet-tags. In other words, the method somehow compresses the size of Treebank-tags. We mean that the 35 tags would be reduced to 5 main tags via WordNet-tagger. We want also to pay your attention that the tags which are not important, like propositions and pronouns, takes no tag in WordNet-tagger like 'about', the first string. We want to mention that this simple method is borrowed from Stackoverflow[source].

https://stackoverflow.com/questions/15586721/wordnet-lemmatization-and-pos-tagging-in-python

3- *lemmas and frequency*

Before to start talking about the method *lemmatizing_and_freq()*, we want to remind you that the lemmatizing process is important to remove the morphological edges like 'ed', 'es',and 'ing' from words to make correct counting on them. In the previous method, *tagging()*, we converted the Treebank-tags into WordNet-tags because we need to take benefit of WordNet-lemmatizer which recognizes WordNet-tags, but not Treebank-tags. We are going now to jump to the implementation. The method starts by initializing the main local variables. The first one is an object of WordNetLemmatizer(). The second one is a list of so-called function words given by nltk-model. This list consists of 179 words that are considered meaningless or unwanted where they give no new information to understand the content of the texts, like pronouns, propositions, wh-questions, and fragments. They just have grammatical uses, in addition, their high frequency in the texts. This is why we do not want to take them into consideration in our word frequency list. We also want to mention that we add more unimportant words to this list to optimize our results. Anyway, these words cannot be lemmatized because they will never have affixes/extensions like verbs, nouns and so on. Then the method initializes a local variable with name *lemmatized_strs* and of type list, with two dimensions. This list that will contain the lemmatized strings or sentences. Observe here that every element of this list will be a list of words, as strings that will be lemmatized. Then the method loops over every string in the tagged strings list and initializes a temporary list of one dimension. This list will take contains the lemmatized words of a string per round. Then the method makes a for-loop, inside the main one, to go through every word and its tag in the current string. It first makes the word to lower case and check that it has a tag. In other words, in this step the method drops the stop words, and the words which gives no new information to us, like prepositions and pronouns as we mentioned earlier. otherwise, the word will get lemmatized by *lemmatizer.lemmatize()* which takes two parameters the first one is the word and the second one, is the word's pos-tag, one of the 5 WordNet tags. Note that the second parameter, pos-tag, is optional but it makes the lemmatizer much more precis in giving predictions. Then it adds the pair to the list, and added to the temporary list *lemmatized_str*, further to the matrix *lemmatized_strs*. As the code runs, another loop inside the main one is needed to find the most frequent words and their frequency. This new loop goes through every single word in the current string, and check if it exists in the global dictionary *self.lemmas_with_freq* which will contain all the lemmatized words as keys with their frequency as values. If the word is in this dictionary, it will increase the frequency with one. Otherwise, it will add new element where the word is the key and 1 is the value. Lastly the mothed need to sort the words according to their frequency in the pages. Since there is no way to directly sort a dictionary based on values. Our method needed to convert the dictionary to a list of tuples. Then it converts the list back to dictionary where now the words are sorted in descending order. In other words, the most frequented words come first in the dictionary. The figure below presents the most frequent words in the first 10 webpages of a website of a famous company called Schibsted.



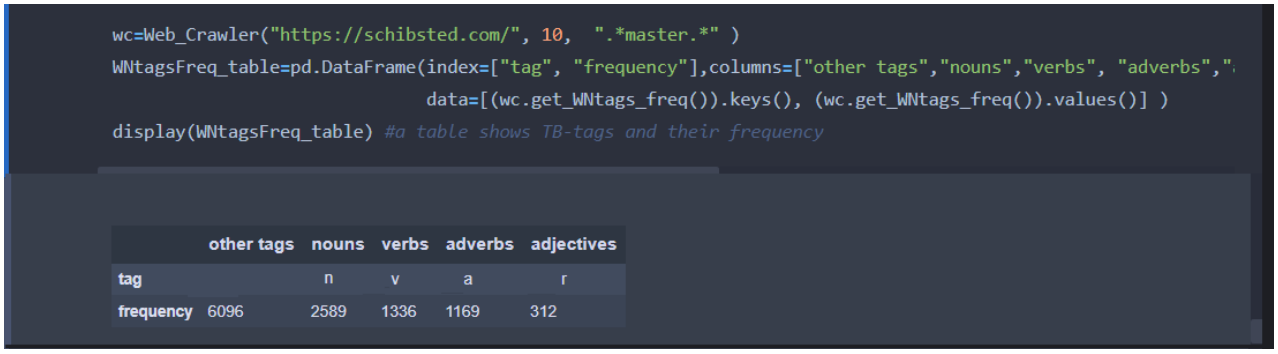
```
wc=Web_Crawler("https://schibsted.com/", 10,  ".*master.*" )

print(wc.lemmatizing_and_freq())
```

```
{'schibsted': 147, 'brand': 143, 'news': 131, 'investor': 123, 'relation': 121, 'contact': 111, 'press': 106, 'venture': 106, 'career':
104, 'read': 103, 'financial': 102, 'privacy': 102, 'medium': 92, 'marketplace': 84, 'journalism': 82, 'people': 76, 'service': 74, 'pro
duct': 73, 'close': 70, 'report': 70, 'share': 68, 'nordic': 66, 'power': 66, 'sustainability': 63, 'work': 63, 'consumption': 61, 'tec
h': 60, 'circular': 60, 'presentation': 55, 'team': 54, 'location': 53, 'invest': 53, 'management': 52, 'policy': 51, 'diversity': 51,
'advertising': 51, 'see': 50, 'meet': 50, 'public': 49, 'job': 49, 'shareholder': 49, 'information': 49, 'release': 49, 'consumer': 48,
'culture': 47, 'blog': 47, 'quick': 47, 'fact': 47, 'artificial': 46, 'intelligence': 46, 'movie': 46, 'ir': 46, 'downloads': 46, 'reaso
n': 45, 'regulatory': 45, 'faq': 45, 'calendar': 45, 'headquarters': 45, 'capital': 45, 'opening': 44, 'empowered': 44, 'company': 39,
'make': 36, 'digital': 35, 'cooky': 33, 'navigatenext': 33, '-': 32, 'online': 31, 'classified': 26, 'new': 23, 'menu': 22, 'business':
22, 'rethink': 21, 'thing': 20, 'playcircleoutline': 19, 'lead': 18, 'story': 18, 'schibsted's': 17, 'great': 17, 'user': 17, 'strateg
y': 17, 'q': 17, 'empower': 16, 'data': 16, 'way': 16, 'create': 16, 'position': 16, 'sweden': 15, 'oslo': 15, 'year': 15, 'acquires': 1
5, 'day': 15, 'difference': 15, 'future': 15, 'search': 14, 'build': 14, 'percent': 14, 'top': 14, 'life': 13, 'group': 13, 'norway': 1
3, 'launch': 13, 'ownership': 13, 'change': 13, 'trust': 13, 'result': 13, 'leader': 12, 'denmark': 12, 'establish': 12, 'aftenposten':
```

***Figure [11]***

4- Tags frequency

The method get_WNtags_freq() takes no parameters and return a sorted dictionary where the keys are WordNet -tags and the values are their frequency. -it starts by initializing a local dictionary. Then It call the method *tagging()* which return a matrix. Every list in this matrix consists of one or more tuples. Every tuple consists of two elements, the first one is the captured word, and the second one is its tag. The method *get_WNtags_freq()* loops over every sentence or string. Then it makes a new for-loop to go through every tag of the words. If the tag is in the local dictionary, it adds its value, frequency, with 1. If it has not already existed in the dictionary, it adds the tag to the dictionary as a new element with value 1. After that our method need to convert the dictionary to a list of tuples. Then it converts the list back to dictionary where now the tags are sorted in descending order. In other words, the most frequented tags come first in the dictionary. I want to pay your attention to that we use data frames from Pandas in the test area to show the tags together with their frequency. The figure below shows Frequency of tags.



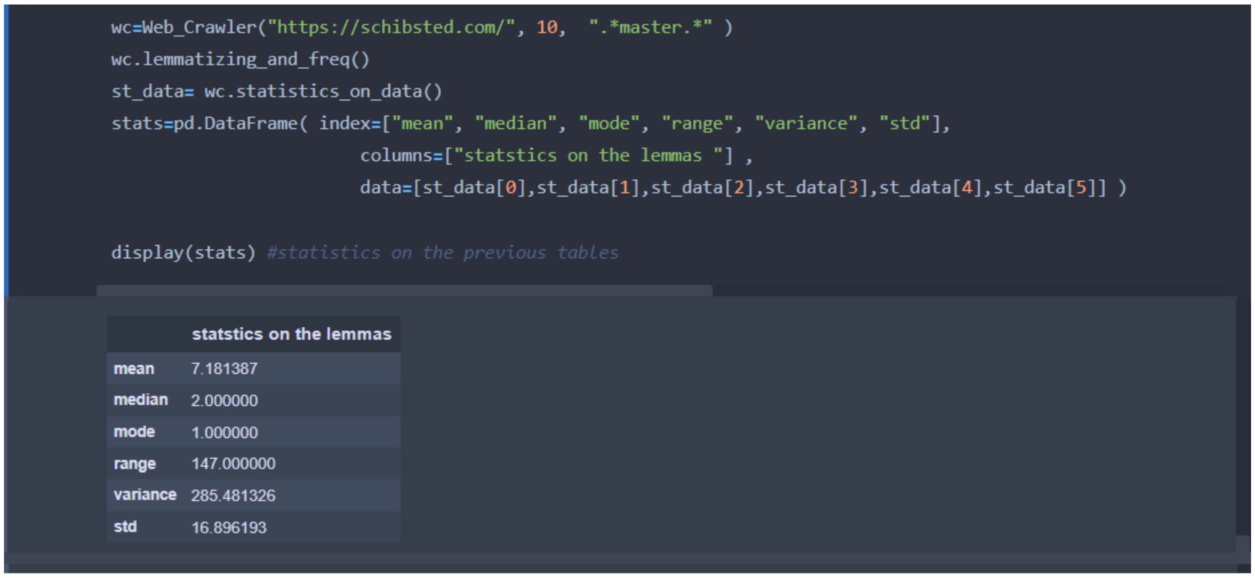
```
wc=Web_Crawler("https://schibsted.com/", 10,  ".*master.*" )
WNtagsFreq_table=pd.DataFrame(index=["tag", "frequency"],columns=["other tags","nouns","verbs", "adverbs","
                            data=[(wc.get_WNtags_freq()).keys(), (wc.get_WNtags_freq()).values()] )
display(WNtagsFreq_table) #a table shows TB-tags and their frequency
```

| | other tags | nouns | verbs | adverbs | adjectives |
|---|---|---|---|---|---|
| **tag** | | n | v | a | r |
| **frequency** | 6096 | 2589 | 1336 | 1169 | 312 |

***Figure [12]***

## 5- Statistics on lemmas

We took initiative to build from scratch a model which provides some basic functions of statistics. It will somehow substitute the use of external libraries like stats. The following functions are Among the ones we made are median, standard deviation, quartiles, z scores, and coefficient of variance. We found useful to make a function which detect outliers as well. An we made a method called statistics_on_data() which do all work inside the main class, Web_Crawler. Note that we use data frames from Pandas to show the tags together with their frequency as u can see in the figure below.

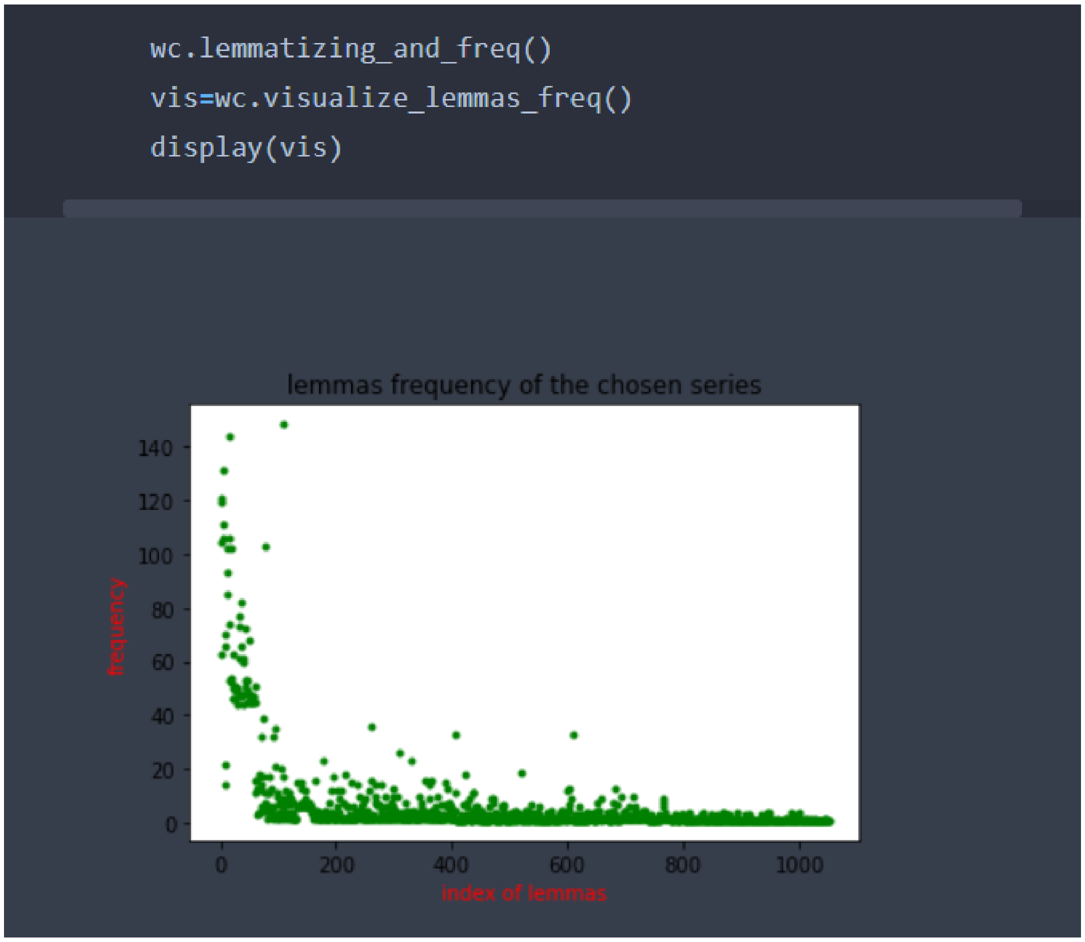


```
wc=Web_Crawler("https://schibsted.com/", 10,  ".*master.*" )
wc.lemmatizing_and_freq()
st_data= wc.statistics_on_data()
stats=pd.DataFrame( index=["mean", "median", "mode", "range", "variance", "std"],
                          columns=["statstics on the lemmas "] ,
                          data=[st_data[0],st_data[1],st_data[2],st_data[3],st_data[4],st_data[5]] )

display(stats) #statistics on the previous tables
```

| | statstics on the lemmas |
|---|---|
| mean | 7.181387 |
| median | 2.000000 |
| mode | 1.000000 |
| range | 147.000000 |
| variance | 285.481326 |
| std | 16.896193 |

***Figure [13]***

We see immediately how wide the range is between the first data point and the last data point (148-1=147). The fact is that the range do not always give a good impression of the variability of the data, because it deals with data which can be outliers or extreme values. The median also got effected by extreme values, but it is more informative than the range. Anyways, it seems that the standard deviation is the most informative in this case. The info. about median shows that half of lemmas frequent 2 times in the webpages. The mode shows that the most frequent value of the frequency of lemmas is 1! The figure below will give a better vision about the distribution of the frequency values of the lemmas.



```
wc.lemmatizing_and_freq()
vis=wc.visualize_lemmas_freq()
display(vis)
```

***Figure [14]***

## 5. Final evaluation

**5.1.** General view:

- On general speaking, we achieved successfully all the requirements with some room for minor errors.
- Obviously, no web crawler will fit for alle websites. But we focus on common probertites among websites.
- We avoided to put restriction on methods, like to make them private, because we wanted to make it easy for you to test every method on its own.
- We intended to combine between getters and setters to make the code readable. And not make many instance or global variables to make the code more readable.
- The rapport might seem a little short but we tried avoid redundancy.

- We could handle Exceptions in a better way but high-level libraries take in consideration exceptions casting.

## **5.2.** Download the website pages:

- Our model pulls the source code of a webpage with its subpages, and add them in lists. We could instead download or write source code of them to txt-files, and then download these files, but that is unnecessary.
- We want to mention also that assigning very high depth can take some time when running. So it may be a good idea to use better GPU, use CPU or try to optimize the code.
- *read_htmlFile()* work perfectly unless URL is broken, some problems related to server appear, webpages-owners put restriction on access to the source code, or some problems to users' pc happen.
- *extract_content()* works as expected unless it appears some problems with the extracted links.
- The method *extract_links()* works well when the given URL is the main page but it can lead to problems when the given URL is its self a subpage, and the links on this subpage are not subpages of it. For example, let us assume that the given URL from user is: https://www.oslomet.no/en, and a not-fully-given link on this page looks like: /studier. Consequently, when concatenation, we will get https://www.oslomet.no/en/studier which are not a valid link!

## **5.3.** emails and phones numbers:

The method extract_emails() which captures emails are optimized. It captured all emails on the given website and its subpages but there is one email which it was not able to deal with. This emailhas a little strange structure like: xxxx[a]xx.xx. While the method which capture telephone numbers shows some percentage of error is 2/14, as showen in figure[x]

## **5.4.** Comments and wild regex:

As long as our method, *extract_coments(),* which extract comments goes through every line in the source code of all pages, we think that there will be no room for error in capturing comments because comments have a unique structure. There are objects of class called Comment.

Regarding regex, our method*, wild_re()*, does a good job. We want you to notice that we solve this part by using the library of Regular Expression, re. We could solve it by beautifulSoup library instead. But REGEX tends to give a higher accuracy and more flexibility although Beautiful Soup's methods is more readable. We want to point out that this method is a little slow when running and this is unsurprising.

## **5.5.** Tagging and lemmatizing:

Data cleaning and the accuracy of tagging can affect dramatically on the accuracy of the lemmatizer, but not vise verse. We do not have labeled data to test the accuracy of the Treebank/tagger but based on our linguistic knowledge we can say that the error percentage of Treebank/tagger is low, but there is no room for error in WordNet-tagger because they are converted directedly from Treebank/tagger. How good a lemmatizer says a lot about how good the used tagger is. How good a lemmatizer says a lot about how good a tagger is. If we have a close look at the lemmas, words which are lemmatized, in Figure xx, we will observe immediately how high the accuracy of the lemmatizer is.  Anyway, we found few words which might be lemmatized wrongly like 'classified' and 'empowered'. They could be lematized as 'classify' and 'empower' if their pos-tag is 'verb'. But if their actual pos-tag is adjective them, the lemmatizer is correct the classification/lemmatizatiing these two words. We are sure about that the lemmatizer failed to remove 's' and '’s' from 'headquarters' and 'schibsted’s'. We think also that the compound words can be tricky for the tagger and lemmatizer to deal with, like for example ‘afterlife’ and ‘colored-hair’. Another thing to mention here is that some words seem to be misspelled on the webpages like ‘navigatenext’ and ‘playcircleoutline’. They cn also be a big challenge for the tagger and the lemmatizer. Last but not least, we want to mention that the method being able to lemmatize just English words can be considered as a downside. But a good number of languages, included Norwegian, have own lemmatizer which can be used when needed.

## 6. References:

1- Web scraping

https://en.wikipedia.org/wiki/Web_scraping

2- beautiful-soup library

https://beautiful-soup-4.readthedocs.io/en/latest/

3- requests library

https://en.wikipedia.org/wiki/Requests_(software)

4- nltk model (for natural languages)

https://nltk.org/

5- A method borrowed from Stack overflow
https://stackoverflow.com/questions/15586721/wordnet-lemmatization-and-pos-tagging-in-python

6- Regulations for web-crawling
https://towardsdatascience.com/is-web-crawling-legal-a758c8fcacde

7- Treebank pos tags list
https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.htmL

8- wordnet pos tags list
https://wordnet.princeton.edu/documentation/wndb5wn