Goldsmiths, University of London

NextTrack: A music recommendation API

Preliminary Report

Valentin Radulescu
June 2025

Introduction

This report details the ideation, research and implementation of a music recommendation system titled *NextTrack,* which suggests which track should be played next based on recent listening behaviour and user-defined preferences. Unlike popular music services which rely heavily on user profiling and data collection, this project will operate as a stateless REST API which enforces a privacy-friendly approach. This means that the system does not store user data, track users or require the creation of persistent user profiles.

The project is based on the "*NextTrack: A music recommendation API*" template which builds upon the information taught in the "*CM3035 Advanced Web Development*" module. The implementation will consist of a back-end API developed in Django and a front-end interface built with React that will demonstrate the functionality and allow for user testing. The recommendation engine itself will be powered by a combination of rule-based filtering and a basic machine learning model applied to open music metadata.

The core motivation behind NextTrack is to explore an alternative to how popular music services handle track recommendations. Platforms like Spotify and YouTube Music typically rely heavily on user data and give opaque recommendations with very limited user control. While the results are often personalized, there's also a lack of transparency going on. Users are not typically aware of how recommendations are made and have limited ability to influence the logic behind them. In the long-term, repeated exposure to certain favourite tracks tends to create a sort of bubble that users get stuck in, where new music discovery becomes increasingly difficult. This problem is exacerbated when multiple users share the same account, introducing 'preference pollution': tracks liked by the other user will keep getting recommended seemingly at random (e.g. nursery songs in a household with a baby, guilty favourites, etc.).

NextTrack aims to address these challenges by offering a transparent, user-controlled, and stateless recommendation experience. The system doesn't rely on implicit data about the user, instead the user provides at front a list of liked tracks and sets optional filters (genre, mood, willingness-to-explore). The API then responds with one or more new tracks based solely on these inputs without needing to track user history or behaviour beyond the current session. Parameters such as randomness or willingness-to-explore will determine how predictable or adventurous the recommendations should be.

Literature Review

Datasets

For this project I researched online music databases to evaluate which would map best to the project requirements.

Million Song Dataset[1]

A very popular dataset that's referenced in many music recommendation papers. It is also largely out of date with the last release being in 2012. A full copy of the dataset (~300gb) isn't available for download anymore from the official website.

The dataset provides rich metadata for each track, having 55 fields which cover: tempo, pitch, tags, terms, external identifiers, artist location, release year, etc. This level of detail would have made it a prime candidate if it wasn't for the age of the data.

Spotify Million Song Dataset[2]

This is a small dataset found on Kaggle with records for around 57k songs. There's not enough metadata to do proper matching, it only covering artist, title, lyrics but it could be used to build a proof-of-concept by using some Natural Language Processing (NLP) technique.

To use this dataset for the final implementation, at minimum it should be amended with genre and release year.

FMA (Free Music Archive)[3]

Another popular dataset, originally presented at the ISMIR conference in 2017 and still relevant, having 2.3k stars on GitHub. It covers a limited number of songs, around 100k, and out of date with the latest release also being in 2017.

It provides rich song metadata: extracted audio features (tempo, key loudness), tags, genre, etc. but its age makes using it in this project questionable.

Spotify API

Spotify is potentially the best source to get information about music and its API provides rich metadata for each track. However, access is rate to that API is limited to around 100 requests per hour[4]. This makes it hard to build any sort of recommendation service on top of Spotify, it wouldn't be possible to serve responses to a meaningful number of users because of the rate limit.

---

[1] http://millionsongdataset.com/
[2] https://www.kaggle.com/datasets/notshrirang/spotify-million-song-dataset
[3] https://github.com/mdeff/fma
[4] https://developer.spotify.com/documentation/web-api/concepts/rate-limits

Discogs and MusicBrainz

Both are massive open music databases that store information on millions of songs. The data covers release information, artist, relationships between songs, etc. They're both prime candidates for use in this project but their sheer size and the complexity of the data requires a lot of forethought into how to properly extract information from them. A recommender system cannot realistically parse millions of entries when producing a result.

Design

Validation

Developing an API poses an interesting challenge when it comes to its validation, especially one where a successful response isn't necessarily the best result. When it comes to music tastes and what one considers a "good recommendation", it's likely that the perceived quality of the API response will vary greatly from user to user. To address these potential issues, the validation will be done in three successive phases.

Initially validation will be done in the form of unit tests. If the API returns a predictable or accurate result, we'll consider the implementation to be valid. Quality of the response cannot be validated at this point, but we can check for quantitative attributes, ex: if the recommended song is in the same genre as the previous one.

As the project approaches the halfway point, validation will be extended by judging the responses based on the developer's own tastes and comparing the results against what other platforms recommend (Spotify, YouTube Music). It's important to have the front-end implementation up to date throughout this process so that it can consume the API without issues and allow for audio playback. This allows for seamless testing.

User testing will be done in the final phase of the project. I will ask 3-5 friends and family to use the app for a few days and fill out a questionnaire at the end. The questions will target the quality of the experience, potential improvements, bugs and missing features. The feedback will then be addressed in the polishing phase week.

Development Plan

Development time: June 9th - September 8th 2025, 13 Weeks

1. Week 1 – Build Prototype & Write Draft Report
2. Week 2 – Database design
3. Week 3 – Identify meaningful features for recommendation algorithm
4. Week 4 – Investigate different types of recommendation algorithms
5. Week 5 – Expose algorithm as a REST API
    - Identify potential users to test the app ahead of time

6. Week 6 – Implement basic interface to consume API
7. Week 7 – Extend interface to support music playback
8. Week 8 – Implement ways for users to save listening history and favorite tracks
    - Send out user survey/questionnaire
9. Week 9 – Polish the app
    - Implement changes according to user survey/questionnaire
10. Week 10 – Buffer time in case development drags
11. Week 11 – Begin writing final report
12. Week 12 – Finalize report
13. Week 13 – Submission

Feature Prototype

A prototype was developed to validate the assumption that a music recommender system could feasibly be built as a Django application. The prototype should contain all the components required so that it can take input from the user, do some computation on the server side and return a result, all modelled as a REST API.

My lack of familiarity with the domain area of music recommendation as well as the perceived difficulty of building such a system as outlined by the literature review led me to focus more on ensuring that the final product is something that produces a result that at first glance looks valid rather than the best or most accurate one.

The first step in building the prototype was to identify a dataset either containing song metadata or one from which metadata could be extracted. This metadata could then be fed into an algorithm that finds similarities between entries. However, finding a suitable dataset wasn't a trivial task as most of those available online have caveats that made them unsuitable for implementing a prototype. The ones that I reviewed fell into one of the following categories:

1. Data is out of date, dataset does not go up to the current year or covers a short timeframe: Million Song Dataset, Free Music Archive (FMA).
2. Data format is too complex to easily adapt to a prototype (can't be easily mapped to JSON or relational DB tables): Discogs, MusicBrainz.
3. The data is given by an API that's rate limited so large amounts of data cannot be downloaded in a reasonable timeframe: Spotify.

In the end I opted to use a much simpler dataset from Kaggle, "Spotify Million Song Dataset"[5] which, despite its name, only covers ~57k songs and provides a minimal set of features for each one: artist, title, URL, lyrics. However, the data spans a good array of genres and artists, it's relatively recent (last updated in November 2022) and has a

[5] https://www.kaggle.com/datasets/notshrirang/spotify-million-song-dataset

simple structure in a format that can easily be parsed (CSV). More importantly, because it provides rich lyrical data, it's easy to see how songs could be compared by the contents of these lyrics. For a prototype, this is a quick and easy way to validate an idea without having to explore and experiment with the more complex datasets out there.

The next step in developing the prototype was implement how the song comparison would work. A popular way to do this with textual data is to use a combination of Term Frequency-Inverse Document Frequency (TF-IDF) to map the words in each text to sparse frequency vectors and then compare them using cosine similarity (Salton & Buckley, 1988). This can be achieved with a few lines of code as shown in the figure below.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# generate a matrix of word vectors for each term in the input corpus
vectorizer = TfidfVectorizer(stop_words='english', max_features=10000)
tfidf_matrix = vectorizer.fit_transform(input_corpus)

# for a certain song (song_id), find 10 songs similar to it
query_vec = tfidf_matrix[song_id].reshape(1, -1)
similarities = cosine_similarity(query_vec, tfidf_matrix).flatten()
top_indices = np.argpartition(similarities, -11)[-11:]
```

As an example, if the input song is "Master of Puppets" by Metallica, the recommended songs would be the ones in the figure below. We can see that most of them are in the rock genre from artists like Nightwish, Black Sabbath and Def Leppard despite the selection process not taking genre into account and not knowing anything about the artists. It may be a case that many rock songs use similar wording, specifically the term "master" might be more prevalent in this genre of music.

	artist	song	text
12817	Metallica	Master Of Disaster	you should run away when you see me play im t...
45821	Nightwish	Wishmaster	master apprentice heartborne 7th seeker war...
30538	Dream Theater	In Presence Of Enemies Pt. 2	welcome tired pilgrim into the circle we hav...
29546	Depeche Mode	Master And Servant	theres a new game we like to play you see a ...
46024	Nirvana	Downer	portray sincerity act out of loyalty defend ...
30298	Doobie Brothers	The Master	just dont know why i keep on tryin must be a ...
29305	Def Leppard	Answer To The Master	when the night time unfolds and the memories t...
25613	Black Sabbath	Master Of Insanity	look all around cant you open your eyes voic...
15079	Oscar Hammerstein	My Lord And Master	spoken the king is pleased he is pleased with...

The final step is to implement an API that serves this information through its endpoints. This is a matter of creating a Django project, creating a model for the data (with title, artist, lyrics fields) and then processing and serving that data through an API endpoint using Django REST framework (DRF).

Storing the data on-disk needs to be handled carefully. The vectors that the TF-IDF model generates are sparse vectors meaning that most of their values are 0, at the same time each vector stores 10k values (features). The size of the data as well as its structure makes it unsuitable for storage in a traditional relational database. To get around this, we'll store the TF-IDF values in a separate file named "tfidf_matrix.npy" and the song metadata in an SQLite database, making sure we can match each song to each feature vector through a shared id.

As an example, the initial CSV containing entries for 57650 songs, including their lyrics, only takes up 71 MB. The TF-IDF data, on the other hand, requires 2.85GB of space. We can see how the current approach to comparing songs could quickly become very resource intensive as the number of songs increases.

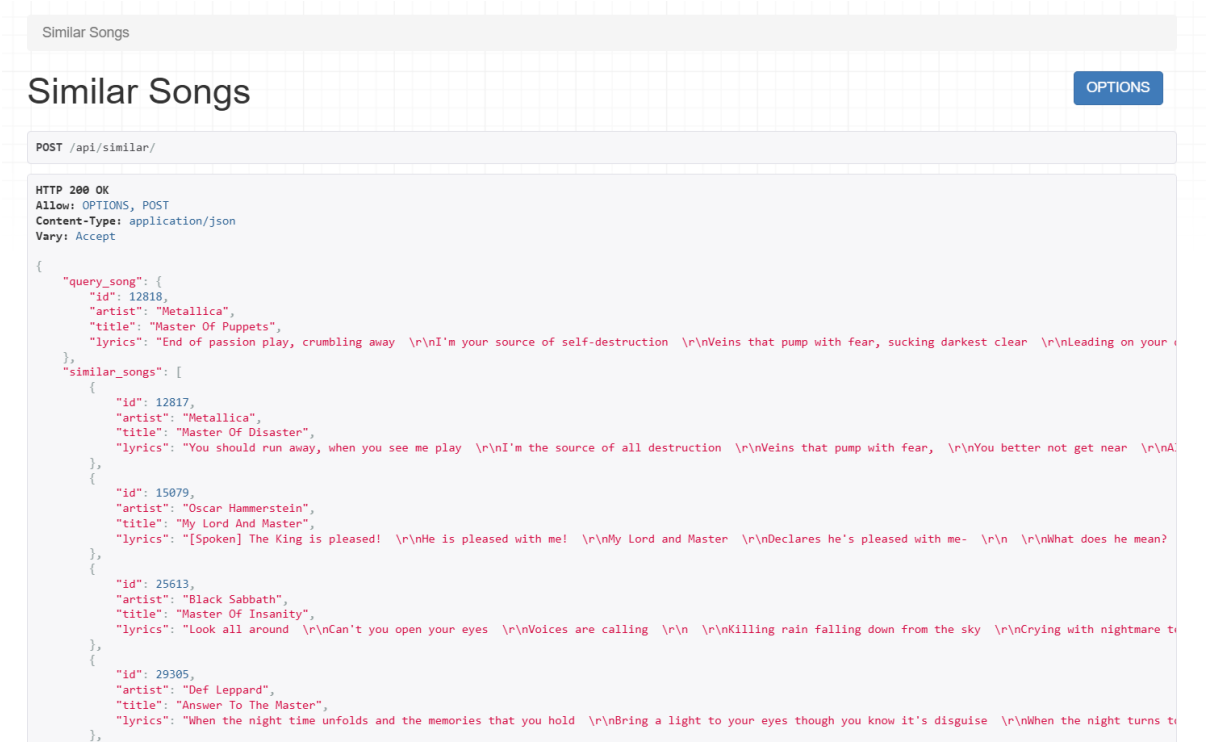

Figure 1 - API response for finding similar songs

Evaluation

The final version of the prototype serves to validate the idea that a song recommendation system could be implemented as an API, but it also outlines a good number of challenges that we'll have to deal with.

The sheer size of the data produced by the TF-IDF algorithm makes it ill-suited for our needs. It is possible to optimize this quite a lot by pre-computing and storing only the top-k songs similar to each song according to cosine similarity. In this case the feature vectors would only be used temporarily during the initial computation. However, this also means that it would take a considerable amount of time to build the database initially as well as when new songs are added.

Another issue with comparing TF-IDF vectors is that we're not dealing exclusively with text. Songs also have audio characteristics, pitch, volume, feel, etc. that aren't necessarily captured in the lyrics. While recommending songs with similar words does work to a certain extent, as seen with "Master of Puppets", it's not a prefect solution and other genres could easily slip in. Going forward lyrics should only weigh a small amount in the recommendation process.

For the final project's data, I will replace the prototype dataset with one that provides multiple metadata features. Potential candidates could be the well-known Discogs and MusicBrainz databases but using a different dataset from Kaggle could also be an option. For example, the "Spotify_1Million_Tracks" dataset[6] has 20 features including tempo, liveliness, energy, key, etc. for 1 million records, making it a prime candidate for training a machine learning model on.

[6] https://www.kaggle.com/datasets/amitanshjoshi/spotify-1million-tracks

References

List of online music databases. (2025, May 30). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/List_of_online_music_databases

Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 513–523. doi:10.1016/0306-4573(88)90021-0