# TTDS Group 20 Project Report

This report details the functionality of our custom Song Lyric Search Engine developed for the *Text Technologies for Data Science* group project. Our website serves as an intuitive GUI allowing the user to perform boolean, phrase, proximity, and ranked lyrics queries in English and Spanish, all whilst filtering by artist, album, and mood is allowed. We have logically split this report into the two components making up our solution: frontend and backend, within which we explore some of the key technologies and approaches we use in our solution. The report then describes the testing suites performed on the developed website and finally concludes with the individual contributions of each group member.

# 1. Backend

## 1.1 Framework

The backend server is developed using Flask, a micro-web framework written in Python.
For this project the backend has a few use cases:

- Managing the database
- Serving results to the frontend
- Starting the scraper and saving (indexing) the data returned by the scraper
- Making calls to the IR tool where necessary
- Providing easy access to database data for other classes

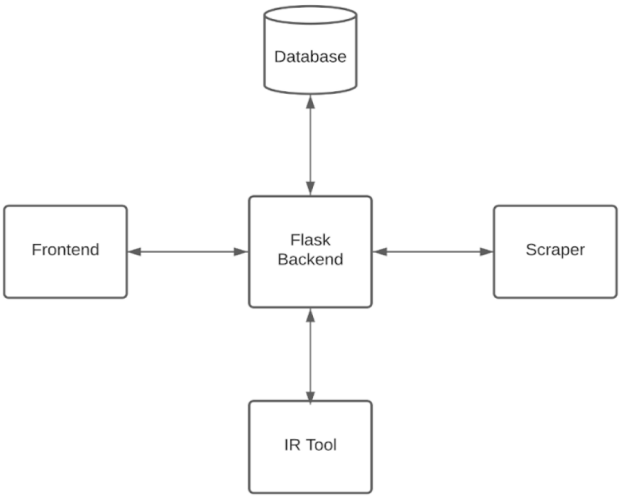

Figure 1: Flowchart of solution components and their relationships.

## 1.2 Database

The database is a PostgreSQL database hosted on Google Cloud. It has 8 tables:

**Songs:** This table contains information about the songs. After a song is scraped, it is saved here with an incrementing unique id as primary key and other information like its title, artist etc. The information about the album of the song is kept in a separate table and a foreign key is used in this table to retain the necessary information about the album a song belongs to.
**Albums:** This table contains information about the albums. Each album has an incrementing unique id as primary key, as well as name, year, and cover (album cover image URL) information.
**Index, Artist_Index, Title_Index:** These three tables contain inverted indexes for lyrics, artist names, and song titles. The lyrics, artist names and titles of each song are indexed by the backend after they are scraped.
**Spanish_Index, Spanish_Artist_Index, Spanish_Title_Index:** These three tables are just like the ones above, but they are used for the Spanish songs.

## 1.3 Web scraping

The project scrapes both lyrics and tracks' metadata from two sources: *lyricsondemand.com* for primarily English songs and *musica.com* for primarily Spanish songs. We have checked that scraping is not explicitly prohibited in the terms and conditions of both sites. Together with a track's title and lyrics, we also scrape its album and any associated album cover (if present), year of

release, as well as the URL to the lyrics page on either website. Thus, we do not host copyrighted lyrics in our database, and our search engine acts as an external aggregator to help users look up songs on the two websites.

We follow the same process to scrape both websites. The scraper starts by getting the lists of artists and the URLs to their discography pages in alphabetical order. Discography pages on *lyricsondemand.com* contain a list of albums (with album covers and years of release) and their tracks, with singles listed separately. The scraper extracts all the metadata for each track from the discography pages, saving the singles' titles as album names, before scraping the lyrics pages. Discography pages on *musica.com* merely list an artist's tracks and the URLs to their lyrics pages. Metadata like album name, album cover, and year of release are extracted, if they exist, together with the lyrics on the lyrics pages. If the lyrics for a particular song in *musica.com* are not detected as Spanish, the song is not added to the index. Otherwise, the scraper then uses the trained mood classifier to predict the mood of the song, and packages a song with its metadata, lyrics, and mood into a tuple and saves this to the database. Lyrics are pre-processed using a project-wide algorithm; namely applying a series of filters: tokenizing, lowercasing, and stemming - using a stemmer appropriate for the language of said lyrics. The inverted indices are then constructed with token position lists indexed first by token and then by unique `songID`. Note how despite initially including stop word removal in our indices, we decided that to meet the expected functionality of phrase and proximity searches we required exact matches of lyrics and thus did not automatically remove stopwords from the indices. Moreover, to further allow the user to perform the full IR stack for both artist names and song titles we also constructed inverted indices for these without the stopword removal preprocessing step. We realized that many song names included an overwhelming proportion of stop words, and it would become too difficult to fulfil the information need of the user if they were searching for many of the songs by title (e.g "Do you want to?" by Xscape), as many of the songs would be unreachable in this way due to their overwhelming proportion of stop words in their titles.

The scraper is implemented in Python as two separate classes, one for each website. We use the library *requests* to make HTTP GET page requests, whose HTML contents are parsed into HTML tag trees with *BeautifulSoup*. Processes spawned using the built-in *multiprocessing.Pool()* are used to get and scrape different artists' tracklists in parallel. The scraper Python script is executed every week to perform live indexing of new songs on the two websites to add to the database, without affecting access to the search engine. The scraper also provides a function for the frontend to get fresh copies of retrieved songs' lyrics for snippet preview.

```
1 response = requests.get(song_link)
2 strainer = SoupStrainer("div", attrs={"id": "letra"})
3 soup = BeautifulSoup(response.content, "lxml", parse_only=strainer)
4 lyrics = soup.find("p")
```

The scraper has also been used to extract the most frequent words per corpus (i.e language) which were then used by the IR team to extract stopwords tailored to our corpora.

The current size of the corpora is summarized in Table 1 below:

| Number of | lines | unique words | artists |
|---|---|---|---|
| English | 27,508,116 | 2,815,218 | 65,722 |
| Spanish | 6,986,399 | 1,268,467 | 55,786 |

Table 1: Different size metrics of the search engine's corpora

## 1.4 Endpoints

The backend server has multiple endpoints to serve the frontend and some other functionalities like starting the scraper remotely. The three main ones are:

1) **Search**

   Route = "/api/search"

   This endpoint is called to execute searches for songs relevant to the user's query. It takes a query, type, language and returns a list of song objects. The query and the search type (if it is a lyric, title or artist search) are sent to the IR Tool along with the required tools to get data from the index table. The IR Tool returns the resulting song ids in the correct order. The songs' metadata with the corresponding ids are retrieved from the database and served to the frontend as a JSON message.

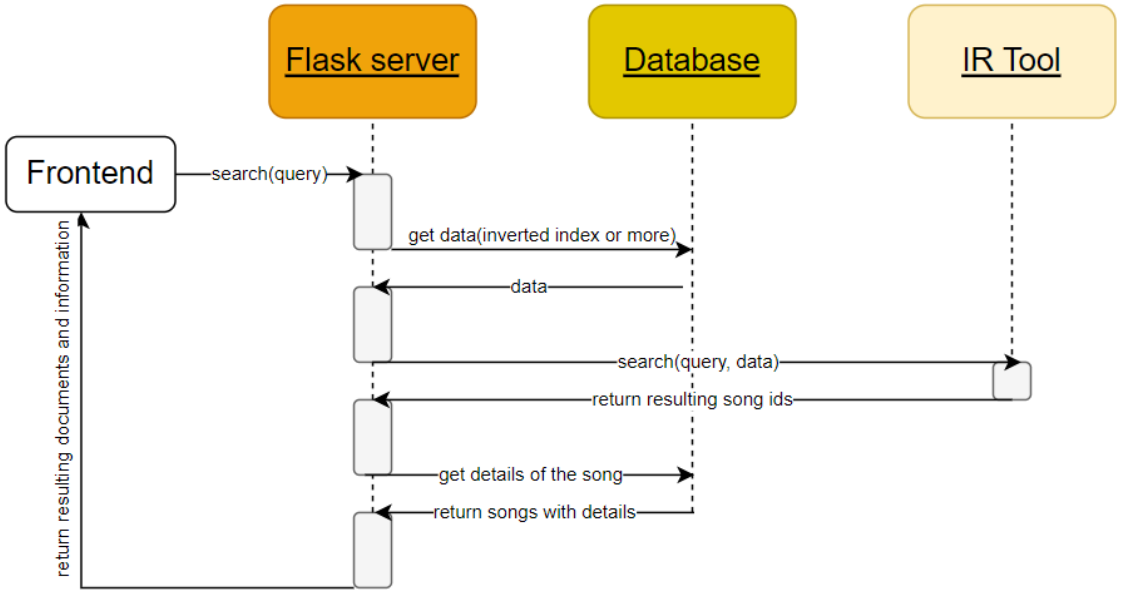


Figure 2: Endpoints' data flow diagram.

2) **Snippet**

   Route = "/api/snippet"

   This endpoint is called to get relevant lyric line snippets for songs previously found as relevant. A query, song url, type and language are received. The lyrics of the song are retrieved by calling the scraper with the song url. If the search type is a title or artist search, the first line of the lyrics is returned as a preview of song contents. If it is a lyric search, the lyrics of the song and the query are sent to the IR Tool and a relevant snippet is served to the backend.

3) **Scrape**
   Route = “api/scrape”
   This endpoint is called to start the scraper. It is mostly used to start scraping once the server is launched and to update/maintain the songs after launch.

## 1.5 IR tool

We began with an extensive refactoring of the best existing IR tool from previous coursework as a backbone, now using the OOP paradigm. This was done to improve maintainability and to make it more modular, allowing us to more easily extend functionality like swapping the inverted index object to perform our searches not just by Lyrics but by Artist or Song Title too (allowing us to search in both the English and Spanish lyrics corpora).

Then we improved the barebones functionality of the different search types (the type of which is dynamically parsed from the query format) as follows: 1. Boolean: after making a bespoke boolean parser (which reads the logic left to right), we engineered it to allow execution of any of the other query types in between boolean operators (AND, OR and NOT). 2. Proximity: built a custom parser using regular expressions to allow some leeway in formatting a proximity search by the user (e.g the use of whitespaces erroneously). 3. Phrase: increased the number and length of the phrase to 2 words or longer, also utilizing an asymmetrical 1-word proximity search under the hood, simplifying both logic and execution. 4. Ranked: the backbone of most of our expected traffic (catering to the information need of a more general user), we investigated different ranking algorithms such as modified TFIDF and BM25 and realized the best performing algorithm was TFIDF for base searches (finding relevant songs) while BM25 is more appropriate for shorter documents such as for song snippet search. Some issues arose here, notably when including BM25 as a valid scoring criterion[8], this involved adapting the data being scraped (and how the inverted indices were scraped) to feed ranked queries information about song lengths needed by BM25, both per song and on average.

Moreover, we realized that to truly capture the information need of the user, it would be useful to be able to perform IR not only to find relevant songs, but also to be able to find relevant lyrics lines (or snippets) within selected songs according to the submitted query. This snippet IR is only performed for “Lyrics” searches, as when looking for artists or song titles, there is no explicit relevance of any one specific song line, and instead, the top line of the lyrics is returned to give the user a preview of the song contents. For Lyrics searches however, we created an adapted IR mini-tool, similar to the one used for song retrieval but that was also able to dynamically construct lyric inverted indices when called on particular songs by the front end. There were some challenges arising here, such as how to perform this snippet search using boolean or say, proximity search queries. We came up with a consistent retrieval algorithm where we turned the original query into a reformulated ranked query that still captured the same information need from the user. As an example, for 2-word proximity search queries, we turned said query into a 2-word ranked search that tried to pick the most relevant lyric line. This was a way to capture the useful part of the song (line) in which the proximity occurred (or was close to), since when searching for originally relevant songs the proximity match could have occurred over multiple lines, and it would have proved prohibitively slow to construct the complex inverted indices to hone in on a multi-line snippet containing the exact proximity search. Note: we were unsatisfied with the results of TFIDF on shorter documents (i.e the individual lines in snippet search), namely we noticed BM25 performing better, likely due to BM25’s preference for documents matching more terms in a multi-term query over documents with more matches of only some terms in the multi-term query[10]. This comes as a consequence of

BM25 dealing with term saturation through its TF/(TF+k) term (where k is itself a function of the document length[5]). In summary TFIDF was performed to find relevant songs for the query (limited the number of calls to the database of inverted indices, as it needs less parameters/information -incidentally the largest speed bottleneck) while BM25 was chosen to find relevant snippets (lines) within the songs due to its ability to find more complete matches (which better represent a user's need for a line closely matching their query).

Lastly, to improve the IR tool there were a few other adjustments we needed to make. For example, needing to adapt the preprocessing to deal with Spanish vs English corpora. Notably, despite using the Porter Stemmer algorithm for the English corpora, we used the Snowball (Spanish) Stemmer for the Spanish corpora, which was able to deal with both Spanish-specific characters[11] (e.g accented words) and also specific verb and residual suffixes[6] that were not integral to a word's meaning and needed to be stemmed. Finally, significant work was put in order to analyze the most common scraped words per language corpora in order to adapt our language-specific stop words list modifying and extending nltk's standard ones. We fitted the most common scraped words to Zipf's Law, and used an elbow test[9] to ascertain the turning point of "statistically stop-like tokens", i.e removing those carrying little "semantic" information, thus lacking differentiation power between relevant songs or lyrics. Note however, that to allow phrase and proximity searches to exactly match a lyric line the user might remember (and thus had an information need to retrieve it exactly) we chose to not discard stop words for these two query types only.

## 1.6 Text Classification

As a fun yet useful feature, we have a text classification model that classifies songs into four moods: sad, angry, relaxed, and happy. The labeled dataset used to train the model is the MoodyLyrics Dataset which consists of over 2000 songs categorized into 4 moods manually by Russel's model (for further details of the dataset please refer to [5]). Initially, three popular text classification models were investigated, unfortunately, Naive Bayes classifiers failed to converge despite increasing maximum iterations and performing data scaling. Therefore, we continued with Linear Support Vector machines (LinearSVC) and Logistic Regression (LogReg) classifiers. Firstly, we trained the models on raw data converted into bag-of-words (BOW) format to have baseline statistics. Later, we tuned the hyperparameters of each model with the grid search method, scaled the BOW format with a TFIDF transformer, removed stopwords, and stemmed the lyrics; all leading to the improvements shown in Fig 3. All the statistics in the table below refer to mean f-1 scores of the specified models, calculated with 5-fold cross-validation method. Looking at the results, the improved LogReg model was chosen to classify lyrics. Each song is classified after it is scraped and saved into the database with the predicted mood.

| score | LinearSVC | | | | LogisticRegression | | | |
|---|---|---|---|---|---|---|---|---|
| | baseline | baseline | improved | improved | baseline | baseline | improved | improved |
| | train | test | train | test | train | test | train | test |
| f1-macro | 0.69 | 0.64 | 0.80 | **0.72** | 0.71 | 0.65 | 0.81 | **0.76** |

Figure 3: Improvement results for mood classifier.

# 2. Frontend

The user interface for the application is a website built using the React library. This library was chosen to ease the development process as it provides a basic file structure already set up to begin using. The application can be easily split into different components and the state of these components can be easily tracked, making the creation of interactive components easier. During development, this React application was connected to the backend server using a proxy as they ran on separate ports. For production, the application was built and included in the server as static build files which are served on the server's port.

When a user visits the website, they can select whether their query will include lyrics, an artist, or a song title and whether they are searching in English songs or Spanish songs. The default arrangement is a search for English lyrics. They can then enter their query directly for a ranked search, using quotations for a phrase search, including AND, OR and NOT for a boolean search, or in the format #?(query1, query2) for a proximity search where '?' is the maximum distance of the two phrases.

While typing a query, suggestions are made for completing the current word, or upon a space, suggesting the next word in the query. This is achieved using unigrams, bigrams, and trigrams collected from a corpus of 100,000 lines from song lyrics. The counts of these n-grams are used to estimate the probability of each word appearing in the context of the previous n-1 words. The word in the vocabulary with the highest probability for each n is then proposed as a suggestion. The user can then select one of these suggestions which will populate the search bar. The code for this method was adapted from Budvin Chathura's Medium article [6]. If spelling mistakes are noted in the query, these will be underlined in red and right-clicking will offer browser suggestions for how to correct these spelling mistakes. Upon submission of the search form, a GET request is sent to the server including the query, language, and the type of query (lyric, artist, or song title).

After IR completes, the server returns the song title, a URL to view the full lyrics, a URL for the album cover if one exists, the artist, album, release year, and the classified mood. These results are displayed in a list across multiple pages. If a URL for the album cover is not found, a default image is displayed in its place.

Upon rendering of a result, another request is sent by each result component to the server. This request includes the query and the URL of the song and retrieves the most relevant snippet from the song to be displayed. Splitting the tasks up in this way allows the results to be displayed faster and each snippet to be fetched separately such that the top results will display snippets faster than those at the end which won't be on display yet anyway.

Once results have been displayed, the user can filter for certain artists, albums, and moods by selecting the checkboxes on the left. They can also move to the next page using the navigation at the top and bottom of the list if they cannot find the desired song on the first page.

# 3. Full-System Tests

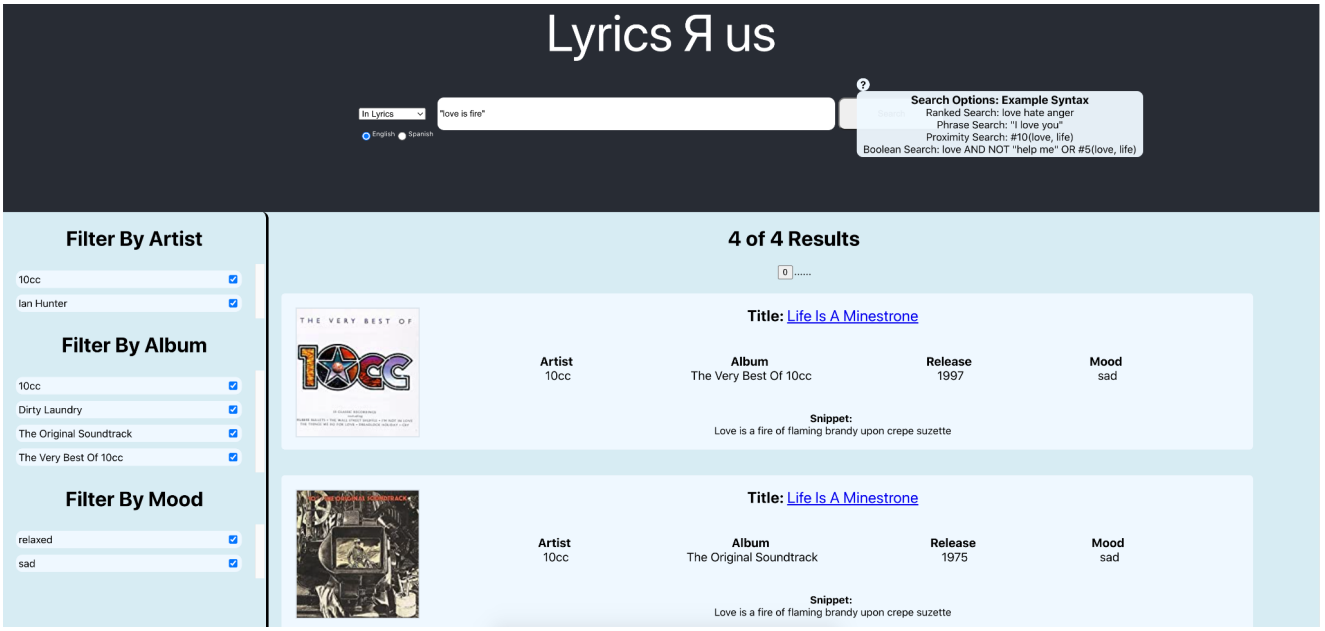


Figure 4: Searching for a phrase in song lyrics

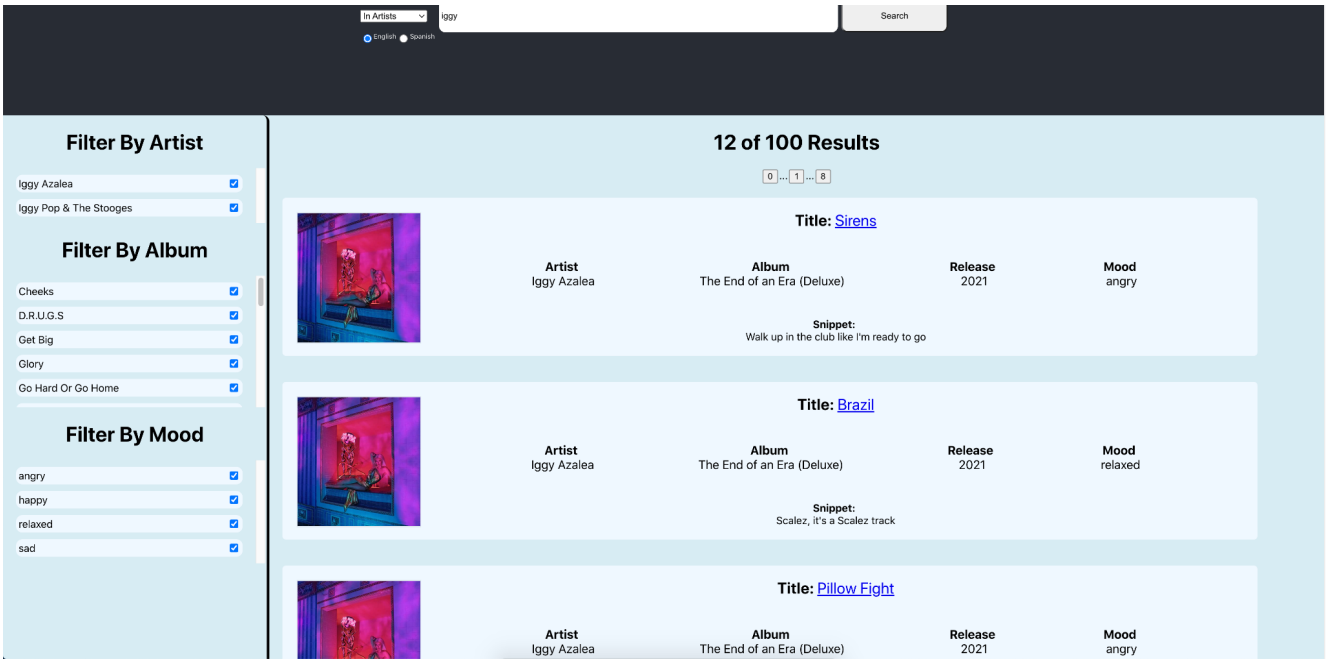


Figure 5: Searching for songs of an artist

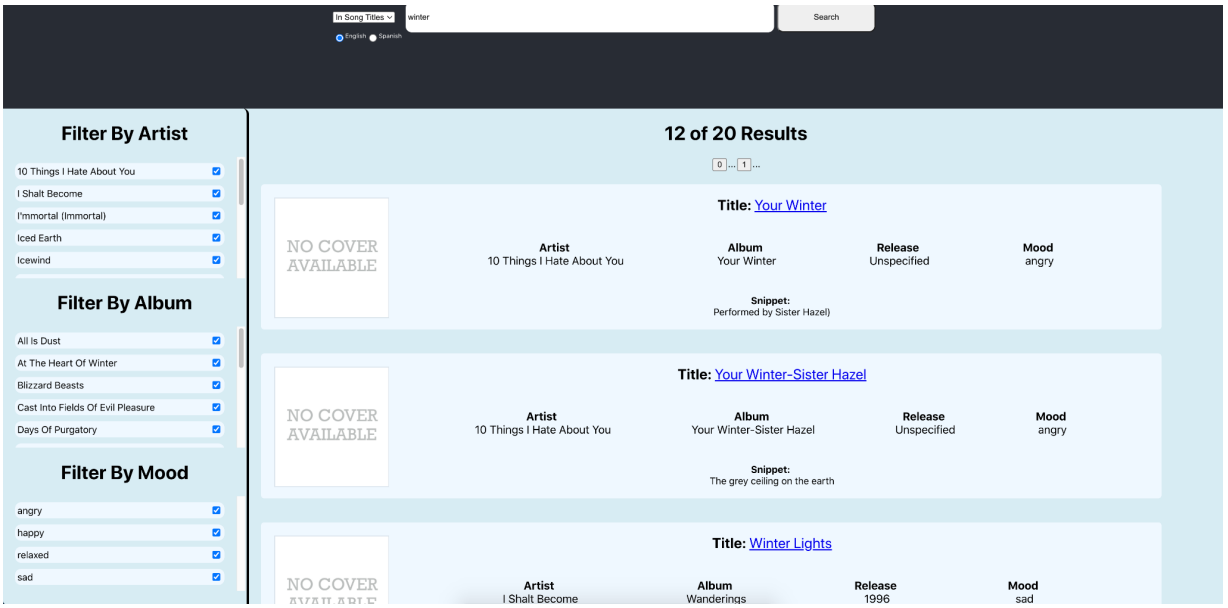


Figure 6: Searching for songs containing the word “winter”

# 4. Individual Contributions

### Anh (s1869527) - Backend: Scraper Team

Trialed multiple lyrics websites to find the most suitable candidate for lyrics and track metadata scraping. Worked with Michael to design the structure for the scraper and implemented the scraper from scratch in Python. Refactored musicascaper and maintained both while testing with other components throughout the project. Communicated with all other teams to decide what helpful data could be extracted from the websites and in which formats they would be saved in, as well as implemented on-demand lyrics scraping for snippet preview. Refactored the function used to save scraped data to the database, plus some query functions. Provided feedback for frontend design.

### Caitlin (s1836526) - Frontend

Set up the React application with a proxy to connect to server endpoints. Provided dynamic search functionality including custom options. Played a principal role in frontend GUI design. Displayed results in desired format and provided ability to move between pages of results. Provided functionality for filtering results based on artist, album, and mood. Adapted query suggestion code to work within the browser as the code provided used functions which were not browser-incompatible. Also adapted to work with React Components and specific query needs (word completion as well as next word suggestion). Ensured English songs were not added to the Spanish database. Adapted server for serving frontend build files without use of proxy to ease deployment.

### Michael (s1751472) - Backend: Scraper Team

Trialed lyric websites with Anh for the English scraper, and websites for the Spanish scraper. Me and Anh then worked to design and implement the scraper from scratch, after first learning the structure of the websites. I implemented the musica scraper, as well as the scraper controller, providing a single point of entry for the two scrapers. I modified the two scrapers to each generate a text file of every lyric from their respective websites, and then parsed these to count unique word frequencies. These word-frequency pairs were then analyzed for stop words.

### Ricardo (s1802059) - IR Tool

Refactored my existing coursework code to form the IR backbone. Expanded query functionality (e.g boolean subqueries). Added separate snippet search IR module (inc. TFIDF/BM25 performance analysis with others). Adapted preprocessing to language-specific corpora, including stopword analysis using Zipf's law, and language-specific stemmers. Performed part of unit tests for IR-related functionality: all query types (inc. logic flow) and integration tests for communication between IR tool, flask server and frontend GUI. Finally, encouraged the team to use productivity and project management tools (e.g Jira Scrum Board, Teams), often as pseudo scrum master, e.g signposting meetings alongside other team members or kick-starting the first project meeting, where I summarized project objectives and timeframes.

### Tutku (s2259604) - IR / Classifier

Started working with Ricky on the IR tool in the first couple of weeks, later assisted him with the performance analyses of two scoring methods (TFDIF vs BM25). Performed a principal role researching the extra features that we implemented and explored. Built the mood classifier model based on the Mood Lyrics Dataset I found from my literature review which Ufuk integrated into the scraper. Also found the article that Caitlin adopted to make query suggestions with n-grams. Lastly had report editing oversight and took responsibility for correct referencing and following a professional project structure.

### Ufuk (s1804378) - Backend Framework and Management

I was responsible for the backend management. I used Flask to create the basic web server and created all the endpoints, routers, and views. Also created the model file which is responsible for generating the database. Created and managed multiple methods for each model, like indexing for the indexer, or turning a Song object into a json object for the frontend and more. Provided the scraper team with a save method that takes the scraped data(lyrics, artist name, album…) and saves them all into the database with correct format and indexing. Integrated the mood classifier made by Tutku to the save function. Created a PostgreSQL database on Google Cloud and managed the connection. My main objective was to make everyone else that worked on the backend not need any Flask knowledge to do any of their work. This was achieved by creating a basic main.py file to run the backend and creating easy to understand functions whenever my friends had to do anything that involved Flask or the database, and a lot of communication. I was also responsible for all the deployment to Google Cloud. Created the Dockerfiles, created firewall rules and managed the VM that hosted our server.

# References


[1] (n.d.). Lyrics On Demand - Song Lyrics, Lyrics of Songs, Free Lyrics, Free Song Lyrics, Country Lyrics, Hip Hop Lyrics, Rock Lyrics, Christian Lyrics, Music Lyrics. Retrieved March 18, 2022, from https://lyricsondemand.com/

[2] (n.d.). Toda la música en MUSICA.COM. Retrieved March 18, 2022, from https://www.musica.com/

[3] (n.d.). Requests: HTTP for Humans™ — Requests 2.27.1 documentation. Retrieved March 18, 2022, from https://docs.python-requests.org/en/latest/

[4] *Beautiful Soup Documentation — Beautiful Soup 4.9.0 documentation*. (n.d.). Crummy. Retrieved March 18, 2022, from https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[5] Çano, E., & Morisio, M. (n.d.). MoodyLyrics: A Sentiment Annotated Lyrics Dataset. *Conference: ISMSI 2017, International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence*. 10.1145/3059336.3059340

[6] Chatura, B. (2020, July 22). *Build a simple autocomplete model with your own Google search history*. Medium. Retrieved March 9, 2022, from https://medium.com/analytics-vidhya/build-a-simple-autocomplete-model-with-your-own-google-search-history-ead26b3b6bd4

[7] *multiprocessing — Process-based parallelism — Python 3.10.3 documentation*. (n.d.). Python Docs. Retrieved March 18, 2022, from https://docs.python.org/3/library/multiprocessing.html#using-a-pool-of-workers

[8] *Okapi BM25*. (n.d.). Wikipedia. Retrieved March 7, 2022, from https://en.wikipedia.org/wiki/Okapi_BM25

[9] Piper, A. (n.d.). *Zipf's Law and thinking about stopwords – The Fish and the Painting*. The Fish and the Painting. Retrieved March 18, 2022, from https://r4thehumanities.home.blog/zipfs-law-and-thinking-about-stopwords/

[10] Seitz, R. (2020, March 20). *Understanding TF-IDF and BM-25 – KMW Technology*. KMW Technology. Retrieved March 10, 2022, from https://kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm-25/

[11] *Spanish stemming algorithm - Snowball*. (n.d.). snowballstem.org. Retrieved March 18, 2022, from https://snowballstem.org/algorithms/spanish/stemmer.html

[12] SpencerAqa. (2021, March 27). *Why Is Python The Best Language for Web Scraping?* SpencerAqa. Retrieved January 15, 2022, from https://spenceraqason.medium.com/why-is-python-the-best-language-for-web-scraping-ec8482bb1432