Spot & Find: An analysis of search systems and their data

Eduardo Correia, João Cardoso, José Eduardo Henriques

Department of Engineering, University of Porto

January 19, 2022

Abstract

In this work, we collected structured and textual data from a music dataset. After that, we refined that information using both Spotify's and genius' APIs, while also filtering out invalid information obtained from the ladder.

We also designed an Information Retrieval system around this data using the search engine *Solr*, allowing users to easily search for artists, their tracks, and respective albums, using different fields, such as the artist's genre, their popularity, the release year of an album, *etc...* By creating a Solr schema configuration, the accuracy of these results was then significantly improved. Finally, a frontend was created to facilitate the usage of the search engine and improve the overall user experience. **Index Terms: Data Extraction, Data Enrichment, Data Refining, Music, Spotify API, Genius API, Information Retrieval, Indexing Process, System Evaluation, Solr, Tokenizer, Filter, Ontology**

1 Introduction

Music has become an essential part of the lives of many people. According to a study from 2016 [1], 68% of smartphone users stream music daily.

Because of this, there are already multiple services that provide music databases. However, there currently isn't a music database that allows people to obtain music lyrics in bulk in addition to additional data, such as the music's energy and duration. The services that have the most similar services are *Spotify*, which doesn't provide music lyrics in bulk, and *Genius*, which provides lyrics and sometimes more general track information, such as the release date.

The *Dataset Preparation* section of this report describes the process taken to combine information from the two most common music services, from Data Collection and Enrichment to Data Refining, including a detailed description of our database model.

The *Information Retrieval* section details the process of using the previously constructed dataset to retrieve information related to any search query, starting with the creation of the JSON document, the schema definition, some example queries and evaluations of their results which demonstrate the *Retrieval Process*, and some features which got removed.

The *Search System* section details the final adjustments and features necessary to retrieve the desired data from the dataset that was being developed with accuracy and ease of use, through a web page and an ontology.

Finally, there is a shortlist of improvements which could be done in the future.

2 Dataset Preparation

The dataset used in the project was retrieved from a challenge dataset [2] for music recommendation research, made by *Spotify*, which contains one million playlists (31.2GB).

I Original Dataset Description

The data comes sliced into one thousand JSON files (each one containing one thousand playlists) and consists of a list of playlists, each one having a list of numerous tracks. Each track contains information such as the track, album, and artist's respective names and Spotify URIs, as well as other useful information.

The dataset is from 2018 and only contains information from 2017 or earlier.

II Data Collection/Enrichment

This subsection details some of our decisions related to collecting information on all of our database's attributes.

A Most Attributes

A Python script (`populate.py`) was created to retrieve the information about each track and album, each track and album's artist(s), and music genre, from the song URIs.

At first, for each track added, every track from its respective album was added as well.

However, because, each album had a very high amount of tracks, which would significantly reduce the album diversity of the final database, this was not kept for the delivery.

B Song Lyrics

Another Python script (`lyrics.py`) utilises the *lyricsgenius* library. This library relies on the *Genius* API and web

[2] "Spotify Million Playlist Dataset Challenge", *AIcrowd*, https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge

scraping to obtain the lyrics for each track. This is because the *Genius* API by itself does not return the lyrics directly.

The *lyricsgenius* library has to use web scraping to retrieve them. Other music lyrics APIs were tried, such as Lyrics API and MusixMatch API, however, each had its flaws, namely as rate limiting and missing data.

III Database Model

Below is a more detailed description of each attribute from each table, disregarding ones whose definition is self-evident.

- **Track:**
  - **uri:** The track's URI from Spotify.
  - **explicit**: A boolean indicating whether the track's name or lyrics contain explicit [3] words. The Album and Artist tables have the same field, relative to their respective names.
- **Album:**
  - **uri:** The album's URI from Spotify
  - **album_type**: An enumeration value indicating whether the album is a regular, single or compilation album.
- **Artist**:
  - **uri:** The artist's URI from Spotify
  - **popularity:** An artist's popularity, from 0 to 100.
  - **release_date:** The release date of the album in YYYY-MM-DD format.
- **Track Features:**
  - **acousticness:** A confidence measure from 0.0 to 1.0 of whether the track is acoustic.
  - **danceability:** How suitable a track is for dancing, from 0.0 to 1.0.
  - **duration_ms:** Duration of the track in milliseconds.
  - **energy:** A measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity.
  - **instrumentalness:** Predicts whether a track contains no vocals, from 0.0 to 1.0.
  - **liveness:** Detects the presence of an audience in the recording.
  - **loudness:** The overall loudness of a track in decibels (dB).
  - **mode:** Mode indicates the modality (major, 1, or minor, 0) of a track.
  - **speechiness:** Presence of spoken words in a track, from 0.0 to 1.0.
  - **tempo:** The overall estimated tempo of a track in beats per minute (BPM).
  - **time_signature:** An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).

An SQLite database was created to store all the data mentioned in the previous subsection.

A UML representation of the created database is shown in Figure 2.

Since the initial dataset is huge, we only used a subset consisting of about 357 playlists to gather information.

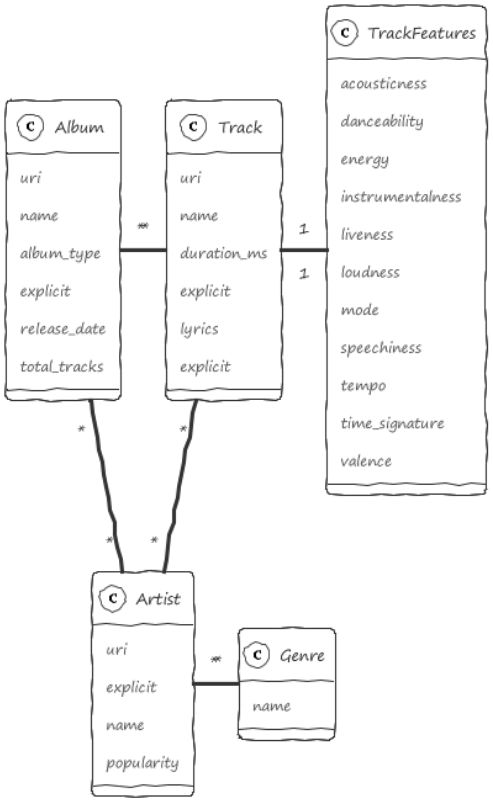

Figure 1: Conceptual model

A Data Refining

The data refining step was handled in the `filter.ipynb` Jupyter notebook, which does the following:

- Loads the database using the *sqlite3* Python library and loads each table into a *pandas* dataframe.
- Verifies if there are any null values in any table. For this dataset, the only null values were in the track's lyrics column.
- Removes empty string fields. Removes tracks with duplicate song lyrics. If there are two tracks with the same lyrics, it drops them both.
- Adds the explicit column to track, album and artist. Verifies if their names and the track's lyrics are explicit and sets the explicit column's value to True if so.
- Using SQL queries and SQLAlchemy executes queries that modify the database to be consistent.

In addition, "feat. <artist>" strings were removed from track names, and any instrumental versions of tracks were removed.

IV Data Issues

This section details a group of past and present issues in the project and their corresponding solutions and workarounds.

A Data Collection/Enrichment

Many tracks did not have lyrics. This was solved by removing those tracks from the database.

A few other tracks had nonsensical lyrics, even with the correct *lyricsgenius* library constructor arguments. A great majority of the nonsensical lyrics were repeated for multiple different tracks, so those were removed.

As previously referred to in "Dataset Information/Most Attributes", searching for each track in an album could have significantly reduced the necessary number of API calls. However, since these calls only took a relatively small amount of time and it would significantly reduce the database's album diversity, this was undone. The former point means that not every album has all of its tracks in the database.

Even with "remove subsection headers" set to true, some tracks still had them in their lyrics. They were removed using SQL queries.

As previously mentioned, some tracks were instrumentals and, therefore, it would not make sense for them to have lyrics. Those were also removed.

Because this database's records aren't updated, the artist's future popularity changes won't be reflected in it.

The *sqlite3* Python library does not run the final filtering queries, those had to be run in SQLite3 itself. The queries are still present in the code, as they do not cause any harm.

V Dataset Characterisation

To better understand and find problems with the dataset, a few charts were created detailing different kinds of information.

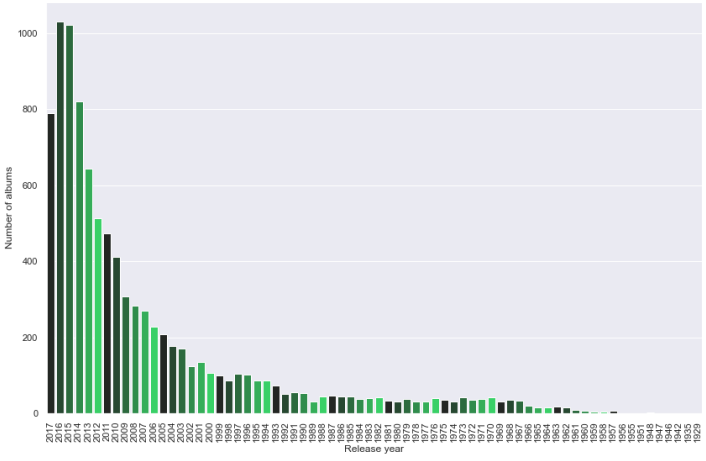

Figure 2: Number of albums over the years

Regarding the year each album was released, it can be concluded that the dataset has a much higher number of recent albums, which is to be expected since many older albums aren't available online, the music production tends to grow with each year and people tend to enjoy more modern music.

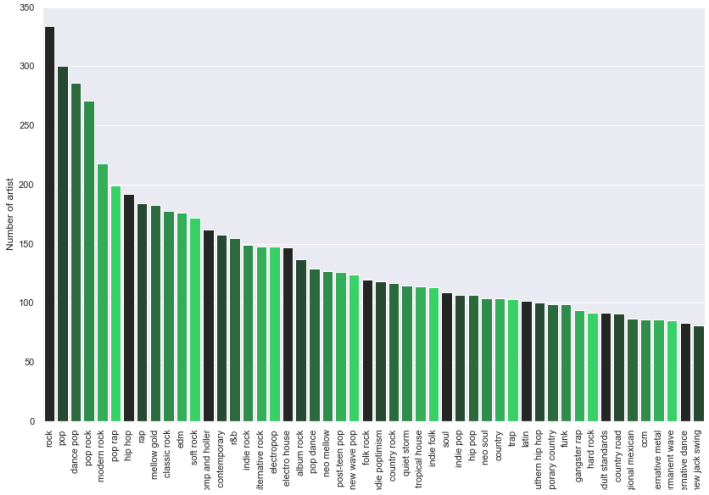

Figure 3: Top genres of artists

Concerning the genres of the artists, pop, rock, and rap are among the most popular and almost 1/10 of the artists are of one of these genres or a variation of it.

Also, after the top 5 genres, the most common genres among artists tend to follow a linear trend.

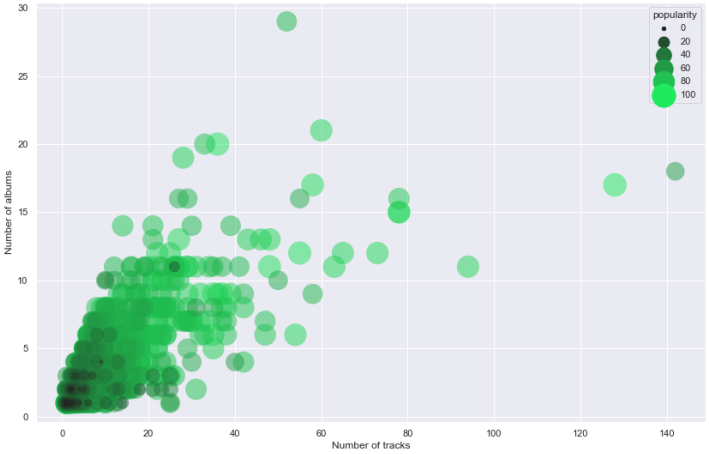

Figure 4: Artist popularity, number of tracks and albums

As can be seen in Figure 5, the popularity of a given artist and its number of albums and tracks are closely related.

There at lots of not popular artists and they have a low number of tracks and respectively albums, as such, they are located in the left lower corner of the chart.

On the other hand, the most popular artists are scarce and located in the upper right corner, as they have a higher number of tracks and albums.

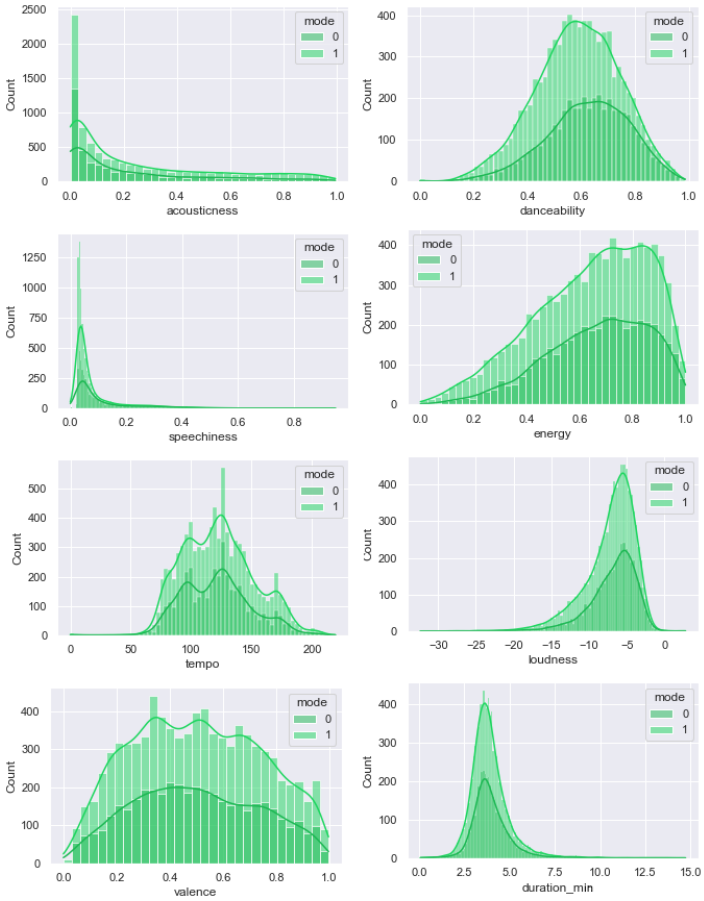

Figure 5: Audio features

Figure 6 represents several different audio features from the dataset tracks, such as acousticness, danceability, energy, speechiness, loudness, tempo, valence, and duration in minutes. It's also coloured differently depending on each track's mode.

Figure 6: Track lyrics wordcloud

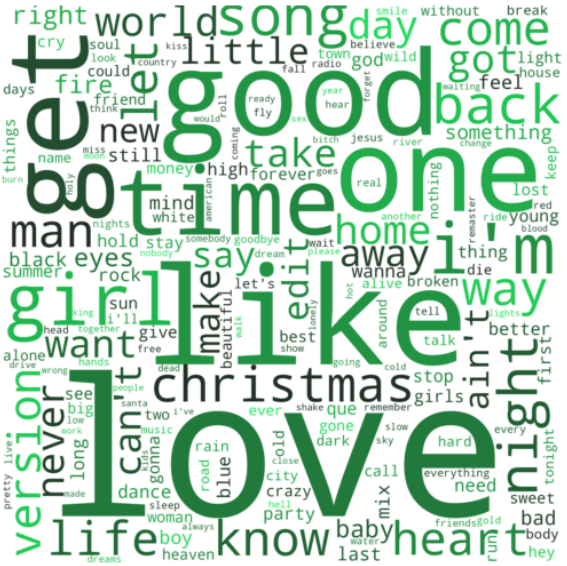

Figure 7: Track names wordcloud

Figures 7 and 8 represent, wordclouds of the most common words present in the tracks' lyrics and names, respectively, not counting stopwords or words with two letters or less.

3 Information Retrieval

The second part of this project is focused on information retrieval, the process of finding material, usually documents, within a collection that satisfies an information need.

For this, several tools could be used, namely *Solr* and *Elasticsearch* (which are both built on top of *Apache Lucene* library.

We decided to use *Solr* due to its maturity, extensive documentation, ease of setup and use, and functionalities.

I Collections and Documents

Initially, all of the data was saved in an SQLite database with multiple tables, so, to load this data to *Solr* and index it, we merged all the data into one JSON file, with an array of JSON objects for nested data. This was done using an auxiliary Python script.

To be able to use multiple documents, different approaches could be taken into account. The first is to use a different *Solr* core for each document type. Although these cores are independently queried, a unique list with all results can be obtained by using the `JOIN` command on the query or by joining the independent results lists. With this approach, three different collections would be used (track, artist, and album).

We decided, however, to use just one core named "music" with our merged JSON data loaded into it.

II Indexing Process

One of the most important steps in Information Retrieval is indexing, which reduces the documents to the informative terms contained in them.

Several field types were created for different variables, one for track/artist/album names (named "name") and another

one for track lyrics (named "lyrics"). The former uses fewer filters since those searches tend to be more specific.

In addition, a few types were created, such as `int`, `float`, `bool` and `generic_text`, which are the default Solr types. These were just used to better organize the project.

Several new fields were created, each representing a field from their corresponding SQL tables. The artists' and album's artist data fields have `multiValue` set to true.

Some fields, such as track, album, and artist URIs have indexed and stored set to false. Those fields were determined to not serve any practical purpose for the end-user.

Every field has `uninvertible` set to false. This is because doing so is highly recommended, according to the documentation [4].

A Used filters

For track/artist/album/albums' artists' names, at both index and query time, the following filters were used:

- *ASCII Folding Filter.* [2]
- *Lower Case Filter.* Converts words to lowercase. [2]
- *English Possessive Filter.* Removes apostrophes, except ones from plural words. [2]
  Example: "dog's" -> "dog"; "dogs'" -> "dogs'"
- *Common Grams Filter.* Joins stop words with common names directly in front of them. [2]
  Example: "the" "dog" -> "the_dog"; "the" "platypus" -> "the_platypus".
- *Hyphenated Words Filter* (only on index time, as recommended by the documentation). Reconstructs hyphenated words that were separated by the tokenizer. [2]
  Example: Input: "A hyphen- ated word" -> Tokenized: "A", "hyphen-", "ated", "word" -> Output: "A", "hyphenated", "word"

For track lyrics, every filter used for the previous field type was also used, except the Common Grams Filter. In addition, at both index and query time (unless otherwise specified), the following filters were used:

- *Synonym Graph Filter & Flatten Graph Filter* (only at index time). Explained after the list.
- *Stop Filter.* Removes stop words.
  Example: "the". No stop word file was used.
- *Porter Stem Filter.* Converts words into their base forms.
  Example: "jumping" gets converted to "jump"

A *Synonym Graph Filter* was used with default arguments, except for the file name. This filter requires a list of synonyms and their equivalent words, which is shown in info box 1.

The following1 table shows the filters used and their respective order.

Field Type	Filter(s)	Index	Query
name	ASCIIFoldingFilter	X	X
	LowerCaseFilter	X	X
	EnglishPossessiveFilter	X	X
	CommonGramsFilter	X	X
	HyphenatedWordsFilter	X	
lyrics	ASCIIFoldingFilter	X	X
	LowerCaseFilter	X	X
	EnglishPossessiveFilter	X	X
	SynonymGraphFilter		X
	StopFilter	X	X
	HyphenatedWordsFilter	X	
	PorterStemFilter	X	X

Table 1: Custom field types

Field name	Field type	Indexed
name	name	Yes
lyrics	lyrics	Yes
duration_ms	int	No
acousticness	int	No
danceability	float	No
energy	float	No
instrumentalness	float	No
liveness	float	No
loudness	float	No
mode	bool	No
valence	float	No
speechiness	float	No
tempo	float	No
time_signature	int	No
explicit	bool	No
artists.popularity	float	No
artists.name	name	Yes
artists.genres	name	Yes
artists.explicit	bool	No
albums.name	name	Yes
albums.album_type	generic_text	No
albums.release_date	generic_text	No
albums.total_tracks	int	No
albums.artists.name	name	Yes
albums.artists.popularity	int	No
albums.explicit	bool	No

Table 2: Schema fields

come => advance, approach, arrive, near, reach
go => depart, disappear, fade, proceed, recede, travel
run => dash, escape, elope, flee, hasten, race, rush, speed
...

InfoBox 1: List of synonyms

[2] "Apache Solr Reference Guide; Filter Descriptions" https://solr.apache.org/guide/8_11/filter-descriptions.html

III Retrieval Process + System evaluation

This section showcases the results before the changes described in the Search System section. The section "Search System -> Retrieval Process + System evaluation" showcases the results after the changes.

To evaluate the different systems, we defined 4 different information needs (IN) based on the possible retrieval tasks.

For each information need, we provide a simple description and the relevant judgment to decide if a document is relevant or not and analyze the top 10 results and their relevance. To calculate the recall, because manually finding every relevant document in the entire dataset is impossible, the universe considered was the top 20 results. Therefore, precision at 10 (P@10) is the number of relevant queries in the top 10 divided by 10, **Average precision (AP) and Discounted Cumulative Gain (DCG) are only calculated using the first 10 elements**. A higher precision indicates the system retrieves more useful documents, a higher AP and DCG indicate that the relevant documents appear earlier in the search results. DCG is also the best at comparing substantially different ranking functions. Because of the way the recall is calculated, it tends to be highly unstable.

Finally, **for the precision/recall (P/R) curves, unlike the rankings, the top 20 results are used for both precision and recall**. The system was evaluated this way because the first approach obtained more useful/stable results for the ranking but was not usable for P/R curves

The query fields used were the indexed ones, i.e. `name`, `lyrics`, `artists.name`, `artists.genres` and `albums.name`.

The following weight boosts were chosen to both test the impact of their use and find the fields which should be weighed the most/least. WF0 serves as the control group for this experiment, since no fields were boosted.

System	name	lyrics	artists.name	artists.genres	albums.name
WF0	0	0	0	0	0
WF1	5	3	1	0	3
WF2	1	3	0	5	0
WF3	3	5	0	0	0

Table 3: Weight boosts for each system

A Live songs

wt Songs that were recorded live during a concert, in counterpart to their album version recorded in a studio. The track's liveness attribute was not considered, since, in this case, every live track said so in their name or lyrics.

Query: live

System	Relevant	P@10	Recall	AP	DCG
WF0	NNNNNRRRN RNRNNNRRNR	0.3	0.375	0.242	0.950
WF1	NRRRRNRRRN NRRRRRRRRR	0.7	0.4375	0.708	2.898
WF2	NNNNNNNNN NNNNNNNNN	0.0	0.0	0.0	0.0
WF3	NNNNNNNNN NNNNNNNNN	0.0	0.0	0.0	0.0

Table 4: Live songs results

B Danceable songs

Hypothesis: Songs that have "dance" in one of their attributes may be danceable. To deem a song as danceable, its danceability value must be greater than 0.7 (there isn't a danceable threshold in Spotify's API reference, 0.7 was picked by trial and error)

Query: dance

System	Relevant	P@10	Recall	AP	DCG
WF0	NNRNRNNRRR NRRRRNRNNR	0.5	0.454	0.410	1.792
WF1	NRNRRRNRRR RNRRRNNNNR	0.7	0.583	0.608	2.710
WF2	NRNNNRRNRR RNNNNNRRNR	0.5	0.555	0.441	1.911
WF3	NRNNNRRNRR RNNNRRNRRR	0.5	0.455	0.441	1.911

Table 5: Danceable songs results

Using sql querries on the original dataset, the percentage of danceable songs in the dataset is 3542 / 12828 * 100% = 27.611 10

Since the best weight system 70% precision at 10, which is close to 3x over-representative of the general population, our hypothesis was verified.

```
SELECT count(*) FROM TRACK WHERE danceability >= 0.7
```

count(*)
3542

Figure 8: Danceable Track Count

```
SELECT count(*) FROM TRACK
```

count(*)
12828

Figure 9: Track Count

C Pop songs remixes

Remix versions of pop genre songs.

Query: pop AND remix

System	Relevant	P@10	Recall	AP	DCG
WF0	NNNNNNNNNN NNNNNNRNRN	0.0	0.0	0.0	0.0
WF1	NNRNRNNNNN NNNNNNRNNR	0.2	0.5	0.367	0.887
WF2	RNNNNNNNNN NNNNNNNRRN	0.1	0.333	1.0	1.0
WF3	RNNNNNNNRN NNNNRNNNNN	0.2	0.666	0.611	1.301

Table 6: Pop songs remixes results

D Rock/Metal songs about love

Rock or metal genre songs that are about love.

Query: (rock OR metal) AND love

System	Relevant	P@10	Recall	AP	DCG
WF0	RNRRNNNNNN NNRRRRNNNN	0.3	0.429	0.806	1.931
WF1	NRRRNNRNRN RNNNRNNNRN	0.5	0.625	0.609	2.196
WF2	NRRNNNNRRN RNRNNNRNNN	0.4	0.571	0.497	1.747
WF3	NRRNNNNNNN NNNNNNNNNN	0.2	1.0	0.583	1.131

Table 7: Rock/Metal songs about love results

E Songs about love

Songs about love.
Query: love

System	Relevant	P@10	Recall	AP	DCG
WF0	RRNNNNNNNR RNRNRRNRNN	0.3	0.375	0.767	1.920
WF1	RRRRRRRRRR RRRRRRRRRR	1.0	0.5	1.0	4.544
WF2	RRNNNNNNNR RNRNRRNRNN	0.3	0.375	0.767	1.920
WF3	RRRRRRRRRR RRRRRRRRRR	1.0	0.5	1.0	4.544

Table 8: Rock/Metal songs about love results

IV Unused Considerations

The following features were considered, but eventually dropped or never implemented:

A Tokenizers

Other than the default one, none of Solr's available tokenizers were useful for this project, therefore, only the former was used.

If the dataset had emails or URLs, then the classic tokenizer would have been useful.

B Filters

- *Fingerprint Filter* (for genre). Genres such as "hip hop" should be searchable by only writing "hop", so it was discarded.
- *ICU Folding Filter*. It will be reconsidered should the removal of certain Unicode characters be necessary.
- *KStem filter* (for lyrics). Porter Stem was used instead since KStem only considers English characters. In addition, KStem purposely does not change some words (such as "dogs" to "dog"), which is not intended in this project.
- *Word Delimiter Graph Filter*. After consideration, it was determined that not every non-alphanumeric character should not be discarded.
- *Classic Filter*. Would be useful if the classic tokenizer was used, otherwise redundant, because of the *English Possessive Filter*.

C Field Properties

- *Large* (for track lyrics). Almost no lyrics are above 512KB, so this property would be useless.
- *Required*. Not necessary, due to the non-existence of null values in the dataset.
- *docValues*. This field is meant to be used for sorting and faceting since it is much faster at doing those. However, after some deliberation, this field was set to false because search speed was prioritized over sorting speed.

4 Search System

The third part of this project is primarily focused on experimenting with the previous search system to find possible improvements and implementing what was detailed in the previous part as future work. In addition, a frontend has been created to allow for new searching related features and improve the overall user experience. Finally, an ontology was created to better visualize the structure of our data.

I Indexing Process Updates

The current indexing process has received significant changes.

Firstly, there are distinct field types for track/album names, track lyrics, artist names and genre names. All of those fields, except for genre name, have a `Regular Expression Tokenizer`. The tokenizer's regex separates every word (group of at least one character in a row which only contain: a to z, A to Z, 0 to 9 and/or underscore), but it considers some multi-word genres (e.g.: "latin rock") as only a single token. It is also case insensitive (it has the (?i) flag at the beginning).

Additionally, an album's release date has now date type, instead of being just a string and an album's type is now an enumeration with three possible values: "album", "single" and "compilation". This was done by creating a type using the `solr.EnumFieldType` class, which reads the possible values from a file named `enumsConfig.xml` and has `docValues` set to true.

A Added/Modified filters

This section details the new added filters and their purpose. The main points of interest are the addition of phonetic filters[4], the genre weighting system and the expansion of the synonym list. The first part refers to the Beider Morse and Double Metaphone Filters added to the main search fields. These add a phonetic version of each token. The former is better for names and the ladder is better for general English words.

The second part refers to our use of a second synonym filter/list and a delimited boost filter to add weights to genre words. This, together with the previously mentioned regex tokenizer, means that genre words (such as "rock" and "pop") are weighed less (15%) in every main field except the genre. Unfortunately, because those fields also have a phonetic filter, the phonetic counterparts of those words are also added, which means the impact of this change was somewhat minimized. A A Protected Term Filter[3] could not be used in this situation, since it does not allow phonetic filters, as of the latest version of Solr (8,11).

Finally, the synonym list has been expanded (about 5x as many synonyms as before) and fixed. Previously, "=>" syntax was used, which only allowed the word on the left of a line to convert to the ones on the right. Now, "," syntax is used, allowing any word from either side to convert to all the words in that line.

[4] "Apache Solr Reference Guide; Phonetic Matching", https://solr.apache.org/guide/8_11/phonetic-matching.html
[3] "Apache Solr Reference Guide; Protected Term Filter", https://solr.apache.org/guide/8_11/filter-descriptions.html#protected-term-filter

Old syntax line example: come => advance, approach, arrive, near, reach

New syntax line example: come, advance, approach, arrive, near, reach

Note: To get a single token for a multiple word genre (such as "hip hop"), the regex inside the tokenizers had to specify every word. Doing so would make Solr take 5 minutes to load, so only the first 50 genre words were used. This means that the only genre words that are detected are single-word ones and the first 50 multi-word ones.

The table below shows the filters used for each field.

Notes:

1. Filters added in this part will be marked with an asterisk.

2. Fields marked with 2X at index or query time mean that they were used twice for different purposes.

3. New fields will only have things tagged as "NEW" if they previously weren't in the "name" field type.

Field Type	Filter(s)	Index	Query
name	ASCIIFoldingFilter	X	X
	LowerCaseFilter	X	X
	EnglishPossessiveFilter	X	X
	SynonymGraphFilter*	X	2X
	DoubleMetaphoneFilter*	X	X
	CommonGramsFilter	X	X
	HyphenatedWordsFilter	X	X*
artist name	ASCIIFoldingFilter	X	X
	LowerCaseFilter	X	X
	EnglishPossessiveFilter	X	X
	SynonymGraphFilterFactory*	X	2X
	DelimitedBoostTokenFilterFactory*		X
	BeiderMorseFilterFactory*	X	X
	HyphenatedWordsFilter	X	
	CommonGramsFilter	X	X
genre name	ASCIIFoldingFilter	X	X
	LowerCaseFilter	X	X
	EnglishPossessiveFilter	X	X
	DoubleMetaphoneFilter*	X	X
	CommonGramsFilter	X	X
	HyphenatedWordsFilter	X	
lyrics	ASCIIFoldingFilter	X	X
	LowerCaseFilter	X	X
	EnglishPossessiveFilter	X	X
	SynonymGraphFilter	X*	(1+1*)X
	LengthFilterFactory*	X	X
	StopFilter	X	X
	DoubleMetaphoneFilter*	X	X
	HyphenatedWordsFilter	X	
	PorterStemFilter	X	X
	RemoveDuplicatesTokenFilter*	X	X
	DelimitedBoostTokenFilter*		X

Table 9: Custom field types

Notes: 1 - Whenever the synonym graph is being used once, it is to get a synonym for each word. Whenever the synonym graph is used a second time, it is to set weights to genre words. The DelimitedBoostToken filter then applies the weights which were previously set.

2 - RemoveDuplicatesTokenFilter only removes duplicate tokens in the same position. Eg: Starting tokens: "car" "car". After RemoveDuplicatesTokenFilter: "car" "car". Both words started at different positions and, therefore, neither was removed.

II Updated Retrieval Process + System evaluation

This section showcases the results after the changes described in the Search System section. The previous section "Information Retrieval -> Retrieval Process + System evaluation" showcases the results before the changes.

To evaluate the different systems, we defined 4 different information needs (IN) based on the possible retrieval tasks.

For each information need, we provide a simple description and the relevant judgment to decide if a document is relevant or not and analyze the top 10 results and their relevance. To calculate the recall, because manually finding every relevant document in the entire dataset is impossible, the universe considered was the top 20 results. Therefore, precision at 10 (P@10) is the number of relevant queries in the top 10 divided by 10, **Average precision (AP) and Discounted Cumulative Gain (DCG) are only calculated using the first 10 elements**. A higher precision indicates the system retrieves more useful documents, a higher AP and DCG indicate that the relevant documents appear earlier in the search results. DCG is also the best at comparing substantially different ranking functions. Because of the way the recall is calculated, it tends to be highly unstable.

Finally, **for the precision/recall (P/R) curves, unlike the rankings, the top 20 results are used for both precision and recall**. The system was evaluated this way because the first approach obtained more useful/stable results for the ranking but was not usable for P/R curves

The query fields used were the indexed ones, i.e. `name`, `lyrics`, `artists.name`, `artists.genres` and `albums.name`.

The following weight boosts were chosen to both test the impact of their use and find the fields which should be weighed the most/least. WF0 serves as the control group for this experiment, since no fields were boosted.

System	name	lyrics	artists.name	artists.genres	albums.name
WF0	0	0	0	0	0
WF1	5	3	1	0	3
WF2	1	3	0	5	0
WF3	3	5	0	0	0

Table 10: Weight boosts for each system

A Live songs

Songs that were recorded live during a concert, in counterpart to their album version recorded in a studio. The track's liveness attribute was not considered, since, in this case, every live track said so in their name or lyrics.

Query: live

System	Relevant	P@10	Recall	AP	DCG
WF0	NNNNNNRNNN NRRNNRNRNR	0.1	0.166	0.143	0.333
WF1	NNNNRRRRRR RNRRRNNRRR	0.6	0.462	0.436	1.982
WF2	NNNNNNRNNN NRRNNRNRNN	0.1	0.143	0.143	0.333
WF3	NNNNNNRNNN NRRNNRNRNN	0.1	0.143	0.143	0.333

Table 11: Live songs results

B Danceable songs

Hypothesis: Songs that have "dance" in one of their attributes may be danceable. To deem a song as danceable, its danceability value must be greater than 0.7 (there isn't a danceable threshold in Spotify's API reference, 0.7 was picked by trial and error)

Query: dance

System	Relevant	P@10	Recall	AP	DCG
WF0	NNNRNNRNRN RNRNNNRRRN	0.3	0.290	0.608	2.710
WF1	NRNRRRNRRR RRRRNNNNNN	0.7	0.608	0.608	2.710
WF2	NNNNNNNNRN NNNNNNRNNR	0.1	0.333	0.111	0.301
WF3	NRNRRRNRRR RRRRNNNNNN	0.7	0.636	0.608	2.710

Table 12: Danceable songs results

Using sql querries on the original dataset, the percentage of danceable songs in the dataset is 3542 / 12828 * 100% = 27.611 10

Since the best weight system 70% precision at 10, which is close to 3x over-representative of the general population, our hypothesis was verified.

```
SELECT count(*) FROM TRACK WHERE danceability >= 0.7
count(*)
    3542
```

Figure 10: Danceable Track Count

```
SELECT count(*) FROM TRACK
count(*)
   12828
```

Figure 11: Track Count

C Pop songs remixes

Remix versions of pop genre songs.

Query: pop AND remix

System	Relevant	P@10	Recall	AP	DCG
WF0	NNNNNNNRNR NNNNNNNNNR	0.2	0.666	0.162	0.605
WF1	RRNNNNNRRN RRRNNRRRRR	0.4	0.333	0.705	2.247
WF2	NNNNNNNRNN NNNNRNRRNN	0.1	0.25	0.125	0.315
WF3	NNNRNNNNNN RNNNRNRNNN	0.1	0.25	0.25	0.431

Table 13: Pop songs remixes results

D Rock/Metal songs about love

Rock or metal genre songs that are about love.

Query: (rock OR metal) AND love

System	Relevant	P@10	Recall	AP	DCG
WF0	RRNNNNRRNN RRRNRNNNRN	0.4	0.444	0.732	2.280
WF1	RNRNNNNNNN RNNNNNNNNN	0.2	0.666	0.833	1.5
WF2	RRRRRNRRRR RRNRRRRRRN	0.9	0.529	0.948	4.19
WF3	RNNRNRNNNN NNNNNNNNNN	0.3	0.666	0.666	1.787

Table 14: Rock/Metal songs about love results

E Songs about love

Songs about love.

Query: love

System	Relevant	P@10	Recall	AP	DCG
WF0	RRRNRNNNNN RNRNNRRRNN	0.4	0.444	0.95	2.518
WF1	RRRRRRRRRR RRRRRRRRRR	1.0	0.5	1.0	4.5436
WF2	RRNNNNNNNN NNRNNRNRNN	0.2	0.4	1.0	1.631
WF3	RRRRRRRRRR RRRRRRRRRR	1.0	0.5	1.0	4.5436

Table 15: Rock/Metal songs about love results

III System comparison

Overall, the new system performed about as well as the old system. The best weight system (WF1) performed overall equal or worse after the updates. One possible reason for that is that the weights were chosen specifically for the Indexing Process section's evaluation, therefore being less ideal for the new system. Another one is that increasing the size and correcting the synonym list caused the overall results to be less accurate. Same possibility for the phonetic filters. One notable improvement from the new system can be seen when using WF2 in "Rock/Metal songs about love". This comes from the change where genre words weigh less towards non-genre fields (using a weight system). There is a chance that a similar improvement could have been seen, for that query, in the other weight systems if the phonetic filters had been removed. This is because, as previously explained in the Added/Modified Filters section, the low-weighted genre words still generate phonetic versions of themselves when under those filters.

IV Frontend

To allow for ease of use of our system, we developed a GUI application, deploying it as a web page built with Vue, together with the Material Design library, Vuetify.

Search is mainly done by entering the respective terms in the search bar.

This then performs an asynchronous request, using Axios, to Solr that's running in the background and retrieves the relevant results, displaying them in a list.

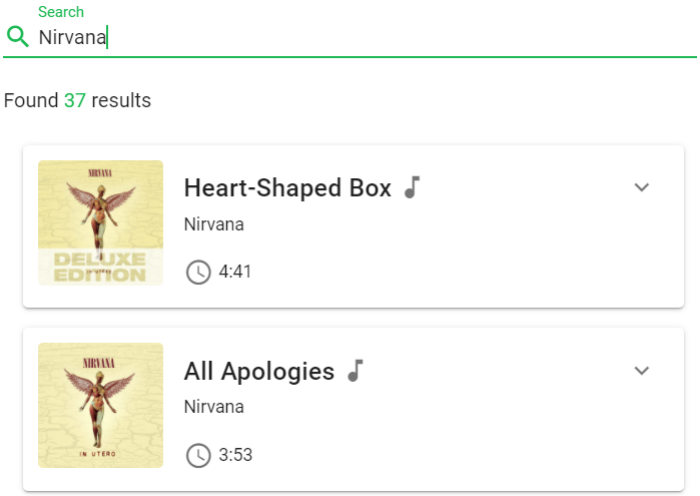

Figure 12: Search results for "Nirvana"

It's possible to search by tracks, albums or artists, and filter explicit results.

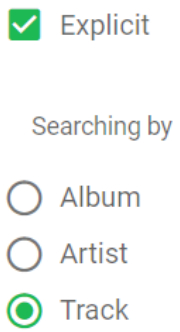

Figure 13: Navigation drawer

Only ten results are retrieved at a given time, however, it's possible to view them all employing pagination. Each time the page is changed, a new request is made to Solr. This is so that the workload by Solr is diminished.

Figure 14: Pagination

V Ontology

An ontology for the search system was developed to analyse the domain knowledge and make domain assumptions explicit. After searching the web for similar existing ontologies, we decided to create our own, specific for our dataset. Using Protégé, we first created the four classes of the domain:

- **Track:** represents a track, as well as its info and features, like track duration and danceability.
- **Album:** represents an album and its info, like the number of tracks and type of album.
- **Artist:** represents an artist and its info, like popularity.
- **Genre:** represents a music genre.

Secondly, three object properties were defined, which connected classes through relations:

- **madeBy:** Track/Album madeBy Artist
- **belongsTo:** Track belongsTo Album
- **does:** Artist does Genre

Then, we created a data property for each class field, where we had to define its domain (the classes it belongs to) and its range (the data type). For example:

- **name:** Domain = {Track, Album, Artist, Genre} Range = String
- **explicit:** Domain = {Track, Album, Artist} Range = Boolean
- **duration_ms** Domain = {Track} Range = Integer

By doing so, we were able to generate a graph to represent the created ontology **??**. Now we could start populating the ontology with individuals from our dataset. Instead of adding individuals one by one, we decided to use the built-in Cellfie plugin, which imports individuals from an excel spreadsheet. Since the dataset is in JSON format, we had to convert it to .xlxs (excel) using an online converter tool. Because the original file was too large, only a portion was converted and used to populate the ontology. For each class, a transformation rule was created, where we had to provide information for the **Individual**, the **Type** and the **Facts**. The Individual references a column from the spreadsheet, which associates a name for each individual in that column. The Type asserts which class is being referenced. The Facts stipulates which data and object properties the Individual should accept, by referencing and labelling a column from the spreadsheet. An example for importing tracks can be seen in figure 15 Once the population is completed, we can query the ontology. Using the built-in SPARQL, we can define prefixes as URIs, which we can use to reference classes, properties, types, etc., without any ambiguity. For example, in figure 16 we can see a query to find all albums made by artist Usher, followed by its result in figure 17.

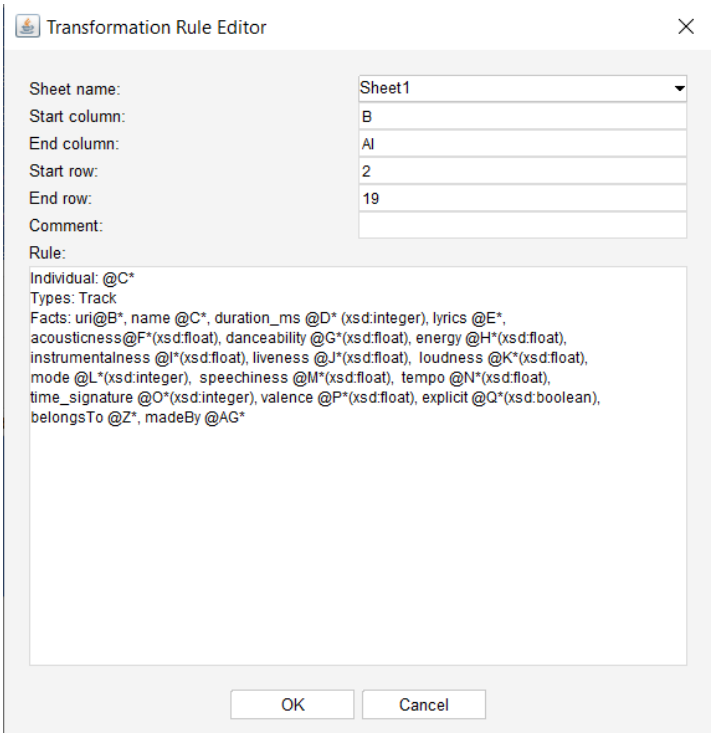

Figure 15: Transformation Rule for 'Track'

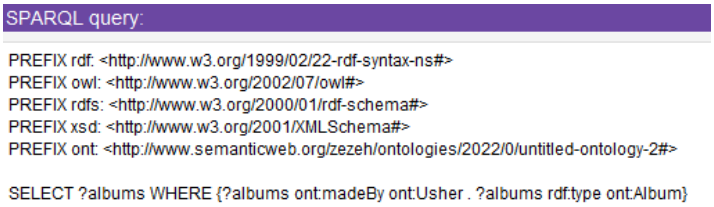

Figure 16: SPARQL query for Usher Albums

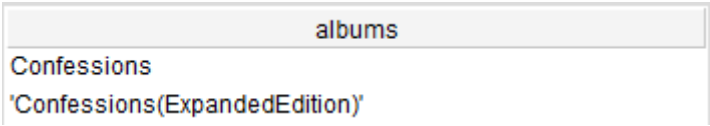

Figure 17: SPARQL result for the query

VI Conclusion

All of the goals for the project were accomplished.

In terms of Dataset Preparation, although there were some difficulties in reconciling the data together to enrich the dataset those were successfully overcome. Therefore, the result is a large dataset with complete and coherent data.

As for Information Retrieval, despite still being somewhat unrefined, the search engine developed works properly and significantly better than exclusively using *SELECT* queries in the SQL database. One thing to note is that the first result of each query in this report tends to not be relevant, which reduced the average precision.

Finally, as for the Search System, a frontend and an ontology were created as planned, as well as more filters for the information retrieval process, which was also revised.

As future work, regarding the frontend, there could be more refined filtering and sorting options for the search results, more information could be displayed and auto-complete, spell-checking and faceting features could be implemented. In addition, removing the phonetic filters from the new system and testing again could prove helpful, as we think it would likely increase the overall precision. Finally, the term weight system (from the Delimited Boost Filter) could also be used to reduce the weight of synonyms compared to the original word, as a lot of the rejected entries had synonyms, especially for the "love" query.

References

[1] Hassan, Charlotte, "68% of Smartphone Owners Stream Music Daily, Study Finds", *Digital Music News* https://www.digitalmusicnews.com/2016/03/11/parks-associates-68-of-u-s-smartphone-owners-listen-to-streaming-music-daily/ Last accessed: 12th December 2021. Published by Charlotte Hassan, March 11th, 2016.

[2] "Spotify Million Playlist Dataset Challenge", *AIcrowd*, https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge Last accessed: 12th December 2021. Published by Spotify.

[3] *Banned Word List*, http://www.bannedwordlist.com/lists/swearWords.txt

[4] *Field Type Definitions and Properties*, - Apache Solr Reference Guide (for solr 8.11) https://solr.apache.org/guide/8_11/field-type-definitions-and-properties.html#field-default-properties Last accessed: 12th December, 2021.

Annex

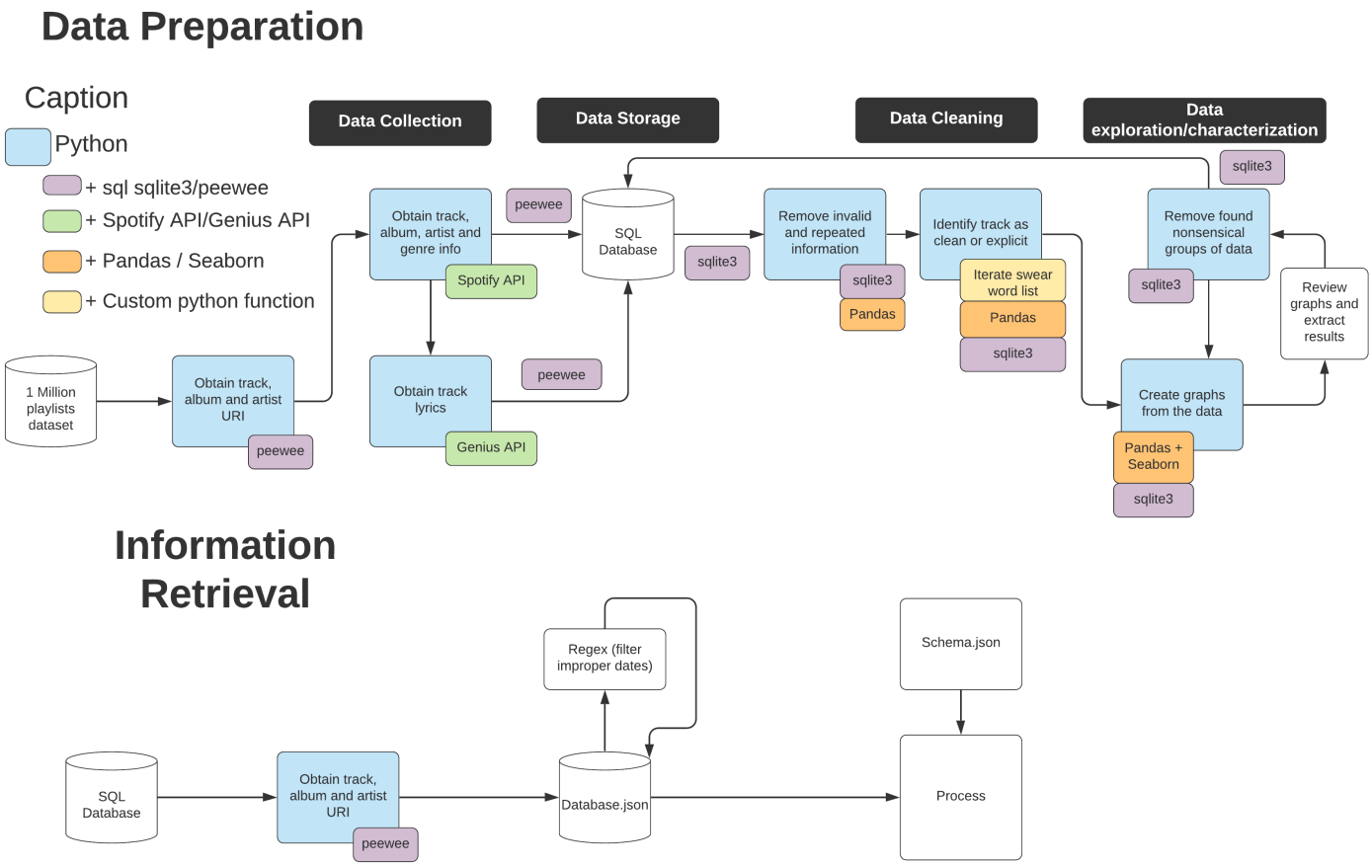

Figure 18: Workflow pipeline

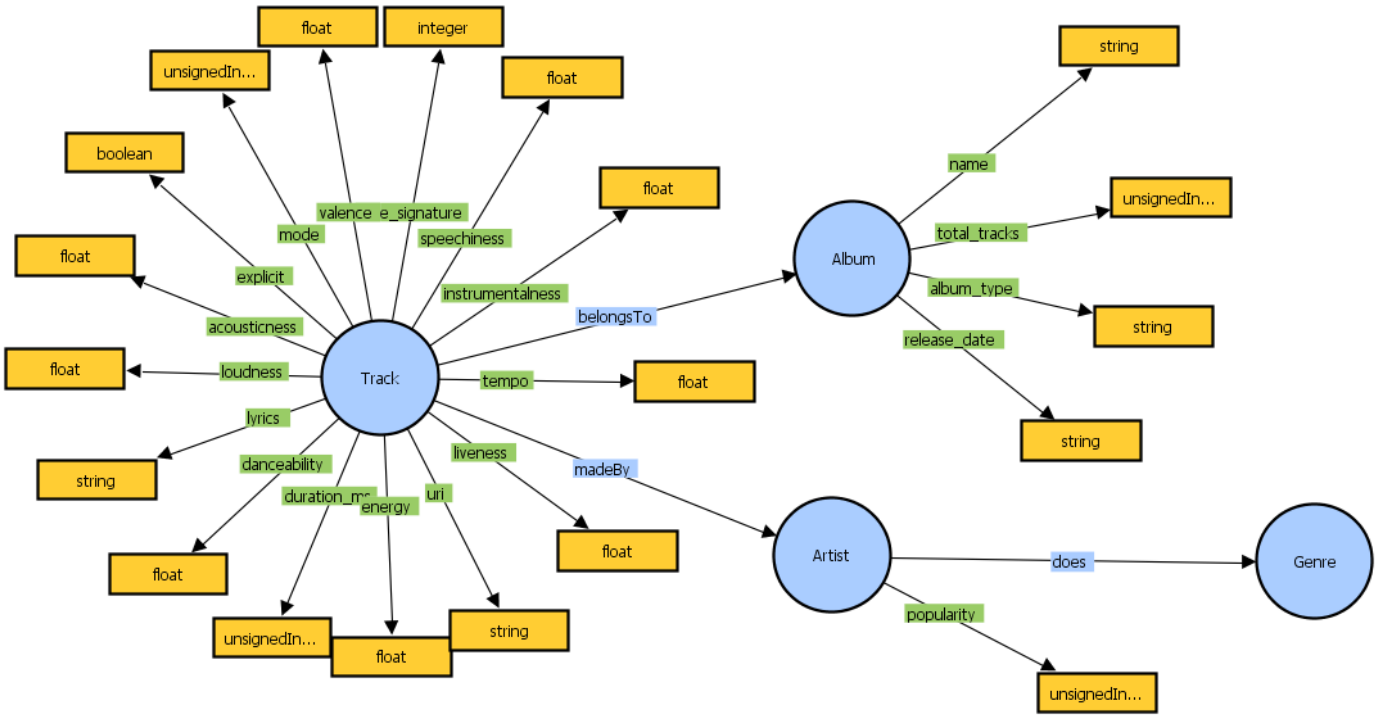

Figure 19: Ontology Graph

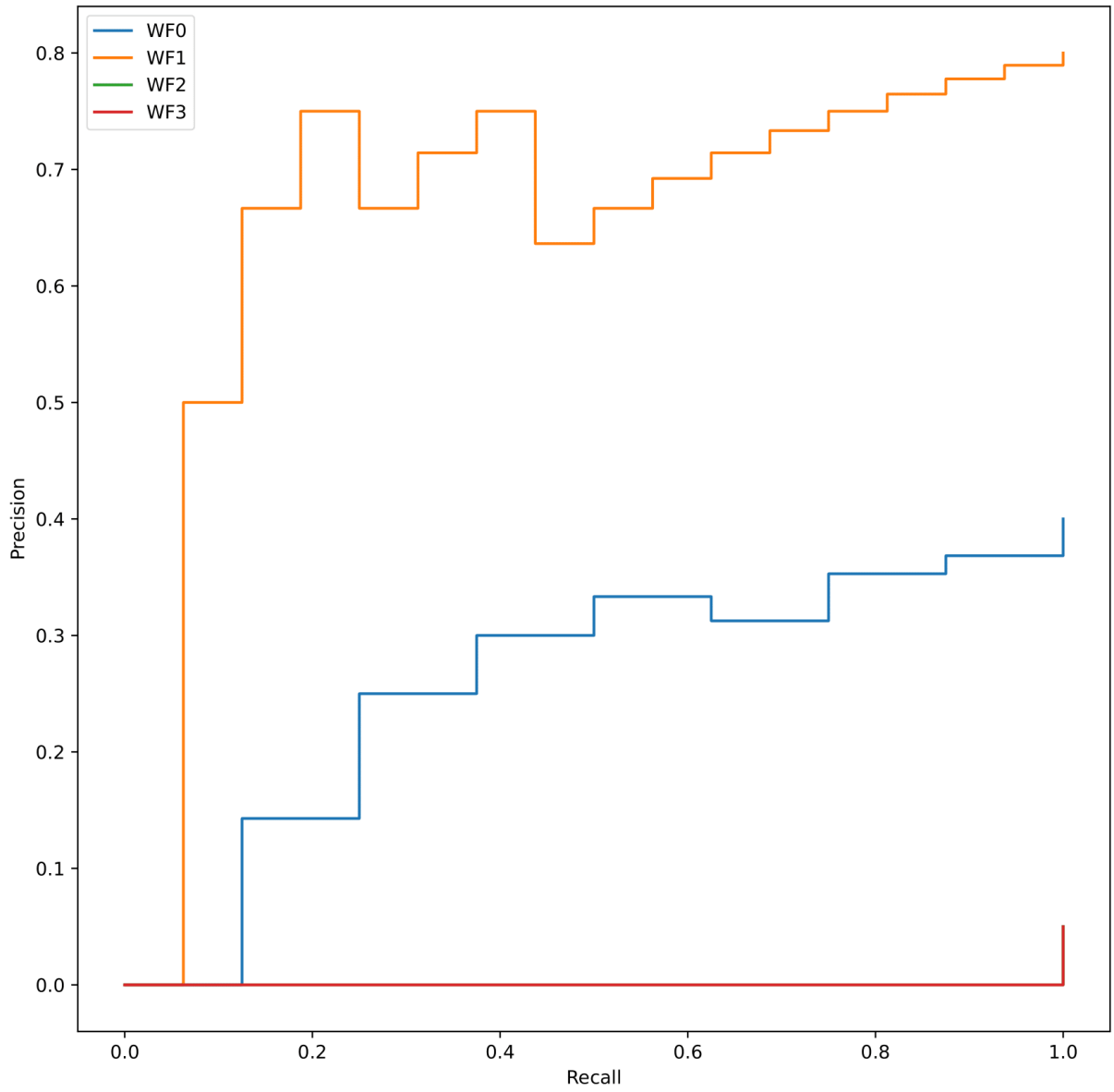

Figure 20: Precision-recall graph checkpoint 2 "live". WF3 overlaps WF2

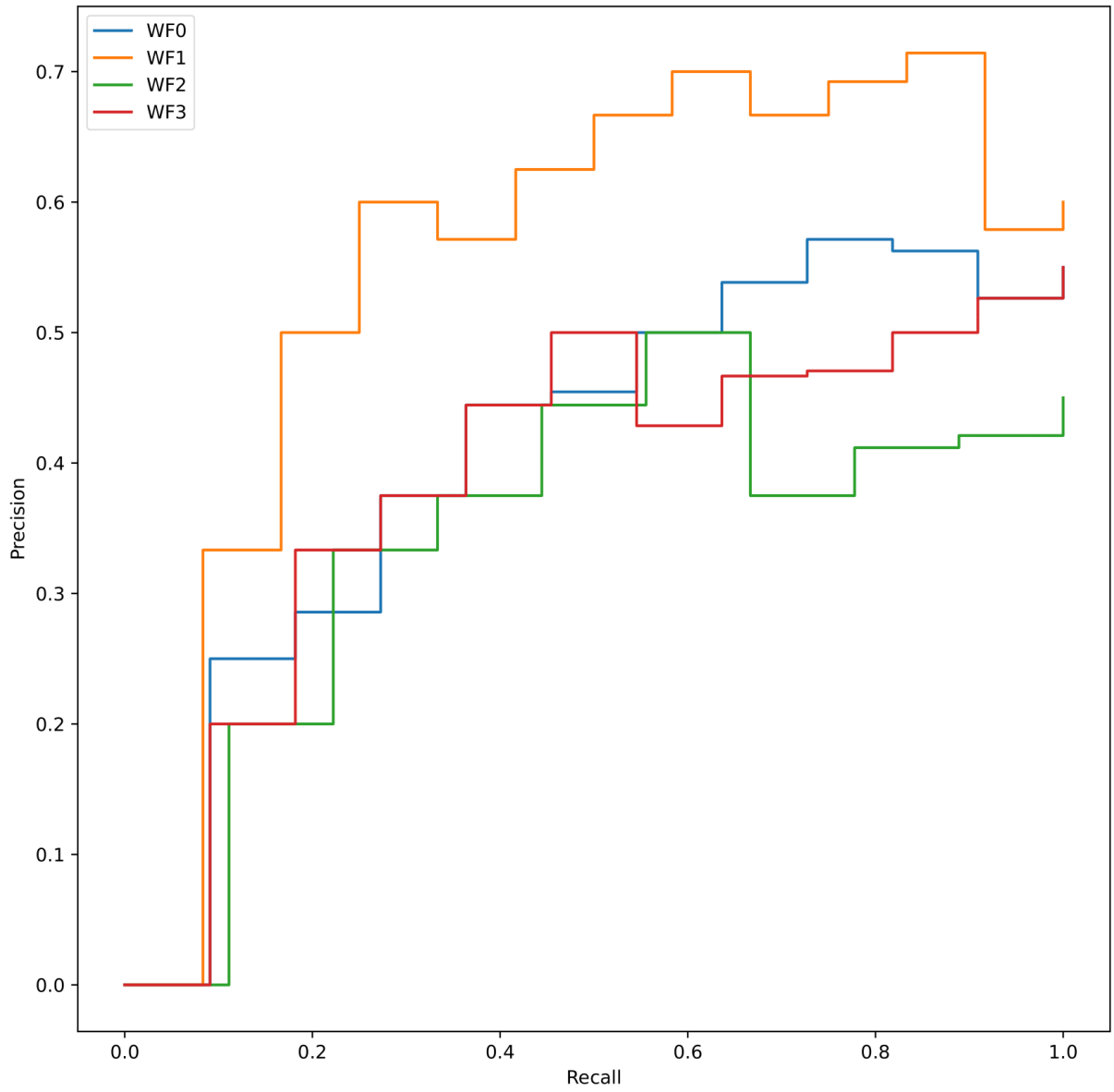

Figure 21: Precision-recall graph checkpoint 2 "dance"

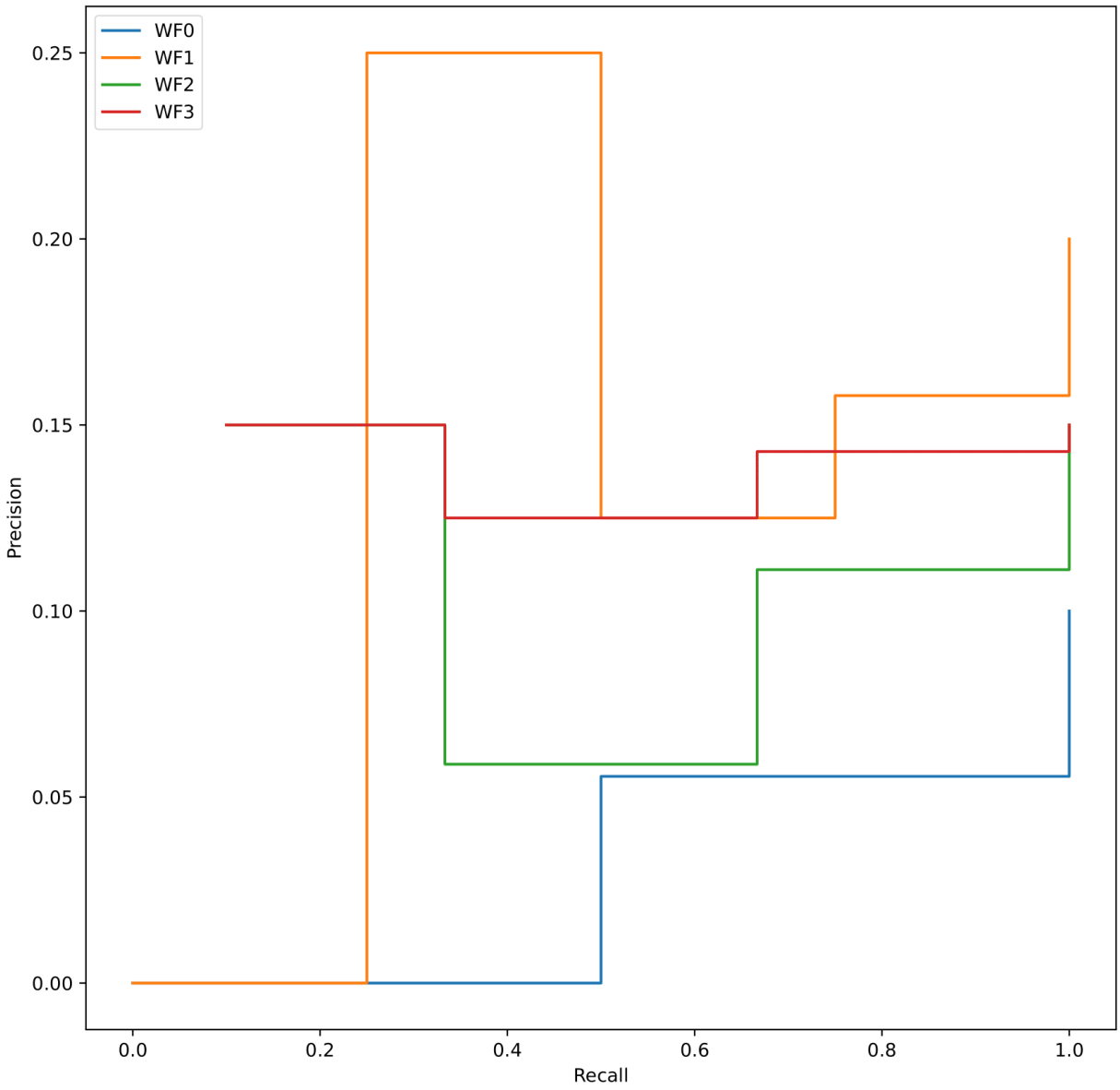

Figure 22: Precision-recall graph checkpoint 2 "pop AND remix"

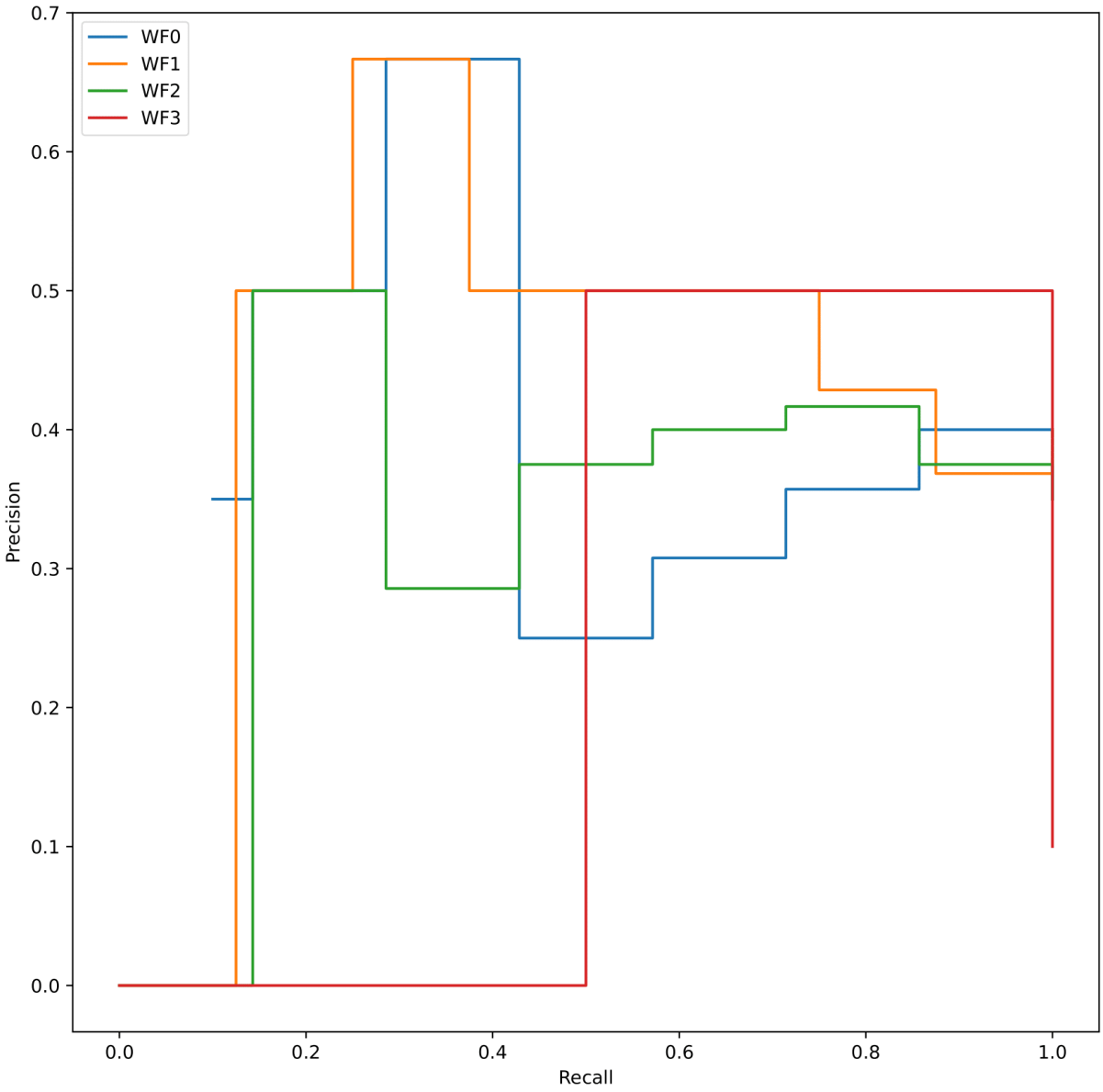

Figure 23: Precision-recall graph checkpoint 2 "(rock OR metal) AND love"

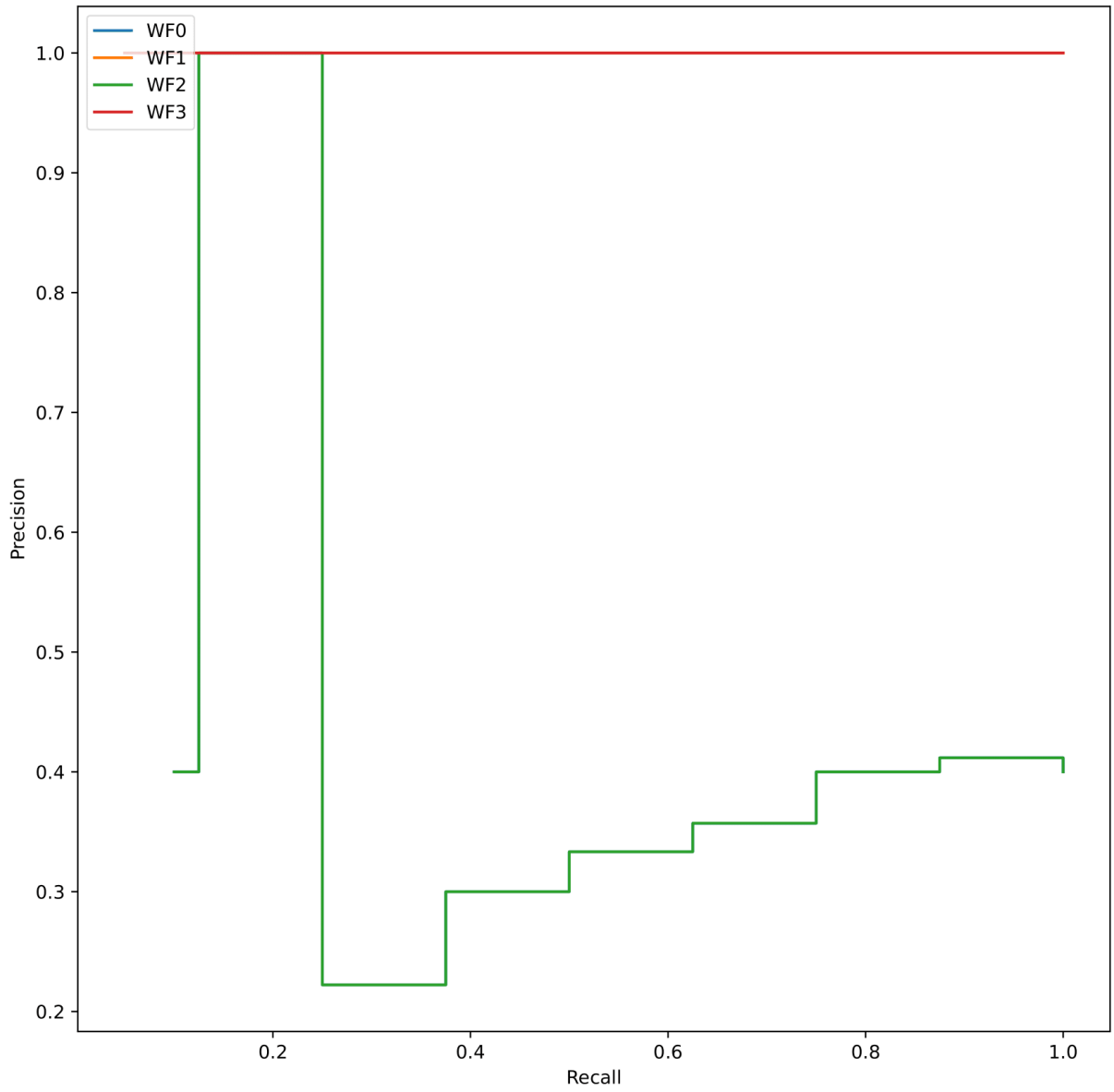

Figure 24: Precision-recall graph checkpoint 2 "loveW. WF3 overlaps WF1; WF2 overlaps WF0

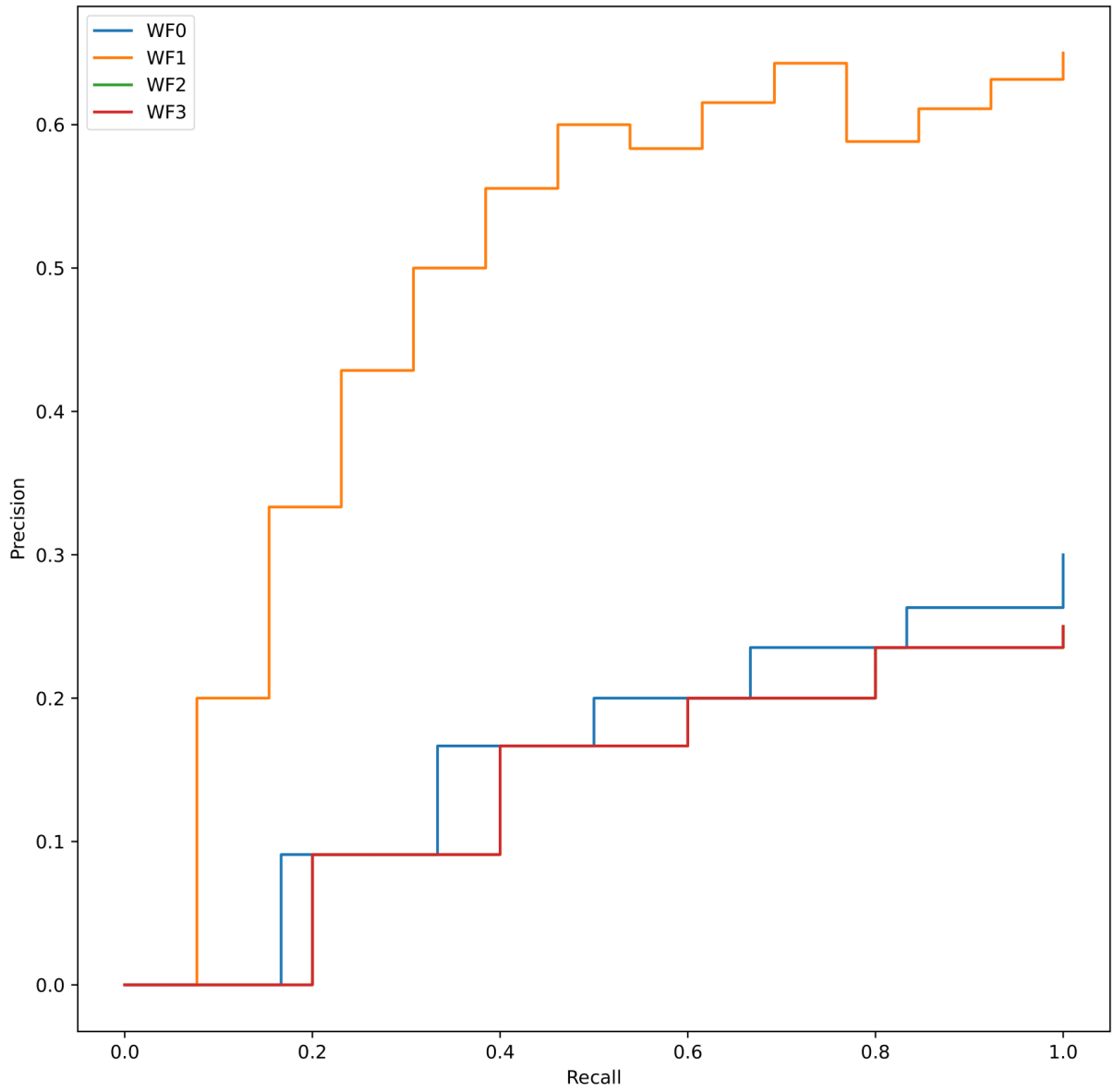

Figure 25: Precision-recall graph new "live". WF3 overlaps WF2

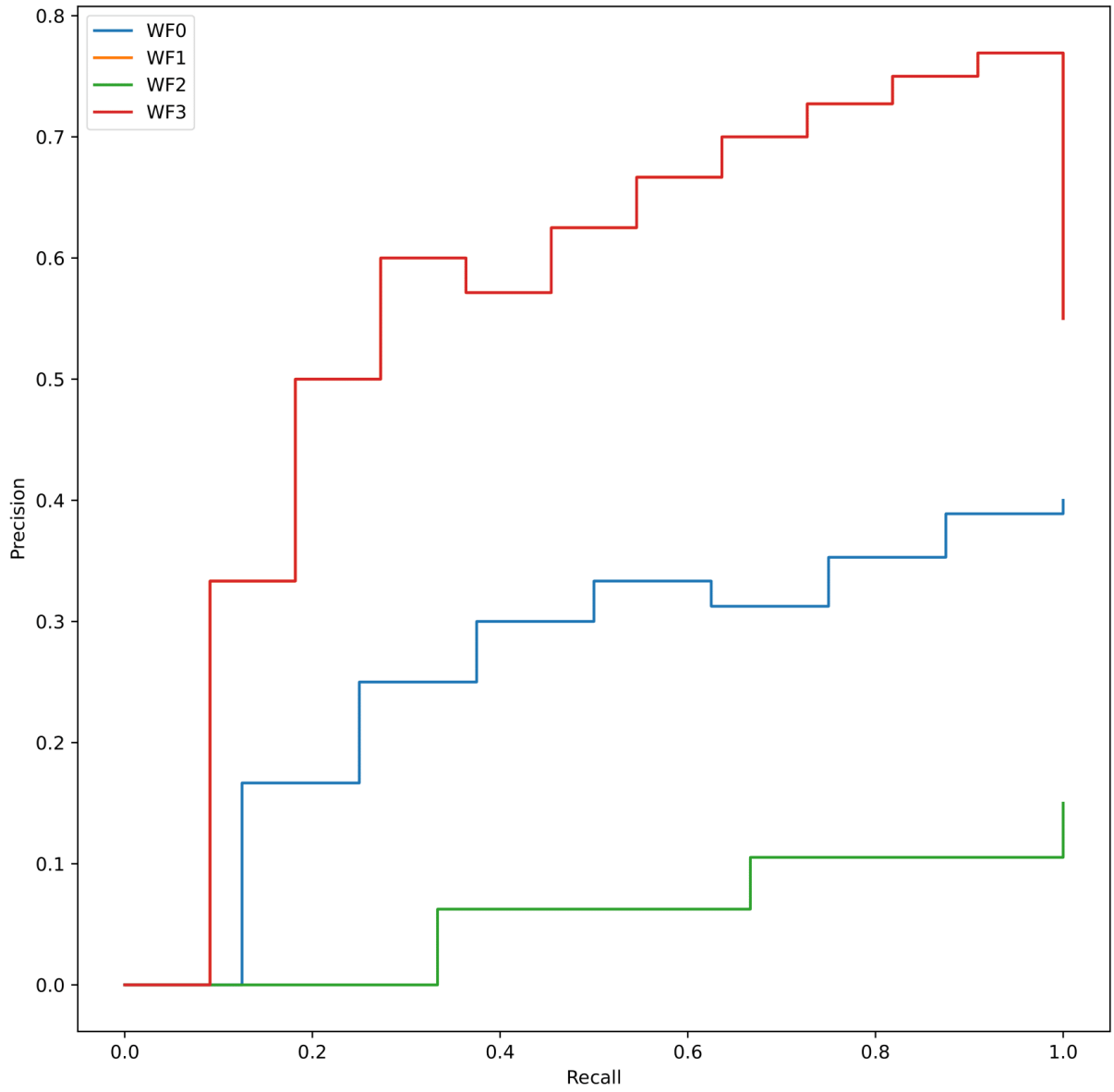

Figure 26: Precision-recall graph new "dance". WF3 overlaps WF1

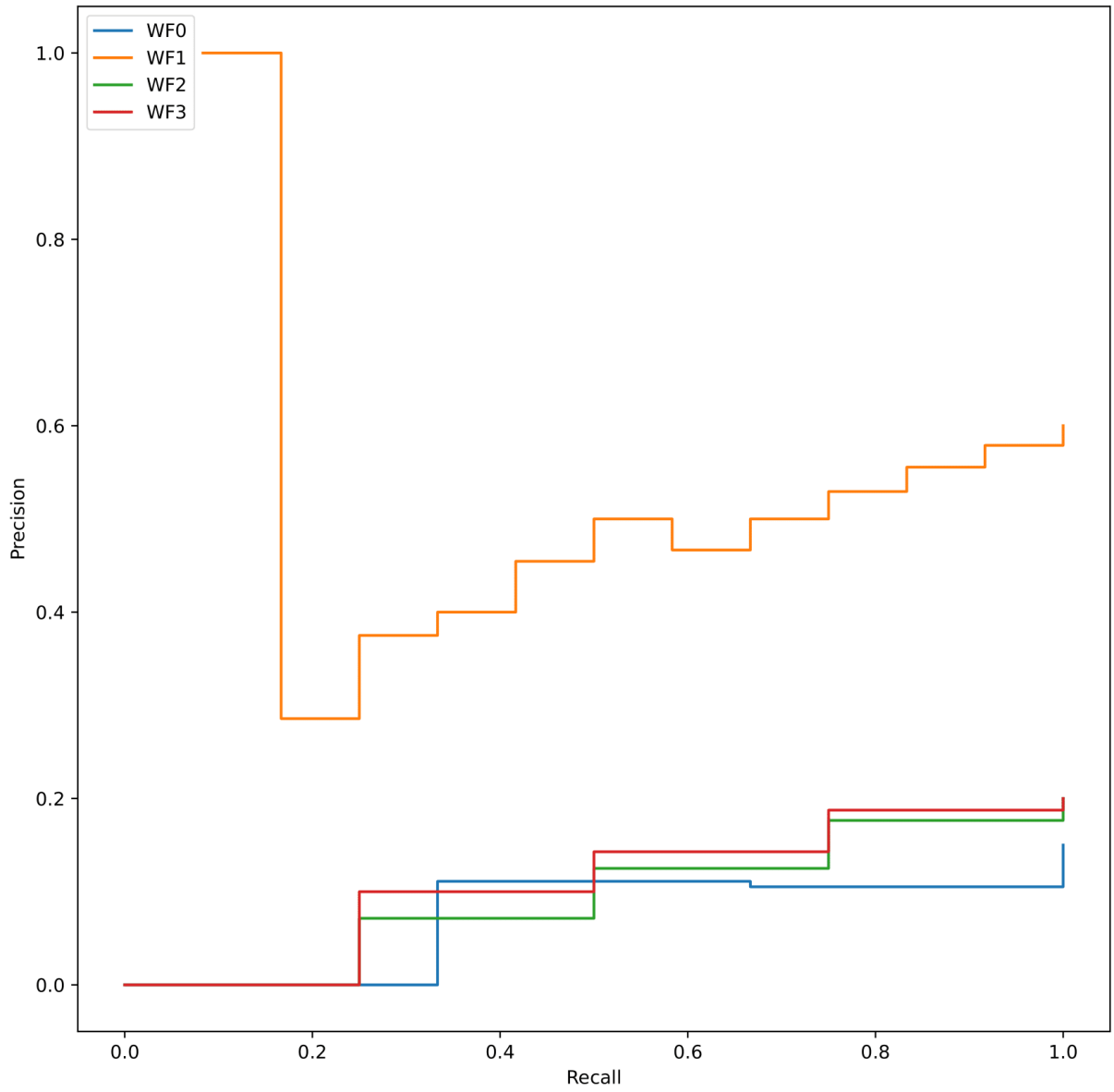

Figure 27: Precision-recall graph new "pop AND remix"

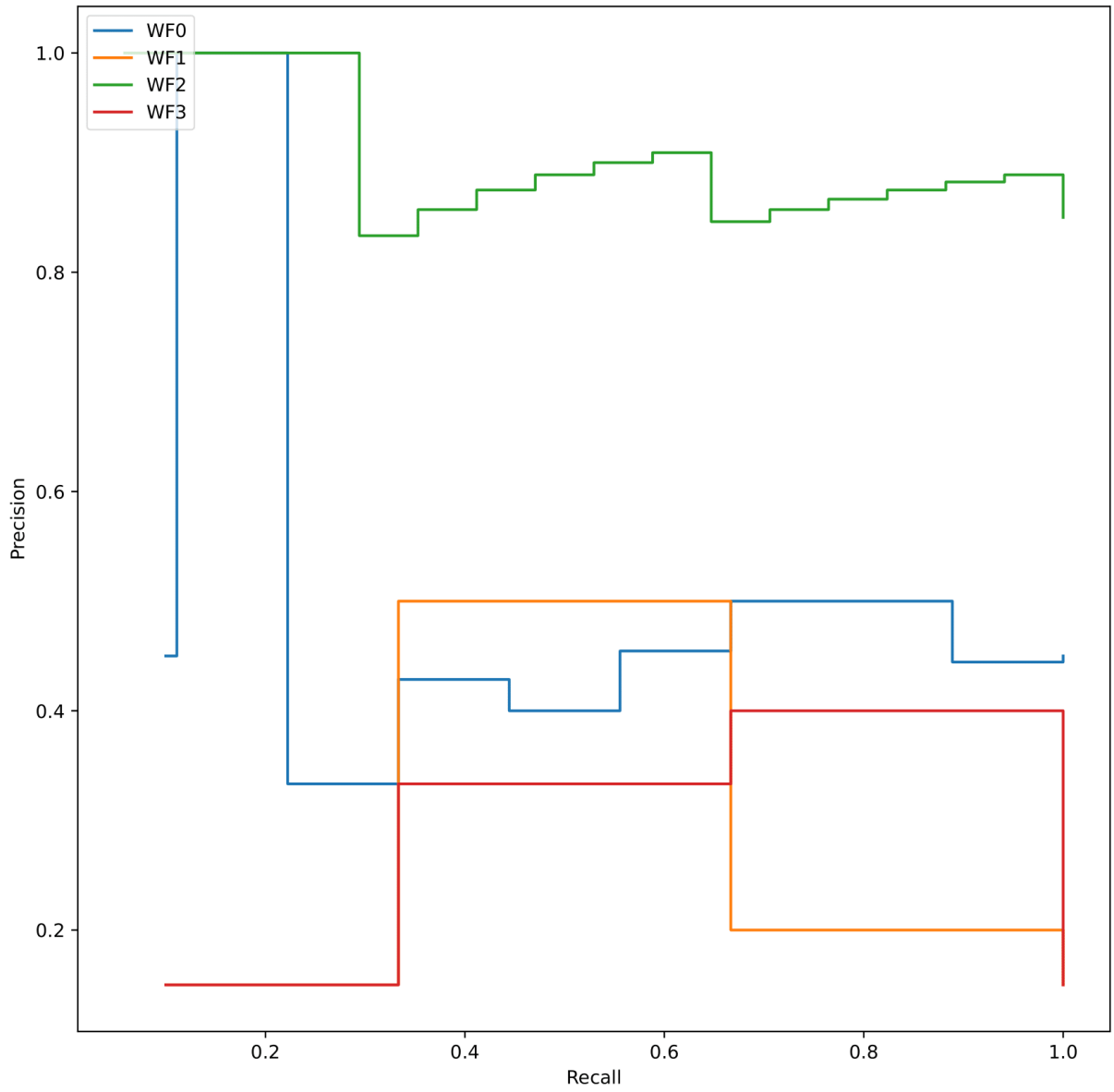

Figure 28: Precision-recall graph new "(rock OR metal) AND love"

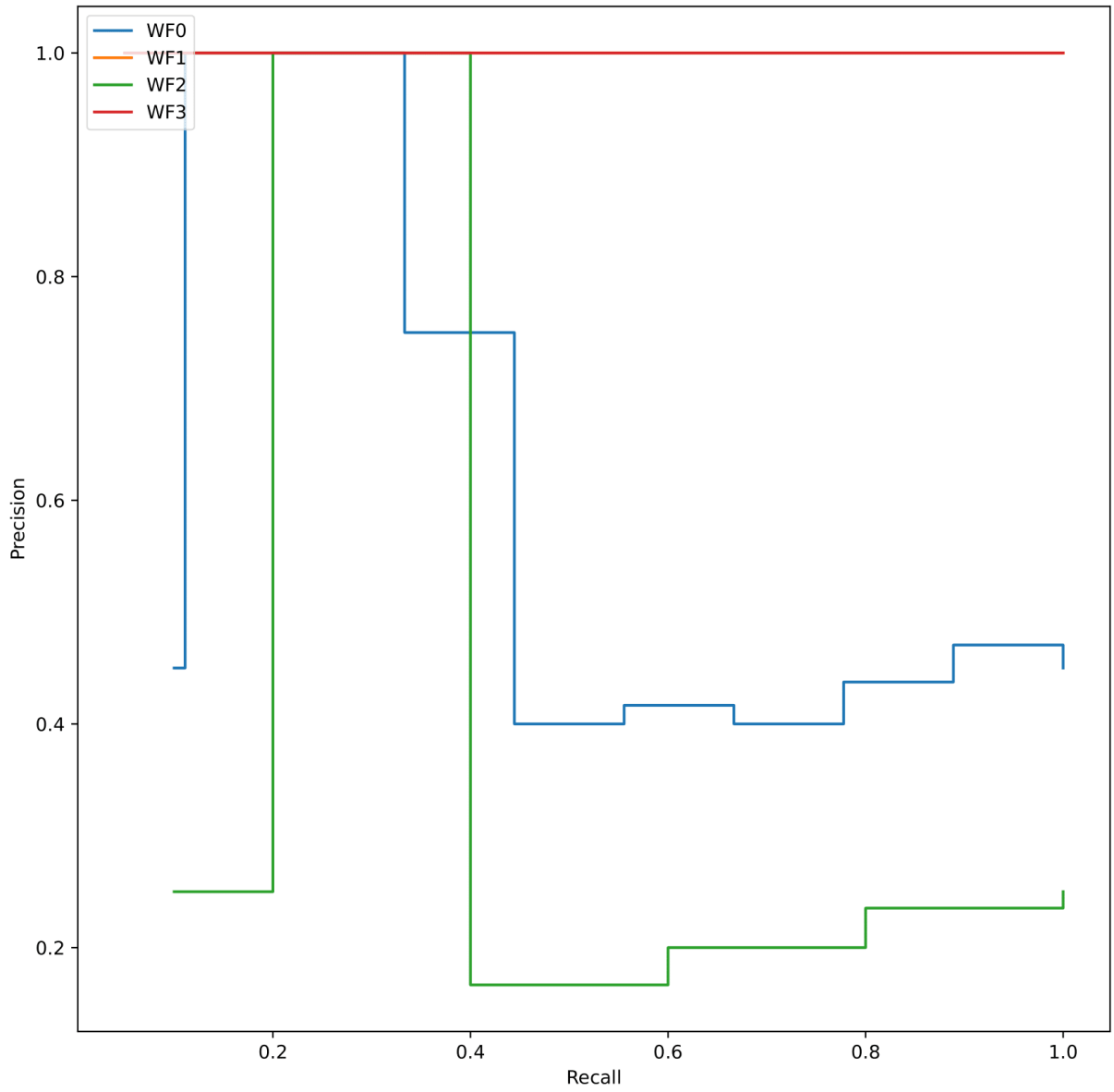

Figure 29: Precision-recall graph new "love". WF3 overlaps WF1