**CE706 - Information Retrieval 2021**

**Assigment 1**

Student ID: 2003667

**Instructions for running your system (Engineering a Complete System)**

*Include here instructions to run your system and control each individual component. You may include screenshots to clarify.*

To run the system a user should have already installed Jupyter Notebook, ElasticSearch and Kibana.

First of all, a user should launch a Juper Notebook server using Jupyter Notebook (Conda). In the result, a user will see a window with similar information as it is shown on the figure 1.

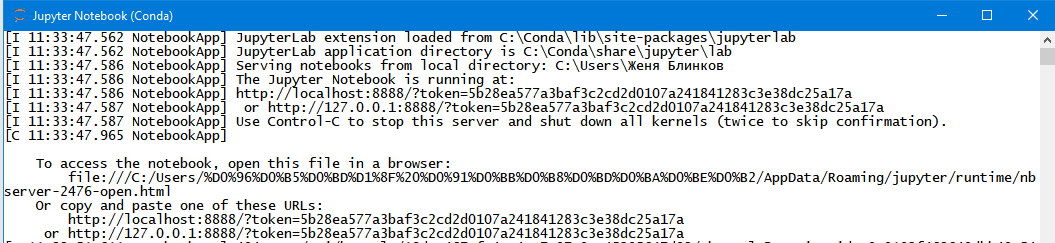


Figure 1: Screenshot of a Jupyter Notebook server launching window.

Having copied and pasted one of those URLs from the figure 2 into a web-browser, a user should open an Information\_Retrieval\_Assignment\_1\_2003667.ipynb file. In the result, the following web-page, that is shown on the figure 2, will be opened.

Although, alternatively, if a user does not have a Jupyter Notebook, he can open it on Google Collaboratory, or just open a provided .py file. In the last case, a further sequence of actions with the programme remains the same

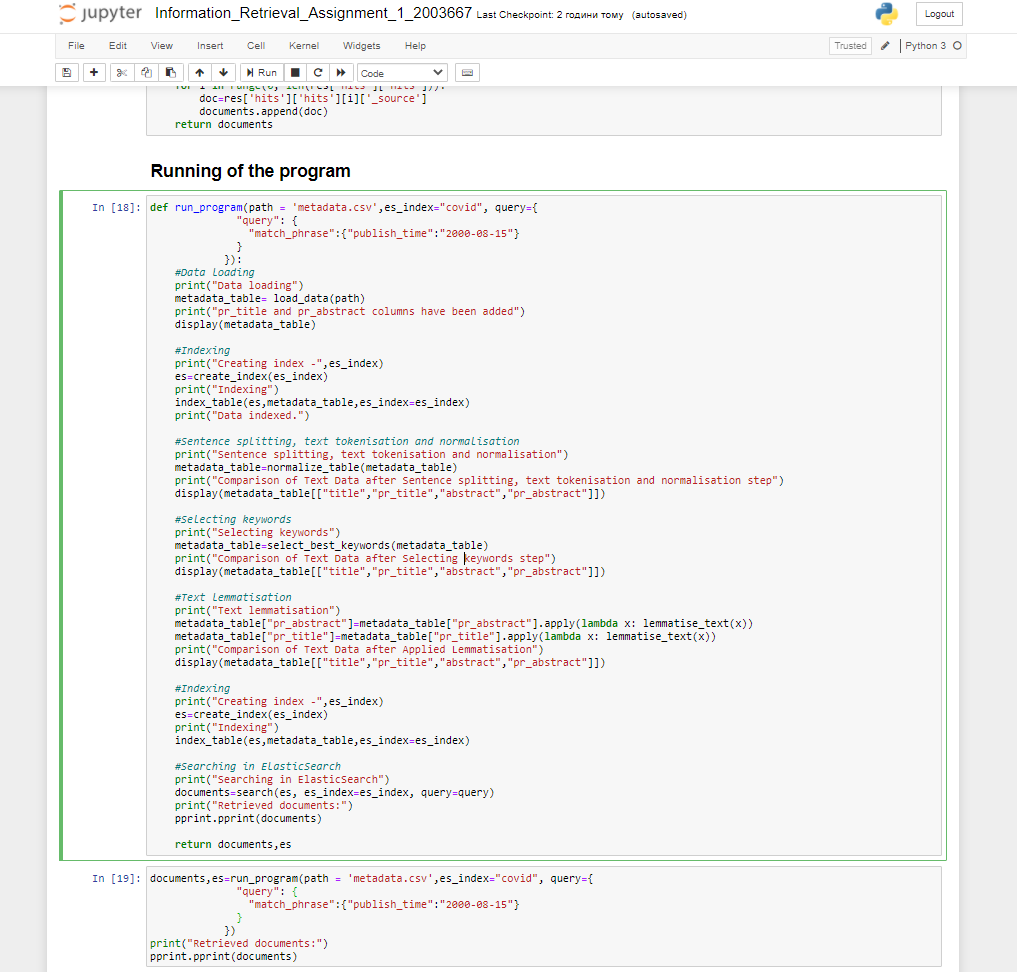


Figure 2: Screenshot of the program-running part of the code for the assignment 1.

According to the figure above, a user can choose which components he would like to run by including or excluding certain parts of the code in the function “run\_program”. An example of a desired sequence of steps can be seen on the figure 3 below.



Figure 3: Screenshot of the program-running part of the code for the assignment 1 with excluded preprocessing steps.

Before running the program, a user should open a command line window to launch ElasticSearch engine, since indexing and searching parts will be unavailable. It can be done using a command “.\elasticsearch.bat” as it is shown on the figure 4.

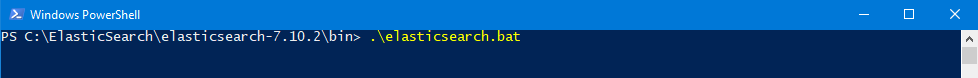


Figure 4: A screenshot of a process of launching ElasticSearch Engine in Windows PowerShell.

As the result, a user should obtain a following log information regarding the successful launch of the ElasticSearch engine. As it can be seen on the figure 5, to get an access to the server, a user can use an IP address, that is 127.0.0.1:9200.

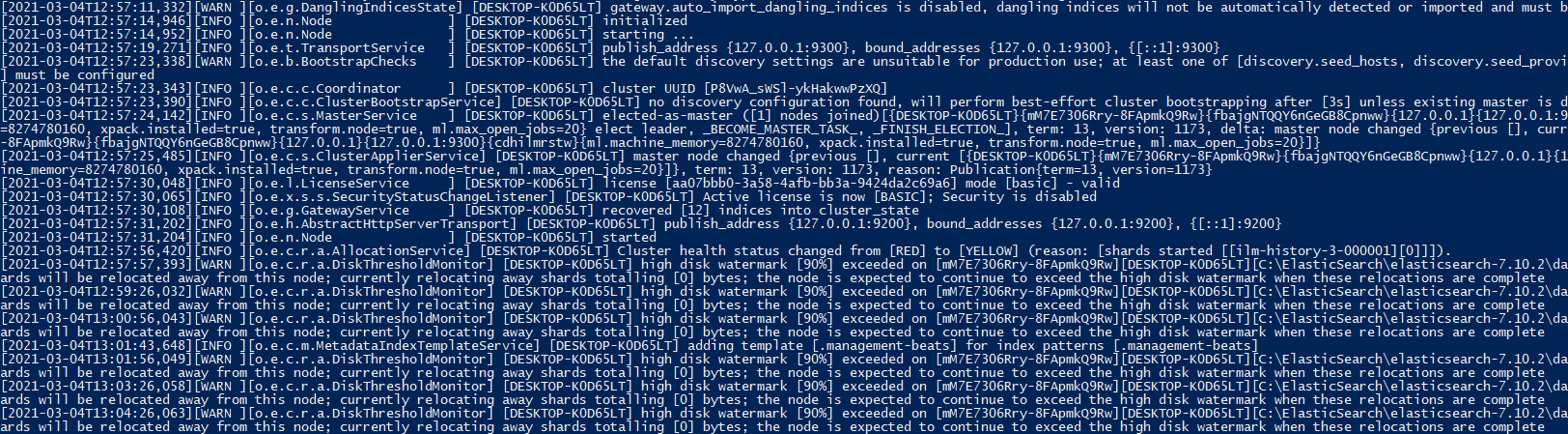


Figure 5: A screenshot of logs on the sucessfully launched the ElasticSearch engine.

Then, a user has to launch a Kibana to be able to interact with ElasticSearch using a convenient GUI. To run it, he should open a Windows command line and enter the Administrator mode. Having done that, the next step is to enter a directory that contains the Kibana .bat file. It is clearly shown on the figure 6.

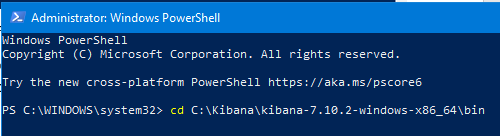


Figure 6: Screenshot of a process of changing directory to access the Kibana .bat file.

Eventually, having entered “.\kibana.bat”, he should obtain following logs and find the address of the Kibana server, as it is shown on figures 7,8.

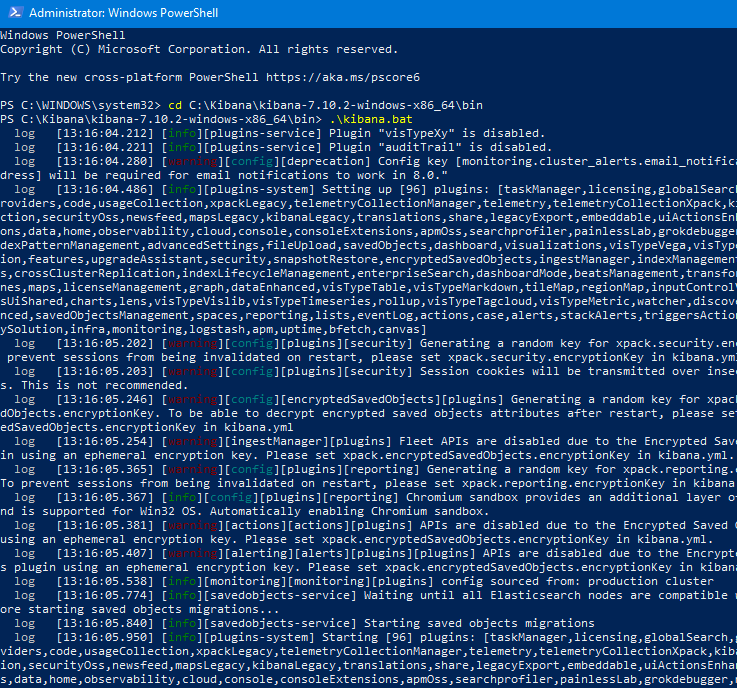


Figure 7: A screenshot of logs on the sucessfully launched the Kibana server.

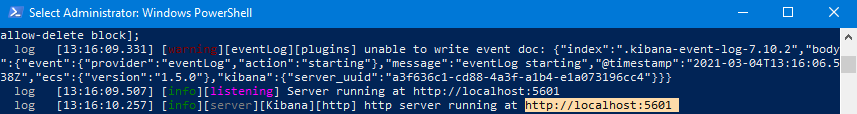


Figure 8: A screenshot of an address of the Kibana server.

Having entered the found web-address, a user should get a following Kibana web-page, as it can be seen on the figure 9.

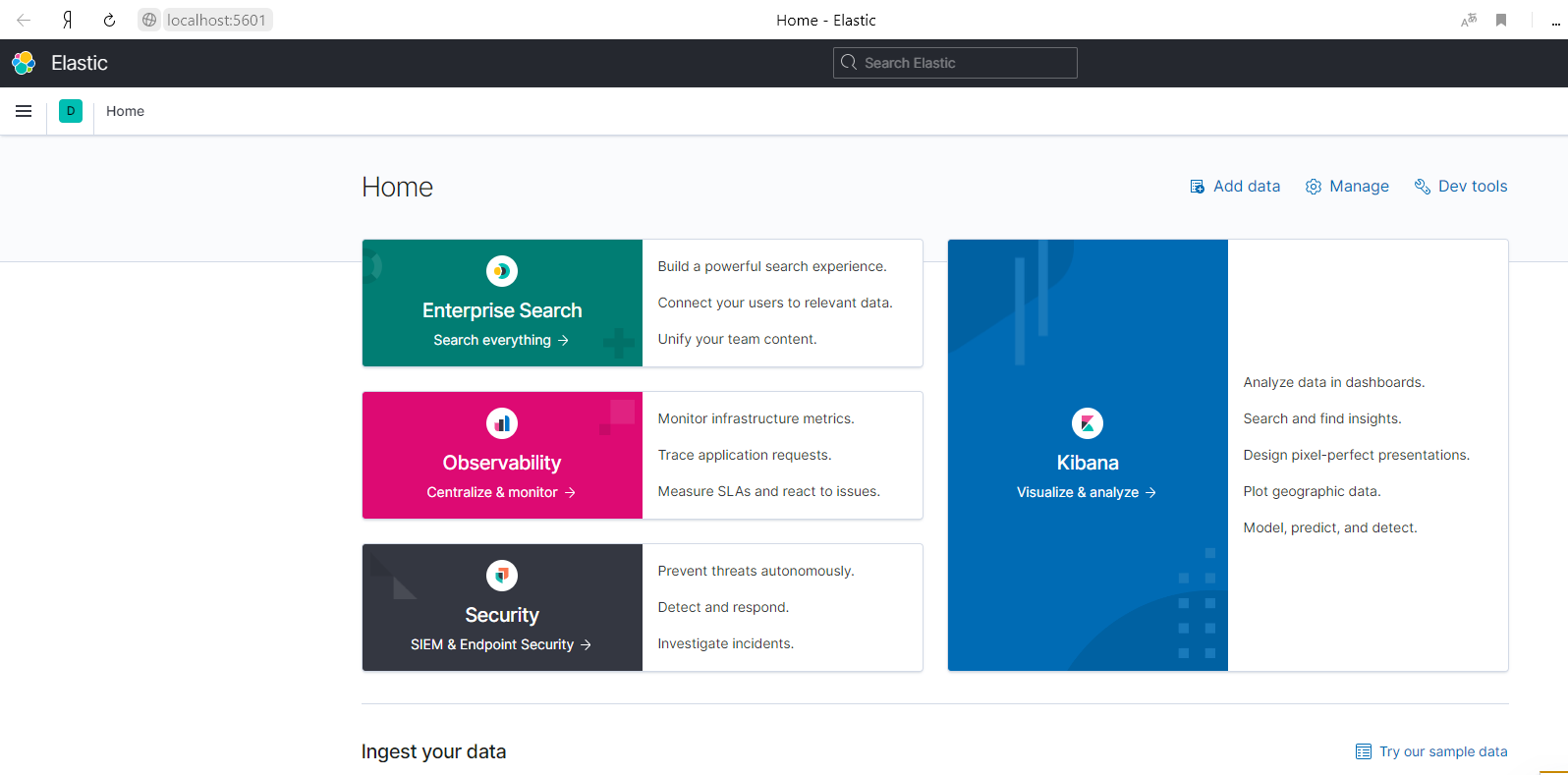


Figure 9: A screenshot of the Kibana GUI.

After making these steps and launching the programme-running part of the code, the implemented search system with back-end module – ElasticSearch and GUI – Kibana is ready for performing search queries.

The search can be performed in two ways:

* Searching by the Python programme
* Searching in Kibana

Searching by the Python programme can be performed when a dataset was uploaded and indexed in ElasticSearch. Usually, data are searched in the bottom of the function ”run\_program”, but, additionally, when the function “run\_program” has done its job, they can be searched by running a code part where the function “search” is called.

A couple of such search examples are depicted on figures. 10,11.



Figure 10: A first example of performing a search by the function “search”.

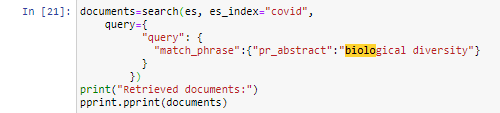


Figure 11: A second example of performing a search by the function “search”.

Searching in Kibana GUI, as it is shown on the figure 12, can be performed when a dataset was uploaded and indexed in ElasticSearch.

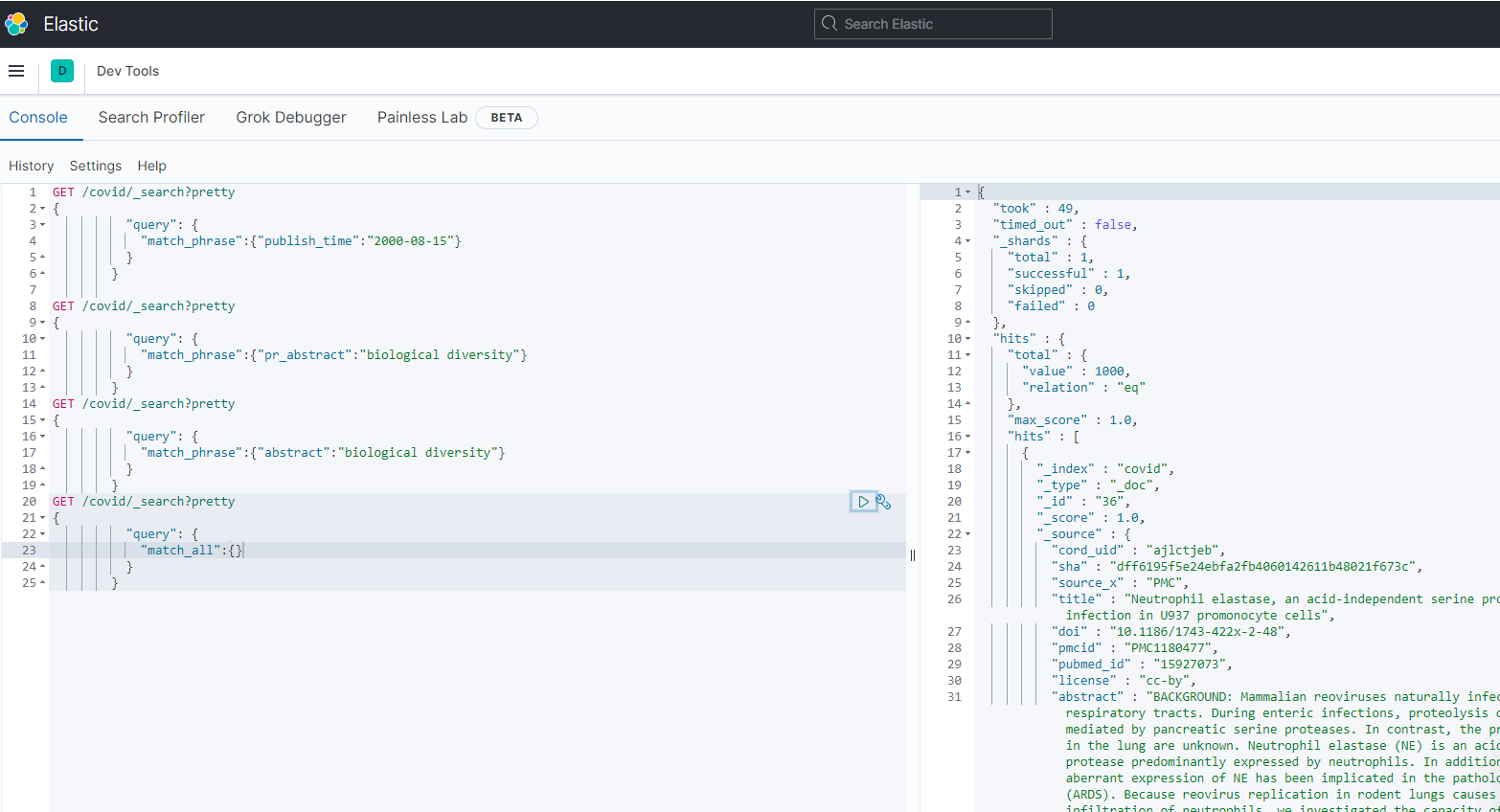


Figure 12: Running search examples in the Kibana GUI.

**Indexing**

*Include here the details of how you download your dataset and index it including any issue that you had and how did you face it. Explain which documents have you selected for your experiments.. You may include screenshots to clarify.*

To work on the assignment 1 there was previously downloaded a CORD-19 dataset that contains thousands of scientific articles dedicated to a research of methods for fight against the COVID-19 disease. Since, the archive contains a massive amount of articles, it was rather hard and took much of time to download and extract it. On figures 13,14 below there is a shown description of this corpus, its structure and a volume of data.

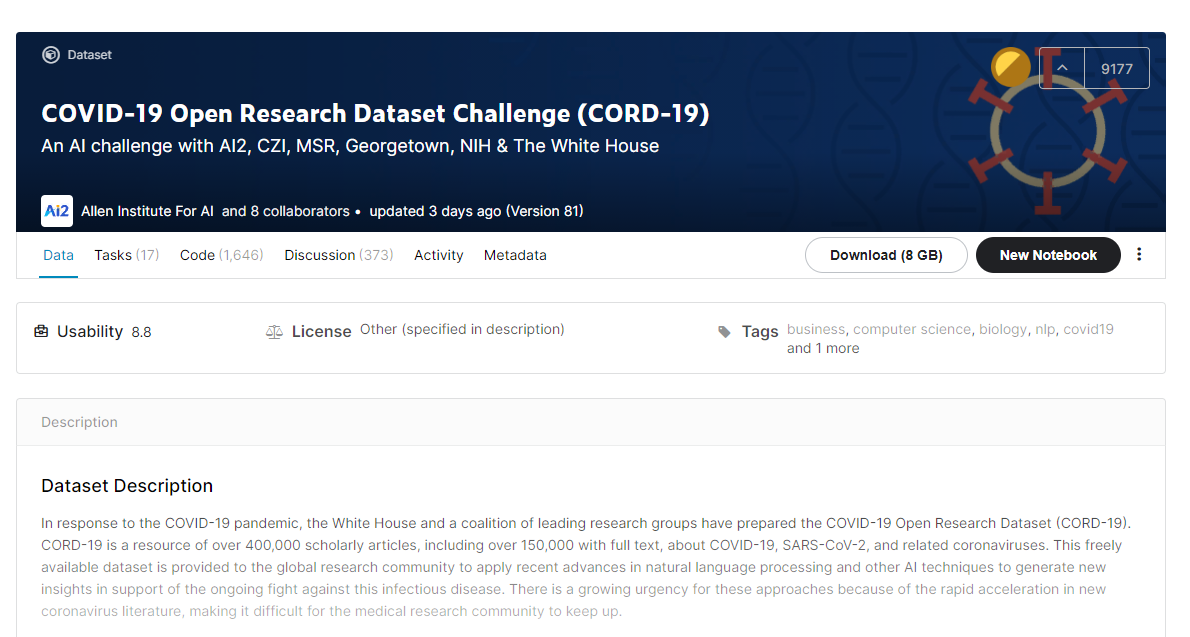


Figure 13: A description of the CORD-19 purpose from its source on the Kaggle website.

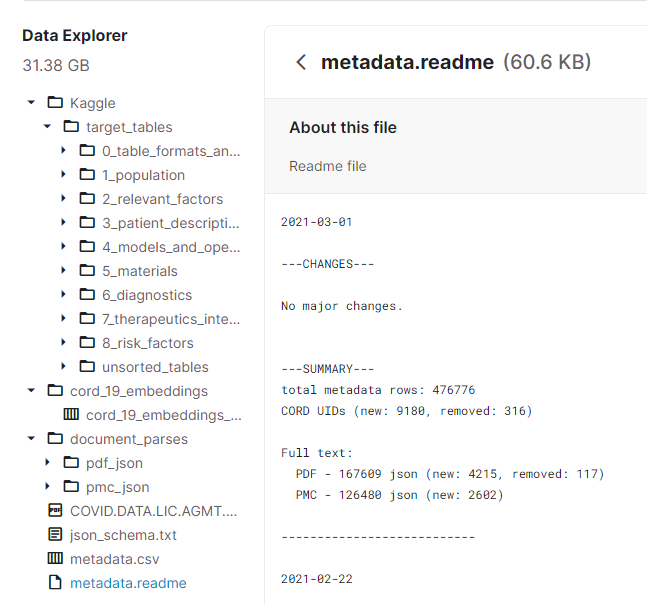


Figure 14: A description of the CORD-19 structure on the Kaggle website.

Having obtained all required information, there was loaded a sample of first 1000 articles from the metadata.csv file using a pyspark. This is a Python library that allows developers to load and structure data into dataframes providing a high level of efficiency and performance.

In the result, the obtained spreadsheet has a following view that can be seen on the figure 15.

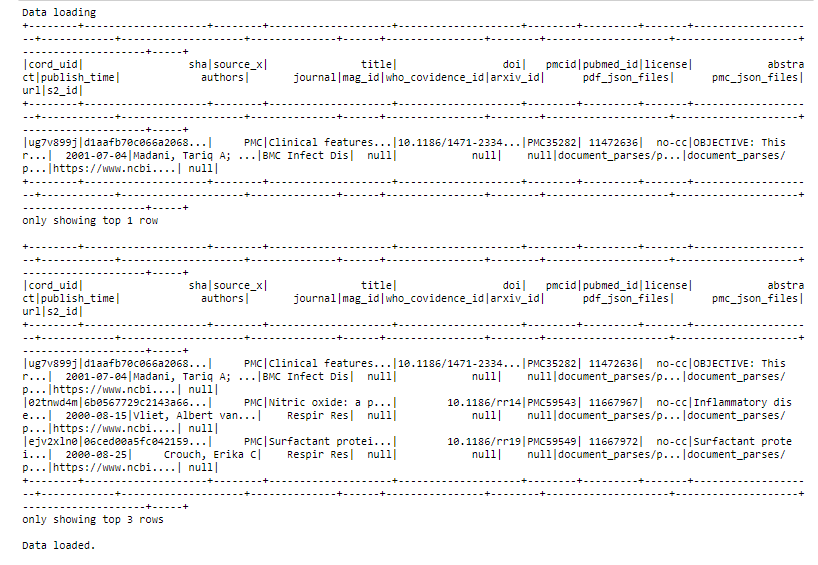


Figure 15: A raw view of first 3 rows from the metadata.csv table.

As it is depicted on the figure above, data represented there have a rather corrupted view. Therefore, there was made a decision to fix this issue by converting it to a Pandas dataframe. This resulted into the much clear representation on the figure 16 below.

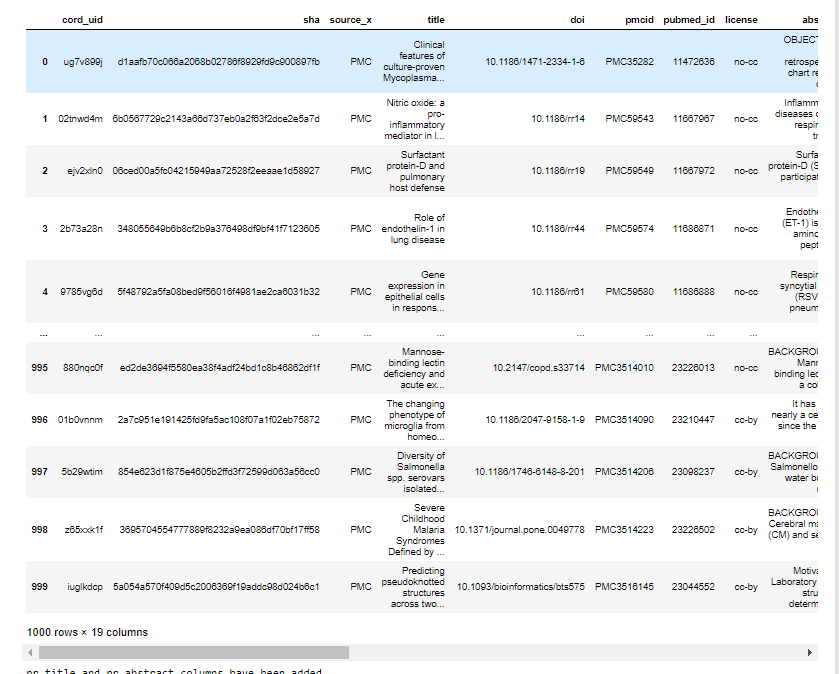


Figure 16: A smooth view of the metadata.csv table content.

On the figure 17 below a content of the function “load\_data” is clearly represented.



Figure 17: A smooth view of the metadata.csv table content.

To index the data properly there was made a decision to create mappings for each column of the obtained table. These mappings point out what is a type of a certain variable and whether it should be indexed or no.

Also, it is of importance to specify a similarity function for columns that contain preferably text data. The metadata.csv table has two of such columns:

* title
* abstract

Therefore, it is required to assign these fields a “BM-25” similarity function that performs slightly better than “classic” that is also known as TF/IDF similarity function.

On figures 18,19,20,21 there were displayed four parts of the structure of aforementioned mappings for data indexing.



Figure 18: The mappings structure – the first part.



Figure 19: The mappings structure – the second part.

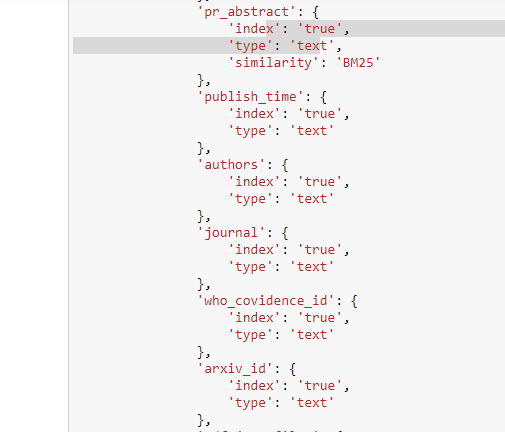


Figure 20: The mappings structure – the third part.



Figure 21: The mappings structure – the fourth part.

As it could be noticed, the mappings structure has two extra fields:

* “pr\_title”
* “pr\_abstract”

These two fields are copies of original ones and are to be processed to present a difference between search results on two versions of a text.

Eventually, when index was created based on those mappings, the data can be uploaded to ElasticSearch using a function that is represented on the figure 22 below.

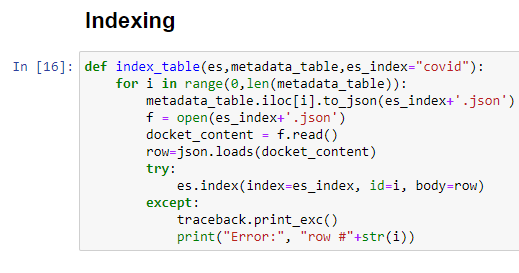


Figure 22: A function to upload data to ElasticSearch and index them.

On the figure 23 below there is provided a part of the code that is responsible for making “Data loading” and “Indexing” steps. Therefore, a user can decide how to work with these parts.

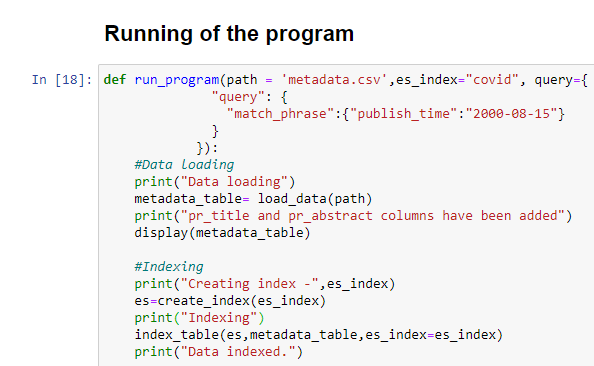


Figure 23: A part of the code to perform “Data loading” and “Indexing” steps

﻿**Sentence Splitting, Tokenization and Normalization**

Sentence splitting went well because it managed to split sentences in the text, providing more efficient keywords selection further

Sentence splitting also went bad because for some sentences where abbreviations occur it split those abbreviations. In general, this problem may not be too critical because there are still the other words to search apart from these ones. For the further programme development, it is worthy finding a certain smart algorithm or a semantic model that could help to address this issue.

Tokenisation went well because by gathering all words occurring in a one sentence it allowed to perform other pre-processing methods such as Lemmatisation or keywords selection, and made it possible to transform the text into a desired form.

Despite this, tokenisation went bad, because it was rather hard to establish right links between two or even several words in the “Selecting keywords” part. Therefore, in the next chapter there will be more detailed explanation how this problem was particularly solved.

Normalisation went fine because it all words in texts became lowercased. As the result, this technique substantially simplified further text pre-processing and keywords selection steps.

On the other hand, a majority of words with a first capital letter merely lost their meaning due to such a rough simplification. Although, this problem was also particularly solved in the next chapter.

On the figure 24, there was depicted a structure for the “TextNormalizer” class. His responsibilities include removing punctuation signs, casting a text to a lowercase, dividing a text into several sentences and words tokenisation. And, on figures 25,26, this class is used.

In the result, on the figure 27 a comparative table of differences between normalised text data and not normalised text data is depicted.

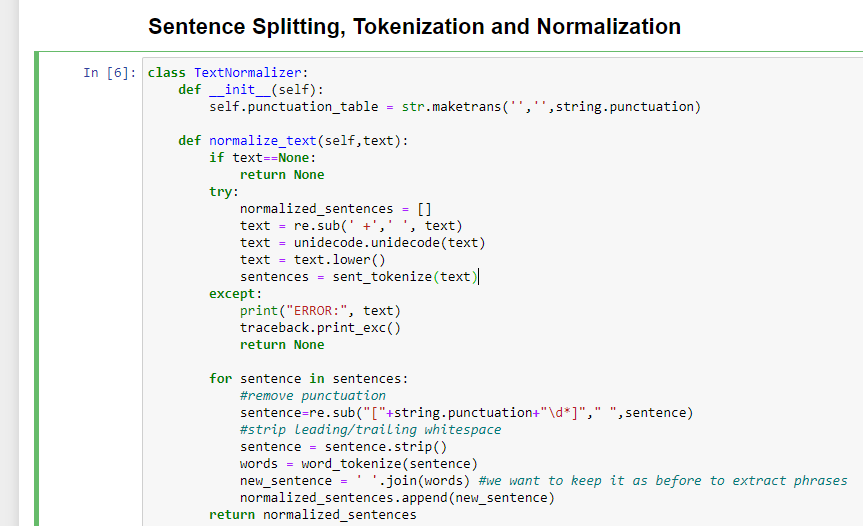


Figure 24: A description of the “TextNormalizer” class

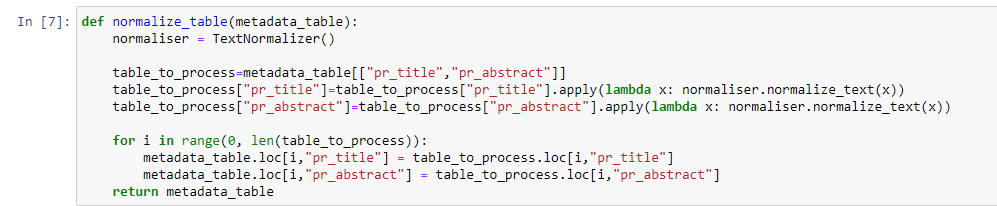


Figure 25: A part of the code to call the “TextNormalizer” class

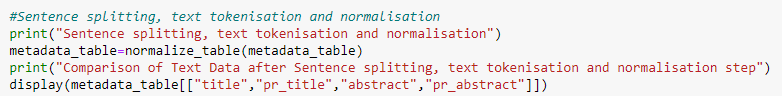


Figure 26: A part of the code to perform the normalisation step

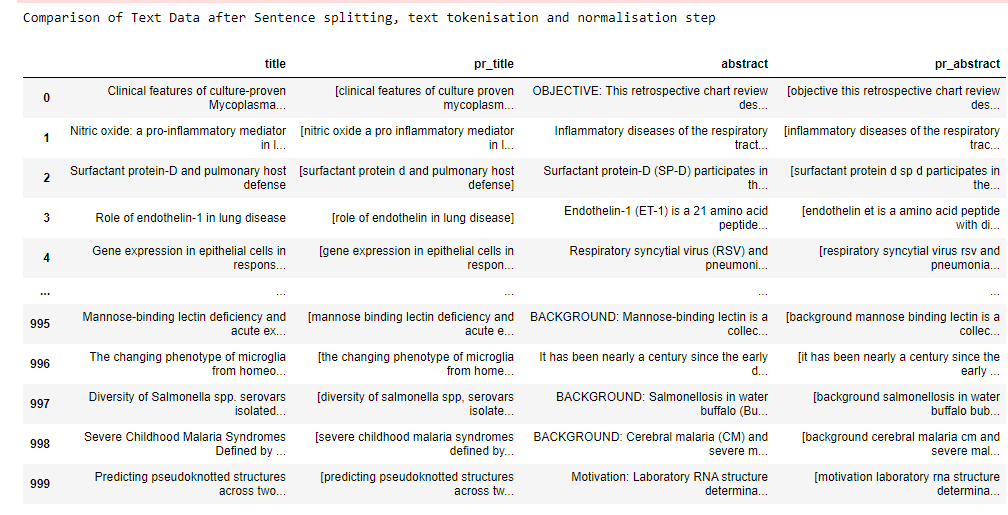


Figure 27: A comparative table of differences between normalised text data and not normalised text data

﻿**Selecting Keywords**

Selecting keywords went well because due to a certain bunch of different implemented keywords selection techniques it managed to preserve some sense characterising an article, and made it much simpler for a search engine to find a relevant document.

On the other hand, it went wrong while using TF/IDF because, a TF/IDF method considers frequency of single words – not phrases. However, this problem was solved by replacing whitespaces between words of phrases with a “\_” sign. Thus, this trick allowed to address that issue and optimise TF/IDF for that kind of words.

Concerning a solution to problems mentioned in the previous chapter, an implemented keywords selection technique – Text Rank made it possible in a certain extent to preserve links between two or even more words. Due to its unusual approach of designing of semantic graphs it allowed to retain a very sense of a considered sentence.

Since this technique evaluates a rank for each word or phrase in a sentence, it was rather convenient to filter those words that make up a noise and retain the most important ones.

Another approach calculates a pointwise mutual information for each bigram or trigram and provides a more efficient keywords selection in comparison with a straightforward bigrams or trigrams frequency distribution, since it considers more important those infrequent words that make sense when they stand next to each other.

In this project, both these methods were implemented and combined to provide much more efficiency for keywords sampling. The idea was to use pros of the Text Rank method to gather only meaningful words or large phrases and then to supplement obtained sentences with keywords gathered by PMI method. Thus, texts should preserve its sense and it would have all important keywords to search.

In addition, a TF/IDF method was also applied to those texts to make them much simpler to search and remove the rest of unimportant words.

On the figure 28 a function, that tokenise sentences in order to remove any stop words, is displayed.

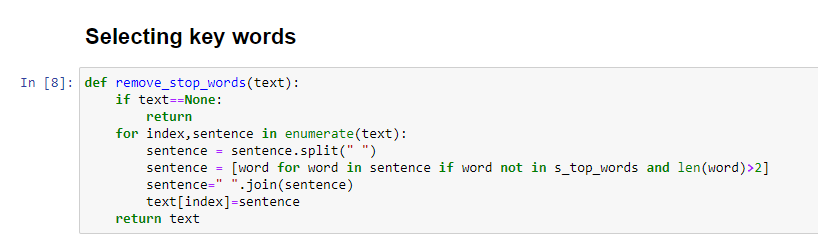


Figure 28: A function that removes stop words.

A function, where the TextRank keywords extraction method was implemented, can be seen on the figure 29. From this, we can notice that TextRank uses English language model to do perform its task.

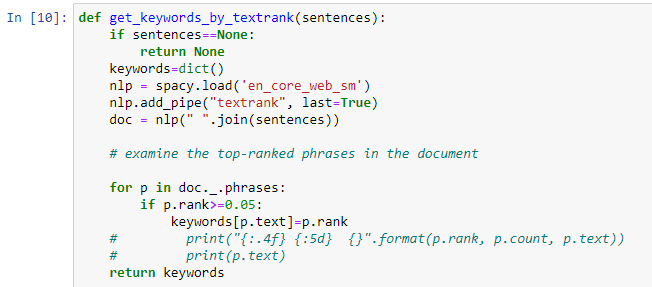


Figure 29: A function of keywords extraction by TextRank.

On the figure 30, it can be seen how results of two methods of keywords extraction merge.

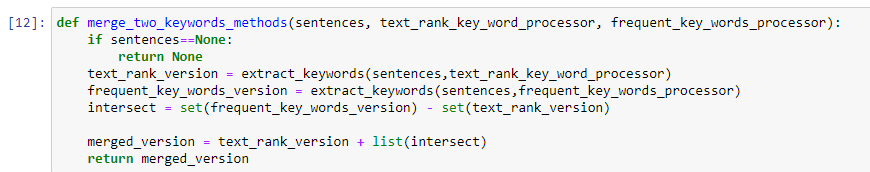


Figure 30: A function that merges results of the PMI method and the TextRank one.

On figures 31,32,33 and 34 we can see the main parts of the function that does a “keywords selection” step.

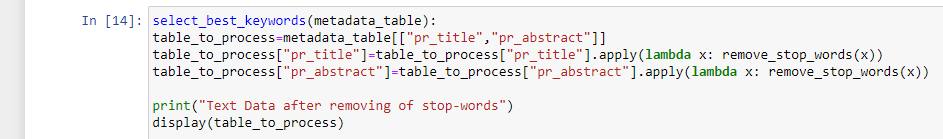


Figure 31: A part where all stop words are removed from the text.

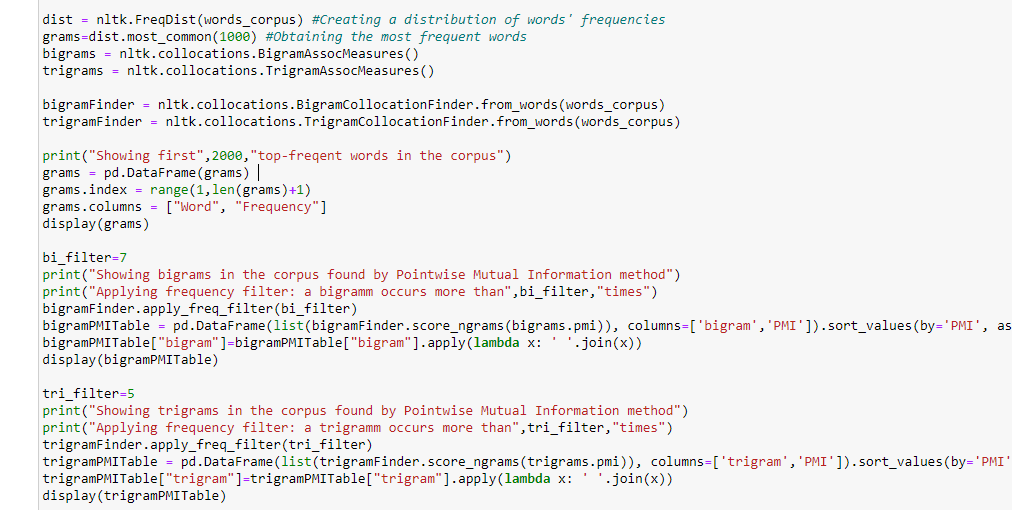


Figure 32: A part where grams, bigrams and trigrams are selected.

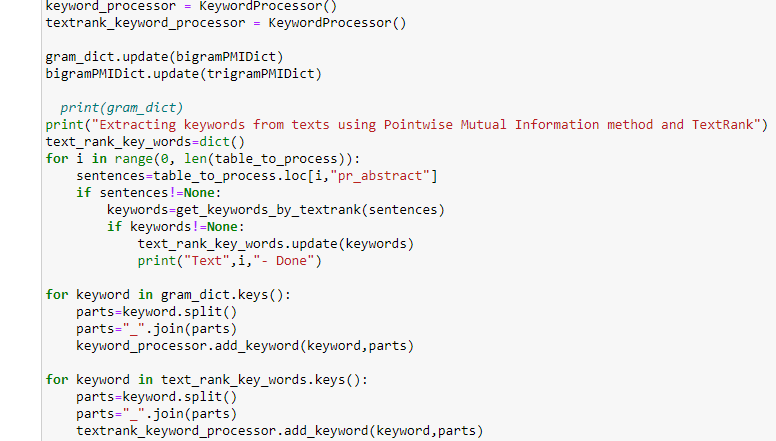


Figure 33: A part where all keywords are extracted from texts by both extracting methods.

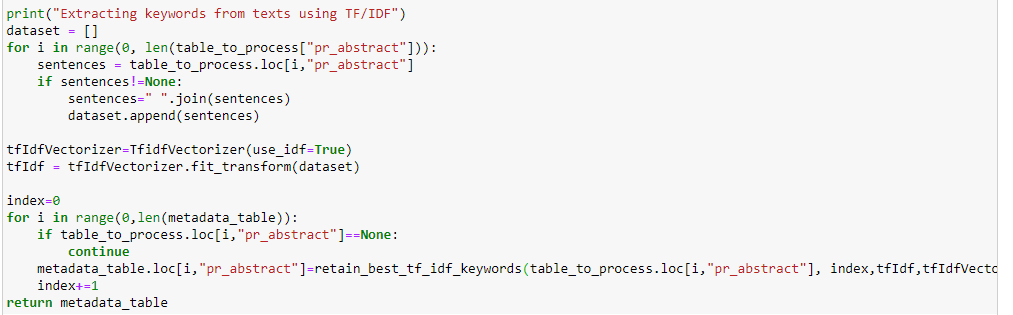


Figure 34: A part where the TF/IDF method is applied to obtained texts.

As it is depicted on the figure 35, the “keywords selection” step is performed.

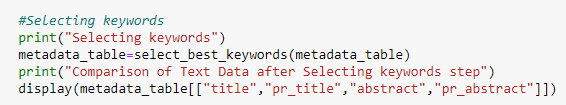


Figure 35: A part of the code in the function “run\_programme” that does the “keywords selection” step.

A table with text data after removing of stop words is shown on the figure 36 below. We can notice that text data do not contain any stop words such as “don’t”, “of” or “this”.

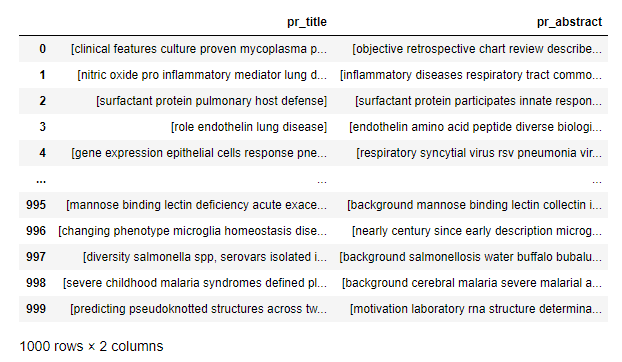


Figure 36: A table that demonstrates a text data without any stop words

On the figure 37 below a comparative table that demonstrates results of keywords selection by PMI and TextRank is depicted. We can notice that some keywords in the column “pr\_abstract” are surrounded by a sign “\_” to keep their structure for TF/IDF.

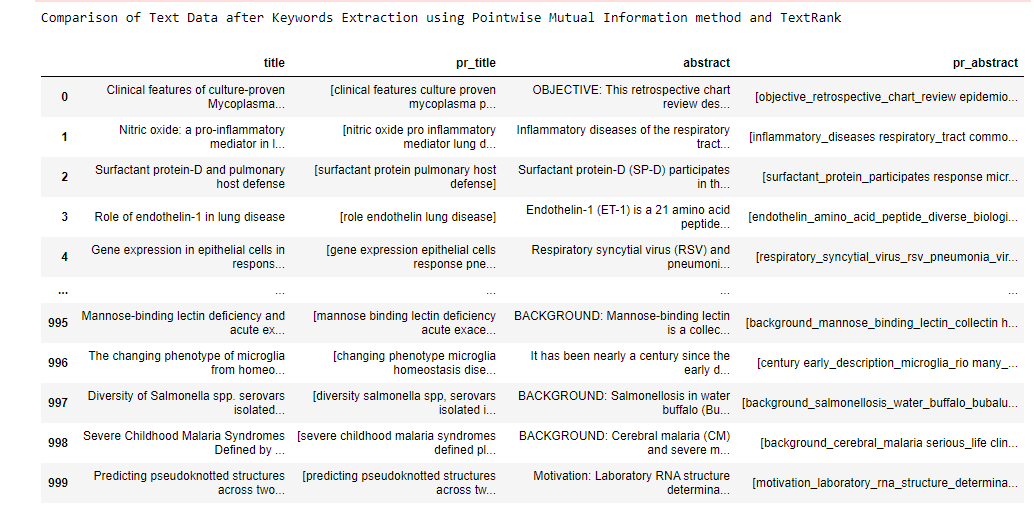


Figure 37: A comparative table of differences between original texts and texts with only keywords.

A result of applying the TF/IDF keywords extraction method is exhibited on the figure 38.   
We can notice that all “\_” signs have already disappeared, because there is no need to keep them for further steps.

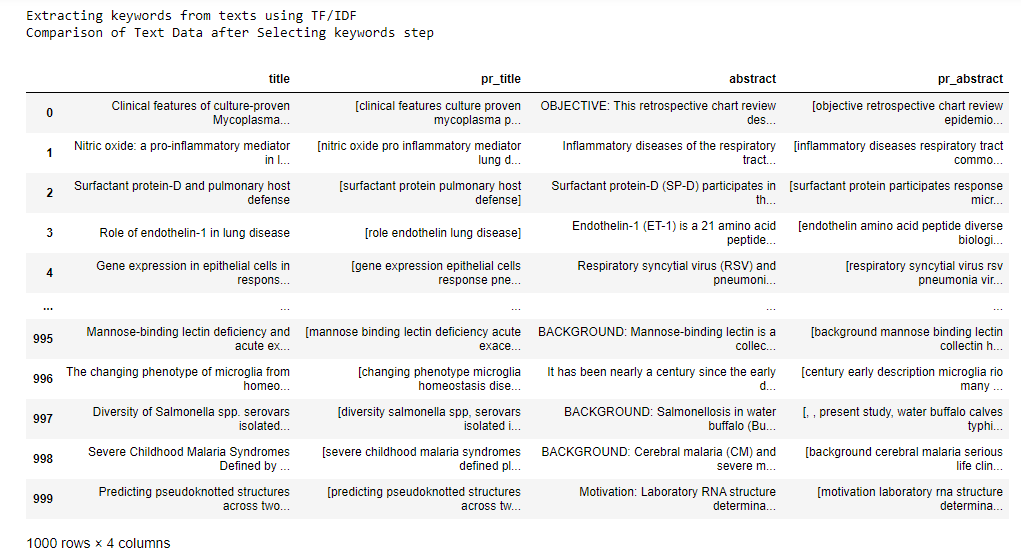


Figure 38: A comparative table of differences between normalised text data and not normalised text data

**Stemming or Morphological Analysis**

*Include here the details of how you did this step including any issue that you had and how did you face it. Present examples for each of the aspects where this step went well. Also include examples for when it when wrong and how you could solve it. You may include screenshots to clarify.*

Initially, implementing of a certain stemming technique was a rather fine decision because it is rather simple and removes word endings for most of words. However, the main problem that occurred in this project is that this method merely cuts a word ending thereby corrupting a word itself and making it unsuitable for a search.

To avoid this problem, there was made a decision to implement a lemmatisation technique. A process of lemmatisation allowed to extract lemmas from words thereby preserving their meaning. Thus, it was a much safer method to process words than a stemming and more efficient.

A function that does lemmatisation of texts can be seen on the figure 39 below. Also, it is called in the code part of the function “run\_program” that is depicted on the figure 40.

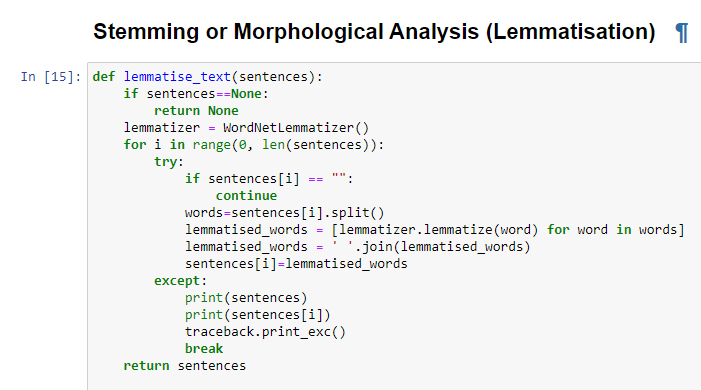


Figure 39: A function that does lemmatisation of texts

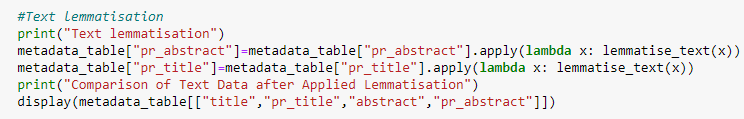


Figure 40: A part of the code of the function “run\_program” that does the “Stemming” step

According to the table 41 below, some changes after applied lemmatisation to texts is clearly demonstrated. For example, a word “features” was transformed to the word “feature”.

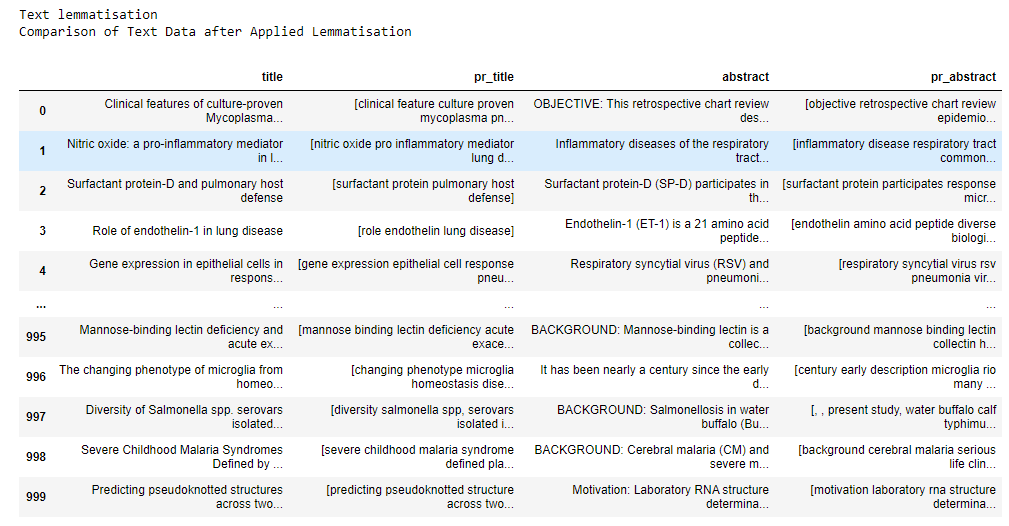


Figure 41: A comparative table of differences between lemmatised text data and not lemmatised text.

**Searching**

*Include here the details of how you did this step including any issue that you had and how did you face it. You may include screenshots to clarify.*

Having an established connection with an ElasticSearch engine, a name of index of a document collection and a query to process, it is quite straightforward and convenient to interact with ElasticSearch using “es.search()” function that is represented on the figure 42 below.

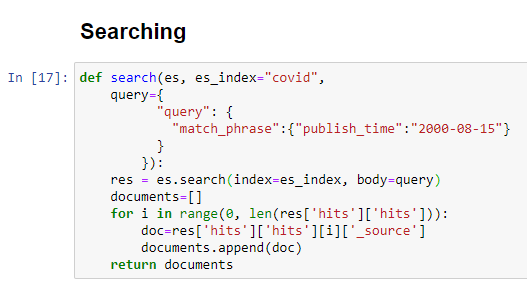


Figure 42: A description of an implemented function “search” that can send queries to an ElasticSearch engine.

Although this, function appears to be trivial, however the main issue was to how to access retrieved documents and represent their content in a smooth and convenient way.

Having considered a JSON format of retrieved documents, there was found a way to extract those documents using properties of a dictionary type in Python. As it is depicted on the figure above, it took several levels of depth to return found documents.

Having done that, a user can perform a search by specifying a desired query. On figures 43,44,45,46 below it can be clearly seen that using Python a user can make up any desired query and obtain a relevant result.

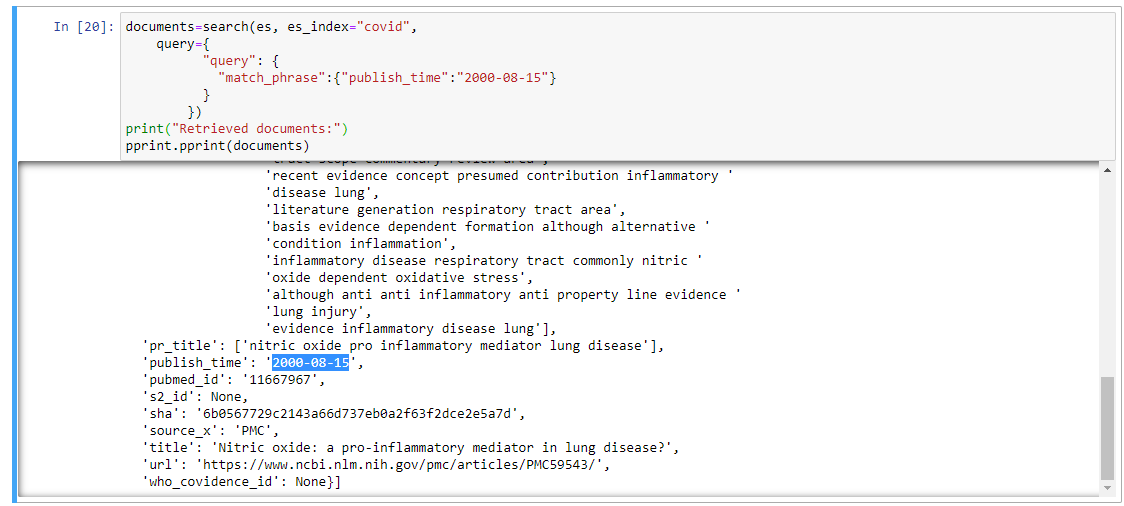


Figure 43 A description of an implemented function “search” that can send queries to an ElasticSearch engine.

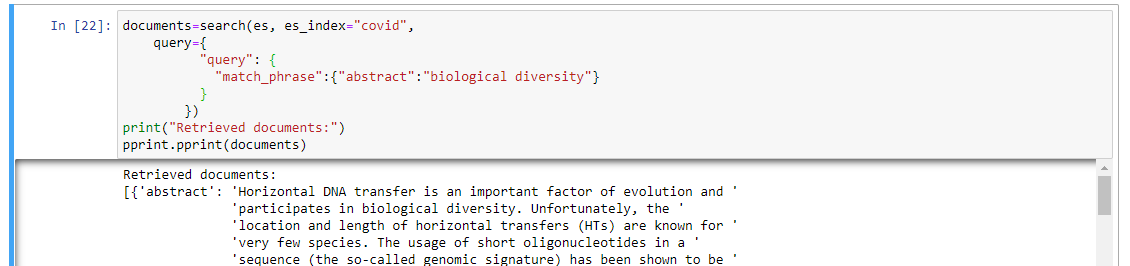


Figure 44: A description of an implemented function “search” that can send queries to an ElasticSearch engine.

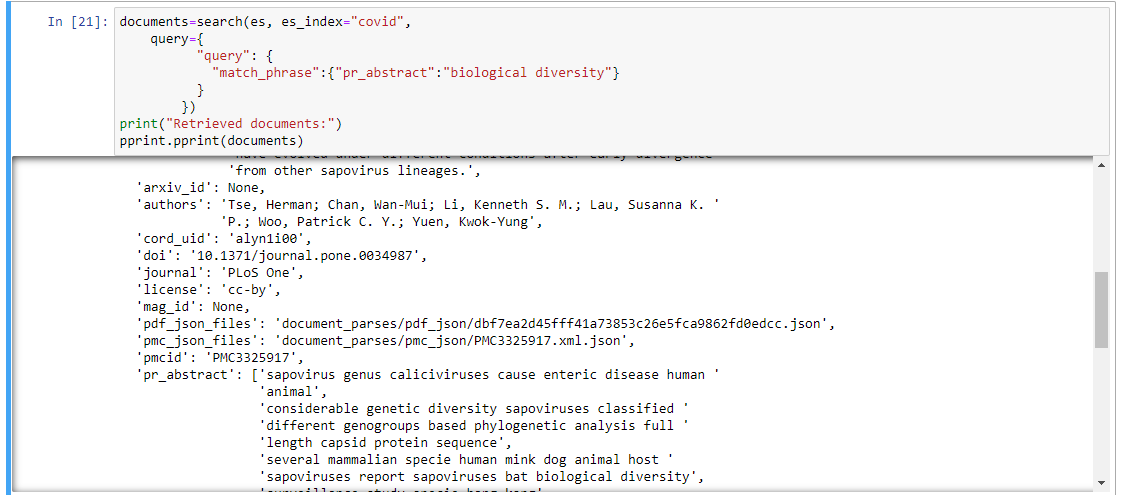


Figure 45: A description of an implemented function “search” that can send queries to an ElasticSearch engine.

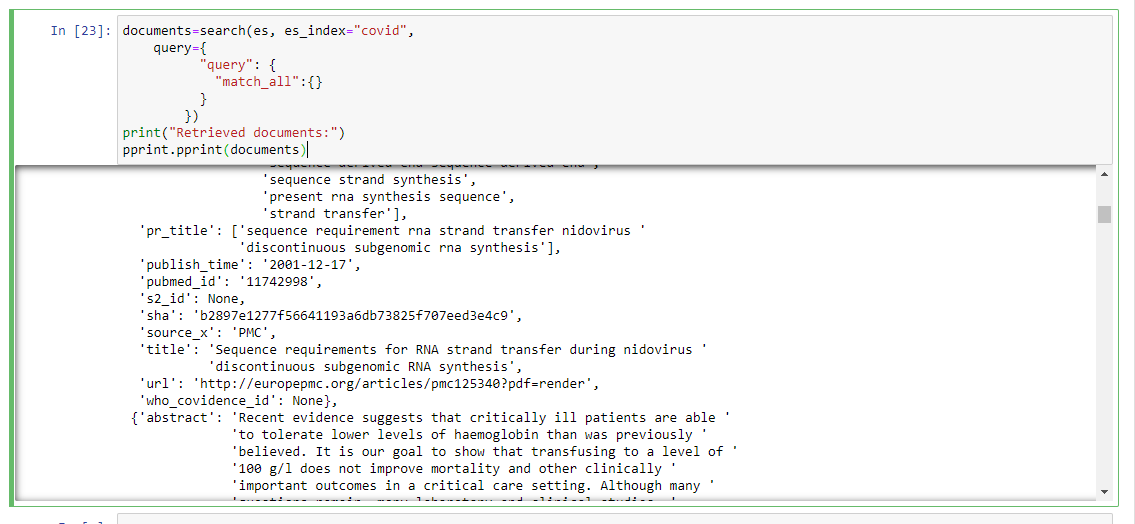


Figure 46: A description of an implemented function “search” that can send queries to an ElasticSearch engine.

Also, a user can open a Kibana to obtain a more convenient and powerful environment to perform a search. An example of work with Kibana can be seen on the figure 47 below.

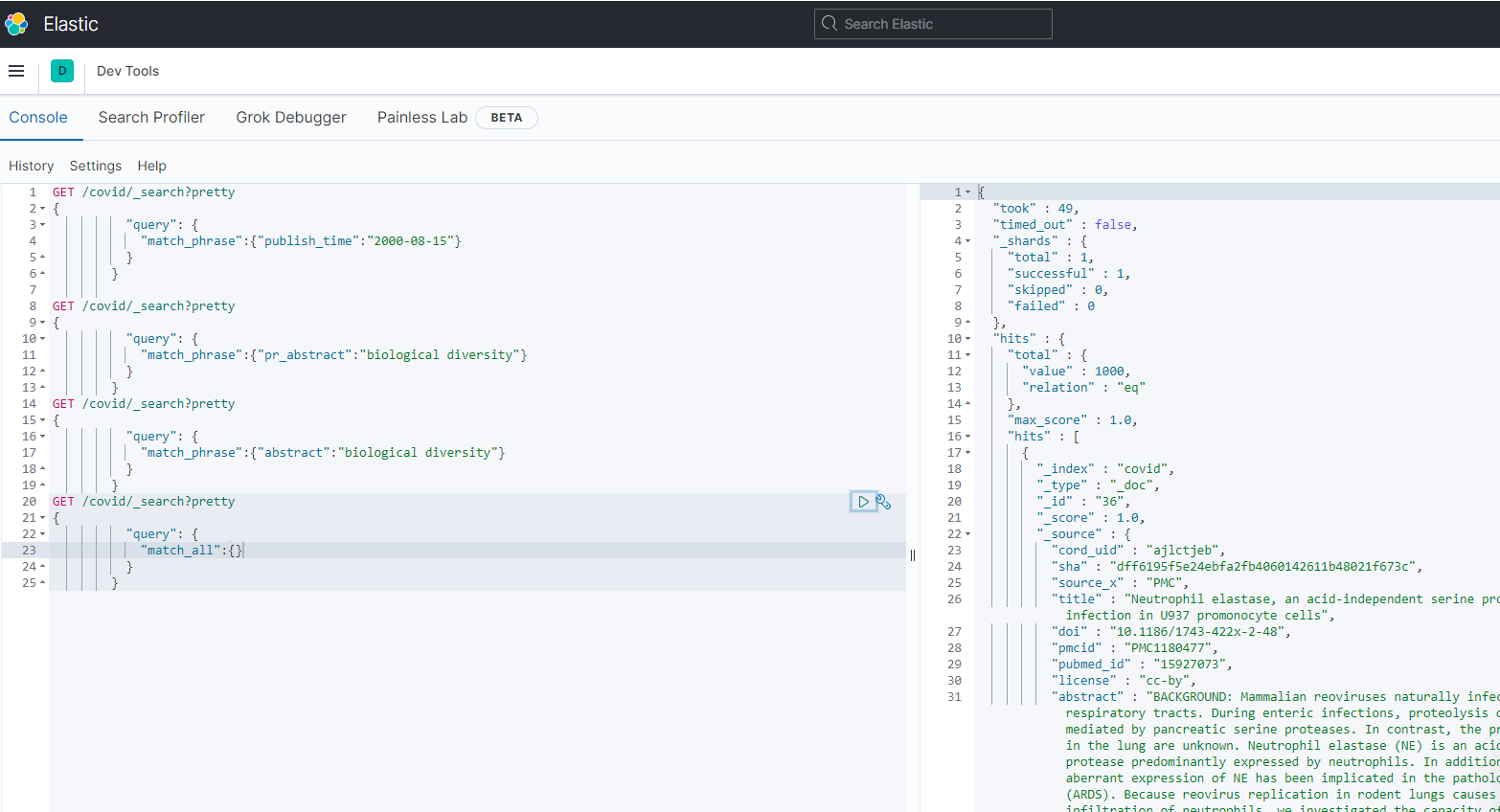


Figure 47: Making up search queries in the Kibana GUI.