Diggerz

An electronic music recommender app

by Olivier Babiard

Building an app for DJs : the challenges

Music digging and record collecting is the most vital part of Djing. An extensive knowledge of artists, labels, genres, music affinity *and a bit of luck* is required to craft good DJ sets.

It is also the most difficult and time consuming task, with millions of releases & artists, plus a vast array of genres & styles to explore.

The best way to find new music starts with using the greatest music database in the world : Discogs. Founded in 2000, it currently stores 17.8M releases with all the relevant metadata. Electronic music accounts for almost 5M of them. It is also the number 1 marketplace in the world for physical media (vinyl, cassettes and CDs).

Building an app for DJs : the solution

In order to alleviate this challenging task, I decided to create an intelligent music discovery system :

- Automated music recommander app
- Taking advantage of the rich metadata provided by the Discogs API
- With intelligent matching based on musical elements provided by Spotify API

Spotify is currently the number 1 streaming platform in the world. They bought The Echo Nest data platform in 2014, a pioneer in music intelligence.

Music intelligence : leveraging Spotify Audio Features

Our music recommender app is based on Spotify API's audio features. These audio features are quantifiable characteristics of audio tracks :

- **accousticness** : presence of acoustic instruments, production style analysis
- **danceability** : rhythm stability, beat strength
- **energy** : perceptual intensity, activity level, dynamic range
- **speechiness** : presence of spoken words, vocal content analysis
- **instrumentalness** : Voice/instrument ratio
- **liveness** : audience presence, live recording detection
- **valence** : musical positiveness, mood detection, emotional content analysis
- **tempo** : BPM (beat per minute) analysis, vital for music matching
- **key** : use for pitch matching

Data Sources & Data collection

The starting point of my project was finding an appropriate data source for electronic music. I found a comprehensive data set on Kaggle (CSV flat file) that contained more than 10 Million tracks, using both Spotify and Beatport data. Ended up using only the Spotify data sets to limit the size of my database, with around 5M electronic music tracks (merged together, the CSV weighted 25Gb).

I complimented this data set with 3 other sources :

- Discogs API : contains rich metadata for releases - the most recent data dump is a 10Gb XML file
- Spotify API : contains EchoNest audio features, vital for my project
- Web Scraping : due to limitations of Discogs API for fetching price statistics, I had to resort to use Selenium to scrape some releases price statistics

I connected to the APIs using Tokens, client_ids and client_secrets. For the Discogs API, I used a Python fork of the official API (using Javascript)

```
# 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))
```

Exploratory Data Analysis

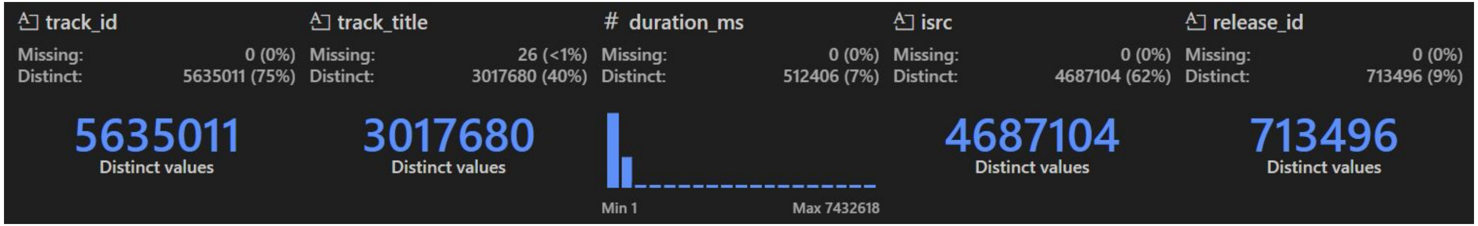
Data Cleaning

Fortunately, data cleaning remained minimal due to the very clean CSV fetched from Kaggle. I ran a few cleaning steps :

- dropping unused columns (like image URLs columns) ;
- checked for NA values ;
- dropped rows without vital values (track_id, artist_id, audio_features).

Creating the data frame

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 into a single CSV : spotify_complete_data.



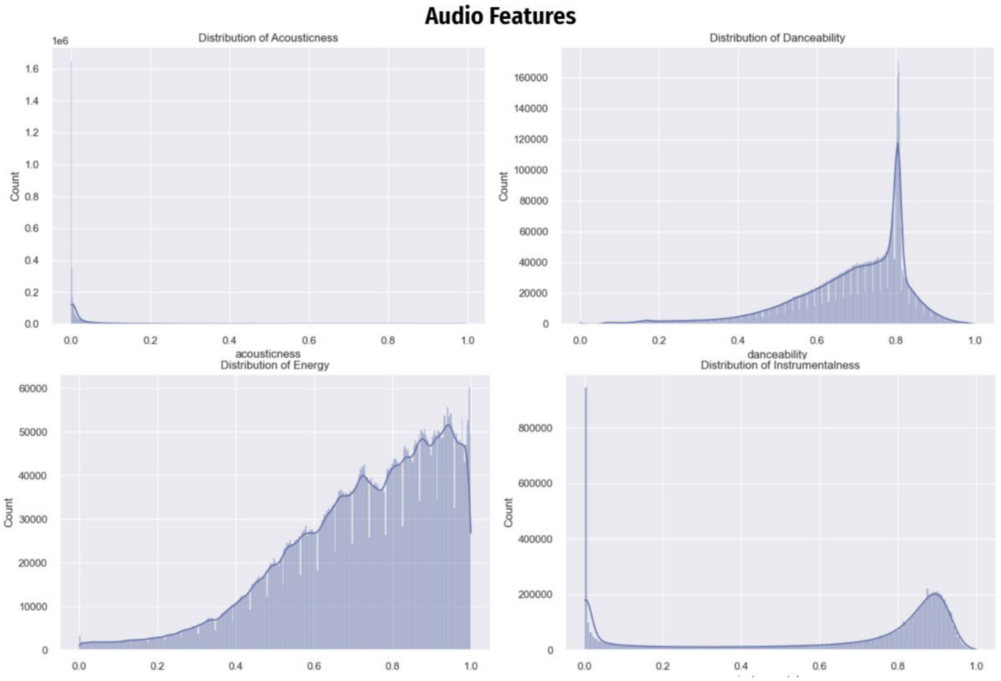
Exploratory Data Analysis

Exploratory Data Analysis

From there, I created a few functions to retrieve some insightful visualizations about my dataframe.

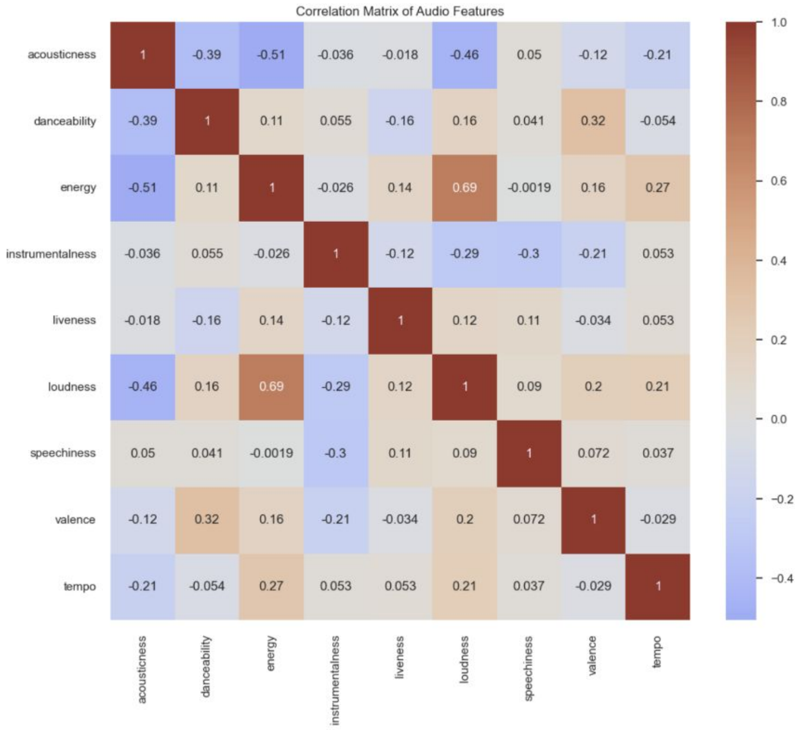
Some key takeaways :

- Globally low acousticness, expected for electronic music genre ;
- High danceability & energy : expected in most electronic music genres, such as House or Techno ;
- Low speechiness : main use of digital instruments and notes ;
- 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)

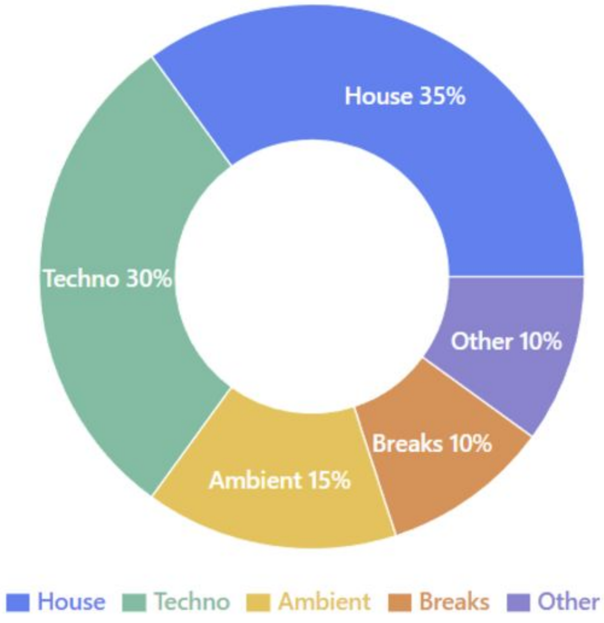


Exploratory Data Analysis cont'd

Correlation Matrix



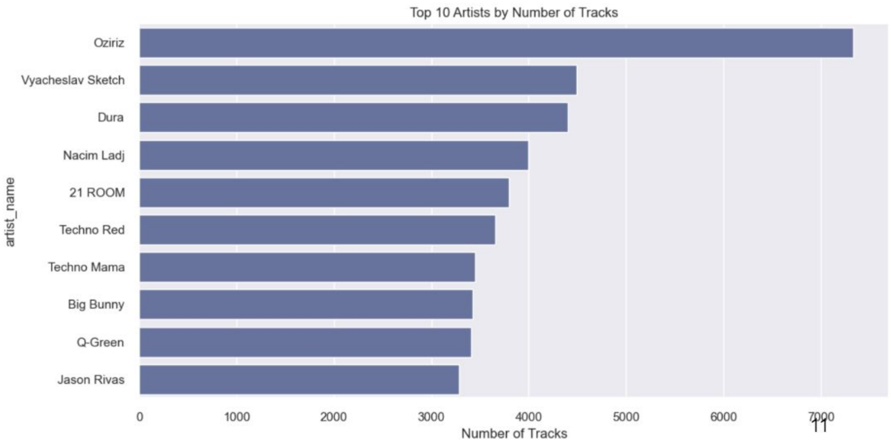
Genre Distribution



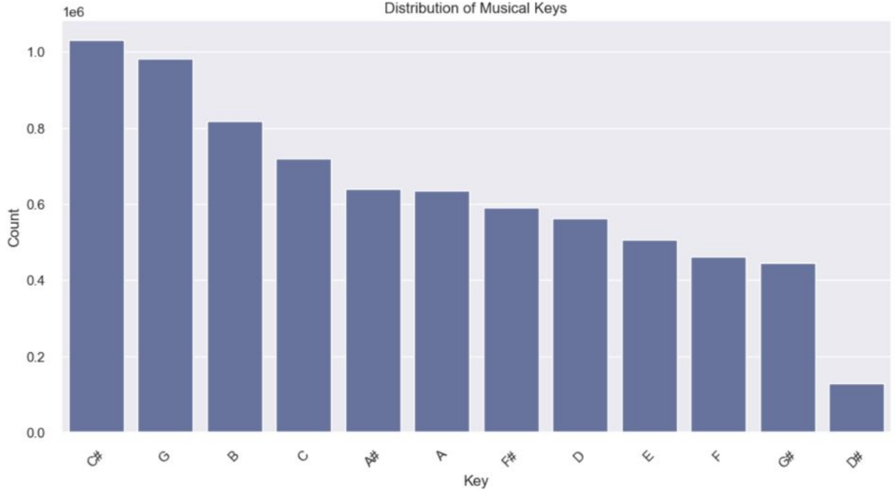
Based on analysis of track database

Exploratory Data Analysis cont'd

Top 10 Artists (volume)



Key Distribution



Database Implementation : SQL & BigQuery

I used a relational database for my Project : MySQL. Musical informations are highly structured, and follow the same patterns :

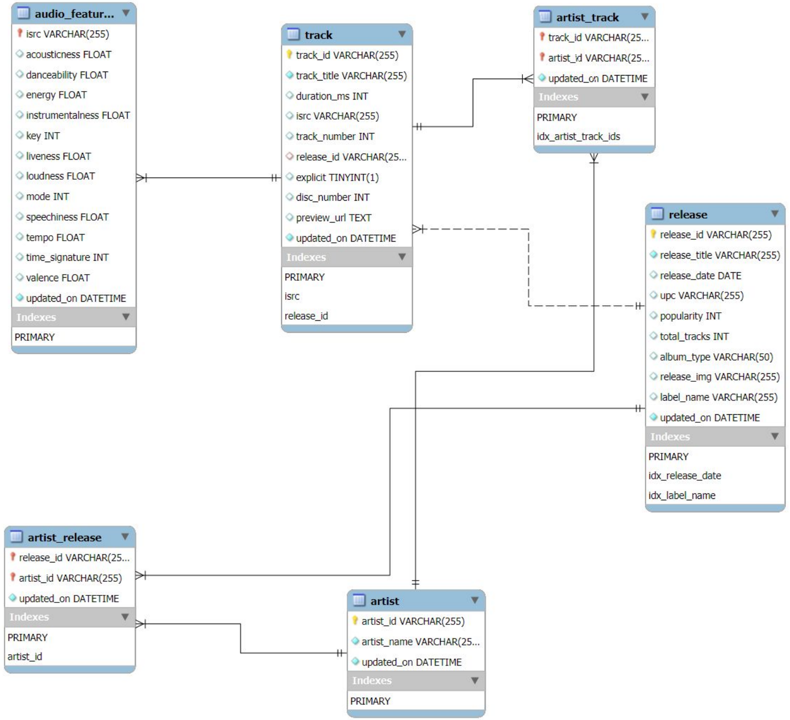
- Consistent informations : track_id, artist_id, release_id, ISRC, duration ;
- Audio features are standardized ;
- Same general pattern for all releases.

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.

Database Implementation : SQL & BigQuery

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



Database Implementation : SQL & BigQuery

Example Queries :

```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

Database Implementation : SQL & BigQuery

I also used BigQuery for my Database. While MySQL suffered from timeouts when handling complex queries, with Error 2013 (Timeouts) ; BigQuery worked flawlessly.

```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

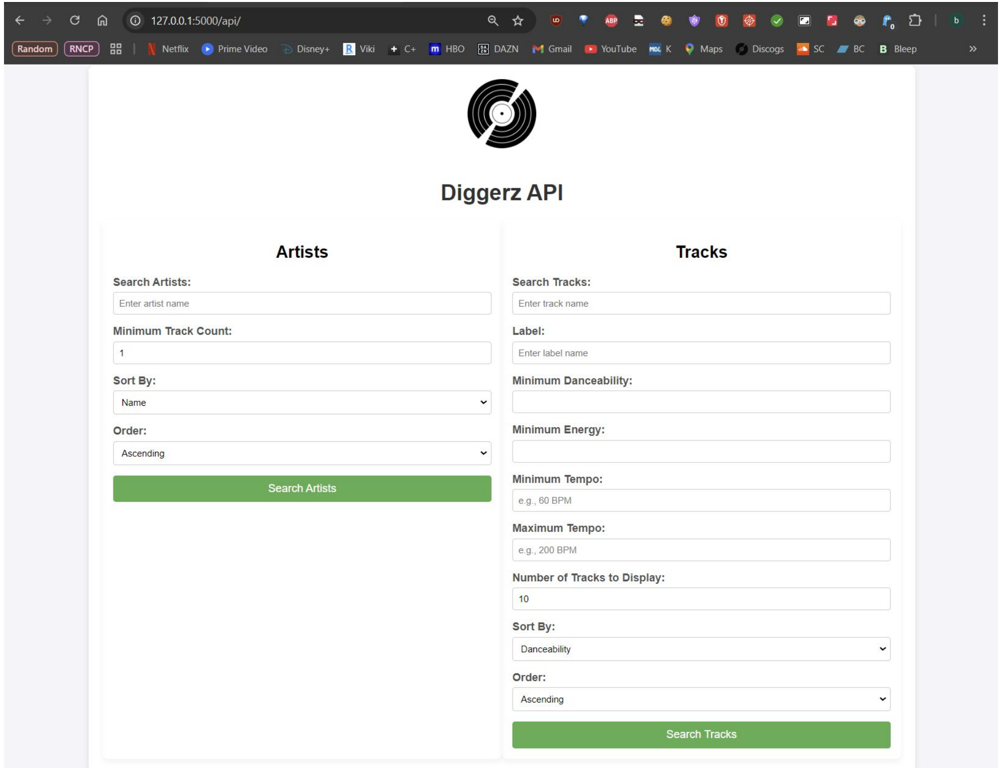
API Development : Using Flask

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

I used 5 endpoints for my API :

- **“/” 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

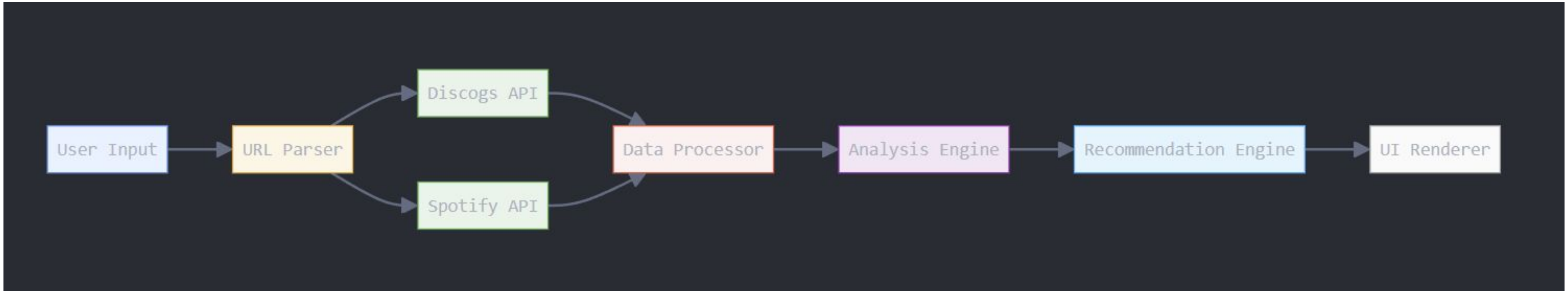
```
C:\Users\olivi\Documents\Ironhack (main)
(streamlitenv) λ cd final-project\

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
```



Streamlit App : process flow

For this project, I used a 3.2Gb CSV file that contains all the Spotify API data for electronic music : more than 5M tracks, with metadata such as artist name, album name, ISRC (unique identifier of a music track), audio preview links and audio features.

The process flow is as follows :



Step 1 : URL parser & Discogs Query

For the first step, 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 Discogs for the album metadata :

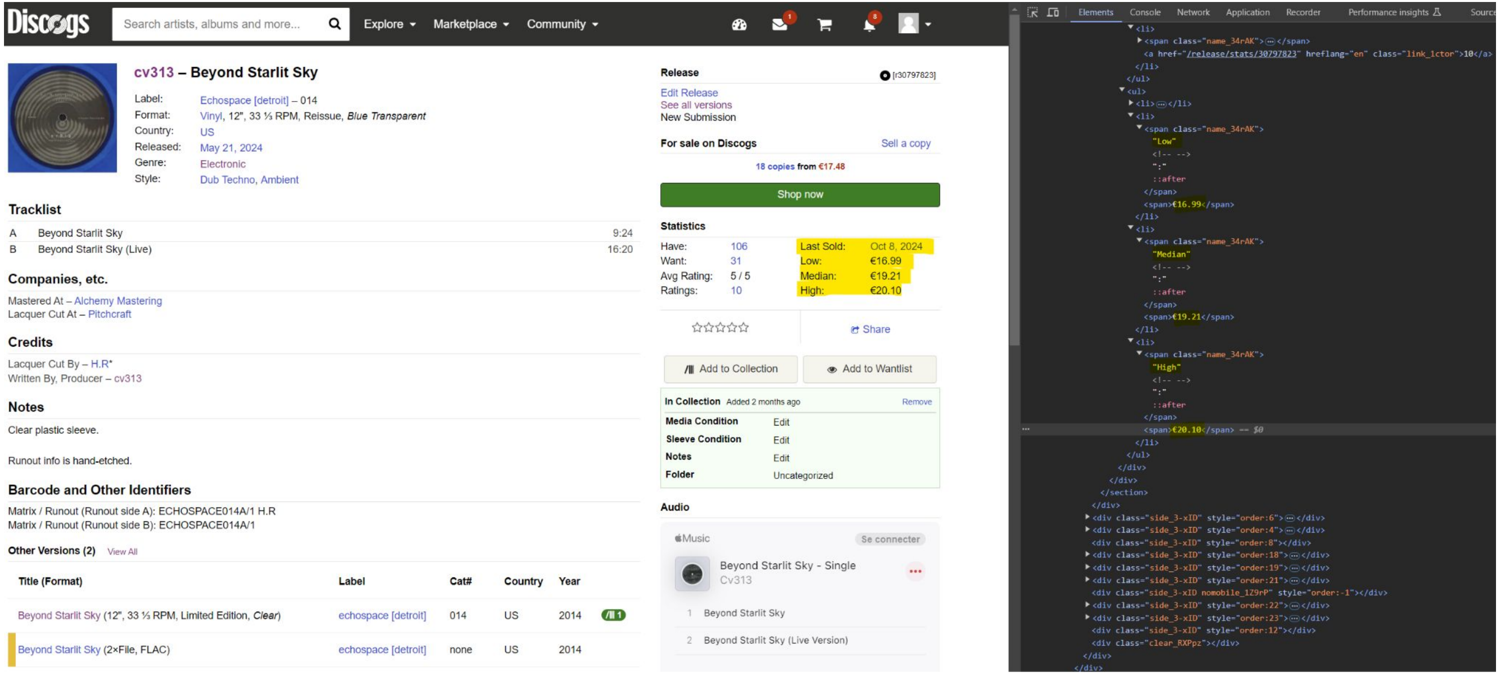
https://www.discogs.com/release/30797823-cv313-Beyond-Starlit-Sky

We will then return the selected album informations :

```
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', []))}")
```

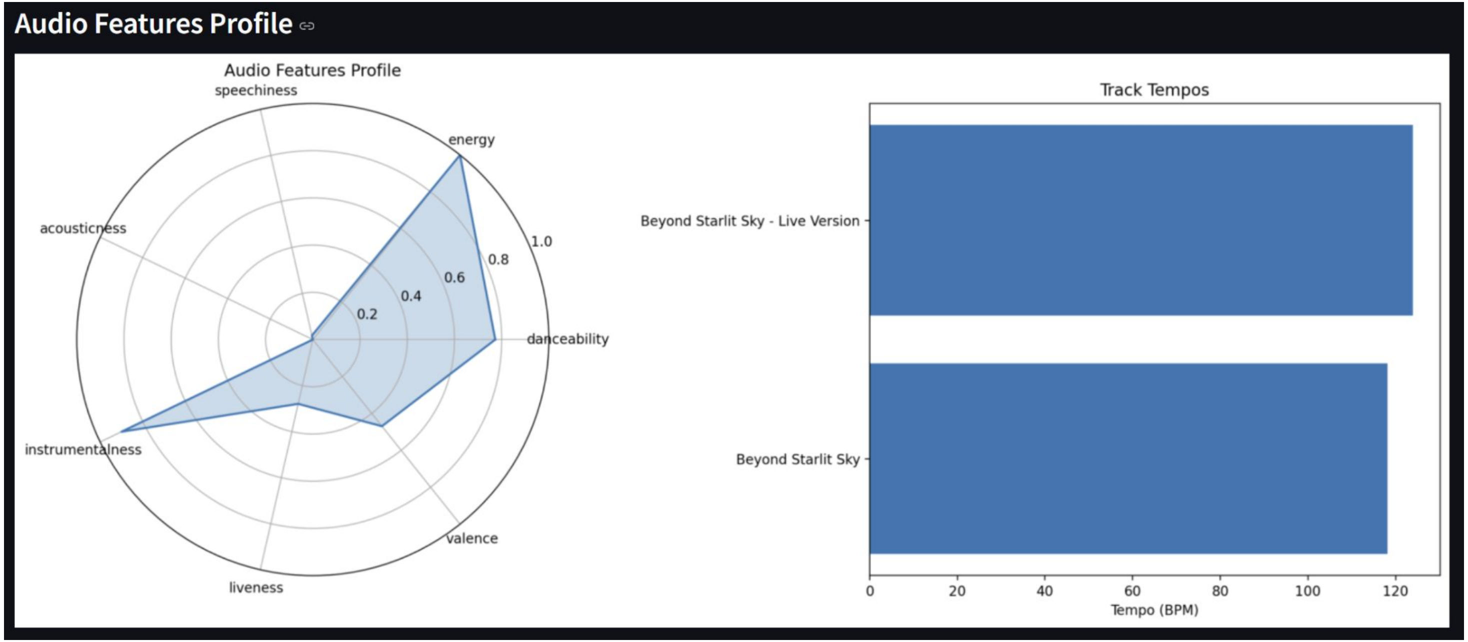
Step 1.5 : Discogs price scraping

Discogs API unfortunately prevents fetching any metadata related to records prices. We need to resort to price scraping in order to retrieve these statistics. Selenium is used here, rather than BeautifulSoup, as Discogs' architecture prevents from fetching informations from its HTML.



Step 2 : Data processing & Audio Profile

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.



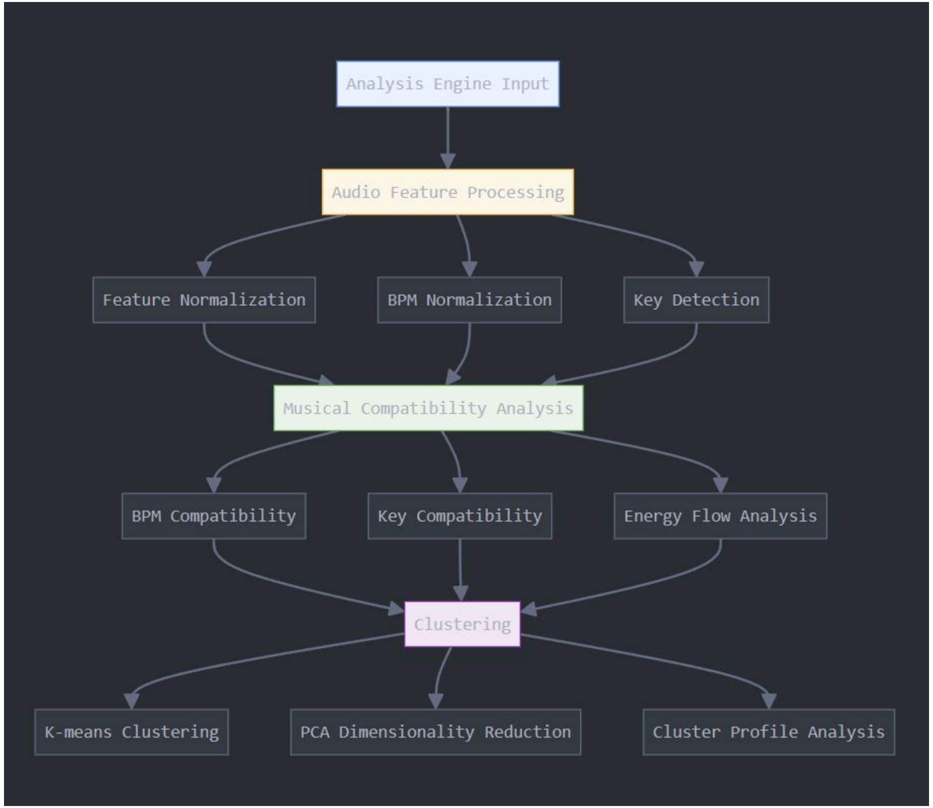
Step 3 : Analysis Engine

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 recommandations.
- 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)



Step 4 : Recommendation Engine

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

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

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

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

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

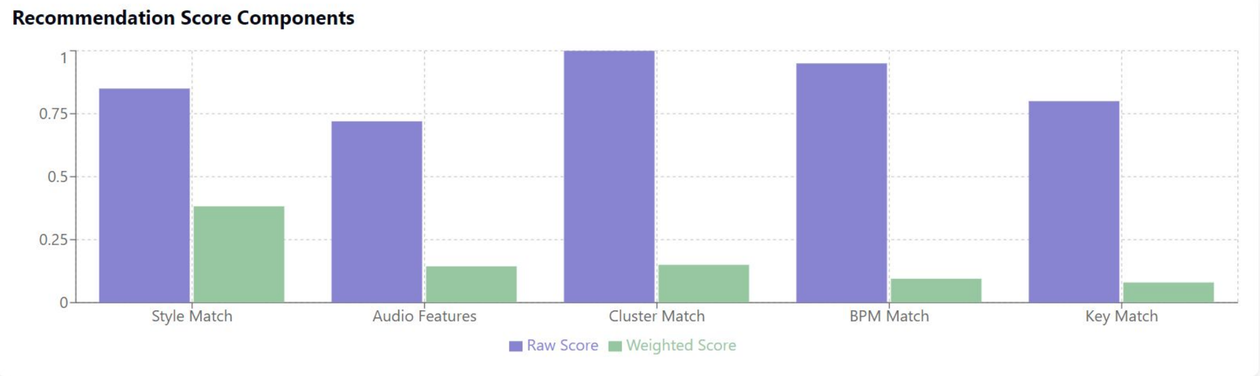
Step 4 : Recommendation Engine cont'd

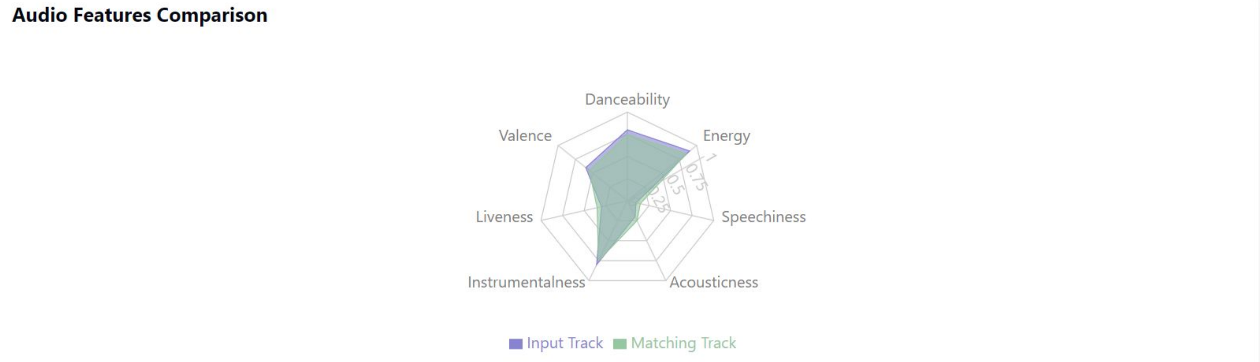
Once the weighted calculated score is done for all the tracks, we compare them to our input albums track(s).

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



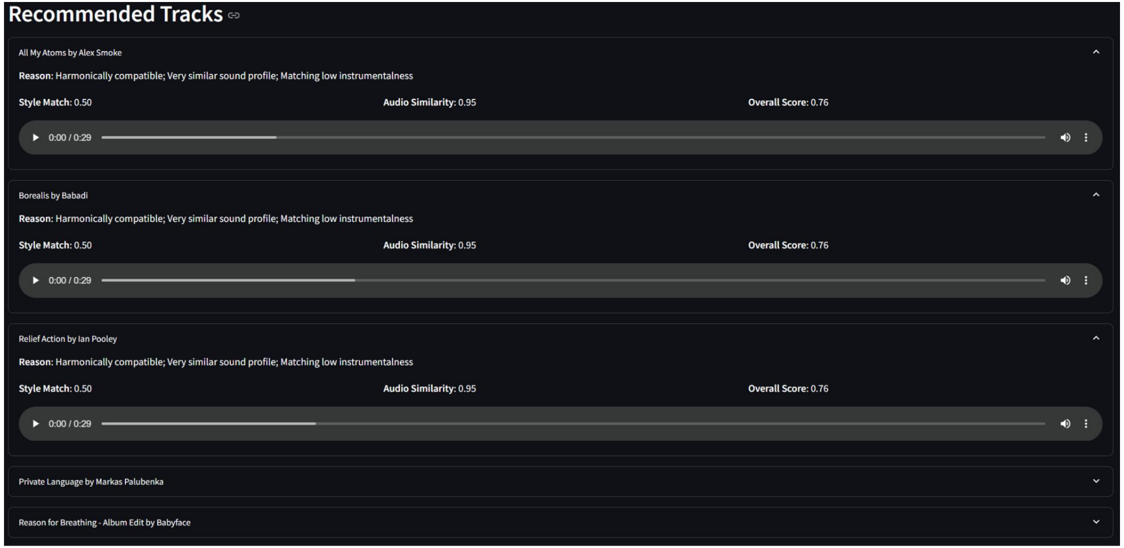


Step 5 : Streamlit as the UI renderer

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 informations (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)

Example of recommended tracks below album informations :



Moving forward : improvements & fixes

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 few ways to do this would be to improve multi-processing, further reducing the CSV files size, introducing caching for repeated queries…

Thank you