Find My Lyrics - A Search Engine for Lyrics

Taes Hahn
s2123694@ed.ac.uk

Valentine Dragan
s1710228@ed.ac.uk

Ioana Bacrau
s1738286@ed.ac.uk

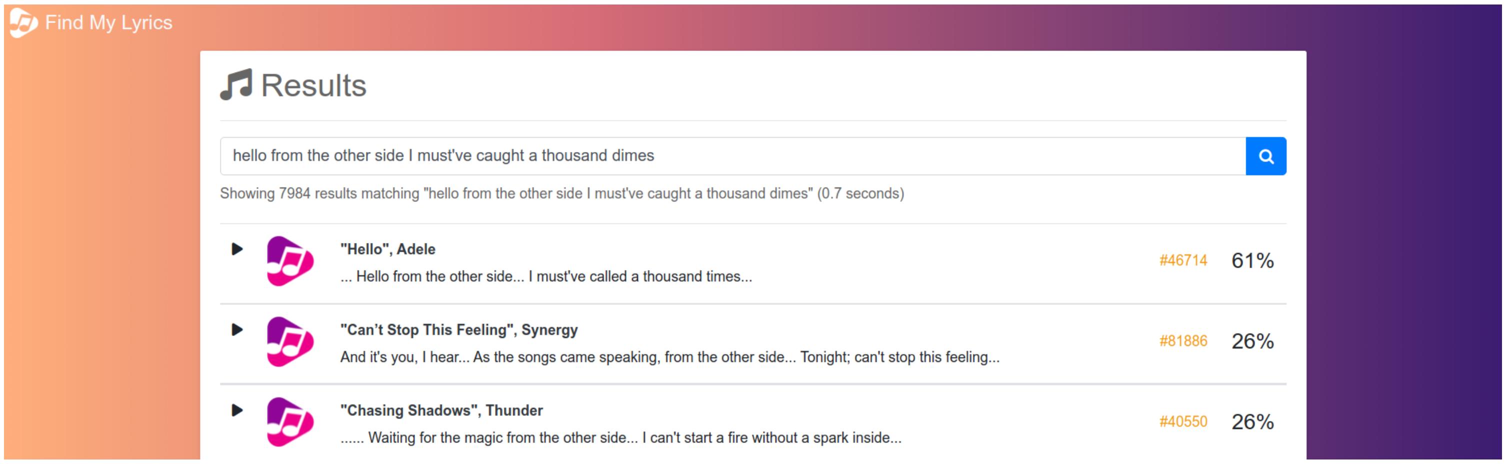
Perseas Christou
s1984267@ed.ac.uk

October 20, 2022

NOTE: PLEASE CONTACT s2123694@ed.ac.uk WHEN MARKING OUR WEBSITE, FOR US TO TURN IT BACK ON. THE URL PROVIDED IN THE SUBMISSION FORM WORKS TOO, BUT THE YOUTUBE VIDEOS DON'T WORK UNLESS WE PROVIDE A DIFFERENT DOMAIN NAME.

Contents

1. Introduction: What is Find My Lyrics?



hello from the other side I must've caught a thousand dimes

Showing 7984 results matching "hello from the other side I must've caught a thousand dimes" (0.7 seconds)

"Hello", Adele
... Hello from the other side... I must've called a thousand times...
#46714 61%

"Can't Stop This Feeling", Synergy
And it's you, I hear... As the songs came speaking, from the other side... Tonight; can't stop this feeling...
#81886 26%

"Chasing Shadows", Thunder
...... Waiting for the magic from the other side... I can't start a fire without a spark inside...
#40550 26%

Figure 1: The User Interface of Find My Lyrics

Find My Lyrics is an innovative app that solves a common problem in an intelligent way. In this day and age, people might hear a song on TV, on their commute, from friends, or even from ads and shows. But more often than not, they only remember a few words from the song - sometimes not even the right words.

Our lyrics search engine solves this problem by employing a novel search algorithm designed to find the right song even when queries aren't perfect. Our search engine combines Live Indexing, a database of over 150k records and growing, specially-designed Search & Retrieval algorithms, a content-based Recommender System, and a modern User Interface, to give users effective and efficient results!

2. System Architecture & Design

The diagram of our system architecture is shown in Figure 2 below. As we wanted to build a modular and easy-to-update search engine, we divided our system into 3 main layers:

1. The Data Layer described in Section 3 is responsible for performing Live Indexing and storing the index, and databases of song details and lyric embeddings.

2. The API Layer described in Section 4 is responsible for the searching, ranking, and retrieval processes (using the novel Sequence Matching algorithm), as well as Recommending similar songs that the user might enjoy.

3. The Web Layer described in Section 5 is responsible for rendering the User Interface and handling user interactions.

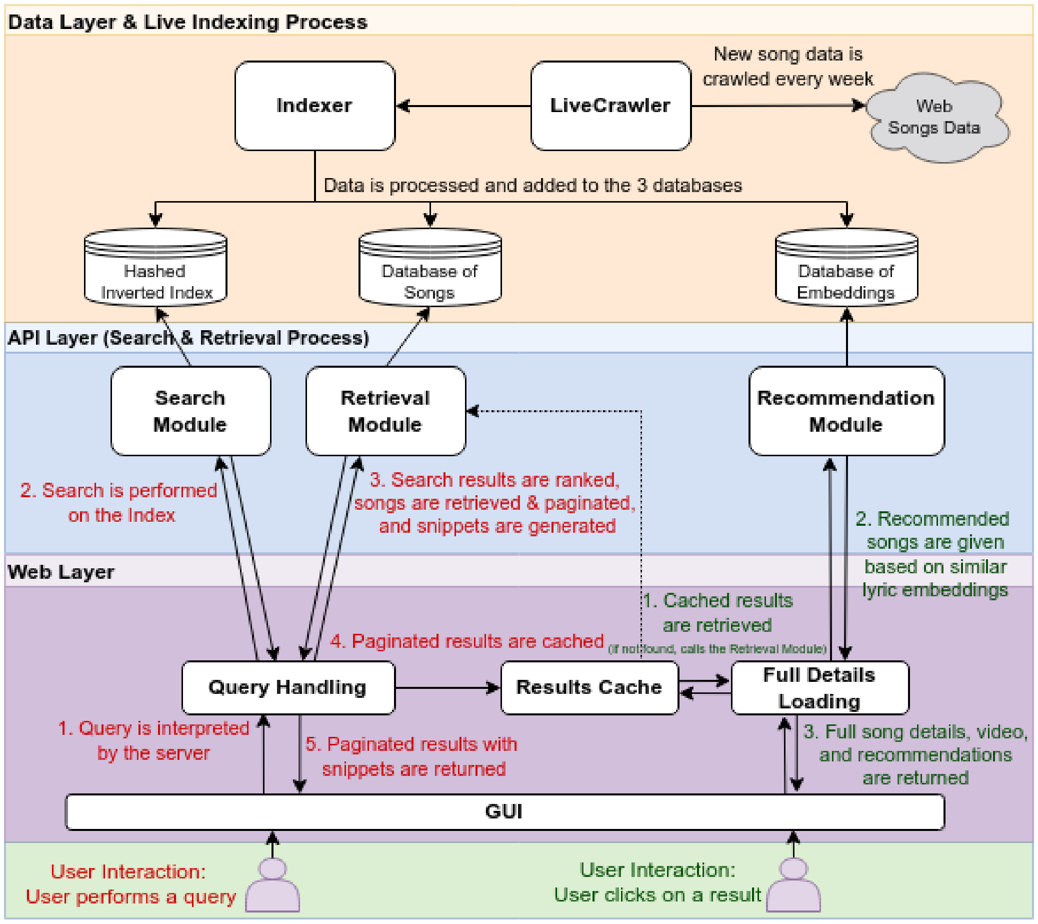

Data Layer & Live Indexing Process
Indexer
LiveCrawler
New song data is crawled every week
Web Songs Data
Data is processed and added to the 3 databases
Hashed Inverted Index
Database of Songs
Database of Embeddings
API Layer (Search & Retrieval Process)
Search Module
Retrieval Module
Recommendation Module
2. Search is performed on the Index
3. Search results are ranked, songs are retrieved & paginated, and snippets are generated
2. Recommended songs are given based on similar lyric embeddings
Web Layer
1. Cached results are retrieved
4. Paginated results are cached
(if not found, calls the Retrieval Module)
Query Handling
Results Cache
Full Details Loading
1. Query is interpreted by the server
5. Paginated results with snippets are returned
3. Full song details, video, and recommendations are returned
GUI
User Interaction: User performs a query
User Interaction: User clicks on a result

Figure 2: Our System Architecture showing the 3 modular layers: Web Layer, API Layer, Data Layer, and the 2 main interactions performed by users.

3. Data Layer & Live Indexing Process

3.1 Data Sources & Crawling

This section describes the 'Data Layer & Live Indexing Process' located at the top of our system architecture in Figure 2. We collected data on the songs released in the last decade. The metadata and lyrics of the songs were collected from two separate sources and combined. In our system, metadata and lyrics of 150,175 were collected as initial data, and those for newly released tracks are automatically updated and collected every week. Our system has been implemented to be able to retrieve based on initial data and additionally collected data for new tracks. The detailed conditions for collecting are as follow:

- Released Date: 2013 - 2022 (Recent 10 years)
- Released Country: United Kingdom, United States
- Lyrics Language: English
- Update Cycle: 1 Week (Every Monday)

3.1.1 Metadata: MusicBrainz

First, we collected metadata about the songs from MusicBrainz. MusicBrainz is an open music encyclopedia that collects music metadata and makes it available to the public. This data is provided in several forms, among which we used the JSON dump file. The Metadata for 1,615,281 tracks were collected as initial data, of which 150,175 cases of lyrics were available to be collected separately.

We utilised only a fraction of the entire data from MusicBrainz, which has a hierarchical data structure (i.e. Release > Media > Track > Recording). The items extracted from the entire data to be utilised in our system are as follows:

Data Level	**Collected Items**
RELEASE	release_id, release_title, artist, released_country, release_date, language
MEDIA	media_position, media_format, media_title, media_track_count
TRACK	track_position, track_id, track_title, track_artist_credits, track_isrcs
RECORDING	recording_isrc (International Standard Recording Code)

3.1.2 Lyrics Data: MusixMatch API

The lyrics data corresponding to the collected metadata was separately collected with ISRC (International Standard Recording Code) from another data source, MusixMatch. MusixMatch is a music data company with more than 14 million lyrics databases in 50 distinct languages. They service a developer REST API for their customers to request lyrics data from their database.

However, there are two limitations for free API. The first one is the number of daily requests to API, which means we could request up to 2000 lyrics per day. The second limitation is the length of the provided lyrics. They only provide first 30% of lyrics per song. We decided that these restrictions are not a big problem in implementing a system for academic purposes.

3.2 Processing the Data

The collected data require serveral post-processing.

- Merge data: We merge metadata and lyrics from two different data sources.
- Remove duplicates: Our collected dataset may have duplicated records. The original dataset treats multiple versions of the same song as distinct records but we want to treat them as one.
- Remove unnecessary phrase: The MusixMatch API automatically insert unnecessary phrase to our lyrics data that affect the performance of the search result. It should be deleted.
- Drop songs whose lyrics data are not available from MusixMatch API.

3.3 Live Indexing

Since new songs are released continuously, updating our database and re-indexing on those new data are very important tasks for our service. To this end, we periodically update our metadata, by comparing up-to-date data of source with that of our current system. After updating metadata, our system requests the lyrics through our second data source, Musix-Match API. And then, The Index module adds this newly collected data into our original index incrementally. All of these live indexing processes are automatically conducted and it has been scheduled on the server every Monday.

When the data is indexed, the lyrics are tokenised, lowercased and stemmed. However, stopwords are not removed because they are much more prevalent and important in songs' lyrics than normal documents. While indexing, the embeddings for each song are also generated and stored, which are utilised in Recommender Module described in Section 4.4.

4. API Layer

The API Layer contains the Search, Retrieval, and Recommendation modules as shown in the System Architecture diagram in Section 2. The algorithms used in for search & ranking have been tailored specifically to the usecase of lyrics matching, inspired by the common substring matching problem (often found in the field of bioinformatics). In the following subsections, we describe the process through which we chose these algorithms, their implementation details, and the implementation of the Recommendation system.

4.1 Designing an Algorithm for the Usecase of Lyrics Searching

Before designing the algorithms of our IR system, we found it critical to consider the common use case of such a song-finding search engine. In general, users hear an unknown song (such as on radio, advertisements, or when shopping) and upon remembering it later, they search the lyrics in an attempt to find the song. However, this use case presents some difficulties for standard IR algorithms as shown in Figure 3 below:

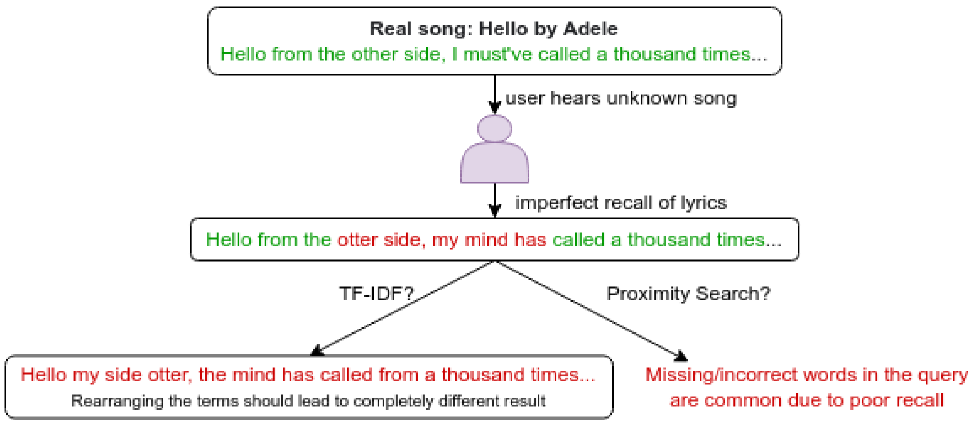

Real song: Hello by Adele
Hello from the other side, I must've called a thousand times...
user hears unknown song
imperfect recall of lyrics
Hello from the otter side, my mind has called a thousand times...
TF-IDF?
Proximity Search?
Hello my side otter, the mind has called from a thousand times...
Rearranging the terms should lead to completely different result
Missing/incorrect words in the query are common due to poor recall

Figure 3: Example of a common use-case for searching the lyrics of a song.

As shown, perfect recall of the lyrics is rare, because users often correctly remember only part of the lyrics due to imperfect memory or language misunderstanding. Secondly, the recalled words are often sequential, because users often remember a line or sequence of words, rather than words spread across the song.

These make standard TF-IDF or Proximity search algorithms less effective. To choose an algorithm suitable for our use case we have performed a "back-to-front" design thinking process. We suppose that hypothetically, we know that our correct song is "Hello" by Adele, and want to compare how different queries should score against this song relative to each other. This comparison is shown in Figure 4 below.

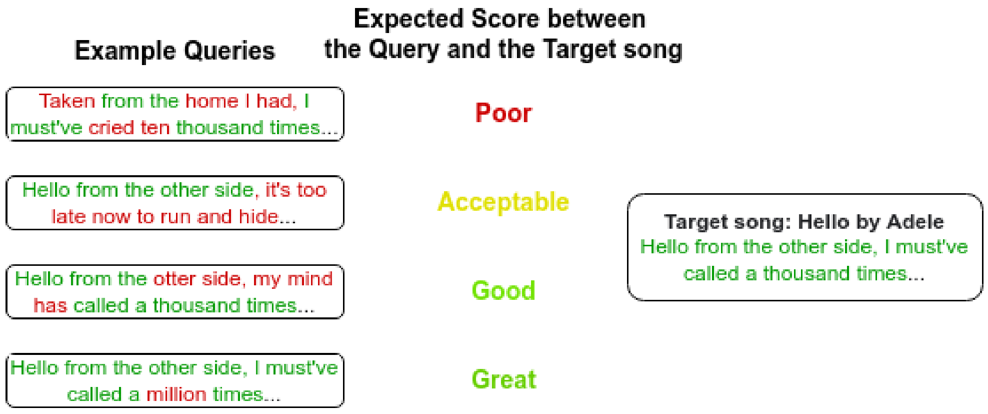

Example Queries
Expected Score between the Query and the Target song
Taken from the home I had, I must've cried ten thousand times...
Poor
Hello from the other side, it's too late now to run and hide...
Acceptable
Hello from the otter side, my mind has called a thousand times...
Good
Hello from the other side, I must've called a million times...
Great
Target song: Hello by Adele
Hello from the other side, I must've called a thousand times...

Figure 4: Example of the scores we would expect between our target song and different queries.

Following this process, we argue that our search & ranking algorithms should favour long continuous sequences of matching words between them and the target songs, while also taking into account that there might be gaps of words that don't match throughout the query. We describe the implementation of this search algorithm below.

4.2 Search Module: Sequence Matching Search and Boolean Search

As justified by the usecase above, the Search Module implements a novel Sequence Matching algorithm which is inspired from the Common Substring Matching problem.

Algorithm 1 Sequence Matching Algorithm, based on Common Substring Matching

```
Given a list of processed query terms $Q$ and a song $S$, we want to compute the longest matching sequences between $Q$ and $S$
$M \leftarrow \{\}$                                   ▷ list of matching sequences between $Q$ and $S$
for each term $Q_i$ in the query do
    for each position $p_j$ at which the term $Q_i$ appears in the song $S$ do
        while $Q_{i+1}$ also appears on $p_{j+1}$ do     ▷ $O(1)$ lookup with hashed inverted index
            Extend the matching sequence
    Add the matching sequence to $M$
Remove overlapping sequences from $M$ that are caused by repeating lyrics (using Interval Merging algorithm)     ▷ $O(M * log(M))$, but M is negligible in size on average

Algorithm Complexity: $O(Q * p + M * log(M)) \approx O(Q * p)$
```

The algorithm above is used to calculate the common matching sequences between a query Q and a song S. To further increase the speed of the Search Module, two optimisation choices were made. Firstly, an initial lookup is performed on the Index to select songs which contain at least 30% of the query terms (an experimentally chosen threshold), so that only songs with ”potential to be high-scoring” are searched. Secondly, using a hashed inverted index allows us to compute the maximum matching sequence in negligible time.

Boolean Search. In the rare cases that no results have been found through the Sequence Matching Search (sometimes due to very short queries), we then employ a Boolean Search and return the results that contain the most unique terms from the query.

4.3 Ranking & Retrieval Module

The Ranking and Retrieval Module ranks the results from the Search Module according to their found matches, and retrieves the details of these songs from the database. The score between a query Q and a song S is calculated as a percentage (between 0% - 100%), and given the list of matched sequences M is calculated as:

$$\sum_{m \in M} \frac{|m|^{1.5}}{|Q|^{1.5}}$$

After the results are ranked, the details of the paginated songs (as described in Section 5) are retrieved from the database and snippets are generated to display the matching lyrics.

4.4 Song Recommendation Module

Motivation and General Idea.

In order to make our search engine more useful to users, we added a content-based Song Recommendation System to our website. This is particularly important since finding lyrics for songs by itself does not provide further usage to our users apart from displaying good results. Therefore, we decided to make our website more interactive, and let our users explore new songs with similar semantics and topics to the ones they search for.

Our Recommendation System is based on Sentence Encodings of the lyrics (using Google’s Universal Sentence Encoder), and computing cosine similarity between encodings.

Implementation.

In order to implement the Recommendation System, we first used Google’s Universal Sentence Encoder during the Live Indexing Process described in Section 3 to create embeddings for each song’s lyrics, and stored them in a Database of Embeddings. The encoder is trained in such a way that it can encode big sentences into high dimensional vectors which capture the words and context of the sentences, making them very useful for text classification, semantic similarity and other NLP tasks.

Once we have embeddings for every song, we can compute the similarity between two songs as the cosine similarity between their embedding vectors. We have found this method to be quite effective at capturing the similar contexts and semantics of songs. Therefore, when a user clicks on a song result, we use this method to recommend similar songs.

However, computing the similarity between an entire database of 150,000+ embeddings (each being a 512-dimensional vector) would be highly inefficient in terms of time. We also found, experimentally, that there is a large number of songs with a "great similarity score" (between 0.7 - 0.8), with 1 in 700 songs having such a similarity score. Therefore, to ensure that our recommendations load up in less than 1 second while also providing good recommendations, we only select a sample of 2000 songs to compute the similarities on. From these, the Top 6 songs in terms of similarity score are recommended to the user, via links that take them to the songs' lyrics and youtube videos.

We have also experimented with dimensionality reduction of the embedding vectors using PCA (principle component analysis) to reduce the computation time, but we found our recommendation results to be better when using the entire 512-dimensional embeddings. As further work, however, we could perform the computation of cosine similarities for each song during indexing, and store in the database the top recommended songs for each song.

5. Web Layer & User Experience

5.1 User Experience

For our web app design, we have paid great attention to the UI/UX aspect. We wanted to create a user interface that is new and invigorating and provides an improved experience for our users, however we also wanted to allow the user to feel familiar with our design.

For our UI we have analysed our biggest competitors in this area (Google, Genius and YouTube) and have taken inspiration from their designs, and furthermore for our code we are using the best and most popular front-end open source toolkit, Bootstrap. We also wanted to set our own visual mark, so we have put together a logo to help brand our business, but also used a gradient for our background that is consistent and easily identifiable through all of our web pages. Another important aspect for our UX was the color pallet for our website: we are using the orange to purple gradient for the background, orange for special keys (such as song ids), a mix of grey's for our other content, and blue for important navigation items such as search buttons and pagination elements.

5.2 Query Handling & Displaying the Results

For our home page and results page we have used a mix of strategies and models implemented by our best competitor in the area of querying and displaying results, Google, but we have also added our own interpretation of what elements and items are important based on our business model.

Our home page aesthetically pleasing and minimalist, and it satisfies it's mail goal: introducing the user to our product and providing the initial search functionality.

Our results page contains a search bar and a paginated list of results (from our Search Module and Retrieval Module) as well as a results counter and a display of our query speed result. Each individual result displays the song's title, author, id, match score as well as a snippet of the lyrics that matched the user's query.

5.3 Expanding a Results with Full Details

By clicking on the name of an item in the results page, our users can navigate to the details page of that song. Our song page is split in 2 sides:

- Left: Lyrics - the main attraction in our business model. To add character we have decided to opt for a lyrics display straight onto our distinct background.
- Right: Video, search and recommendations - we're outsourcing the video from YouTube, and the recommendations list gets generated by the Recommendation Module.

5.4 Web pages audit results

To test our pages we have used Google Chrome' DevTools Lighthouse. Table 1 below shows our audit results in terms of Performance, Accessibility, and Best Practices, against our main 3 competitors.

Lighthouse	Performance			Accessibility			Best Practices		
Pages	Home	Result	Item	Home	Result	Item	Home	Result	Item
FindMyLyrics	96	90	83	95	82	83	90	89	67
Google	100	90	92	99	78	92	-	-	-
YouTube	41	88	92	50	89	92	32	69	92
Genius	79	76	83	58	84	75	59	79	75

Table 1: Page audit comparison

6. Staging and Deploying

We have used GCP to collaborate and host our project. The entire project complete with data files and other resources is hosted on our Google VM. Initially we planned to deploy both the code and the configuration to the Google App Engine server. However our deployable service's dependencies turned out to be not supported by the cloud. We have remediated this issue by configuring our Google VM to be a staging environment, running the app on our Flask local server and tunneling the port that the app is running on with Ngrok (this has also created a DNS for our app which is a dependency for YouTube's API). Our app is now available at 3 points: VM external IP, and 2 Ngrok DNS' over both http and https.