# INFORMATION RETRIEVAL CW

7071CEM COURSEWORK

Suleiman Adamu Yakubu
11796778
yakubus6@uni.coventry.ac.uk

# Introduction

This report is meant to cover the learning outcomes of the 7071CEM (Information Retrieval) module. The module mainly concerns itself with the procedures and techniques used to obtain relevant information from a large collection within an acceptable time and put this information to other uses. Informally, the module concerns itself with the process of building search engines.

For the coursework, we have been set a couple of tasks, namely; creating a vertical search engine, (along with all the tasks this entails such as crawling, scraping, indexing, and querying), and also to create a document clustering system. As such, this report will be split into two main parts to cover each task.

The search engine created for this coursework is what is called a vertical search engine. This means the search engine searches through a very specific collection of information. In this case, the engine returns only articles matching a query. The articles also have to be written by a member of the Coventry University School of Economics, Finance, and Accounting (SEFA). For the purpose of this task, the information returned consists of the following:

1. The title of the article
2. A link to the article text
3. The authors of the article that belong to SEFA
4. The links to each author's pureportal profile page
5. The date of publication

# Task 1: Vertical Search Engine

In building a search engine, a lot of factors come into play; first, the search engine must have a knowledge base against which it can compare user queries. These documents are usually acquired by crawling a web document and scraping the information obtained therein. The information is then passed through some representation function and then saved in an index (sometimes called an inverted index). The user queries are also passed throught the representation function. This enables the queries to be compared against the index (knowledge base) and results returned.

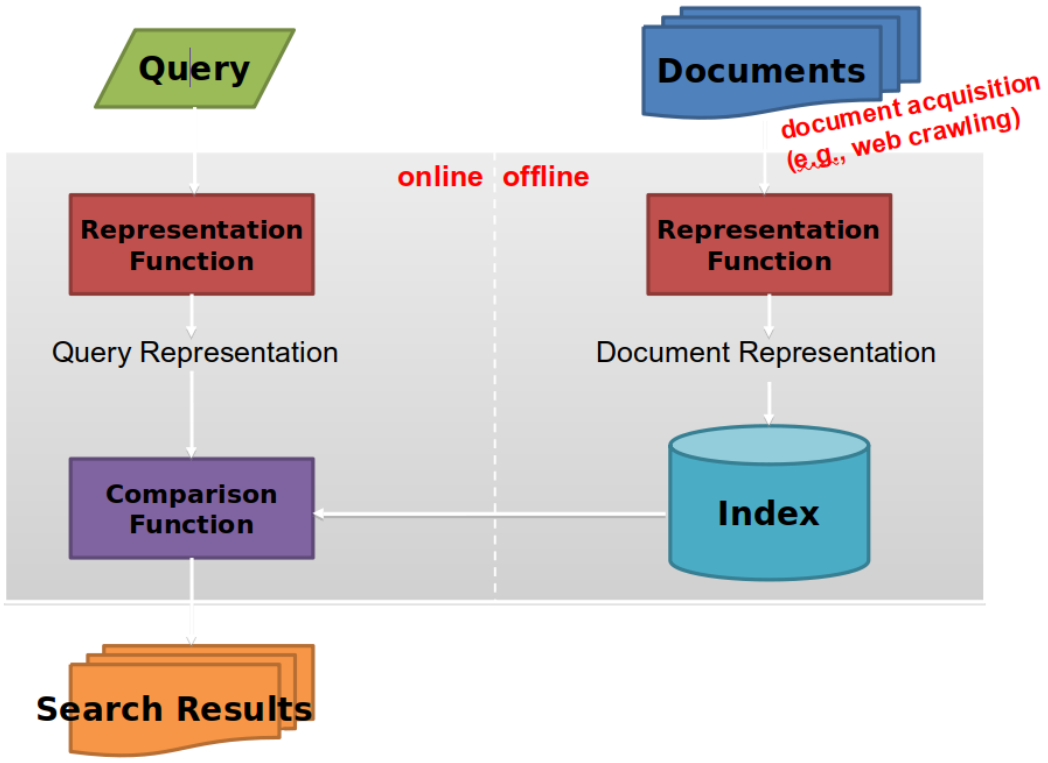


*Figure 1: Architecture of a search engine (Mousavi Seyed, 2022)*

For this report, building the vertical search engine means building the various parts mentioned above and putting them together. The parts of the search engine include the following:

1. A web crawler: the purpose of the crawler is to visit the relevant pages and return with the required information. This information is stored in some way and used to create the inverted index. As such, the web crawler also performs web scraping.
2. An indexer: The indexer's job is to create an inverted index. Ideally, this index will consist of the article text, the author names, the publication date, and other information that is deemed relevant. In this specific case, the titles of the articles are indexed for searching. Before the titles can be indexed however, they have to be passed through a representation function.
3. A query processor: This is a function that preprocesses the query, checks it against the index, and returns the relevant results. In the case of our search engine, we return a list containing all the articles that contain the query in their title.

## 1.1 Crawling

Crawling in this report was achieved using a Python library called Scrapy. This library provides functions and methods to make creating a crawler and scraper efficient and convenient. There are settings that can be used to make sure crawlers created using the Scrapy library are polite (i.e they obey the robots.txt file they find on the servers of the pages they crawl), and to set a time delay before crawling consecutive pages. These are shown below:

```
custom_settings = {
     "FEEDS": {
         "authors.csv": {
             "format": "csv",
             "overwrite": True
         }
     },
     "ROBOTSTXT_OBEY": True,
     "DOWNLOAD_DELAY":  2
 }
```

The above code stipulates where the downloaded information is stored, whether the spider obeys the robots.txt file, and how long it should take between downloading consecutive pages (2 seconds).

For this report, The School of Economics, Finance, and Accounting (SEFA) of Coventry University's publications page (which is located at https://pureportal.coventry.ac.uk/en/organisations/school-of-economics-finance-and-accounting/publications/) was crawled and information about the publications (such as title, publication link, publication date) and authors was retrieved.

The crawler saves this information in a csv file created for this purpose. This csv file is then passed to the representation function to preprocess the titles, and then an index is created using the processed titles.

**NB:** The crawler was set to retrieve only authors that are affiliated with SEFA (i.e. authors with links to their Pureportal profile page). As such, if the author of a publication is not affiliated, or is no longer affiliated with the school for whatever reason, the crawler ignores that author. Another important point to note is that the crawler only returns the title of the publication for indexing, not the actual text of the publication. The code to create and run the crawler can be found in Appendix A.

## 1.2 Preprocessing

To create an index and allow it to be searchable by queries, the titles and queries have to be passed through some representation function. This function is responsible for the following:

- Tokenization: This involves breaking the text into tokens. Each token will be a word in the document or query.
- Stripping punctuations and whitespace from titles or queries: The leading and trailing whitespace, and punctuation is stripped from the text.
- Removing stopwords: commonly used words that do not add any semantic meaning to the text are removed. These are called stopwords. Examples include: the, a, an, etc. The

PorterStemmer from the NLTK library has a library of stopwords that was used for this purpose

- Stemming: stemming involves converting a word to its root form. This process gets rid of plurals, and derivatives of a word. As an example integrate, integration, integrates, and integrated are all stemmed to the word "integrat". One drawback of this method is that it can not handle synonyms. For example, it treats doorway, and portal as different words. To handle this use case, a lemmatizer can be used. Lemmatizers can incorporate the semantics of words they are processing. However, they can be complex to implement and can also slow down performance.

Please refer to Appendix C for the relevant code.

## 1.3 Indexing

For the purposes of searching, there are various data structures that can be used to check queries against. For instance, an incidence matrix could be used. The drawback to this is that an incidence matrix takes up space needlessly as it needs to also store a zero to indicate documents where a particular word is not found. A good way around this is to use an inverted index.

An inverted index is an index of words that shows in which document each word occurs. By doing this, we do away with the need for extra space that the incidence matrix has because we do not need to store information about the documents that do not contain the word (Roy, 2022).

The index contains each word, along with a posting list that contains the document IDs of the documents (titles) in which the word can be found. Finally the index also contains the word's document frequency.

In the process of creating the index, some preprocessing has to be done with the document. The document needs to be broken up into tokens (ie tokenized) and stopwords and punctuation need to be removed. The Python NLTK (Natural Language ToolKit) library was useful in this regard. After the preprocessing, the document is then passed to a function that

creates the index. The index is simply a document that queries can then be searched against. The code can be found in Appendix B.

## 1.4 Searching

Searching starts when a query is passed to the system. The query is passed the same representation function the titles are passed through. This representation function is responsible for the preprocessing as outlined in section 1.2. Each word of the query is then searched against the index. Each word in the query will yield a posting list of documents that contain it. An intersection of all the posting lists returned by a multi-word query will return documents that contain all the words of the query. The union of the posting lists will also return documents that contain either one word in the query or some combination of the query words. Please refer to Appendix D for the relevant code.

## 1.5 Presenting the results

During the course of this coursework, the question of how to present the search results came up quite often and the decision I took was to make this project look as close to a real-life search engine as possible. As such, for searching and presenting the results, a small Flask app was created. This app was hosted locally using Flask's local server.

Flask is a web development library that allows developers to use Python on the backend of their web applications. Flask needs to be installed on a virtual environment and so, virtualenv was used for it. The flask app has two pages; one with a search box for searching and a results page to display the search results.

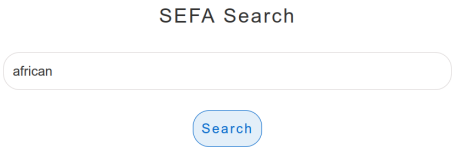


*Figure 2a: The search interface*

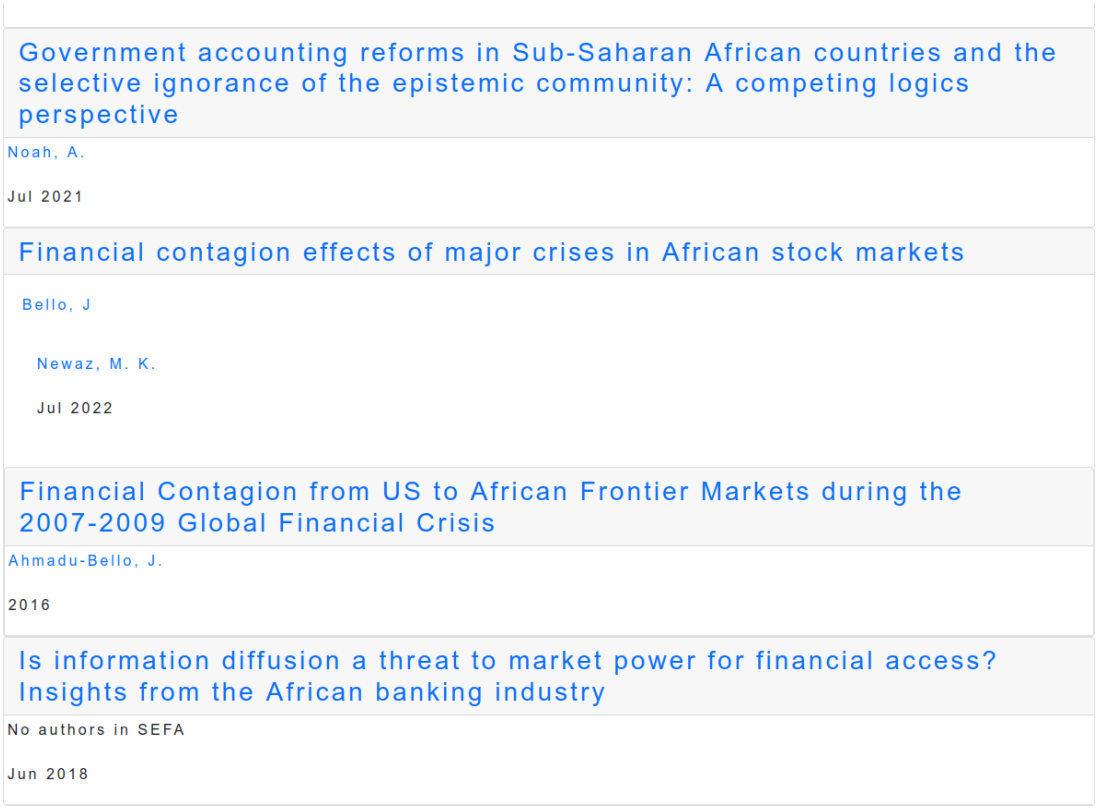


*Figure 2b: The results page*

It is worth noting that the search in the search results, the links to the article text and the authors' pureportal links are integrated into the HTML. As such, clicking the author name, or the article title will lead to the relevant pages.

The web pages were created using HTML, CSS, and a templating language called Jinja that allows Python -like code to be run in the browser. The code for the app can be found in Appendix E.

## 1.6 Sub-conclusion

The search engine works as it should. Even though the user interface is basic, it returns the correct results when queried. However, it doesn't properly rank the results. Ideally, the results will be ranked in a descending order of some metric like cosine similarity with the query. However, due to time constraints, that was not implemented

# Task 2: Document Clustering

Clustering refers to an unsupervised machine learning task in which the machine learns the features of a collection and groups them according to certain criteria. These criteria are often unknown to the programmer. As such, document clustering allows us to discover some unknown structure in our collection by grouping together similar documents (Zhai ChengXiang & Massung Sean, 2016).

For this report, the document clustering algorithm used is the K-Means algorithm which, while relatively simple, can be quite effective. For this task, the steps taken to complete it are as follows:

- Collect a group of documents
- Extract the relevant features from them. In this step, we convert each document into a vector representation of itself
- Find out the optimal cluster number using elbow method
- Cluster the documents using K-Means
- Evaluate the results

This task was carried out using Google Colab. The SciPy, Matplotlib, WordCloud, Pandas, FeedParser, and SciKit Learn libraries were used as well.

## 2.1 K-Means

The K-Means algorithm is a simple algorithm. It operates on the principle that items which are close to each other are more likely to be similar with K being the number of clusters produced. Building on that principle, if documents can be represented as vectors in a multi-dimensional space, then the distance between them can be empirically calculated. Each cluster created by the K-Means algorithm is grouped around a centroid which is the cluster's center point. Each document is placed in the cluster according to which centroid the document is closest to (Harvey M. Deitel & Paul J. Deitel, 2019).

One of the problems of the K-Means clustering algorithm is that the K (number of clusters) has to be known beforehand. This problem can be somewhat mitigated by using the elbow method, which can show the optimal number of clusters for a given collection. This is shown in the plot below:

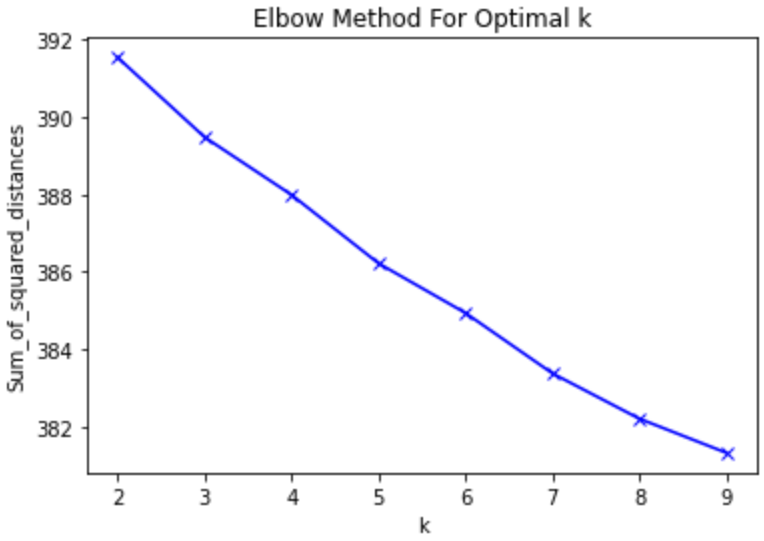


*Figure 3: Elbow method*

The plot only shows a slight perturbation at k=3, and k=5. The sum of squared distances (inertia) of the clusters decreases at a relatively constant rate. This implies that we will get tighter clusters if we increase the number of clusters (k). However, for this project, I will limit the number of clusters to 3. This is because we have only 420 samples and I want each cluster to have a reasonable number of documents. We will evaluate the model later using the silhouette score. The code for the above plot can be found in Appendix H.

## 2.2 Collecting the documents

For this task, the documents were collected from the relevant RSS feeds of the New York Times, Daily Mail, and Buzzfeed. This was done using the FeedParser library. The feeds come in as Python dictionaries. The health and sports documents were collected from the Daily Mail RSS health and sports feeds (which have 150 documents each) while the politics documents

were collected using a combination of the Buzzfeed and New York Times politics feed. Thus, 120 political documents were collected.


```
{'content': [{'base': 'https://www.dailymail.co.uk/sport/index.rss',
    'language': None,
    'type': 'text/plain',
    'value': ''}],
  'credit': 'Getty Images',
  'guidislink': False,
  'href': '',
  'id': 'https://www.dailymail.co.uk/sport/football/article-11079289/Rooneys-DC-United-blows-chance-climb-bottom-Conference-away-Charlotte.html?ns_mchannel=rss&ns_campaign=1490&ito=1490',
  'link': 'https://www.dailymail.co.uk/sport/football/article-11079289/Rooneys-DC-United-blows-chance-climb-bottom-Conference-away-Charlotte.html?ns_mchannel=rss&ns_campaign=1490&ito=1490',
  'links': [{'href': 'https://www.dailymail.co.uk/sport/football/article-11079289/Rooneys-DC-United-blows-chance-climb-bottom-Conference-away-Charlotte.html?ns_mchannel=rss&
ns_campaign=1490&ito=1490',
    'rel': 'alternate',
    'type': 'text/html'},
   {'href': 'https://i.dailymail.co.uk/1s/2022/08/04/08/61056725-0-image-a-9_1659596691005.jpg',
    'length': '10286',
    'rel': 'enclosure',
    'type': 'image/jpeg'}],
  'media_content': [{'type': 'image/jpeg',
    'url': 'https://i.dailymail.co.uk/1s/2022/08/04/08/61056725-0-image-a-9_1659596691005.jpg'}],
  'media_credit': [{'content': 'Getty Images', 'scheme': 'urn:ebu'}],
  'media_thumbnail': [{'height': '115',
    'url': 'https://i.dailymail.co.uk/1s/2022/08/04/08/61056725-0-image-a-9_1659596691005.jpg',
    'width': '154'}],
  'published': 'Thu, 04 Aug 2022 07:04:54 GMT',
  'published_parsed': time.struct_time(tm_year=2022, tm_mon=8, tm_mday=4, tm_hour=7, tm_min=4, tm_sec=54, tm_wday=3, tm_yday=216, tm_isdst=0),
  'summary': 'D.C United suffered a 3-0 defeat to Charlotte that started off in the worst way possible. An early own goal that deflected in off  centerback Steve Birnbaum in the 13th
minute.',
  'summary_detail': {'base': 'https://www.dailymail.co.uk/sport/index.rss',
   'language': None,
   'type': 'text/html',
   'value': 'D.C United suffered a 3-0 defeat to Charlotte that started off in the worst way possible. An early own goal that deflected in off  centerback Steve Birnbaum in the 13th
minute.'},
  'title': "Rooney's DC United blows its chance to climb out of the bottom of the Conference away to Charlotte",
  'title_detail': {'base': 'https://www.dailymail.co.uk/sport/index.rss',
   'language': None,
   'type': 'text/plain',
   'value': "Rooney's DC United blows its chance to climb out of the bottom of the Conference away to Charlotte"}}
```


*Figure 2: Sample of a single entry from an RSS feed*

Each of the feeds is read into a dictionary and then used to create a Pandas dataframe. This dataframe has 3 columns; id, content, and category. The content column holds all the document content. It is this column that is used to create the corpus of documents. The code for this can be found in Appendix F and G.

## 2.2 Preprocessing

To learn the features of the document, irrelevant words need to be eliminated. The document also needs to be broken down to tokens (tokenized), have stopwords removed, and then stemmed. This collection of stemmed tokens is then passed to a vectorizer to create a feature vector. The python code for the preprocessing is shown below:

```
nltk.download("punkt")
```

```
from nltk.tokenize import word_tokenize
```

```
from nltk.stem import PorterStemmer
ps = PorterStemmer()
filtered_docs = []
for doc in documents:
   tokens = word_tokenize(doc)
   tmp = ""
   for w in tokens:
       if w not in sw:
           tmp += ps.stem(w) + " "
   filtered_docs.append(tmp)
print(filtered_docs)
```

## 2.3 Vectorization

Vectorization (also called feature extraction) pertains to converting the documents into vectors. This way, the documents can be processed by the machine learning model. The document is first stripped of stopwords, tokenized, and then converted into a numerical vector that represents the features contained in the text. The code to vectorize is shown below:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(filtered_docs)
```

The output of this operation is a feature vector that can then be compared to other vectors for similarity.

## 2.4 Clustering

A K-Means model is created using the sklearn library. The vectorized document is then passed to the model and trained. The clusters are also displayed. It should be noted that the algorithm does not have a notion of the cluster names. The documents are assigned clusters with numerical names. The figure below shows the cluster labels and how many documents are assigned to each cluster. The code can be found in Appendix 8

```
cluster no. of input documents, in the order they received:
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1 0 0 0
 0 1 1 0 1 0 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 1 1 0 1 1 0 1
 1 1 0 0 0 0 1 0 1 1 0 1 0 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 0 0 1
 0 1 0 0 0 1 0 1 1 0 0 0 0 0 1 1 1 1 0 1 0 0 1 1 0 0 0 1 0 1 0 0 1 0 1 1 0
 0 1 0 1 1 0 1 0 0 1 1 1 1]
Counter({0: 208, 1: 112, 2: 100})
```

*Figure 5: Cluster labels and distribution*

### 2.4.1 Visualizing and testing the model

The cluster labels provide no information as to how the model clustered the collection of documents. However, there are ways to visualize the keywords associated with the centroid of each cluster. This will give a better idea of how the model clustered each document. One way to do this will be to use a word cloud to represent each cluster. The visualizations are shown below:


*Figure 6: Words associated with cluster 0 (Health)*

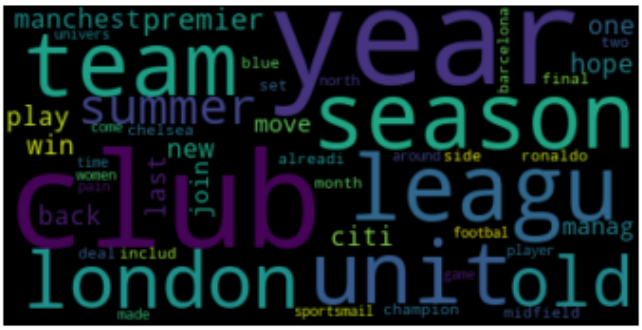


*Figure 7: Words associated with cluster 1 (Sports )*

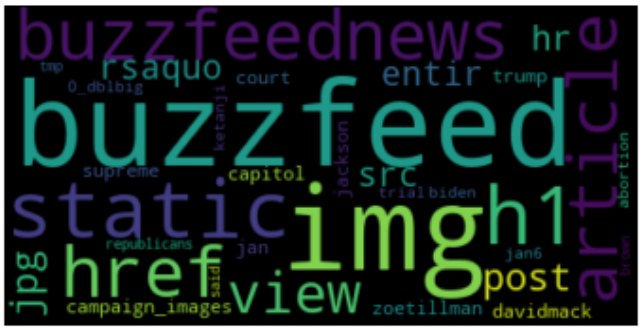


*Figure 8: Words associated with cluster 2 (Politics)*

From the word cloud visualizations above, it can be safely concluded that clusters 0, 1, and 2 are Health, Sports, and Politics respectively.

After the model has been trained and the clustering carried out, the model is tested with various inputs. This can help us ascertain if the model places new documents in the correct clusters.

Also, the silhouette score of the clusters is calculated using sklearn's silhouette_score module. The model has an average silhouette score of 0.022619533173523595. The silhouette score gives an idea of how close each point in a cluster is to other points in the same cluster, and other points in other clusters. The score ranges from between -1 to 1. A score of -1 indicates

that points in a cluster are closer to points in other clusters than the points in the cluster they were assigned (Naik, 2020).

## 2.5 Discussion

The silhouette score above indicates that our clustering model is somewhat in the middle. This means, some of the points are closer to points in their own cluster while others are closer to points outside their own cluster. This might suggest the model is right about half the time with its predictions.

One reason for the above problem is the scarcity of documents for training. As a rule, unsupervised learning algorithms get better when they are fed more data. This implies that if we could increase the number of documents inputted in the model, then we would have better results.

Another reason could be the number of clusters. As shown in the elbow plot earlier, the more clusters we have, the closer each point in the cluster as evidenced by the lower inertia. So a higher value of k (number of clusters) might have yielded better accuracy. This however needs to be balanced, after all, if we have a very high number of clusters, then the inertia would automatically go down. For example, if k = number of samples, then each document will be in a cluster on its own. This means that each document is closer to itself than other clusters and as such inertia will be low.

## 2.6 Conclusion

For this coursework, the task to build a search engine and cluster a set of documents using was successful. The KMeans clustering algorithm was used. The search engines return the correct results and the ranking uses a small heuristic. The documents with more of the query words come first. A better ranking algorithm could have been used such as the BM25 algorithm or using cosine similarity.

The clustering task was successfully done. The KMeans algorithm returned three clusters.

# References


Harvey M. Deitel, & Paul J. Deitel. (2019). *Intro to python for computer science and data science: Learning to program with AI, big data and the cloud*. Pearson.

Mousavi Seyed. (2022). *Lecture 1: Information Retrieval. [Powerpoint slides].* Faculty of Engineering, Environment, and Computing, Coventry University (Ed.). Aula Coventry.

Mousavi Seyed. (2022). *Lab 5: KMeans Clustering*. Faculty of Engineering, Environment, and Computing, Coventry University (Ed.). Aula Coventry.

Naik, K. (Producer), & Naik, K. (Director). (2020, Aug 25,). *Silhouette (clustering)- validating clustering models- unsupervised machine learning.* [Video/DVD] https://www.youtube.com/watch?v=DpRPd274-0E

Roy, D. (Producer), & Roy, D. (Director). (2022, Feb 2,). *Developing a basic indexer and retriever.* [Video/DVD] YouTube.

Zhai ChengXiang, & Massung Sean. (2016). Text clustering. *Text data management and analysis: A practical introduction to information retrieval and text mining* (). ACM Books.

# Appendices

## Appendix A: The crawler

```
import scrapy
from scrapy.crawler import CrawlerRunner
from crochet import setup, wait_for
setup()

class PurePortalSpider(scrapy.Spider):
 name = "SEFAAuthors"

 start_urls = [

"https://pureportal.coventry.ac.uk/en/organisations/school-of-economics-finance
-and-accounting/publications/"
 ]

 custom_settings = {
     "FEEDS": {
         "authors.csv": {
             "format": "csv",
             "overwrite": True
         }
     },
     "ROBOTSTXT_OBEY": True,
     "DOWNLOAD_DELAY":  2
 }

 def parse(self, response):
   page = response
   li = page.css('li.list-result-item')
   title = li.css('h3.title span::text').get()
   for item in page.css('li.list-result-item'):
     title = item.css('h3.title span::text').get()
     authors = item.css('a.person span::text').getall()
     date = item.css('span.date::text').get()
     pub_link = item.css('h3.title a.link::attr(href)').extract()
     author_profile = item.css('a.person::attr(href)').getall()
     yield {
         "Title":title,
         "Authors":authors,
         'Date': date,
         'Pureportal': author_profile,
```

```
            'Pub Link': pub_link
        }

    next_page = response.css('a.nextLink::attr(href)').extract()

    if next_page is not None or len(next_page) > 0:
        next_page = response.urljoin(next_page[0])
        #print(next_page)
        yield scrapy.Request(next_page, callback=self.parse)


@wait_for(100)
def run_spider():
 my_spider = CrawlerRunner()
 s = my_spider.crawl(PurePortalSpider)
 return s

run_spider()
```

## Appendix B: Creating the inverted index

```
import string

import pandas as pd
import numpy as np
import re
import sys
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
nltk.download('punkt')
nltk.download('stopwords')
from nltk.corpus import stopwords
stopword = stopwords.words('english')
import collections
from nltk.stem.porter import PorterStemmer

file_path = '/home/sy/Desktop/IR/vertical_search/sefa_scholar/authors.csv'

def create_inverted_index(file_path):
 df_crawled_info= pd.read_csv(file_path)
 df_titles=df_crawled_info["Title"]
  inverted_index={}
```

```
 doc_id = 0
 for x in df_titles:
   #print('review', doc_id)
   #no_dashes = [t.replace('-', ' ') for t in x]
   #a_string.translate(str.maketrans('', '', string.punctuation))
   token = word_tokenize(x)
   token = [t.translate(str.maketrans('', '', string.punctuation)) for t in
token]
   token = [x for x in token if x]
   token_lower = [w.lower() for w in token]
   #no_dashes = [t.replace('-', ' ') for t in token_lower]
   removing_stopwords = [w for w in token_lower if w not in stopword]
   stemmed_title = [stemmer.stem(w) for w in removing_stopwords]

   for a in stemmed_title:
     val = inverted_index.get(a)
     if val == None:
       temp_list = [1,[doc_id]]
       inverted_index[a] = temp_list
     else:
       temp_list =  inverted_index[a]
       if doc_id not in temp_list[1]:
         temp_list[1].append(doc_id)
         temp_list[0] += 1

   doc_id+=1

 ordered_inverted_index =
collections.OrderedDict(sorted(inverted_index.items()))
 return ordered_inverted_index

index_of_titles = create_inverted_index(file_path)
```

## Appendix C: Query preprocessing

```
def processQuery(text):
 #token = word_tokenize(text)
 #token_lower = [w.lower() for w in token]
 #stemmed_title = [stemmer.stem(w) for w in token_lower]
   token = word_tokenize(text)
   token = [t.translate(str.maketrans('', '', string.punctuation)) for t in
token]
   token = [x for x in token if x]
   token_lower = [w.lower() for w in token]
   removing_stopwords = [w for w in token_lower if w not in stopword]
```

```
    stemmed_query = [stemmer.stem(w) for w in removing_stopwords]

    return stemmed_query
```

## Appendix D: Searching and results

```
def search(stemmed_query, index_of_titles):
    all_pl=[]
    for term in stemmed_query:
        pl = index_of_titles.get(term)
        if pl is not None:
            #print(s)
            #print(pl)
            all_pl.append(pl[1])

    return all_pl
def search_results(all_pl):
    df = pd.read_csv(file_path)
    result_ids = list(set.intersection(*map(set,all_pl)))
    #df_search = [df.columns[0:5]]
    #the_result = df.iloc[result_ids]
    the_result = df.iloc[result_ids].T.to_dict('dict')
    return the_result
```

## Appendix E: The flask app

### The app start point: app.py

```
from flask import Flask, render_template, request
from inverted_index import create_inverted_index, processQuery, file_path,
search, search_results
app = Flask(__name__)

@app.route('/', methods =['GET', 'POST'])
def home():

    return render_template('index.html', )


@app.route('/search', methods=['GET', 'POST'])
def result():
    if request.method == "POST":
        # getting input with name = fname in HTML form
        query = request.form.get("search_term")
```

```
        index = create_inverted_index(file_path=file_path)
        stemmed = processQuery(query)
        pre_result = search(stemmed, index)
        result_list = search_results(pre_result)

        return render_template('results.html',result_list=result_list)
    else:
        return render_template('index.html')
```

## The index page (index.html)

```
<html>
    <head>
        <meta name ="viewport" content = "width=device-width,
initial-scale=1.0">
        <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
        <link rel = "stylesheet" type = "text/css" href="static/index.css">
        <title>Search</title>
    </head>
    <body>
        <div class="container text-center">
            <h1 class="header info">SEFA Search</h1>
        </div>
        <form class = "grid" action = "/search" method = "POST">
            <input type = "text" name = "search_term" id = "search_term"
placeholder="Search SEFA" required>
            <input type="submit" class = "file_submit" value="Search">
        </form>
    </body>
</html>
```

## The results page (results.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!--<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">--
>
    <link rel="stylesheet"
```

```
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css">
   <link rel = "stylesheet" type = "text/css" href="static/index.css">
   <title>Results</title>
</head>
<body>
   {% for result in result_list %}
   <div class="card">
       <a href="{{ result_list[result]['Pub Link'] }}"><h3
class="card-header">{{ result_list[result]['Title'] }}</h3></a>
       {% if (result_list[result]['Authors'] is not float) %}
           {% if '.,' in result_list[result]['Authors'] %}
               {% set Authors = result_list[result]['Authors'].split('.,') %}
               {% set links = result_list[result]['Pureportal'].split(',') %}
               {% for i in range(Authors|length) %}
       <div class="card-body">
                   <a href="{{links[i]}}" class="card-link" target="_blank"><p
class="p-1"><span>{{ Authors[i] }}</span></p></a>
               {% endfor %}
           {% else %}
               <a href="{{ result_list[result]['Pureportal'] }}"
class="card-link" target="_blank"><p class="p-1"> {{
result_list[result]['Authors'] }}</p></a>
           {% endif %}
       {% else %}
           <p class="p-1">No authors in SEFA</p>
       {% endif %}
       <p class="p-1">{{ result_list[result]['Date'] }}</p>
       </div>
   </div>
   {% endfor %}
</body>
</html>
```

## The styling for the pages (index.css)

```
html,body{
   /* margin: 10px; */
   padding: 7% 10% 0 10%;
   list-style: none;
   box-sizing: border-box;
   font-family: Arial,sans-serif;
   letter-spacing: 3px;
}
```

```css
/* element */
 .let_space{
   letter-spacing: 2px;
}

img{
   padding-left: 25%;
   width: 70%;
}

.c_grid{
   display: grid;
   justify-items: center;
   align-items: center;
   grid-gap: 2vmax;
}

.grid{
   display: grid;
   grid-gap: 2vmax;
}

.box2{
   grid-template-columns: 1fr 1fr;
}
 .box3{
   grid-template-columns: 1fr 1fr 1fr;
}

/* element */

.table_cont{
   overflow-x: scroll;
   width: 70vw;
}

table {
   border-collapse: collapse;
   letter-spacing: 3px;
   /* width: 100%; */
}

td, th {
   border: 1px solid #dddddd;
```

```
    text-align: center;
    padding: 1vmax;
    width: fit-content;
}

form{
    margin: 2vmax 10vmax;
    /* padding-top: 15%; */
   /* border: 1px solid #dddddd; */
}

label{
    font-size: 1.5vmax;
}

input{
    padding: 2%;
    font-size: 1.3vmax;
    border: 1px solid #dddddd;
    outline: none;
    border-radius: 30px;
    border-color: #d9d4d4;
}

select{
    padding:1vmax;
    font-size: 1.3vmax;
    border: 1px solid #dddddd;
    outline: none;
    letter-spacing: 3px;
}


option{
    font-size: 1.5vmax;
    letter-spacing: 3px;
}

.file_submit{
 margin-top: 1vmax;
 border: 1px solid #0066cb;
 outline: none;
 background-color: #e3eff9;
 color: 0066cb;
```

```
 padding: 1vmax;
 cursor: pointer;
 transition: 0.3s;
 letter-spacing: 3px;
 padding: 15px;
 max-width: fit-content;
 margin: auto;
}

a:link {
   text-decoration: none;
}

.file_submit:hover{
 background-color: white;
 color: #0066cb
}
```

## Appendix F: Clustering Setup and document gathering

```
import pandas as pd
import feedparser
from wordcloud import WordCloud
```

```
health = feedparser.parse('https://www.dailymail.co.uk/health/index.rss')
sport = feedparser.parse('https://www.dailymail.co.uk/sport/index.rss')
politics = feedparser.parse('https://www.buzzfeed.com/politics.xml')
politics_feed =
feedparser.parse('https://rss.nytimes.com/services/xml/rss/nyt/Politics.xml')
```

## Appendix G: Creating the documents df

```
my_dict = {
   'id': [],
   'content': [],
   'category': 'Health'
}

my_dict2 = {
   'id': [],
   'content': [],
   'category': 'Politics'
}

my_dict3 = {
```

```
    'id': [],
    'content': [],
    'category': 'Sports'
}

id_count = 0

for entry in health.entries:
 #print(entry['summary'])
 my_dict['id'].append(id_count + 1)
 my_dict['content'].append(entry['summary'])
 id_count += 1

for entry in sport.entries:
 #print(entry['summary'])
 my_dict2['id'].append(id_count + 1)
 my_dict2['content'].append(entry['summary'])
 id_count += 1

for entry in politics.entries:
 #print(entry['summary'])
 my_dict3['id'].append(id_count + 1)
 my_dict3['content'].append(entry['summary'])
 id_count += 1

for entry in politics_feed.entries:
 #print(entry['summary'])
 my_dict3['id'].append(id_count + 1)
 my_dict3['content'].append(entry['summary'])
 id_count += 1


health_df = pd.DataFrame(my_dict)
politics_df = pd.DataFrame(my_dict3)
sports_df = pd.DataFrame(my_dict2)
full_df = health_df.append(politics_df).append(sports_df)
```

## Appendix H: Running the elbow method

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(stop_words={'english'})
article_list = full_df['content']
X = vectorizer.fit_transform(article_list)

import matplotlib.pyplot as plt
```

```
from sklearn.cluster import KMeans
Sum_of_squared_distances = []
K = range(2,10)
for k in K:
  km = KMeans(n_clusters=k, max_iter=200, n_init=10)
  km = km.fit(X)
  Sum_of_squared_distances.append(km.inertia_)

plt.plot(K, Sum_of_squared_distances, 'bx-')
plt.xlabel('k')
plt.ylabel('Sum_of_squared_distances')
plt.title('Elbow Method For Optimal k')
plt.show()
```

## Appendix I: Preprocessing the  documents

```
documents = full_df['content']
documents
import nltk
nltk.download("stopwords")
from nltk.corpus import stopwords

sw = stopwords.words('english')
print(sw)
nltk.download("punkt")
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

ps = PorterStemmer()
filtered_docs = []
for doc in documents:
   tokens = word_tokenize(doc)
   tmp = ""
   for w in tokens:
       if w not in sw:
           tmp += ps.stem(w) + " "
   filtered_docs.append(tmp)

print(filtered_docs)
```

## Appendix J: Training the model

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(filtered_docs)
```

```
#print(X.todense())
from sklearn.cluster import KMeans
K = 3
model = KMeans(n_clusters=K, init='k-means++', max_iter=100, n_init=1)
model.fit(X)

print("cluster no. of input documents, in the order they received:")
print(model.labels_)

from collections import Counter
cluster_label = model.fit_predict(X)
cluster_count = Counter(cluster_label)
cluster_count
```

## Appendix K: Testing the model

```
Y = vectorizer.transform(["The covid virus infects all people but is more fatal
for old people"])
prediction = model.predict(Y)
print(prediction)

Y = vectorizer.transform(["football is an amazing sport"])
prediction = model.predict(Y)
print(prediction)

Y = vectorizer.transform(["Messi and Ronaldo have been among the best players
ever seen"])
prediction = model.predict(Y)
print(prediction)

Y = vectorizer.transform(["Trump and Biden are very different presidents"])
prediction = model.predict(Y)
print(prediction)
```

## Appendix L: Visualizing the centroids

```
print("Cluster centroids: \n")
order_centroids = model.cluster_centers_.argsort()[:, ::-1]
terms = vectorizer.get_feature_names_out()
doc1 = []
doc2 = []
doc3 = []
for i in range(K):
   print("Cluster %d:" % i)
   for j in order_centroids[i, :50]: #print out 50 feature terms of each
```

```
cluster
       #print (' %s' % terms[j])
       if i == 0:
         doc1.append(terms[j])
       elif i == 1:
         doc2.append(terms[j])
       elif i == 2:
         doc3.append(terms[j])
   print('------------')
import matplotlib.pyplot as plt
#text = order_centroids[0, :10]
#
# Create and generate a word cloud image:
wordcloud = WordCloud().generate(' '.join(doc1))

#Display the generated image:
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
print(doc2)
wordcloud = WordCloud().generate(' '.join(doc2))

#Display the generated image:
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
wordcloud = WordCloud().generate(' '.join(doc3))

#Display the generated image:
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

## Appendix M: Evaluation

```
from sklearn.metrics import silhouette_score
silhouette_avg = silhouette_score(X, cluster_label)
silhouette_avg
```