ECOLE POLYTECHNIQUE DE BRUXELLES

2025-26

INFO-H600

Spotify Million Playlists Challenge

DELCHAMBRE Thomas

ESSAKILI Ouaïl

GUELENNE Zoé

MAAMER Ryan

MARCHESI Paul

NGUYEN Thanh

Professor

SACHARIDIS Dimitris

Disclaimer : *Artificial intelligence was used responsibly throughout this project. It assisted in resolving coding issues encountered during the analysis, and improving the clarity of this report. All modelling decisions, computations, and interpretations were carried out with care, and AI-generated suggestions were cross-checked before being integrated into the final report.*

2025

Contents

1 Introduction **1**

2 Q1: Approach to handling big dataset **2**

2.1 Motivation and Overall Objective 2

2.2 Preprocessing the Million Playlist Dataset 2

2.3 Initial Dask-Based Aggregation 3

2.4 Streaming External-Merge Construction of the Vector DB 3

2.5 Complexity and Trade-Offs 5

2.6 Summary: Dask vs. Streaming external merge comparison 6

3 Q2: Exploratory Data Analysis **7**

3.1 Dataset Overview 7

3.2 Playlist Structure 7

3.3 Playlist Diversity 8

3.4 Popularity Bias and Top Tracks 8

3.5 Playlist Name Semantics 9

3.6 Playlist Track Uniqueness and Repetition 10

3.7 Distribution of Duplicate Tracks per Playlist 11

3.8 Duplicate Track Statistics & Most Repetitive Playlists 11

3.9 Conclusion 12

4 Q3: Definition of similarity between tracks **14**

4.1 Definition 14

4.2 Application 16

4.2.1 Similarity Track Distribution Across the Database 16

5 Q4: Playlist Similarity **17**

5.1 Problem Statement . . . 17

5.2 Method 1 & Method 2 . . . 18

5.2.1 First definition . . . 18

5.2.2 Second definition . . . 18

5.2.3 Application . . . 19

5.3 Method 3: topics based using Latent Dirichlet Allocation . . . 22

5.3.1 Methodology . . . 22

5.3.2 Example Results . . . 25

5.3.3 Complexity considerations . . . 26

5.3.4 Discussion and Limitations . . . 26

6 Q5: Resolution of the "playlist continuation" challenge **27**

6.1 Method 1: Topics Based Recommendation Method . . . 27

6.1.1 Example of Playlist Continuation . . . 27

6.1.2 Evaluation of the Recommendation Method . . . 28

6.2 Other considerations and potential improvements . . . 29

6.2.1 Increasing the LDA Training Set Size . . . 29

6.2.2 Mitigating Popularity Bias in Playlist-Based Similarity Measures . . . 30

6.2.3 Method 2: Combining Topic-Based Recommendation with Dot-Product Scoring 30

7 Appendices **32**

7.1 Appendix LDA . . . 32

7.1.1 Hyperparameters . . . 32

7.1.2 Implementation: `get_similar_playlists` . . . 32

7.1.3 Advantages of this definition . . . 33

1 Introduction

The *Million Playlist Dataset Challenge* provides a large-scale benchmark for studying music recommendation systems. It consists of one million playlists, each containing metadata and a set of tracks, themselves associated with descriptive attributes.

While this dataset offers valuable opportunities for developing similarity-based recommendation methods, its substantial size (approximately 35 GB) poses significant challenges in terms of data processing and computation. The objective of this project is to address these challenges and explore scalable approaches for analysing the dataset and designing recommendation strategies.

First, the dataset must be ingested and processed efficiently, which motivates the exploration of scalable strategies for handling large volumes of data. This step was carried out using a combination of Dask-based preprocessing attempts and a disk-streaming approach designed to avoid exceeding RAM limitations.

Second, statistical and aggregate analyses of playlists and tracks are performed to better understand the structure of the dataset. These analyses provide a global view of playlist characteristics, as well as track and artist popularity. They also reveal naming patterns and thematic trends across playlists, which will later support our discussion of the LDA-based topic modelling approach.

Third, similarity measures for both tracks and playlists are defined and implemented. Track similarity is derived from the assumption that two tracks are similar if they frequently co-occur across playlists, while playlist similarity is first defined using symmetric overlap-based measures on their constituent tracks. In addition, a topic-based approach using Latent Dirichlet Allocation (LDA) is introduced, in which playlists are represented by latent semantic topic distributions learned from their textual content (titles, descriptions, track names and artist names).

Finally, this latent playlist representation is leveraged to address the playlist continuation task: a track is recommended for a given playlist if it frequently appears in playlists that are highly similar to it within the learned topic space.

2 Q1: Approach to handling big dataset

2.1 Motivation and Overall Objective

The motivation behind the data-processing pipeline arises from the chosen representation for measuring similarity between tracks (cf.4.1). The objective is to construct a normalised vector space in which each track is associated with a unique vector, and the similarity between two tracks is defined through the scalar product of their respective vectors.

The representation selected for this purpose relies on **playlist co-occurrence**: the vector of a track consists of the list of playlist identifiers (`pid`) in which that track appears. Under this model, two tracks are considered similar if they frequently co-occur across playlists. Constructing such a "vector database" therefore requires **aggregating, for each track, the complete set of playlist IDs in which it is present**. Each track must appear exactly once in the final database, accompanied by a second column containing the list of all associated `pids`.

2.2 Preprocessing the Million Playlist Dataset

The *Million Playlist Dataset* contains one million playlists distributed across 1,000 `JSON` files, each describing 1,000 playlists. Because the `JSON` structure is large and deeply nested, an initial preprocessing stage consisted of flattening the data and serialising it into efficient `Parquet` tables. For each `JSON` file, both a playlist table and a track table were extracted, the latter containing one row per (`track_id`, `pid`) pair. A track appearing in 500 playlists therefore occurs 500 times in this table. These `Parquet` files served as the basis for the construction of track vectors.

2.3 Initial Dask-Based Aggregation

Principle

An initial attempt relied on Dask to aggregate the track table by `track_id`. Conceptually, this aggregation should:

- group all rows corresponding to a given `track_id`;
- collect the associated `pids` into a single list per track;
- output one row per track containing its full playlist vector.

For small subsets of playlists, Dask produced correct results, **although the computation was already relatively slow** because the aggregated vectors were not persisted and each query for a track required recomputing the entire `groupby` operation.

Scalability Issues

As the number of playlists increased, the approach rapidly became infeasible. The aggregated vectors became incomplete: tracks expected to appear in more than fifty thousand playlists were reported in only a few hundred of them. In addition, the time required to compute the aggregation, and even to retrieve a single track vector after the aggregation had been defined, increased to nearly forty minutes, **rendering the method impractical for large portions of the dataset.**

The difficulty arises from the aggregation step itself. When Dask is instructed to aggregate by `track_id`, the operation is not limited to combining values within individual partitions; it requires merging all partitions into a single global structure. As a consequence, all occurrences of the same `track_id` must be brought together in a data structure large enough to handle millions of tracks and thus millions of vectors. As the number of playlists grows, the size of this combined structure exceeds available RAM memory, leading to incomplete results and excessive execution times. These limitations indicate that,given the similarity-based framework and the level of Dask expertise available during the project, a Dask-only solution did not appear to be the most practical option for processing the full dataset.

2.4 Streaming External-Merge Construction of the Vector DB

After the Dask-based approach proved unsuitable, a second strategy was adopted in which **the track vector database is constructed incrementally and written directly to disk**, thereby avoiding

the need to store large structures in RAM. A complete view of the pipeline is schematized in fig 2.1 (See Appendix **??** for a summary table of the two methods comparison).

Step 1: Per-File Track–Playlist Extraction

The 1,000 dataset files, each describing 1,000 playlists, were used. For each file, a tracks database was created by listing all tracks contained in its playlists and associating each track with the corresponding playlist identifier (`pid`). However, as one track can appear several times across files if it is associated with several playlists, a global aggregation was still required.

Step 2: Sorting Intermediate Files by `track_id`

To achieve this, all 1,000 intermediate aggregated files were first sorted by `track_id`. Track identifiers are long strings and `pandas` provides a built-in string comparison mechanism that was used for this ordering step. This sorting step is crucial: once all files are individually sorted by `track_id`, they can be merged efficiently.

Step 3: Streaming External Merge and On-the-Fly Aggregation

Once sorted, the files were merged using a streaming external-merge procedure: a small number of rows from each file (15) were loaded, merged according to `track_id`, and written to the new vector database. Whenever identical identifiers appeared, their partial `pid`-lists were concatenated. Each time a line was written to the output, it was replaced in memory by the next available line from the corresponding intermediate file.

Resulting Data Layout and Index

This process gradually produced **a final collection of approximately 150 `Parquet` files, each covering a contiguous interval of track identifiers.** For each file, the smallest and largest `track_id` were recorded, forming an index that allows an easy identification of the file containing any given track. Only one of these 150 files must therefore be loaded into memory to retrieve a track vector, which significantly reduces RAM usage if computation is necessary.

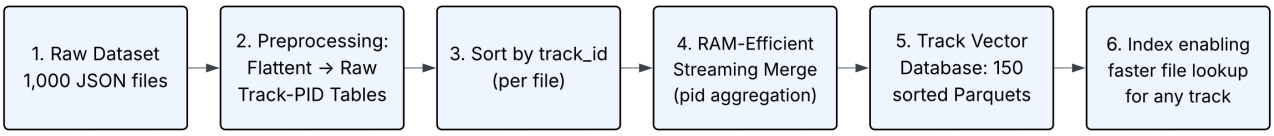


Figure 2.1: Full Pipeline for Building the Track Vector Database

2.5 Complexity and Trade-Offs

Construction Cost

This construction requires substantial preprocessing time due to the sorting and merging steps, giving the pipeline an overall complexity of $O(n \log n)$. However, once completed, the built index makes vector retrieval much faster, with a complexity of $O(\frac{n_{\text{tracks}}}{150})$, where n_{tracks} is the number of tracks (a bit more than two million).

Lookup Strategies for a Single Track

The choice of using 150 output files was arbitrary, although it provided satisfactory performance in practice. Nonetheless, this number can be discussed from a complexity standpoint.

Locating the correct file is $O(1)$ and retrieving the corresponding row can be done in two ways:

1. loading the whole file into memory and ask `panda` to retrieve the correct row, with complexity $O(\frac{n_{\text{tracks}}}{150})$;
2. performing an efficient research within the sorted file, with a complexity of $O(\log(\frac{n_{\text{tracks}}}{150}))$.

The first option was retained for its implementation simplicity.
Regarding the first option, multiplying the number of files by k would reduce lookup complexity by a factor k. However, the computational time was judged sufficient enough for the actual application. Although increasing the number of output files by a factor k would reduce lookup complexity by the same factor, one might imagine pushing this idea to the extreme by choosing $k = \frac{n_{tracks}}{file}$, so that each track would have its own indexed location. Such a structure would allow constant-time access $O(1)$. However, with the available hardware, storing such a fine-grained index is not feasible because it would require significantly more RAM. The current index is implemented as simple Python dictionaries and although more sophisticated solutions such as hashing schemes or deterministic mapping rules could overcome these limitations, they were not pursued.

Using k times more files in the second strategy would reduce the lookup complexity from $O(\log(\frac{n_{tracks}}{file}))$ to $O(log(\frac{n_{tracks}}{file}) - log(k))$, which represents only a limited improvement.

Retrieving Multiple Track Vectors

When retrieving n track vectors, a hashing-based index would again provide the optimal complexity $O(n)$. In the implemented approach, however, the process begins by determining which

output files contain the n requested track identifiers. If the n tracks are distributed across n_{files} files, each file is loaded once, and the relevant vectors are extracted. The resulting complexity is $O(n \cdot n_{\text{files}} \cdot (n_{\text{tracks/files}}))$.

For sufficiently large n, increasing the number of files by a factor k improves performance by a factor between 1 and k, but the gains are not linear, which justifies the decision not to use a larger number of files.

A partial implementation of this retrieval mechanism using Dask was also explored. However, due to limited expertise with the framework, its complexity was not analysed in detail. A comparison of the practical performance of the two methods is provided later in the report (cf.2.6).

2.6 Summary: Dask vs. Streaming external merge comparison

Table 2.1 summarises the main differences between the initial Dask-based solution and the final streaming external-merge approach.

	Dask-based aggregation	**Streaming external merge**
Aggregation mechanism	Global `groupby` on `track_id` across all partitions	Per-file sorting followed by streaming merge on `track_id`
Memory usage	High; frequent tracks require large global structures	Low; only a small number of lines from each file in memory
Scalability	Fails on the full million-playlist dataset (incomplete vectors, very long runtimes)	Designed to scale to the full dataset under RAM constraints
Preprocessing cost	Moderate but grows quickly with data size	Higher upfront cost ($\mathcal{O}(n \log n)$) but one-time
Lookup time	Potentially involves recomputing or reusing heavy aggregations	Direct file lookup via index, then local search within a single file

Table 2.1: Comparison between the initial Dask-based solution and the streaming external-merge approach.

3 Q2: Exploratory Data Analysis

3.1 Dataset Overview

The Million Playlist Dataset was analyzed to understand its scale and diversity. It contains a total of 1,000,000 playlists, encompassing 2,262,292 unique tracks and 287,742 unique artists. These results show both the massive size of the dataset and its rich musical variety.

3.2 Playlist Structure

The number of tracks and the total duration per playlist were examined to provide insight into user behavior. On average, playlists contain 66.3 tracks, with a median of 51, a minimum of 5, and a maximum of 376. In terms of duration, playlists last on average 259.7 minutes, with a median of 197.0 minutes, a minimum of 1.6 minutes, and a maximum of 10,584.6 minutes.

Figure 3.1: Distribution of Number of Tracks per Playlist

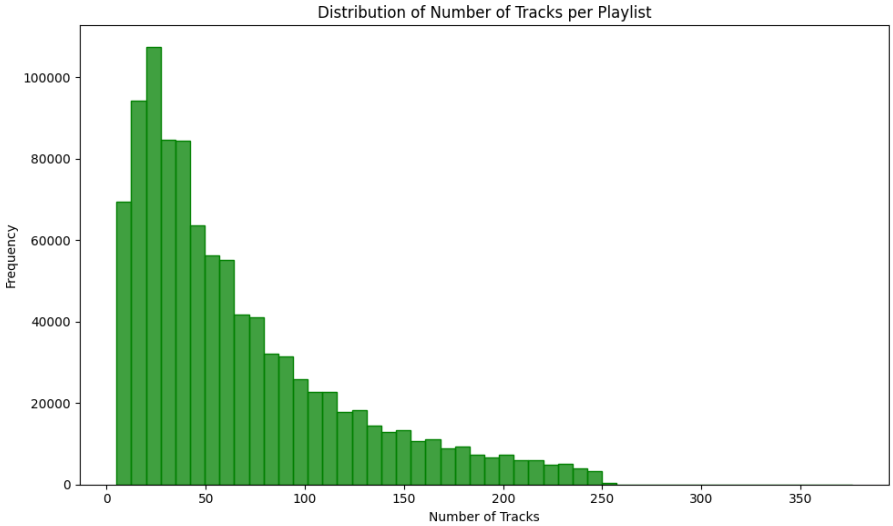
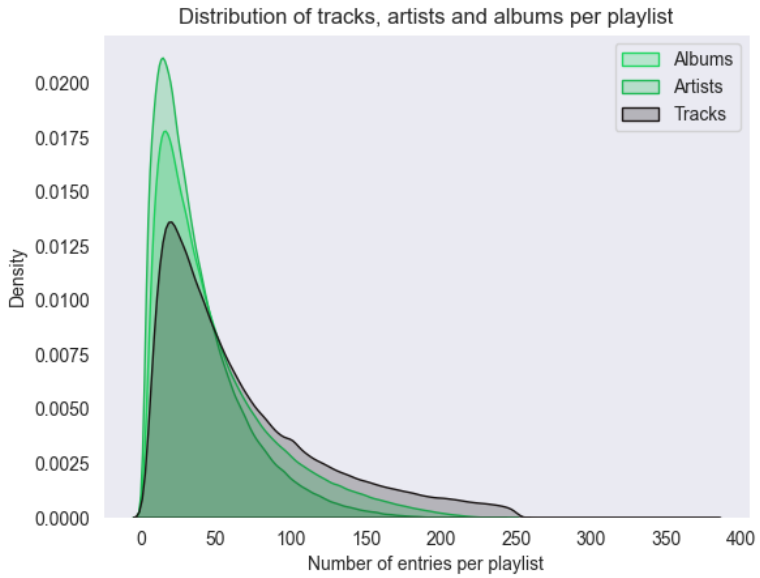


Figure 3.1 shows a right-skewed distribution. Most playlists contain between 30 and 70 tracks, while a few exceed 300 tracks, suggesting very long or highly curated playlists.

3.3 Playlist Diversity

Diversity was measured using the number of unique artists and albums per playlist. On average, playlists include 38.1 artists (median of 30) and 49.6 albums (median of 38). These values suggest that most playlists encompass a wide range of artists and albums, reflecting varied listening preferences.

Figure 3.2: Distribution of the number of Albums, Artists and Tracks per playlists



To further highlight this diversity, Figure 3.2 illustrates the composition of the dataset by showing the distribution of tracks, unique artists, and albums per playlist. The distributions are right-skewed, with a distinct peak indicating that the majority of playlists contain approximately 50 entries. This clustering around a moderate size suggests that typical users prefer medium-length playlists.

Furthermore, the close alignment between the track, artist, and album curves highlights a high level of diversity within these collections, as playlists tend not to be dominated by a single artist or album.

3.4 Popularity Bias and Top Tracks

The frequency of artist occurrences across playlists highlights a strong popularity bias. Drake appears 847,000 times, Kanye West 413,000 times, and Kendrick Lamar 354,000 times as shown Figure 3.3. Other frequently appearing artists include Rihanna, The Weeknd, Eminem, and Ed Sheeran. These trends confirm that a small group of mainstream artists dominates playlists, reflecting global listening habits and platform-driven exposure. Figure 3.4 shows that a few tracks

dominate playlist frequency, with *Closer*, *HUMBLE.*, and *One Dance* leading. This skewed distribution highlights strong popularity bias, where a handful of songs appear repeatedly across user playlists.

Figure 3.3: Top 10 Most Frequent Artists

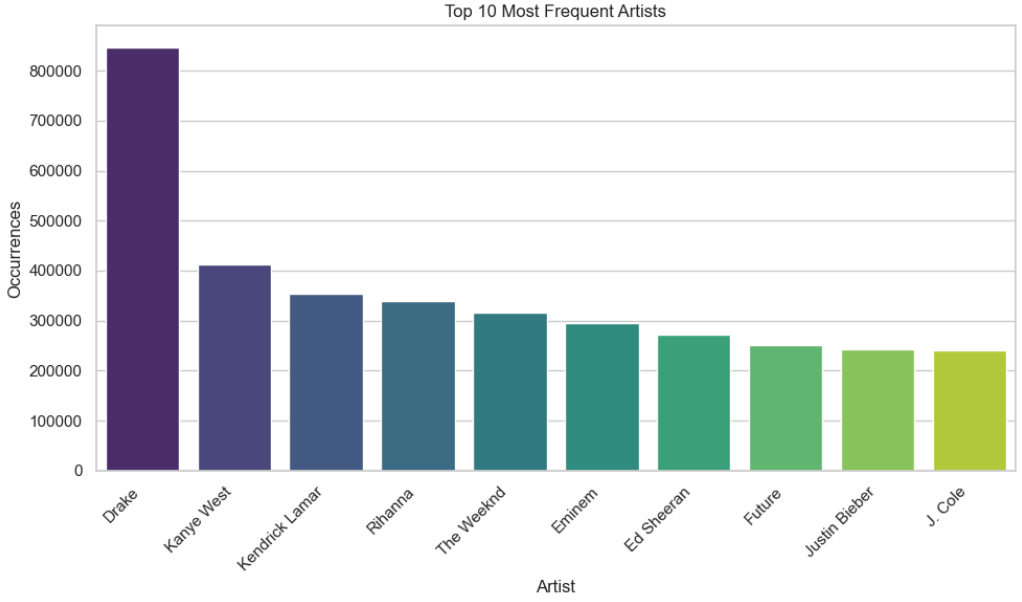
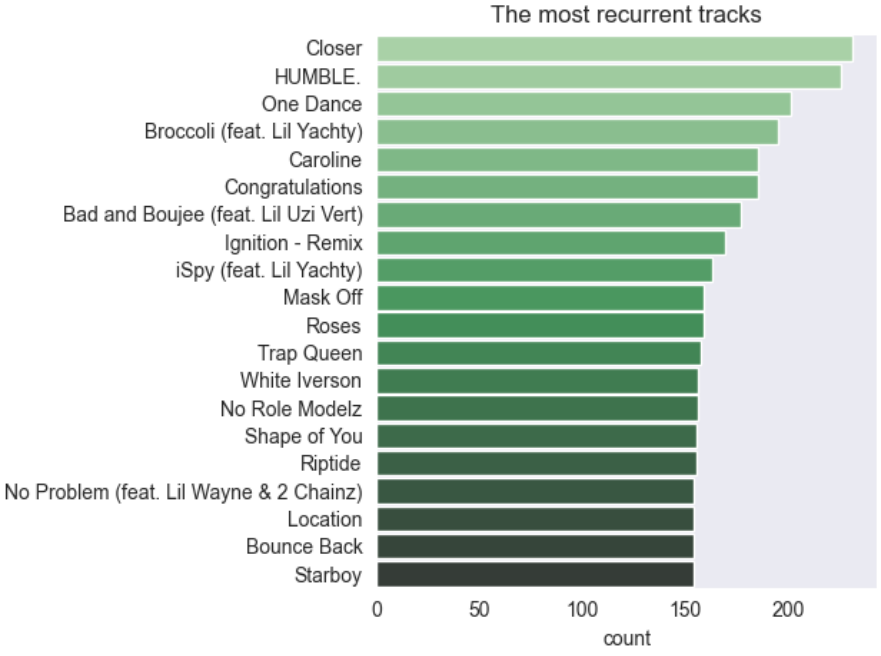


Figure 3.4: Top 20 Most Frequent Tracks



3.5 Playlist Name Semantics

The vocabulary used in playlist titles reveals common themes and naming patterns. Across all playlists, 1,435,706 words were identified, of which 10,736 are unique. The most frequent terms

include *music*, *country*, *summer*, *songs*, *chill*, *rock*, *party*, and *workout*.

Figure 3.5: WordCloud of playlist titles

Figure 3.6: Top 20 playlist names

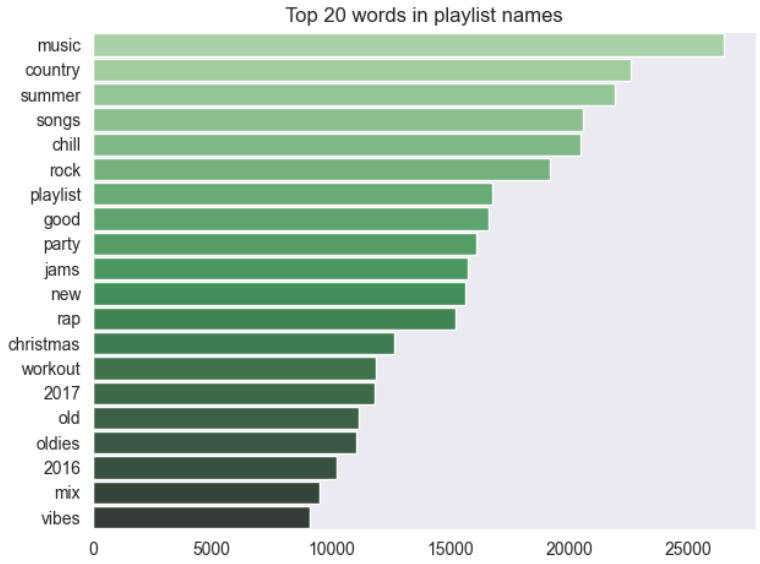


Figure 3.5 and figure 3.6 show that playlist names are rich in emotional, seasonal, and genre-related language, reflecting how users label playlists based on mood, activity, or musical style.

3.6 Playlist Track Uniqueness and Repetition

To evaluate redundancy within playlists, a uniqueness ratio was computed for each playlist, defined as the number of distinct tracks divided by the total number of tracks.

> A ratio equal to 1.0 indicates that all tracks are unique, whereas values below 1.0 reveal the presence of repeated tracks.

The results demonstrate that most playlists exhibit high uniqueness, with ratios clustered close to 1.0. Nevertheless, a substantial subset of playlists contains internal repetitions. Specifically,

292,851 playlists were identified with at least one duplicated track. Examples include playlists with ratios of 0.98, 0.95, or even 0.93, which indicate that several tracks were added multiple times within the same playlist.

The analysis reveals two key aspects of playlist behavior:

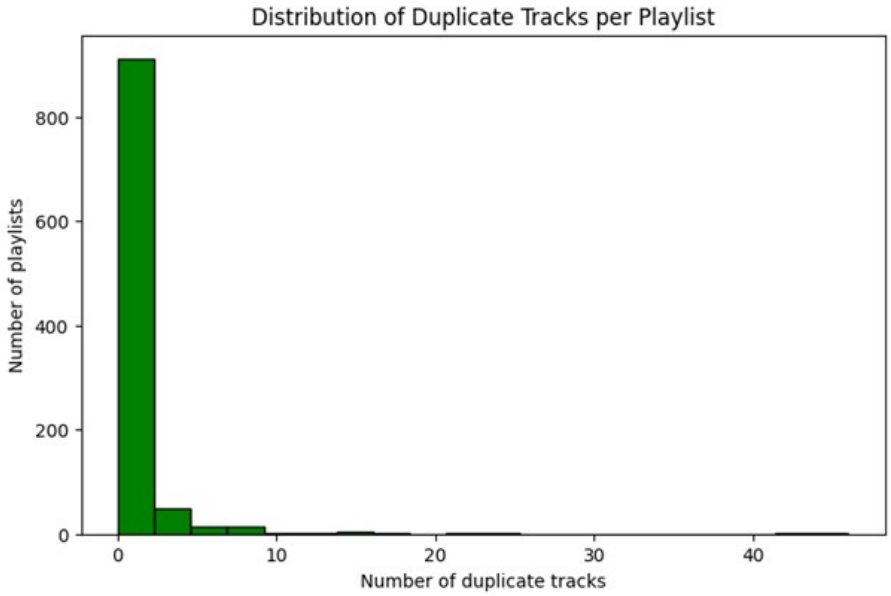
1. Most users curate playlists without redundancy.
2. A notable share repeat tracks, reinforcing popularity bias, as dominant songs appear not only across playlists but also within them.

Overall, uniqueness ratios highlight both widespread unique curation and the persistence of redundancy in many playlists.

3.7 Distribution of Duplicate Tracks per Playlist

Among 1,000 sampled playlists, 30% contained at least one duplicate, with some playlists showing up to 6 repeated tracks. The distribution is heavily skewed (Figure 3.7), most playlists are fully unique, but a notable minority includes repetitions. It reflects either user emphasis on favorite songs or lack of deduplication in playlist creation.

Figure 3.7: Distribution of Duplicate Tracks per Playlist

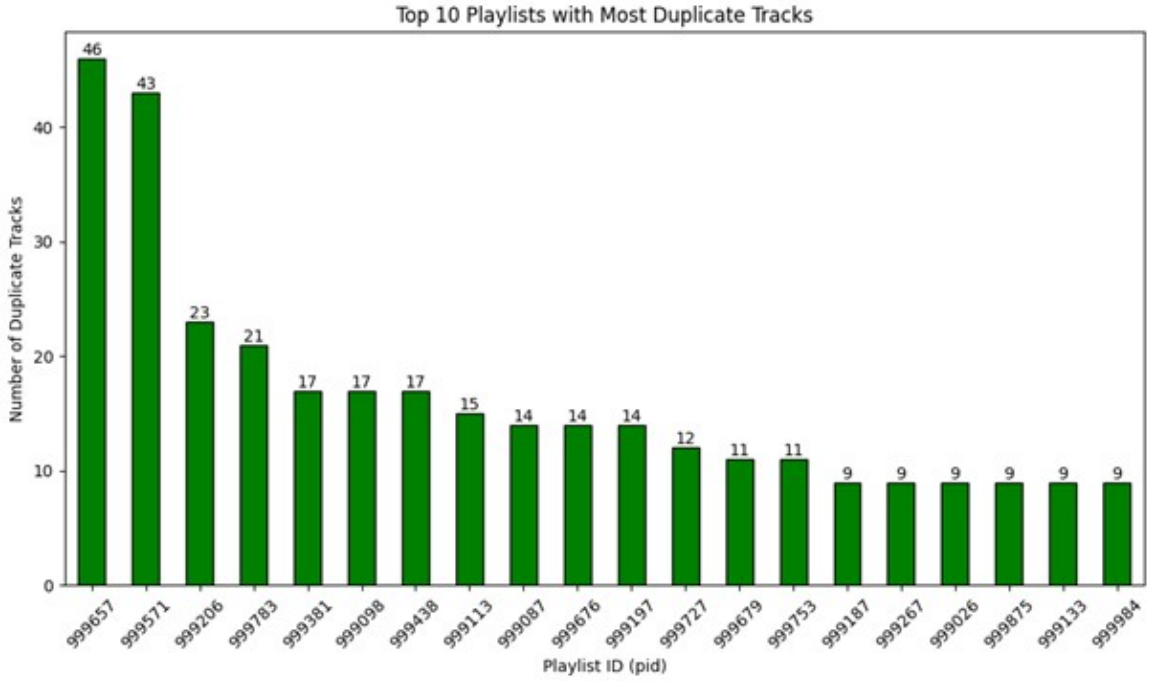


3.8 Duplicate Track Statistics & Most Repetitive Playlists

Across 1,000 sampled playlists, duplication is generally rare: the average number of repeated tracks is 0.89 (SD = 2.91), with a median of 0 and 25% of playlists containing at least one duplicate.

In total, 288 playlists (28.8%) exhibit repetition, usually involving only one or two tracks. However, extreme cases exist. For instance, the most repetitive playlist includes 46 duplicates (pid 999657), followed by others with 43, 23, and several between 9 and 21.

Figure 3.8: Top 10 Playlists with Most Duplicate Tracks



3.9 Conclusion

The exploratory data analysis of the Million Playlist Dataset highlights several key insights. The dataset is massive in scale, comprising one million playlists, millions of tracks, and hundreds of thousands of artists.

Playlist structures show strong variability: while most playlists contain between 30 and 70 tracks, some extreme outliers exceed 300 tracks and last thousands of minutes. Diversity analysis reveals that playlists typically include dozens of artists and albums, reflecting broad listening preferences.

A clear popularity bias is also evident, with mainstream figures such as Drake, Kanye West, and Kendrick Lamar dominating the dataset.

The semantics of playlist naming demonstrate that users often label playlists with emotional, seasonal, and activity-related terms, linking music consumption to mood and context.

Finally, the analysis of duplicate tracks reveals an important nuance in user behavior: although the vast majority of playlists are fully unique, a considerable subset still contains repeated tracks. This indicates that some users either intentionally emphasize favorite songs or unknowingly recre-

ate redundancy when curating music. Together, these findings show that while the dataset is predominantly diverse and well-structured, internal repetition remains a meaningful characteristic that reinforces existing popularity patterns within the platform.

4 Q3: Definition of similarity between tracks

4.1 Definition

As explained in Chapter 2, the similarity between two tracks can be evaluated using the dot product of their corresponding vectors. Each track is associated with a pid vector , $\in \mathbb{B}^{10^6}$ whose components take boolean values. These vectors are mostly sparse. Rather than storing all values, including zeros, only the playlist id where the track appears are kept.
For example:

`track_id:`	5ChkMS8OtdzJeqyybCc9R5
`Artist_name:`	Michael Jackson
`track_name:`	Billie Jean
`pid:`	$[5,47,65,89,89,331,342,342,374,384,\ldots]$

(4.1)

One can observe that the code accepts a vector containing duplicate components. In this example, the playlist 342 appears twice, indicating that the playlist 342 contains “Billie Jean” two times, these cases have already been discussed in 3.1. To remain within the vector space $\mathbb{B}^{10^6}$, only unique values must be kept. This is achieved by applying the set operator.

In the vector space $\mathbb{B}^{10^6}$, the most natural way to define addition is as follows:
Let a,b two vectors of $\mathbb{B}^{10^6}$ Then the addition is defined as

$$a + b = \{a_i \text{ or } b_i | i = 1, \ldots, 10^6\} \tag{4.2}$$

Suppose two list pid A, B such as in 4.1 and define

The operator set $is a projector into the space \mathrm{B}^{10^6}$. Then set($A$) is the list of unique items in the list A

len(A) as the number of items in A

$A \cap B$ is the list of unique common elements between A and B[1] .

The similarity between two tracks a_i and b_j, with respective pid list A_i and B_j is

$$\langle a_i, b_j \rangle = \frac{\text{len}(A_i \cap B_j)}{\sqrt{\text{len}(\text{set}(A_i)) \cdot \text{len}(\text{set}(B_j))}} \tag{4.3}$$

This definition ensures that for any track a_i b_j in the database $\langle a_i, a_i \rangle = 1$ (i), $\langle a_i, b_j \rangle \in [0,1]$ (ii) and that $\langle a_i, b_j \rangle = \langle b_j, a_i \rangle$ (iii)

Proof:

(i)

$$\begin{aligned}\langle a_i, a_i \rangle &= \frac{\text{len}(A_i \cap A_i)}{\sqrt{\text{len}(\text{set}(A_i)) \cdot \text{len}(\text{set}(A_i))}} \\ &= \frac{\text{len}(A_i \cap A_i)}{\sqrt{\text{len}(A_i \cap A_i) \cdot \text{len}(A_i \cap A_i)}} \\ &= 1\end{aligned}$$

(ii) $\text{len}(A_i) \geq 1 \ \forall i$

(iii) $\text{len}(A_i \cap B_j) = \text{len}(B_j \cap A_i)$

However, additional constraints must be imposed on the definition of the addition in order for the numerator to form an inner product. Indeed, the operator len is not linear.

Proof: Suppose the vector $[1,2] \in \mathbb{B}^{10^6}$

$$\text{len}(([1,2]+[1]) \cap [1,2]) \neq \text{len}([1,2] \cap [1,2]) + \text{len}([1] \cap [1,2])$$
$$2 \neq 3$$

which proves that the numerator of 4.3 is not an inner product, even if the interpretation is very similar[2]. To avoid this issue, one could use the vector space $\mathbb{N}^{10^6}$ rather than $\mathbb{B}^{10^6}$, but the option of boolean space have been preferred for memory reason.

[1] With that definition set(A) = $A \cap A$ but only if A is not in $\mathbb{B}^{10^6}$

[2] Using $\langle , \rangle$ is therefore a notational abuse, but it will continue to be used throughout this paper.

4.2 Application

The time required to compute the similarity product between two tracks is approximately 0.05 s. This is relatively slow and the approach to improve it has been discussed in Chapter 2.

4.2.1 Similarity Track Distribution Across the Database

To evaluate the similarity distribution, a Monte-Carlo–like simulation was chosen. A total of 45,300 tracks were randomly selected across the database. These tracks have an average len(pid) of 30, but a standard deviation of 350, indicating a large disparity between popular and unpopular tracks.
For each track, the similarity with 5 tracks randomly selected from the 45,300 tracks was evaluated. This resulted in more than 225,000 similarity values, almost exclusively concentrated near zero, with a mean on the order of 10^{-5}. This mean increases to 0.03 for the 10 most popular tracks within the 45,300 tracks.
It can be observed that the information contained in a single track is quite small, especially for unpopular tracks.

5 Q4: Playlist Similarity

In this section, a definition of the similarity score between playlists, ranging from 0 to 1, is introduced, along with a methodology for computing it using the dataset. Implementation aspects are discussed, and illustrative examples are provided.

The first approach directly follows the method described for track similarity in 4.1. A second approach, based on the vocabulary used, relies on Latent Dirichlet Allocation (LDA) and allows addressing the "playlist continuation" challenge for playlists composed only of a title or a description, and thus working even if no song are present in the playlist.

5.1 Problem Statement

Given a large collection of Spotify playlists, the goal is to define a similarity score between two playlists that is **numerical** and **bounded** between 0 and 1.

Formally, let $\mathscr{P} = \{P_1, \ldots, P_N\}$ denote the set of playlists. Our objective is to define a function

$$s : \mathscr{P} \times \mathscr{P} \to [0,1]$$

such that $s(P_i, P_j)$ is large when P_i and P_j are musically similar, and close to 0 when they are unrelated.

5.2 Method 1 & Method 2

5.2.1 First definition

Several approaches can be considered to measure the similarity between two playlists. The first approach estimates similarity by counting the number of common tracks divided by the square root of the product of the lengths of the two playlists.

Let $a = \{a_1, ..., a_{n_a}\}$, $b = \{b_1, ..., b_{n_b}\}$ two playlists where a_i, b_j are tracks, the sim_z is calculated such

$$\text{sim}_z(a,b) = \frac{\text{len}(\text{set}(a \cap b))}{\sqrt{\text{len}(a)\,\text{len}(b)}} \tag{5.1}$$

where

set(a) as the list of unique tracks in playlist a

len(a) as the number of items in playlist a

$a \cap b$ is the list of unique common elements between A and B[1].

This definition ensures that for any playlist a and b in the database $\text{sim}_z(a,a) = 1$ and $\text{sim}_z(a,b) \in [0,1]$

However, one might expect the result to be close to zero for a large portion of the dataset.[2].

5.2.2 Second definition

Having already defined the similarity product between two vectors in $\mathbb{B}^{10^6}$. One may consider mapping $\mathscr{P} \to \mathbb{B}^{10^6}$. The most natural way to do this is to sum all the tracks of the same playlist together.

Let $a = \{a_1, ..., a_{n_a}\}$ be a playlist, where a_i are tracks and A_i is the pid list [3] i-th track. Then, one defines the vector $A \in \mathbb{B}^{10^6}$ such that

$$A = \sum_{i=1}^{n_a} A_i \tag{5.2}$$

[1] With that definition set(a) = $a \cap a$

[2] One can calculate the expected sim_z in the case of playlists filled with random tracks. This would involve the combinatorial term $\binom{n_{tracks/playlist}}{n_{tracks}}$ which explains why the similarity would remain close to zero

[3] Recall that for example: $pid_{BillieJean}$: [5, 47, 65, 89, 89, 331, 342, 342, 374, 384, ...]

Then, the similarity between two playlists is defined in the same way as the similarity between two tracks.

$$\langle a, b \rangle = \frac{\text{len}(\text{set}(A \cap B))}{\sqrt{\text{len}(\text{set}(A))\,\text{len}(\text{set}(B))}} \tag{5.3}$$

Intuitively, if a playlist a contains 50 tracks and each track appears in multiple Spotify playlists, then the vector A associated with playlist a is the union of all playlists in which its tracks appear. This representation encodes a second–order signal: two playlists are considered similar if their tracks tend to appear in similar playlists across the entire database.

Moreover, to choose the most suitable track a_{n_a+1} to continue an existing playlist, one can look for the track a_{n_a+1} that maximizes the similarity product between the playlist a and all tracks not already included in the playlist

$$a_{n_a+1} = \sup_{\{k \mid a_k \notin \{a_1, a_2, .., a_{n_a}\}\}} \langle a, a_k \rangle \tag{5.4}$$

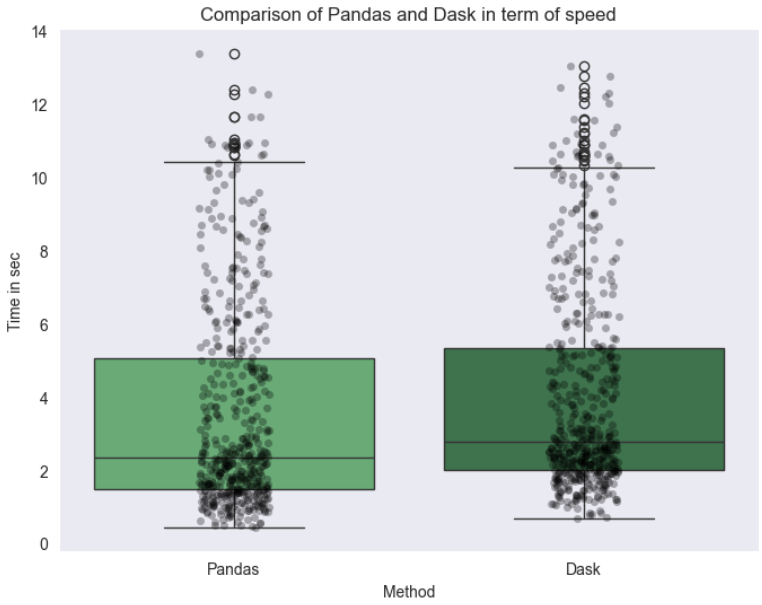
A random component may be added to avoid always suggesting the same tracks, but some limitations of this formula remain. For example, it tends to propose only popular tracks rather than new or less-known artists and presents some limitations in term of complexity.

5.2.3 Application

Speed comparison

The code allowed for two different approaches, which are explained in chapter 2. In Figure 5.1, a speed comparison between the two approaches is presented. The Pandas solution is generally faster, exhibiting a lower median execution time and a tighter distribution of processing times compared to Dask. This validates our architectural choice to use a custom indexing strategy with Pandas, which avoids the overhead and memory management issues encountered with Dask when performing random access on this specific dataset structure.

Figure 5.1: Speed comparison between method using Pandas and Dask



Similarity distribution

As the computational time required to evaluate the similarity between two playlists is relatively long, around 2–5 seconds, the formula was applied only to 1,000 pairs of playlists. The results, as expected, show that $\text{sim}_z(a,b) \leq \langle a,b \rangle$. The weight of zeros is much more significant for tracks than for playlists, as the information contained in the playlist vector is larger. The mean vector length is about 1,000, although the disparity remains large.

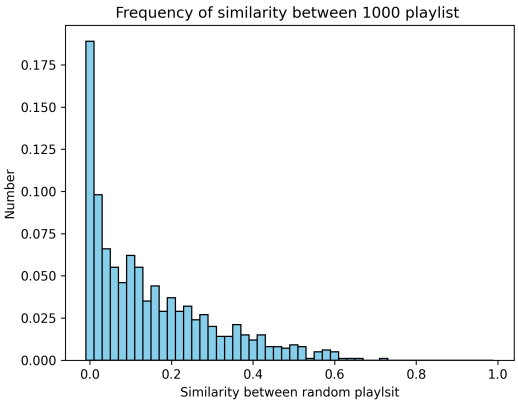


Figure 5.2: Similarity evaluated on 1,000 pairs playlist

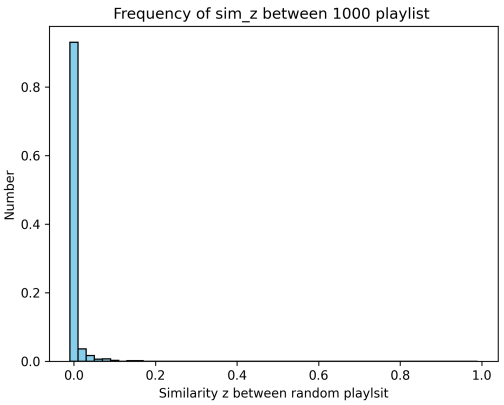


Figure 5.3: sim_z evaluated on 1,000 playlist

Other application

The formula also allows evaluating the similarity between artists. To do so, one must first build a playlist containing all the tracks of a given artist. This can be done for both artists under consider-

ation.

Then, using these two playlists, their similarity is evaluated, which provides a measure of similarity between the two artists. The code was not originally designed for this purpose, which is why performing such a calculation takes approximately 5–7 seconds.

For example, let us consider approximately eight major artists for each decade since 1980.

1980	**1990**	**2000**	**2010**	**2020**
Michael Jackson	Mariah Carey	Beyoncé	Adele	Billie Eilish
Madonna	Britney Spears	Eminem	Bruno Mars	Dua Lipa
Prince	Whitney Houston	Rihanna	Drake	The Weeknd
Pat Benatar	Nirvana	Usher	Ed Sheeran	BTS
George Michael	Madonna	Justin Timberlake	Katy Perry	Harry Styles
Bon Jovi	Dr. Dre	Linkin Park	The Weeknd	Bad Bunny
David Bowie	Celine Dion	Kanye West	Billie Eilish	Taylor Swift
Phil Collins		Lady Gaga	Lady Gaga	Lizzo

Table 5.1: Couple majors artists by decade

The similarity in each decade is then perform on all the possible pairs and result are plot on the figure 5.4. There is less point for the 1990s because one of these artist have not been found in the database, probably because a typographical mistake.

In any case it shows better result for the decade 2000s and 2010s. The low result for 2020s is probably due to the fact that the database comes from 2020 and that some of those artist were not famous before the recent years.

Many additional analyses could be performed based on the similarity. For example, investigating whether there are similarities across different decades, languages, or genres of music such as rock, R&B, rap, etc. But this will suffice for now.

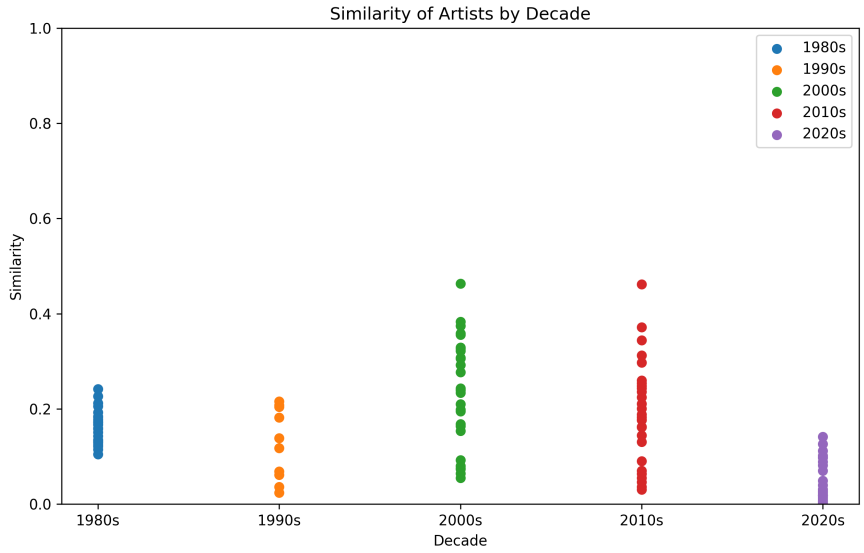


Figure 5.4: Similarity between major artists

5.3 Method 3: topics based using Latent Dirichlet Allocation

Here, each playlist is represented by a *latent topic distribution* obtained from a Latent Dirichlet Allocation (LDA) model trained on the textual information derived from the playlists [Gee25],[Res23]. The similarity score is then defined as the cosine similarity between the corresponding topic distributions [Wik25b].

5.3.1 Methodology

Step 0: Textual description of a playlist

The Million Playlist Dataset provides, for each playlist, structured information such as a unique identifier (`pid`), a title (e.g. “Chill Acoustic Evening”), an optional free-text description and a list of tracks, each with: a track name, an artist name and an album name. From these fields, one can constructs a textual document for each playlist by concatenating several pieces of information. In practice, the following text fields are combined:

- the playlist title;
- the playlist description (when present);
- the track titles;
- the artist names.

This yields, for each playlist P_i, a raw text string T_i that summarises its content:

$$T_i = \text{concat}(\text{title}, \text{description}, \text{track names}, \text{artist names}).$$

The resulting dataset is stored in a DataFrame `df_playlists` with at least the following columns:

`pid`	`name`	`text`
Playlist identifier	Playlist title	Concatenated textual representation T_i

Table 5.2: Column of the stored DataFrame

Step 1: Pre-processing and vectorisation

The raw text T_i is then pre-processed before being fed into a machine learning model. The typical pre-processing pipeline includes:

1. lowercasing all characters;
2. removing punctuation and non-alphabetic characters;
3. tokenising the text into words;

Then the collection of documents $(T_1, \ldots, T_N)$ is transformed into a sparse bag-of-words matrix[4] using a `CountVectorizer`.

Let V denote the *vocabulary*, i.e. the complete set of unique words extracted from all playlists after preprocessing. Its size is $|V| = d$, meaning that each playlist will be represented by a vector of dimension d.

The `CountVectorizer` converts each playlist text T_i into a numerical vector $\mathbf{x}_i \in \mathbb{N}^d$. Each vector has d values, one for each word in the vocabulary. The value $x_{i,j}$ tells how many times word j appears in playlist i.

By placing all vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ on top of each other, on can obtain the bag-of-words matrix:

$$X_{\text{bow}} \in \mathbb{R}^{N \times d}.$$

Here, N is the number of playlists and d is the number of words in the vocabulary. A exemple of this is given in tab5.3.

[4] A *bag-of-words* representation is a classical text-processing technique in which each document is converted into a numerical vector that counts how many times each word appears, while ignoring grammar, word order, and sentence structure [Wik25a]. The term *sparse* indicates that most entries in this matrix are zero, since each playlist only contains a very small subset of all possible vocabulary terms.

Playlist	**Text** T_i	`chill`	`lofi`	`beats`	`study`	`electro`
T_1	"chill lofi beats"	1	1	1	0	0
T_2	"lofi study"	0	1	0	1	0
T_3	"chill electro"	1	0	0	0	1

Table 5.3: Example of the Bag-of-Words representation (X_{bow}matrix): playlists T_i, vocabulary terms, and corresponding frequencies $x_{i,j}$. Where the vocabulary extracted from the three playlists is: V = ["chill","lofi","beats","study","electro"], $|V| = d = 5$.

Step 2: How to build the topics

LDA does not start with predefined themes. The topics are learned automatically from the data. Here, LDA was trained on a set of 50,000 playlists. To build these topics, the model looks at all playlists and all words they contain. It then searches for groups of words that tend to appear together in many playlists. Each such group becomes a latent topic.

Table 5.4: Example of top words associated with each latent topic.

Topic	**Top words (highest probabilities)**
Topic 1	`chill`, `lofi`, `relax`, `sleep`
Topic 2	`rock`, `guitar`, `band`, `live`
Topic 3	`edm`, `techno`, `dance`, `club`

Step 3: How to embed [5] the playlists

In practice one fixes:

$$K = \texttt{N\_TOPICS} = 30,$$

which offers a compromise between expressiveness and computational cost. The model is trained with the main hyperparameters given in Annex7.1.1.

Calling `lda.fit_transform(X_bow)` produces a matrix called `playlist_topic_matrix`. Its shape is $N \times K$, meaning one row per playlist and one column per topic:

$$\Theta = \texttt{playlist\_topic\_matrix} \in \mathbb{R}^{N \times K}.$$

Each row $\boldsymbol{\theta}_i$ represents the topic distribution of playlist P_i. All values are non-negative, and they sum to 1:

$$\theta_{i,k} \geq 0, \qquad \sum_{k=1}^{K} \theta_{i,k} = 1.$$

This vector places each playlist in a K-dimensional space (i.e., 30 dimensions). The value $\theta_{i,k}$ indicates how strongly playlist P_i is associated with topic k (cf.tab5.5). In other words, LDA checks

[5] Turn them into vectors

which words of a playlist match the words that characterise each topic, and assigns a degree of membership accordingly. The resulting topic vector reflects how much each hidden theme contributes to the playlist.

Table 5.5: Example of topic distributions $\boldsymbol{\theta}_i$ for three playlists ($K = 3$ topics).

Playlist	Topic 1	Topic 2	Topic 3
P_1	0.70	0.20	0.10
P_2	0.10	0.80	0.10
P_3	0.30	0.10	0.60

Step 4: Similarity measurement with Cosine similarity on topic distributions

Let P_i and P_j be two playlists with corresponding topic distributions $\boldsymbol{\theta}_i$ and $\boldsymbol{\theta}_j$, extracted from `playlist_topic_matrix`. One can define their similarity score as the cosine similarity between these two vectors:

$$s(P_i, P_j) = \text{cosine}(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \frac{\boldsymbol{\theta}_i \cdot \boldsymbol{\theta}_j}{\|\boldsymbol{\theta}_i\|_2 \, \|\boldsymbol{\theta}_j\|_2}. \tag{5.5}$$

Because each $\boldsymbol{\theta}_i$ is a probability distribution (i.e. a non-negative vector whose entries sum to 1), we have:

$$0 \leq s(P_i, P_j) \leq 1 \quad \text{for all } i, j,$$

which satisfies the requirement that the similarity score lies in $[0, 1]$.

Intuitively, $s(P_i, P_j)$ is large when the two playlists give high probability mass to the same topics and small when they concentrate on different topics. For example, two “chill lofi” playlists will have very similar topic distributions and hence a cosine similarity close to 1, whereas a “classical piano” playlist and a “hard techno” playlist should be almost orthogonal in topic space, with a similarity close to 0. Several advantages of this method, justifiying its usage, are presented in the annex 7.1.3.

5.3.2 Example Results

This section shows illustrative examples of the behaviour of the similarity function. Note that playlist identifiers and titles are for illustration only, the actual output of the function in the notebook has the same structure.

Consider a query playlist P_q that is dominated by a topic related to acoustic and lofi music. Its topic distribution could look like:

$$\boldsymbol{\theta}_q = [0.02, 0.60, 0.30, 0.08, 0, \ldots, 0]$$

where topic 2 corresponds to “acoustic / lofi chill” and topic 3 to “soft pop”.

Applying `get_similar_playlists` on this playlist might return a table of the form of tab 5.6:

Rank	pid	Name	Similarity
1	105732	Chill Acoustic Evening	0.96
2	82311	Lofi Study Session	0.93
3	217890	Acoustic Morning Coffee	0.91
4	90213	Soft Pop & Lofi Mix	0.89
5	130004	Relaxing Indie Acoustic	0.87

Table 5.6: Top–5 most similar playlists returned by the similarity function.

All returned playlists have similarity scores ≥ 0.87, indicating that their topic distributions are strongly aligned with that of the query playlist. Qualitatively, their titles confirm that they share a similar theme (chill, acoustic, lofi).

5.3.3 Complexity considerations

Let N be the number of playlists and K the number of topics. Once the topic matrix Θ is computed, the cost of one similarity query is dominated by the cosine similarity computation, which is $O(NK)$. For moderate K (here $K = 30$) this is very fast.

On the other hand, training the LDA model is naturally more expensive, but it is a one-time cost.

5.3.4 Discussion and Limitations

The proposed similarity function is global and topic-based, which makes it robust to small differences in track lists. However, it also has limitations.

Firstly, it depends on the quality of the topic model. If LDA fails to capture meaningful musical themes, the similarity scores may be less interpretable.

Secondly, it ignores non-textual information such as play counts, skip behaviour, or audio features (tempo, energy, etc.) which could be integrated via Fourier transform methods or similar technics.

6 Q5: Resolution of the "playlist continuation" challenge

6.1 Method 1: Topics Based Recommendation Method

The recommendation strategy relies on two main steps.

Step 1: Nearest Playlists. For a given query playlist, the cosine similarity with all playlists in the dataset is computed. The 500 most similar playlists are then selected, using the similarity scores $s_{LDA}(p)$. These playlists define the neighbourhood from which candidate tracks are derived.

Step 2: Tracks Scoring and Ranking. All tracks appearing in these 500 neighbour playlists are collected, excluding those already present in the query playlist. For each candidate track t, a score is assigned according to:

$$\text{score}(t) = \sum_{p \ni t} s_{LDA}(p),$$

which corresponds to a weighted sum of the similarity scores of all playlists that contain t. Tracks occurring in playlists with higher similarity receive greater scores. Candidate tracks are ranked in decreasing order of their score, and the top 500 tracks are selected as the continuation of the playlist.

Discussion. This method avoids computing explicit track–to–track similarities and instead exploits playlist similarity to propagate relevance. It can be viewed as an item-based collaborative filtering approach operating in the latent topic space produced by the LDA embeddings.

6.1.1 Example of Playlist Continuation

To illustrate the behaviour of the proposed method, playlist `pid = 29`, named *groovy* was selected from the dataset (of 5GB, not the complete one). Using the retrieval method of section6.1, the

highest-scoring tracks selected for continuation of the playlist are found.

Table 6.1 reports the top-10 recommended tracks returned by the model. The suggestions include popular pop and electronic titles which correspond well to the musical profile of the original playlist.

Table 6.1: Top–10 recommended tracks for playlist `pid = 29`.

Track Name	**Artist**	**Score**
It Ain't Me	Kygo	133.30
Stay (feat. Alessia Cara)	Zedd	123.65
Despacito (Remix)	Luis Fonsi	121.71
Issues	Julia Michaels	119.76
Shape of You	Ed Sheeran	109.86

For comparison, Table 6.2 shows a subset of the tracks that actually belong to playlist `pid = 29`. The recommended items exhibit strong stylistic alignment with these ground-truth tracks, indicating that the model effectively captures shared musical themes through its neighbour-based scoring mechanism.

Table 6.2: Excerpt of ground-truth tracks from playlist `pid = 29`.

Track Name	**Artist**
I Like Me Better	Lauv
Feels	Calvin Harris
Two High	Moon Taxi
Mama	Jonas Blue
Slow Hands	Niall Horan

This example shows that the system produces coherent playlist extensions by leveraging similarity between playlists and aggregating neighbour votes to rank candidate tracks.

6.1.2 Evaluation of the Recommendation Method

This section describes how the recommendation procedure is evaluated on the *Million Playlist Dataset*. The goal is to verify whether the model is able to recover the missing tracks of a playlist when only a small subset of its tracks is provided as input.

Evaluation Protocol. Instead of evaluating a single playlist, the procedure is repeated on a large set of playlists. For each playlist with at least $K+1$ tracks, K tracks are sampled as seed tracks, while the remaining tracks constitute the ground truth. The model receives only the seed tracks and must infer the missing ones.

To assess performance, the Recall@500 metric is used:

$$\text{Recall@500} = \frac{|\text{Recommended}_{500} \cap \text{GroundTruth}|}{|\text{GroundTruth}|}.$$

A higher Recall@500 indicates that a larger proportion of the hidden tracks is successfully recovered among the 500 recommendations.

Experimental Setup. Using $K = 25$, all playlists containing at least 26 tracks were selected as eligible evaluation candidates (733). For each playlist, seed and ground-truth splits were generated, recommendations were computed, and Recall@500 was measured.

Results. Before running any inference, it should be noted that one "slice" of data has been put aside to run this final test. Thus, this "slice" of data was not used during the training phase of LDA, avoiding any type of overfit. A total of 733 playlists were evaluated (i.e. the recommendation task was tested 733 times). The method achieved an average Recall@500 of 0.12, with a median of 0.09. This indicates that, on average, the system successfully retrieves approximately 12% of the tracks (on average) hidden from each playlist, demonstrating that the LDA based recommendation method provides a strong signal for playlist continuation.

6.2 Other considerations and potential improvements

6.2.1 Increasing the LDA Training Set Size

The original Kaggle dataset contains approximately **35 GB** of playlist–track data, far larger than the subset used in our experiments (about 50k playlists).

A larger training set would lead to significantly **higher-quality and more stable topics**. In practical terms, this yields more informative playlist embeddings and improves downstream recommendation quality, especially for rare or diverse playlists.

The primary drawback of doing such an increase is the **computational cost**. Standard LDA implementations such as `scikit-learn` cannot scale to tens of gigabytes of data: both memory usage and training time grow prohibitively with the number of documents. Processing the full 35 GB dataset would therefore require a distributed solution (e.g., Spark MLlib LDA), as suggested by spark [Apa25].

6.2.2 Mitigating Popularity Bias in Playlist-Based Similarity Measures

As the original track scoring function (i.e. base on s_{LDA} only) increases directly with the number of playlists in which a track appears, a natural extension of the proposed approach is to introduce an explicit **penalty for track popularity**.

Indeed, tracks that occur in a large number of the top-500 selected playlists tend to receive disproportionately high scores, even when their contributions are weighted by playlist–query similarity. As a result, **highly popular tracks may be favoured over less frequent but stylistically relevant ones.**

To mitigate this effect, the accumulated similarity score can be normalised by a popularity-dependent factor, thereby reducing the dominance of ubiquitous tracks while preserving the influence of playlist similarity.

Let n_t denote the number of playlists in which track t appears. The popularity-aware score is defined as:

$$\text{score}_{\text{pop}}(t) = \frac{\sum_{p \ni t} s_{pl}(p)}{\log(1 + n_t)}.$$

The addition of 1 inside the logarithm ensures numerical stability and prevents division by zero, under the assumption that each track appears in at least one playlist in the dataset.

However, the notion of popularity bias itself can be questioned. From a practical perspective, popularity may also be interpreted as a positive signal: a popular track has already appealed to a large number of users, which may legitimately increase its likelihood of being recommended, independently of its presence in highly similar playlists. Beyond methodological considerations, popularity may therefore contribute positively to the perceived relevance of recommendations. This raises the question of whether popularity bias should be regarded as a true limitation from a business perspective for a platform such as Spotify, or rather as **an intentional trade-off between personalisation and global appeal.**

6.2.3 Method 2: Combining Topic-Based Recommendation with Dot-Product Scoring

The recommendation procedure could be enhanced by combining two complementary similarity signals: (i) the topic-based similarity obtained from the LDA model used until now (cf.section6.1), and (ii) the playlist similarity defined via the dot-product–based measure introduced in section 5.2.2. The intuition is that the LDA similarity captures high-level semantic structure, while the dot-product similarity captures co-occurrence patterns derived directly from user behaviour.

Let $s_{\text{LDA}}(p)$ denote the LDA cosine similarity between the query playlist and a neighbour playlist p, and let $s_{\text{DP}}(p)$ denote the "dot-product–based similarity" score computed as in Section 5.2.2. We introduce two weighting coefficients, α and β, which control the respective importance of the two similarity components. They are to be found empirically [1].

The combined playlist similarity is thus defined as:

$$s_{\text{comb}}(p) = \alpha\, s_{\text{LDA}}(p) + \beta\, s_{\text{DP}}(p).$$

Process

Step 1: Once the top 500 most similar playlists are selected according to s_{LDA}, candidate tracks are collected, as in the original LDA-based method.

Step 2: For each playlist selected, the similarity score of section 5.2.2 is computed. Here, note that the pid list is already built.

Step 3: Each playlist receive its similarity score using $s_{\text{comb}}(p)$

Step 4: For each candidate track t, a recommendation score is assigned by summing the $s_{\text{comb}}(p)$ of all the playlist in its pid list of t:

$$\text{score}(t) = \sum_{p \ni t} s_{\text{comb}}(p).$$

Step 5: Tracks are then ranked according to score(t), and the top 500 items are returned as the playlist continuation.

[1] One could think of adapting the value of those coefficients depending the current number of tracks in the input playlist, giving bigger α when fewer tracks (i.e., more weight toward description/words because not enough tracks to use $s_{\text{DP}}(p)$) and vice versa.

7 Appendices

7.1 Appendix LDA

7.1.1 Hyperparameters

- `n_components = 30` (number of topics K);
- `learning_method = "online"` (suitable for large data);
- `max_iter = 10`;
- `random_state = 42` (for reproducibility);
- `n_jobs = 1` (to limit disk usage in the Colab environment).

7.1.2 Implementation: `get_similar_playlists`

The computation is encapsulated in a helper function (conceptually similar to the following):

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def get_similar_playlists(query_idx, top_k=10):
    # 1. Extract the topic vector of the query playlist
    query_vec = playlist_topic_matrix[query_idx].reshape(1, -1)

    # 2. Compute cosine similarity with all playlists
    sims = cosine_similarity(query_vec, playlist_topic_matrix)[0]
```

```
    # 3. Sort by decreasing similarity
    order = np.argsort(-sims)

    # 4. Remove the trivial self-match at index query_idx
    order = order[order != query_idx]

    # 5. Keep the top-k most similar playlists
    top_indices = order[:top_k]

    # 6. Build a small DataFrame with playlist info and scores
    result = df_playlists.loc[top_indices, ["pid", "name"]].copy()
    result["similarity"] = sims[top_indices]

    return result.reset_index(drop=True)
```

7.1.3 Advantages of this definition

Using cosine similarity on LDA topic distributions brings several benefits:

- **Semantic similarity:** Two playlists can be similar even if they do not share any exact tracks, as long as they explore similar musical topics.
- **Dimensionality reduction:** Instead of comparing sparse vectors in a very high-dimensional vocabulary space, we work in a compact K-dimensional topic space.
- **Bounded score:** The cosine similarity between non-negative vectors lies naturally in $[0,1]$, so no additional normalisation is needed.
- **Efficiency:** Cosine similarity can be computed very efficiently using vectorised operations (e.g. via `sklearn.metrics.pairwise.cosine_similarity`).

Articles

[Apa25] Apache Spark Contributors (2025). “Latent Dirichlet Allocation (LDA) Spark MLlib Documentation”. In: *https://spark.apache.org/docs/latest/ml-clustering.html#latent-dirichlet-allocation-lda*. Consulté le 13 décembre 2025.

[Gee25] GeeksforGeeks (2025). “Topic Modeling Using Latent Dirichlet Allocation (LDA)”. In: *https://www.geeksforgeeks.org/nlp/topic-modeling-using-latent-dirichlet-allocation-lda/*. Consulté le 2 décembre 2025.

[Res23] Research, IBM (2023). “What is Latent Dirichlet Allocation (LDA)?” In: *https://www.ibm.com/think/topics/latent-dirichlet-allocation*. Consulté le 2 décembre 2025.

[Wik25a] Wikipedia contributors (2025a). “Bag-of-words model”. In: *https://en.wikipedia.org/wiki/Bag-of-words_model*. Consulté le 2 décembre 2025.

[Wik25b] — (2025b). “Cosine similarity”. In: *https://en.wikipedia.org/wiki/Cosine_similarity*. Consulté le 2 décembre 2025.