Data Analytics

Diggerz

AI music recommender app

Olivier Babiard

Table of Contents

1. Introduction

1.1 Project Overview

Diggerz is an innovative music discovery platform specifically designed for DJs. The project addresses one of the most challenging aspects of DJing: the time-consuming process of discovering and curating music for sets. By combining data from Discogs, the world's largest music database, with Spotify's advanced audio features, Diggerz creates an intelligent system that streamlines the music discovery process for DJs.

1.2 Business Case

Music digging and record collecting form the foundation of DJing, requiring extensive knowledge of artists, labels, genres, and music affinity. However, with over 17.8 million releases on Discogs (including around 5 million electronic music releases), the task of discovering new music has become increasingly complex. DJs face several challenges:

- Managing vast amounts of musical information across multiple platforms
- Time-consuming process of manually searching and evaluating tracks
- Need for precise musical matching for seamless DJ sets
- Difficulty in discovering similar tracks across different eras and formats

1.3 Technical Solution

Diggerz addresses these challenges through:

- Intelligent Music Discovery
- Automated recommendation system based on track analysis
- Integration with Discogs API for comprehensive metadata
- Leveraging Spotify's Echo Nest technology for audio feature analysis
- Smart matching algorithms based on musical characteristics

1.4 Audio Feature Analysis

EchoNest is a music intelligence company that was bought by Spotify in 2014. It features a range of quantifiable musical metrics, now used in the Spotify API.

These key musical elements include:

- Acousticness (production style analysis)
- Danceability (rhythm stability, beat strength)
- Energy (perceptual intensity, dynamic range)
- Tempo (BPM analysis for mix compatibility)
- Key (pitch matching for harmonic mixing)
- Additional features like valence, instrumentalness, and liveness

1.5 Scope and Functionality

The App structure is as follows :

1. Processing Discogs URLs to identify input tracks (using the release ID embedded in the URL) ;
2. Analyzing track characteristics using Spotify's audio features, through an API call ;
3. Generating five relevant track recommendations, using a weighted average score for matching with the input album ;
4. Providing detailed audio feature analysis for comparison ;
5. Offering an intuitive interface for music discovery through a Streamlit web-interface.

2. Data Sources & Data Collection

I mainly used 4 different sources for data analysis, and music matching :

2.1 Kaggle Dataset

My main source for music matching was retrieved on Kaggle : I found a comprehensive [10M+ tracks database](), merging data from Beatport (one of the world leaders of the electronic music digital store) and Spotify (4.7M tracks with audio features).

I ended up using only the Spotify data, as the sheer size of the database (7.4Gb unmerged, more than 25Gb merged) made some queries quite difficult and long to execute.

I used the 5 Spotify CSVs and the EchoNest audio features CSV, and merged them together :

```
 0   track_id          object
 1   track_title       object
 2   duration_ms       int64
 3   isrc              object
 4   release_id        object
 5   preview_url       object
 6   release_title     object
 7   release_date      object
 8   upc               float64
 9   popularity        int64
 10  total_tracks      int64
 11  album_type        object
 12  release_img       object
 13  label_name        object
 14  artist_id         object
 15  artist_name       object
 16  acousticness      float64
 17  danceability      float64
 18  energy            float64
 19  instrumentalness  float64
 20  key               int64
 21  liveness          float64
 22  loudness          float64
 23  mode              int64
 24  speechiness       float64
 25  tempo             int64
 26  time_signature    int64
 27  valence           float64
dtypes: float64(9), int64(7), object(12)
memory usage: 1.6+ GB
```

2.2 Discogs API Integration

I decided to use the Discogs API to retrieve the metadata about the “input album”. Discogs is the platform of choice for any DJ or music enthusiast, with the most comprehensive database in the world, the presence of a marketplace for physical media (mainly vinyls, CDs, cassettes) and the ability to “save” your collection with value statistics.

We can connect to the API using an account on the Discogs Platform :

```
d = discogs_client.Client(
    'my_user_agent/1.0',
    consumer_key='hZZUdNwRHsUxlgReVdCA',
    consumer_secret='TUAxCaABSkDcmQgeRhIbRRvnHAopOIkH',
    token=u'FbbkQDyGoGsJlnSqVfFwqfvUWnrtDcBiWmyHOHjX',
    secret=u'my_token_secret'
)
```

I created an App within Discogs Account settings, to get a consumer_key, a consumer_secret and a token for API calls.

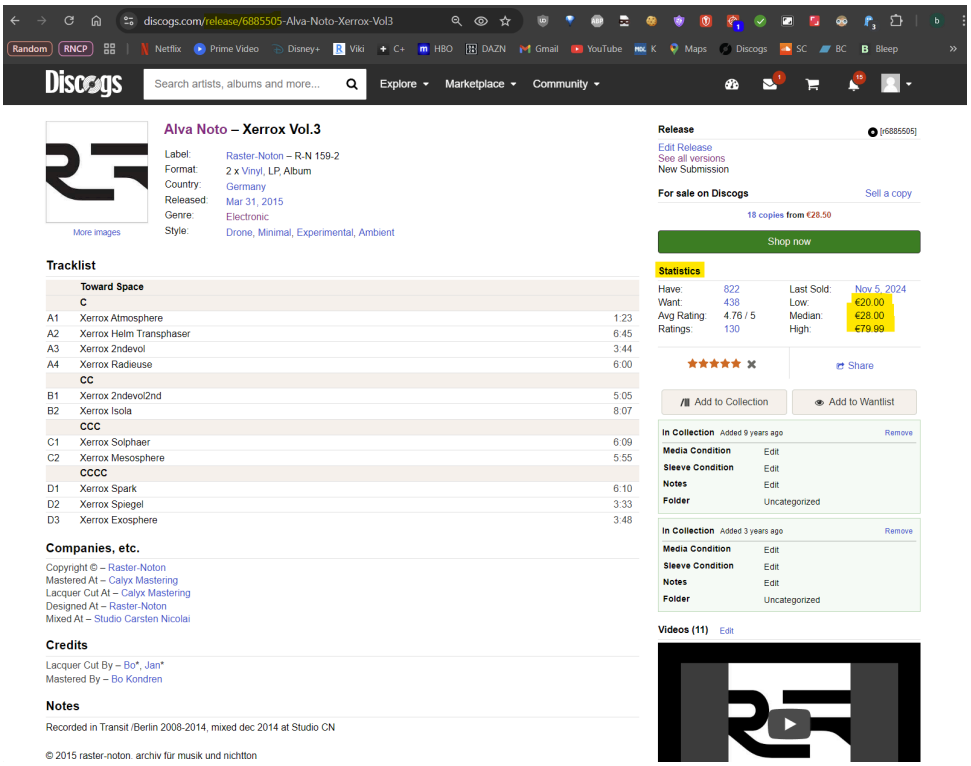
We can then retrieve the “input album” metadata, using a dedicated function:

```
def get_discogs_info(self, url: str) -> Dict:
    """Get detailed release information from Discogs."""
    release_type, release_id = self.parse_discogs_url(url)

    try:
        if release_type == 'master':
            master = self.discogs.master(release_id)
            release = master.main_release
        else:
            release = self.discogs.release(release_id)

        # Extract basic information
        info = {
            'artist': release.artists[0].name if release.artists else "Unknown Artist",
            'album': release.title,
            'tracks': [track.title for track in release.tracklist
                       if track.title and isinstance(track.title, str)],
            'genres': release.genres if hasattr(release, 'genres') else [],
            'styles': release.styles if hasattr(release, 'styles') else [],
            'year': release.year if hasattr(release, 'year') else None,
            'label': release.labels[0].name if release.labels else "Unknown Label",
            'catalog': release.labels[0].catno if release.labels and hasattr(release.labels[0], 'catno') else "Unknown",
            'format': release.formats[0]['name'] if release.formats else "Unknown Format"
        }
```

2.3 Web Scraping Implementation (Selenium)

I decided to scrape the Discogs release page for relevant price statistics.

Here's an example of a Discogs Release page :



I used Selenium rather than BeautifulSoup to scrape the HTML, as the website is Javascript heavy, and wouldn't work with BeautifulSoup.

```
Tabnine | Edit | Test | Explain | Document | Ask
def _scrape_price_stats(self, url: str) -> Dict:
    """Scrape price statistics from Discogs webpage using Selenium."""
    driver = None
    try:
        # Initialize WebDriver for this session
        driver = webdriver.Chrome(service=self.driver_service, options=self.chrome_options)

        # Load the Discogs release page
        print(f"Accessing URL: {url}")
        driver.get(url)

        # Wait for the "release-stats" section to load
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.ID, "release-stats"))
        )

        # Locate the price statistics section
        price_section = driver.find_element(By.ID, "release-stats")
        print("Found release-stats section")

        # Initialize a dictionary to store price data
        price_info = {'low': None, 'median': None, 'high': None}

        # Locate all <li> elements in the price section containing price stats
        li_elements = price_section.find_elements(By.TAG_NAME, "li")
        print(f"Found {len(li_elements)} price elements")

        for li in li_elements:
            # Each <li> should have two spans: one for the label, one for the value
            spans = li.find_elements(By.TAG_NAME, "span")
            if len(spans) == 2:
                label = spans[0].text.strip().lower()
                value = spans[1].text.strip()
                print(f"Found price info - Label: {label}, Value: {value}")
```

2.4 Spotify API Integration

Finally, I used Spotify API to match the Discogs release & tracks, to fetch its audio features before running the matchmaking engine on the tracks dataframe i extracted from the CSV files.

```
# Spotify & Discogs Client Credentials
CLIENT_ID = "26c65df3e5844f1dbe355d82d80c9f6f"
CLIENT_SECRET = "2d4d2b147bc942b999564a5e8649b987"
DISCOGS_TOKEN = "FbbkQDyGoGsJlnSqVfFwqfvUWnrtDcBiWmyHOHjX"

#Initialize SpotiPy with user credentials
sp = spotipy.Spotify(auth_manager=SpotifyClientCredentials(client_id=CLIENT_ID,
                                                           client_secret=CLIENT_SECRET))
```

We can then execute the get_spotify_features() function to search for the Discogs album on Spotify, and retrieve its audio features:

```
def get_spotify_features(self, artist: str, album: str) -> Optional[Dict]:
    """Get Spotify audio features and metadata for an album."""
    try:
        # Search for album
        query = f"album:{album} artist:{artist}"
        results = self.spotify.search(q=query, type='album', limit=1)

        # If no results, try searching only by album name
        if not results['albums']['items']:
            print("Initial search failed, trying without artist...")
            query = f"album:{album}"
            results = self.spotify.search(q=query, type='album', limit=1)
            if not results['albums']['items']:
                return None  # Album not found

        album_id = results['albums']['items'][0]['id']
        album_info = self.spotify.album(album_id)

        # Get tracks and their audio features
        tracks = self.spotify.album_tracks(album_id)['items']
        track_ids = [track['id'] for track in tracks]
        audio_features = self.spotify.audio_features(track_ids)

        # Combine track info with audio features
        tracks_with_features = []
        for track, features in zip(tracks, audio_features):
            if features:
                track_info = {
                    'name': track['name'],
                    'preview_url': track['preview_url'],
                    'duration_ms': track['duration_ms'],
                    **{k: features[k] for k in self.all_features}
                }
                tracks_with_features.append(track_info)
```

3. Exploratory Data Analysis & Data Cleaning

3.1 Data Cleaning process

Fortunately, data cleaning remained minimal due to the very clean CSV fetched from Kaggle. I ran various cleaning steps, such as dropping unused columns (like image URLs columns), checked for NA values, and dropped rows without vital values (track_id, artist_id, audio_features).

3.2 EDA Findings

As described earlier, I worked with 5 Spotify CSVs (sp_artist_release, sp_artist_track, sp_artist, sp_release, sp_track) and the EchoNest audio features CSV. I then merged all the data after cleaning some unused columns and removed rows with vital information missing.

Merged CSV was named spotify_complete_data and resulted in a dataframe with 7.523.890 rows (representing unique tracks, duplicates were removed) and 28 columns :

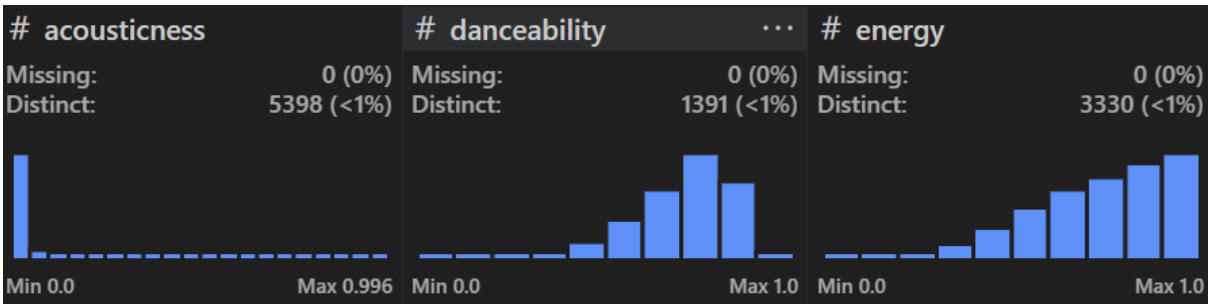
```
print("Dataset Shape:", df.shape)
print("\nDataset Info:")
df.info()
```

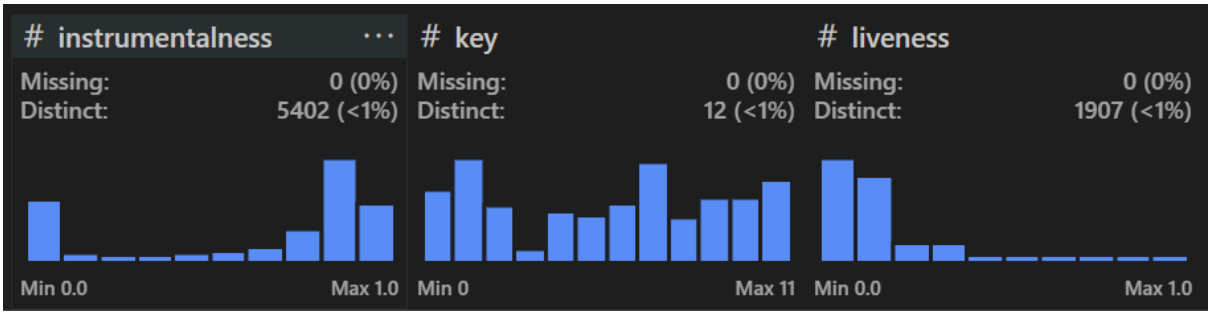
```
Dataset Shape: (7523890, 28)
```

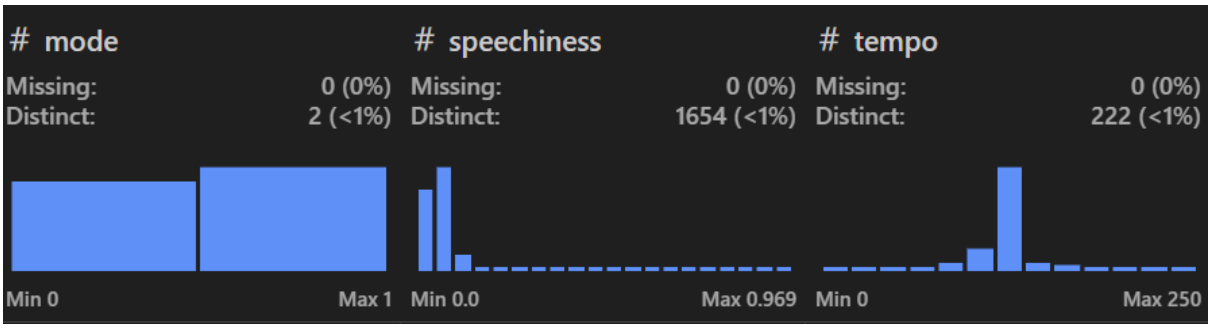
The only few numerical columns were audio_features and track_duration (in ms). All the other columns were of type Object (strings).

Here's an overview of some important columns:

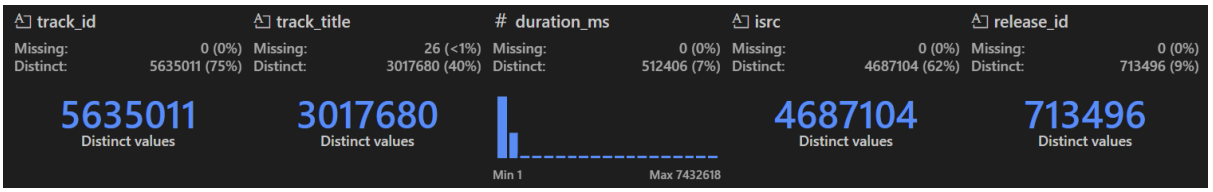
Audio Features



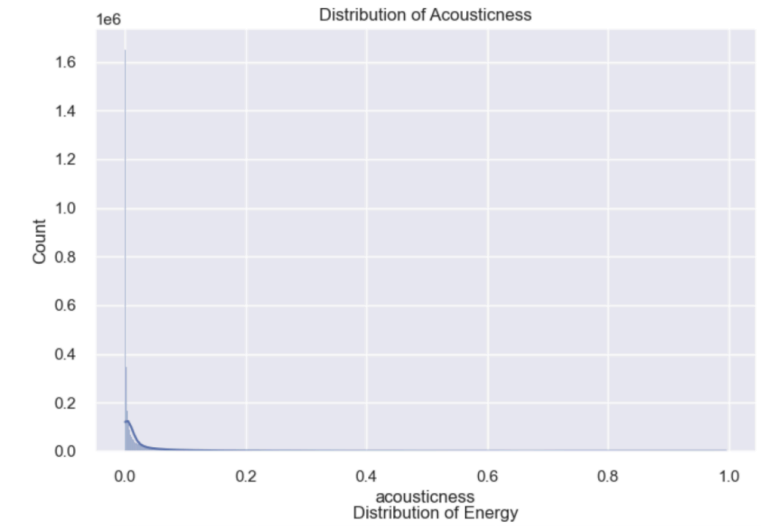


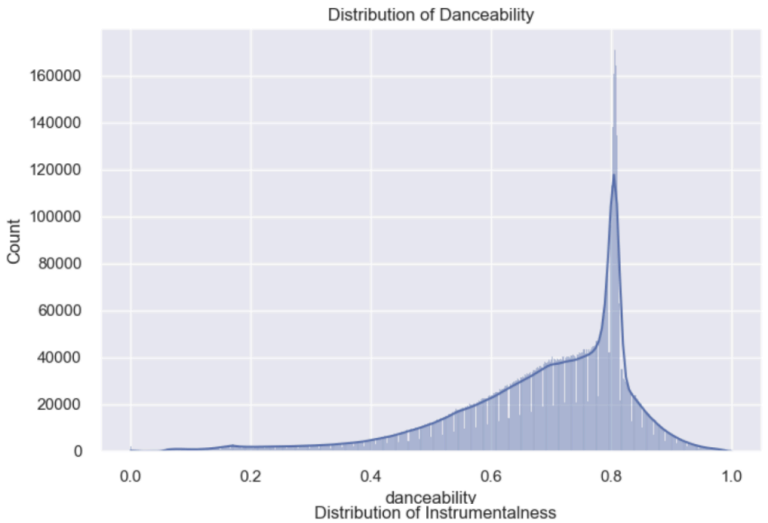


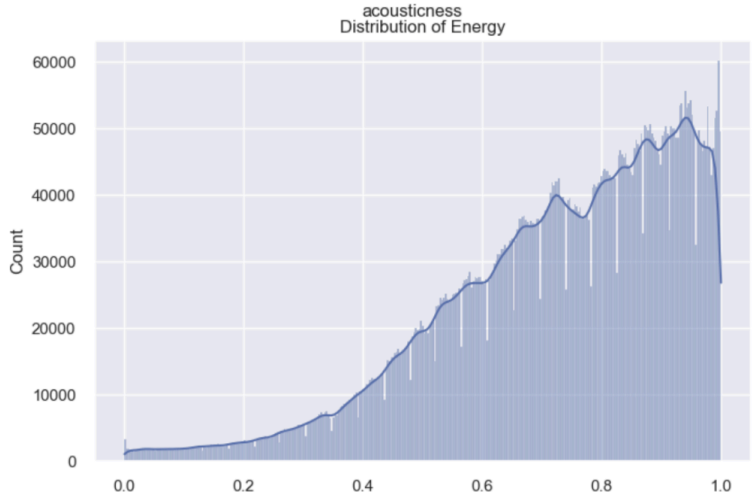
Track information

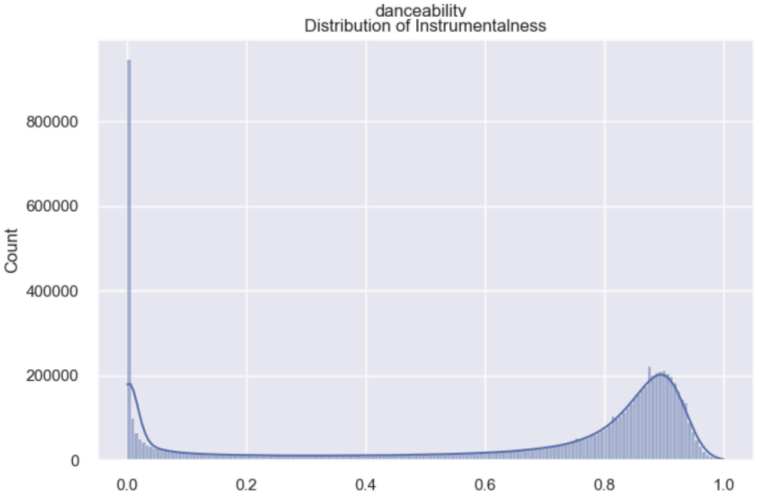


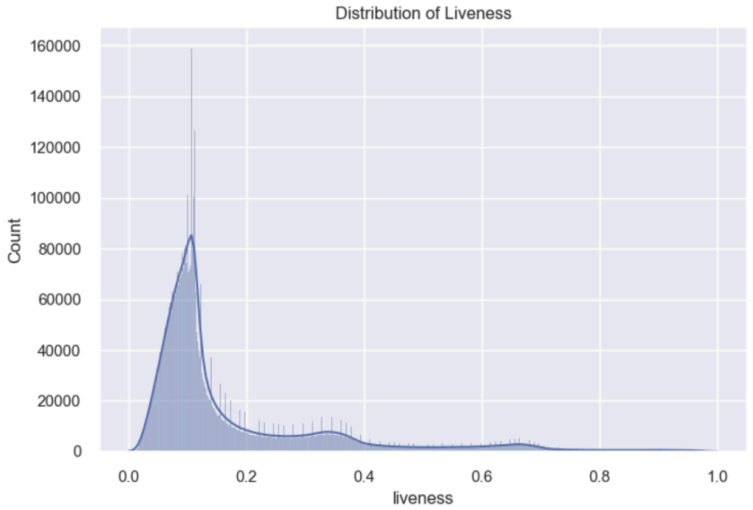
Audio Features

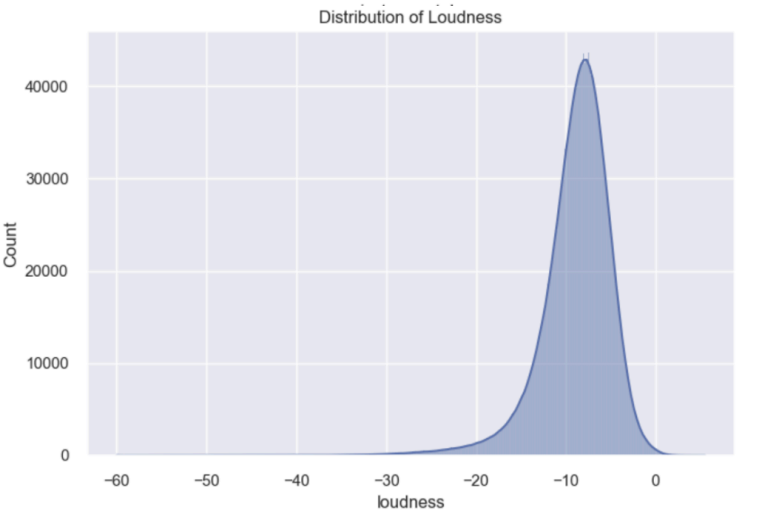


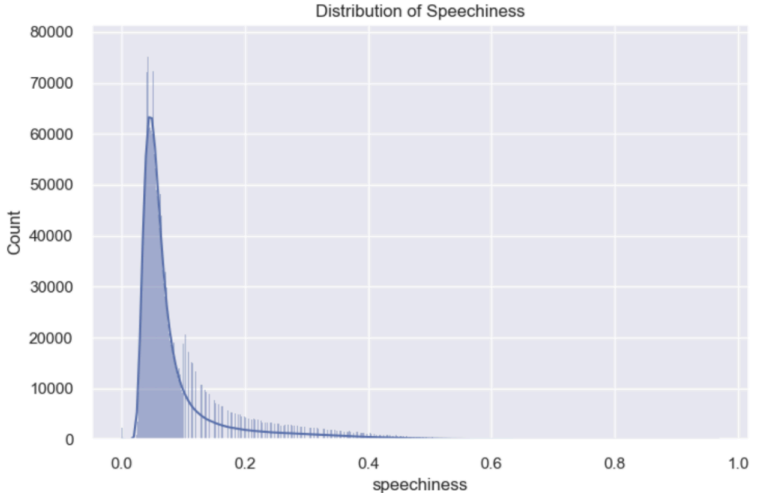


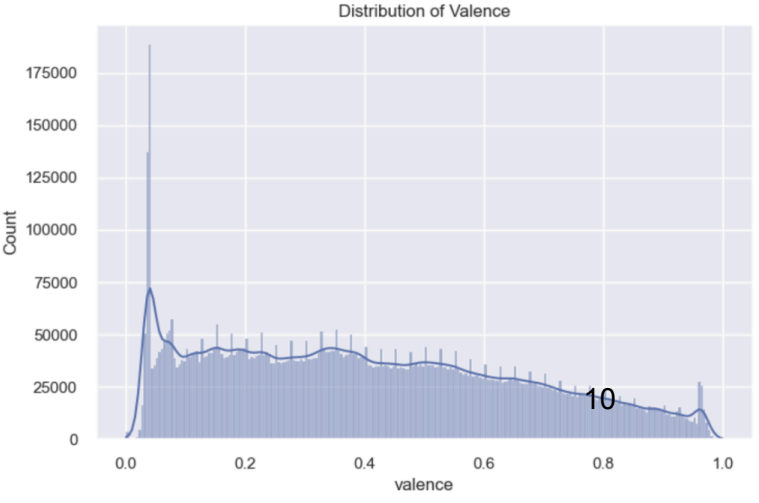




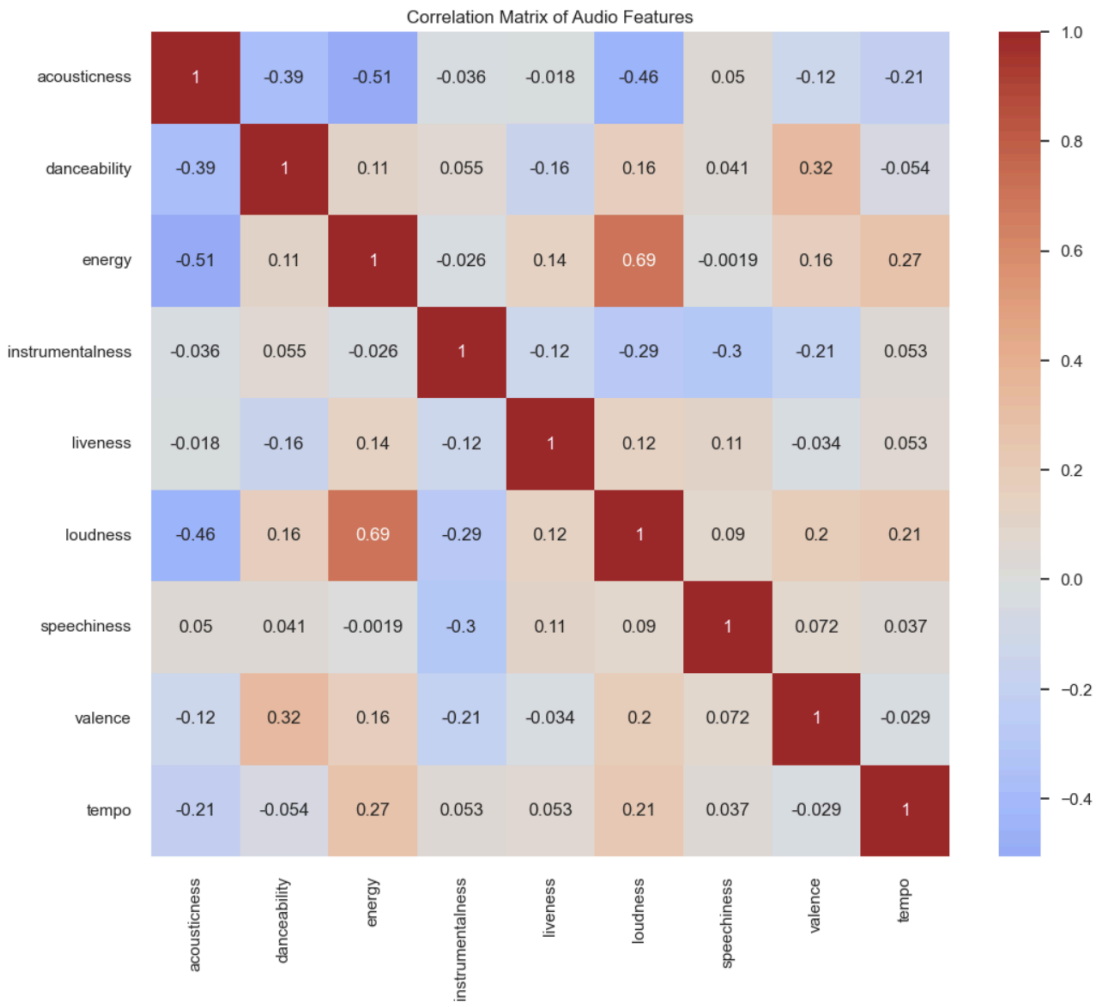




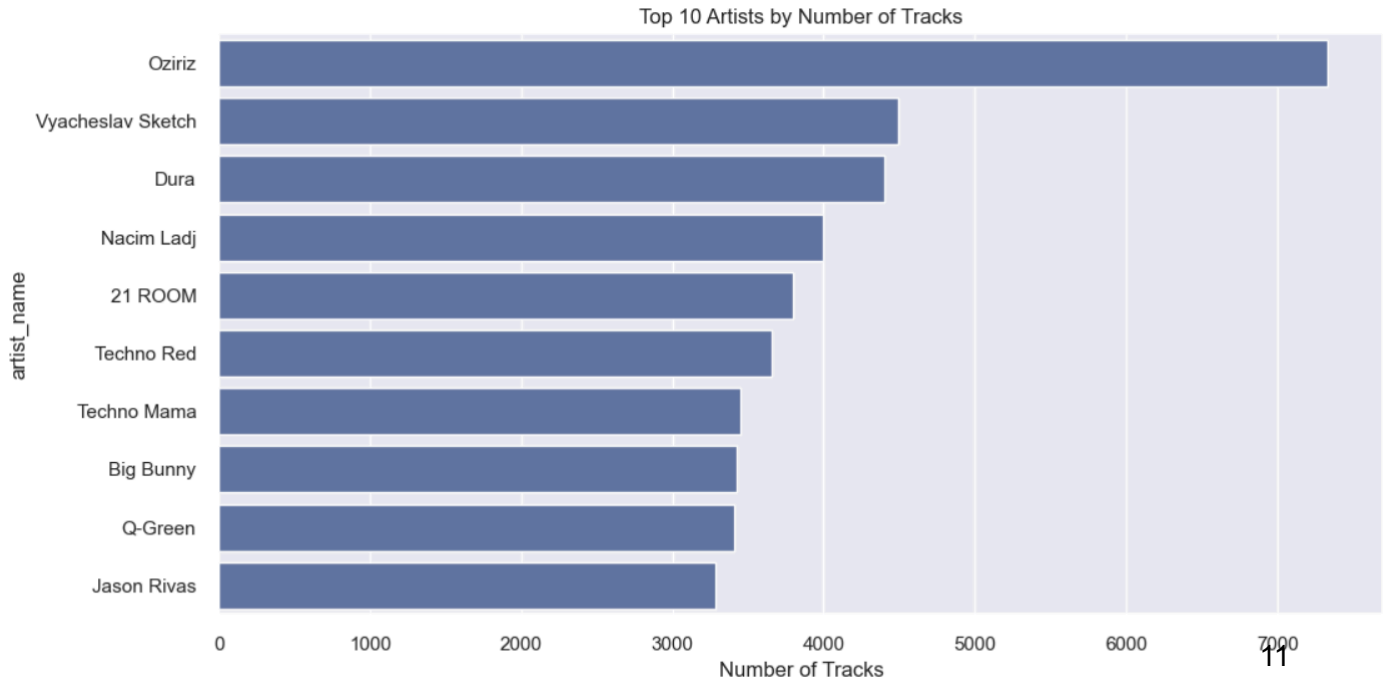




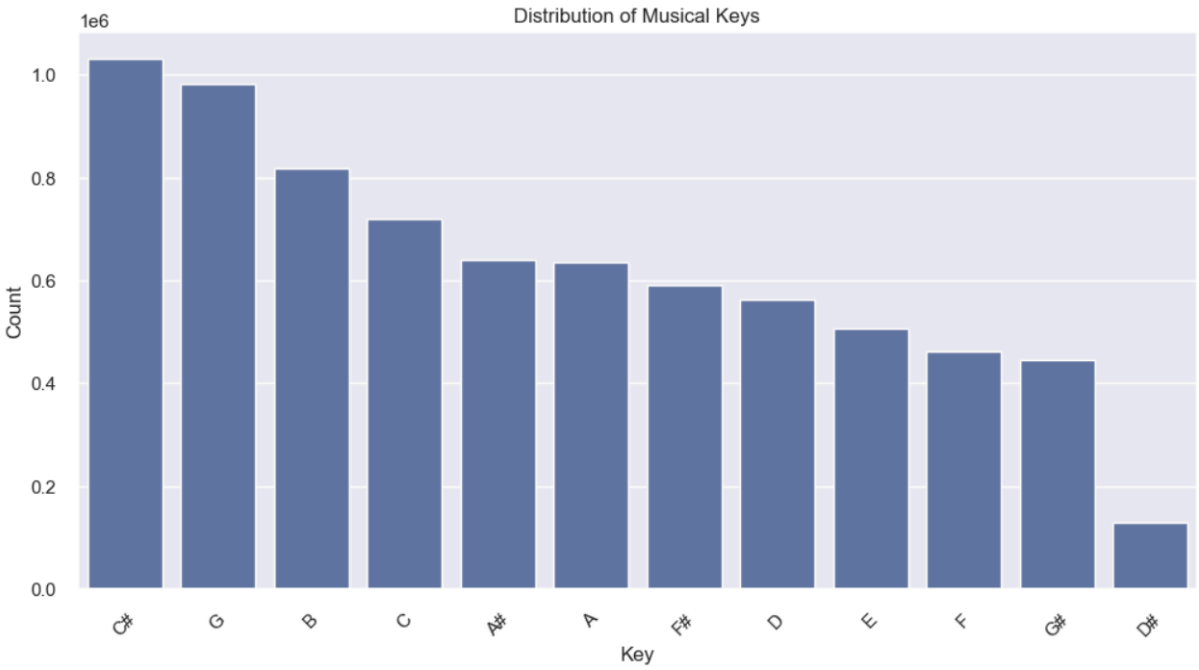
Correlation Matrix



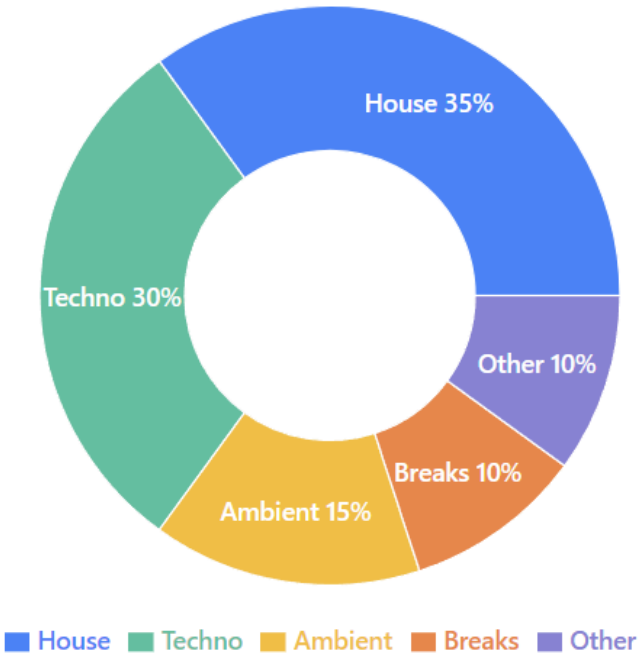
Top 10 Artists (volume)



Key Distribution



Genre Distribution



Based on analysis of track database

3.3 EDA Main Takeaways

Key findings from audio feature analysis:

- Most electronic tracks have high danceability (0.6-0.8)
- Energy levels typically range from 0.7-0.9
- Instrumentalness shows clear distinction between vocal/instrumental tracks
- House & Techno account for 65% of the genres in the dataframe

Key tempo findings:

- House music clustered around 120-128 BPM
- Techno ranging from 125-140 BPM
- Ambient/downtempo below 100 BPM

Key correlations discovered:

- Strong positive correlation between energy and loudness
- Moderate correlation between danceability and valence ; energy and tempo
- Accousticness shows negative correlations with tempo, danceability, loudness and energy (in that order)

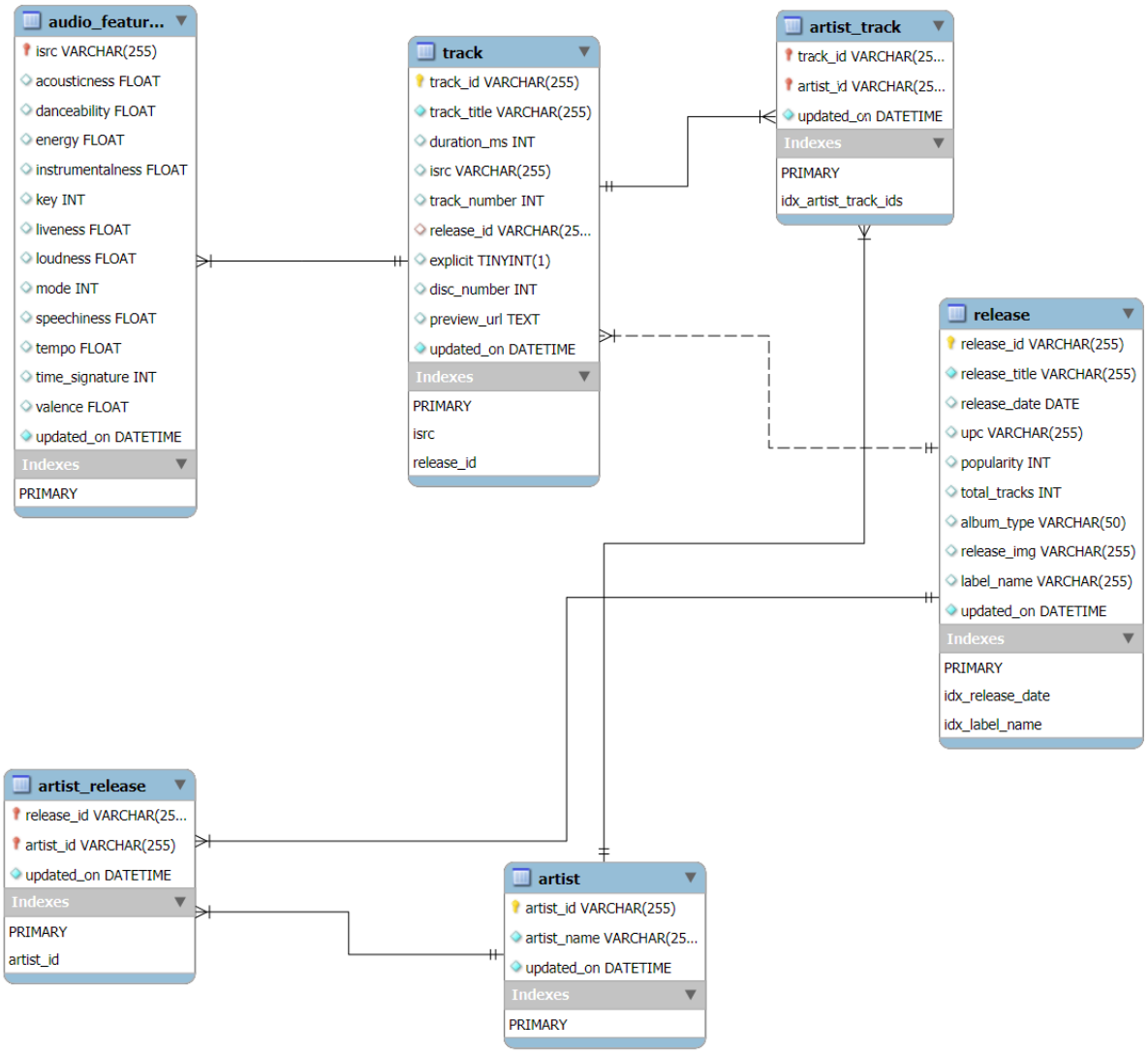
4. Database Implementation

4.1 SQL Database Design

I also connected my dataframe to a SQL database, using a relational and structured frame.

However, dealing with such a large database proved difficult when working on MySQL workbench, due to the sheer size of the imported dataframe. I had to limit some of the queries to a single year using release_date infos, in order to avoid timeouts (*Error 2013 - MySQL timeout* popped up quite a lot, even when changing internal my.ini settings to Timeout = 1000s). BigQuery didn't present such issues when handling large queries, thankfully.

I used SQLAlchemy to create the Tables within MySQL, then reversed engineered my table to get my Entity Relationship Diagram :



Connection & database schemas were made using SQLAlchemy :

```python
def create_mysql_engine():
    """Create SQLAlchemy engine for MySQL connection"""
    connection_string = f"mysql+pymysql://{DB_CONFIG['user']}:{DB_CONFIG['password']}@{DB_CONFIG['host']}/{DB_CONFIG['database']}"
    return create_engine(connection_string)
```

```python
def create_database_schema(engine):
    """Create database tables"""
    try:
        print("Creating database schema...")
        with engine.connect() as conn:
            # Create ARTIST table
            conn.execute(text("""
            CREATE TABLE IF NOT EXISTS artist (
                artist_id VARCHAR(255) PRIMARY KEY,
                artist_name VARCHAR(255) NOT NULL,
                updated_on DATETIME NOT NULL
            )
            """))

            # Create RELEASE table
            conn.execute(text("""
            CREATE TABLE IF NOT EXISTS `release` (
                release_id VARCHAR(255) PRIMARY KEY,
                release_title VARCHAR(255) NOT NULL,
                release_date DATE,
                upc VARCHAR(255),
                popularity INT,
                total_tracks INT,
                album_type VARCHAR(50),
                release_img VARCHAR(255),
                label_name VARCHAR(255),
                updated_on DATETIME NOT NULL
            )
            """))

            # Create TRACK table
            conn.execute(text("""
            CREATE TABLE IF NOT EXISTS track (
                track_id VARCHAR(255) PRIMARY KEY,
                track_title VARCHAR(255) NOT NULL,
                duration_ms INT,
                isrc VARCHAR(255) UNIQUE,
                track_number INT,
                release_id VARCHAR(255),
                explicit BOOLEAN,
                disc_number INT,
                preview_url TEXT,
                updated_on DATETIME NOT NULL,
                FOREIGN KEY (release_id) REFERENCES `release`(release_id)
            )
            """))
```

Example Queries in MySQL :

```sql
-- Get alva noto tracks
SELECT a.artist_name, t.track_title
FROM artist a
JOIN artist_track at ON a.artist_id = at.artist_id
JOIN track t ON at.track_id = t.track_id
WHERE a.artist_id = '1zrqDVuh55auIRthalFdXp'
LIMIT 20;

--------------------------------------------------
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

artist_name	track_title
alva noto	U_01-2-0
alva noto	Chamomile Day - Alva Noto Remodel
alva noto	Spray
alva noto	Early Winter (For Phill Niblock)
alva noto	Silence
alva noto	Grains
alva noto	Movement 7 - Live
alva noto	Uni Asymmetric Sweep
alva noto	Bit
alva noto	Plateaux 1
alva noto	Trioon I
alva noto	Reverso
alva noto	Module 4
alva noto	Microon II
alva noto	05-10-06 Astoria
alva noto	Ans (For Evgeny Murzin)
alva noto	Xerrox Neige
alva noto	Kinder der Sonne - Reprise
alva noto	Uni Normal
alva noto	Scape I - Live

```sql
-- Label Analysis
SELECT
    label_name,
    COUNT(*) as release_count
FROM `release`
WHERE label_name IS NOT NULL
GROUP BY label_name
ORDER BY release_count DESC
LIMIT 20;
```

Result Grid | Filter Rows: | Export:

label_name	release_count
recordJet	1473
Recovery House	1240
Bonzai Back Catalogue	1188
Club Session	1183
Nothing But	1163
RH2	1048
Mental Madness Records	996
HOT-Q	943
VinDig	914
Rimoshee Traxx	902
Recovery Tech	785
Armada Music	780
Armada Music Albums	772
Soundfield	767
Nervous Records	742
Diffuse Reality Records	736
Ultra Records	724
Black Hole Recordings	681

```sql
-- 4. Label Specializations
CREATE TEMPORARY TABLE temp_label_features AS
SELECT
    r.label_name,
    af.danceability,
    af.energy,
    af.valence,
    af.instrumentalness,
    af.acousticness
FROM `release` r
JOIN track t ON t.release_id = r.release_id
JOIN audio_features af ON t.isrc = af.isrc
WHERE YEAR(r.release_date) >= 2023
    AND r.label_name IS NOT NULL
LIMIT 100000;

SELECT
    label_name,
    COUNT(*) as track_count,
    CASE
        WHEN AVG(danceability) >= 0.7 THEN 'Dance-focused'
        WHEN AVG(instrumentalness) >= 0.7 THEN 'Instrumental-focused'
        WHEN AVG(acousticness) >= 0.7 THEN 'Acoustic-focused'
        WHEN AVG(energy) >= 0.7 THEN 'High-energy'
        ELSE 'Mixed'
    END as label_specialty
FROM temp_label_features
GROUP BY label_name
HAVING track_count >= 10
ORDER BY track_count DESC
LIMIT 15;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

label_name	track_count	label_specialty
LW Recordings	1809	High-energy
HOT-Q	1332	High-energy
Nothing But	1097	High-energy
Breaks Music Group	725	Dance-focused
5272433 Records DK	613	Instrumental-focused
recordJet	581	Mixed
Ring Mode Records	440	Instrumental-focused
Diffuse Realty Records	425	Instrumental-focused
otherm:nd	393	Dance-focused
NOV4 Records	384	Instrumental-focused
Clepsydra	344	Dance-focused
Artistfy Music	318	Mixed
2960653 Records DK	302	Instrumental-focused
Global Player Music	301	Mixed
7AGE Music	264	Mixed

```sql
-- 3. Music Type Categories based on Audio Features
CREATE TEMPORARY TABLE temp_music_categories AS
SELECT
    t.track_id,
    t.track_title,
    r.release_date,
    af.danceability,
    af.energy,
    af.valence,
    CASE
        WHEN af.danceability >= 0.7 AND af.energy >= 0.7 THEN 'Dance/Party'
        WHEN af.energy >= 0.7 AND af.valence <= 0.3 THEN 'Intense/Dark'
        WHEN af.energy <= 0.3 AND af.instrumentalness >= 0.7 THEN 'Ambient/Chill'
        WHEN af.acousticness >= 0.7 THEN 'Acoustic'
        WHEN af.energy >= 0.7 AND af.valence >= 0.7 THEN 'Upbeat/Happy'
        ELSE 'Other'
    END as music_category
FROM track t
JOIN `release` r ON t.release_id = r.release_id
JOIN audio_features af ON t.isrc = af.isrc
WHERE YEAR(r.release_date) >= 2023
LIMIT 100000;

-- Analysis of music categories
SELECT
    music_category,
    COUNT(*) as track_count,
    ROUND(AVG(danceability), 3) as avg_danceability,
    ROUND(AVG(energy), 3) as avg_energy,
    ROUND(AVG(valence), 3) as avg_valence
FROM temp_music_categories
GROUP BY music_category
ORDER BY track_count DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

music_category	track_count	avg_danceability	avg_energy	avg_valence
Other	42685	0.662	0.643	0.396
Dance/Party	29933	0.778	0.856	0.475
Intense/Dark	15475	0.558	0.883	0.146
Ambient/Chill	5635	0.451	0.166	0.204
Acoustic	3293	0.48	0.414	0.284
Upbeat/Happy	2979	0.614	0.886	0.819

```sql
SELECT
    release_year,
    COUNT(DISTINCT track_id) AS track_count
FROM temp_release_tracks
GROUP BY release_year
ORDER BY release_year DESC
LIMIT 10;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

release_year	track_count
2023	279143
2022	388952
2021	433384
2020	493699
2019	452674
2018	363435
2017	349656
2016	323017
2015	292266
2014	277278

```sql
-- Top 10 artists in track count
SELECT
    a.artist_name,
    COUNT(DISTINCT at.track_id) AS track_count
FROM artist a
JOIN artist_track at ON a.artist_id = at.artist_id
GROUP BY a.artist_id
ORDER BY track_count DESC
LIMIT 10;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

artist_name	track_count
Oziriz	7351
Vyacheslav Sketch	4496
Dura	4410
Nacim Ladj	4003
21 ROOM	3810
Techno Red	3672
Jason Rivas	3545
Techno Mama	3445
Big Bunny	3439
Q-Green	3419

4.2 BigQuery Implementation

The database was also connected to BigQuery, ensuring fast retrieval of data using complex queries. I created a project, then a dataset containing a single table (I passed a truncated CSV file containing 2022 data only). Some example queries below :

4.2.1 Audio Feature analysis by Popularity

```sql
-- Audio Feature Analysis by Popularity
WITH popularity_categories AS (
    SELECT *,
    CASE
        WHEN popularity >= 75 THEN 'Very Popular'
        WHEN popularity >= 50 THEN 'Popular'
        WHEN popularity >= 25 THEN 'Moderate'
        ELSE 'Less Popular'
    END as popularity_category
    FROM `olivier-442117.spotify_data.tracks_2022`
)
SELECT
    popularity_category,
    COUNT(*) as track_count,
    ROUND(AVG(danceability), 3) as avg_danceability,
    ROUND(AVG(energy), 3) as avg_energy,
    ROUND(AVG(valence), 3) as avg_valence,
    ROUND(AVG(tempo), 1) as avg_tempo
FROM popularity_categories
GROUP BY popularity_category
ORDER BY track_count DESC;
```

Résultats de la requête

INFORMATIONS SUR LE JOB | RÉSULTATS | GRAPHIQUE | JSON | DÉTAILS DE L'EXÉCUTION | GRAPHIQUE D'EXÉCU

La mise en cache des métadonnées est désactivée. Vous pouvez accélérer les requêtes sur des tables externes en activant la mise en cache

Ligne	popularity_category	track_count	avg_danceability	avg_energy	avg_valence	avg_tempo
1	Less Popular	434294	0.673	0.726	0.386	127.4
2	Moderate	15226	0.62	0.676	0.378	124.6
3	Popular	1165	0.598	0.637	0.399	121.7
4	Very Popular	63	0.625	0.607	0.352	120.1

4.2.2 Audio Feature analysis for Top Performers

```sql
-- Audio Feature Analysis for Top Performers
WITH top_artists AS (
    SELECT DISTINCT artist_name
    FROM (
        SELECT artist_name
        FROM `olivier-442117.spotify_data.tracks_2022`
        GROUP BY artist_name
        ORDER BY COUNT(*) DESC
        LIMIT 10
    )
)
SELECT
    a.artist_name,
    ROUND(AVG(t.popularity), 2) as avg_popularity,
    ROUND(AVG(t.danceability), 3) as avg_danceability,
    ROUND(AVG(t.energy), 3) as avg_energy,
    ROUND(AVG(t.valence), 3) as avg_valence,
    ROUND(AVG(t.tempo), 1) as avg_tempo,
    COUNT(*) as total_tracks
FROM top_artists a
JOIN `olivier-442117.spotify_data.tracks_2022` t
ON a.artist_name = t.artist_name
GROUP BY a.artist_name
ORDER BY avg_popularity DESC;
```

Résultats de la requête

INFORMATIONS SUR LE JOB | RÉSULTATS | GRAPHIQUE | JSON | DÉTAILS DE L'EXÉCUTION | GRAPHIQUE D'EXÉCUTION

La mise en cache des métadonnées est désactivée. Vous pouvez accélérer les requêtes sur des tables externes en activant la mise en cache des métadonnées. En

Ligne	artist_name	avg_popularity	avg_danceability	avg_energy	avg_valence	avg_tempo	total_tracks
1	Mauro Rawn	0.73	0.666	0.644	0.695	121.1	466
2	Techno Peaktime Hunter	0.29	0.727	0.622	0.318	128.6	442
3	Gianluigi Toso	0.02	0.753	0.766	0.642	127.3	579
4	Benny Montaquila DJ	0.0	0.745	0.73	0.557	128.1	739
5	Big Bunny	0.0	0.8	0.796	0.353	123.9	568
6	Techno Mama	0.0	0.801	0.87	0.379	126.4	714
7	Rousing House	0.0	0.789	0.762	0.558	118.1	478
8	Q-Green	0.0	0.777	0.762	0.354	125.8	815
9	Spiral Helix	0.0	0.651	0.673	0.55	127.6	860
10	Mr Dee Swiss House	0.0	0.694	0.772	0.432	123.3	838

4.2.3 Top 10 artists in 2023

Requête sans titre | EXÉCUTER | ENREGISTRER | TÉLÉCHARGER | PARTAGER

```sql
-- Top Artists
SELECT
    artist_name,
    COUNT(*) as total_tracks,
    ROUND(AVG(popularity), 2) as avg_popularity,
    ROUND(AVG(danceability), 3) as avg_danceability,
    ROUND(AVG(energy), 3) as avg_energy
FROM `olivier-442117.spotify_data.tracks_2022`
GROUP BY artist_name
HAVING total_tracks > 5
ORDER BY avg_popularity DESC
LIMIT 10;
```

Résultats de la requête

INFORMATIONS SUR LE JOB | RÉSULTATS | GRAPHIQUE | JSON | DÉTAILS DE L'EXÉCUTION

La mise en cache des métadonnées est désactivée. Vous pouvez accélérer les requêtes sur des tables externes en activ

Ligne	artist_name	total_tracks	avg_popularity	avg_danceability	avg_energy
1	Beyoncé	16	87.0	0.729	0.666
2	The Weeknd	16	84.0	0.59	0.655
3	Drake	14	83.0	0.644	0.504
4	Swedish House Mafia	18	74.94	0.546	0.596
5	Caamp	12	69.0	0.574	0.532
6	Kygo	14	68.0	0.625	0.636
7	Big Thief	20	66.0	0.526	0.532
8	LoFi Waiter	10	66.0	0.663	0.122
9	Calvin Harris	14	64.0	0.71	0.706
10	Tove Lo	12	64.0	0.668	0.637

5. API Development

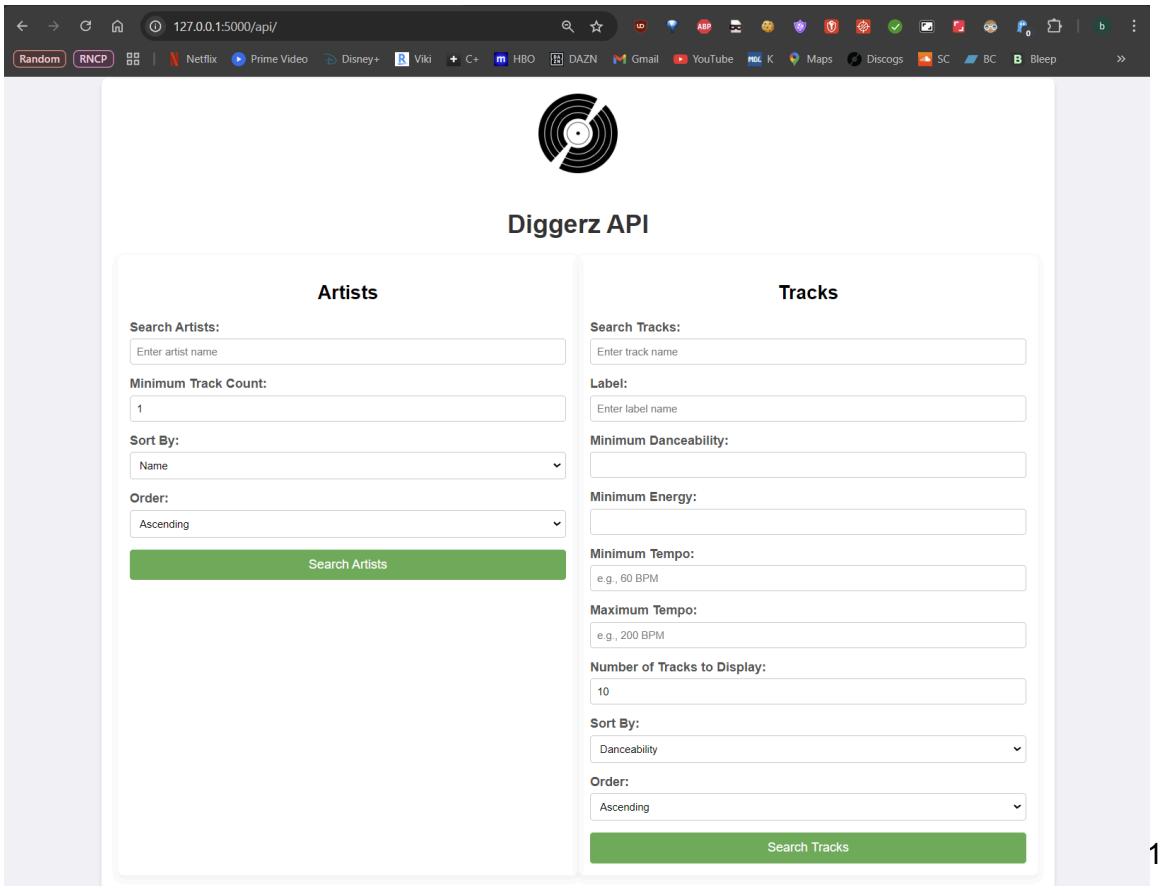
5.1 Flask API Architecture

The Diggerz API is built using Flask, providing RESTful endpoints for accessing music data and recommendations. I also used SQLAlchemy for querying the database.

Here's the CLI prompt to run the local instance :

```
C:\Users\olivi\Documents\Ironhack (main)
(streamlitenv) λ cd final-project\

C:\Users\olivi\Documents\Ironhack\final-project (main -> origin)
(streamlitenv) λ cd flask-api\

C:\Users\olivi\Documents\Ironhack\final-project\flask-api (main -> origin)
(streamlitenv) λ python run.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 498-559-366
```

With a HTML interface :

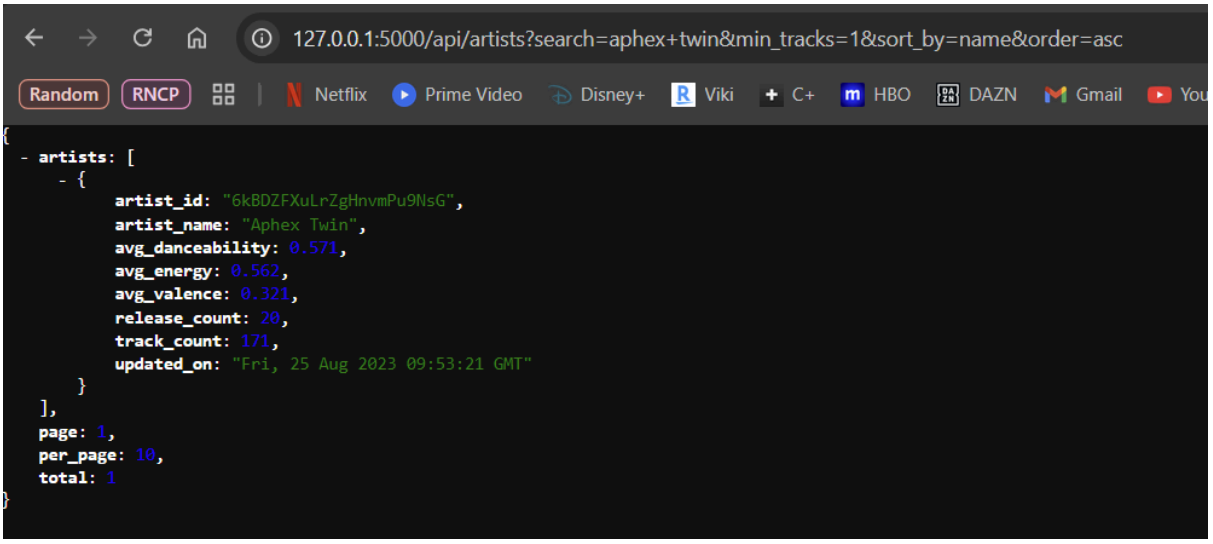


5.2 Endpoint Design

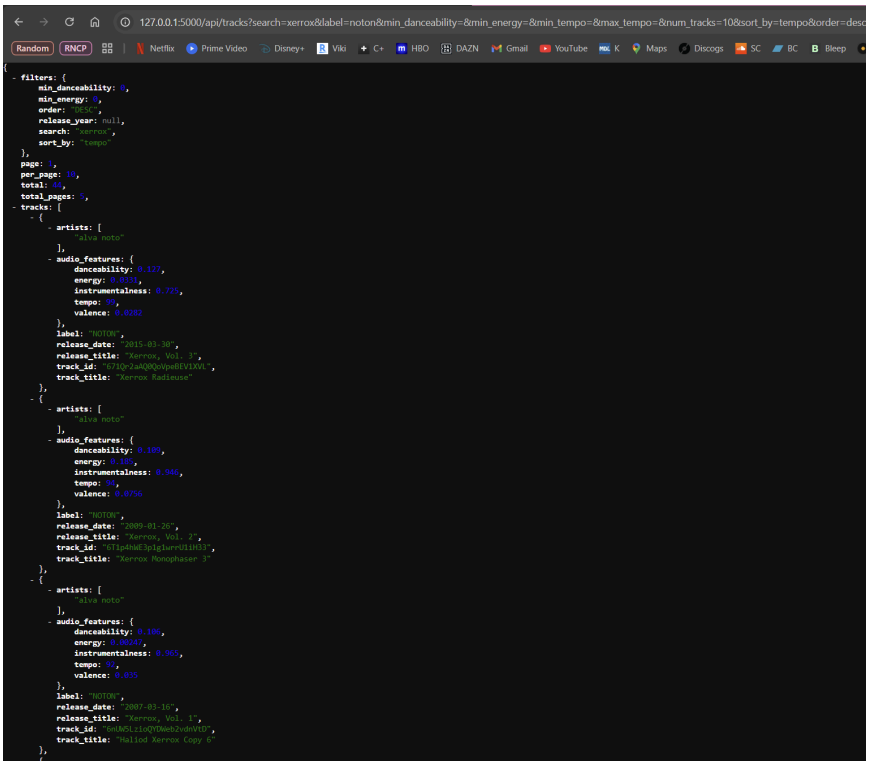
The routes consist in different endpoints, with GET requests :

- **“/” Route** : with HTML rendering and dynamic querying
- **“/artists” Route** : to get artists information
- **“/artists/<artist_id>” Route** : to get a specific artist information
- **“/tracks” Route** : to get tracks information
- **“/tracks/<track_id>” Route** : to get a specific track information

Query for “Aphex Twin” artist

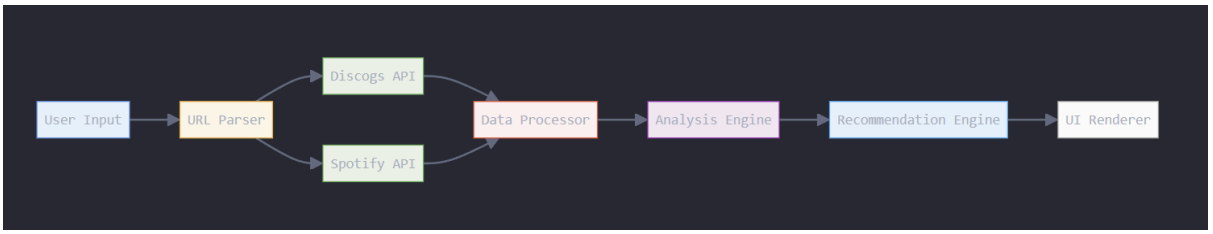


Query for “alva noto” tracks



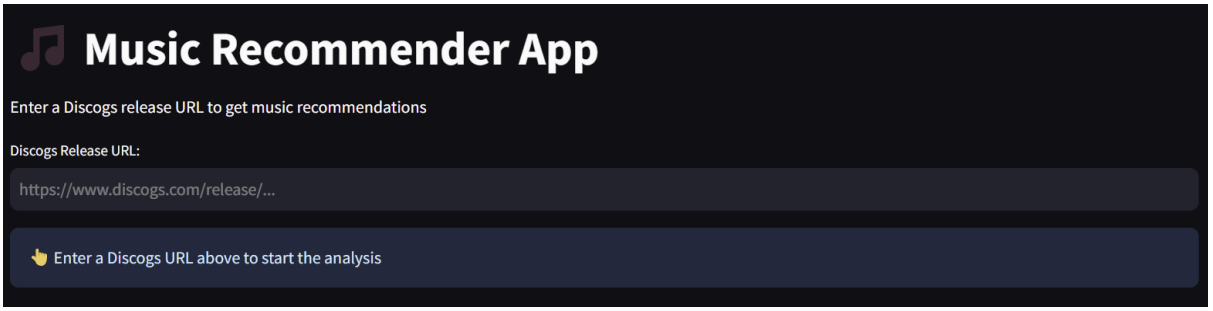
6. Streamlit Application

The core of my project is the Diggerz app. It prompts a user to input a Discogs URL of a release he likes for recommendations. Here's the process flow :



6.1 User Interface Design

For the first step, the user is required to input a release URL from an album he likes from Discogs :



The URL parser will then identify the release_id that will be used to query the Discogs API for the album metadata :

https://www.discogs.com/release/30797823-cv313-Beyond-Starlit-Sky

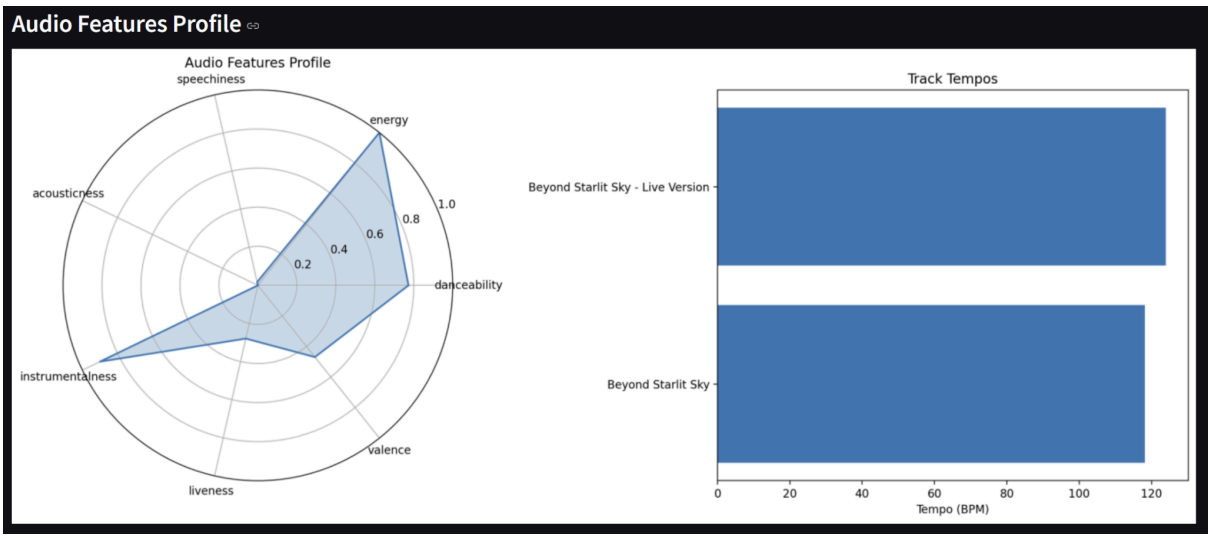
We will then return the selected album information :

```
st.markdown(f"**Artist**: {discogs_info.get('artist', 'Unknown Artist')}")
st.markdown(f"**Album**: {discogs_info.get('album', 'Unknown Album')}")
st.markdown(f"**Label**: {discogs_info.get('label', 'Unknown Label')}")
st.markdown(f"**Catalog**: {discogs_info.get('catalog', 'Unknown')}")
st.markdown(f"**Format**: {discogs_info.get('format', 'Unknown Format')}")
st.markdown(f"**Year**: {discogs_info.get('year', 'Unknown Year')}")
st.markdown(f"**Styles**: {', '.join(discogs_info.get('styles', []))}")
```

We will also scrape the album price statistics using Selenium, as covered in the Data Collection section.

6.2 Data processing pipeline

Once the query has been made to Discogs API, the app will then query the Spotify API using Discogs metadata, and retrieve its Audio features in order to build an “audio profile” of the selected album.



6.3 Analysis Engine

The Core of the App is the analysis engine :

- We will fetch the album danceability, energy, key and BPM and normalize them to eliminate any bias using scaling : all features are rated between 0 and 1.

- Key mapping : made for harmonic relationships.

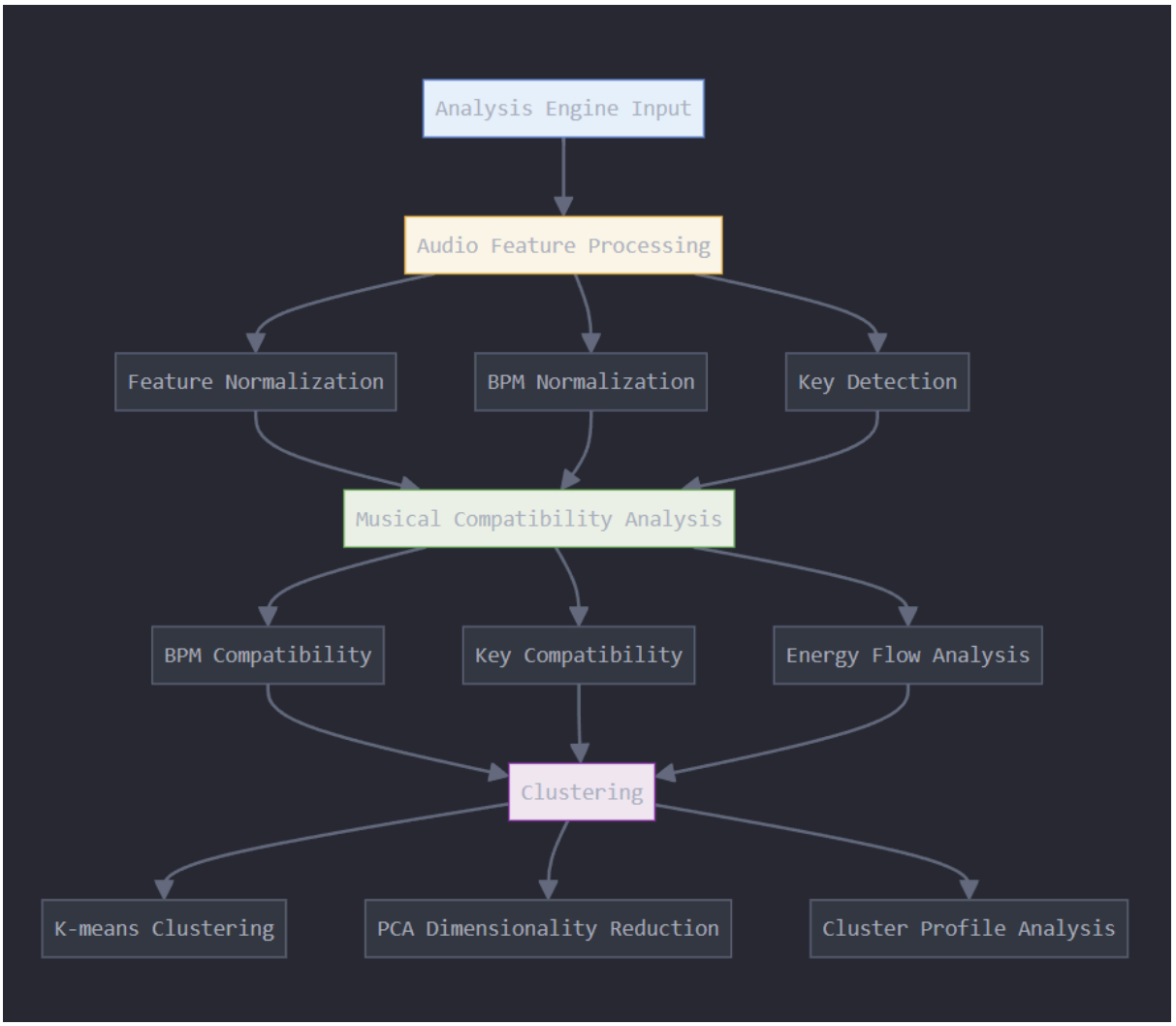
- Energy : tracks energy progression between songs, ensures smooth transitions, maintains dancefloor energy.

Machine learning algorithms : K-means clustering (8 clusters) and PCA :

- Clustering allows the reorganization of tracks into “bins” of similar audio features ; creating “musical neighborhoods”.

- This type of grouping reveals relationships between tracks and improves recommendations.

- PCA (Principal Component Analysis) : Improves clustering efficiency by group all audio features into 2 main characteristics : "Energy Level" (combining energy, tempo, and danceability) & "Mood" (combining valence, instrumentalness, and acousticness)



6.4 Recommendation engine

Once the Spotify CSV audio features data has been processed, the recommendation engine comes into play using a **weighted scoring system** :

6.4.1 Style/Genre Matching (45% of final score)

- Primary Focus: Most important factor in recommendations
- Implementation: Uses TF-IDF (term frequency-inverse document frequency) vectorization to compare styles (turns “styles” into numbers to be compared, using vectors)
- Purpose: Ensures genre consistency and stylistic relevance
- Example: A Deep House track will primarily match with other Deep House tracks, but might also match with related styles like Tech House

6.4.2 Audio Feature Analysis (20% of final score)

- Components: Analyzes danceability, energy, speechiness, etc.
- Method: Uses cosine similarity between feature vectors
- Advantage: Captures the "sound" of tracks beyond genre labels
- Application: Helps find tracks that "sound similar" even if labeled differently

6.4.3 Cluster Matching (15% of final score)

- Function: Groups similar tracks into musical "neighborhoods"
- Benefit: Speeds up recommendation process
- Impact: Helps identify tracks with similar overall characteristics
- Usage: Gives preference to tracks in the same cluster as the input track

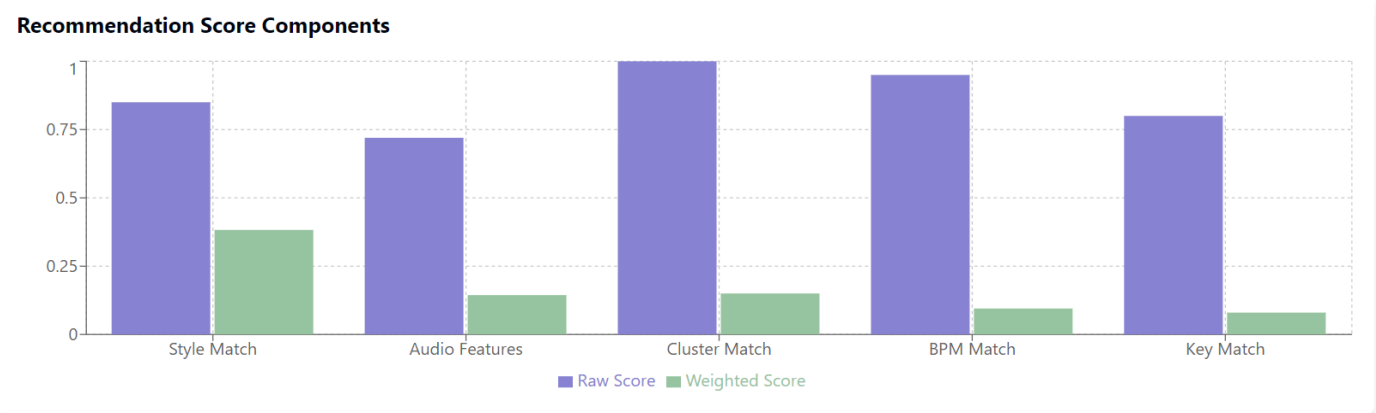
6.4.4 Musical Compatibility (20% combined)

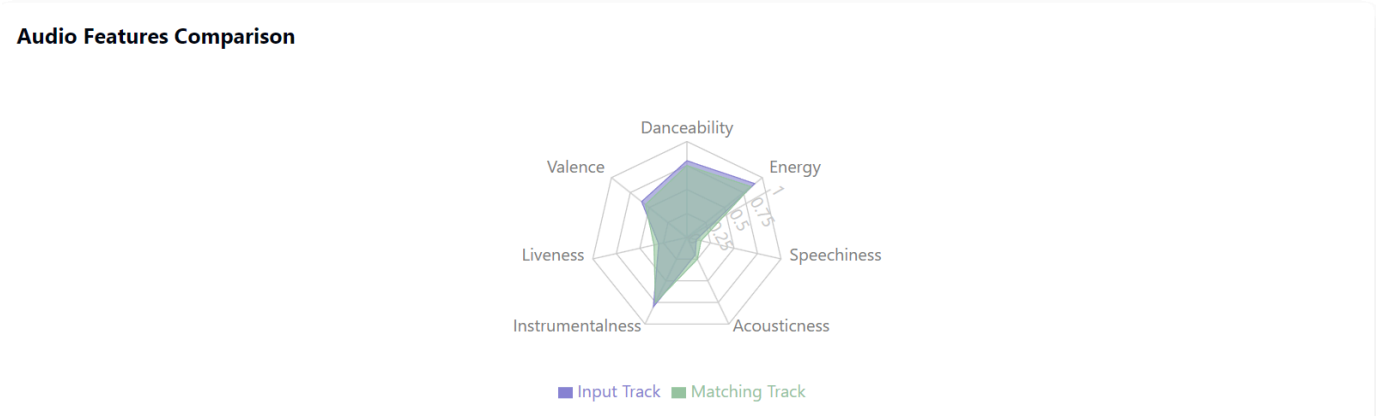
- BPM Matching (10%): Ensures tracks can be mixed together
- Key Compatibility (10%): Follows harmonic mixing principles
- Goal: Makes recommendations DJ-friendly
- Result: Suggested tracks work well together in a mix

Once the weighted calculated score is done for all the tracks, we compare them to our input albums track(s). Tracks with the best compatibility of genre, audio features, key (pitch), tempo (BPM) and energy levels are sent as recommendations to the end user.

There is a function that filters the recommendations to ensure :

- Lowest matches are excluded using a score threshold (0.4)
- Duplicate tracks and versions removal (eg. remix of the same track)
- Good diversity in the recommendations
- Good balance between diversity and similarity



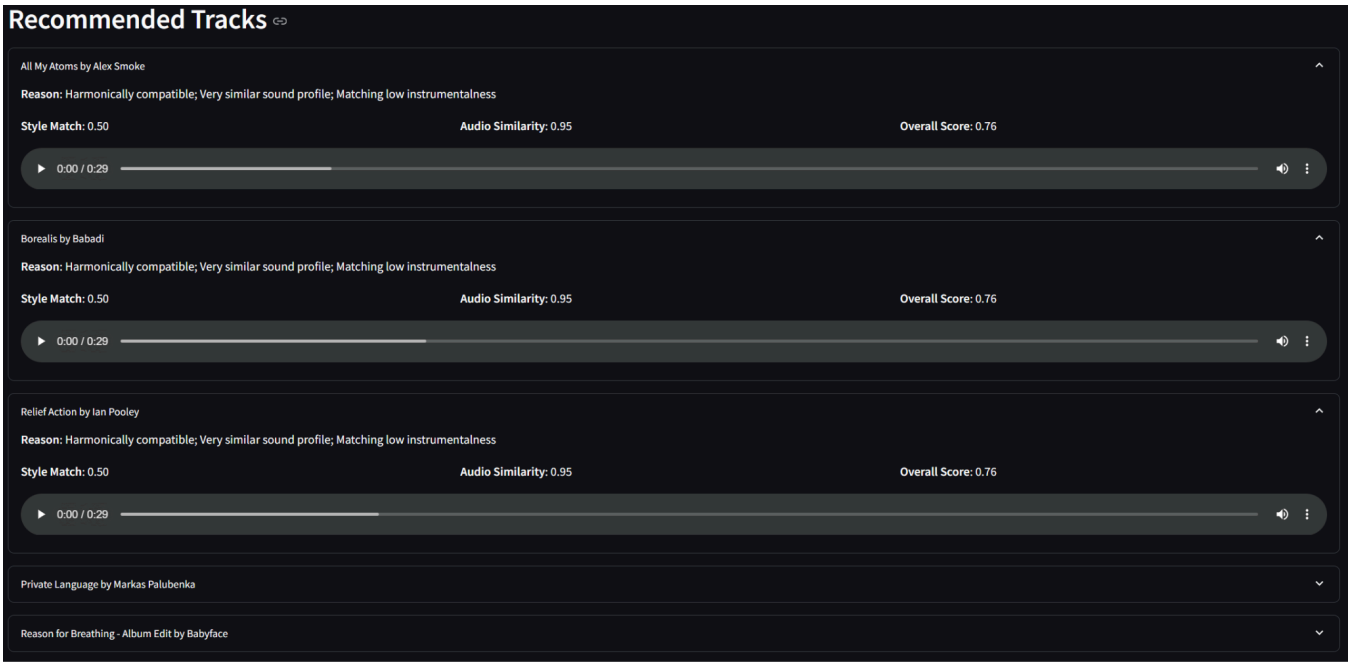


6.5 Result presentation

I've chosen Streamlit as my UI renderer, as it works seamlessly with my python code. It displays :

- An input box for the Discogs URL
- The input album information (metadata)
- Album prices from Discogs (Low/Median/High)
- The tracks recommendations (track name, artist name, style match, audio similarity, an overall score, and some audio previews)

Examples of recommended tracks



7.GDPR Compliance & Data Privacy

All the data is publicly available and doesn't infringe any personal information. CSVs were built using the open Discogs, Spotify and Beatport APIs.

8. Future improvements

The most challenging part of this project was getting accurate recommendation results. Some enhancements would be possible for the next iterations of the app :

- **Useful filters in the Streamlit UI** : such as BPM range, Date range, labels filters (eg. returning recommendations within the same label), styles multi-selection and formats available for purchase (Digital, Vinyl, CDs) ;
- **Buying Options** : through Beatport (digital files), Bandcamp (direct-to-artist purchases) or Discogs (physical medias, new or second-hand) ;
- **Increased recommendations accuracy** : by using the "get related artists" query within Spotify API, and prioritize recommendations of related artists ;
- **Discogs record collection analysis** : by accepting a Discogs' user collection URL to return a "music profile" through EDA, and return relevant labels, tracks and albums as recommendations ;
- **Performance optimizations** : It currently takes between 10 - 12 minutes to execute a query and returns recommendations. A couple ways to do this would be :
    - to make direct queries to the database (using the created API) rather than through the pandas dataframe ;
    - introducing caching for repeated queries...

9. References & Resources

Kaggle Dataset

https://www.kaggle.com/datasets/mcfurland/10-m-beatport-tracks-spotify-audio-features/data

Spotify API

https://developer.spotify.com/documentation/web-api

Discogs API (Python fork)

https://python3-discogs-client.readthedocs.io/en/latest/quickstart.html

Google Slides

📖 **Diggerz - Final Project**

GitHub Repo

https://github.com/oliebab/final-project