CORD-19 Collect SCOPUS data

In general, this notebook is designated to collect additional data via the Elsevier SCOPUS API to enrich the analysis. https://dev.elsevier.com/ (https://dev.elsevier.com/)

First, relevant packages must be imported into the notebook.

In [1]:

```
import numpy as np
import pandas as pd
import csv
import ast
import collections
import matplotlib.pyplot as plt
import re
import time
import json
from urllib.parse import urlparse
from collections import Counter
from elsapy.elsclient import ElsClient
from elsapy.elsdoc import FullDoc, AbsDoc
from elsapy.elssearch import ElsSearch
from pybtex.database import parse_file, BibliographyData, Entry
```

Retrieve columns from the CORD19 CSV and store the data to a variable.

In [2]:

```
CORD19_CSV = pd.read_csv('../data/cord-19/CORD19_software_mentions.csv')
```

Check the length of the column containing doi's.

In [3]:

```
len(CORD19_CSV['doi'])
```

Out[3]:

```
77448
```

Display the column doi to see if there are inconsistencies such as NaN's. Subsequently, the existence of NaN's requires specific consideration.

In [4]:

```
doi = CORD19_CSV['doi']
doi
```

Out[4]:

```
0                                  NaN
1           10.1016/j.regg.2021.01.002
2            10.1016/j.rec.2020.08.002
3         10.1016/j.vetmic.2006.11.026
4                   10.3390/v12080849
                      ...
77443       10.1007/s11229-020-02869-9
77444                              NaN
77445      10.1101/2020.05.13.20100206
77446       10.1007/s42991-020-00052-8
77447      10.1101/2020.09.14.20194670
Name: doi, Length: 77448, dtype: object
```

Create a series with solely unique values and neglect NaN's. It is important to sort the unique values. Otherwise, the method is creating different results after restarting the notebook.

In [5]:

```
doi_counted = doi.value_counts().sort_index(ascending=True)
doi_counted
```

Out[5]:

```
10.1001/jamainternmed.2020.1369       1
10.1001/jamanetworkopen.2020.16382    1
10.1001/jamanetworkopen.2020.17521    1
10.1001/jamanetworkopen.2020.20485    1
10.1001/jamanetworkopen.2020.24984    1
                                     ..
10.9745/ghsp-d-20-00115               1
10.9745/ghsp-d-20-00171               1
10.9745/ghsp-d-20-00218               1
10.9758/cpn.2020.18.4.607             1
10.9781/ijimai.2020.02.002            1
Name: doi, Length: 74302, dtype: int64
```

The following function determines the requested information from the Scopus API.

In [6]:

```
#Code adapted from https://github.com/ElsevierDev/elsapy/blob/master/exampleProg.py
def fetch_scopus_api(client, doi):
    """obtain additional paper information from scopus by doi
    """
    doc_srch = ElsSearch("DOI("+doi+")",'scopus')
    doc_srch.execute(client, get_all = True)
    try:
        scopus_id=doc_srch.results[0]["dc:identifier"].split(":")[1]
        scp_doc = AbsDoc(scp_id = scopus_id)
        if scp_doc.read(client):
            scp_doc.write()
        else:
            print ("Read document failed.")
        return scp_doc.data
    except:
        return None
```

Thusly, the configuration file is set up and contains an API key. For further information visit the following website: https://github.com/ElsevierDev/elsapy/blob/master/CONFIG.md (https://github.com/ElsevierDev/elsapy/blob/master/CONFIG.md)

In [7]:

```
con_file = open("config.json")
config = json.load(con_file)
con_file.close()
```

Moreover, the client is initialized with the API-Key. It is important to know that the API key needs to be valid to fetch data. Otherwise, the SCOPUS API responds with an authorisation/access error.

In [8]:

```
client = ElsClient(config['apikey'])
```

For demonstration purposes, the following cells show which data is returned by the Scopus API.

In [9]:

```
return_example = fetch_scopus_api(client, '10.1016/j.dsx.2020.04.012')
print(json.dumps(return_example, indent=2))
```


```
{
  "affiliation": [
    {
      "affiliation-city": "New Delhi",
      "affilname": "Jamia Hamdard",
      "affiliation-country": "India"
    },
    {
      "affiliation-city": "New Delhi",
      "affilname": "Jamia Millia Islamia",
      "affiliation-country": "India"
    },
    {
      "affiliation-city": "New Delhi",
      "affilname": "Indraprastha Apollo Hospitals",
      "affiliation-country": "India"
    }
  ],
  "coredata": {
    "srctype": "j",
    "eid": "2-s2.0-85083171050",
    "pubmed-id": "32305024",
    "prism:coverDate": "2020-07-01",
    "prism:aggregationType": "Journal",
    "prism:url": "https://api.elsevier.com/content/abstract/scopus_id/850831
71050",
    "dc:creator": {
      "author": [
        {
          "ce:given-name": "Raju",
          "preferred-name": {
            "ce:given-name": "Raju",
            "ce:initials": "R.",
            "ce:surname": "Vaishya",
            "ce:indexed-name": "Vaishya R."
          },
          "@seq": "1",
          "ce:initials": "R.",
          "@_fa": "true",
          "affiliation": {
            "@id": "60019286",
            "@href": "https://api.elsevier.com/content/affiliation/affiliati
on_id/60019286"
          },
          "ce:surname": "Vaishya",
          "@auid": "6602902951",
          "author-url": "https://api.elsevier.com/content/author/author_id/6
602902951",
          "ce:indexed-name": "Vaishya R."
        }
      ]
    },
    "link": [
      {
        "@_fa": "true",
        "@rel": "self",
```

```
        "@href": "https://api.elsevier.com/content/abstract/scopus_id/850831
71050"
      },
      {
        "@_fa": "true",
        "@rel": "scopus",
        "@href": "https://www.scopus.com/inward/record.uri?partnerID=HzOxMe3
b&scp=85083171050&origin=inward"
      },
      {
        "@_fa": "true",
        "@rel": "scopus-citedby",
        "@href": "https://www.scopus.com/inward/citedby.uri?partnerID=HzOxMe
3b&scp=85083171050&origin=inward"
      }
    ],
    "source-id": "5700165201",
    "pii": "S1871402120300771",
    "citedby-count": "209",
    "prism:volume": "14",
    "subtype": "ar",
    "dc:title": "Artificial Intelligence (AI) applications for COVID-19 pand
emic",
    "openaccess": "1",
    "prism:issn": "18780334 18714021",
    "prism:issueIdentifier": "4",
    "subtypeDescription": "Article",
    "prism:publicationName": "Diabetes and Metabolic Syndrome: Clinical Rese
arch and Reviews",
    "prism:pageRange": "337-339",
    "prism:endingPage": "339",
    "openaccessFlag": "true",
    "prism:doi": "10.1016/j.dsx.2020.04.012",
    "prism:startingPage": "337",
    "dc:identifier": "SCOPUS_ID:85083171050",
    "dc:publisher": "Elsevier Ltd"
  }
}
```


Further analysis will be conducted with such data as fetched above. Therefore, two notebooks are created to analyse data linked to:

- affiliation (CORD-19-analyse-affiliation-data-CS5099)
- coredata (CORD-19-analyse-coredata-CS5099)

Due to the ethical guidelines of this project, the retrieved data is not stored completely in the directory. Before storing fetched data, the following information is removed from the response:

- "affilname" and "affiliation-city"
- "dc:creator"

Therefore, the following functions transform the fetched information into an ethically correct form.

In [10]:

```
def remove_unethical_entries(json_holder):
    """
    This function receives a JSON and removes ethically sensitive data from it. Therefore,
    When the removal is finished, the cleaned JSON is returned to the invoking place of thi
    """
    #Checking if JSON is None
    if json_holder is None:
        return json_holder

     #Checking if JSON is NaN
    if json_holder == "NaN":
        return None

    #JSON starts with { else [
    string_helper = str(json_holder)
    #print(string_helper[0])
    if string_helper[0] == "{":
            if 'affiliation-city' in json_holder:
                del json_holder['affiliation-city']
            if 'affilname' in json_holder:
                 del json_holder['affilname']
            if 'dc:creator' in json_holder:
                 del json_holder['dc:creator']
            return json_holder
    elif string_helper[0] == "[":
        #print(json_holder)
        for element in json_holder:
            if 'affiliation-city' in element.keys():
                del element['affiliation-city']
            if 'affilname' in element.keys():
                 del element['affilname']
            if 'dc:creator' in element.keys():
                 del element['dc:creator']
        return json_holder
```

In [11]:

```
def transform_to_ethical_correct_data(df):
    """
    This functions retrives a a dataframe and invokes the function remove_unethical_entries
    Subsequently, the output is an ethically correct DataFrame which is returned to the inv
    """
    df_index = df.index
    len_df_index = len(df_index)
    i = 0

    while i < len_df_index:
        print("Progress: "+str(i+1) +"/"+str(len_df_index)+" Index position: "+str(df_index
        df['affiliation'][df_index[i]] = remove_unethical_entries(df['affiliation'][df_inde
        df['coredata'][df_index[i]] = remove_unethical_entries(df['coredata'][df_index[i]])
        i = i + 1
    return df
```

Thusly, the already fetched SCOPUS API information is read from the disk for further processing.

In [12]:

```
df_current_extra_info = pd.DataFrame()
bool_show_df = False
try:
    df_current_extra_info = pd.read_pickle('extra_info_CS5099.pkl')
    bool_show_df = True
except:
    print("The DataFrame is empty")
    bool_show_df = True
```

If fetched information is available, it will be shown below. The data is read from the disk.

In [13]:

```
if bool_show_df == True:
    print(df_current_extra_info)
else:
    print("There is no fetched information available.")
```

```
                                             affiliation  \
0      [{'affiliation-country': 'United States'}, {'a...
1      [{'affiliation-country': 'United States'}, {'a...
2      [{'affiliation-country': 'United States'}, {'a...
3      [{'affiliation-country': 'United States'}, {'a...
4      [{'affiliation-country': 'United States'}, {'a...
...                                                  ...
74297           {'affiliation-country': 'United States'}
74298  [{'affiliation-country': 'United States'}, {'a...
74299           {'affiliation-country': 'United States'}
74300  [{'affiliation-country': 'Turkey'}, {'affiliat...
74301                                               None

                                               coredata
0      {'srctype': 'j', 'eid': '2-s2.0-85083266658', ...
1      {'srctype': 'j', 'prism:issueIdentifier': '7',...
2      {'srctype': 'j', 'prism:issueIdentifier': '8',...
3      {'srctype': 'j', 'prism:issueIdentifier': '9',...
4      {'srctype': 'j', 'prism:issueIdentifier': '11'...
...                                                  ...
74297  {'srctype': 'j', 'eid': '2-s2.0-85092678139', ...
74298  {'srctype': 'j', 'eid': '2-s2.0-85087468210', ...
74299  {'srctype': 'j', 'eid': '2-s2.0-85092677974', ...
74300  {'srctype': 'j', 'prism:issueIdentifier': '4',...
74301                                               None

[74302 rows x 2 columns]
```

In [14]:

```
def contains_only_None(dic):
    """
    This function inspects an dictionary and returns True if it solely contains None values
    """
    return len(dic) == sum(value == None for value in dic.values())
```

In [15]:

```
def transpose_and_structure(df):
    """
    This function receives a DataFrame and returns it after tranposing.
    """
    df = df.T
    if 'affiliation' not in df.columns:
        df['affiliation'] = None
    if 'coredata' not in df.columns:
        df['coredata'] = None
    return df
```

In [16]:

```
def append_fetched_data_to_df(df_current_extra_info, dic):
    """
    This function appends or inserts newly fetched data to the DataFrame containing scopus
    Moreover, this function is replacing None values with retrieved data.
    df_current_extra_info holds the current entries read from the disk
    df_newly_fetched_transposed holds the newly fetched information in form ready to be ins
    """

    #checking if the dictionary contains value to append or insert to the existing informat
    if contains_only_None(dic):
        #If a dictionary contains solely None values, they will be prepared for appending/i
        placeholder_entries = pd.DataFrame(np.empty((len(dict_new_extra_info),2),dtype=obje
        df_newly_fetched_transposed = placeholder_entries
        print(df_newly_fetched_transposed)
    else:
        #Prior appending, the dictionary is transformed to a DataFrame
        df_newly_fetched = pd.DataFrame(dic)
        #For readability, the DataFrame is transposed
        df_newly_fetched_transposed = transpose_and_structure(df_newly_fetched)
        #The data is modified accordingly to the ethical guidelines
        df_newly_fetched_transposed = transform_to_ethical_correct_data(df_newly_fetched_tr
        print(df_newly_fetched_transposed)

    #Insert newly fetched rows which were previously not successful inserted/appended
    for index, row in df_newly_fetched_transposed.iterrows():
        #Insert to current extra info DataFrame because the row is existent
        if index in df_current_extra_info.index and row is not None:
            df_current_extra_info.loc[index] = row
        #Append to current extra info DataFrame because the row is new
        if index not in df_current_extra_info.index:
            df_current_extra_info = df_current_extra_info.append(row, ignore_index=True)

    #Returning the DataFrame with newly fetched entries
    return df_current_extra_info
```

Thusly, the function handles the storing of newly fetched information to the disk.

In [17]:

```
def store_df_columns(df):
    """
    This function stores a DataFrame to a local file on the disk.
    """
    df.to_pickle('extra_info_CS5099.pkl')
```

The length of the DataFrame containing the current information is assigned to a variable to be used for further processing. Therefore, the length will be used within a while loop as a starting index.

In [18]:

```
len_df_current_extra_info = len(df_current_extra_info)
len_df_current_extra_info
```

Out[18]:

```
74302
```

Subsequently, the SCOPUS API is fetched and stored within a dictionary. Besides, the print function is used to show the state of the process by displaying the latest fetched information.

In [19]:

```
dict_new_extra_info = dict()
len_dois = len(doi_counted)

def trigger_fetching():
    threshold = 0
    i = len_df_current_extra_info
    while i < len_dois:
        dict_new_extra_info[i] = fetch_scopus_api(client, doi_counted.index[i])
        print("Fetching index position: " + str(i) + " -> " +  doi_counted.index[i])
        i = i + 1
        threshold = threshold + 1
        if threshold > 99:
            df_combined_extra_info = append_fetched_data_to_df(df_current_extra_info, dict_
            store_df_columns(df_combined_extra_info)
            threshold = 0
            print("New batch saved.")
```

Thusly, the SCOPUS API fething process is triggered.

In [20]:

```
trigger_fetching()
```

The existing and newly fetched information is combined into one DataFrame and shown.

In [21]:

```
df_combined_extra_info = append_fetched_data_to_df(df_current_extra_info, dict_new_extra_in
df_combined_extra_info
```

```
Empty DataFrame
Columns: [affiliation, coredata]
Index: []
```

Out[21]:

	affiliation	coredata
0	[{'affiliation-country': 'United States'}, {'a...	{'srctype': 'j', 'eid': '2-s2.0-85083266658', ...
1	[{'affiliation-country': 'United States'}, {'a...	{'srctype': 'j', 'prism:issueIdentifier': '7',...
2	[{'affiliation-country': 'United States'}, {'a...	{'srctype': 'j', 'prism:issueIdentifier': '8',...
3	[{'affiliation-country': 'United States'}, {'a...	{'srctype': 'j', 'prism:issueIdentifier': '9',...
4	[{'affiliation-country': 'United States'}, {'a...	{'srctype': 'j', 'prism:issueIdentifier': '11'...
...	...	...
74297	{'affiliation-country': 'United States'}	{'srctype': 'j', 'eid': '2-s2.0-85092678139', ...
74298	[{'affiliation-country': 'United States'}, {'a...	{'srctype': 'j', 'eid': '2-s2.0-85087468210', ...
74299	{'affiliation-country': 'United States'}	{'srctype': 'j', 'eid': '2-s2.0-85092677974', ...
74300	[{'affiliation-country': 'Turkey'}, {'affiliat...	{'srctype': 'j', 'prism:issueIdentifier': '4',...
74301	None	None

74302 rows × 2 columns

Verifying that the returned None values are due to non-existent data and not to an invalid API-Key or later available information. Thusly, the function is used to fetched specific entries which could not be fetched previously.

In [22]:

```
def enrich_data():
    """
    This function fetches again the scopus API and solely asks for information which previo
    """
    #Add a new column to the DataFrame containg the DOI's which are used to fetch the API
    ser_doi = pd.Series(doi_counted.index[:len_data])
    df_current_extra_info_checker = df_combined_extra_info

    #Fetching solely None entries
    len_df_current_extra_info_checker = len(df_current_extra_info_checker)
    dict_new_extra_info_checker = dict()
    i = 0
    #Specify range to fetch. Otherwise comment out the next two lines.
    #i = 70000
    #len_df_current_extra_info_checker = 5000
    while i < len_df_current_extra_info_checker:
        #Fetch entries which miss informtion for affiliation and/or coredata
        if df_current_extra_info_checker['affiliation'][i] == None or df_current_extra_info
            dict_new_extra_info_checker[i] = fetch_scopus_api(client, ser_doi[i])
            print("Fetched again index position: " + str(i) + " -> " +  ser_doi[i])
        i = i + 1

    #Check if at least one of the fetched values is not None, otherwise the process is fini
    if contains_only_None(dict_new_extra_info_checker):
        print("The scopus API did not returned new information for existing None values.")
    else:
        #There is new information to insert to the existing DataFrame
        df_combined_extra_info_fetched_again  = append_fetched_data_to_df(df_current_extra_
        store_df_columns(df_combined_extra_info_fetched_again)
```

Print out the number of existent doi's and the length of the DataFrame holding the latest entries.

In [23]:

```
len_dois = len(doi_counted)
len_dois
```

Out[23]:

```
74302
```

In [24]:

```
len_data = len(df_combined_extra_info)
len_data
```

Out[24]:

```
74302
```

The next cell invokes the enrich_data() function when the SCOPUS API was fetched once with all doi's. Consequently, the DataFrame holding the latest fetched information must have the same length as the number of existent doi's.

In [25]:

```
if len_dois == len_data:
    enrich_data()
else:
    print("There are entries which are not fetched yet from the scopus API.")
```

```
Fetched again index position: 13 -> 10.1002/0471142700.nc0430s27
Fetched again index position: 15 -> 10.1002/0471142735.im0700s95
Fetched again index position: 16 -> 10.1002/0471142735.im0700s97
Fetched again index position: 43 -> 10.1002/2211-5463.13058
Fetched again index position: 52 -> 10.1002/acr.24487
Fetched again index position: 53 -> 10.1002/acr2.11148
Fetched again index position: 54 -> 10.1002/acr2.11164
Fetched again index position: 55 -> 10.1002/acr2.11174
Fetched again index position: 56 -> 10.1002/acr2.11207
Fetched again index position: 68 -> 10.1002/adtp.202000034
Fetched again index position: 69 -> 10.1002/adtp.202000129
Fetched again index position: 85 -> 10.1002/aepp.13096
Fetched again index position: 86 -> 10.1002/aepp.13100
Fetched again index position: 88 -> 10.1002/aepp.13128
Fetched again index position: 99 -> 10.1002/aisy.202000070
Fetched again index position: 108 -> 10.1002/ajh.21744
Fetched again index position: 142 -> 10.1002/ame2.12036
Fetched again index position: 143 -> 10.1002/ame2.12047
Fetched again index position: 144 -> 10.1002/ame2.12083
```