# Web Crawling and Topic Modelling

---

Develop a model which will make it possible to identify specific subject matter being discussed/present on web pages by using a combination of web crawling and natural language processing.

---

# 1. Setup environment

## 1.1 Install relevant libraries

Major packages needs to be installed as below:

- **Step 1: Install Python 3.9 environment**
- **Step 2: Uncomment the code shown blow**
- **Step 3: Run the code once to install packages**

In [2]:
```
# !pip install nltk
# !pip install spacy
# !pip install scikit-learn
# !pip install gensim
# !pip install BeautifulSoup
# !pip install textrazor
```

## 1.2 Import relevant libraries

Major packages used are as follows:

- **Natural Language Processing (NLP):** `NLTK`, `spaCy`
- **Topic modelling:** `gensim`, `sklearn`
- **Web scraping:** `requests`, `BeautifulSoup`, `urllib3`

In [1]:
```
# NLTK package for retreiving stopwords and tokenizer
import nltk
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# SpaCy package for retreiving stopwords and lemmas
import spacy
from spacy.lang.en.stop_words import STOP_WORDS
from spacy.lang.en import English

# String for punctuations
import string

# RegEx pattern matching library
import re

# Topic Modelling packages from sklearn: NMF, LDA, SVD
```

```
from sklearn.decomposition import NMF, LatentDirichletAllocation, TruncatedSVD
from sklearn.feature_extraction.text import CountVectorizer

# Use gensim paackage to perform text processing, topic modelling and evaluation
import gensim
import gensim.corpora as corpora
from gensim.utils import simple_preprocess
from gensim.models import CoherenceModel, ldamodel, tfidfmodel, Nmf
from gensim.models.phrases import Phrases, ENGLISH_CONNECTOR_WORDS

from time import time
from tqdm import tqdm
from glob import glob

import os
import json
import math
import numpy as np
import pandas as pd
from collections import Counter
import requests
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util import Retry
from bs4 import BeautifulSoup

import urllib3
urllib3.disable_warnings()

import numpy as np
np.set_printoptions(threshold=np.inf)
```

## 2. Web Scraping

Perform web scraping on specified URL and retreive the relevant text data such as webpage title, headers, paragraphs and emphasised keywords using `request` and `BeautifulSoup` package.

In [45]:
```
def get_response_by_url(url):
    """
    Retreive html web content from a given URL.
    Request is set to timeout after 5s and no TLS cert verification.
    When response received successfully, status indicated as 200, else unsucessful.
    When URL is inaccessible (error), status indcated as -1 and response as None.

    Returns a tuple of URL name, response status and response content.
    """
    #print("retreiving:", url)
    try:
        resp = requests.get("https://" + url, params=[("q", f"{url}"), ("u", "DEFAULT AG
        if resp and resp.text:
            # https://www.restapitutorial.com/httpstatuscodes.html - 200 is successful s

            # print(f"url:{url}, resp.status_code: {resp.status_code}")
            return url, resp.status_code, resp.text
    except Exception as e:
        pass

    status = -1
    response_content = None
    return url, status, response_content

def extract_content_by_tag(content, tags_list:list):
    """
    Parse the response content retreived from the URL.
```

```
    The expected tags are the following:
    1. Webpage title: ['title']
    2. Webpage headers: ['h1', 'h2', 'h3', 'h4', 'h5']
    3. Webpage emphasised keywords: ['b', 'strong']
    4. Webpage plain texts: ['p', 'li', 'span']
    Other tags will be ignored to avoid irrelavant text data.

    Returns the parsed content as string.
    """

    parsed_content = BeautifulSoup(content, 'html.parser')
    parsed_content = parsed_content.find_all(tags_list)
    parsed_content = ' '.join([tag.get_text().strip() for tag in parsed_content])
    return parsed_content
```

# 3. Text and Token Processing

## 3.1 Text Processing

Single-stage text processing combining **(i) tokenization**, **(ii) lemmatization** and **(iii) removal of stopwords, numbers and special characters**.

- Stopwords are collected from two libraries, `NLTK` and `spaCy` for comprehensiveness.
- Special characters contain punctuations, numbers and manually inputed characters to cover edge cases.
- Lemmatization is done by utilising `spaCy`'s text processing capabilities rather than `NLTK` for better accuracy.

In [3]:

```
##### Lemmetisation #####
# Load spacy dictionary `small English`
# Used to parse text and perform lemmatization
nlp = spacy.load("en_core_web_sm")

##### Get stopwords from NLTK and SpaCy corpus #####
nltk.download('stopwords')
stopwords1 = nltk.corpus.stopwords.words("english")
stopwords2 = list(STOP_WORDS)
stop_words = list(set(stopwords1 + stopwords2))

##### Get punctuations and special characters #####
punctuations = string.punctuation
additional_puncs =  "â©â£–“”’…" # add edge cases here
punctuations = punctuations + additional_puncs

##### RegEx pattern to remove specified characters #####
pattern = re.compile("[" + punctuations + "0-9" +  "]+")
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\haipewang5\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [4]:

```
##### Lemmatization, tokenization and stopword removal #####
def text_preprocess(text):
    """
    Use spaCy's English parser to process text data to retreive text lemmas,
    followed by tokenization and removal of stopwords and special characters.

    --- Lemmatization ---
    SpaCy will perform Part-of-Speech (POS) tagging behind the scene.
```

```
    This allows us to retreive the word lemmas based on POS and context.
    From orevious testing, it appears that spaCy performs better at lemmatization compar

    --- Tokenization & Cleaning ---
    Lemmas will be lower-cased and stripped of white spaces.
    Any special characters, numbers and punctuations will be removed.
    Any words contained in the list of stopwords will be removed.

    Returns cleaned tokenized texts.
    """

    text = text.replace("_", " ")
    parser = English()
    parsed_doc = parser(nlp(text))
    cleaned = [word.lemma_.lower().strip() for word in parsed_doc]
    cleaned = [re.sub(r"[^\w\s]", " ", re.sub(pattern, " ", word).strip()) for word in c
                                              if word not in stop_words and word not
    cleaned = [word for word in cleaned if len(word) > 1 and len(word) < 15]
    return cleaned
```

### 3.2 Token Processing

- filter out the URL if the sum number of tokens below specified threshold.
- Wegiht the tokens by duplicating for several times to improve the improve the significance of important tokens

In [15]:
```
def filter_on_tokenlen(df, threshold=0):
    # Filters out observations with token length below the threshold
    criteria1 = df["title_token"].apply(len) == threshold
    criteria2 = df["header_token"].apply(len) == threshold
    criteria3 = df["body_token"].apply(len) == threshold
    criteria4 = df["emph_token"].apply(len) == threshold

    valid_df = df[~ (criteria1 & criteria2 & criteria3 & criteria4)].copy()
    empty_df = df[criteria1 & criteria2 & criteria3 & criteria4].copy()
    return valid_df, empty_df
```

In [16]:
```
def weighted_token(df, weight=2):
    # duplicate the important tokens to with a specific weights
    df["title_token"] = df["title_token"].apply(lambda x: x*weight)
    df["header_token"] = df["header_token"].apply(lambda x: x*weight)
    df["emph_token"] = df["emph_token"].apply(lambda x: x*weight)
    return df
```

## 4. Topic Modelling

First bigram tokens are generated and added to the token list before being passed to the model. After building the model and generating the topics, coherence score per topics are generated for topic evaluation.

Three approaches will be taken for the topic modelling algorithms as shown below.

**Latent Dirichlet Allocation (LDA)**

A generative probabilistic model for identifying hidden topics or themes within a large corpus of text documents. The basic idea behind LDA is that each document is composed of a mixture of topics, and each topic is a probability distribution over a fixed vocabulary of words.

**Non-negative Matrix Factorization (NMF)**

A non-probabilistic, linear-algebraic machine learning algorithm for unsupervised clustering, feature extraction, and dimensionality reduction to decompose a document-term matrix into a product of two non-negative matrices representing topics and their associated word distributions.

In [5]:
```
def generate_bigram(tokens_list:list, min_count=1, threshold=0.1,
                    lower=2, upper=0.9, keep=5000):
    """
    Models and generates the following:

    --- Bigram Tokens --
    Bigrams are word-pairs that creates a different contextual meaning combined.
    Using `gensim` bigram phraser, bigram word-pairs are generated and added
    to the list of tokens.

    --- Term-Frequency Matrix ---
    Dictionary of Word IDs against the respective word frequency is generated.
    This matrix is used to train the model. Extreme cases will be removed to
    improve efficiency and reduce redundant or extreme words by filtering for
    the minimum and maximum word count, and the size of matrix.

    --- Corpus ---
    Bag-of-words (BOW) is generated based on the tokens.

    Returns the bigram tokens, tf matrix and corpus
    """
    # Build the bigram phrase model
    bigram = Phrases(
        tokens_list,
        min_count=min_count,
        threshold=threshold,
        connector_words=ENGLISH_CONNECTOR_WORDS
        )

    # Generate a list containing single-word and bigram tokens
    bigram_tokens = bigram[tokens_list]

    # Build the id-to-word matrix dictionary
    bigram_id2word = corpora.Dictionary(bigram_tokens)

    # Filter out the extreme cases to improve efficiency and accuracy
    #bigram_id2word.filter_extremes(
    #    no_below=lower, no_above=upper, keep_n=keep
    #)

    # Build the bag-of-word corpus form the tokens
    bigram_corpus = [bigram_id2word.doc2bow(word) for word in bigram_tokens]

    return bigram_tokens, bigram_id2word, bigram_corpus
```

In [13]:
```
def generate_topics(corpus, id2word, algorithm="LDA", k=6, seed=100):
    """
    Build the model based on the specified algorithm

    --- Latent Dirichlet Allocation ---
    Runs LDA algorithm based on word distribution in the topics and
    topic distribution in the documents. LDA is a probabilistic model
    and it is the most commonly used algorithm.

    --- Non-negative Matrix Factorization ---
    Runs NMF algorithm and breaksdown DTM into non-negative topic-word matrix
    and topic-document matrix. Unlike LDA, NMF is non-probabilistic and
```

```python
    is based on linear algebra to retreive topics.

    --- Term Frequency - Inverse Document Frequency ---
    Calculates TF-IDF of the words in the set of documents. TF-IDF represents
    relevance of a word in the corpus.

    Returns the model and the topics generated.
    """
    if algorithm == "LDA":
        model = ldamodel.LdaModel(
            corpus=corpus,
            id2word=id2word,
            num_topics=k,
            random_state=seed,
            update_every=1,
            chunksize=200,
            iterations=100,
            passes=10,
            alpha='auto',
            per_word_topics=True
        )
    elif algorithm == "NMF":
        model = Nmf(
            corpus=corpus,
            id2word=id2word,
            num_topics=k,
            random_state=seed,
            chunksize=200,
            passes=10,
            kappa=.5,
        )
    else:
        print("Model not available. Input: 'LDA', 'NMF'")
        return

    # Store the top-20 words in each topics and the overall probability score
    probas={}
    counts={}
    for i in range(len(model.top_topics(corpus))):
        for word in model.top_topics(corpus)[i][0]:
            prob = word[0]
            term = word[1]
            probas[term] = probas.get(term, 0) + prob
            counts[term] = counts.get(term, 0) + 1

    # Get the average probability per topics
    topics={}
    for word, prob in probas.items():
        topics[word] = prob / counts[word]

    return model, topics
```

In [14]:

```python
def evaluate_coherence(model, tokens, id2word, method='c_v'):
    """
    Build coherence model to evaluate how 'coherent' the keywords are
    within the topics. Higher coherence shows well connected words.

    Returns a list of coherence score per topic.
    """
    coherence_model = CoherenceModel(
        model=model,
        texts=tokens,
        dictionary=id2word,
        coherence=method
    )
```

```
    return coherence_model.get_coherence()
```

In [17]:
```
def sort_weigh_topics(topics, coherence):
    # Sort topics by adjusted probability score in descending order
    return {k: (v / coherence) for k, v in sorted(topics.items(), key=lambda x: x[1], re


def weighted_average_topics(topics_list):
    # Get the weighted average of topics probabilities across token columns
    topics = {}
    probas={}
    counts={}
    for topic in topics_list:
        for term, prob in topic.items():
            probas[term] = probas.get(term, 0) + prob
            counts[term] = counts.get(term, 0) + 1

    # Get the average probability per topics
    topics={}
    for word, prob in probas.items():
        topics[word] = prob / counts[word]

    return topics
```

# 5. Main Pipeline

## 5.1 Steps for main function

- Stpe 1: Load original input datasets for web crawling and topic modeling (csv file)
- Stpe 2: Load previous invalid URL
- Stpe 3: Load records of visited URL
- Stpe 4: Web scraping and retreive response for each URL
- Stpe 5: Text and Token processing to parse raw HTML content
- Stpe 6: Topic Modelling
- Stpe 7: Results Tagging
- Stpe 8: Save and output results

In [ ]:
```
def main(filepath, model="LDA", mode="by_page_url"):
    """
    The main function to control the whole procedure of web crawling and topic modeling.

    Paramerters:
    filepath: file path for original input file for web crawling and topic modeling
    model: topic model, can take LDA, NMF.
    mode: text processing mode, can take by_page_url or by_panelist_id to extract tokens

    Return: generates topics for specific model and mode on the input file.
    """
    start_time = time()


    # 1. Load original input datasets for web crawling and topic modeling
    df = pd.read_csv(filepath, encoding="utf-8")
    df['PageUrl'] = df['PageUrl'].astype(str)


    # 2. Load previous invalid URL
    # Declate the output file path
```

```
    output_file_path = "output_" + model + "_" + mode
    if not os.path.exists(output_file_path):
        os.makedirs(output_file_path)

    invalid_filename_path = os.path.join(output_file_path, "invalid_urls.txt")
    if os.path.exists(invalid_filename_path):
        with open(invalid_filename_path, 'r', encoding="utf-8") as f:
            invalid_urls = f.read().splitlines()
    else:
        invalid_urls = []

    # Filter dataframe from already known invalid URLs
    df_filter = df[~ df['PageUrl'].isin(invalid_urls)].copy()


    # 3. Load records of visited URL
    if mode == "by_page_url":
        visited_filename_path = os.path.join(output_file_path, "visited_urls.csv")
        if os.path.exists(visited_filename_path):
            visited_urls_df = pd.read_csv(visited_filename_path)

            # Fitler out URLs that has been visited and modelled previously
            df_filter = df_filter[~df_filter['PageUrl'].isin(visited_urls_df['PageUrl'].

            # Get the previously visited and modelled URL for concat later
            df_visited = visited_urls_df[visited_urls_df['PageUrl'].isin(df_filter['Page

        else:
            columns = ["PageUrl", "weighted_topics", "coherence_model"]
            visited_urls_df = pd.DataFrame(columns=columns)


    # 4. Web scraping and retreive response for each URL
    df_filter['response'] = df_filter['PageUrl'].apply(lambda x: get_response_by_url(x))
    df_filter['response_status'] = df_filter['response'].apply(lambda x: x[1])
    df_filter['response_text'] = df_filter['response'].apply(lambda x: x[2])

    # Identify new invalid URL, bad response status, exception or response emtpy
    invalid_df = df_filter[(df_filter['response_status'] != 200) | (df_filter['response_

    # Add the newly found invalid URLs due to no response into the main URL list
    new_invalid_urls = invalid_df['PageUrl'].unique().tolist()
    invalid_urls = list(set(invalid_urls + new_invalid_urls))

    # Filter dataframe from new invalid URLs
    df_filter = df_filter[~ df_filter['PageUrl'].isin(new_invalid_urls)]


    # 5. Text and Token processing to parse raw HTML content
    tags = {
        'title': ['title'],
        'header': ['h1', 'h2', 'h3', 'h4', 'h5'],
        'emph': ['green', 'red', 'b', 'strong'],
        'body': ['p', 'li', 'span']
    }

    for key, tag_list in tags.items():
        df_filter[key] = df_filter['response_text'].apply(lambda x: extract_content_by_t

    # Text Preprocessing to extract tokens
    for key in tags.keys():
        df_filter[key + "_token"] = df_filter[key].apply(text_preprocess)

    # Handeling by_panelist_id mode by merging tokens of same panelist_id into one
    if mode == "by_panelist_id":
        # merge extracted tokens into the same panelist_id for further modeling
```

```
        df_filter = df_filter.sort_values(by=['panelist_id'])
        for idx in list(df_filter['panelist_id'].unique()):
            rows = df_filter.loc[df_filter['panelist_id'] == idx].index
            if len(rows) > 1:
                new_df = df_filter.loc[rows[0]].copy()
                temp_all_df = df_filter.loc[rows]
                df_filter = df_filter.drop(rows)
                new_df['response_text'], new_df['title_token'], new_df['header_token'],
                for index, one_df in temp_all_df.iterrows():
                    new_df['response_text'] += one_df['response_text']
                    new_df['title_token'] += list(one_df['title_token'])
                    new_df['header_token'] += list(one_df['header_token'])
                    new_df['body_token'] += list(one_df['body_token'])
                    new_df['emph_token'] += list(one_df['emph_token'])
                df_filter.loc[len(df_filter.index)] = new_df


    # Token process to weighte the important tokens
    df_filter_weighted = weighted_token(df_filter)

    # Remove row if all the token are empty
    df_valid, df_empty = filter_on_tokenlen(df_filter_weighted)

    # Add the invalid URLs due to lack of response into the main URL list
    insufficient_response_urls = df_empty['PageUrl'].unique().tolist()
    invalid_urls = list(set(invalid_urls + insufficient_response_urls))

    # Save the invalid URL list into text file for future usage to avoid executing for p
    with open(invalid_filename_path, 'w', encoding="utf-8") as f:
        f.write('\n'.join(invalid_urls))


    # 6. Topic Modelling
    if not df_valid.empty:
        # a. Bigram modelling
        df_valid["bigram_model"] = df_valid.apply(
            lambda col: generate_bigram([col['title_token'], col['header_token'], col['b

        # b. Topic model algorithm
        df_valid["topic_model"] = df_valid["bigram_model"].apply(lambda x: generate_topi

        # c. Coherence model
        df_valid["coherence_model"] = df_valid.apply(lambda col: evaluate_coherence(col[
                    col["bigram_model"][1]), axis=1)

        # d. Sort topics by score
        df_valid["weighted_topics"] = df_valid.apply(
            lambda col: weighted_average_topics([sort_weigh_topics(col["topic_model"][1]
    else:
        # if the URL has been visited before, diectly use the previous topic results
        # it is only suitable for "by_page_url" mode, because "by_panelist_url" mode mer
        if mode == "by_page_url":
            df_valid = df_valid.merge(visited_urls_df, on="PageUrl", how="left")


    # 7. Results Tagging
    if mode == "by_page_url":
        columns = ["PageUrl", "weighted_topics", "coherence_model"]
        df_tagging = df_valid.loc[:,columns].copy()

        # concate previous visited files if exist, otherwise create a new file
        if not os.path.exists(visited_filename_path):
            df_tagging.to_csv(visited_filename_path, index=False, encoding="utf-8-sig")
        else:
            visited_urls_df = pd.concat([visited_urls_df, df_tagging], ignore_index=True
            visited_urls_df.to_csv(visited_filename_path, index=False, encoding="utf-8-s
```

```
            df_tagging = pd.concat([df_tagging, df_visited], ignore_index=True)

        # remove copulation rows by URLs before merging
        df_tagging = df_tagging.drop_duplicates("PageUrl")
    elif mode == "by_panelist_id":
        columns = ["panelist_id", "weighted_topics", "coherence_model"]
        df_tagging = df_valid.loc[:,columns].copy()


    # 8. Save and output results
    # Tag the original dataset with the results
    if mode == "by_page_url":
        df_output = df.merge(df_tagging, on="PageUrl", how="left")
    elif mode == "by_panelist_id":
        df_output = df.merge(df_tagging, on="panelist_id", how="left")

    df_output["weighted_topics"] = df_output["weighted_topics"].fillna("N.A.")
    df_output["coherence_model"] = df_output["coherence_model"].fillna("N.A.")

    # Save the output file
    output_filename = re.findall(r"\\(.+\.csv)", filepath.replace(".csv", "_output.csv")
    output_path = os.path.join(output_file_path, output_filename)
    df_output.to_csv(output_path, index=False, encoding="utf-8-sig")

    print("Run time:", round(time() - start_time, 4), "s")
    return
```

## 5.2 Execute main function

In [ ]:
```
# Specifying folder directory and retreive filepaths to run the script
folder = "sample_URL_95CI_5E"
filepaths = glob(folder + "\*.csv")
print("No. of files:", len(filepaths))
```

In [ ]:
```
# Run the script sample filepaths with LDA model and by_page_url mode
for file in filepaths:
    print("Executing model on", file)
    main(file, model="LDA", mode="by_page_url")
```

In [ ]:
```
# Run the script sample filepaths with LDA model and by_panelist_id mode
for file in filepaths:
    print("Executing model on", file)
    main(file, model="LDA", mode="by_panelist_id")
```

In [ ]:
```
# Run the script sample filepaths with NMF model and by_page_url mode
for file in filepaths:
    print("Executing model on", file)
    main(file, model="NMF", mode="by_page_url")
```

In [ ]:
```
# Run the script sample filepaths with NMF model and by_panelist_id mode
for file in filepaths:
    print("Executing model on", file)
    main(file, model="NMF", mode="by_panelist_id")
```

---

# 6. Additional: Existing Commercial Tool

- Installation and registration to use the existing commercial tool `textrazor`

- Calling `textrazor` API and directly model topic from URL

---

## 6.1 Setup TextRazor

- Sign up and create your account on textrazor website
- Validate your email and receive your API key
- Install textrazor `pip install textrazor`
- *Note: textrazor is not open-source, there is a limit of 500 API calls per API key, do not spam*

In [2]:
```
# Import textrazor library
import textrazor

# Input API key received during registration, please don't spam with my API key
textrazor.api_key = "5c2b9b14e0e00b418ea577c64403ef39e6df8717f40a74bb1fb6a6bc"
```

In [3]:
```
# Call textrazor client and retreive topic and entity analysis using `analyze_url()`
client = textrazor.TextRazor(extractors=["entities", "topics"])
```

## 6.2 Run `textrazor` directly on the given URL

In [4]:
```
def retreive_topics(response, threshold):
    # Create list to store the topics and the cut-off threshold score
    topics = {}
    confidence_threshold = threshold

    # Retreive topic labels and score from the model
    for topic in response.topics():
        if topic.score > confidence_threshold:
            topics[topic.label] = topics.get(topic.label, topic.score)

        #print(topic.label, topic.score)

    # Sort topics based on the confidence score
    topics = {k: v for k, v in sorted(topics.items(), key=lambda x: x[1], reverse=True)}

    return topics
```

In [5]:
```
def retreive_entities(response, threshold):
    # Create dictionary to store the topics, entities and the cut-off threshold score
    entities = {}
    relevance_threshold = threshold

    # Retreive entity ids and relevance score from the model
    for entity in response.entities():
        if (entity.relevance_score > relevance_threshold):
            entities[entity.id] = entities.get(entity.id,
                                                (entity.relevance_score, entity.confidenc

            if entities[entity.id][0] < entity.relevance_score:
                entities[entity.id] = (entity.relevance_score, entity.confidence_score,

        # print(entity.id, entity.relevance_score, entity.confidence_score, entity.freeb

    # Sort the entities based on relevance score
    entities = {k: v for k, v in sorted(entities.items(), key=lambda x: x[1], reverse=Tr

    return entities
```

In [73]:

```
# Run text analysis using textrazor API on the URL
# url = "https://theguardian.com/business/2022/dec/01/big-uk-high-street-bank-slow-react

import random
random.seed(42)

url_response = {}

folder = "sample_URL_95CI_5E"
filepaths = glob(folder + "\*.csv")
print("No. of files:", len(filepaths))

for filepath in filepaths:
    print("Executing model on", filepath)
    # randomly pick two links
    df = pd.read_csv(filepath, encoding="utf-8")
    urls = random.sample(list(df['PageUrl'].astype(str)), k=10)
    for url in urls:
        response = client.analyze_url(url)
        url_response[url] = response
```

```
No. of files: 29
Executing model on sample_URL_95CI_5E\sample_2022-12-01_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-02_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-03_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-04_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-05_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-06_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-07_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-08_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-09_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-10_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-11_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-12_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-13_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-14_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-15_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-16_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-18_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-19_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-20_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-21_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-22_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-23_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-24_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-26_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-27_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-28_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-29_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-30_0.csv
Executing model on sample_URL_95CI_5E\sample_2022-12-31_0.csv
```

In [16]:

```
# Retreive topics and entities based on set threshold

results = pd.DataFrame()
url_list = []
topics_list = []

for url in url_response.keys():
    topics1 = retreive_topics(url_response[url], 0.2)
    entities1 = retreive_entities(url_response[url], 0.5)

    url_list.append(url)
    topics_list.append(dict(list(topics1.items())[:10]))
```

```
results["PageUrl"] = url_list
results["weighted_topics"] = topics_list
results.to_csv("TextRazor_result.csv", index=False)
```