GoatGoatGo:
A music search engine

Coursework 3 - Group #60
Text Technologies for Data Science
School of Informatics, University of Edinburgh

Group members:
Alexandru-Ioan Chelba (s1865890)
Alessandro Speggiorin (s1659400)
Gregoris Georgiou (s1950427)
Kyriakos Kyriakou (s2281922)
Michal Sadowski (s1809955)
Zhenwu Wang (s2312154)

Date: Wednesday 22nd March, 2023

Contents

1 Introduction

GoatGoatGo is a music search engine built to help people quickly access their favourite songs' lyrics. More precisely, GoatGoatGo provides an intuitive interface to search songs by lyrics, title and artist. Furthermore, the search engine, designed with the goal of optimising retrieval effectiveness and relevance, provides several advanced search and filtering options to better align with the users' information needs. It can be accessed at: `http://goatgoatgo.com:3000/`.

The following section describes the system's architecture overall. Sections 3 to 6 delve into the details of the implementation of each part of the system, while the last section offers a brief overview of what has been presented and what we learnt and discusses possible areas of future work.

2 System Architecture

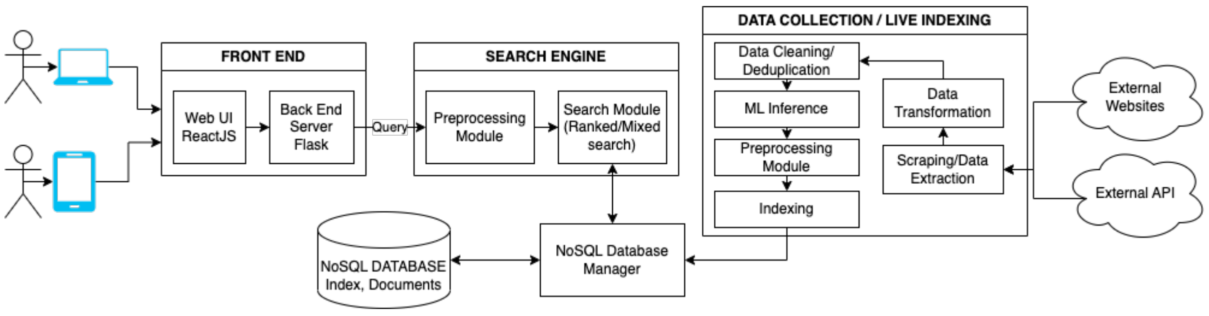


Figure 1: System Architecture

- **MongoDB Index:** In order to effectively store our indexes and documents, we opted for MongoDB [1] as the NoSQL database of choice due to its scalable and high-performant nature. However, due to some of its limitations regarding maximum individual document size, we designed an automatic self-rebalancing linked data structure to store large postings (more details in Section 3.2.3). Nonetheless, we developed a full infrastructure to easily interact with collections (Songs, Title and Lyrics indexes) as well as a Flask API [2] for data read/write access from external modules (i.e. Live Indexing) as shown in Figure 1.
- **Live Indexing & Data Collection:** The live indexing module, designed and deployed as an independent component, provides a complete, fully automated and scheduled pipeline for data collection (i.e. crawling/scraping) from external data sources and APIs, data deduplication and live indexing update.
- **Machine Learning for Genre Classification:** The ML genre classification module is an integral part of the search engine architecture that enhances live indexing by allowing genre inference for later filtering and searching of songs.
- **Search Engine Module:** The search engine module is built on a Flask API, which processes user queries and interacts with the MongoDB database to fetch relevant results. The module consists of two main components: ranked search and boolean, phrase, or proximity search. The ranked search utilises an efficient ranking algorithm, such as BM25 or TF-IDF (depending on the user's choice), to prioritise and return documents based on their relevancy to the query. On the other hand, boolean, phrase, or proximity search methods are also available for users who prefer these approaches.
  Upon receiving a search request, the Flask API processes the input JSON data to extract

[1] https://www.mongodb.com/
[2] https://flask.palletsprojects.com/en/2.2.x/

relevant search parameters, such as the user's query, search type, method, algorithm, and result limit. Based on the specified method, the API utilizes the RankedSearch or Search class to process the query and return a list of relevant song IDs. Lastly, one of the critical decisions we made in designing our system architecture was to deploy the search API and other modules on the same Virtual Machine (VM) as the MongoDB database. This decision was motivated by the need to reduce latency since accessing a database on a remote server can significantly slow down the system's response time. By deploying the search API and database on the same VM, we minimised the overhead associated with network communication and achieved faster search results for our users.

- **Flask back-end and ReactJS front-end:** We had an intermediate Flask layer between the core search API and the ReactJS front-end, which helped with the implementation of easier back-end tasks workload-wise. That is, anything we could do without needing to access the big database, we did it there. The front-end was designed with the user in mind, seeking to make it as easy as possible for the user to navigate our website.

In summary, our system architecture is designed to be scalable, high-performing, and flexible.

3 Data

This report section describes the different data processes we perform, including preprocessing, indexing techniques, self-rebalancing MongoDB indexes, a live indexing pipeline, and machine learning for genre classification. We aim to deliver an efficient and precise search engine that allows users to find songs according to their preferences.

3.1 Dataset

To build our songs search engine, we leveraged a Kaggle[3] dataset consisting of 5.9 million songs, each with attributes such as song title, artist, year, genre, features, and views. After conducting exploratory data analysis, we discovered that the dataset contained six genres, including rap, pop, r&b, country, rock, and misc. However, we noticed that the misc genre contained many irrelevant, noisy data, such as chapters of books, screenplays, and poems, which were unsuitable for our task. Despite removing the misc genre, we still encountered some irrelevant data, such as movie scripts and podcast transcripts. To ensure the accuracy and reliability of our search engine, we implemented a more stringent filtering process, retaining approximately the top 3,000 most popular artists and their songs (obtained by scraping the most popular artists from lyricsondemand.com [4]), while removing any entries with low-quality or unreliable data. By employing this approach, we were able to obtain a cleaner dataset of about 380,000 songs, which resulted in faster and more accurate search results for our users.

3.2 Preprocessing & Indexing

3.2.1 Preprocessing techniques

To retrieve information from large document collections, the document text has to be preprocessed initially. It is very significant to mention that as we aim to offer a search tool for two different type of documents, namely "title" and "lyrics", the two types of strings are preprocessed slightly differently.

The first step, only done for lyrics strings, was to remove all the text like "intro" or "verse 1" found inside square brackets in lyrics using **.sub()** since these would appear in every song and would make information retrieval worse and more time-consuming, where the index would be

[3]https://www.kaggle.com/datasets/nikhilnayak123/5-million-song-lyrics-dataset
[4]https://www.lyricsondemand.com/

much larger. Following that, for both titles and lyrics, we tokenized using **re.findall()** method, which removed any white spaces and punctuation, returning a list of all the terms left in the string. Numbers were not removed from the strings since they contain valuable information possibly contained in songs and titles, where a user might want to utilize them for search. The next step was to make all terms lowercase using **.lower()**. Next, stopwords were only removed for lyrics documents by finding what terms are stopwords and compressing them. The removal only occurred for lyrics since if they were removed from titles, all tokens appearing in a title could disappear, i.e. ”Take on me”. Since titles are usually small, stopwords contain very valuable information for a song title. It has to be mentioned that we decided to not use any kind of normalization of the data, like stemming or lemmatization, since we prioritize offering precision over recall in our results, as the user most probably would like. The nature of the context of songs, where a massive amount of them contain similar words with the same stem, i.e. loving, loved, love, would hurt the precision of our results. We understand this will hurt the results of a query containing a token with the same stem as the token appearing in the lyrics or title, but we are willing to accept this trade-off. Lastly, we decided to concatenate the artist’s name to songs titles or lyrics, which pass through the same preprocessing steps. As a result, the user can add the artist to the query for title or lyrics search to boost results and again increase precision. This addition also enables for artist search since if the artist’s name is unique and not commonly used in songs, the results will contain the songs of this artist ranked first.

3.2.2 Inverted index

The positional inverted index has to be stored effectively, so it can be accessed efficiently. Two different inverted indexes were built, one for titles and one for lyrics, having the same structure.

The inverted index was stored as a dictionary of dictionaries in a json file and then used to populate a MongoDB. Each term has its own dictionary stored in json. The first key is the ”term id”, where the corresponding value is the string of the token, and the second key is the ”doc frequency”, which has as a value an integer representing the number of documents the token appears in. The last key is ”documents”, which has as a value a dictionary where the keys are the song ids of the songs containing this term, and the values are lists of the positions of the term in the specific song. You can see an example of how one token is stored in 2. Note that the term frequency, the number of times a token appears in a specific song title or lyrics, can be found by measuring the length of the corresponding position list using **len()**.

```
{
  "_id": "<term>",
  "doc_frequency": int,
  "documents":
  {
      "<doc_id_1>": ["<pos_1>", "<pos_n>"],
      "<doc_id_n>": ["<pos_1>", "<pos_n>"]
  }
}
```

Figure 2: Example of a token stored in the inverted index

3.2.3 Self-Rebalancing MongoDB Index

In order to efficiently store the inverted index structure presented in Section 3.2.2, we adopted MongoDB as the database of choice. To be more specific, our inverted index design decision was primarily driven by the need for a scalable and efficient solution to allow fast documents and postings retrieval over an increasingly large collection of documents. Due to this fact, we

opted for MongoDB, a high-performance NoSQL database with the main advantage of providing rapid access to JSON-like formatted documents [2]. Therefore, due to the nature of our task, we decided to store all the documents under consideration (*Songs* information, *lyrics & titles* inverted indices) as separate collections in the database. More precisely, the dictionary-like structure of our indexes and documents allowed us to exploit MongoDB-specific and optimized functionalities for data storing and in-memory loading. Consequently, due to the B-Tree structure underlying MongoDB indexes, access to documents and token postings can be performed in $O(logn)$ where n is the number of unique keys in the index [3]. However, it is worth noting that the use of MongoDB is merely for document storage and efficient atomic read-and-write access. In other words, no MongoDB-specific query functionalities have been exploited for relevant document retrieval (more details in Section 4).
Furthermore, as MongoDB imposes a size limit of *16 MB* per document, we fully designed an automatic self-rebalancing data structure for storing postings exceeding the allowed limit. More precisely, postings violating such size constraints are automatically chunked into smaller partitions stored in a separate collection and linked through pointers. Therefore, as new documents are added or removed from the database, partitions are automatically increased or shrank accordingly to optimise storage and access. Due to this, our proposed solution allows to virtually grow our indexes indefinitely as new documents are introduced with the small overhead of merging postings chunks back during retrieval.

3.3 Live Indexing Pipeline

With the goal of enriching the collection of song lyrics available to users, we designed an automated pipeline for data collection and live indexing.

- **Data Collection & Scraping:** To address the issue of data gathering and collection, we developed a set of customisable modules to extract data from various sources, including publicly available lyrics APIs (i.e. Genius.com [5]) as well as websites. More precisely, we implemented a set of extendable classes for custom source crawling and data scraping. In this context, the crawler defines how to explore a source and discover new song lyrics (i.e. navigate a website roadmap) as well as enforcing a definable, constraint and *polite* policy for respectful scraping as defined by the source under consideration (i.e. the number of calls per second). Furthermore, the crawler ensures that no duplicate documents are scraped by keeping track of what has been scraped as well as matching ids against songs already stored in the database. By contrast, the scraper is only responsible for providing a customizable interface for source-specific page/endpoint extraction. As mentioned, the crawler enforces scrapers' behaviour to prevent avoidable and redundant calls.
- **Live Indexing** Documents extracted from multiple sources are then preprocessed and converted to an index-like structure by adopting the same format described in Section 3.2. Therefore, new song lyrics, as well as tokens and the corresponding postings are inserted in MongoDB through a custom Flask API. To be more specific, the API provides a communication channel with MongoDB for new song lyrics insertion, as well as automatic index updates for specific title and lyrics tokens (i.e. token posting list with the newly inserted documents, document and term frequency).

Lastly, it is noteworthy that due to its architecture, the live indexing pipeline can be fully deployed on a separate machine to avoid overloading the main search engine with underlying background processes. Hence, the pipeline is scheduled to be executed automatically every 48 hours.

[5]https://genius.com/

3.4 Machine Learning for Genre Classification

We are constantly striving to improve the live indexing pipeline discussed in Section 3.3 of our paper, and one of the ways we have achieved this is by introducing machine learning for genre classification. Our live data collection of songs did not contain genre labels, which made it difficult to filter and search for songs based on genre. To overcome this limitation, we trained a deep learning model (MLP classifier [6]) on the Kaggle static dataset [7], using the songs lyrics. Using a train-test split of 90-10 and the TF-IDF vectorizer for feature extraction, we extensively tuned the model's hyper-parameters using grid-search and cross-validation. Our preprocessing steps included removing stopwords, lowercasing, and replacing contractions (e.g., "we'll" $\rightarrow$ "we will"). Of the 380k songs available to us, we used 50k for training the model, with 10k for each genre. By shuffling and stratifying the data during the split to test-train, we achieved an average accuracy of 62%. With this successful implementation of machine learning for genre classification, we can now confidently filter and search for songs based on the genre in our live data collection. Appendix 2 shows a classification report and a confusion matrix of the results.

4 Information Retrieval

This section outlines all the types of search algorithms we provide in our search engine.

4.1 TF-IDF Search (Ranked search)

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used to evaluate the importance of a term in a document relative to the entire collection of documents. The resulting scores are used to rank documents based on their relevance to the query terms. The formula for calculating TF-IDF is:

$$score(d,q) = \sum_{t \in q \cap d} w_{t,d} \quad , where \quad w_{t,d} = 1 + log_{10} tf(t,d) \cdot log_{10}(\frac{N}{df(t)})$$

where:

- t is the term in the document d
- $tf(t,d)$ is the term frequency of term t in document d
- $df(t)$ is the document frequency of term t (number of documents it appears in)
- N is the total number of documents in the collection

4.2 TF-IDF with Cosine similarity (TFIDF + CS) (Ranked search)

As Christian Buck et al. [1] suggest, instead of adding up each term-weight to calculate the TF-IDF score (see 4.1), we can build a vector for each document using these term-weights as its elements, and compare it with the query weight vector using cosine similarity [8]. Then the documents are ranked by the highest cosine similarity with the query, which can be from 0 to 1. The cosine similarity formula used is:

$$CosineSimilarity(Query, Document) = \frac{Dotproduct(Queryweightvector, Documentweightvector)}{||Queryweightvector|| * ||Documentweightvector||}$$

[6]Used Scikit-Learn https://scikit-learn.org/stable/

[7]https://www.kaggle.com/datasets/nikhilnayak123/5-million-song-lyrics-dataset

[8]https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/

4.3 Okapi BM25 (Ranked search)

Similar to TF-IDF, BM25 [9] rewards term frequency and penalizes document frequency using the inverse document frequency formula. On the other hand, BM25 also considers document length and term frequency saturation. The BM25 score is calculated by:

$$score(d, q) = \sum_{i=1}^{|q|}(\log \frac{N - df(q_i) + 0.5}{df(q_i) + 0.5} + 1) \cdot \frac{tf(q_i, d) \cdot (k_1 + 1)}{tf(q_i, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avgdl})}$$

where the new terms introduced are:

- q_i is the ith query term q_i
- $|d|$ is the length of the document d in terms, and $avgdl$ is the average document length over all the documents of the collection, both stored and accessed in the database.
- k_1 and b are free parameters, chosen as commonly used, $k_1 = 2.0$ and $b = 0.75$.

4.4 Proximity Search

Proximity search is an advanced search technique which helps find documents where the query terms appear close together within a certain distance. This approach is useful for identifying documents that are more relevant to the query by considering the spatial relationship between the terms. Documents are ranked based on the proximity of the query terms within the text.

In the implemented search system, proximity search is performed by first extracting the proximity value and query terms from the input string, e.g. #1(smooth, criminal). Then, it retrieves the term positions of both query terms in each document. It iterates through these positions to check whether the absolute difference between the positions is within the specified proximity range. If it is, the document ID is added to the query result. The search system can also be configured to ensure that the proximity search returns only documents with the terms appear in the specified order.

4.5 Phrase Search

Phrase search is a specific type of search technique that allows users to find documents containing an exact sequence of query terms, enclosed within quotes, e.g. "Michael Jackson". The search engine considers the terms as a single unit and returns documents that contain the exact phrase in the specified order. This method is useful for locating documents containing specific expressions, idiomatic phrases, or quotes.

The implemented search system handles phrase search by constructing a series of proximity searches with a distance of 1 for each pair of consecutive query terms. It then intersects the results of these proximity searches to find documents containing the entire phrase. This approach ensures that the terms appear in the specified order within the documents by setting the appropriate configuration in the search system.

4.6 Boolean Search

Boolean search is a search technique that allows users to combine query terms using Boolean operators such as AND, OR, and NOT. This approach enables users to create more complex and precise search queries, providing greater control over the search results.

The implemented search system processes Boolean search by first checking the presence of each Boolean operator in the query. Based on the operator detected, it splits the query into

[9] https://kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm-25/

sub-queries and processes each sub-query recursively. The results of the sub-queries are then combined using set intersection (AND), set union (OR), or set difference (NOT) operations to produce the final query result. This approach allows the search system to handle complex Boolean queries and provide relevant search results efficiently.

4.7 Combined Queries

The implemented search system allows users to combine multiple search techniques in a single query, providing greater control over the search results. Users can use Boolean operators to combine the results of different search techniques and refine the query based on specific criteria.

For example, users can use proximity search to find documents where the query terms appear close together within a certain distance and then use Boolean search to exclude certain documents containing a specific phrase. To perform this type of search, users can use a query such as #1(smooth, criminal) AND NOT "Michael Jackson". This query combines proximity search with phrase search and Boolean search, providing more precise and relevant search results.

In the implemented search system, combined queries are processed recursively using the parse method. The search system extracts the different components of the query and applies the appropriate search technique to each component. Then, it combines the results using the specified Boolean operator to produce the final query result. This approach allows the search system to handle complex queries efficiently and provide relevant search results to the users.

5 Performance

Ranked search time performance (in seconds)			
Query type	TF-IDF	TF-IDF + CS	OKAPI BM25
short query	0.031	0.324	0.096
medium query	1.064	7.385	3.665
long query	2.588	15.764	6.036
query with popular terms	4.040	26.710	10.865
Advanced search time performance (in seconds)			
Query type	Proximity search	Phrase search	Boolean search
query with non-popular terms	0.240	0.282	0.254
query with popular terms	18.515	3.381	33.283

Table 1: Comparison of Ranked and Advanced Search Average Time Performance for Different Query Types Using Various Retrieval Models

In this section, we test the performance of the search engine. Performance testing was conducted on different query types using various IR models to compare their search time. The experiment was conducted 10 times for each query, and in Table 1 you can find the average time. For short queries, TF-IDF had the fastest search speed, while for longer and more complex queries, TF-IDF + CS had the slowest. In terms of advanced search, phrase search had the fastest search speed for queries with non-popular terms, while boolean search had the slowest search speed for queries with popular terms. It has to be mentioned that we show the average retrieval times calculated in the back-end, but since we retrieve all results for not ranked searches, front-end needs some more time to compile the results, that's why the user might notice the waiting time is more than the mentioned.

In terms of the quality of results for ranked search, TF-IDF AND BM25 are very precise, and the top results contained the required songs along with their remixes. On the other hand, TF-IDF, with cosine similarity, performs poorly for short queries, but for long queries, it is very precise and returns the required results. Below you can find the exact queries used.

Queries Tested:

- Short Query: ”slim shady” (2 terms)
- Medium Query: ”May I have your attention, please? Will the real Slim Shady please stand up?” (14 terms)
- Long Query: ”May I have your attention, please? May I have your attention, please? Will the real Slim Shady please stand up? I repeat, will the real Slim Shady please stand up? We’re gonna have a problem here.” (37 terms)
- Ranked search query with popular terms: ”love feel girl life die”.(Similar amount of non stopword terms as medium query)
- Queries with non-popular terms:#10(slim, shady), ”slim shady”, slim AND shady
- Query with popular terms: love girl (i.e. #10(love, girl), ”love girl”, love AND girl).

6 User Interface

After conducting informal user research on music search engines, including existing ones like Spotify, Genius, Apple Music, and Tidal, we found that our potential users have diverse search needs. Users search for music by song title or lyrics, sometimes including the author’s name; many use mobile devices for on-the-go searches. The accuracy of search results is a top priority, and users prefer functionality to aesthetics in UI design.

We aimed to cater for all these requirements. Our design emphasizes simplicity, modernity, and functionality, utilizing bright colours like blue, red, and orange. The layout is clear, with large fonts and straightforward icons for improved readability. It is responsive to the screen’s size.

We built the UI using ReactJS, with an intermediary Flask back-end connecting the search API and front-end. The search box is at the centre of the homepage, featuring live spell-checking as words are being typed in, the options to select the search algorithm, the number of results to be displayed (only for ranked search), and choosing between searching by song title or lyrics. A 300-character query limit is in place for performance reasons.

The search results page displays a list of results, each containing the song title, its author’s name and a snippet of its lyrics, and allows filtering the results by genre. Clicking on a result leads to a detailed page about the song, which contains information about the author, genre, and complete lyrics.

7 Conclusion

We had fun doing the project. It posed multiple system design and implementation challenges. We were a bit overwhelmed at the beginning with its sheer scale, but as we worked through it, everything became clearer and, in the end, we were able to connect all the parts together. One significant challenge that we are still in awe of is optimizing the system, especially since we were supposed to implement our data index and search algorithms from scratch. Albeit we did some work in that area, certainly we are now able to better appreciate the speed at which big search engines like Google operate.

Overall, the project challenged our understanding of the material taught in the course as well as our software engineering skills, and we definitely have a more in-depth understanding of what building a search engine entails. Further improvements could be made in optimizing the back-end processes, as well as in improving some UI dynamics. For example, caching the results of the search so that when the user exits a song’s page and back to the results page, they do not have to wait for all the results to load again.

References

[1] Christian Buck and Philipp Koehn. Quick and reliable document alignment via tf/idf-weighted cosine distance. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 672–678, 2016.

[2] Anjali Chauhan. A review on various aspects of mongodb databases. *International Journal of Engineering Research & Technology (IJERT)*, 8(05):90–92, 2019.

[3] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, Dimitrios Zissis, and Dimosthenis Anagnostopoulos. Mongodb vs postgresql: A comparative study on performance aspects. *GeoInformatica*, 25:243–268, 2021.

8 Individual Contributions

Alexandru-Ioan Chelba

I contributed to the report by writing the sections 1, 7 and also detailing some UI features in sections 2 and 6. I worked on the ReactJS framework and its Flask back-end. I created the website design together with Zhenwu, using ReactJS libraries such as MaterialUI and Bootstrap. In the Flask back-end, I implemented the live query suggestion feature, live results filtering by genre, and generally bridged the gap between the front-end and the main search REST API of our system. It was fascinating to learn how much power the ReactJS framework has, and how easy it makes it for the developer to implement relatively advanced features in few lines of code. Flask was also a novelty for me, although less so since I am more familiar with Python than with JavaScript.

Alessandro Speggiorin

My role in the group was primarily related to the back-end implementation, infrastructure setup and deployment of the application components. More precisely, I was responsible for the development of the crawlers and scrapers in order to be able to extract content from multiple sources, including APIs and websites. Furthermore, I implemented the live-indexing functionality and its automatic scheduling, including data deduplication and the Flask API required in order to allow the live-indexing module to interact with MongoDB. It is worth noting that the development of this section involved using data formatting and indexing scripts developed by Kyriakos and Gregoris. Moreover, I was responsible for deploying MongoDB and developing the required classes and infrastructure to allow efficient read and write access to song lyrics and inverted index tokens and postings (to be used in our Search module). In this context, I also implemented the self-rebalancing data structure to allow the storage of increasingly large postings and documents in MongoDB. Besides, my role in the group also involved writing scripts for automatic VM environment setup (in collaboration with Kyriakos) as well as automatic deployment of the core components to make the web app accessible. Lastly, I was responsible for writing Sections in the report related to my work including Sections 3.2.3, 3.3 and parts of Section 2.

Gregoris Georgiou

I have worked mostly on the back-end with Alessandro and Kyriakos. I researched and write the code for pre-processing the data collections and the queries accordingly with the type of search (title, lyrics). I also created the inverted indexes for them. I worked with Kyriakos in the ML part to classify missing genre of the songs which are live indexes. I helped in the filtering of the dataset from misc songs and also on how to filter only songs from the most famous 3000 artists. I implemented also TFIDF with cosine similarity and the OKAPI BM25 algorithm used for ranked search. On the report I wrote the parts corresponding to my contributions. Overall, I offered input and ideas to the team and wherever I could be a helpful asset, like testing and debugging, I offered my help.

Kyriakos Kyriakou

I have worked on the back-end of the search engine. More specifically, worked in the data collection (Kaggle dataset) and transformation (to JSON format) of the data (along with Alessandro). Furthermore, I applied applied a more memory efficient way to preprocess and index the data (without loading everything on memory) and worked on the creation of indexing. Moreover, I created the filtering of the dataset from kaggle to remove misc genre category from the data.

Also, I worked in developing the machine learning genre classification model with Gregoris. I wrote the script to populate the databases in MongoDB as well. I used GCP VM machines on daily basis for running data transformation, indexing, preprocessing, and machine learning (collaborating with Allesandro). Additionally, took part in the decision process for the architecture design. Finally, for the report, I wrote ML Section 3.4, Dataset section 3.1, and parts of 5 Performance Section. Overall, it was a fun and challenging project that gave me a great knowledge of how a fully functional search engines are working and deployed.

Michal Sadowski

As part of the team, my contributions to the music search engine project included several crucial tasks. Firstly, I implemented the TF-IDF search algorithm, followed by its integration into the system. I also implemented the Phrase, Proximity, and Boolean search algorithms, making parsing of combined queries possible, which added valuable functionality to the system. To enable access to these features, I encapsulated them into a REST API, which could be accessed by the front-end team. I also provided assistance to my colleagues and contributed to the bug-fixing tasks.

Furthermore, I played a significant role in the deployment of the system by configuring the domain name and DNS setup, ensuring that the system was easily accessible by users. In addition to these, I set up essential admin tasks like the repository and initial planning on the kanban board. Furthermore, I created the architecture diagram that proved useful in helping the team understand the system's components and their interactions. The process of deciding how the architecture would look like was a group effort and everyone contributed to the final design.

Overall, it was a great experience working on this project, and I am proud of the contributions I made to the team.

Zhenwu Wang

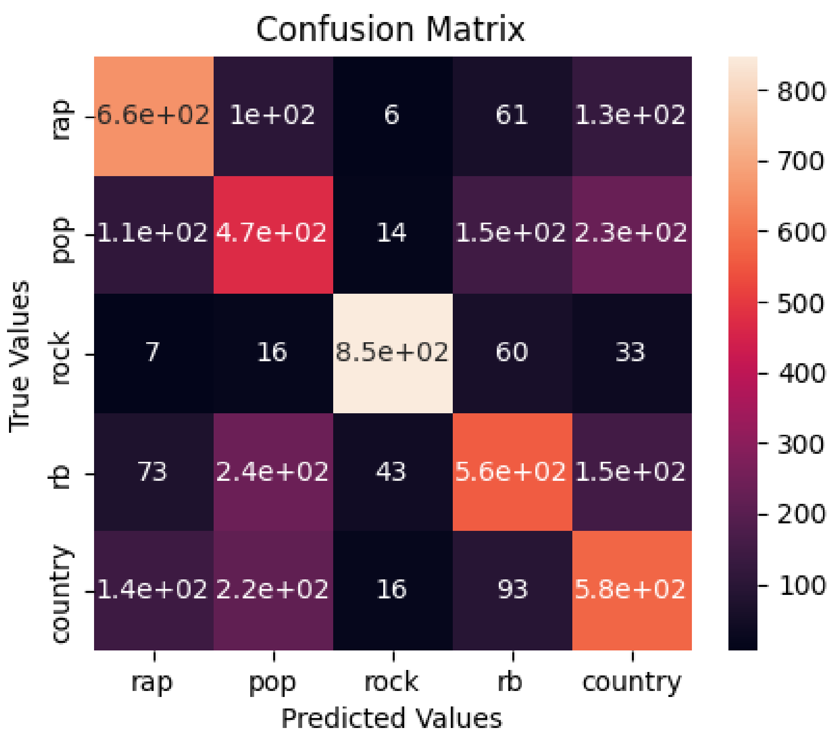
I worked with Alexandru-Ioan Chelba on the front-end of this music search engine project. My main contributions included understanding user requirements and designing and implementing the website based on them, which involved style, layout, search result display, and responsive design. I also designed the logo for the music search engine, as seen on the report's cover page. Additionally, I joined the back-end team led by Alessandro Speggiorin, Gregoris Georgiou, Kyriakos Kyriakou, and contributed to implementing some features. In this report, I detailed my specific contributions to the project (6). Overall, I provided valuable input and ideas to the team and was always willing to offer help in any way I could.

APPENDIX

A Machine Learning for Genre Classification

Genre	Precision	Recall	F1-score	Support
country	0.663	0.690	0.677	953
pop	0.453	0.482	0.467	978
rap	0.915	0.880	0.897	963
r&b	0.606	0.526	0.563	1065
rock	0.514	0.552	0.533	1041
Accuracy	0.622	0.622	0.622	0.622
Macro avg	0.630	0.626	0.627	5000
Weighted avg	0.627	0.622	0.624	5000

(a) Classification Report of ML genre classification



(b) Confusion Matrix of genre classification model

Table 2: Genre Classification Results