VerseVault: a lyrics-based Search Engine

ANDRÉ LIMA, GUILHERME ALMEIDA, JORGE SOUSA, and JOSÉ CASTRO

The dynamic nature of the music industry continually shapes and transforms song structures and lyrical content. With the widespread popularity of platforms like Genius [7], lyrics are now more extensively documented than ever before. However, despite the abundance of data, there is a limited exploration of its potential. This report introduces *VerseVault*, a search system utilizing Solr, designed to empower users to explore songs through text-based queries, unraveling identical textual sections in diverse contexts.

The report provides a comprehensive review of the VerseVault project, covering key aspects such as its original concept and inception, the intricacies of the data collection pipeline, primary search scenarios targeted for enhancement, and the overall implementation process. By examining each phase of the project, this report sheds light on the evolution of VerseVault and its contribution to the exploration of song lyrics through a text-based search system, validating a semantic search approach after thoroughly evaluating different querying methods and functionalities and complemented by a user-friendly in-browser GUI.

1 INTRODUCTION

With the foundation of Genius [7] in 2009, lyrical documentation of the widespread musical scene slowly became centralized as millions of users contributed to a thorough and substantiated textual database of innumerable pieces of audio media. Through a *community-driven* approach, its website offers testified metadata about musical structure along with its content to any person who is willing to access it. Nevertheless, the rich intricate knowledge this mass of information can provide is rather inaccessible, as there appears to be no interconnected search engine that takes the structural aspect of a verse into account when dealing with its lyrical subject matter. In this way, *VerseVault* aims to become a platform that allows users to observe a song's composition as well as its verbal theming - and display how these have changed over time in music production. This document is organized into the following sections:

- Introduction
- Data Collection & Processing
- Data Treatment
- Data Characterization
- Information Needs
- Information Indexing
- Information Retrieval
- Search System Improvements
- Evaluation
- GUI - Web App
- Conclusions
- Future Work
- References
- Appendix

Authors' address: André Lima, up202008169@edu.fe.up.pt; Guilherme Almeida, up202006137@edu.fe.up.pt; Jorge Sousa, up202006140@edu.fe.up.pt; José Castro, up202006963@edu.fe.up.pt.

2 DATA COLLECTION & PROCESSING

In this section, we will go over the data sources used for data extraction and collection and the subsequent processing operations we performed on the attained information.

2.1 Data Sources

The first data source *VerseVault* uses to collect information is Last.fm [10], a popular music website that builds a detailed profile of each user's musical taste by recording details of the tracks the user listens to. Aside from monitoring personal activity, the site also retains several global, all-encompassing charts that register the most listened to tracks, artists and albums by every user on the site, while documenting each of these with entities called *tags*. These *tags* are, generally, one-word textual bodies that represent a semantic detail of a given item, like a track's genre or an artist's gender. The data we collect from this source revolves exclusively around tracks, artists, albums and their respective tags.

The second data source is the key to obtaining the extensive and rich textual repertoire that is needed for this project: lyrist [11]. lyrist [11] is a RESTful lyrics API that allows users to search for song lyrics based on a track's name and author and returns the content found through requesting these to the Genius [7] website, which features an immense and deeply scrutinized lyrical database. With this, we extract accurate and well-structured lyrics to the tracks we acquired beforehand.

2.2 Data Pipeline

The technology used in this section is mostly comprised of Python scripts. The pipeline used for data extraction is as follows:

(1) First, we collect the top-charting artists and tracks using the Last.fm [10] API, scraping their metadata (song release date, f.e.) to local *csv* files. The *genre* attribute is only available through the processing of *Last.fm*'s [10] *tag* usage - keywords that can refer to many semantic details pertaining to the item in question -, which we also save for later usage through filtering.
(2) Afterwards, for each collected song, we obtain the corresponding lyrics by locally running lyrist [11] queries and storing this textual content to another file, just like before.
(3) Subsequently, we use the previously retrieved *tags* and filter through them to achieve a compilation of valid *genre* tags to attach to *track* entities, as most tags are unneeded for the scope of this project and do not represent music genres, as well as essentially lacking any type of relevance if used as search filters.
(4) Before leaving data processing behind, we make use of the spaCy [5] tool to perform Named Entity Recognition on the sampled textual information as the final step in our pipeline, saving the found entities to the database.
(5) Finally, with a complete backlog of information, we have decided to use a cloud-based MongoDB [12] instance to store our collection by uploading our local data to the server.

We preferred this approach instead of a local instance as it is easier to set up and work collaboratively without using things such as *git-lfs* [8].

The diagram for the complete data extraction pipeline is shown in Figure 4.

2.3 Domain Model

The domain model represents the core entities and their relationships within our Search Engine's initial vision. Its visual representation is present in Figure 5.

In the following subsections, we make a brief description of our entity's attributes.

2.3.1 *Track*

The Track entity represents an individual audio track within an album. It includes the following attributes:

- *name:* Signifies the name of the track.
- *releaseDate:* Represents the date on which the track was published.

2.3.2 *LyricSection*

The *LyricSection* entity defines different segments within a track's lyrics (e.g.: intro, chorus, etc.), categorized by type. It is characterized by the following attributes:

- *type:* Represents the type of section within a track.
- *content:* Represents the textual content of the section.

2.3.3 *Album*

An Album is an entity that encapsulates information about a collection of audio tracks. It holds the following attributes:

- *name:* Represents the name of the album.
- *releaseDate:* Denotes the date when the album was officially released.

2.3.4 *Artist*

The Artist entity signifies a music artist involved in the creation of albums and tracks. It includes the attributes:

- *name:* Represents the name of the artist.
- *genre:* Contains the name of the artist's genre.
- *birthCountry:* Contains a string representing the artist's birth country.

3 DATA TREATMENT

Because of how our pipeline is set up, there is rarely an instance in which we have to interact with the collection directly after everything is fetched. This occurs, however, when lyrics for a track are not found, or an error occurs while fetching them. To resolve this, we simply discard this singular *Track* entity from our database.

4 DATA CHARACTERIZATION

Our collection consists of 46,938 documents, obtained using the aforementioned pipeline. These documents occupy a total storage space of 144.97MiB.

Each document represents a single track and it is comprised of the following fields:

- name: the track's name
- duration [optional]: the track's duration in milliseconds
- url: the track's URL on Last.fm
- artist: the name of the artist who published the track
- publishedAt [optional]: the timestamp at which the track's wiki was published on Last.fm
- lyrics: an array of sections that comprise the track's lyrics
  - title [optional]: the section's title
  - content: the section's content
- genres: an array of genres that the track belongs to
- album [optional]: the album that the track belongs to
  - name: the album's name
  - image [optional]: the album's cover image
- entities: an array of NER (Named Entity Recognition) entries
  - text: the content of the entry
  - start: the position of the start of the entry
  - end: the position of the end of the entry
  - type: the entity type (time, person, ...)

An example of a document from our collection is provided in Figure 6.

Since our collection's documents have optional properties, we analyzed the number of documents in which those properties are missing and obtained the results provided in Figure 1.

Missing data	Frequency
Missing publishing date of wiki	68.27%
Missing duration	20.66%
Missing sections with title	12.79%
Missing album	8.80%
Missing album's image	3.04%

Fig. 1. Percentage of documents where data is missing.

These results will influence our decisions on the next milestones as we decide which fields to give more importance to for a given search query. Furthermore, these will also limit our users' abilities to satisfy their information needs since not all tracks have these important pieces of data.

4.1 Distribution of tracks across genres

Each track can be associated with many genres. The number of documents for the top 20 most popular genres in our collection can be seen in Figure 7.

These results are not expected since pop is one of the most popular music genres. However, it is not present in Figure 7.

After investigating this, we discovered that our pipeline removed the tag "pop" from our collection, since its description on Last.fm [10] does not contain the word "genre". This is something we expect to fix in the upcoming milestones.

4.2 Distribution of tracks over time, according to the wiki publishing date

The distribution of the number of tracks over time, according to the wiki publishing date, can be seen in Figure 8. We can see that our collection is comprised mainly of documents from 2008 until 2010 and from 2021 until 2023. The distribution is, therefore, bimodal.

These results are unexpected as well. Since our pipeline processes the most popular songs and albums on Last.fm [10] first, we would expect the number of tracks over time to go down, instead of there being a spike around 2009.

4.3 Text Analysis in Lyrics

The lyrics of a track will be our main indexable component. As such, we will go more in-depth into its analysis.

Regarding the word count in each track's lyrics, some statistics were computed and are represented in Figure 2.

	Word Count
Mean	321.52
Std. Dev.	189.15
Min	1
25%	192
50%	287
75%	409
Max	5425

Fig. 2. Statistics regarding the word count for each track

These results show that our collection is diverse, in terms of the length of the lyrics.

We also decided to do a word cloud diagram of the lyrics of the tracks in our collection (Figure 9 to understand what kind of words are more common in our collection.

Even though the depth of the analysis provided by a word cloud is limited, we can see that verbs are very common in our collection. Furthermore, some words are very similar, such as "oh", "ooh" and "ah".

5 INFORMATION NEEDS

The information needs going forward are centered around key descriptive bodies that we ascertained as important in our data collection and characterization process. To specify these, we created several relevant search scenarios. These include:

5.1 Lyrical Content in Section

- *Scenario:* I want to find a track talking about a specific subject inside a particular structural section.
- *Query Example:* Find songs that talk about playing guitar in the chorus.

5.2 Lyrical Content

- *Scenario:* I want to find a song talking about a given topic.
- *Query Example:* Find songs that talk about slavery in Africa.

5.3 Lyrical Content in Genre

- *Scenario:* I want to find a song of a given genre that talks about a specific topic.
- *Query Example:* Find rap songs that talk about immigration.

These scenarios cover a range of user intentions, from exploring specific lyrical content to discovering songs based on genres. As *VerseVault* evolves, refining the search capabilities for these scenarios will enhance the user experience and make the platform more versatile.

6 INFORMATION INDEXING

With a curated dataset on our hands, the next step in our project is to start retrieving information.

First, we need to choose a text-based search engine that fits our application's needs. Although the decision was initially split between Apache Solr's [3] and Elasticsearch's [4] capabilities, Solr ended up being the chosen platform, as it allowed us to have a more low-level view of how our dataset is indexed. Built upon Apache Lucene[2], Solr provides thorough full-text search capabilities and can handle large amounts of data through an easy-to-use API.

6.1 Document Definition

The collection obtained in Milestone 1 will be indexed into a core named *tracks*. Each document in this core can be one of two types, specified by the *doc_type* property: "track" or "lyric_section". A *track* contains most of the information about a track, such as its name, duration, genres, album, and *NER* (Named Entity Recognition) information, among others. A *lyric_section* contains a title, if present, and its content. The documents with type *lyric_section* are indexed as nested child documents of a document with type *track*. This was done to preserve the relationship between a section and its content, allowing us to search for specific information inside a given section of a track.

6.2 Indexing Process

As the first step in our indexing process, an initial schema was created to allow the indexing of our collection into Solr [3]. In this schema, only the field types present in the default Solr[3] schema were used, such as *string*, *text_general*, *plong*, and *pdate*. After that, we progressively refined this schema to better satisfy the information needs described previously, as well as improve the system's robustness against non-exact match querying. This refined schema will be described in Sections 6.2.1 through 6.2.3.

Next, we performed some necessary transformations on our data, such as adding a unique *id* field to all documents, including the lyrical sections. Furthermore, in order to not increase significantly the amount of Apache Lucene [2] documents created by Solr [3], we decided to not index the *entities* field as nested documents and, therefore, we had to flatten that object into four different fields: *entities.text*, *entities.start*, *entities.end*, and *entities.type*. The same process was applied for the *album* object. Finally, we converted all timestamps in our dataset to UTC format, since it's the only format recognized by Solr [3].

As part of Milestone 3, we created a new field in the *track* documents, called *content_vector*, which is used to store the embeddings of a track's lyrics. These embeddings will then be used by the semantic search schema that will be described later in this report.

Finally, the documents are uploaded and indexed by Solr [3] via its JSON API.

6.2.1 Indexed Fields

The fields to be indexed are chosen based on their necessity and usefulness in fulfilling the information needs. The fields left out of

the indexing process do not hold any type of value regarding the type of queries that will be made. Still, some are interesting to have stored for further usage in Milestone 3, such as the *url*, *entities.start*, *entities.end*, and *album.image*. These fields will not be covered below and can be later integrated into the UI.

In terms of song metadata, the fields *name*, *duration*, *artist*, *publishedAt*, *genres* and *album.name* are indexed, as they represent important information used in the querying process. The lyric section's *content* and *title* are also indexed as they represent the main textual data to be searched. About the *entities* array, only *entities.text* will be indexed, as it is the only one relevant to the search functionality. Furthermore, since the main goal of our project is to help people find a song, we decided to implement phonetic matching too, thus needing another field to index the tokens created by the phonetic filter: **content_phonetic**. This field is not stored, since its retrieval for purposes other than the search process is not relevant.

6.2.2 Field Types & Indexed Analyzers

Analyzers are an extremely important part of the indexing process - as previously mentioned, they strengthen our system to make it adaptable to imperfect queries. To correctly build a strong index, we needed to create new field types used to parse our collection, using the help of several analyzers provided by Solr [3]. Below is an explanation of each new type's composition:

- **basic_text_t**: a similar, but simpler type to *text_general*. Doesn't have stop word removal or synonym matching, but has ASCII folding, which helps with the multilingual nature of our dataset. This field type is useful for smaller pieces of text that don't need a very extensive analysis. To this end, we employ:
  - the *standard tokenizer*;
  - the *lowercase filter* to turn every token lowercase;
  - the *ASCII Folding filter* to convert UTF-8 characters to their ASCII equivalents;
- **artist_t**: a similar type to the *basic_text_t* but uses the *letter tokenizer*, easing the matching of examples in which there are clear delimiters. Only used in the *artist* field.
  - the *letter tokenizer*, which creates tokens from strings of contiguous letters, discarding all non-letter characters;
  - the *lowercase filter*;
  - the *ASCII Folding filter*;
- **lyrics_content_phonetic_t**: a type that backs the added phonetic fields. This highly specific type is needed to use *Beider-Morse Phonetic Matching filter*. A phonetic filter was used to improve our system's tolerance to similar words when said out loud, but not when written, which can help when song lyrics are misheard.
- **lyrics_content_en_t**: a type that more extensively analyzes the text it is provided. As the name suggests, it is used for indexing the contents of lyrics, assuming they were written in English.
  - the *standard tokenizer*, which, according to Solr's documentation, works fairly well for a wide variety of use cases;
  - the *lowercase filter*;
  - the *ASCII Folding filter*;
  - the *Keyword Repeat filter*, to keep the original tokens and a stemmed copy of them;
  - the *Porter Stem factory* to stem the non-keyword tokens;
  - the *Remove Duplicates Token filter*, in order to remove tokens that were not stemmed;

The only fields using *text_general* are the *title* of the lyrical sections and the *name* of the tracks, since these fields are not used in any of our queries and, as such, don't require an extensive indexing process.

Numerical data is handled with the *plong* type, and temporal data (*publishedAt* field) is indexed with the *pdate* type. *string* usage is kept to fields that only need exact matching.

6.2.3 Schemas

The initial schema we used is equivalent to the refined schema, but uses no custom field types and has every non-string textual field as being of the *text_general* type. Our refined schema is shown below in Figure 3.

Field Name	Type	Indexed
_id	string	false
name	text_general	true
duration	plong	true
url	string	false
artist	artist_t	true
publishedAt	pdate	true
genres	basic_text_t	true
lyrics.title	text_general	true
lyrics.content	text_general	true
lyrics.content_en	lyrics_content_en_t	true
lyrics.content_phonetic	lyrics_content_phonetic_t	true
album.name	text_general	true
album.image	string	false
entities.text	basic_text_t	true
entities.start	plong	false
entities.end	plong	false
entities.type	string	false

Fig. 3. Refined Schema Definition

As for what was developed during Milestone 3, we've created two new schemas: the semantic schema and the synonym schema. The first one is a schema specifically made for semantic search, which indexes the embeddings of a given track as a Dense Vector Field. This type allows us to compare, later on, the embeddings of a piece of user input with the embeddings of the lyrics in our collection. The synonym schema, on the other hand, indexes the content of a lyric section with a synonym filter, which means that each token is transformed into itself and its synonyms in the indexing process. Both of these approaches are meant to make our system more precise and tolerant of inexact matches. These will be explained more thoroughly later on.

7 INFORMATION RETRIEVAL

In this section, we set out to create and present two search systems, one for each schema. These search systems were developed during Milestone 2 and will then be evaluated on each of the information needs previously selected in Section 5 (these were also refactored in the second and third milestones to fit into the interests of our project at this time). This evaluation, along with results and conclusions, can be found later below in Section 9. Aside from a schema, each search system will also have an associated query format. These are specified through Apache Lucene's query syntax, using the *q* query field and the *q.op* query operator. Assuming 'query' is the user input in the search functionality, the structure for the two query systems developed follows:

- For the initial system, we will simply search in the lyrical content field of each track for the provided user input.
  - **q:** `{!parent which="doc_type:track" score=max} {!edismax qf="content"}query`
  - **q.op:** OR
- For the refined system, we will take advantage of the eDisMax query parser functionalities, such as field boosting and slop. These weights and parameters were tested through trial-and-error.
  - **q:** `{!parent which="doc_type:track" score=total} {!edismax qf="title^0.5 content_en^4 content_phonetic^2" pf="content^10"}(query)~3`
  - **q.op:** OR

7.1 Information Need 1

Scenario: I want to find a track talking about a specific subject inside a particular structural section.

Information Need: Find songs that talk about playing guitar in the chorus.

User Input: play guitar

7.2 Information Need 2

Scenario: I want to find a song talking about a given topic.

Information Need: Find songs that talk about slavery in Africa.

User Input: slavery africa

7.3 Information Need 3

Scenario: I want to find a song of a given genre that talks about a specific topic.

Information Need: Find rap songs that talk about immigration.

User Input: foreigners immigration

8 SEARCH SYSTEM IMPROVEMENTS

This section describes the different concepts contemplated for the improvement of our system after the creation of the refined schema, coupled with an explanation of how they were implemented, their results, and conclusions drawn.

8.1 Semantic Search vs Synonyms

The textual content present in our project is highly non-technical, disorganized, unstructured, diverse, and, for the most part, ambiguous - meaning that retrieving documents for a query that will usually consist of a few specific words will always end up failing to account for contextual relevancy when, for example, those exact words are not present in a song. A good, real-life example of this inconsistency occurs when lyrics are written metaphorically: the feeling of loneliness may be described as "drowning"; "high" is commonly used as both an allegory for happiness but is also very common slang for drug usage; and so on. Two possible solutions to help our search system circumvent relevancy becoming lost are:

- **semantic search:** the process of attributing meaning to a search. Whereas lexical search finds exact matches, semantic search aims to improve accuracy by taking into account the context of the textual content present in both the database and the query.
- **synonym usage:** the process of querying using an auxiliary word pool that's semantically equivalent to a query's lexical terms to retrieve more relevant documents and therefore increase the performance of a search system.

To thoroughly analyze the effect these two methods have on our search system, we adapted the already existing refined schema into two new cores that each employed one of the solutions mentioned.

- The **semantic schema** employs a technique called embeddings. These embeddings are created with the Sentence Transformers[16] Python library paired with the *all-MiniLM-L6-v2* [13] model to transform each document's lyrical content (*lyrics.content* field) into a 384-dimensional dense vector space, which will be stored in a new *content_vector* field, whose type is Solr's own *DenseVectorField*. At query time, this mapping process will also be applied to the query, in which a K-Nearest Neighbours algorithm will try to match documents with the target query vector. In terms of dense vector search[1], K-NN is the best-documented algorithm Solr has available, hence its usage. As for why we chose the *all-MiniLM-L6-v2* model, in our testing, it bested other general-purpose models like the more popular *all-MPNet-Base-v2*[14] and the multi-language *paraphrase-multilingual-MPNet-Base-v2*[15] in terms of the relevancy of the documents retrieved.
- The **synonym schema** employs the standard Solr *SynonymGraph* filter backed by a large synonym list (45270 synonyms), compiled from various sources by Hanson Robotics[9]. Its usage is purely restricted to query time.

After creating the new cores, we also developed a query format for each method, thus constituting two new search systems. Finally, we measured their performance against each other, as well as the *initial* and *refined* systems, described in the previous sections of this report. The query formats are as follows:

- For the semantic query, we have to use Solr's K-NN capabilities through the K-NN Query Parser. So, assuming 'query' is the user-input search and 'query_vector' is the dense vector resulting from the embedding process, its query structure will be as follows:
  - **q:** `{!knn f=content_vector topK=20} {embedding}`
  - **q.op:** OR
- For the synonym query, we will simply reuse the query structure we have been using until now, using Lucene's

Parser with eDisMax functionalities. Then, assuming 'query' is the user-input search, its structure will be as follows:
- **q:** `{!parent which="doc_type:track" score=total} {!edismax qf="title^5 content^3" pf="content^10"}(query)`
- **q.op:** OR

Because of the nature of the language model, the semantic search system's queries were adapted as follows, in order to retrieve more relevant results: for information need 1, '*play guitar*' was converted to '*playing the guitar*'; for information need 2, '*slavery africa*' was converted to '*talks about slavery in africa*'; for information need 3, '*immigration foreigners*' was converted to '*literally immigration and talks about foreigners*'

Regarding the first information need, the results instantly revealed a big problem with synonym usage: most of the semantic meaning of certain sentences was lost because two or more words that usually complement each other would be replaced by synonyms. Thus, documents that have completely different meanings are retrieved, affecting the precision negatively. For example, the word "play" gets matched with many of its synonym counterparts, leading to results that score highly because of the word "game". Concerning the semantic approach, however, the result was quite the contrary. For instance, it achieved good results in the first ten documents but lost precision in the latter half because the underlying meaning of the query was lost; unrelated songs that talked about performing or playing other instruments, in general, were retrieved instead.

Regarding the second information need, it is clear that the search results for the synonyms approach diverge immensely from the original meaning of the query. As an example, since the word 'slavery' is, in our list of synonyms, a synonym of 'work', the results will be heavily skewed towards songs that feature that word. In fact, the only slavery-related results amounted to 3 entries out of the 20 analyzed results and, even then, they did not pertain to an African context. The semantic method managed to pinpoint songs talking about apartheid just as well as the other schemas, but its focus on attributing meanings led to documents that talked about the struggle against oppression in other contexts (p.e. Cherokee nation genocide in the band Europe's 'Cherokee').

When talking about the third information need, the synonyms reveal yet again lackluster performance, failing to collect even a single relevant document, owing to the recurring problem of matching with synonyms that do not have the same in-context meaning as the source query. Inversely, the semantic search performed quite well, picking up on the dense contextual framing of the topic of immigration, going as far as to return songs that do not mention "immigration" or "foreigners" in a literal sense but have these themes as their main subject matters, such as Steppenwolf's "Monster", which talks about the mass immigration wave when the United States of America was formed.

Overall, it was clear that the usage of synonyms was not appropriate in our project, as its use case fell flat due to the overwhelming unstructured semantic nature of music lyrics. A possible problem might also arise in the nature of the synonym pool used, as its immense amount of synonym choices might have negatively impacted our search engine's performance. Contrarily, the semantic search system turned out to be very helpful, even if it is not as good at exact-query matching. In the end, this outcome aligns with our goal of modeling a search engine that lets users search songs about topics while not having an exact example to query in mind, and we chose it as our selected search system to use in the finalized version of the project - the web app.

9 EVALUATION

To evaluate each of our search systems' performances, we conducted a manual evaluation of each of them.

The manual evaluation consists of uploading the schema to the Solr core, uploading our data to it, and running the corresponding queries listed in the previous section against the resulting index. For each query, we went through each search result and decided if a document was relevant or not based on the scenario provided (1 for relevant, 0 for not relevant). The sample size of the documents for each query has a maximum value of 20 and they are ranked 1 through N according to their retrieved order (e.g. rank 1 is the first retrieved document), meaning that higher-ranked documents should semantically be more relevant than lesser-ranked ones. After confirming each document's relevancy, important metrics are calculated: the **Precision@N** (P@N), which is the ratio between relevant items retrieved and total retrieved (N) items; the **Interpolated Precision-Recall Curve**, which is a curve that measures, for each level of recall, the corresponding level of precision of the system; and the **Average Precision**, which is the average precision value for a given search query. In the subsections below, we will analyze the results we gathered and describe the main takeaways we got from them.

9.1 Initial Search System

When using the initial search system to perform queries, we obtained the following number of results:

- Information Need 1 - 20 documents
- Information Need 2 - 20 documents
- Information Need 3 - 15 documents

This search system presented a Mean Average Precision of 45%. The precision-recall curve for this system can be seen in Figure 10 and the precision@ values can be seen in Figure 11. The corresponding individual assessments are present in Figure 23.

In Figure 10, it can be seen that the results for the first and the third information needs had close precision-recall values. Overall, this schema tends to fall off in precision at the early stages of recall. This is corroborated by 11, from which we can conclude that the documents ranked past the fifth position are not relevant, for the most part.

9.2 Refined Search System

When using the refined search system to perform queries, we obtained the following number of results:

- Information Need 1 - 20 documents
- Information Need 2 - 20 documents
- Information Need 3 - 20 documents

This search system presented a Mean Average Precision of 56%. The precision-recall curve for this system can be seen in Figure 12

and the precision@ values can be seen in Figure 13. The corresponding individual assessments are present in Figure 24.

9.3 Semantic Search System

When using the semantic search system to perform queries, we obtained the following number of results:

- Information Need 1 - 20 documents
- Information Need 2 - 20 documents
- Information Need 3 - 20 documents

This search system presented a Mean Average Precision of 73%. The precision-recall curve for this system can be seen in Figure 14 and the precision@ values can be seen in Figure 15. The corresponding individual assessments are present in Figure 25.

9.4 Synonym Search System

When using the semantic search system to perform queries, we obtained the following number of results:

- Information Need 1 - 20 documents
- Information Need 2 - 20 documents
- Information Need 3 - 20 documents

This search system presented a Mean Average Precision of 10.3%. The precision-recall curve for this system can be seen in Figure 16 and the precision@ values can be seen in Figure 17. The corresponding individual assessments are present in Figure 26.

9.5 Overall Evaluation After evaluating each search system, we calculated the average precision-recall curve for each one, as well as the average precision@ values. These metrics can be seen in Figure 18 and 19, respectively.

As we can see by those metrics, the synonym search system had, by far, the worst performance. As explained previously, this is due to the synonym filter using words with a completely different meaning in the same context when performing searches. For two of the three information needs the search system could not find a single relevant result, as its synonym matched songs with a completely different meaning. Overall, this system failed purely due to the erratic nature of song lyrics and the lack of context-sensitivity in the usage of synonyms.

The initial search system, basic in its inception, employed the most rudimentary Solr functionalities and displayed middling results. This system suffers from a lack of features used in its querying and falls flat when trying to retrieve a bigger number of relevant documents, but it still retains decent exact-match capabilities.

The refined search system supported a more robust set of filters and tokenizers, aided by field boosts and phrase slop, retrieved a bigger set of documents and boasted higher precision than both the initial and synonyms schema. Its biggest downfall is the lack of context awareness, something the semantic search system and synonym system aimed to overcome. This problem can be seen when trying to tackle the third information need, in which the refined system kept matching the token 'foreigner' with songs that used the word as a synonym for 'foreign car'.

The semantic system outperformed every other system in every metric, and was the key to effectively bring context awareness into our search system. While its leniency is noted when tackling highly specific search needs, its success in retrieving relevant documents cannot be understated. Of note is the fact that many never-before-seen relevant tracks were being collected for the first time, even after thorough testing of 3 different search systems. All in all, it became the top choice for which search system to use going forward.

10 GUI - WEB APP

A minimalist and intuitive GUI was developed using Next.js[17], providing the users with the ability to query our data collection using the semantic search system.

After submitting the input query, the request is handled by a Flask[6] API that queries Solr directly through its REST API. The Flask API is needed to generate the embeddings used in the Solr Dense Vector Search, done by a Python script that, as previously explained, uses the Sentence-Transformers library to generate a dense vector sourced from the input query. This query is exactly the same as described previously in Section 8.1.

You can find images for the different scenarios in figures 20, 21 and 22.

11 CONCLUSIONS

The initial data pipelining process retrieved a great document collection to further our work over the course of this project. Through repeated testing of different search systems, we were able to find one that could fit our information needs and overarching goals for VerseVault. In the end, we were also able to couple the search engine with a user-friendly and informative GUI.

12 FUTURE WORK

A potential way for future improvement could involve implementing signals to prioritize tracks with more relevant lyrical sections, allowing for a more nuanced understanding of the lyrical content and further enhancing the system's ability to meet user information needs. This strategic enhancement would align with the intricacies of song lyrics and contribute to an even more refined and context-aware search experience. Furthermore, experimenting with other eDisMax parameters and tuning them could prove fruitful, although the current results are already satisfactory. Additionally, a most approachable idea is the implementation of a MoreLikeThis feature, implemented with the help of the GUI. If proven to be useful, it would be a fantastic addition to our search system, and would help circumvent the contextual leniency brought forward by the semantic search, allowing users to find documents more catered to their needs.

REFERENCES

[1] Apache Software Foundation. About dense vector search in solr. https://solr.apache.org/guide/solr/latest/query-guide/dense-vector-search.html, 2023. [Online; accessed 7-December-2023].

[2] Apache Software Foundation. Apache lucene. https://lucene.apache.org/, 2023. [Online; accessed 15-November-2023].

[3] Apache Software Foundation. Apache solr. https://solr.apache.org/, 2023. [Online; accessed 15-November-2023].

[4] Elasticsearch B.V. Elasticsearch Website. https://www.elastic.com/, 2023. [Online; accessed 10-October-2023].

[5] Explosion. spaCy Website. https://spacy.io/, 2023. [Online; accessed 6-October-2023].

[6] Flask. Flask. https://flask.palletsprojects.com/en/3.0.x/, 2023. [Online; accessed 5-December-2023].

[7] Genius. Genius Website. https://genius.com/, 2023. [Online; accessed 1-October-2023].
[8] Github Co. About git large file storage. https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-git-large-file-storage, 2023. [Online; accessed 15-November-2023].
[9] Hanson Robotics. Hanson robotics synonyms file. https://github.com/hansonrobotics/hr-solr/blob/master/synonyms.txt, 2023. [Online; accessed 10-December-2023].
[10] Last.fm. Last.fm Website. https://www.last.fm, 2023. [Online; accessed 1-October-2023].
[11] Lyrist. Lyrist Website. https://lyrist.vercel.app/, 2023. [Online; accessed 1-October-2023].
[12] MongoDB, Inc. MongoDB Website. https://www.mongodb.com/, 2023. [Online; accessed 4-October-2023].
[13] Nils Reimers. About all-minilm-l6-v2. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, 2023. [Online; accessed 7-December-2023].
[14] Nils Reimers. About all-mpnet-base-v2. https://huggingface.co/sentence-transformers/all-mpnet-base-v2, 2023. [Online; accessed 5-December-2023].
[15] Nils Reimers. About paraphrase-multilingual-mpnet-base-v2. https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2, 2023. [Online; accessed 5-December-2023].
[16] Nils Reimers. Sentence transformers. https://www.sbert.net/, 2023. [Online; accessed 7-December-2023].
[17] Vercel. Next.js. https://nextjs.org/, 2023. [Online; accessed].

A APPENDIX

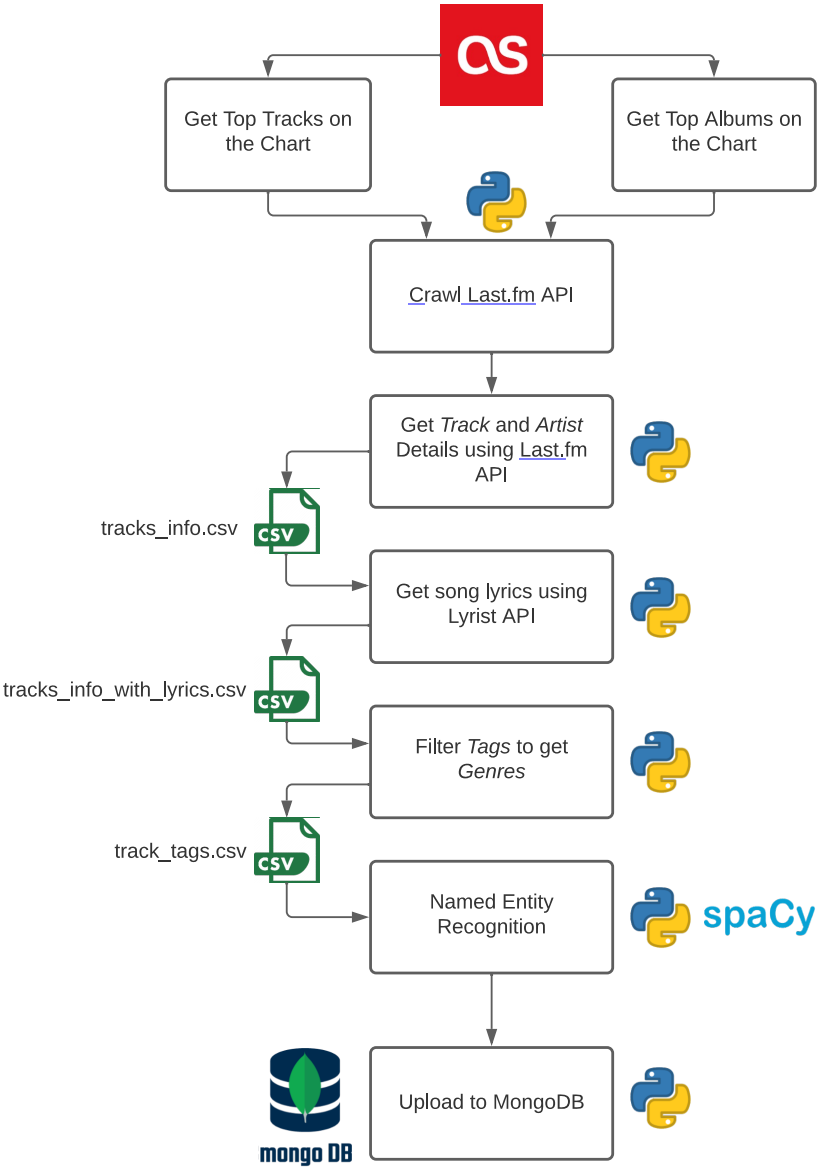

Fig. 4. Data collection and processing pipeline.

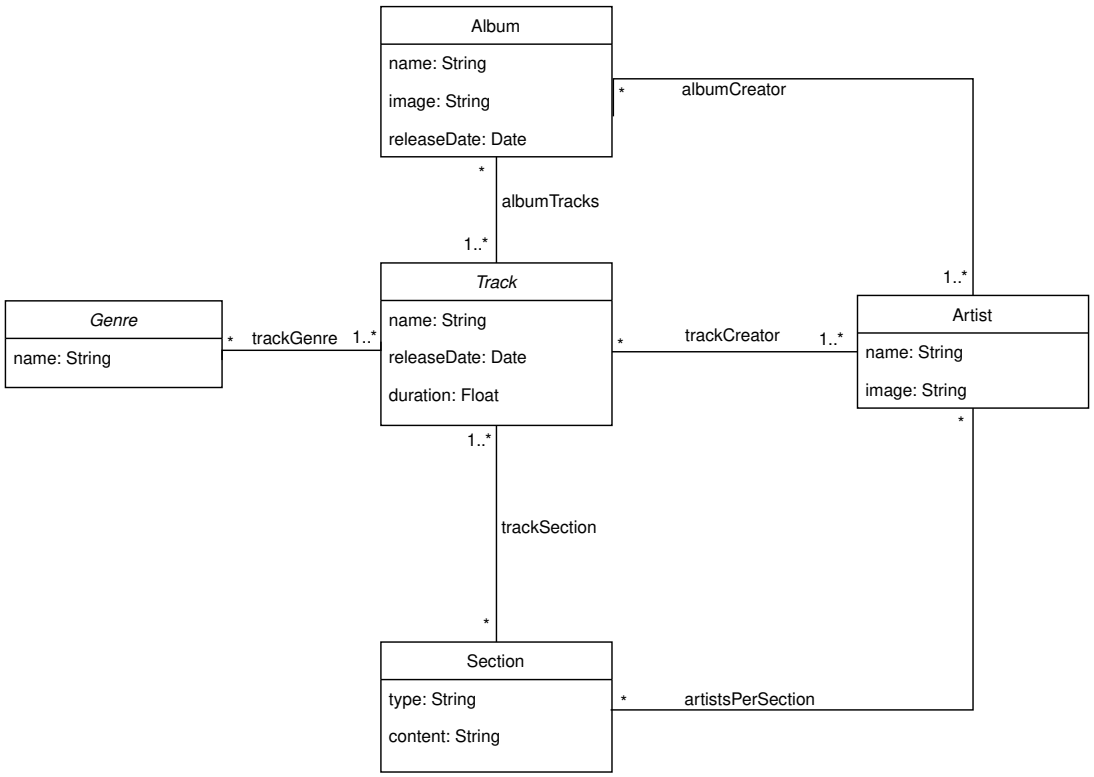

Fig. 5. VerseVault's Domain Model.

```
_id: ObjectId('65270c2c3d5fe092ffa700d2')
name: "Why'd You Only Call Me When You're High?"
duration: 162000
url: "https://www.last.fm/music/Arctic+Monkeys/_/Why%27d+You+Only+Call+Me+Wh…"
artist: "Arctic Monkeys"
publishedAt: "04 Sep 2021, 18:23"
▾ lyrics: Array
  ▾ 0: Object
      title: "Verse 1"
      content: "The mirror's image tells me it's home time
                But I'm not finished, 'caus…"
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
  ▸ 5: Object
▾ genres: Array
    0: "indie rock"
▾ album: Object
    name: "Why'd You Only Call Me When You're High?"
    image: "https://lastfm.freetls.fastly.net/i/u/300x300/f579e414e20f40969185e411…"
▾ entities: Array
  ▾ 0: Object
      text: "Carryin"
      start: 139
      end: 146
      type: "ORG"
```


Fig. 6. Example of a document representing a track.

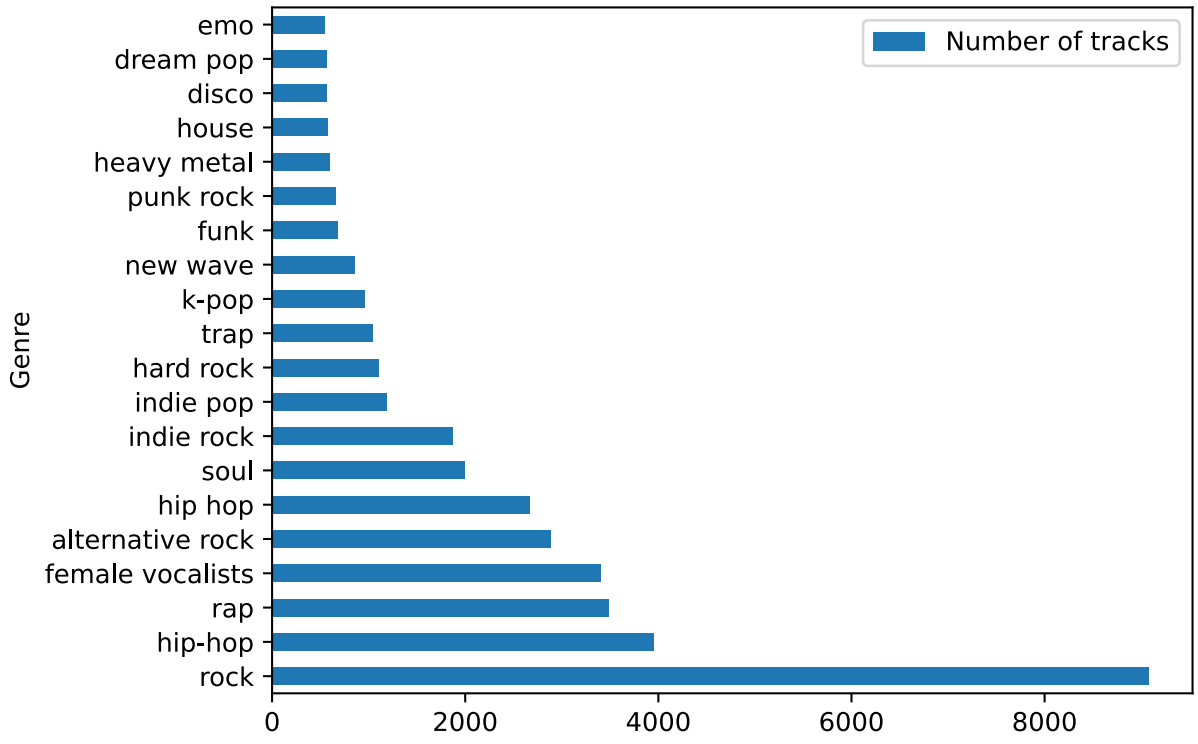

Fig. 7. Distribution of tracks across the top 20 genres.

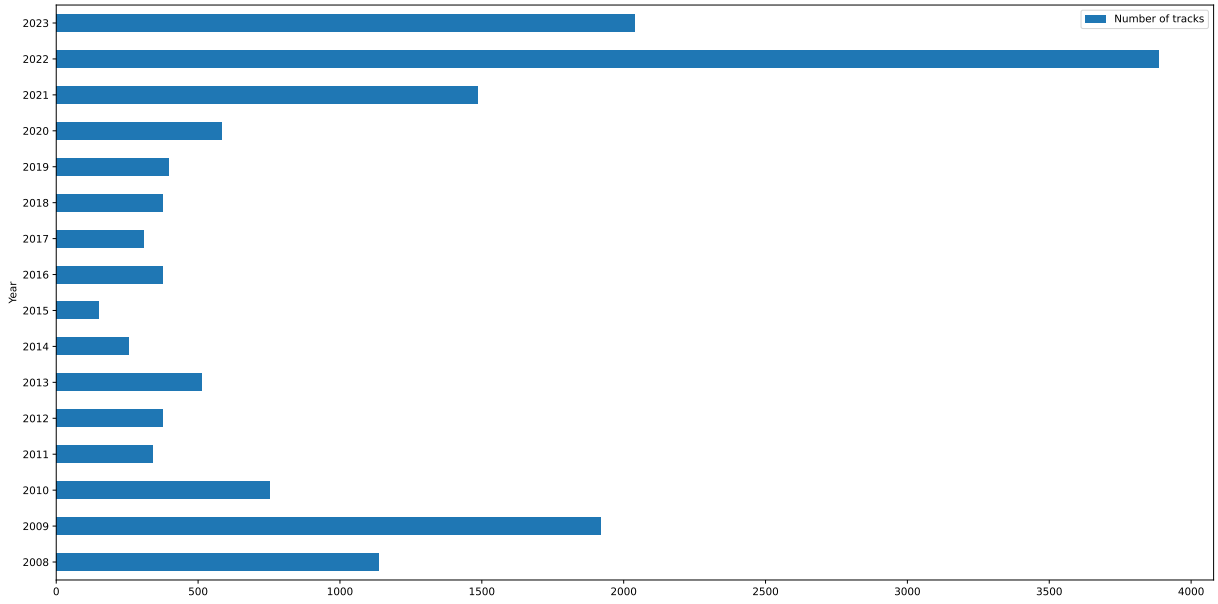

Fig. 8. Distribution of tracks over time, according to the tracks' wiki publishing date.

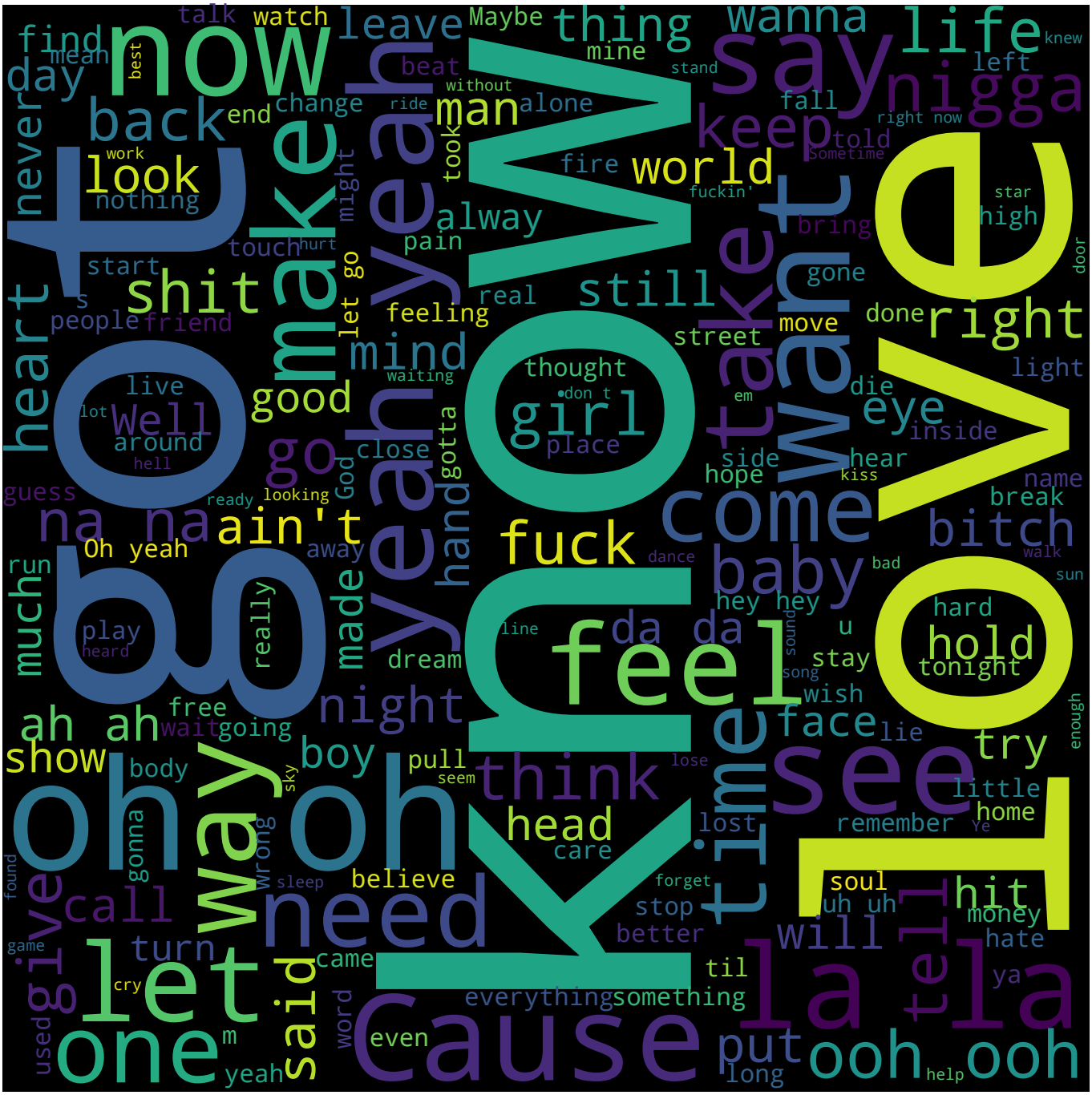

Fig. 9. Word cloud representation of the most frequent words in the lyrics of our collection's tracks

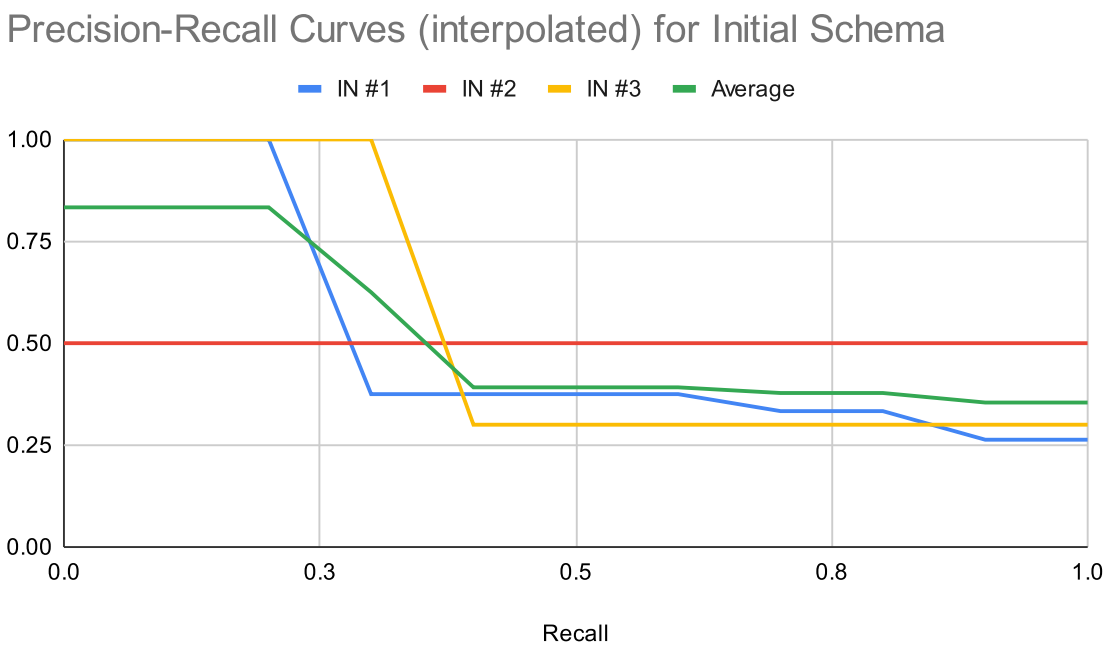

Fig. 10. Initial Schema - Precision-Recall Curves (interpolated)

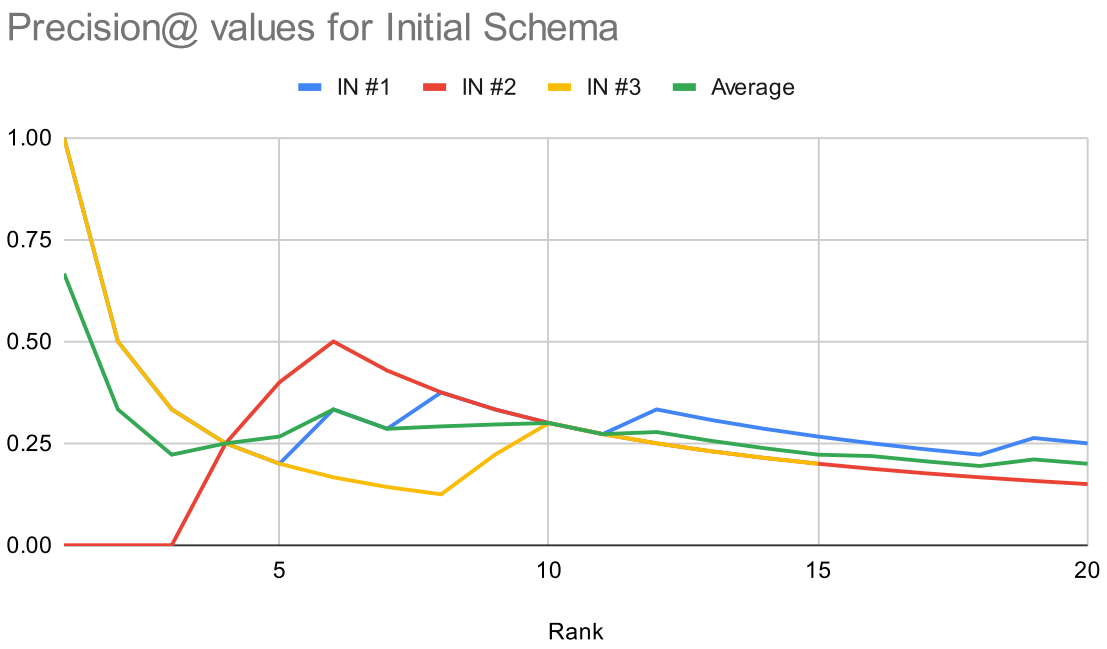

Fig. 11. Initial Schema - Precision@ Values

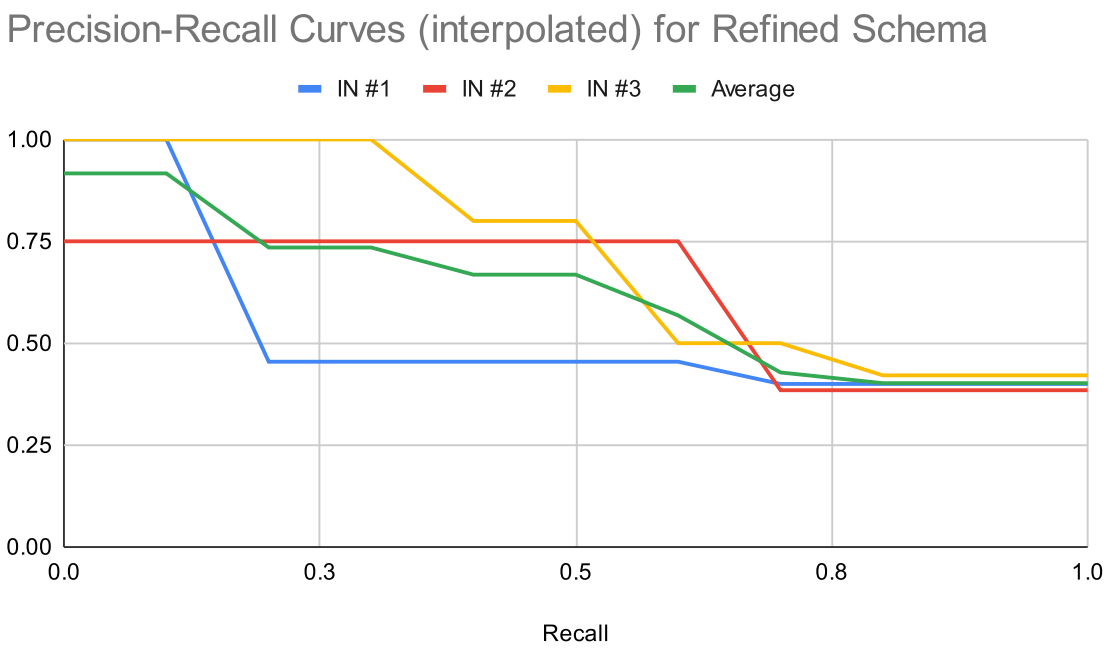

Fig. 12. Refined Schema - Precision-Recall Curve (interpolated)

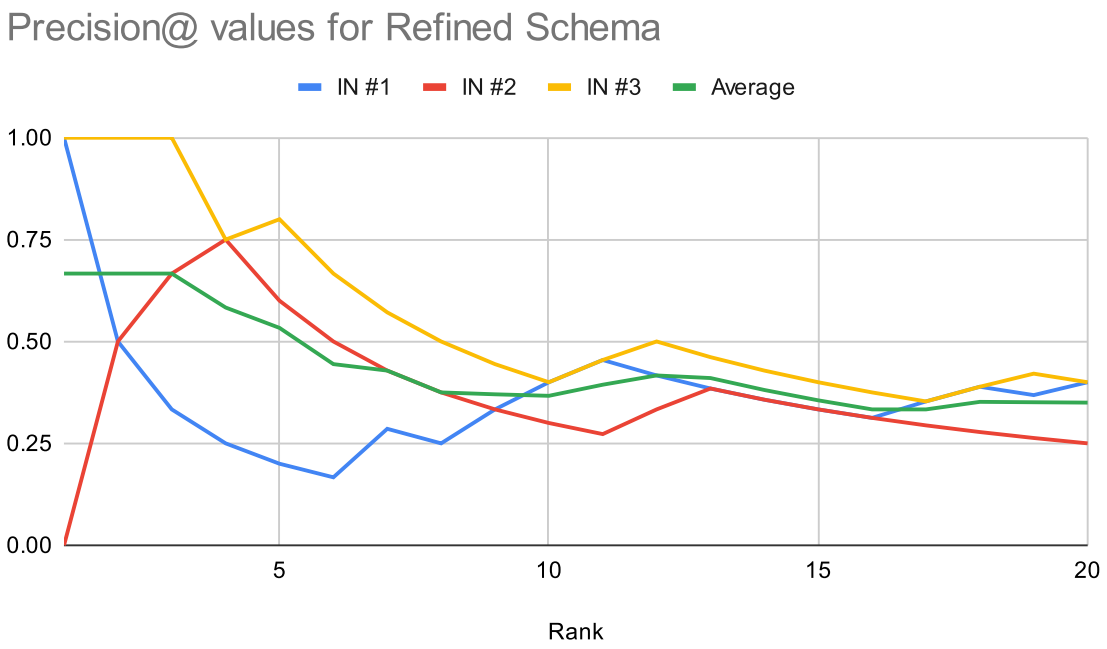

Fig. 13. Refined Schema - Precision@ Values

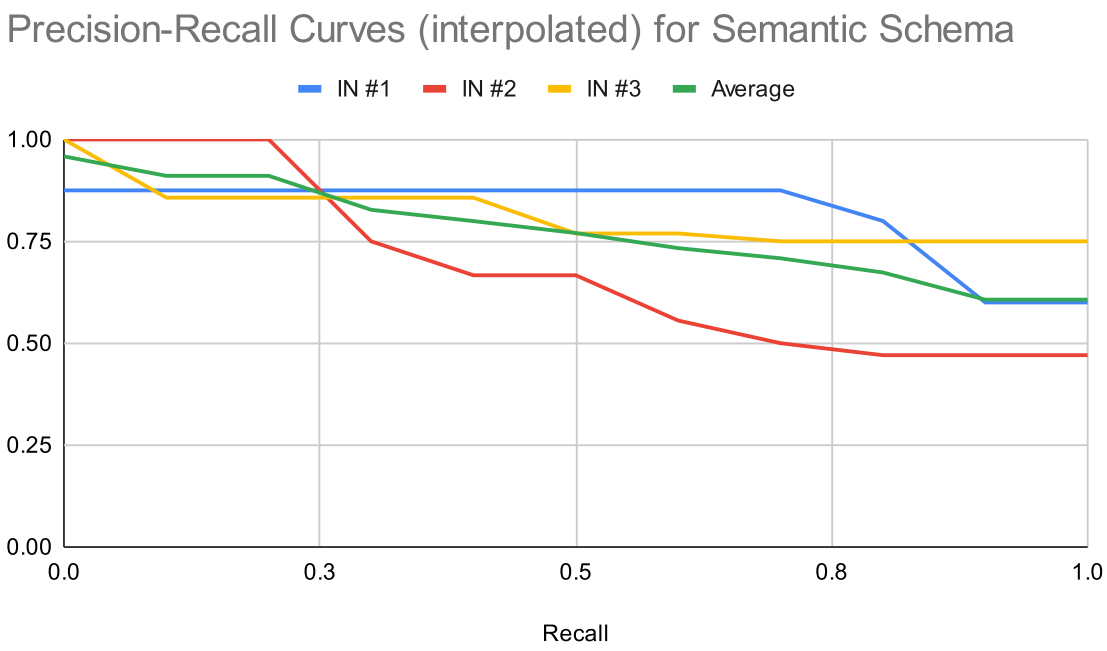

Fig. 14. Semantic Schema - Precision-Recall Curves (interpolated)

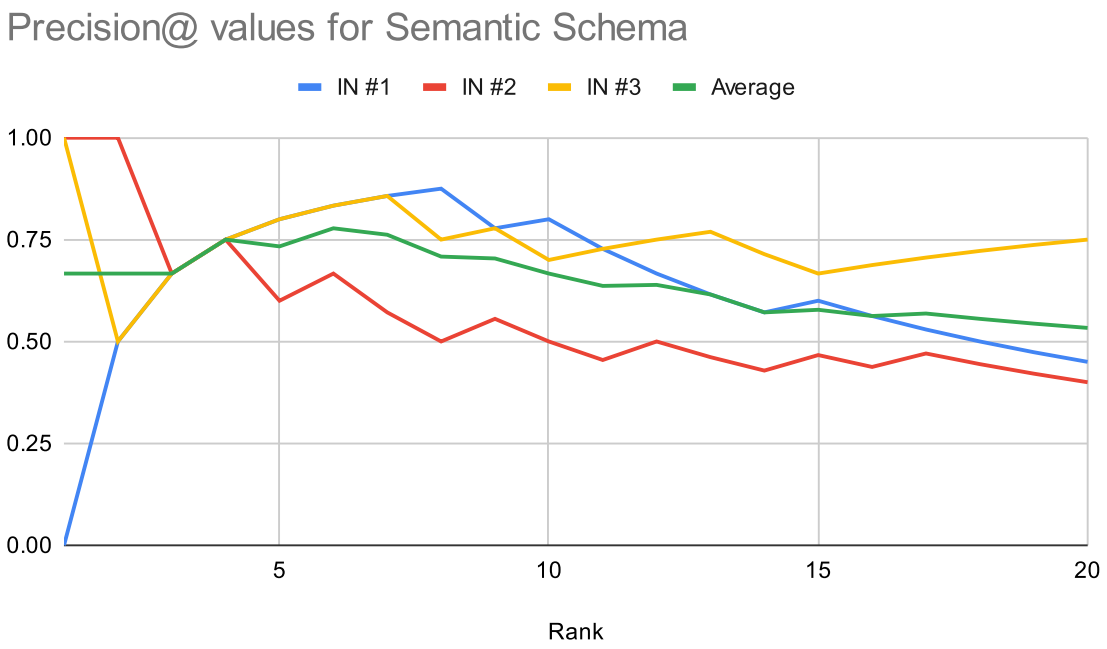

Fig. 15. Semantic Schema - Precision@ Values

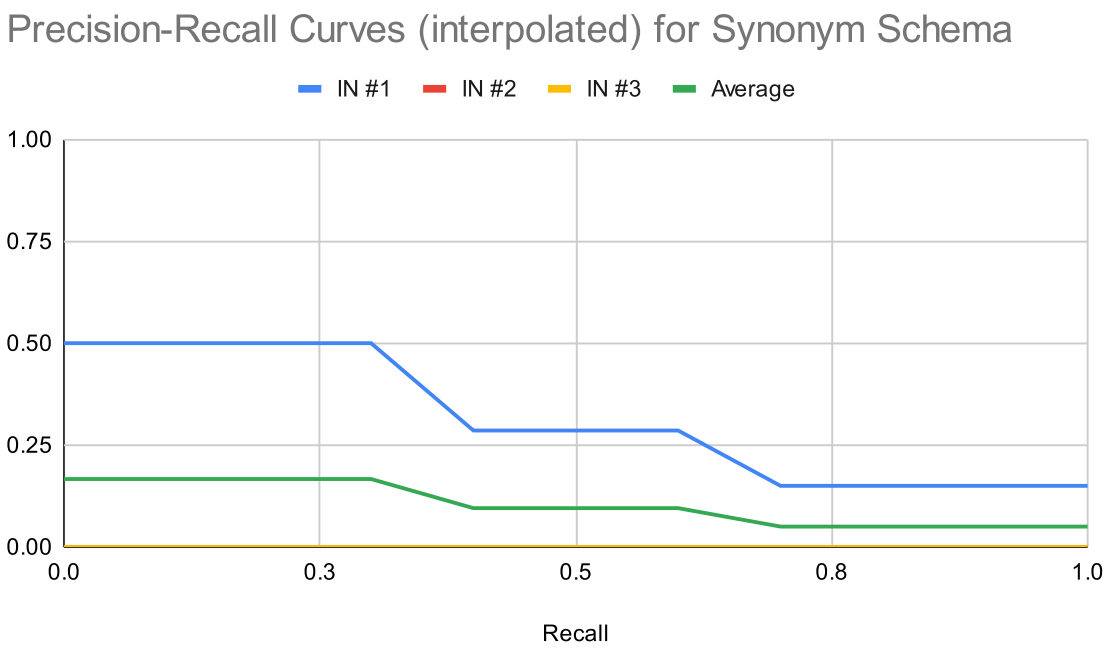

Fig. 16. Synonym Schema - Precision-Recall Curves (interpolated)

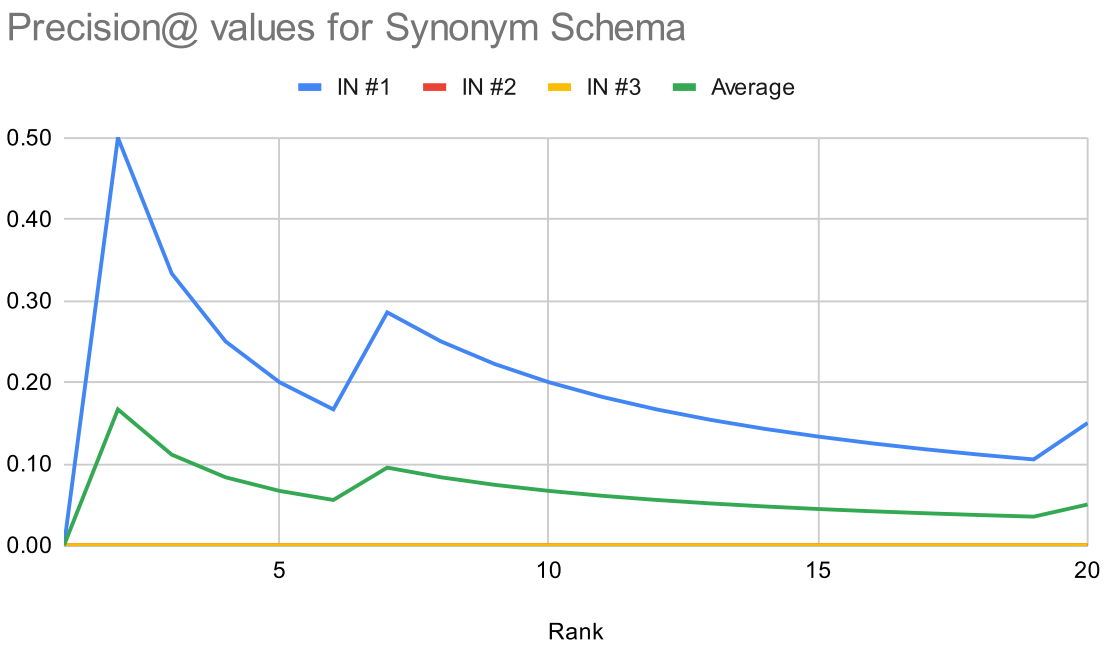

Fig. 17. Synonym Schema - Precision@ Values

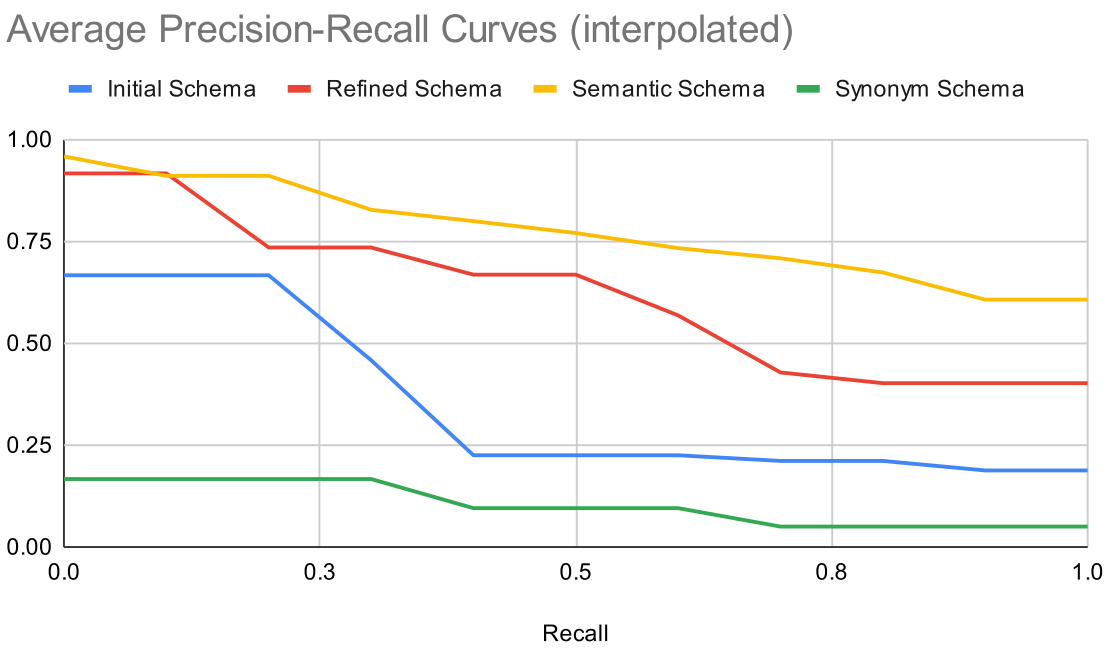

Fig. 18. All Schemas - Average Precision-Recall Curves Comparison

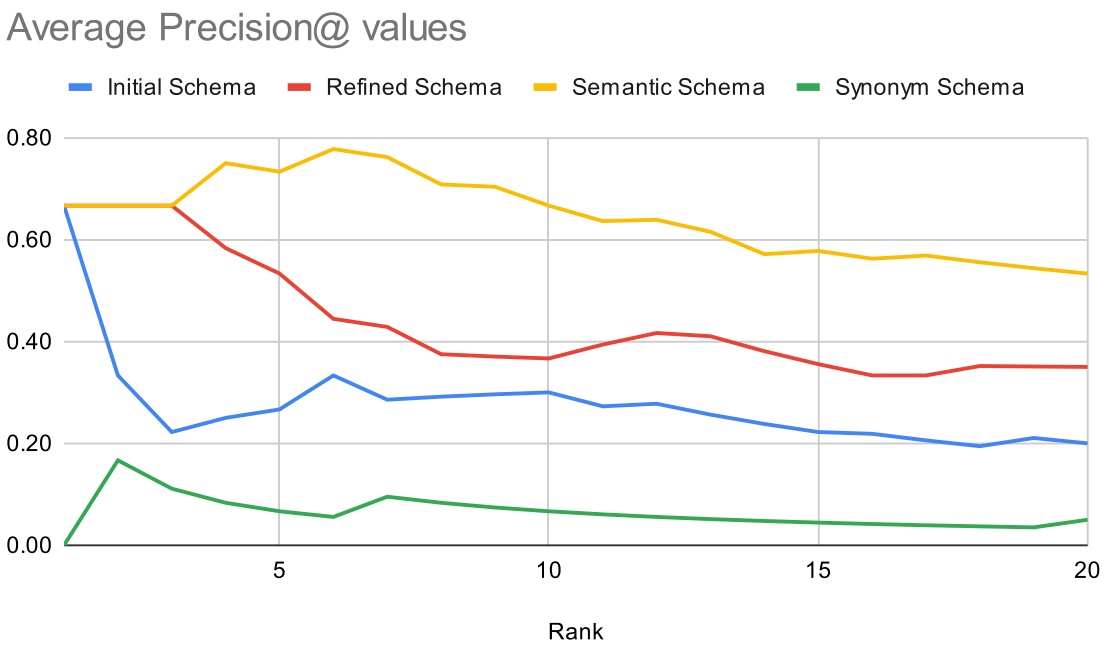

Fig. 19. All Schemas - Average Precision@ Values Comparison

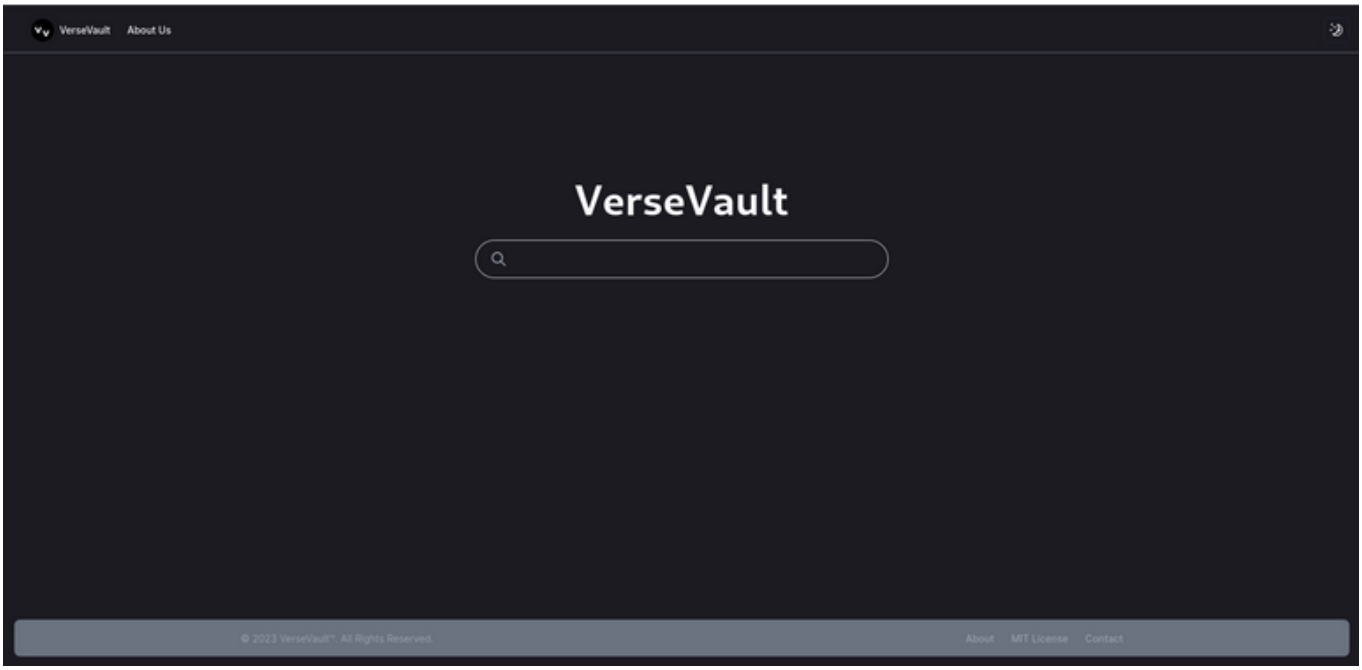

Fig. 20. VerseVault's HomePage

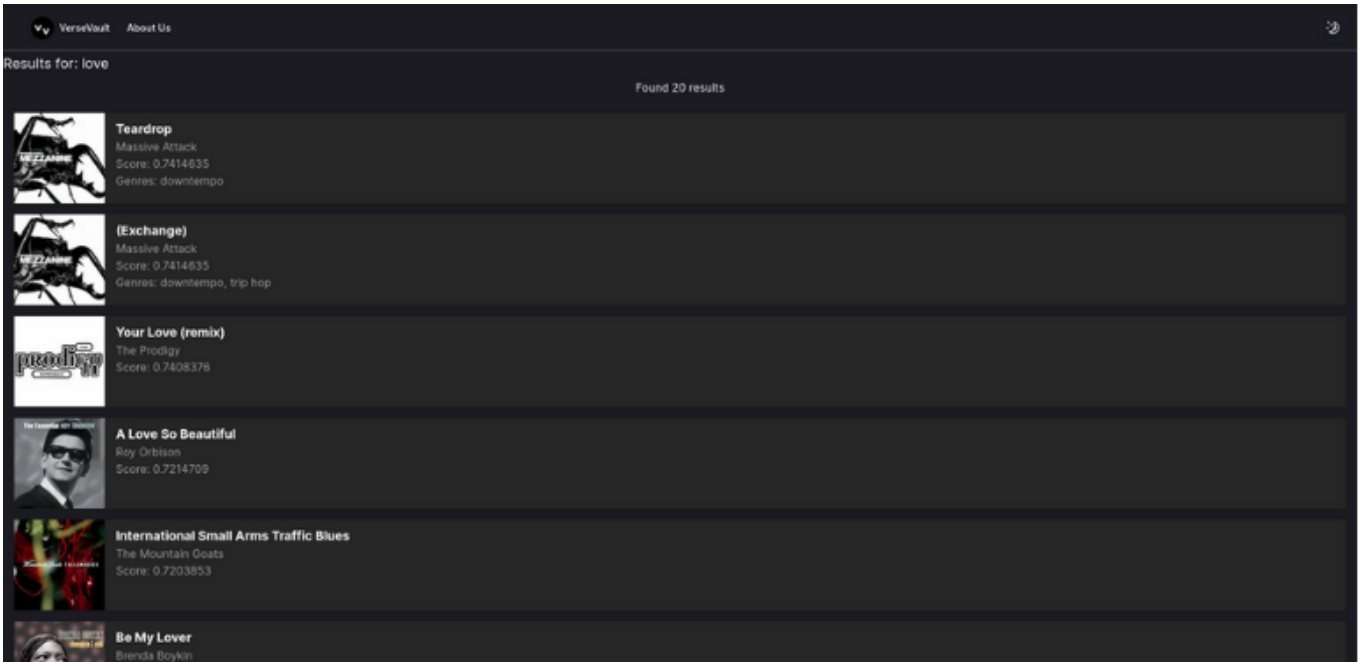

Fig. 21. VerseVault's Results Page

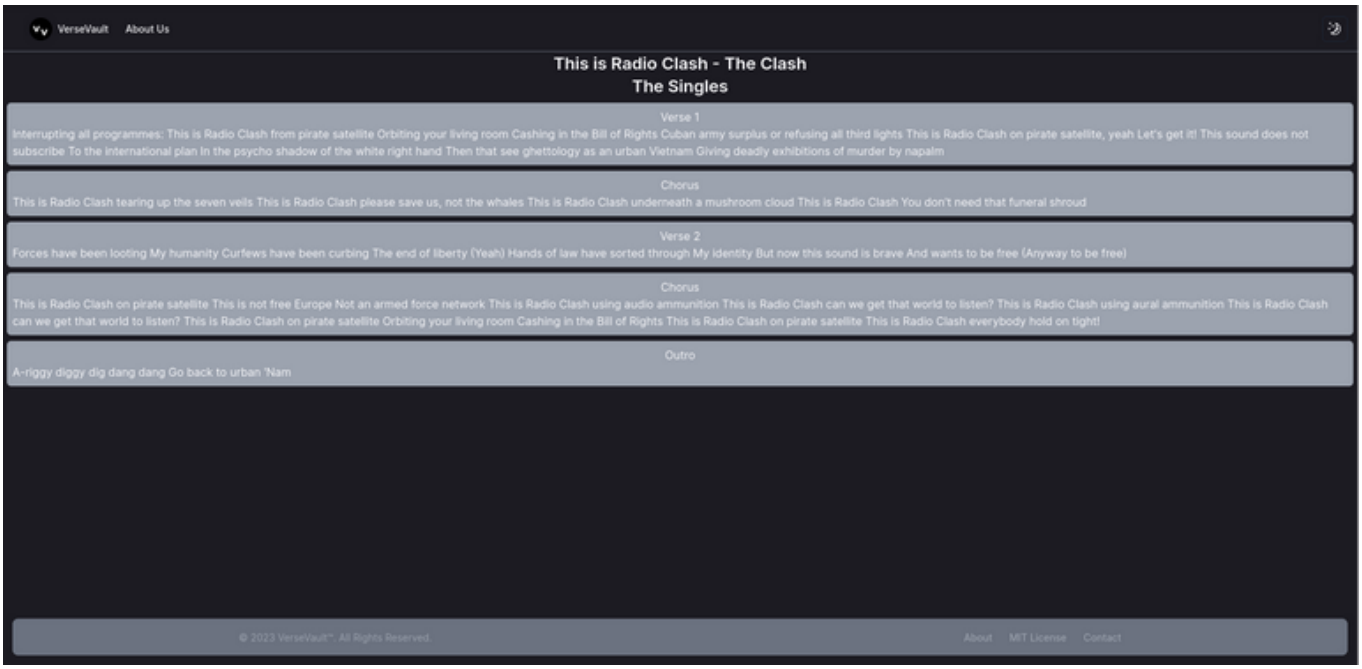

Fig. 22. VerseVault's Track Result Page

Rank	IN #1	IN #2	IN #3
1	R	N	R
2	N	N	N
3	N	N	N
4	N	R	N
5	N	R	N
6	R	R	N
7	N	N	N
8	R	N	N
9	N	N	R
10	N	N	R
11	N	N	N
12	R	N	N
13	N	N	N
14	N	N	N
15	N	N	N
16	N	N	-
17	N	N	-
18	N	N	-
19	R	N	-
20	N	N	-

Fig. 23. Individual Assessments of the Initial Search System

Rank	IN #1	IN #2	IN #3
1	Y	N	R
2	N	Y	R
3	N	Y	R
4	N	Y	N
5	N	N	R
6	N	N	N
7	Y	N	N
8	N	N	N
9	Y	N	N
10	Y	N	N
11	Y	N	R
12	N	Y	R
13	N	Y	N
14	N	N	N
15	N	N	N
16	N	N	N
17	Y	N	N
18	Y	N	R
19	N	N	R
20	Y	N	N

Fig. 24. Individual Assessments of the Refined Search System

Rank	IN #1	IN #2	IN #3
1	N	R	R
2	R	R	N
3	R	N	R
4	R	R	R
5	R	N	R
6	R	R	R
7	R	N	R
8	R	N	N
9	N	R	R
10	R	N	N
11	N	N	R
12	N	R	R
13	N	N	R
14	N	N	N
15	R	R	N
16	N	N	R
17	N	R	R
18	N	N	R
19	N	N	R
20	N	N	R

Fig. 25. Individual Assessments of the Semantic Search System

Rank	IN #1	IN #2	IN #3
1	N	N	N
2	R	N	N
3	N	N	N
4	N	N	N
5	N	N	N
6	N	N	N
7	R	N	N
8	N	N	N
9	N	N	N
10	N	N	N
11	N	N	N
12	N	N	N
13	N	N	N
14	N	N	N
15	N	N	N
16	N	N	N
17	N	N	N
18	N	N	N
19	N	N	N
20	R	N	N

Fig. 26. Individual Assessments of the Synonym Search System