Information Retrieval System: Lyric Themes

The Islanders

Avlonitis Ektoras

Lazarević Miloš

Vavakas Alexandros

Table of Contents

[Project Overview 3](#_Toc122119735)

[Data Set Interpretation 4](#_Toc122119736)

[Main Technologies 5](#_Toc122119737)

[Term Frequency – Inverse Document Frequency (TF-IDF) Vectorizer 5](#_Toc122119738)

[From BERT (Bidirectional Encoder Representations from Transformers) => S-BERT 5](#_Toc122119739)

[List of Libraries Used 6](#_Toc122119740)

[Downloading the Data: 6](#_Toc122119741)

[Preprocessing, TF-IDF: 6](#_Toc122119742)

[Processing, BERT: 6](#_Toc122119743)

[Project Walkthrough 7](#_Toc122119744)

[1) Crawling and Scraping 7](#_Toc122119745)

[2) TF-IDF Ranking, Preprocessing and Cosine Similarity 10](#_Toc122119746)

[A) Preprocessing: 10](#_Toc122119747)

[B) Vector space model: 11](#_Toc122119748)

[C) Query preparation: 11](#_Toc122119749)

[D) Cosine similarity: 11](#_Toc122119750)

[3) Using S - BERT: 13](#_Toc122119751)

[4) Results: 15](#_Toc122119752)

[Limitations 16](#_Toc122119753)

[Future Work 17](#_Toc122119754)

[References 18](#_Toc122119755)

# Project Overview

The goal of this project is to retrieve themes from a user inputted query and semantically connect them to lyrical data from songs. Like a search engine, the user would input their desired theme, such as “fighting for love” and would then receive songs with semantically matching lyrics as output. In order to accomplish this, we first download the data from a website, store it locally for preprocessing, after which we proceed to implement different techniques to achieve the desired result. In this document, we will explain these processes step-by-step, outlining the strengths and limitations of each. Finally, we will discuss what can be done as future work to improve this project.

# Data Set Interpretation

In order to crawl the target website correctly and retrieve the target documents, the first step is to inspect the website. We used the browser’s inspect tool to see the HTML elements we will be crawling through in our code. Using the highlighting feature, we identified the elements and their class names, which we later used in the crawler to direct it to the documents we will be scraping. Due to the structure of the website, these elements were often links which led to subsequent pages on the website. This structure is expected, as we are trying to fetch each song from each artist, which can either be grouped alphabetically or by trend. For this project, trend and seasonality are not relevant factors, as we are scraping all data. As a result, we must crawl through different letter (alphabetical) groupings, based on the artists’ name’s initial letter. To visualize this, the crawling process went as follows:

Website => Alphabetical Letter => Artist Name => Song => Lyrics

Fortunately, the lyrical data was most often stored in a single HTML element, which simplified the scraping method as we approached it.

It is important to mention that each website we found had different tolerance for automatic browsing (the crawling process in our case). Essentially, the crawling process involves HTTP requests to the website server to get to the desired page.[[1]](#footnote-1) By making a large amount of these requests, in a relatively short amount of time, the website would block the crawler from proceeding further. Multiple infringements of this would lead to an IP address ban. To get around this, we needed to find a way to throttle our crawler. While this is a valid workaround, it significantly lengthens the time taken to scrape all the target documents. Most applicably, this refers to websites with very low tolerance. In order to get the best results for our project, in a reasonable amount of time, we needed to find a website with the most amount of tolerance.

# Main Technologies

## Term Frequency – Inverse Document Frequency (TF-IDF) Vectorizer

TF-IDF combines two concepts which are useful for our project topic. Term Frequency (TF) and Inverse Document Frequency (IDF). The term frequency is the number of times a word or phrase appears in a document, and the inverse document frequency is a measure of how common the word or phrase is across all documents. In our project, we can use it to determine the relevance of a document to a particular query or topic.[[2]](#footnote-2)

These two techniques are largely important in our project as they directly correlate with the type of query that is being inputted as well as with the lyrics in the documents. It weighs terms based on frequency, creating a vector-space model.

At this point, we can use various methods to preprocess the documents allowing us to prepare them for fine-tuning through S-BERT.

## From BERT (Bidirectional Encoder Representations from Transformers) => S-BERT

S-BERT is a variant of BERT that is fine-tuned on handling sentence pairs using sentence embedding. We can think of this as having two identical BERTs in parallel that share the exact same network weights.

Once we have finished preprocessing with TF-IDF, we needed to proceed to implement word embeddings as the data we had until that point was not sufficient. This is because the vector space model misses semantic and syntactic information. It also assumes that terms are independent and lacks the control of a Boolean model (requiring a term to appear in a document).

Considering that we needed contextual and semantic data, a sentence embedding method appeared to be a better alternative. The S-BERT variant of BERT seemed best suited for this scenario as it has been pre-trained on a large dataset of text. It is based on the transformer architecture, which is a type of neural network that is well-suited to processing sequential data such as text. S-BERT stands for "sentence BERT," as it was specifically designed to process individual sentences and understand their meaning in context.[[3]](#footnote-3)

# List of Libraries Used

## Downloading the Data:

* Urllib
* Requests
* BeautfiulSoup
* Time
* Random
* CSV

## Preprocessing, TF-IDF:

* Pandas
* SKlearn
* Numpy
* NLTK

## Processing, BERT:

* Seaborn
* Matplotlib
* FAISS
* Sentence\_Transformers

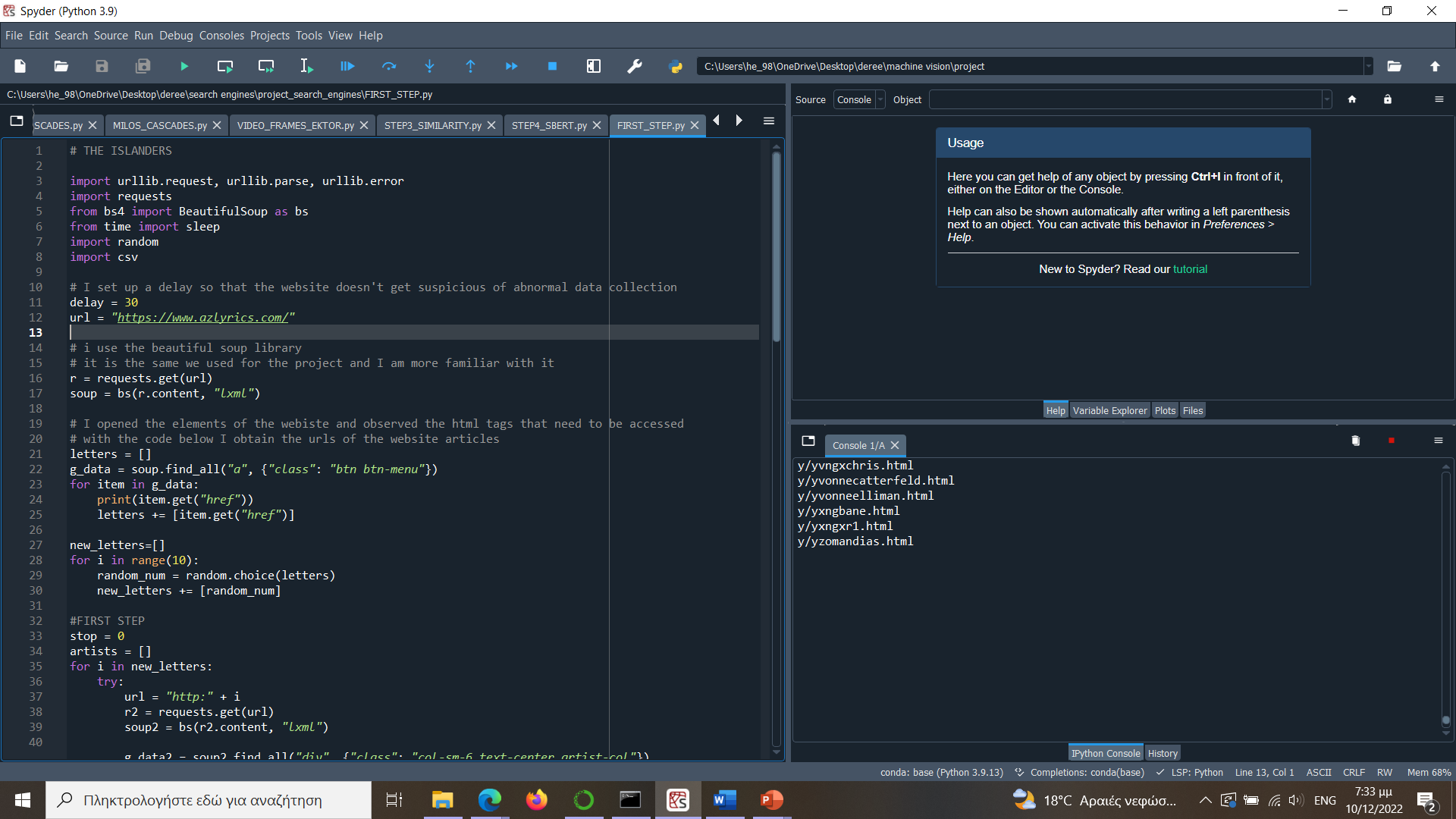
# Project Walkthrough

## Crawling and Scraping

We have selected the website [AZ Lyrics](https://www.azlyrics.com)[[4]](#footnote-4) as our data set. While crawling it, we made numerous requests to the website, because we wanted to retrieve multiple lyrics from songs from various artists.

Since the website can get overused and block us from collecting our data, we tried to slow the speed using a delay, that waits for some seconds after fetching one lyric. Through trial and error and after being banned a few times, we ended up with a delay = 25. That way we managed to collect almost 1200 lyrics from songs from the website.

We need to find the exact tags and code accordingly to take the pointer for scraping out the URLs. For these steps we used the libraries: Beautiful Soup, urrlib and requests. That’s how we pulled data out of HTML and XML files, made HTTP requests to read web pages and parse them.



Next are the steps we took to end up with the lyrics.

First, we created a list called ***letters*** of the scraped URLs for the different letter in the navigation bar of the website that takes us through the artists whose name start with that letter.

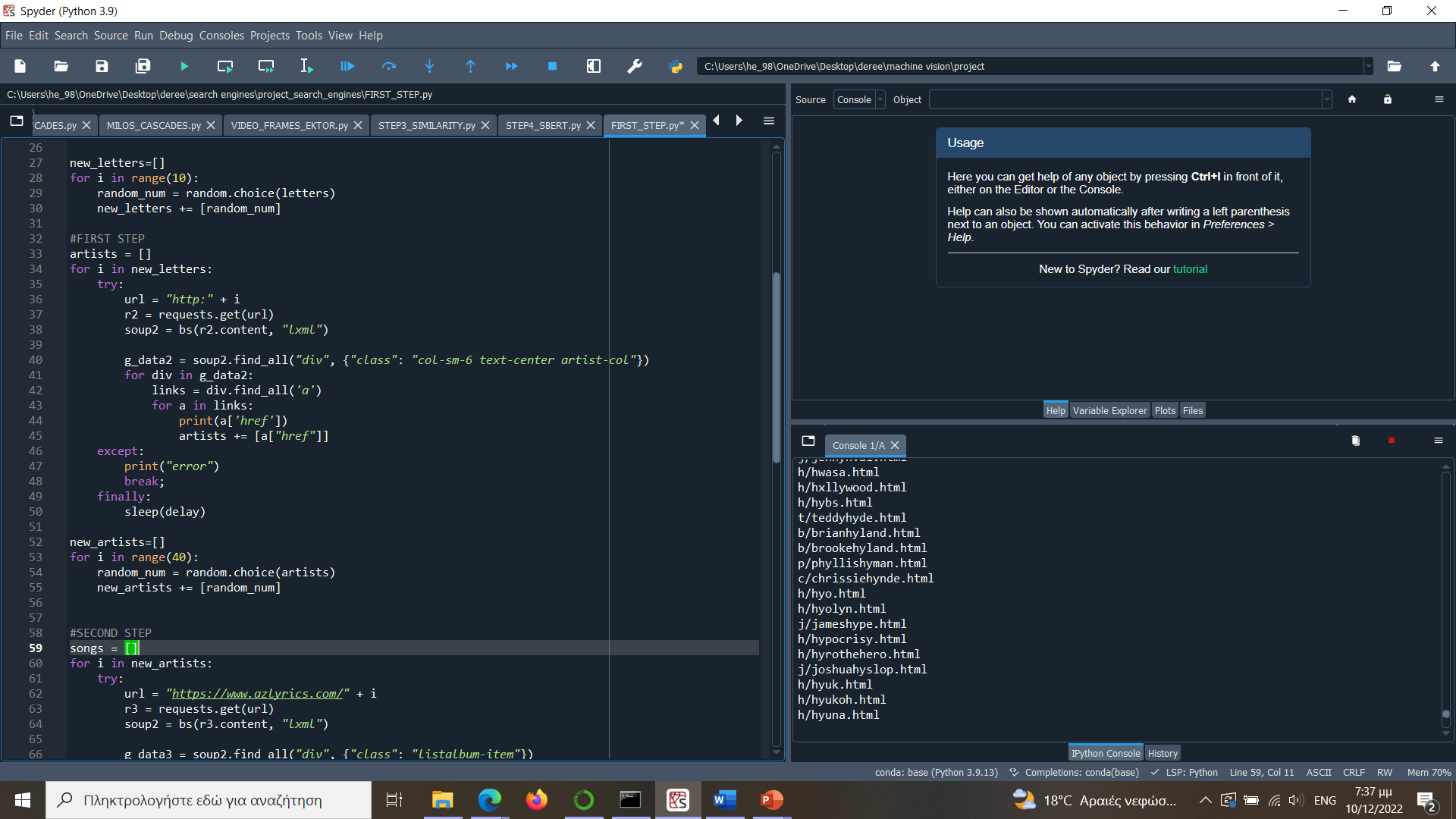


We took randomly 10 URLs from that list and appended them in the *new\_letters* list.

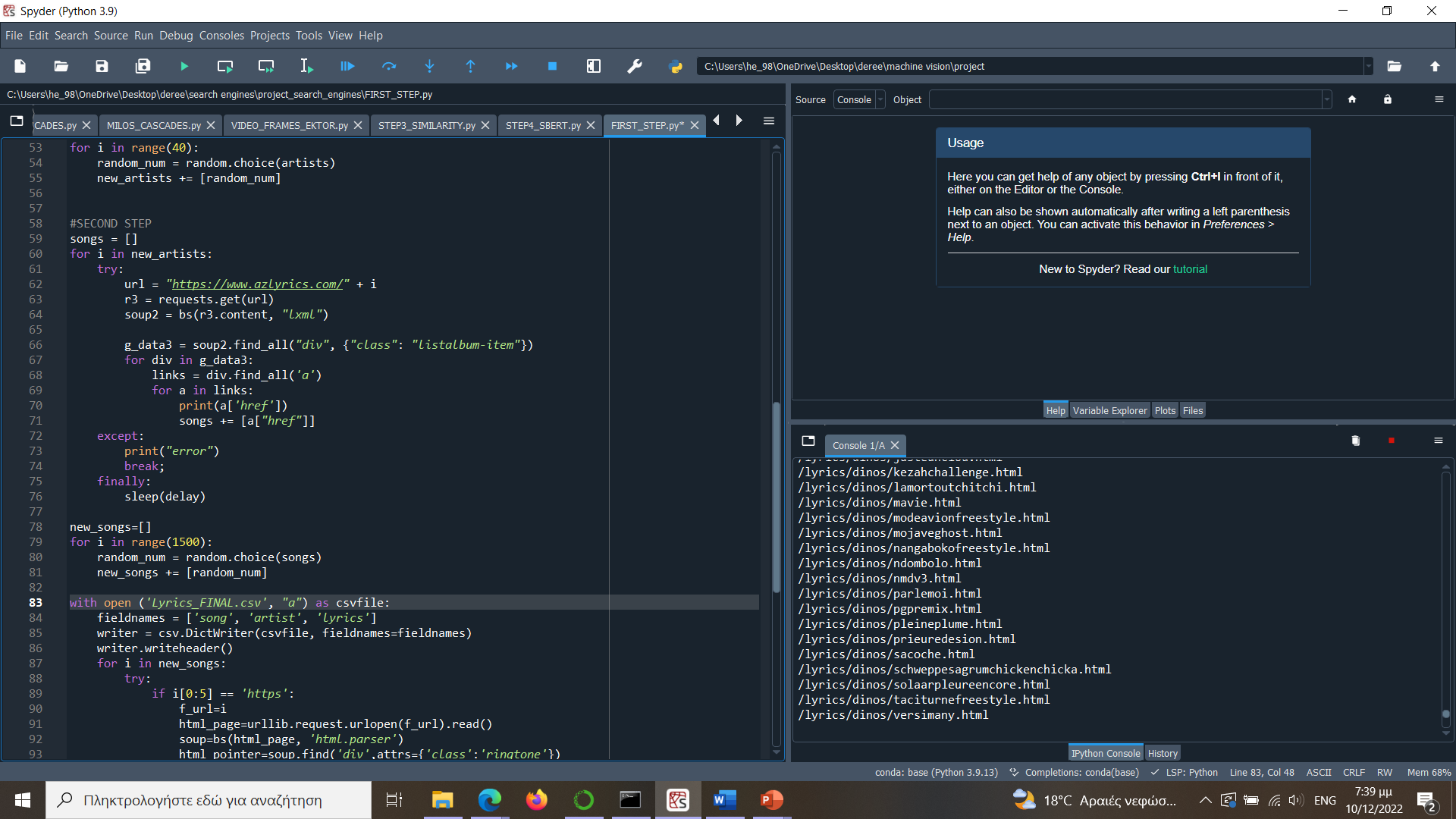
Εικόνα που περιέχει κείμενο, οθόνη, στιγμιότυπο οθόνης, εσωτερικό

Περιγραφή που δημιουργήθηκε αυτόματα

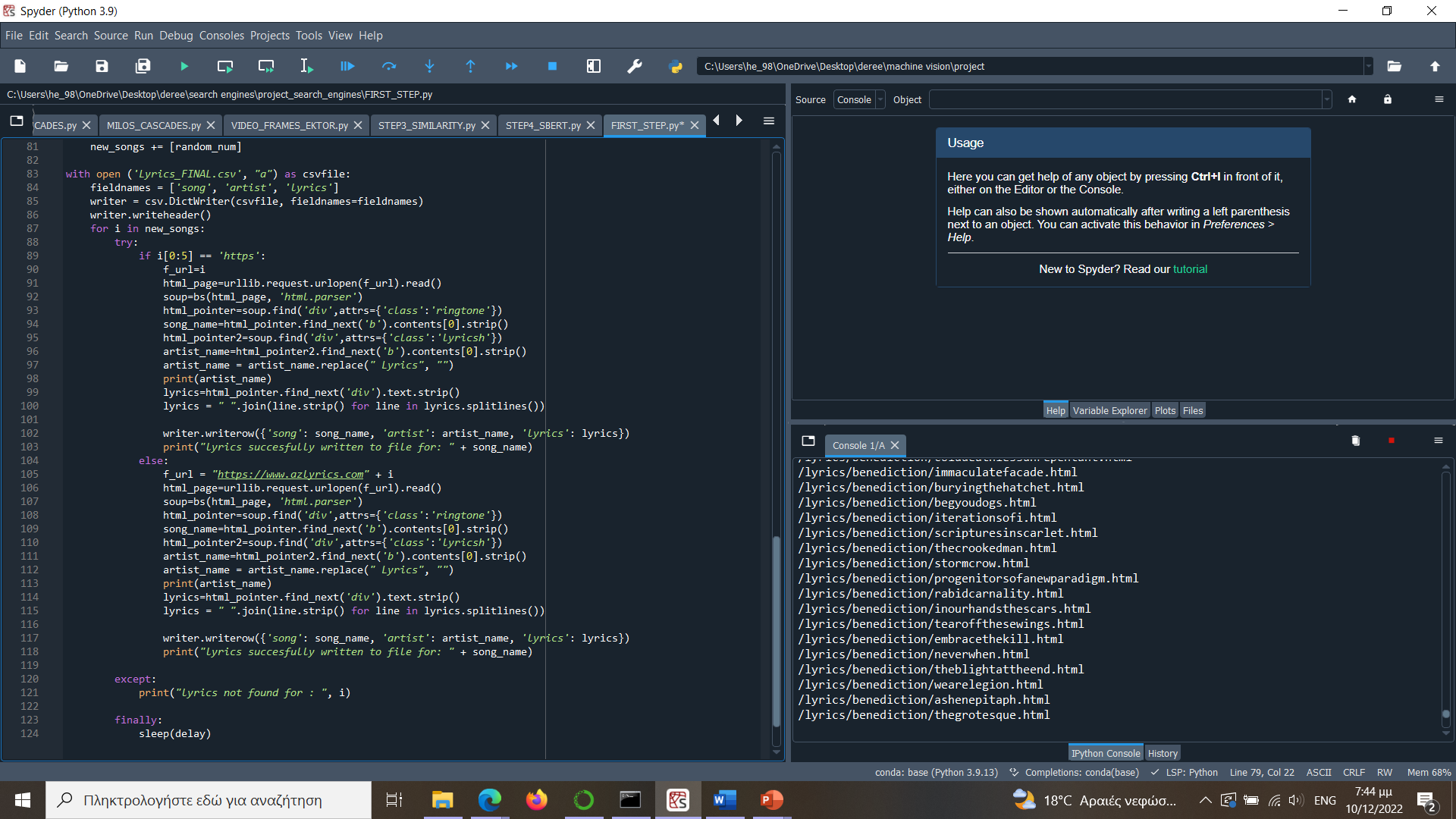
From those 10 URLs (10 letters), we crawled through each artist whose name start with that letter and appended them in a new list called ***artists***. We chose randomly 40 artists from that list and appended them in a list called ***new\_artists***.



From those 40 artists, we scrap through all their songs and store the songs’ URLs in a new list called *songs*. We choose randomly 1500 of those songs and append them in the ***new\_songs*** list.



Then, we use those song URLs to retrieve the lyrics and store them in a csv file. Some of the URLs start with “https”, whilst others start with “/lyrics/…”. Therefore, we insert an if statement in our loop so that both options are examined.



As we run the code, we observe that almost 20% of the songs produced an error and the lyrics were not found. Thus, we successfully managed to collect lyrics for about 1200 songs. This took about 10 hours (songs\*delay = 1500 \*25 seconds ≈ 10,5 hours). These lyrics are appended to a csv filed called ***Lyrics.csv***. The csv file includes the song’s name, the artist, and the lyrics for each song, separated with a comma.

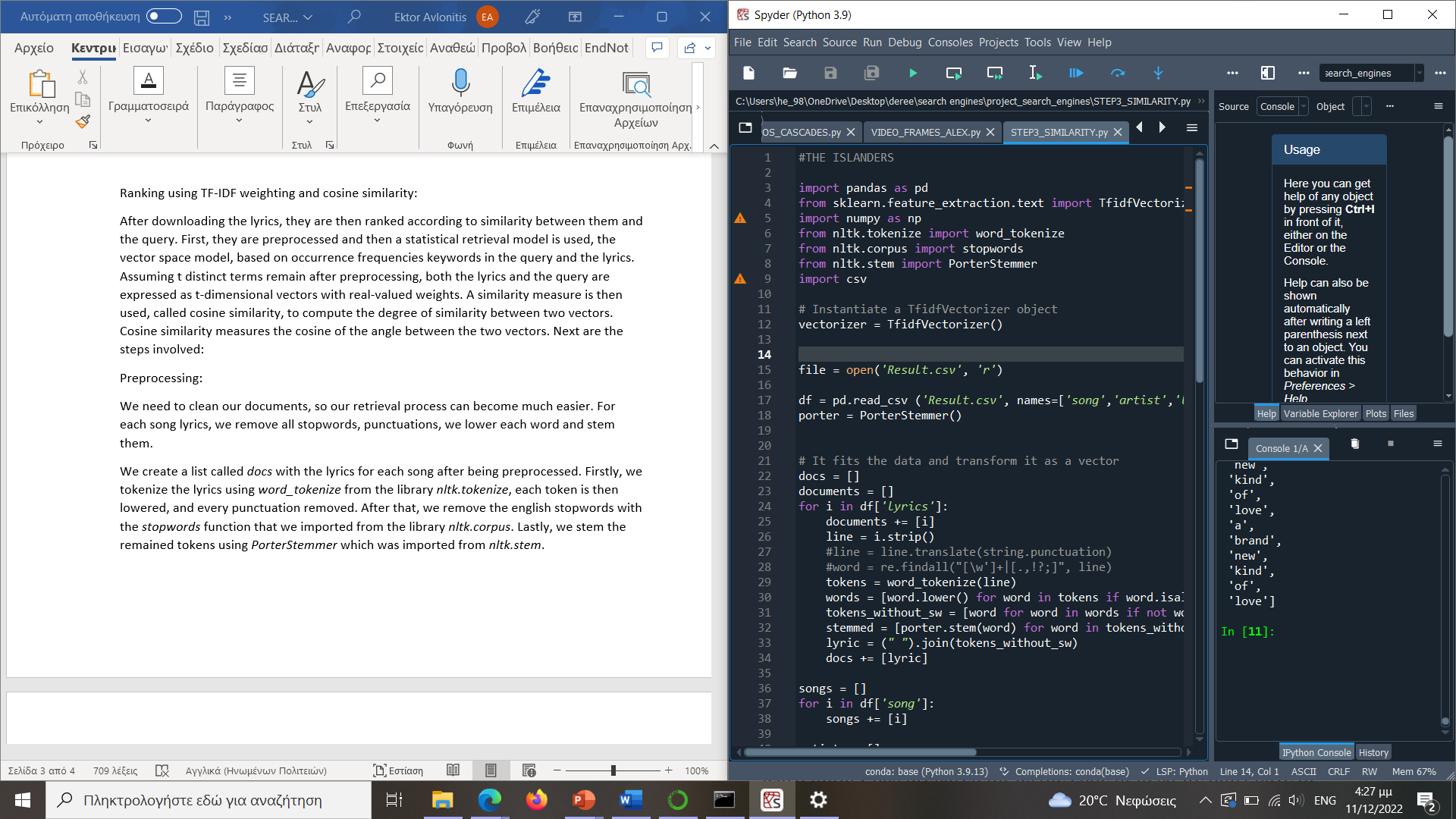
## TF-IDF Ranking, Preprocessing and Cosine Similarity

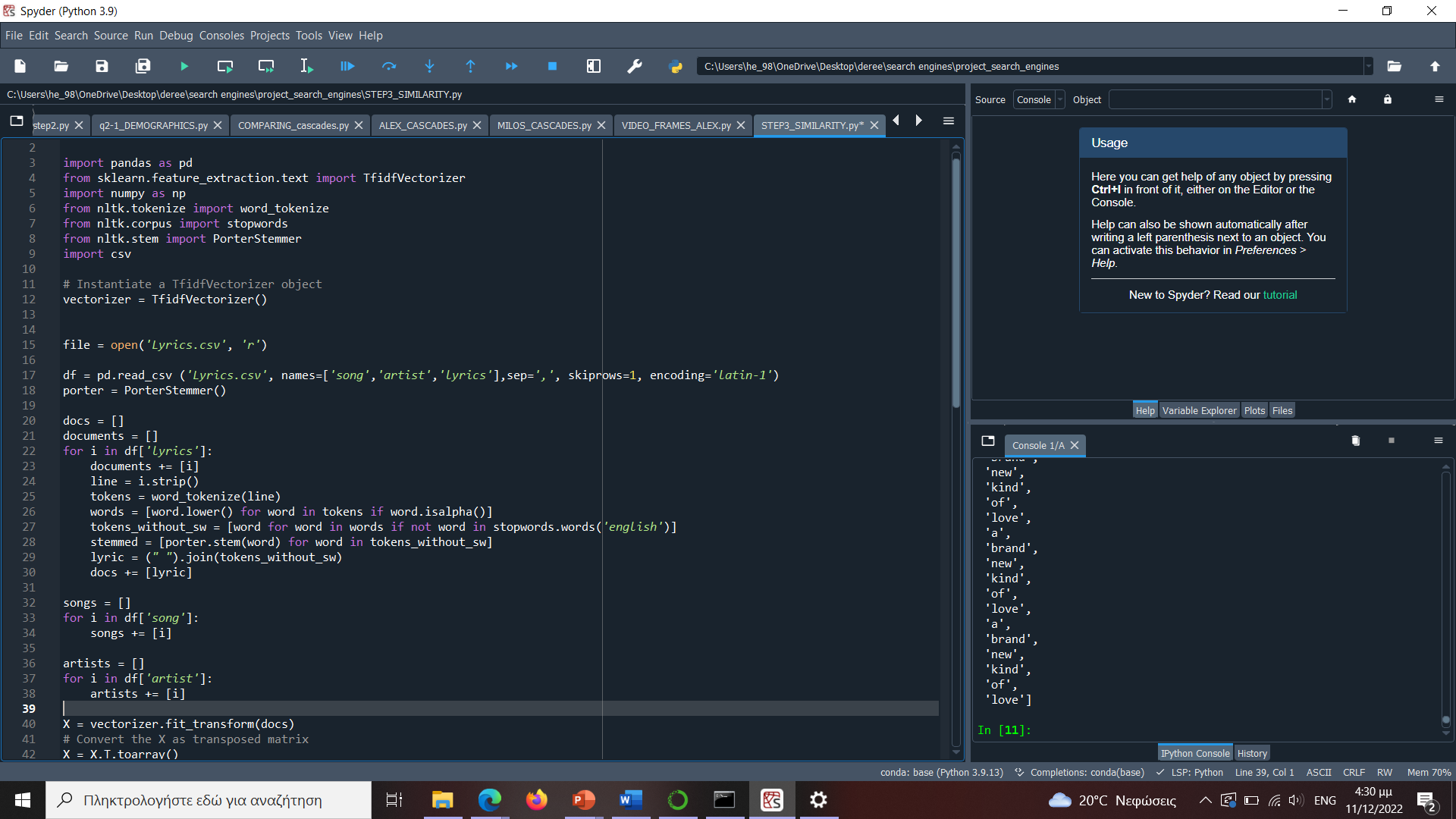
After downloading the lyrics, they are then ranked according to similarity between them and the query. First, they are preprocessed and then a statistical retrieval model is used, the vector space model, based on occurrence frequencies keywords in the query and the lyrics. Assuming t distinct terms remain after preprocessing, both the lyrics and the query are expressed as t-dimensional vectors with real-valued weights. A similarity measure is then used, called cosine similarity, to compute the degree of similarity between two vectors. Cosine similarity measures the cosine of the angle between the two vectors. Next are the steps involved:

### Preprocessing:

We need to clean our documents, so our retrieval process can become much easier. For each song lyrics, we remove all stopwords, punctuations, we lower each word and stem them.

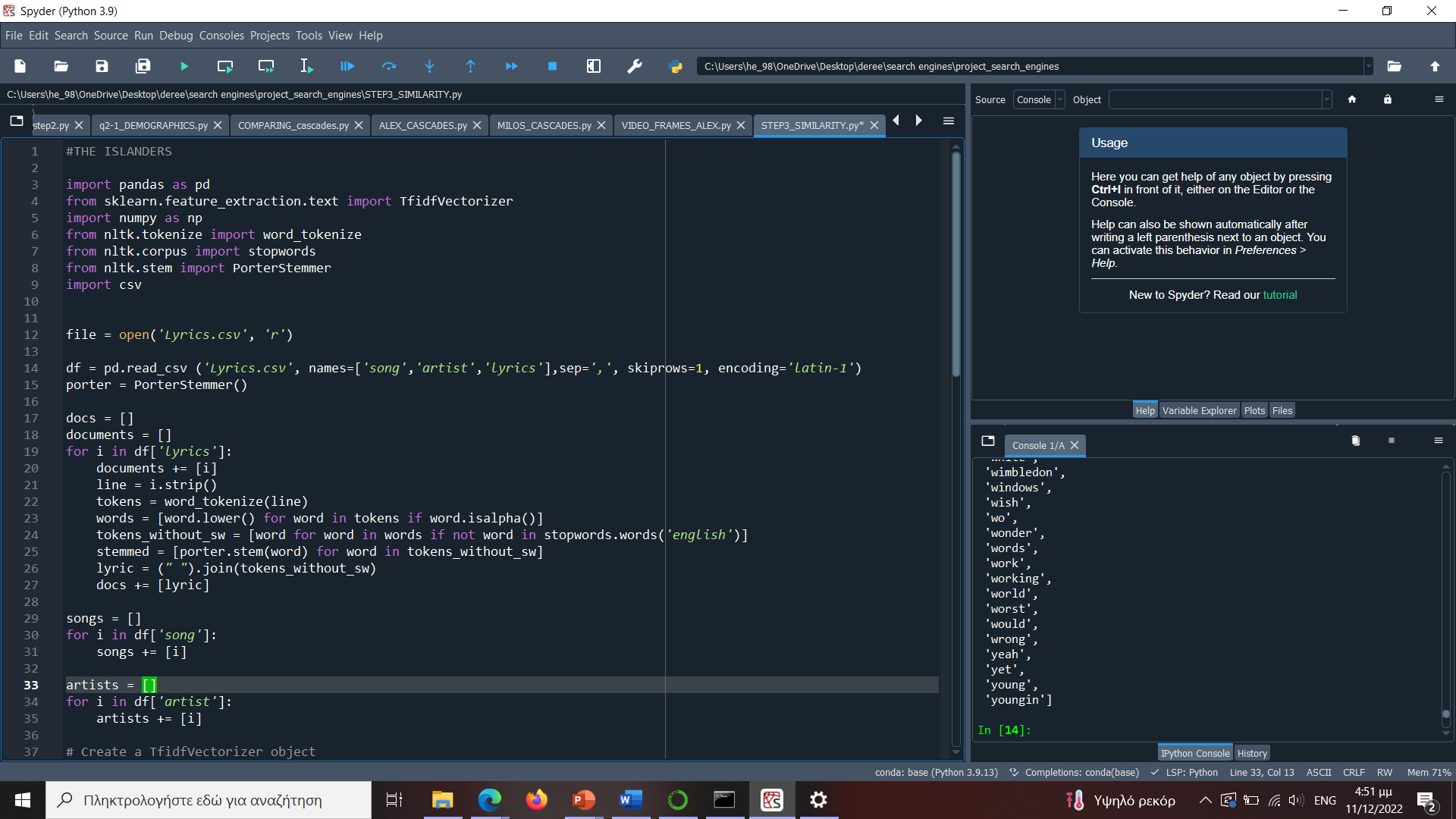
We create a list called *docs* with the lyrics for each song after being preprocessed. Firstly, we tokenize the lyrics using ***word\_tokenize*** from the library ***nltk.tokenize***, each token is then lowered, and every punctuation removed. After that, we remove the english stopwords with the ***stopwords*** function that we imported from the library ***nltk.corpus***. Lastly, we stem the remained tokens using ***PorterStemmer*** which was imported from ***nltk.stem***.



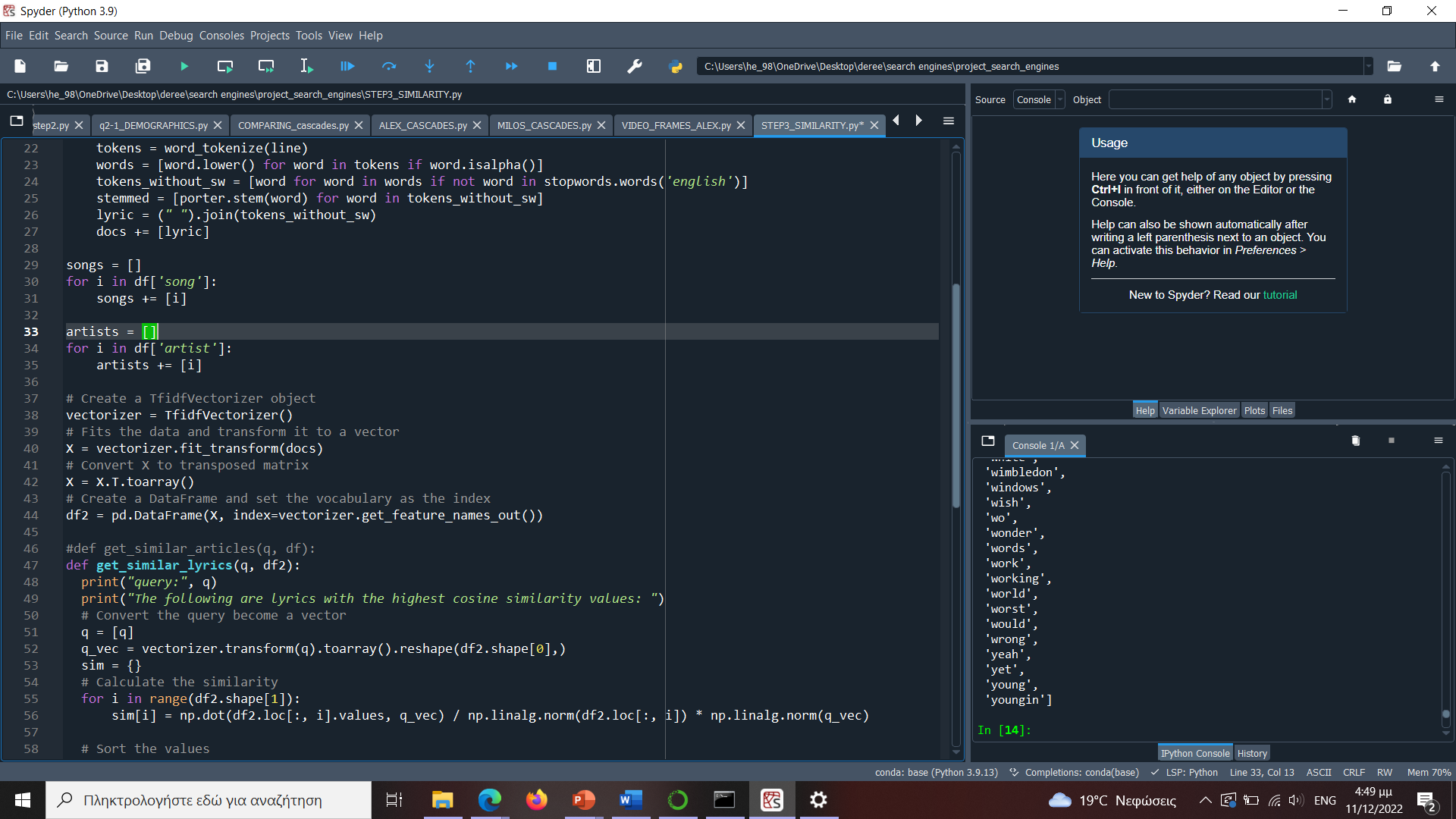


### Vector space model:

The collection of lyrics is presented in the vector space model by a term-lyric matrix. From the library ***sklearn.feature\_extraction.text*** we import the function ***TfidfVectorizer*** to create the weighted vectors for each document. The code uses the TF-IDF method to do this.



The result matrix will become a representation of the lyrics. It is called, also, a Term-Document Matrix, because its rows are the tokens represented in all documents and the columns are every document. The values of the Matrix are the frequencies of each word in the specific document.



### Query preparation:

For the similarity to be calculated, the query needs to be preprocessed the same way as we did for each song lyrics. Thus, for the query we remove all stopwords, punctuations, we lower each word and stem them.

### Cosine similarity:

After creating the matrix, and our query is prepared, we created a function to find lyrics based on the highest similarity between the document and the query. Firstly, we transform the query to a vector on the matrix that we already have. Then we calculate the similarity between that and each song lyrics.

To calculate the similarity, we used the cosine similarity function:

The formula calculates the dot product divided by the multiplication of the length on each vector. The code looks as follows:

A screenshot of a computer

Description automatically generated

As we can see, a dictionary is created with the document id as the key, and its cosine similarity result as the value. The dictionary is then sorted and the 50 most similar lyrics to the query are then appended to new csv called ***Lyrics\_after\_tfidf.csv***. The csv file includes the song’s name, the artist, and the lyrics for each song, separated with a comma, but sorted based on the cosine value.

After running the code multiple times, we observed that there are some extra copies of the songs with more than one artist. The crawler goes through that song more than once and therefore creates multiple copies of the song’s lyrics. Therefore, we insert an if statement to collect only unique values of similarities and therefore only unique songs (avoid duplicates) in our new csv file: ***Lyrics\_after\_tfidf.csv***.

## Using S - BERT:

After using the vector space model and the cosine similarity to rank the documents, we need to implement an embedding for a new ranking.

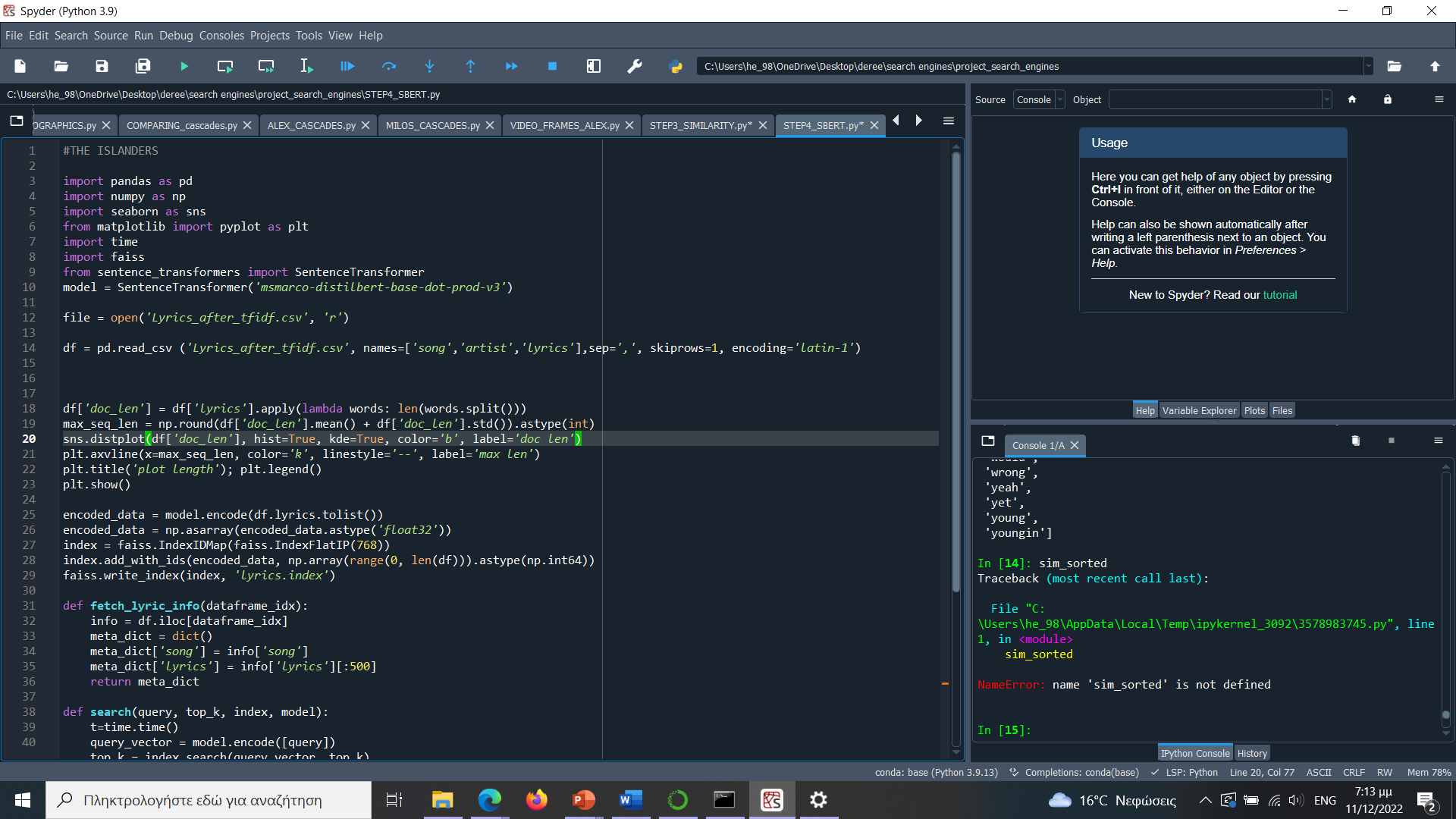
We use a contextual embedding to determine also the intent and contextual meaning of the words and to improve our ranking system. We are using S-BERT which produces sentence embeddings. Unlike

After selecting the embedding model, we are moving on to encoding the data and storing with encoding. For encoding the data, we use a sentence transformer model which performs great in Semantic Textual Similarity which is Asymmetric. We used the model below importing it from the ***sentence\_transformers*** library.

A screenshot of a computer

Description automatically generated with medium confidence

For storage we used FAISS (Facebook AI Similarity Search), which is a library that allows developers to quickly search for embeddings of multimedia documents that are like each other. It solves the limitations of traditional query search engines that are optimized for hash-based searches and provides more scalable similarity search functions.



We have encoded our lyrics, where each song’s lyrics has been encoded with a 768-dimensional vector and stored to disk with ***lyrics.index*** name.

A screenshot of a computer

Description automatically generated with medium confidence

Note here we have used ***index.add\_with\_ids*** and this encodes data in the order of data-frame and stores their index ids too.

Afterwards we write two functions to encode user query and fetch similar song lyrics from FAISS index directory.

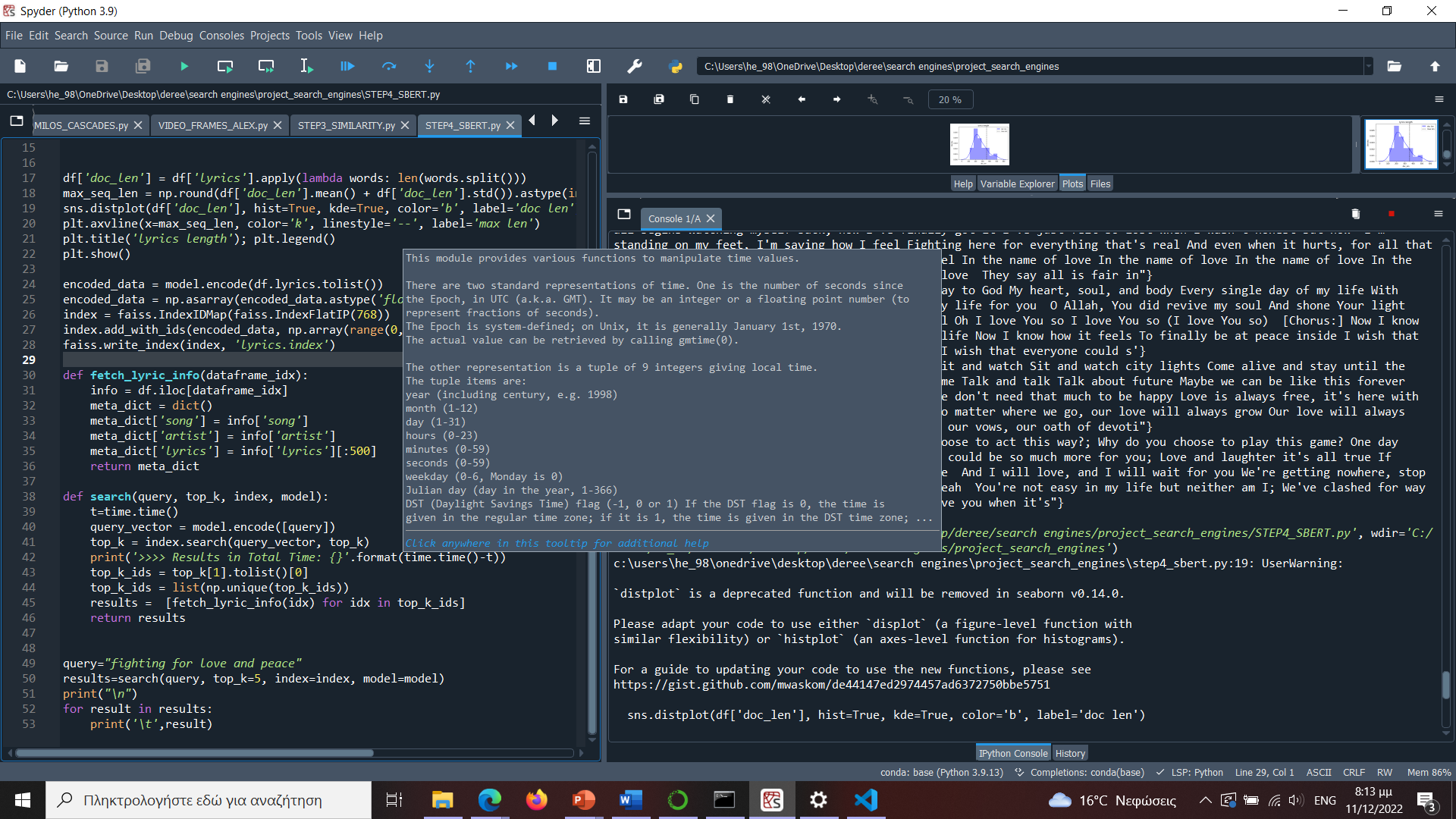
A screenshot of a computer

Description automatically generated with medium confidence

In the ***fetch\_lyric\_info*** function the ***dataframe\_idx*** represents the index value extracted from ***lyric.index*** which can be used for adding information later.

In the *search* function, the inputs are the user query (***query***), the number of results we want returned (***top\_k***), the index to query (***index***) and the model to encode the user-query (***model***).

The query is written on the panel and the results are printed then on the kernel.



## Results:

Finally, here are the IR system’s results for the query: “fighting for love and peace”. If you observe the fourth line from the top, our system has successfully interpreted the semantic meaning of the query and correlated it to the lyrics of the song. Notice the absence of the word “fighting” while its theme can be found as context in the song.

A screenshot of a computer

Description automatically generated with medium confidence

Chart, histogram

Description automatically generated

# Limitations

While we have successfully reached the desired result with our project, it is equally important to reflect on the limitations of our information retrieval system setup:

1. Speed of Processing: The most prominent limitation is the speed of our IR system. Currently, it takes many hours to scrape the minimum number of needed documents from the website to make this project relevant. This compromises the reliability of our IR tool as it cannot be used to gather perhaps important documents in short amount of time.

1. Document Storage: We store lyrical data in CSV files. This can pose a problem in larger document retrieval as it can quickly become too large for local storage. For further purposes, transferring this large amount of data from the local machine to elsewhere would be very inconvenient if not impossible.
2. Compatibility: The code used for scraping is specifically customized for the website we used. If retrieving documents from another website is needed, it would be necessary to rewrite the IR system’s code altogether.
3. Interface: As of now, our tool does not include an input interface for the query. Meaning, our IR system cannot be used as a product on the internet and elsewhere to achieve its purposes. It is only usable in a developer environment.

# Future Work

To finalize our project, we would like to outline some additional features and adjustments that can be made in the future to improve our IR system. We will also tackle the previously mentioned limitations.

1. Optimization and compatibility: We could work on optimizing the python code so that out tool could possibly work with different websites and/or datasets. Secondly, we would like to figure out ways of rewriting the code that would make the document retrieval process much faster. Both will improve the IR system’s usability.
2. Relocating the retrieved documents: Choosing a different storage system than the current local storage. This will prevent the documents from consuming too much space locally and make them easily transferable.
3. Presentability: Adding a user interface to our IR system would give it the capability of working outside a developer environment. In addition, it would make it much more appealing and user-friendly.
4. New Features: Aside from what has already been mentioned, we would like to add more functionality to our IR system. Adding a filter option could allow the user to sort the output documents according to their preference. Perhaps this IR system could be made as an application that supports a profile system where the user can save the lyrics of their favorite themes or topics.

With the above said, this concludes our report on the IR system we have created.

# References

* “An Overview of HTTP - Http: MDN.” HTTP | MDN, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>.
* Ramadhan, Luthfi. “TF-IDF Simplified.” Medium, Towards Data Science, 4 Feb. 2021, https://towardsdatascience.com/tf-idf-simplified-aba19d5f5530.
* Reimers, Nils, and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks." arXiv preprint arXiv:1908.10084 (2019).
* “AZLyrics.” AZLyrics.com, https://www.azlyrics.com/.

1. “An Overview of HTTP - Http: MDN.” HTTP | MDN, https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. [↑](#footnote-ref-1)
2. Ramadhan, Luthfi. “TF-IDF Simplified.” Medium, Towards Data Science, 4 Feb. 2021, https://towardsdatascience.com/tf-idf-simplified-aba19d5f5530. [↑](#footnote-ref-2)
3. Reimers, Nils, and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks." arXiv preprint arXiv:1908.10084 (2019). [↑](#footnote-ref-3)
4. “AZLyrics.” AZLyrics.com, https://www.azlyrics.com/. [↑](#footnote-ref-4)