Information Retrieval System: Lyric Themes

The Islanders

Avlonitis Ektoras

Lazarević Miloš

Vavakas Alexandros

Table of Contents

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] 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.

[1] "An Overview of HTTP - Http: MDN." HTTP | MDN, https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

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]

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]

[2] Ramadhan, Luthfi. “TF-IDF Simplified.” Medium, Towards Data Science, 4 Feb. 2021, https://towardsdatascience.com/tf-idf-simplified-aba19d5f5530.

[3] Reimers, Nils, and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks." arXiv preprint arXiv:1908.10084 (2019).

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

1) Crawling and Scraping

We have selected the website AZ Lyrics[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.

```
import urllib.request, urllib.parse, urllib.error
import requests
from bs4 import BeautifulSoup as bs
from time import sleep
import random
import csv
```

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.

```
letters = []
g_data = soup.find_all("a", {"class": "btn btn-menu"})
for item in g_data:
    print(item.get("href"))
    letters += [item.get("href")]

new_letters=[]
for i in range(10):
    random_num = random.choice(letters)
    new_letters += [random_num]
```

[4] "AZLyrics." AZLyrics.com, https://www.azlyrics.com/.

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***.

```
artists = []
for i in new_letters:
    try:
        url = "http:" + i
        r2 = requests.get(url)
        soup2 = bs(r2.content, "lxml")

        g_data2 = soup2.find_all("div", {"class": "col-sm-6 text-center artist-col"})
        for div in g_data2:
            links = div.find_all('a')
            for a in links:
                print(a['href'])
                artists += [a["href"]]
    except:
        print("error")
        break;
    finally:
        sleep(delay)

new_artists=[]
for i in range(40):
    random_num = random.choice(artists)
    new_artists += [random_num]
```

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.

```
songs = []
for i in new_artists:
    try:
        url = "https://www.azlyrics.com/" + i
        r3 = requests.get(url)
        soup2 = bs(r3.content, "lxml")

        g_data3 = soup2.find_all("div", {"class": "listalbum-item"})
        for div in g_data3:
            links = div.find_all('a')
            for a in links:
                print(a['href'])
                songs += [a["href"]]
    except:
        print("error")
        break;
    finally:
        sleep(delay)

new_songs=[]
for i in range(1500):
    random_num = random.choice(songs)
    new_songs += [random_num]
```

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.

```
with open ('Lyrics_FINAL.csv', "a") as csvfile:
    fieldnames = ['song', 'artist', 'lyrics']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    for i in new_songs:
        try:
            if i[0:5] == 'https':
                f_url=i
                html_page=urllib.request.urlopen(f_url).read()
                soup=bs(html_page, 'html.parser')
                html_pointer=soup.find('div',attrs={'class':'ringtone'})
                song_name=html_pointer.find_next('b').contents[0].strip()
                html_pointer2=soup.find('div',attrs={'class':'lyricsh'})
                artist_name=html_pointer2.find_next('b').contents[0].strip()
                artist_name = artist_name.replace(" Lyrics", "")
                print(artist_name)
                lyrics=html_pointer.find_next('div').text.strip()
                lyrics = " ".join(line.strip() for line in lyrics.splitlines())

                writer.writerow({'song': song_name, 'artist': artist_name, 'lyrics': lyrics})
                print("lyrics succesfully written to file for: " + song_name)
            else:
                f_url = "https://www.azlyrics.com" + i
                html_page=urllib.request.urlopen(f_url).read()
                soup=bs(html_page, 'html.parser')
                html_pointer=soup.find('div',attrs={'class':'ringtone'})
                song_name=html_pointer.find_next('b').contents[0].strip()
                html_pointer2=soup.find('div',attrs={'class':'lyricsh'})
                artist_name=html_pointer2.find_next('b').contents[0].strip()
                artist_name = artist_name.replace(" Lyrics", "")
                print(artist_name)
                lyrics=html_pointer.find_next('div').text.strip()
                lyrics = " ".join(line.strip() for line in lyrics.splitlines())

                writer.writerow({'song': song_name, 'artist': artist_name, 'lyrics': lyrics})
                print("lyrics succesfully written to file for: " + song_name)

        except:
            print("lyrics not found for : ", i)

        finally:
            sleep(delay)
```

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.

2) 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:

A) 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***.

```
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
```

```
docs = []
documents = []
for i in df['lyrics']:
    documents += [i]
    line = i.strip()
    tokens = word_tokenize(line)
    words = [word.lower() for word in tokens if word.isalpha()]
    tokens_without_sw = [word for word in words if not word in stopwords.words('english')]
    stemmed = [porter.stem(word) for word in tokens_without_sw]
    lyric = (" ").join(tokens_without_sw)
    docs += [lyric]
```

B) 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.

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

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.

```
# Create a TfidfVectorizer object
vectorizer = TfidfVectorizer()
# Fits the data and transform it to a vector
X = vectorizer.fit_transform(docs)
# Convert X to transposed matrix
X = X.T.toarray()
# Create a DataFrame and set the vocabulary as the index
df2 = pd.DataFrame(X, index=vectorizer.get_feature_names_out())
```

C) 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.

D) 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:

$$cosine(q, d) = \frac{q * d}{|q| * |d|}$$

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

```
def get_similar_lyrics(q, df2):
  print("query:", q)
  print("The following are lyrics with the highest cosine similarity values: ")
  # Convert the query become a vector
  q = [q]
  q_vec = vectorizer.transform(q).toarray().reshape(df2.shape[0],)
  sim = {}
  # Calculate the similarity
  for i in range(df2.shape[1]):
      sim[i] = np.dot(df2.loc[:, i].values, q_vec) / np.linalg.norm(df2.loc[:, i]) * np.linalg.norm(q_vec)

  # Sort the values
  sim_sorted = sorted(sim.items(), key=lambda x: x[1], reverse=True)
  print(sim_sorted)
  # Print the lyrics and their similarity values
  x = 0
  result = open('Lyrics_after_tfidf.csv', "a", encoding="utf-8")
  fieldnames = ['song', 'artist', 'Lyrics']
  writer = csv.DictWriter(result, fieldnames=fieldnames)
  writer.writeheader()
  last_v = 0.0
  for k, v in sim_sorted:
    if v != 0.0:
      if v!= last_v:
          last_v = v
          print("Similarity Value:", v)
          writer.writerow({'song': songs[k], 'artist': artists[k], 'Lyrics': documents[k]})
          #sbert += [documents[k]]
          x += 1
          if x == 50:
              break;
          print(documents[k])
          print()
      else:
          continue
```

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***.

3) 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.

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('msmarco-distilbert-base-dot-prod-v3')
```

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.

```
import faiss
```

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.

```
encoded_data = model.encode(df.lyrics.tolist())
encoded_data = np.asarray(encoded_data.astype('float32'))
index = faiss.IndexIDMap(faiss.IndexFlatIP(768))
index.add_with_ids(encoded_data, np.array(range(0, len(df))).astype(np.int64))
faiss.write_index(index, 'lyrics.index')
```

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.

```
def fetch_lyric_info(dataframe_idx):
    info = df.iloc[dataframe_idx]
    meta_dict = dict()
    meta_dict['song'] = info['song']
    meta_dict['lyrics'] = info['lyrics'][:500]
    return meta_dict

def search(query, top_k, index, model):
    t=time.time()
    query_vector = model.encode([query])
    top_k = index.search(query_vector, top_k)
    print('>>>> Results in Total Time: {}'.format(time.time()-t))
    top_k_ids = top_k[1].tolist()[0]
    top_k_ids = list(np.unique(top_k_ids))
    results =  [fetch_lyric_info(idx) for idx in top_k_ids]
    return results
```

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.

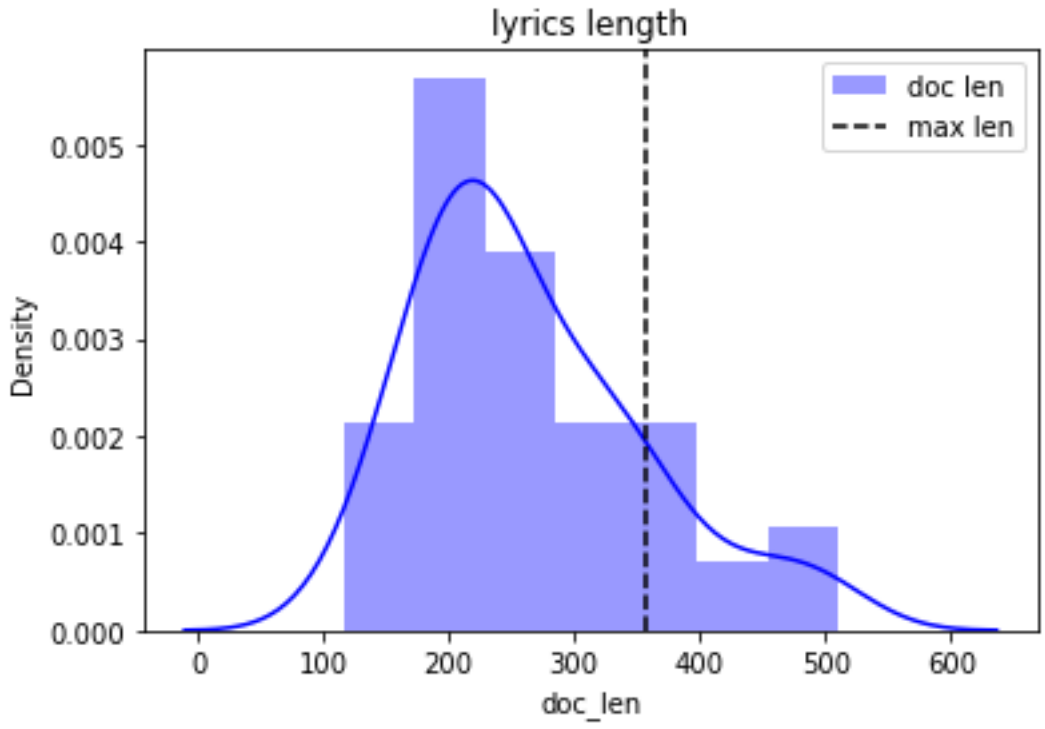
```
query="fighting for love and peace"
results=search(query, top_k=5, index=index, model=model)
print("\n")
for result in results:
    print('\t',result)
```

4) 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.

```
>>>> Results in Total Time: 0.42114758491516113


    {'song': '"True Love"', 'artist': 'Maher Zain', 'lyrics': 'I long for a world so pure and free I wish for others
all IÂ\x92d wish for me To live right, avoiding what is wrong And focus everyday on the ultimate goal  Forever,
whatÂ\x92s in my heart Is all of the love from Allah  [Chorus:] True love, itÂ\x92s a gift I will never let go of true
love I will give my body and soul for true love Everyday in my heart I feel it grow With true love, oh oh True love,
love, love yeah!  Each day IÂ\x92ll extend my hand Give my all and do whatever I can For a good life Of joy an'}
    {'song': '"In The Name Of Love"', 'artist': 'Delta Goodrem', 'lyrics': "Started on the outside, worked my way in
Tracing the road to where it all begins Watching myself back, now I've finally got it I've just felt so lost when I
wasn't honest But now  I'm standing on my feet, I'm saying how I feel Fighting here for everything that's real And even
when it hurts, for all that I am worth I'm fighting here for everything I feel In the name of love In the name of love
In the name of love In the name of love In the name of love In the name of love  They say all is fair in"}
    {'song': '"I Love You So"', 'artist': 'Maher Zain', 'lyrics': 'I pray to God My heart, soul, and body Every single
day of my life With every breath I solemnly promise To try to live my life for you  O Allah, You did revive my soul And
shone Your light into my heart So pleasing You is now my only goal Oh I love You so I love You so (I love You so)
[Chorus:] Now I know how itÂ\x92s like To have a precious love in my life Now I know how it feels To finally be at peace
inside I wish that everybody knew How amazing it feels to love You I wish that everyone could s'}
    {'song': '"Don\'t Need Much"', 'artist': 'Marie Digby', 'lyrics': "Sit and watch Sit and watch city lights Come
alive and stay until the sunrise And now you know everything there is to me Talk and talk Talk about future Maybe we can
be like this forever Live is good, it won't always be this way  But we don't need that much to be happy Love is always
free, it's here with you and me We don't need that much to be happy No matter where we go, our love will always grow Our
love will always grow  Side by side we stood by the ocean We said our vows, our oath of devoti"}
    {'song': '"Wait"', 'artist': 'Addison Road', 'lyrics': "Why do you choose to act this way?; Why do you choose to
play this game? One day you'll wake up and find that we're all gone Life could be so much more for you; Love and
laughter it's all true If smiles are worth a million words just show me one  And I will love, and I will wait for you
We're getting nowhere, stop messing around and just be you Yeah yeah, yeah yeah  You're not easy in my life but neither
am I; We've clashed for way too long The truth is hitting home But I will love you when it's"}
```



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.

2) *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.

3) *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.

4) *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/.