Table of Contents

[2 Task 1 – Search Engine 1](#_Toc91934877)

[2.1 Crawler 2](#_Toc91934878)

[2.1.1 Brief explanation of how it works 7](#_Toc91934879)

[2.2 Indexer 7](#_Toc91934880)

[2.2.1 Brief explanation of how it works 8](#_Toc91934881)

[2.3 Query Processor 9](#_Toc91934882)

[3 References 11](#_Toc91934883)

# Task 1 – Search Engine

Search engine is basically an engine developed to retrieve information(s), for example “Google Search Engine”[1], retrieves all kinds of information and is a general-purpose search engine. So far Google has been the best search engine that we can find these days and it gives results highly relevant to the user query.

The other kind of search engine is called a “Vertical Search Engine”, this search is built for a specific application unlike Google. For example, Skyscanner [2] and Indeed.com [3] are vertical search engines and they retrieve specific information related to flight results by Skyscanner and Jobs search results by Indeed.com. And both vertical search engines have outclassed google in a very specialized domain.

Here in this project, I will be developing a vertical search engine specialized to retrieve the following information from Coventry University SLS website [4],

1. Publication Title
2. Publication Link
3. Publication Author(s) name
4. Author(s) profiles link

However, my search engine will be just a prototype.

## Crawler

To extract information, we first need to first build a crawler. Crawler is the first part of building a search engine, what a crawler does is keeps on crawling to different webpages to find information related to user queries.

My crawler starts from seed URL [5], and extracts a total of 658 publication, which is also the total number of publications on the CU SLS website as shown in Figure – 1.

Graphical user interface, text, application, email

Description automatically generated

Figure 1: Total Publications

The crawler then extracts the following information in a sequential way as shown in Figures-2 and 3 ,

1. Publication Title
2. Publication Link
3. Publication Author(s) name
4. Author(s) profiles link
5. Co-Author(s) names

Text, application

Description automatically generated

Figure 2: Data ExtractionText, application

Description automatically generated

Figure 3: Data Extraction

Pre-Processing is the process of filter your extracted information and removing any unnecessary information, thereby making your data cleaner and more efficient.

After my crawler extracted the above-mentioned information, I saved all this data in a “./Data\_Extracted\_file.csv” file as shown in Figure-4.

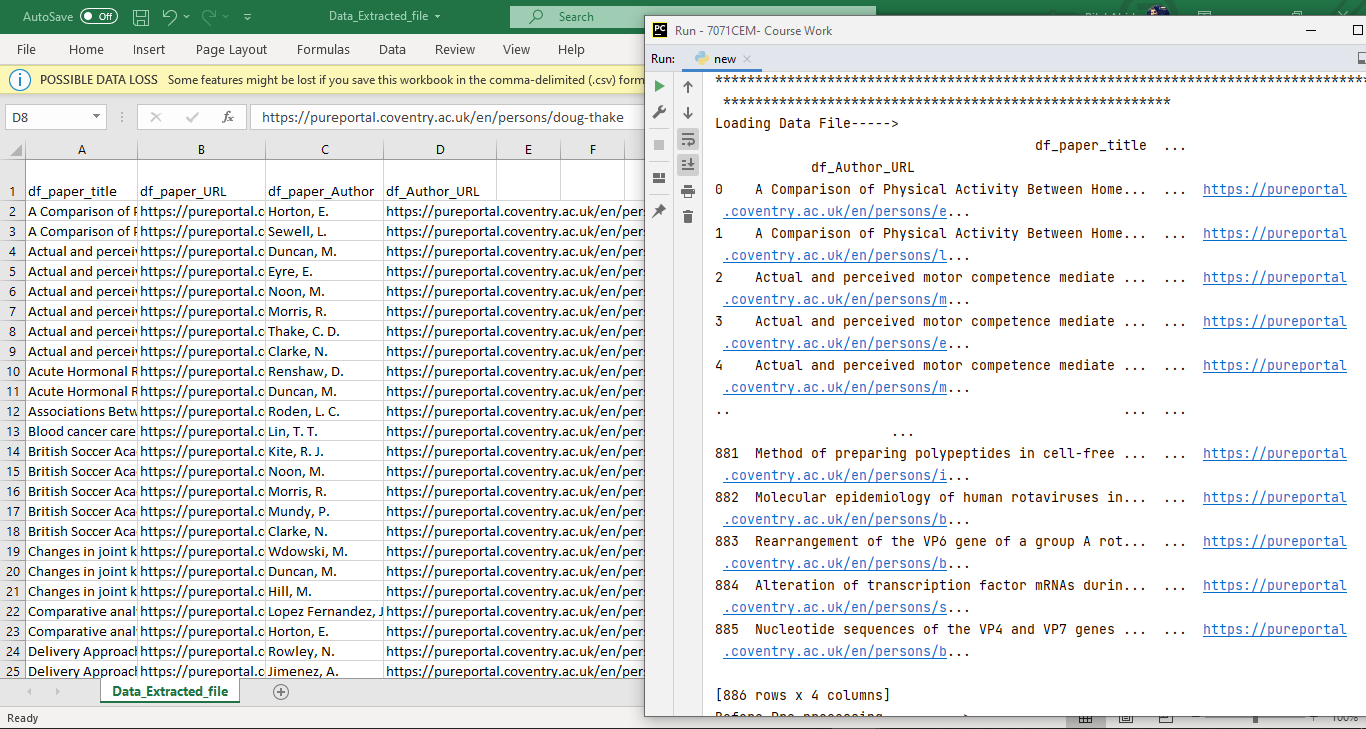


Figure 4: Data\_Extracted\_file.csv

Now I need to perform some pre-processing on this raw data stored inside “Data\_Extracted\_file.csv”. After loading this file into my program, first I separated all the documents stored in that file by using “.split(“\n”)” method. Now each of my document is separated from each other. After that I converted all these documents to lowercase and removed the stop words using the (nltk) library. Now the pre-processed data is stored inside “filtered\_docs”. Figure-5 and 6, shows data before and after pre-processing.

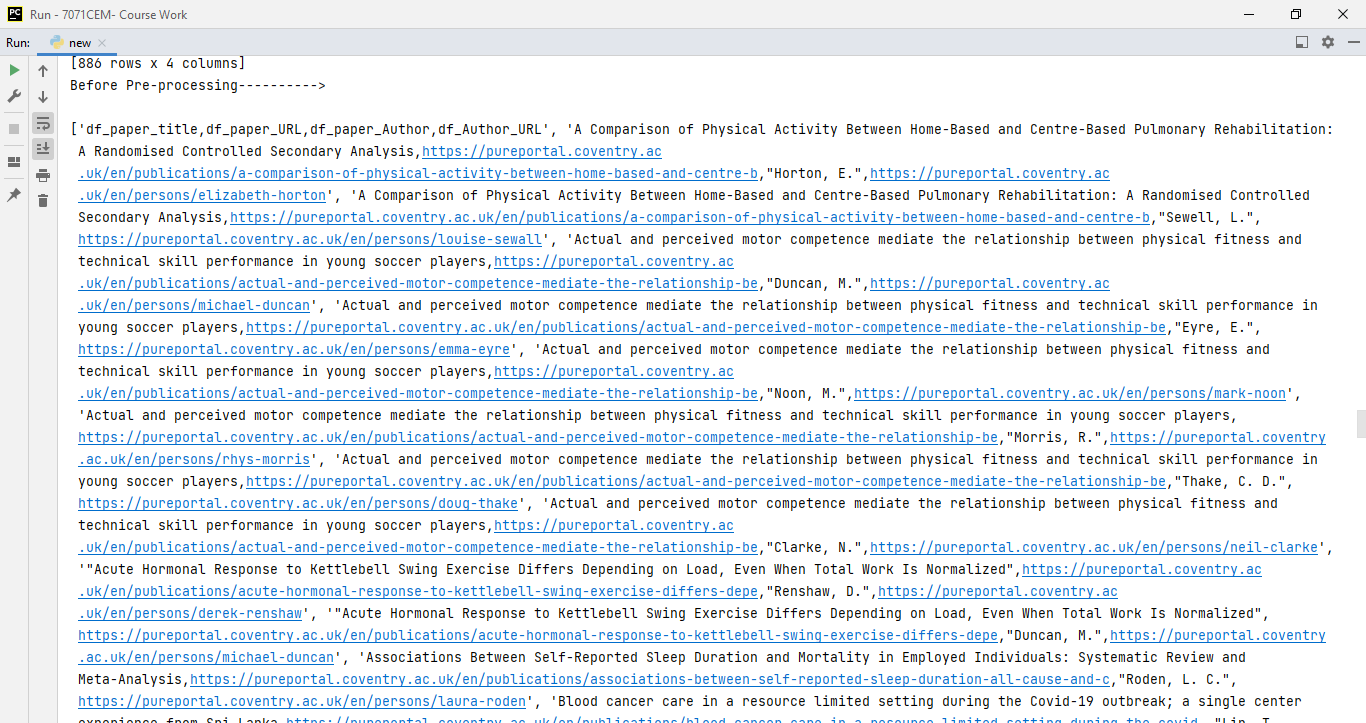


Figure 5: Data before pre-processing

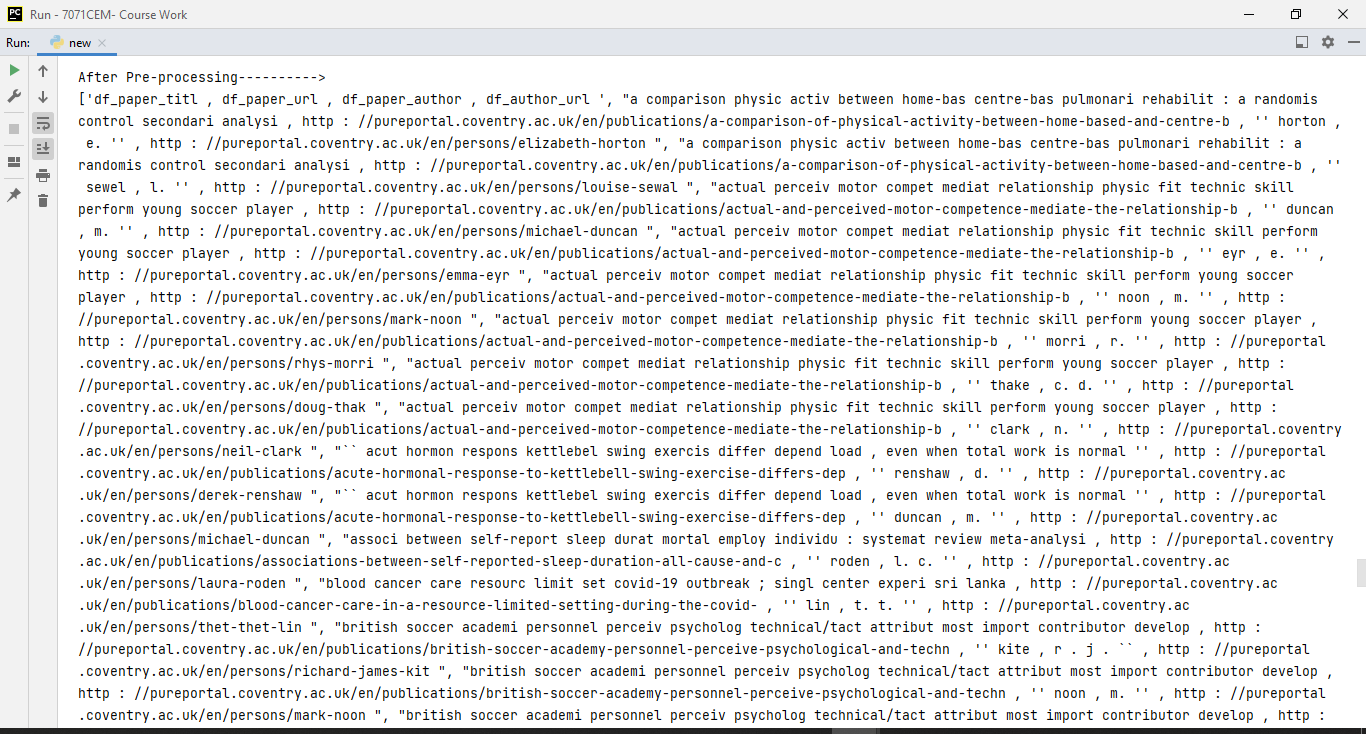


Figure 6: Data after pre-processing

My crawler right now is programmed to run manually.

Breadth First Search (BFS) method first stores all the URL/ links and then the crawler fetches each of these links. I have used BFS in my crawler as shown in Figures (7.1, 7.2, 8.1, 8.2).

Text

Description automatically generated

Figure 7.1: BFS Search

Text

Description automatically generated

Figure 8 : BFS Search

Text

Description automatically generated

Figure 9 : BFS Search

Text

Description automatically generated

Figure 10 : BFS Search

### Brief explanation of how it works

I have programmed my crawler using BFS, it first stores all the pagination links (number of pages = 14) inside an empty queue (“Queue”) that I have created, then it goes through the first page extracts all the raw data (publication and authors URL’s and raw data strings), these URLs are also stored inside the “Queue” and all the raw data is stored inside “Data\_Extracted\_file.csv” file.

## Indexer

Indexer is used to index/ relate the data elements together, so when later when our Query processor runs the query, the indexer will show the relevant index information for the given query.

Here I have created an Incidence index matrix data structure using the TF-IDF method as shown in Figure-11 and 12.

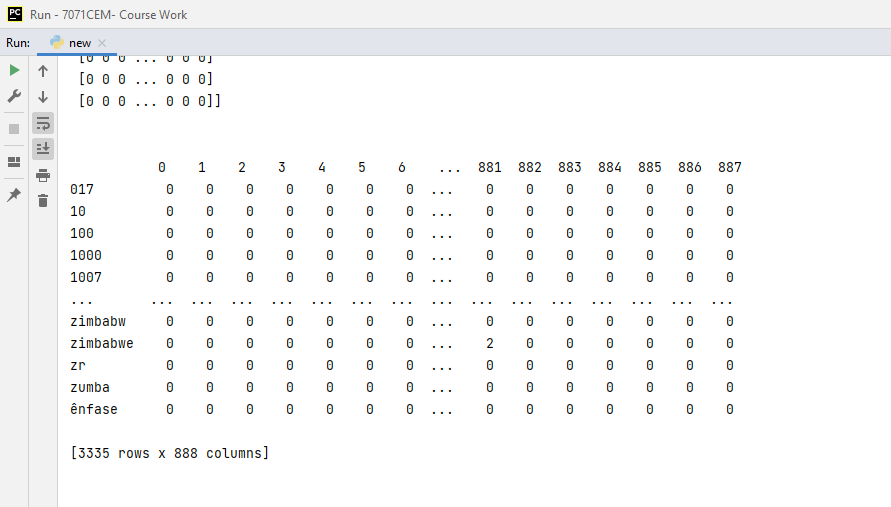


Figure 11: Incidence Matrix Index

Graphical user interface, application, table, Excel

Description automatically generated

Figure 12: Incidence Matrix Index csv file

Whereas,

TF = is the term frequency, and IDF = Inverse Document Frequency.

And each of these cells inside the Incidence index matrix is referenced/ called based on two attributes (Term and DocID).

### Brief explanation of how it works

After filtering the information stored inside “filtered\_docs”, I imported the “pandas” and “sklearn” library to manipulate the data and convert all the documents into vector (vector space). Here I have used “Bag of words model” approach for my index using “CountVectorizer” as opposed to the “set of words models” which doesn’t allow for duplicate words.

Now the complete vector space is stored inside “X” (it stores all documents vectors inside it) as shown in Figure, which is converted to array first and then using method(vectorizer.get\_features\_names\_out()), I extracted all the features names and stored them inside a Data Frame object (“pd”) using the pandas library. Now all that’s left was to save this Incidence Matrix to csv file.

## Query Processor

The query processor I developed can process keywords without the need for Boolean queries (AND, OR, NOT etc). I’ve created a subroutine ( get\_Query(q1, df) ), to run my query processor.

Inside this subroutine first I performed pre-processing on the user query (stop words removal, stemming and lower cased), then I converted the query into a vector. After that I calculated the similarity between the user input query and the data stored inside the Incidence matrix using the cosine similarity. And then I simply sorted and printed the articles with highest cosine similarities.

Figure-13 to 15, shows query processor working,

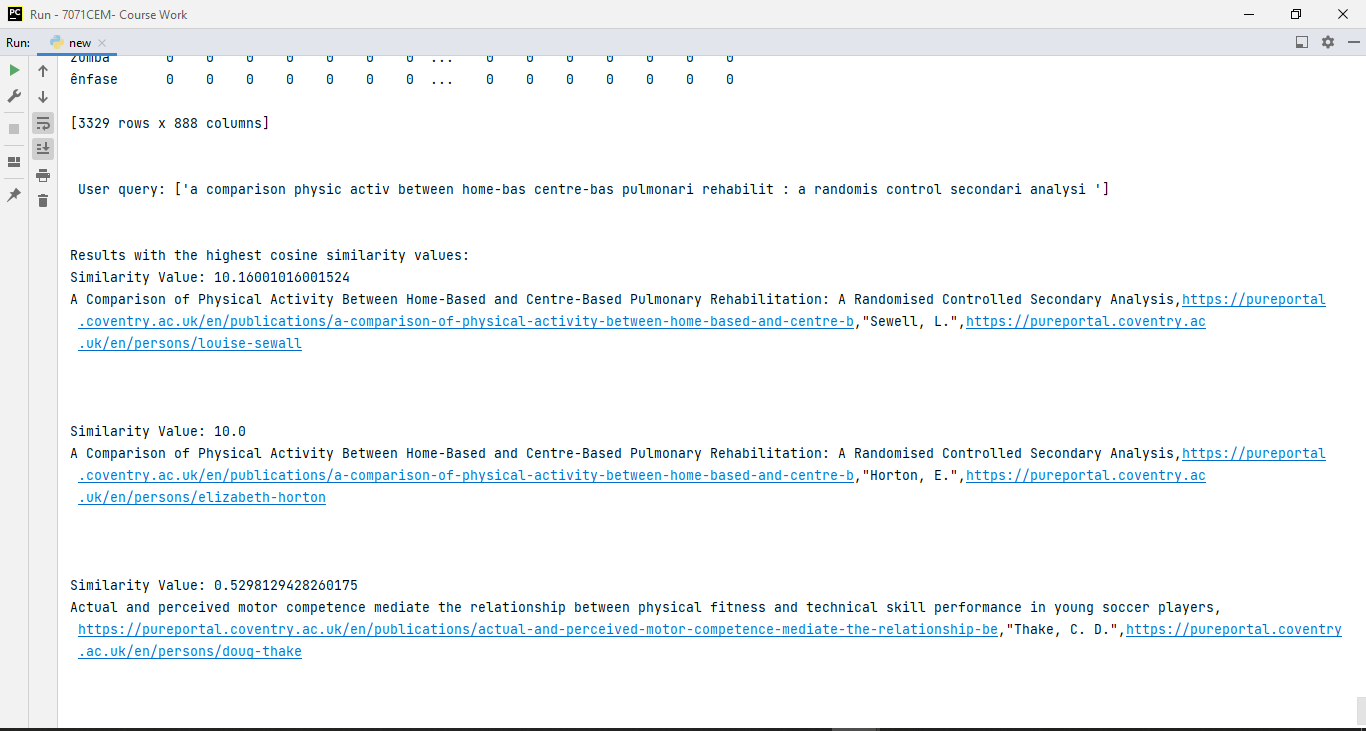


Figure 13: First Query

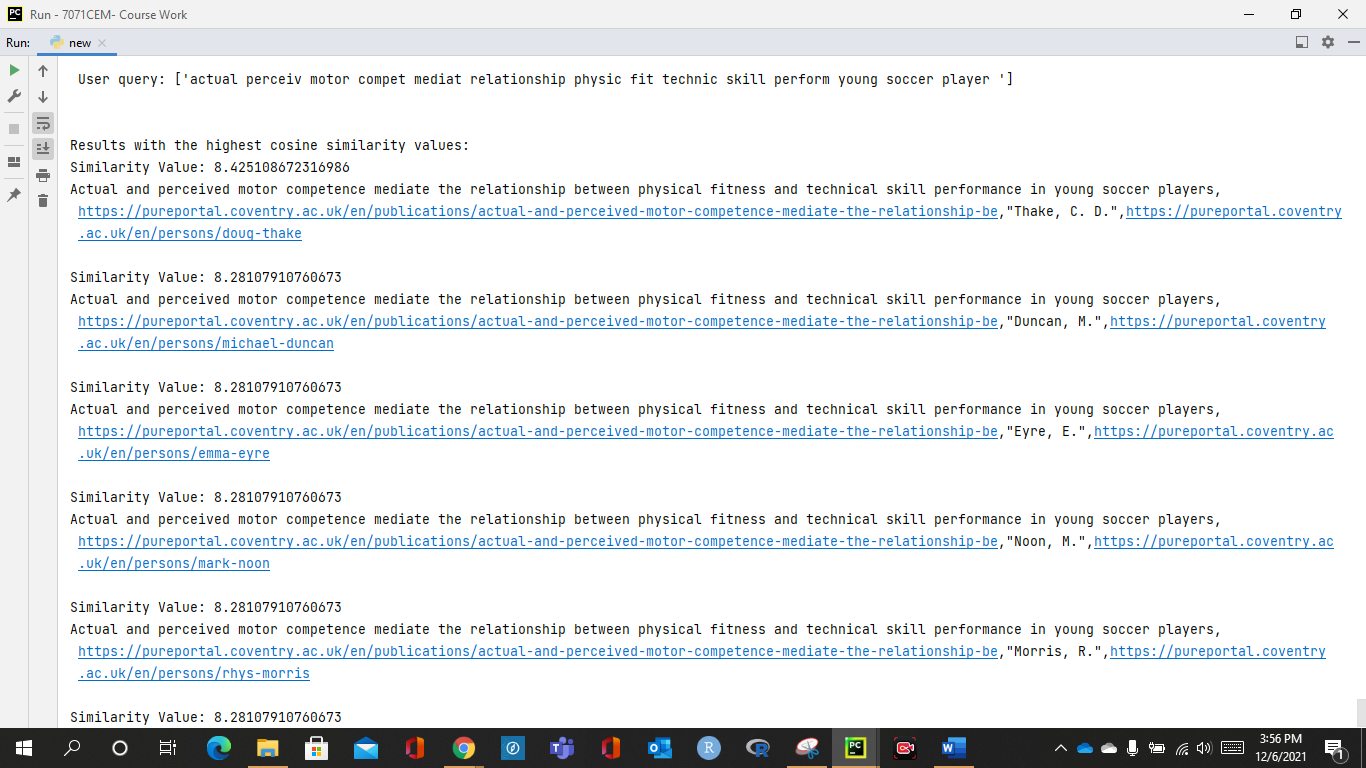


Figure 14 : Second Query

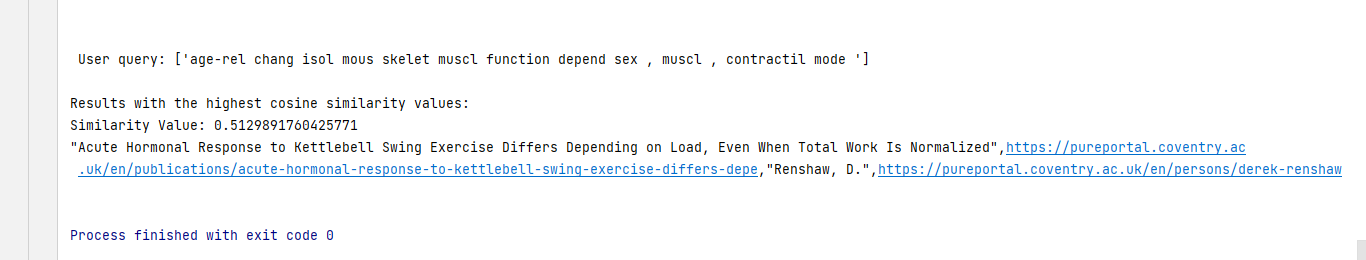


Figure 15: Another Query

# References

1. <https://www.google.com/>
2. <https://www.skyscanner.net/>
3. <https://uk.indeed.com/?from=gnav-homepage>
4. <https://pureportal.coventry.ac.uk/en/organisations/school-of-life-sciences>
5. <https://pureportal.coventry.ac.uk/en/organisations/school-of-life-sciences/publications/>
6. <https://www.bbc.co.uk/news/10628494#mysite>