Text

Description automatically generated with medium confidence

Module Title: Information Retrieval

Module Code: STW7071CEM

Date of Submission: 07/07/2023

Intake: March 2023

Submitted By: Submitted To:

Suraj Neupane Siddhartha Neupane

Contents

[Introduction: 3](#_Toc139560500)

[Task 1: Vertical Search Engine: 4](#_Toc139560501)

[Crawler: 5](#_Toc139560502)

[Retrieving data: 6](#_Toc139560503)

[Saving Data: 7](#_Toc139560504)

[Pagination: 8](#_Toc139560505)

[Inverted Index: 9](#_Toc139560506)

[Pre-Processing words: 9](#_Toc139560507)

[Creating Inverted Index: 11](#_Toc139560508)

[Query Processing: 12](#_Toc139560509)

[Ranked retrieval: 14](#_Toc139560510)

[TF-IDF: 15](#_Toc139560511)

[Cosign similarity: 15](#_Toc139560512)

[Task 2: Text Classification: 18](#_Toc139560513)

[Dataset 18](#_Toc139560514)

[Naïve Bayes Prediction: 19](#_Toc139560515)

[Appendix: 21](#_Toc139560516)

[Project Structure: 21](#_Toc139560517)

[User Interface 21](#_Toc139560518)

[Crawler Time: 21](#_Toc139560519)

[Conclusion: 21](#_Toc139560520)

# Introduction:

This document describes the process of creating a vertical search engine and a text classification modal. For vertical search engine, publication data from Coventry University's Research Centre for Intelligent Healthcare is collected. For the extraction of data, a simple crawler is used using python’s beautiful soup. From the extracted data, an inverted index is created. Inverted index is used to collect unique terms from the collection of extracted data. Documents form search query is filtered using inverted indexes and it’s ranked by using cosign similarity. Purpose of this document is to get the indebt process of creating a vertical search engine and tools and methods used can be found in details in this document. Text classification is used to classify the certain documents from the context of the document. In the document, technologies and methods used to classify documents can be found in details as well. In this assignment Python programming language is used because of its similarity and versatility.

# Task 1: Vertical Search Engine:

Information retrieval is the process of obtaining relevant information from a large collection of data. In today's digital age, where the volume of information is exponentially increasing, efficient retrieval systems are crucial. One such system is the vertical search engine.

Unlike traditional search engines that provide general results, vertical search engines focus on specific industries, niches, or domains. They are designed to cater to users searching for specialized information in fields like finance, healthcare, travel, and more. Vertical search engines offer targeted results by indexing and organizing content from relevant sources within their specific domain.

Vertical search engines employ various methods to enhance search accuracy and deliver targeted results within specific domains. Three key methods utilized are Boolean search, inverse indexing, and rank retrieval.

Inverse indexing involves creating an index that maps keywords to the documents where they appear. This enables rapid retrieval of documents containing specific keywords. Inverse indexing enhances search efficiency by reducing the search space.

Rank retrieval algorithms determine the relevance of documents to a given query. These algorithms analyze factors such as keyword frequency, document popularity, and user behavior to rank search results. Rank retrieval ensures that the most relevant and valuable documents appear at the top of search results.

By leveraging these methods, vertical search engines provide users with accurate and targeted information within specialized domains. These techniques improve search efficiency, enable precise control over results, and ensure the most relevant content is easily accessible.

Task 1 is separated into different components and we will go through each sections one by one to reach functional vertical search engine.

## Crawler:

A web crawler is a program that systematically scans websites to gather information. It plays a crucial role in information retrieval by indexing web pages, collecting data, and building a searchable database. Web crawlers automate the process of gathering data, making it faster and more efficient for search engines to provide up-to-date and comprehensive search results.

Since we are using python as out base programming language, for crawler component*, beautiful soup* is used. As mentioned in assignment following link is used to for gathering data and search engine will utilized the data on that page.

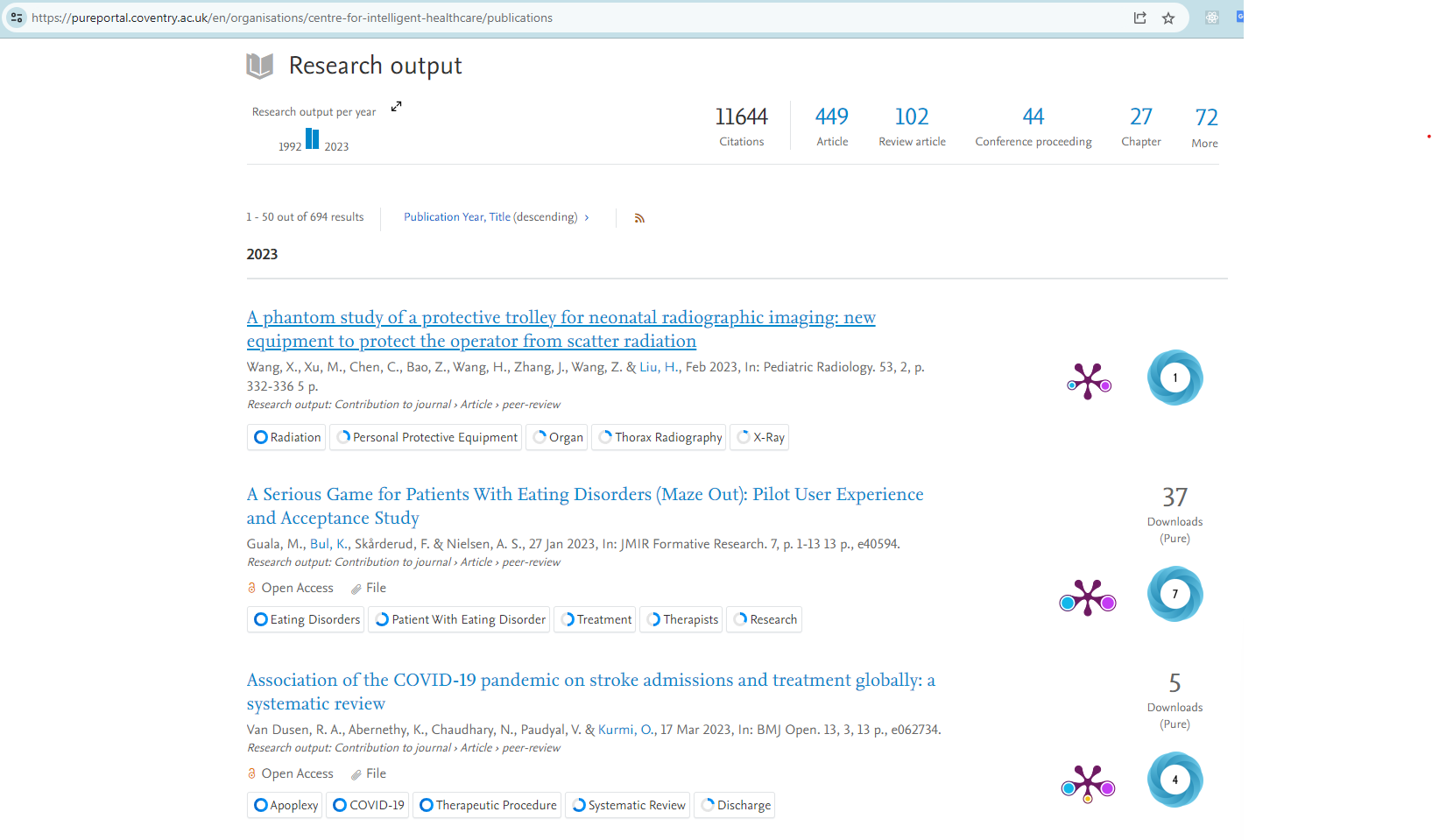


Figure 1: Information source (URL and general view) from assignment

Following image is screenshot of crawler component which demonstrate the URL being used to retrieve data.

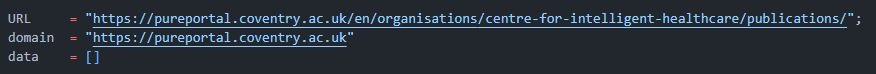


Figure 2: URL and domain name used in crawler component

Crawler component is made from following steps.

### Retrieving data:

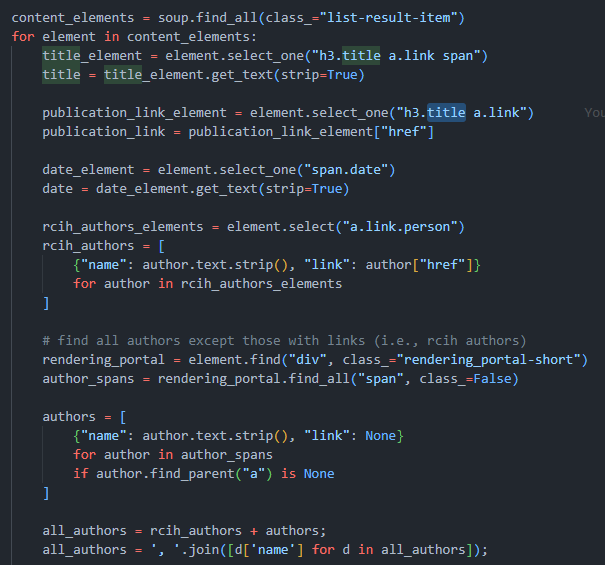


Figure 3: Crawler component, retrieving data

In above code, all important data is extracted. As we can see, there are different category of authors in above code, which are, authors not associated with link, authors with profile link and all authors combination of both. It’s done because we will need 2 different types of author to display on search results and we need all authors because we want to search all author. All author part is utilized in inverse index.

### Saving Data:

Those Retrieveddata is saved on a dictionary called *data*.

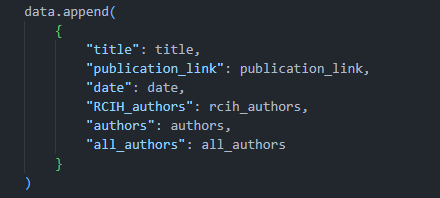


Figure 4: data dictionary

After all data is extracted from URL, it will be saved in a .csv file for easy use from “pandas” library.

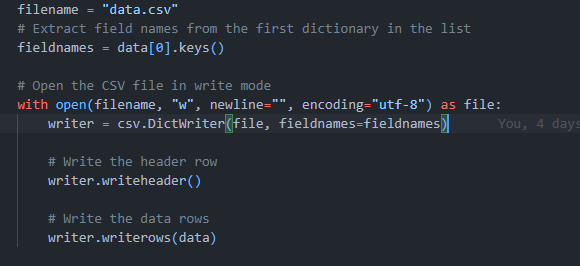


Figure 5: code to write data in csv file

*Note: In above code, .csv file will be overwritten each time crawler extracts the data. It’s done because, it will take a lot more time to update data one by one and there is no significant performance drawback and functional drawback by overrating old files entirely.*

### Pagination:

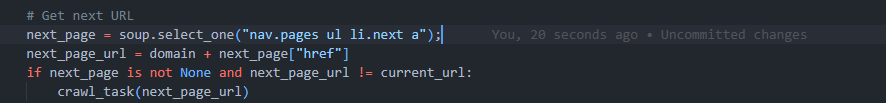


Figure 6: code to deal with pagination

As seen in above code, pagination is done by finding the link for next page. Once it stop finding next page, it will stop the crawler and save data. It checks if current page and next page are not to prevent an infinite loop in case something breaks on source website.

Example of retrieved data in data.csv:

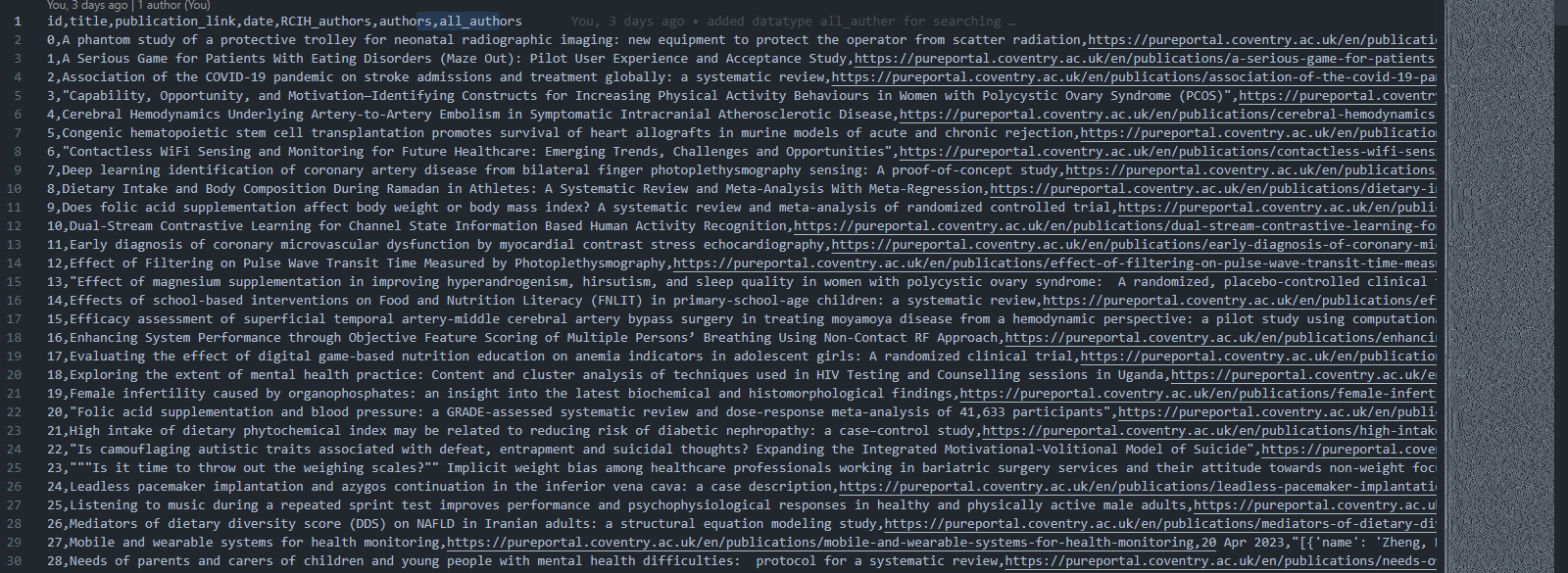


Figure 7: example of retrieved data in data.csv

## Inverted Index:

The inverted index is a crucial component in vertical search engines, offering several advantages that enhance their efficiency and effectiveness. One advantage is improved search performance. The inverted index organizes terms and their associated documents, allowing for quick retrieval based on specific keywords. This enables faster search times, even when dealing with large datasets. Additionally, the inverted index enhances relevance ranking. By keeping track of term frequencies, it allows search engines to prioritize documents with a higher occurrence of relevant terms. This results in more accurate and targeted search results, improving the overall user experience. Furthermore, the inverted index enables advanced features like faceted search, filtering, and drill-down capabilities, allowing users to refine their searches and narrow down results in a vertical-specific context. Overall, the inverted index significantly boosts the search capabilities of vertical search engines, enabling efficient and relevant information retrieval within specialized domains.

Process followed to create an inversed index can be found below:

### Pre-Processing words:

Lemmatizing words is crucial for an inverted index in vertical search engines because it reduces variations of word forms to their base or root form. By lemmatizing, different forms of a word are mapped to a common term, enabling more accurate and comprehensive retrieval of relevant documents. This ensures that users can find information regardless of the specific word forms used in their queries.

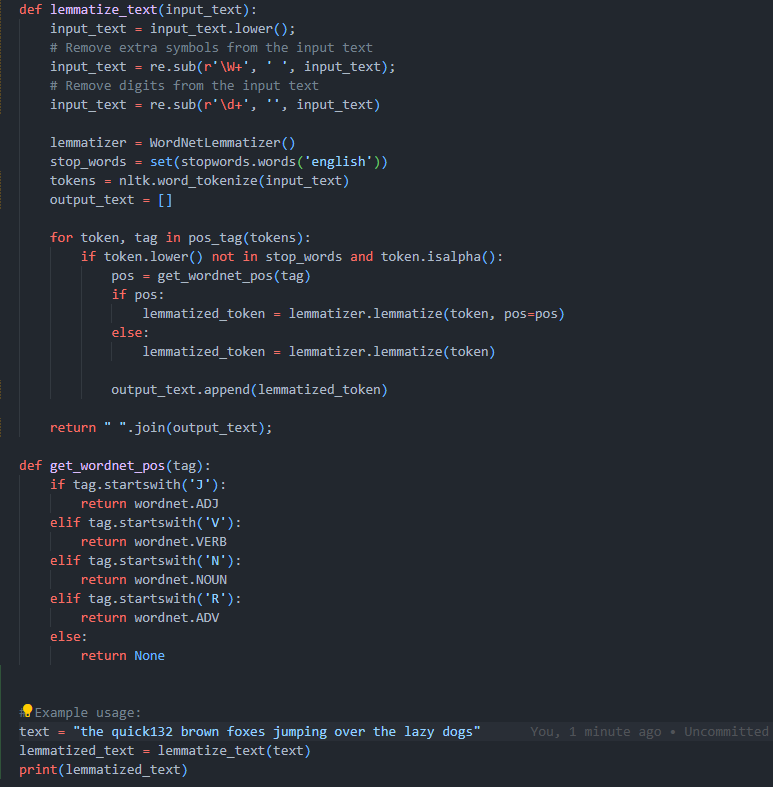


Figure 8: Code for lemmatizing text - with example

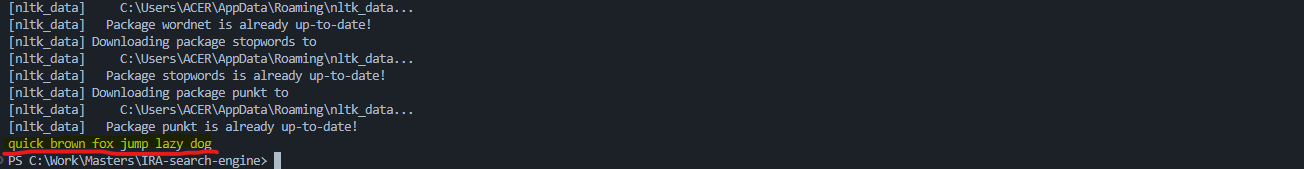


Figure 9: output of above example

*Note: above function is with an example, which only made for demonstration purpose, it cannot be found in final version.*

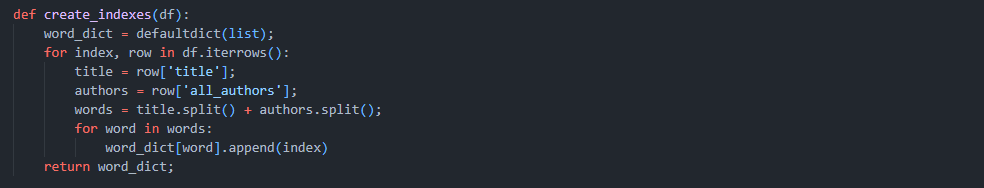
Creating Inverted Index: **

Figure 10: Algorithm for creating inversed index

Above code shows the algorithm to create inverted index. In above example words from title and all authors are saved as text in data-frame and an index is made for each words with location of the document as value. Output is saved in a file because it’s doesn’t needs to be updated in each query and result from above is reusable.

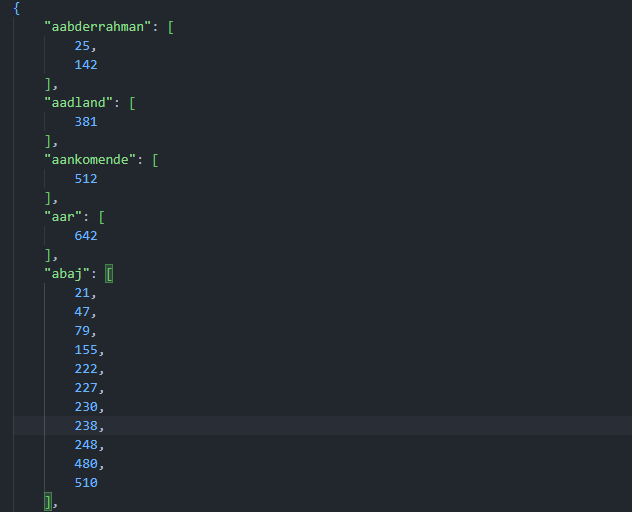
**

Figure 11: small example of inverted index output

## Query Processing:

When user inputs a query, it goes through pre-processing similar to inverted index since we want to find the particular data from the index we saved.

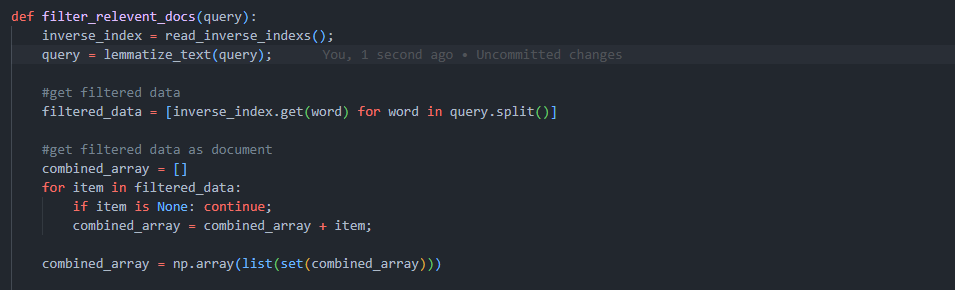


Figure 12: relevant documents from inverted index

In above code, after lemmatizing query, it filter out relevant document number from inverted index we saved before and it extracts the actual document data from document number to process further.

Even without calculating relevant score, query processing with inverted index is capable of searching results.

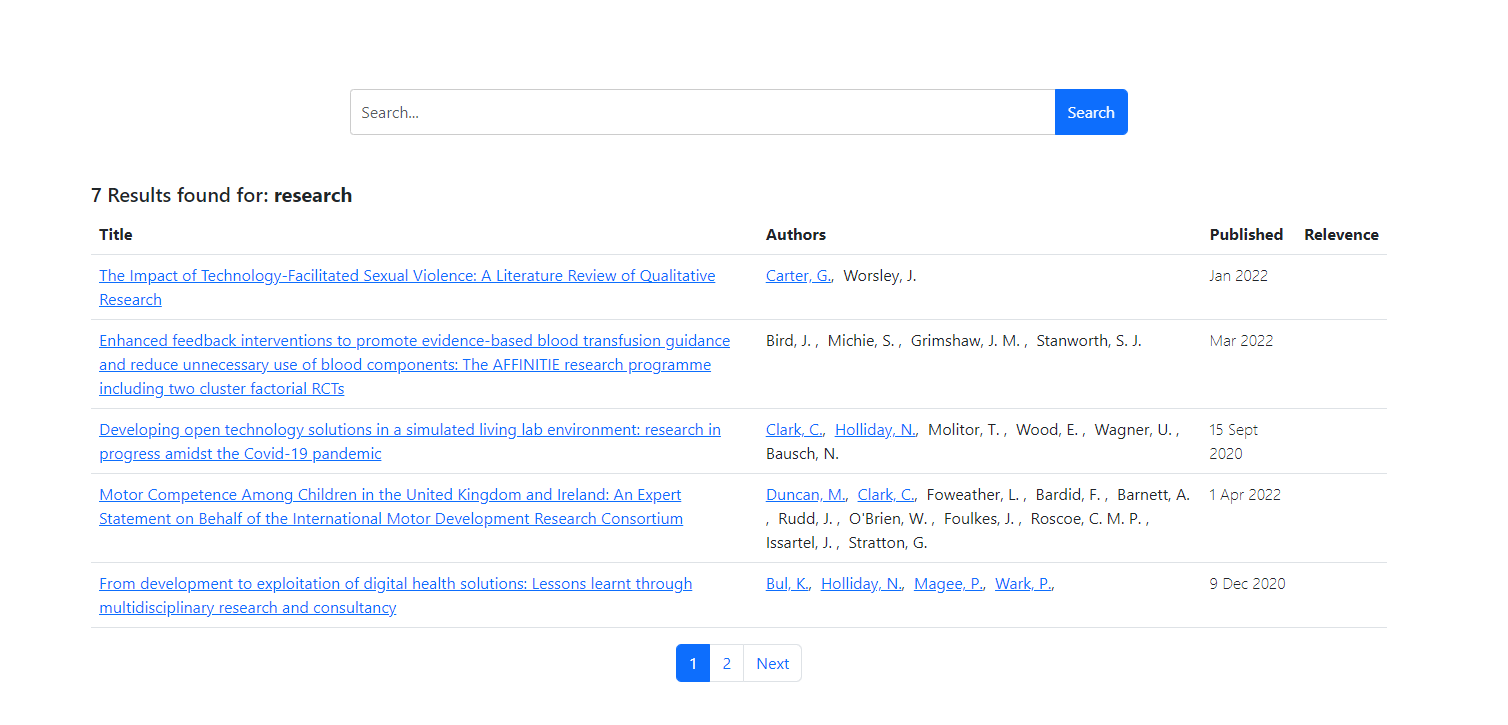


Figure 13: example 1 of vertical search without relevance score

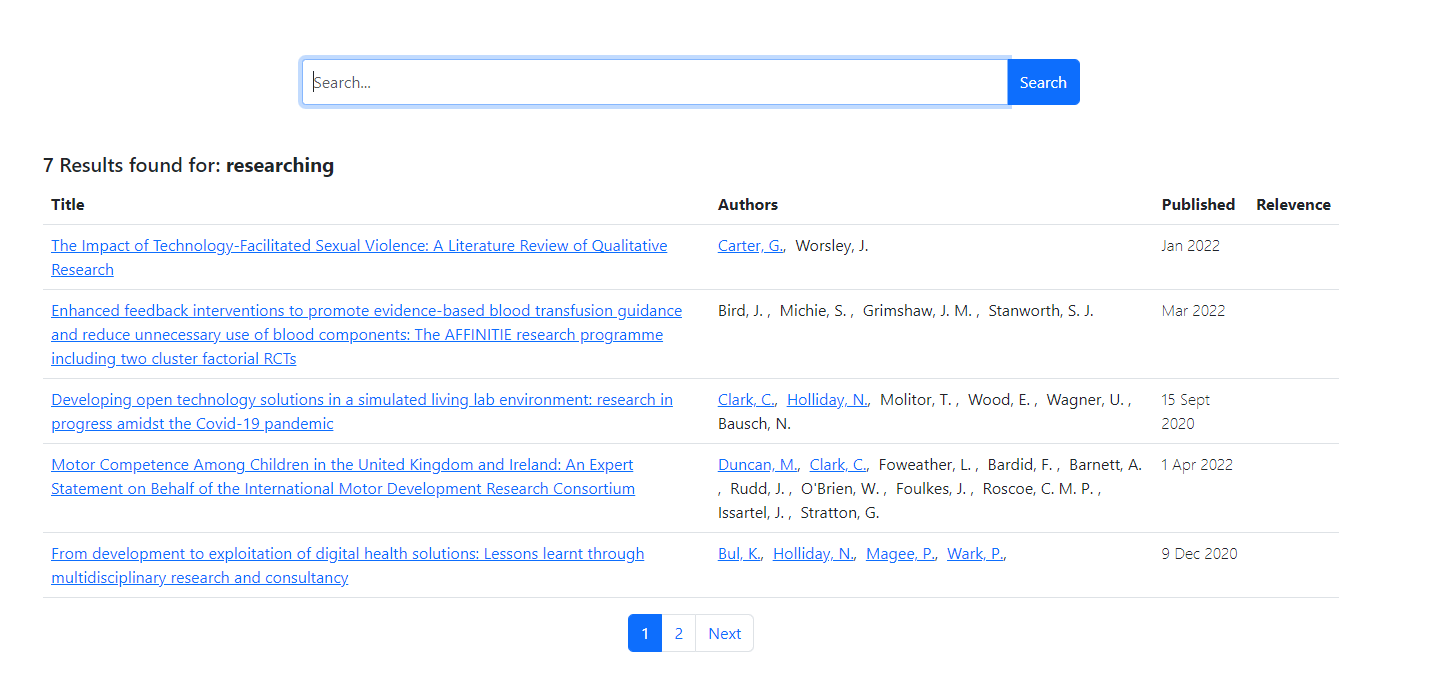


Figure 14: example 2 of search without relevance score with similar query to example 1

*Note: In above examples, search query is similar “process” and “processing” but they produce exact same result. This demonstrate the use of lemmatized words. Also, in above examples, documents are ranked by default index number.*

## Ranked retrieval:

Ranked retrieval is a technique used in information retrieval to rank documents based on their relevance to a given query. It is essential because it helps users find the most relevant information efficiently.

To rank relevant documents, various methods can be employed. One commonly used approach is TF-IDF (Term Frequency-Inverse Document Frequency) metric, which calculates the importance of a term in a document relative to its occurrence across all documents. Another method is the Vector Space Model, which represents documents and queries as vectors in a high-dimensional space and measures their similarity.

Additionally, Word2Vec, a popular word embedding model, can capture semantic relationships between words and enhance retrieval accuracy. These techniques collectively contribute to effective ranked retrieval, empowering users to quickly access relevant information amidst vast amounts of data.

For this assignment we will going to use TF-IDF metric using a python library called “sklearn”.Continuing from above progress of filtering relevant document, in this section we will going to find out which document is more relevant by scoring a rank using TF-IDF metric.

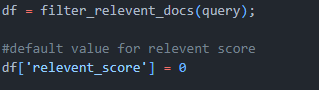


Figure 15: code to get previous data and adding a default score

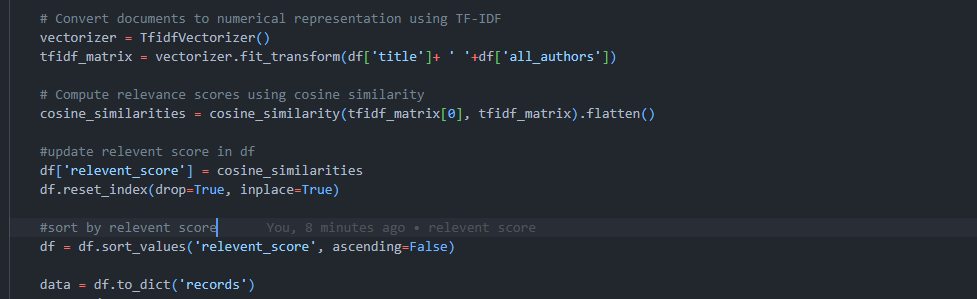


Figure 16: code for ranked retrieval using TF-IDF

### TF-IDF:

In above code, we transform two of the data into a TF-IDF vector, based on formula:

Where:

**TF (Term Frequency)** represents the frequency of a term in a document. It measures the importance of the term within the specific document and is usually calculated as the number of times a term appears in a document divided by the total number of terms in that document.

*TF = (Number of occurrences of a term in a document) / (Total number of terms in the document)*

**IDF (Inverse Document Frequency)** measures the rarity or uniqueness of a term across all documents in a collection. It helps give more weight to terms that are less common and more discriminative. IDF is typically calculated as the logarithm of the total number of documents divided by the number of documents containing the term, with the addition of 1 to avoid division by zero.

*IDF = log((Total number of documents) / (Number of documents containing the term)) + 1*

### Cosign similarity:

After the calculation of TF-IDF vector, cosign similarity between each document is calculated. Cosine similarity is used to measure the similarity between two vectors as demonstrated in picture below.

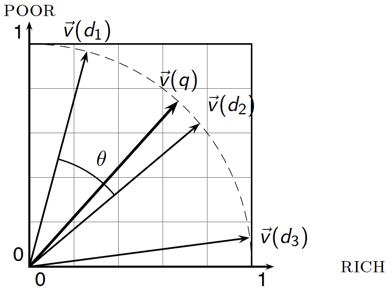


Figure 17: Cosign similarity between query vector and document vector

In above figure, we can see that query vector (q) is closest to document (d2). Which means, document (d2) is more relevant with query than any other documents. Code seen [here](#tfidf) will calculate the Cosign similar for all filtered documents and most relevant documented is ranked at top and can be seen at top on search result.

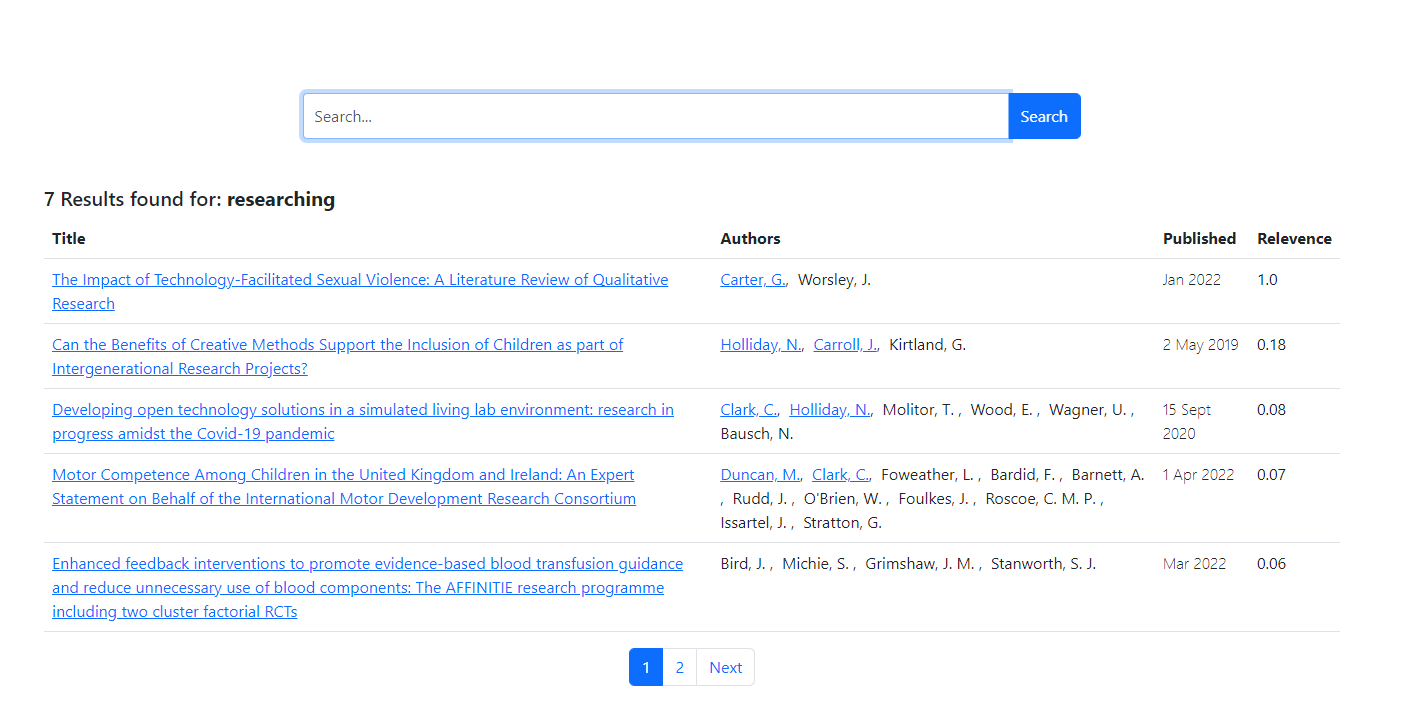


Figure 18: example 1 of search result with ranked retrieval using TF-IDF

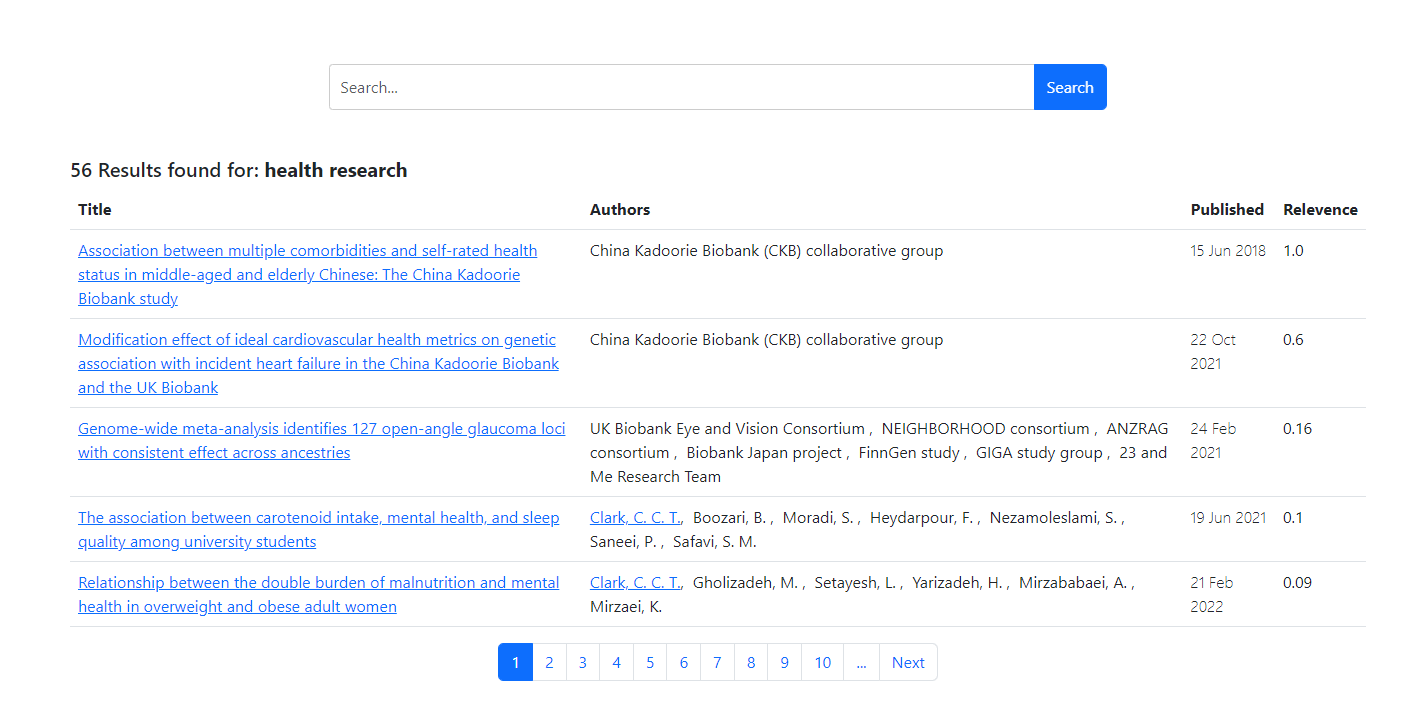


Figure 19: example 2 of search result with ranked retrieval using TF-IDF

# Task 2: Text Classification:

Text classification is the task of assigning predefined categories or labels to text documents based on their content. One popular method for text classification is the Naïve Bayes algorithm. Naïve Bayes is a simple yet effective classification method based on Bayes' rule from probability theory. It assumes that the features (words) in a document are conditionally independent given the class label. Naïve Bayes relies on a simple representation of documents, such as the bag-of-words model. In the case of text classification, the Multinomial Naïve Bayes variant is commonly used. It treats each word as a separate feature and calculates the likelihood of each class based on the frequency of words in the document. While the approach ignores the word order and document structure, Multinomial Naïve Bayes can still provide satisfactory results for many text classification tasks.

In following section, we will look at the process took place to implement Naïve Bayes algorithm for text classification.

Dataset:

Dataset is prepared from randomly selection news articles about giving topics in assignment (which are Health, business, and Sport). Dataset is saved in json file and later copied in .csv file for simplicity.  
Data is preprocessed similar to [task 1](#_Pre-Processing_words:) data when it’s being used.

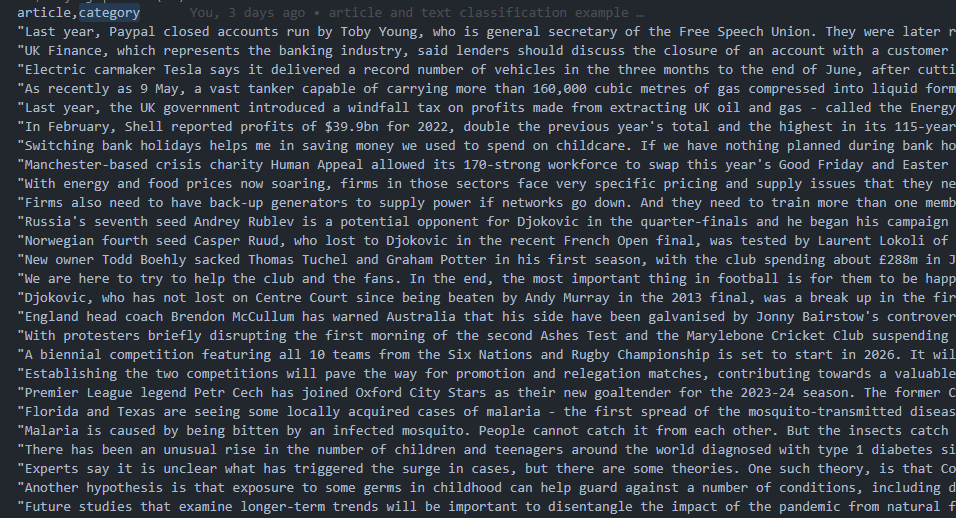


Figure 20: small view of saved dataset from various sources

## Naïve Bayes Prediction:

It is based on Bayes' theorem and assumes that the features (words) in a document are independent of each other, given the class label. The formula for Naïve Bayes can be expressed as:

P(y | x) = (P(x | y) \* P(y)) / P(x)

Where:

P(y | x) is the posterior probability of class y given the features x,

P(x | y) is the likelihood probability of features x given class y,

P(y) is the prior probability of class y,

P(x) is the evidence probability.

By estimating these probabilities from the training data, Naïve Bayes can classify new documents by calculating the probability of each class and assigning the document to the class with the highest probability.

Here is the code for implanting Naïve Bayes in dataset we prepared:

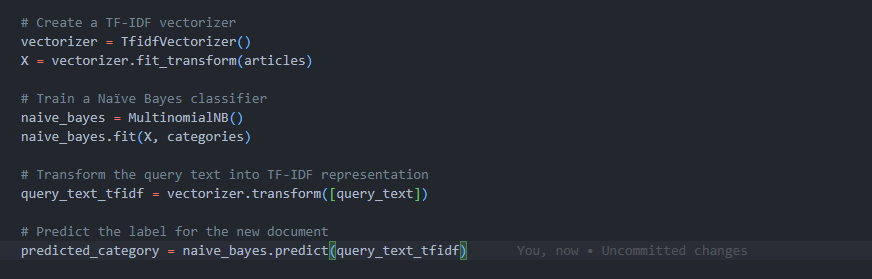


Figure 21: Implementation of Naive Bayes on prepared dataset

In above code, articles are converted into TF-IDF vectors similar to [ranked retrieval](#_TF-IDF:). However, instead of finding most related article, we will find most related category. As seen in above code, after TF-IDF vectors, those vectors is trained in a modal. Those models are implanted into this formula:

P(y | x) = (P(x | y) \* P(y)) / P(x)

In our case, P(y | x) is probability of category given the features articles. Predicted input is returned and displayed to user (with a pie chart).

# Appendix:

## Project Structure:

## User Interface

## Crawler Time:

# Conclusion: