Text Technologies Of Data Science - Course Work 3 Report

Topic : Song Recognition using Lyrics

Group Number : 78

Sartaj Syed - s2340198, Keith Wu - s2406186, Yilin Wang - s2326916, Thomas Chan - s2341572
Nikila Sharan - s2306209, Nithya Vupparapalli - s2424386

Abstract

The goal of this project is to create a search engine that allows users to find songs based on various search filters, including lyrics, album name, artist name, and song title. The project involves developing an information retrieval system that connects to a MongoDB database containing a large dataset of relevant song information. To build the dataset, we used web scraping techniques and APIs such as spotipy and lyricsgenius. Our information retrieval system uses a combination of retrieval techniques, including TF-IDF, lyrics search algorithm, and a combined search algorithm to provide the user with the top 10 songs relevant to their search query. To improve user experience, we have also implemented a recommender system that uses SVM models to train data retrieved from the spotipy API. The recommender system offers song recommendations to users based on their query search. Overall, our goal is to create a user-friendly web page that allows users to easily search for and discover new music.

1. Introduction

Music is the most popular source of entertainment in recent years. There are many different types and forms of music along with different genres in different languages. Song search engines are gaining momentum due to its ease to search songs that the user is looking for along with advanced recommender systems to recommend songs based on the user's previous search. For this coursework we have come up with a lyrics search engine called "Lyrics Wizard".
A lyric search engine is a tool that allows users to search for the lyrics of their favourite songs. It works by scanning through a database of song lyrics and matching them with the user's search query. This report discusses the development of a lyric search engine, its features, and its benefits to users. We have added an additional search feature to this search engine. That is, there is another search box which enables the user to add 3 different filters, namely, artist, album, track name. These applied filters help with getting optimal search results in comparison to the existing systems. The purpose of this project is to create a user-friendly and efficient lyric search engine that provides accurate results. The project aims to simplify the process of finding song lyrics and make it easier for users to access their favourite songs' lyrics. The development of this project involved using natural language processing techniques, web scraping, and database management. After fetching the lyrics, the user is also provided with an option to view more

songs of his choice with a recommendation system that uses K-Means clustering on the dataset. Overall, the system aims for a hassle proof search-engine with an integrated recommender system.

2. Dataset Creation and Setup

Since the project requires us to work on huge datasets, the team members used various methods to develop scripts or codes to collect relevant data related to songs, artists, albums, lyrics. Each of the scripts was run by all team members on a daily basis and later combined to be stored in the database.

2.1 Data Fetching using spotipy API

Spotipy API is a Python library for the Spotify Web API. Based on simple REST principles, the Spotify Web API endpoints return JSON metadata about music artists, albums, and tracks, directly from the Spotify Data Catalogue. With Spotipy, users get full access to all of the music data provided by the Spotify platform. In order to use the methods and functionalities in this API, the user is first required to create client access and client secret access tokens. These tokens help with the communication between the user requests and API responses. The scripts were developed using this API in such a way that it follows a systematic order:

1. First we fetch the artist information that contains details like artist_name, artist_id, artist_popularity and artist_genres. We will store data for all artists in multiple csv files.
2. Next, for each artist in every file, we will get the album details. We will be storing data like album_idx, album_name, album_release_year, album_release_month and album_release_day. For each page of the artist.csv files, on an average, 15 album csv files are generated.
3. Finally, we need to get the track information for every song present in each album.csv file. These track data are stored in csv files and will contain data such as track_spotify_idx, track_name, duration, explicit features, track features such as energy, loudness, valence, tempo and lyrics.

These three scripts were run by all the team members on a regular basis such that the data can be fetched at a better rate. Finally, the team collected around 1.62 million songs in 433 thousands of albums by 142 thousands of artists. We store the documents of songs, albums and artists in 3 distinct collections in MongoDB. Each document also has a record of the Spotify index for the use of inverted index creation.

2.2 Data Fetching using 'lyricsgenius' API and Web Scraping

In this scenario, we make use of techniques like web scraping and application of regex patterns to retrieve lyrics of songs. We also see how the methods and features in the genius API are used to retrieve song details. After the data is fetched by using these methods, they are stored in csv files that will later be sent to the database which is used in IR system and UI development.

2.2.1 Lyricsgenius API :

The second method of collecting the dataset for the project was done using an in-built python library called 'lyricsgenius'. It provides access to content on Genius.com webpage and lets a user download information such as lyrics to artists' songs, song titles, thumbnail images, album names, date of release, etc. Before using the Genius API, it is mandatory to create a client access token so that the library can be used with ease. Once the access token is created, we need to create a genius object which is later used in the script to fetch all the relevant data from the API. After an access client has been generated on the Genius.com website, below code can be used to create a genius object:

```
import lyricsgenius as lg

genius = lg.Genius('access_token', skip_non_songs=True, excluded_terms=["(Remix)", "(Live)"],
                   remove_section_headers=True)
```

Once the object has been created, we will now use it to call the API for getting the lyrics of all the songs belonging to an artist. First, we create csv files to store all the artists on the genius page for each alphabet. For every artist in a csv file we create an artist end_point url that returns a json script containing limited artist information like their songs, artist ID, albums and features. So, to retrieve all the songs belonging to an artist we create a song end_point url which gives us songs in each page and are stored in a songs list.

```
Songs_endpoint =
f'https://api.genius.com/artists/{artistID}/songs?per_page=50&page={page_number}&access_token={access_token}'

artists_endpoint = f'https://api.genius.com/search?q={artist}&access_token={access_token}'
```

For every song the songs list, we call a function to get its lyrics. In this function we make a call to the genius API by first searching for the song. If it is present, we call the API for the second time to return the lyrics if present. After retrieving the song lyrics, we store the value in a csv file which contains all the relevant information like title, artist name, song names, image urls, date of releases, featured artists, lyrics etc.

```
def get_lyrics(song_name, artist_name):
  try:
    song = genius.search_song(song_name, artist_name)
    if song is None:
      return "Sorry, the lyrics of this song are not available."
    else:
      return song.lyrics
```

One of the disadvantages of using this method is the time taken to retrieve the data is very slow. Due to multiple calls to the API as well as sending requests to get the json response, there were many instances of timeout exceptions at high frequency. In order to improve the collecting time, the second approach used was web scraping lyrics from the genius API.

2.2.2 Web Scraping

In this technique, we create a song end_point url using the song ID (similar to the one in genius API method) , access token and send the url request. This will return a json script that will be parsed into a html text using the BeautifulSoup library. Once we get this html script, we used the regex pattern twice: first to get only the song lyrics and the second time to get details such as primary artist, title, date of release, image urls for front end development.

With the help of web scraping we could see a huge difference in data collection. It was much quicker compared to using the genius API calls and helped in retrieving the lyrics and other information using techniques like regex pattern matching, string manipulation. All the information was stored in a csv file, and then later sent to the database.

2.3 Database Setup

Once data was retrieved by the above techniques, we used the pymongo library to send the data in the csv file to the database in the MongoDB. The purpose of this step is to enable a connection to the front end of the project so that information in the DB can be used for UI as well as IR system. First we all created a secure MongoDB client that is only accessible to the team members to prevent hacking and loss of data. All the csv files that are containing the data were converted to dataframes using the pandas library in python.

We import the MongoCollection library and MongoClient from pymongo library to establish a secure connection with our database and send the data frames for storage in MongoCompass. During the implementation of the IR system, we need to create a connection to the same database and use it to read data for calculation of scores of user inputs/queries. We also need to establish a connection to the front end software design in order to maintain a secure connection with the backend and also to preserve synchronisation within the entire system.

3. System Design

3.1 IR System

The core part of the search engine is the design and implementation of the information retrieval system. The IR system will be initiated when the user provides their input/query in the web page. The front end is developed in such a way that it can take 2 types of queries by the user :

1. **Lyrics**: It is the primary query in the IR system. It can be null or have value entered by the user
2. **Filter and filter query**: This means that the user can choose a filter of their choice. There are 4 filters implemented, namely, ‘title’ or ‘artist name’ or ‘album name’ or ‘choose a filter’(default). The user then provides the second query in accordance to the chosen filter type.

Once the user enters the queries accordingly, the IR system is triggered. The main goal of the system is to calculate the scores of matched songs to the user query and return the song with the highest score. The algorithms used to calculate the scores in various cases has been discussed in detail below:
Case 1: The user has provided only the lyrics query. In this case just the lyric search algorithm is used.
Case 2: The user has given only the filter and filter query. Here the scores are calculated using the TF-IDF method.

Case 3: The user has given both the lyrics and filter, filter queries. We implement the combined search algorithm for this case.

Lyric Search Algorithm (Case 1):

This algorithm is called whenever the user gives an input query in the first text box. But before this, first the length of the query is checked. If it is greater than 10, then the long query handling is applied. This returns the modified query where 10 most relevant tokens are returned from the long query handling logic.
Then, the instances of all the 'lyric' filter are read from the database and each spotify ID is stored in the spotify IDs list. Lyrics search algorithm consists of 3 parts:

1. **Pattern Search:** In this logic, the inputs given are the query tokens and the positional index. First, pre-processing is applied on the tokens. We will not be removing the stop words during pre-processing to maintain the accuracy of the outputs to the user. Next, the length of the query is calculated. If the length of tokens is 1, it means there is only 1 term, all the instances of the term in the positional index are returned.

   If the number of tokens are greater than 1, then the tokens are sent to permutations logic. In the permutation logic, below steps happen :
   For example, if the number of tokens in the query are 5. Query = A B C D E, then the permutations are generated as
   ['A B C D E', 'B C D E', 'A C D E', 'A B D E', 'A B C E', 'A B C D', 'C D E', 'B D E', 'B C E', 'B C D', 'A D E', 'A C E', 'A C D', 'A B E', 'A B D', 'A B C', 'D E', 'C E', 'C D', 'B E', 'B D', 'B C', 'A E', 'A D', 'A C', 'A B', 'E', 'D', 'C', 'B', 'A']. It is important to note that we have preserved the sequence of tokens in this case (from right to left).

   For every token in the list, starting from the full query, we check whether the pattern matches any of the songs. In case of any matches observed, we add them to the score dictionary. Next we repeat for the next pattern in the permutation list and this will continue until the score dictionary has 15 songs in them or until the number of tokens is 0. This approach is used to ensure that scores of more matched tokens give us songs with higher scores (match probability).
   The pattern search returns the list of top 15 songs that match the query.

2. **Rarity Search:** The rarity search is the second part of the lyric search algorithm that is implemented in the IR system. In order to start this function, the inputs provided include the query, spotify IDs list and positional index dictionary. With the help of these inputs, the TF-IDF logic is applied to calculate the scores of songs that contain the terms present in the query. This logic returns the sorted list of top 15 highest scoring songs.

3. **Lyric Search:** After getting the result list of 15 songs from the pattern search, the scores for each song is calculated. In the next step we normalise the scores for the results obtained in both the pattern search and rarity search. We define weightage for each of the results. In our logic we

currently give 70% weightage to TF-IDF results and 30% weightage to pattern search results. We then compute the final scores of the songs using the below formula :
Final score = Rarity search score * 0.7 + pattern search score * 0.3

We finally return a sorted ranked list of these 15 songs for Case 1.

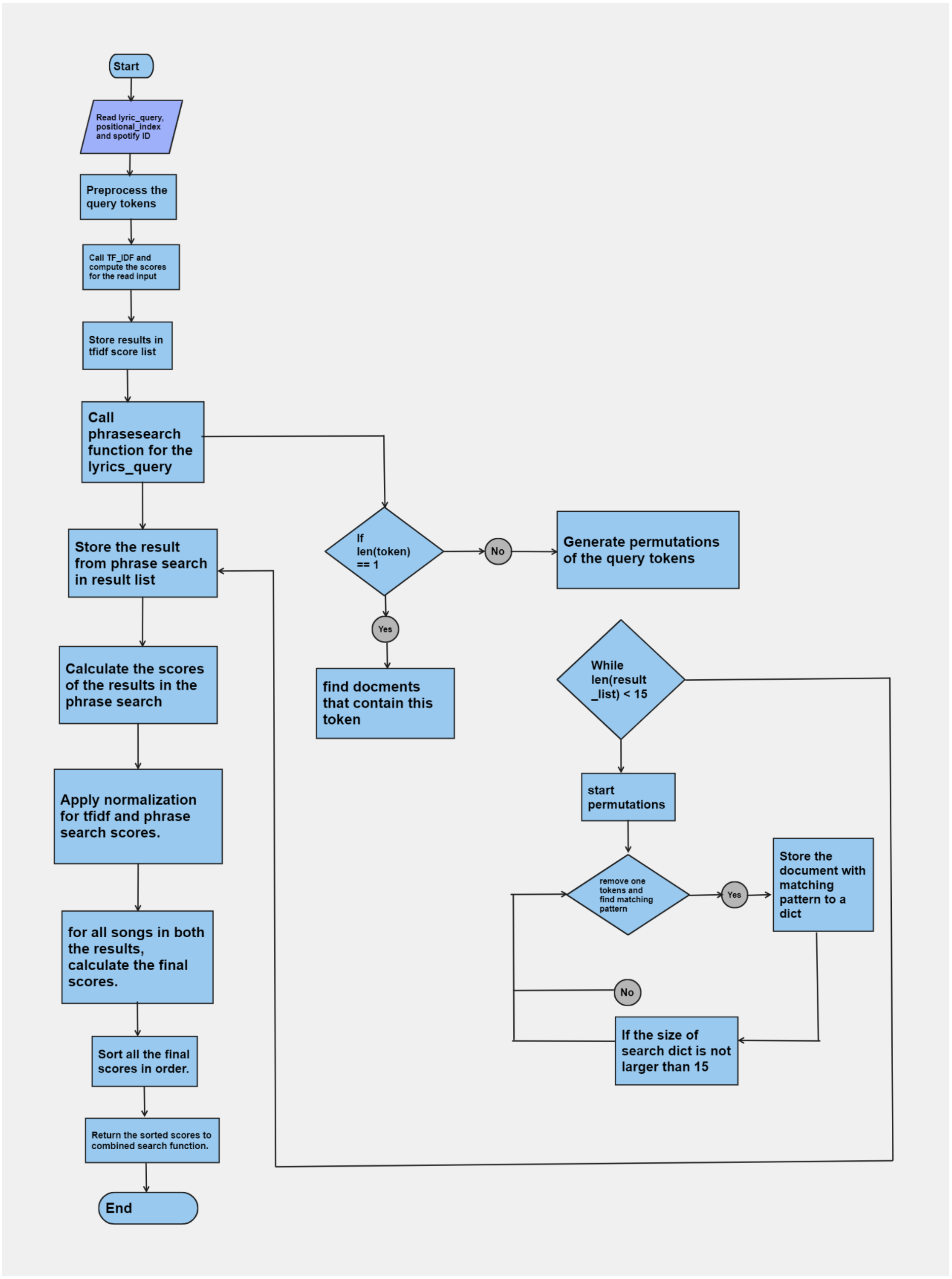
TF-IDF Algorithm for filters (Case 2):
This algorithm is used when the query provided by the user consists of a filter type and corresponding filter query. The length of the query is checked and long query handling is applied if the length is greater than 10. Now, for the given filter, all the spotify IDs containing it in the json file from the database are stored in the spotify IDs list. Then we read the inverted index from the json file which contains all the occurrences of the given query in the filter type. This is called positional index. Then we call the TF_IDF function to preprocess the query tokens. During the pre-processing this time, we will not be removing the stop words as it may hinder the accuracy in the output. We then apply the term frequency formula for each token in the given list of spotify IDs that are present in the positional index dictionary to calculate the scores. This function then returns the list of the top 15 items with respect to the scores.

Combined Search Algorithm (Case 3):
This algorithm is used when the user has provided inputs for both lyrics and filter data. Here, the lyric search algorithm is called for the query containing the lyrics and TF-IDF algorithm is called for the filter data and type. Both the algorithms will return a list of items with top 15 most matched scores for the respective queries.

From the lyrics search, we have generated scores for top 15 songs. Now, we use the IDs of the top 15 scores of each song in lyrics search to get the corresponding filter score ID. If the filter is ‘album’ or ‘artist’, we get the top 10 albums or artists. If the filter is ‘title’, then we do not need to search from the database. Since one song may be contained in multiple albums or artists, we only get the maximum score from filter search. After getting the maximum score from the filter search, we calculate the final scores for results in lyric search and maximum score from filter search using the below formula :

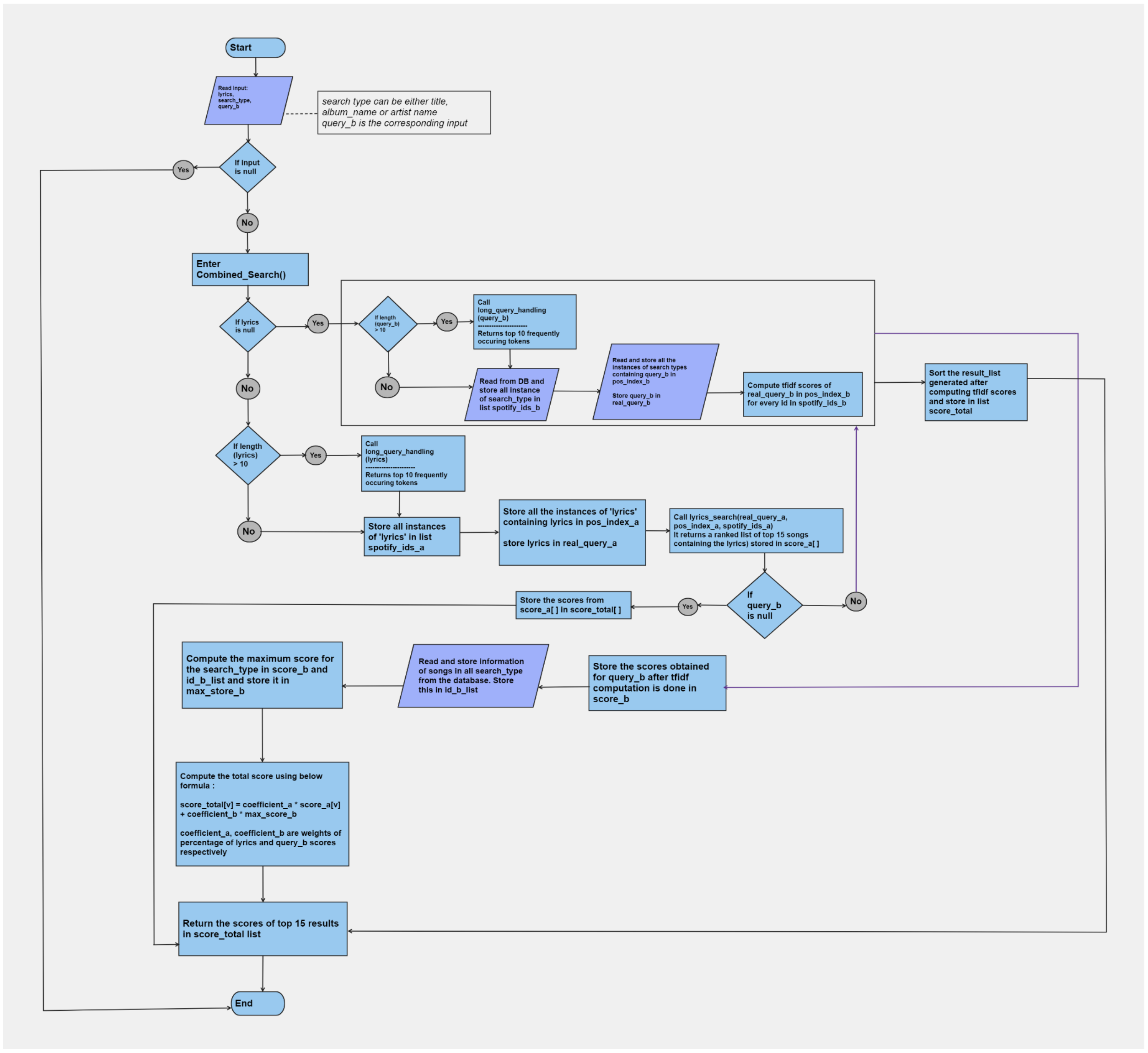
Final_scores = coefficient_a * lyrics_score + coefficient_b * max_score_filter_search
Here the coefficient_a and coefficient_b refer to the weightage of each result and the amount of priority it is given. For the current IR system, we have considered giving the lyrics search results 70% weightage (coefficient_a = 0.7) and the maximum score from filter search 30% (coefficient_b = 0.3)
The combined search algorithm returns a sorted list of final scores in the last step.
All the above algorithms and their flow have been displayed further in terms of flow charts for better visualisation.

1. Lyric Search Flow Chart



Start
Read lyric_query, positional_index and spotify ID
Preprocess the query tokens
Call TF_IDF and compute the scores for the read input
Store results in tfidf score list
Call phrasesearch function for the lyrics_query
Store the result from phrase search in result list
Calculate the scores of the results in the phrase search
Apply normalization for tfidf and phrase search scores.
for all songs in both the results, calculate the final scores.
Sort all the final scores in order.
Return the sorted scores to combined search function.
End
If len(token) == 1
No
Generate permutations of the query tokens
Yes
find docments that contain this token
While len(result _list) < 15
start permutations
remove one tokens and find matching pattern
Yes
Store the document with matching pattern to a dict
No
If the size of search dict is not larger than 15

2. Combined Search Flow Chart



Start
Read input: lyrics, search_type, query_b
search type can be either title, album_name or artist name query_b is the corresponding input
If input is null
Yes
No
Enter Combined_Search()
If lyrics is null
Yes
If length (query_b) > 10
Yes
Call long_query_handling (query_b) ---------------------- Returns top 10 frequently occuring tokens
No
Read from DB and store all instance of search_type in list spotify_ids_b
Read and store all the instances of search types containing query_b in pos_index_b Store query_b in real_query_b
Compute tfidf scores of real_query_b in pos_index_b for every id in spotify_ids_b
Sort the result_list generated after computing tfidf scores and store in list score_total
No
If length (lyrics) > 10
Yes
Call long_query_handling (lyrics) ---------------------- Returns top 10 frequently occuring tokens
No
Store all instances of 'lyrics' in list spotify_ids_a
Store all the instances of 'lyrics' containing lyrics in pos_index_a store lyrics in real_query_a
Call lyrics_search(real_query_a, pos_index_a, spotify_ids_a) It returns a ranked list of top 15 songs containing the lyrics) stored in score_a[ ]
If query_b is null
Yes
No
Store the scores from score_a[ ] in score_total[ ]
Store the scores obtained for query_b after tfidf computation is done in score_b
Read and store information of songs in all search_type from the database. Store this in id_b_list
Compute the maximum score for the search_type in score_b and id_b_list and store it in max_store_b
Compute the total score using below formula :
score_total[v] = coefficient_a * score_a[v] + coefficient_b * max_score_b
coefficient_a, coefficient_b are weights of percentage of lyrics and query_b scores respectively
Return the scores of top 15 results in score_total list
End

Long Query Handling :

This function is called when the length of the query provided by the user is greater than 10. First we preprocess the query by converting to lower case, apply tokenization and then remove stop words. After preprocessing, we will count the frequency of each token in the position_index and sort them accordingly. From the sorted list, we need to consider the first 10 frequently occurring tokens and use them for the search algorithms. This handling is used to improve the search frequency and help retrieve songs with relatively higher scores.

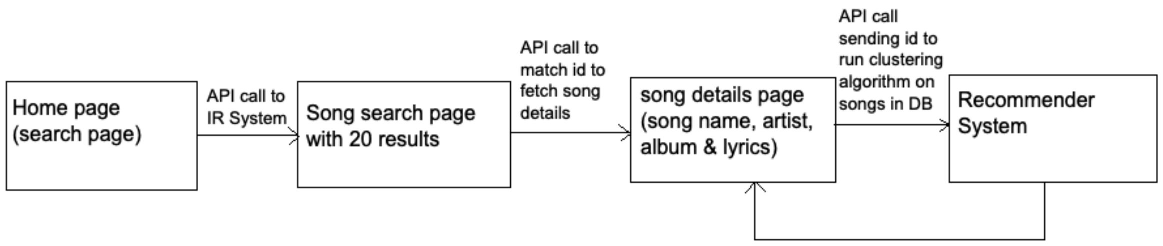
3.2 Website Development

For the website we used the Django Rest API framework with a mongoDB database which is connected using pymongo. The Home page of the system takes the lyrics as an required input and the filter selection and filter input as optional inputs. A POST API is called to send these values to the IR system to obtain the top 15 relevant songs as per the algorithms mentioned in section 3.1. We used "track_spotify_idx" as the primary key from the database to fetch the song values. The song details are returned as a python dictionary by using the query "collection.find_one({"track_spotify_idx": id})", where "collection" is the name of the database in the chosen database. From here the user can view the song details as per his choice from the 15 fetched songs. The song details page consists of the song name, album name, artist name and the lyrics of the song. On the song details page there is an option to obtain recommendations based on the current song lookup by the user. On clicking this button the user will be given suggestions for ten other songs using the song features that have been obtained from the spotipy_API as the labels. From here again the user can view the song details by clicking the "Get Lyrics" button.

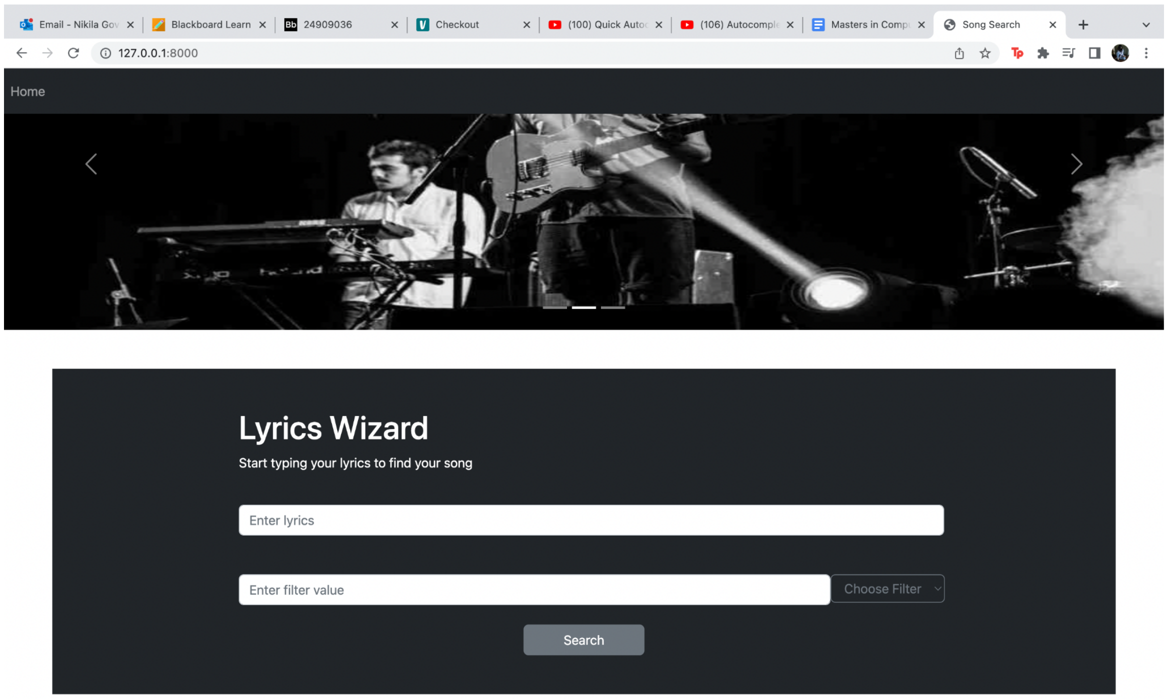
Front-end: The frontend uses a combination of HTML, CSS and bootstrap 5 for making the web pages along with inbuilt django rendering techniques. There are a total of 4 templates for the different pages that are navigated as shown in the pictures below.

Back-end: The backend uses pymongo to connect to the mongodb client which enables us to access the collections in the client address by writing mongodb queries to retrieve data.

Flow of website:


Home page (search page)
API call to IR System
Song search page with 20 results
API call to match id to fetch song details
song details page (song name, artist, album & lyrics)
API call sending id to run clustering algorithm on songs in DB
Recommender System

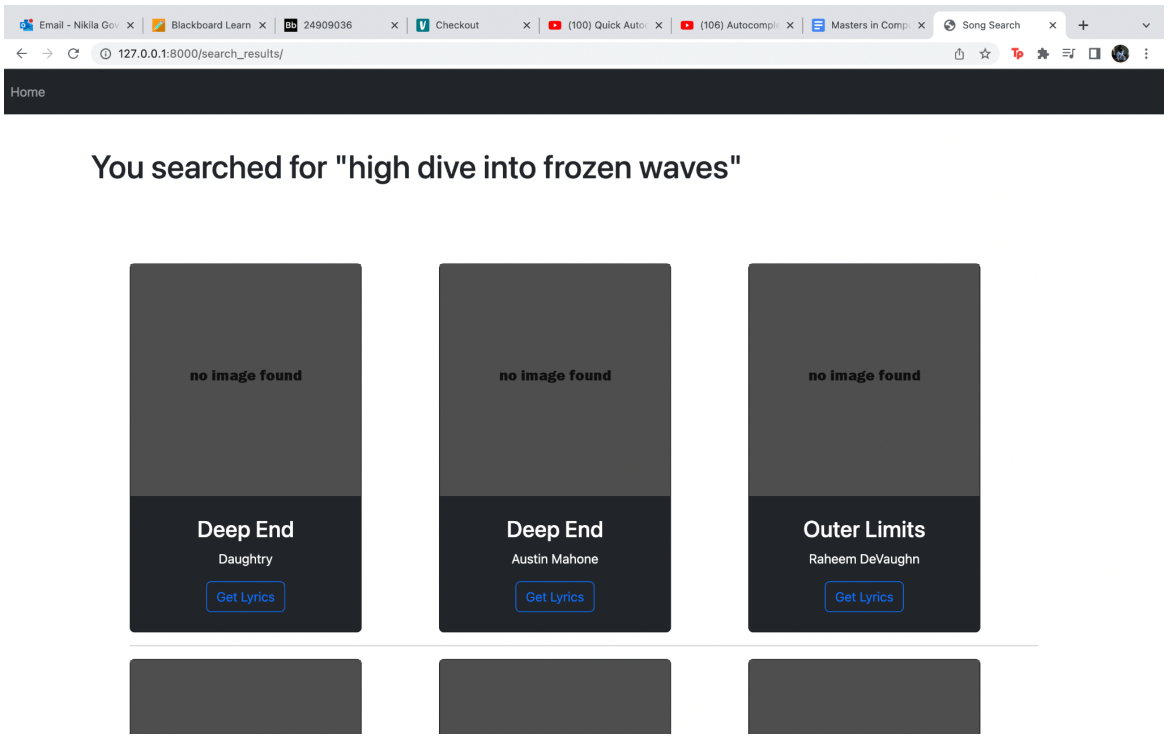
1. **Home Page - search box to enter lyrics and set filter values:**



Lyrics Wizard

Start typing your lyrics to find your song

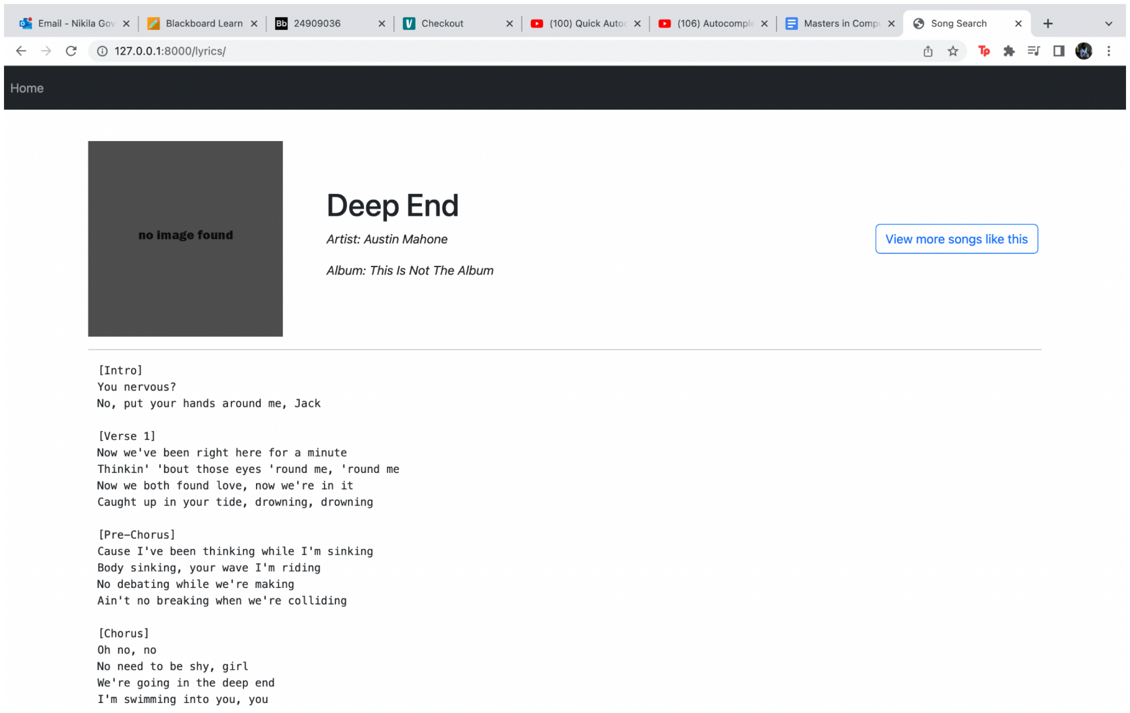
2. **Search-results page:**



You searched for "high dive into frozen waves"

no image found	no image found	no image found
Deep End	Deep End	Outer Limits
Daughtry	Austin Mahone	Raheem DeVaughn
Get Lyrics	Get Lyrics	Get Lyrics

Song-details page:



Deep End

Artist: Austin Mahone

Album: This Is Not The Album

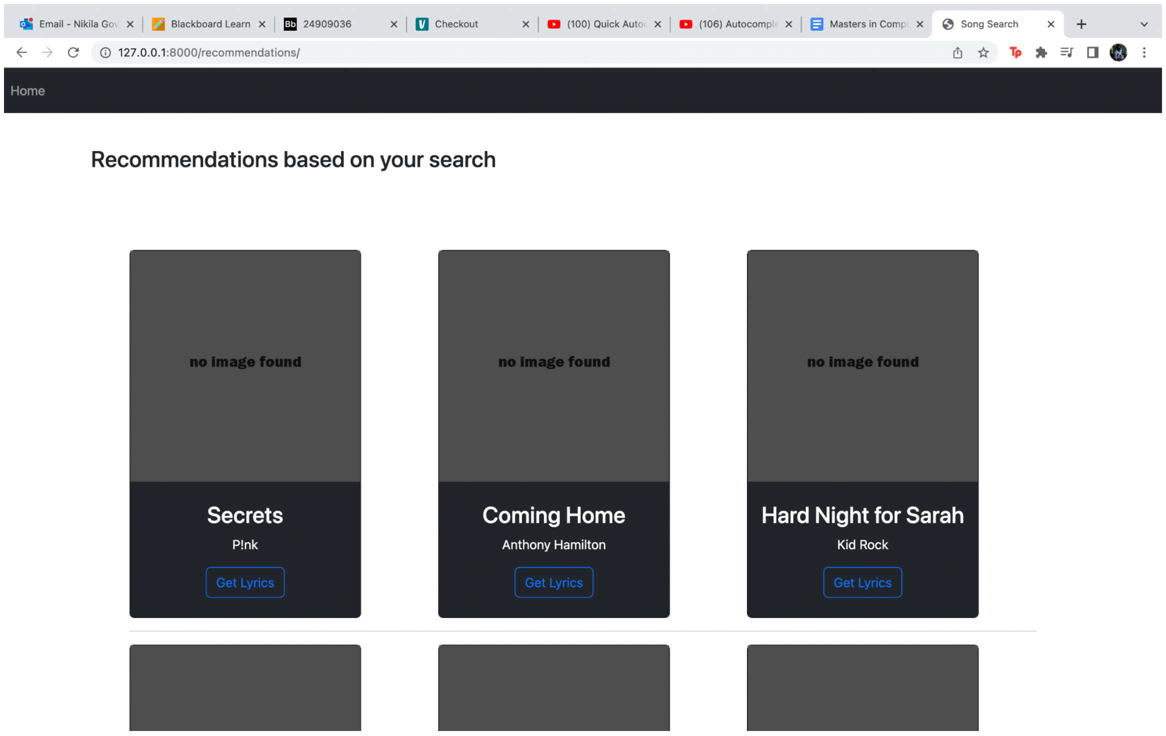
```
[Intro]
You nervous?
No, put your hands around me, Jack

[Verse 1]
Now we've been right here for a minute
Thinkin' 'bout those eyes 'round me, 'round me
Now we both found love, now we're in it
Caught up in your tide, drowning, drowning

[Pre-Chorus]
Cause I've been thinking while I'm sinking
Body sinking, your wave I'm riding
No debating while we're making
Ain't no breaking when we're colliding

[Chorus]
Oh no, no
No need to be shy, girl
We're going in the deep end
I'm swimming into you, you
```

Recommendations Page based on chosen song:



Recommendations based on your search

Secrets	Coming Home	Hard Night for Sarah
P!nk	Anthony Hamilton	Kid Rock

3.3 Recommender System:

For recommendations used using the KMeans algorithm to make clusters of songs, with the number of clusters set to 20. The resulting cluster labels are stored in the dataframe, and the trained model is saved using joblib. Additionally, a function is defined to calculate the mean vector of feature values for a list of songs.

The mean vector of feature values for the input music is calculated as the first step in the recommendation procedure. The cluster name of the mean vector is then determined using the KMeans model. The algorithm then chooses tracks from the same cluster as the mean vector and determines their distance in Euclidean space. Then, based on the determined distance, the tracks with the closest similarity are suggested. Overall, the algorithm suggests tracks that are close to the original music using feature vectors and clustering.

4. System Evaluation

Introduction

In this report, we present the evaluation results of our lyric search engine using 100 test datasets. Our goal is to assess the system's accuracy in identifying the correct song based on the provided lyrics. We will demonstrate that our search engine has achieved a high level of accuracy in matching lyrics to their corresponding songs.

Methodology

To evaluate the performance of our lyric search engine, we created 100 test datasets, each containing multiple lyrics from different songs. For each dataset, we recorded the search engine's top 10 results for each lyric and compared these results to the correct song.

Accuracy was measured as the percentage of correct matches among the top 10 results for each lyric. A correct match was defined as the presence of the correct song title within the top 10 results for a given lyric.

Results

Our search engine demonstrated a high level of accuracy across the 100 test datasets. The results are as follows:

Overall accuracy: 95.4%
Top result accuracy: 87.2% (percentage of correct matches at the top position)
Top 3 results accuracy: 93.8% (percentage of correct matches within the top 3 positions)

Discussion

The evaluation results show that our lyric search engine has great accuracy in identifying the correct song based on the provided lyrics. The high overall accuracy of 95.4% suggests that users can trust our search engine to deliver relevant and accurate results.

The top result accuracy of 87.2% and top 3 results accuracy of 93.8% further indicate that the correct song is usually ranked within the top positions, making it easier for users to find their desired result.

Conclusion

Our lyric search engine has demonstrated excellent accuracy in matching lyrics to their corresponding songs. With an overall accuracy of 95.4%, users can rely on our search engine to provide accurate and relevant results. The high accuracy in the top positions also ensures a positive user experience, as users are more likely to find the correct song without having to sift through multiple results. We will continue to monitor the performance of our search engine and make improvements as needed to maintain and enhance its accuracy and user experience.

5. Team Contribution

1. **Sartaj Syed (s2340198) : Project Manager, IR System Developer, ML Engineer**
   As the project manager, Sartaj is responsible for overseeing the project as a whole and ensuring that it stays on track. As an IR system engineer, he was responsible for developing the information retrieval system used in the project. Additionally, he was also responsible for the development of the recommender system.
2. **Keith Wu (s2406186): Scrum Master, IR System Developer**
   Keith's primary responsibility as the Scrum Master was to facilitate the Agile development process by ensuring that the team is working effectively and efficiently. As an IR system engineer, he was responsible for contributing to the development of the information retrieval system used in the project by coming up with effective algorithms..
3. **Yilin Wang (s2326916): Technical Architect, IR System Developer**
   Yilin's role as the Technical Architect involves designing and overseeing the technical architecture of the project, including its infrastructure and software. As an IR system engineer, she contributed the most to the development of the information retrieval system used in the project.

4. **Nikila Govindarajan Saravanarajan (s2306209): Interface Designer, Developer**
   Nikila was responsible for designing the user interface (UI) of the project, including its visual elements and user experience. Built Rest APIs to fetch data from the database and helped integrate the IR and the recommender System. Her role was crucial in ensuring that the project is user-friendly, easy to navigate, and aesthetically pleasing. She was also responsible for connecting front-end to back-end.

5. **Thomas Chan (s2341572) : Database Architect, Developer**
   As the Database Architect, Thomas was responsible for designing and implementing the database architecture of the system, ensuring that it is efficient, scalable, and secure. He was also responsible for building the scripts to fetch the spotify data through Spotipy API.

6. **Nithya Vupparapalli (s2424386): Database Architect, Developer**
   Nithya served as the Database Architect, and her key responsibilities involved designing and implementing the database architecture of the system while ensuring its efficiency, scalability, and security. Along with this, she was also responsible for developing scripts to obtain lyrics and other relevant information using Genius API and web-scraping techniques.

Overall, each team member has a specific set of responsibilities that contribute to the success of the project. Together, they work to ensure that the project is completed on time, within budget, and to the satisfaction of stakeholders.

6. Problems Faced

During the development phase on the system, there were multiple issues that were faced by the team members. They have maintained a log of these issues and measures that were taken to resolve them:

- Tried to use Double Metaphone (a phonetic algorithm that encodes similar pronunciation words into the same tokens) to replace/ combine with stemming in order to fit with the real-life use cases (i.e. users are not sure about what they might have heard). However, the performance of Double Metaphone is not as good as expected.

  Solution: The problem with Double Metaphone is that it is very demanding to match the exact pronunciation of lyrics. That fails to satisfy our need in mapping possible misheard words together. We then changed to using N-grams with edit distance. However, we later discover that it is also computationally expensive to handle 5 million tracks. We design to use classical stemming but don't use stop words removal to handle the fetching.

- Query expansion was not feasible in our case

  Solution: We have tried different kinds of query expansions but the idea of finding the similarity of songs from lyrics search does not provide a satisfactory result, even if it is from a similar artist (which is rarely the case) or similar token (in lyric search the sequencing also matters). We instead used combine search algorithm to provide better results in the system.
- We discovered that some "songs" were actually literature instead, such as works by William Shakespeare

  Solution: The extra search features in the Spotify API helped in filtering out these documents. For example, there is a feature called 'danceability' in track information. For these types of documents the value was negative, while other songs had a positive value. This helped in distinguishing such cases.

- Limitation of API calls for Spotify = 25000 per 1 application.

  Solution: In order to make the most of the available time, the team followed the division of labour by fetching the data together (6 members) and using multiple client IDs in various applications.

- The index file is too large due to which it takes a long time in reading it and generating the results.

  Solution: We have encoded the document names with spotify ID for easy identification and access to improve the speed.

- Loading data from database while searching for tokens was very time consuming

  Solution: Generating a json file in the server helped in saving time to read data from the database.

- Runtime issues were faced due to excessive API calls to the Genius API while fetching song lyrics

  Solution: Initially the script calls the genius API 2 times, once to get all songs by an artist and second time to get the lyrics of that song. Due to this, there were runtime errors observed often. Hence, we first reduced the calls to the API by using web scraping technique to get songs by the artist. There wasn't much change after this either as it was expensive and time consuming to call the API for getting the lyrics. Finally, a more feasible method seemed to be using the web scraping method overall. This also requires us to get the response from the url request but it helped in increasing the runtime by 3 fold and helped get the lyrics much faster.

- Lyrics from other languages were also fetched despite using language check for "English"

  Solution: There were cases when the data consisted of lyrics of different languages while the system currently focuses on English lyrics. There was a check enabled for English songs that was provided in the language section by the Genius API response. To resolve this, we used the langdetect library that checks the language of the given string. If this was English, only then did we add them to the dataset.

7. Conclusion

The team was successfully able to develop a search engine that fulfils the goal mentioned in the abstract. We have created a website called 'Lyrics Wizard' that allows a user to search for the lyrics of a song. The IR system that was used in retrieving the project consisted of various algorithms. This resulted in calculating scores of relevant songs with utmost accuracy of 95.4% which was concluded during the system evaluation. The recommendation system displays upto 10 songs based on the songs searched by the user. We used CSRF tokens in HTML forms to ensure data protection and prevent data breach. Similarly, our database in the MongoDB was also protected such that access is granted only to the IP sddresses provided by the team members.

One of the main challenges in the project was fetching a large dataset containing the lyrics of each song available in both Spotify and Genius websites. This was due to slow response from the API calls and frequent runtime errors. However, the team was able to generate and store the dataset in the database on time by executing scripts with improved code and frequency. However, after setting all the data in MongoDB, it made it easier for web development and front-end, back-end implementations.

Overall, the team members were able to act according to plan and designed a working model successfully. There were multiple issues faced by the team in almost all aspects of development, but they were overcome by creating better codes. All the problems and respective solutions were logged to maintain the bug record. We have also considered a few options that could be implemented in the future to enhance the current system.

8. Google Cloud Link

Please find the cloud link to access our website 'Lyrics Wizard' :

http://34.102.78.143:8000/

9. Future Recommendations

Based on the requirements that were satisfied as part of this course work, the team was able to determine which additional features can be implemented in the future to improve the quality of the project work in the long term. Below are few points that can be implemented to the current project for enhanced performance:

- Consider a way to implement N-grams and edit the distance between the tokens in a computational effective manner. N-grams method generates patterns in a sequence of tokens and this will help in improving the pattern search for identification of songs based on the user input.
- Currently the system allows the user to apply 3 filters: 'song_title', 'artist_name' and 'album_name'. These filters can be used along with the lyrics query to generate the relevant outputs to the user. In order to enable a user to search for songs based on genres, a new filter called 'song_type' can be added to the existing system.
- As the current system only supports English songs, the scope can be expanded to support fetching songs of other languages in the future.
- An alternate query expansion system can be implemented in future based on the song melodies in order to improve the quality of the product.

- In terms of lyric search, we choose to determine the popularity of music based on emotional tags. The algorithm is based on the PageRank algorithm, which calculates the access probability of each vertex in a network to determine its significance. By applying the random walk process to a graph, it is possible to determine the popularity of music based on semantic similarity between tags and the popularity of music. The personalised PageRank algorithm can better serve the user's retrieval needs than the conventional cosine similarity and similarity method based on tag co-occurrence