Spotify Frequent Itemset Mining & Recommendation Engine

Aditya Gupta
Mehta Family School of
Data Science & Artificial Intelligence
IIT Guwahati
aditya.dsai21@iitg.ac.in

Anuj Gupta
Department of Physics
IIT Guwahati
anuj.g@iitg.ac.in

***Abstract*—This project focuses on creating a music recommendation system using the Spotify Million Playlist Dataset. We will preprocess and analyze a subset of the dataset along with audio features of tracks to generate song recommendations for a user input. Leveraging big data frameworks like MapReduce and Vector Databases, we aim to provide song recommendations based on playlist relations of tracks and track audio features. We aim to create a interface that allows users to enter the current tracks they are listening to and provide them recommendations along with insights on the basis on which the recommendations were made. We also use frequent itemset mining as a mode of recommendation, which leverages playlist data to gain insights into which tracks users often listen to in combination.**

***Index Terms*—spotify, music, recommendation, frequent itemset mining, vector similarity search, approximate nearest neighbours, pyspark**

I. Motivation

The motivation behind this project stems from the ever-increasing volume of music data available in platforms like Spotify and the growing complexity of users' music preferences. The data set has millions of playlists containing hundreds of tracks, so analyzing this vast dataset can unlock valuable insights into user preferences, music trends, and hidden connections between songs. The use of big data frameworks is motivated by the need to handle large-scale data efficiently. These technologies enable the processing of massive music datasets in real-time, ensuring that the system remains responsive even as the data continues to grow.

The availability of playlist data and not just tracks allows for improving on purely content based recommendation methods. The information of frequently occurring combinations of songs that users listen to can be used to improve on content based recommendation. This idea of frequent itemset mining can also provide recommendations by generating playlists based on frequent co-occurrence of tracks that serve as playlists based on popularity.

II. Introduction

The general outline of this project report is as follows, we first introduce the dataset, the Spotify Million Playlist Dataset that is going to be used in this project. Then we discuss a content based recommendation method, which uses **vector similarity search** on embeddings created using track features and audio features. We go over implementation details of this method and compare the use of vector databases as compared to traditional NoSQL databases.

Next, we introduce recommendation using **frequent itemset mining** and in particular discuss our look implemention of the **A-priori** algorithm in the PySpark framework, analysing how this algorithm can be parallelized over PySpark RDDs. For both methods, we go over their theoretical and implementation details. Following this we discuss methods of combining the 2 approaches and also go over some fine details of the implementation. We conclude by looking at some specifics of the front-end along with some results and a conclusion.

III. Dataset

A. Dataset description

We are using the Spotify Million Playlist Dataset, which consists of one million playlists. Each playlist in the MPD contains a playlist title, the track list (including track IDs and metadata), other metadata fields (last edit time, number of playlist edits, and more).

The dataset includes public playlists created by US Spotify users between January 2010 and November 2017. Playlists that meet the following criteria were selected at random:

- Contains at least 5 tracks
- Contains no more than 250 tracks
- Contains at least 3 unique artists
- Contains at least 2 unique albums
- Has at least one follower (not including the creator)

B. Dataset Format

The dataset is provided in a form of 1000 slices containing 1000 playlists each. The dataset is given in JSON format, each playlist has some metadata and a JSON array of tracks. The total size of the dataset is approximately 33GB containing roughly 66 million tracks of which 2.2 million are unique tracks. Below is a snippet of a JSON of the dataset, showing all the attributes of the playlists and tracks provided.

```
{
    "name": "musical",
    "collaborative": "false",
    "pid": 5,
    "modified_at": 1493424000,
    "num_albums": 7,
    "num_tracks": 12,
    "num_followers": 1,
    "num_edits": 2,
    "duration_ms": 2657366,
    "num_artists": 6,
    "tracks": [
        {
            "pos": 0,
            "artist_name": "Degiheugi",
            "track_uri": "spotify:track:7
               vqa3sDmtEaVJ2gcvxtRID",
            "artist_uri": "spotify:artist
               :3V2paBXEoZIAhfZRJmo2jL",
            "track_name": "Finalement",
            "album_uri": "spotify:album:2
               KrRMJ9z7Xjoz1Az4O6UML",
            "duration_ms": 166264,
            "album_name": "Dancing Chords
                and Fireflies"
        },
        {
            "pos": 1,
            "artist_name": "Degiheugi",
            "track_uri": "spotify:track:2
               3EOmJivOZ88WJPUbIPjh6",
            .
            .
            .
        },
        .
        .
        .
    ],

}
```

For the purpose of this project, we did not use the entire 1 million playlists of the dataset due to computational constraints. We used a subset containing 100k playlists.

IV. Content Based Recommendation

A. Theoretical Framework

In the realm of recommendation systems, for this project we specifically want to implement content based recommendation that uses item profiles, i.e., for a user input item we recommend items that have similar item profiles to that of the input item. The general idea is to create a database of item to item profile mappings. When a user queries for recommendations following an input item, we create the item profile of the input item and then use a search algorithm to find similar items in the database based on item profiles.

For the purpose of this project, we are are not building user profiles and do not store personal preference of users, we do not aim to do recommendation based on what a user's preferences are. The aim of this project is to provide a general purpose recommendation system that generalises to all users. Using user profiles with the approaches discussed in this project would definitely improve the quality of personalised recommendation, however acquiring a dataset of user information and their preferred music domains is not feasible as most music platforms do not make their users' data publically available.

Since we intend to recommend without building user profiles, content based recommendation is a very well suited method as it doesn't require any user data to begin with. In fact we dont need playlist data either, only track information and features are sufficient. An added benefit is that content based recommendation is able to recommend newly added tracks and gives equal attention to popular and less popular tracks, allowing newly added tracks to also be discovered by users. The major challenge with this method is building high quality features to make item profiles.

For this approach we have two major tasks at hand,

- Creating a database of item profiles
- Using an efficient search approach to retrieve similar items

B. Creating track embeddings

For the task of music recommendation, item profiles represent a comprehensive set of characteristics that define each individual track, encapsulating details such as genre, tempo, and acoustic features. These profiles act as a multidimensional representation of the musical content. This set of characteristics or features can be represented in a condensed vector space. Representing item profiles as dense vectors allows us to define formal ways of computing similarity of items and also allows for understanding relations between tracks. For content based recommendation of music tracks, we define a item profiles as a high dimensional vector representing various audio and track features. This vector is referred to as *track embedding* in this project report.

To create track embeddings we used various audio and track features. Audio features used include:

- *Energy*: A perceptual measure of intensity and activity.
- *Key*: The key(pitch) the track is in. Integers map to pitches using standard Pitch Class notation.
- *Loudness*: The overall loudness of a track in decibels (dB).
- *Danceability*: Describes how suitable a track is for dancing based on a combination of musical elements including

tempo, rhythm stability, beat strength, and overall regularity.

- *Mode*: Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived.
- *Speechiness*: Speechiness detects the presence of spoken words in a track.
- *Acousticness*: A confidence measure of whether the track is acoustic.
- *Instrumentalness*: Predicts whether a track contains no vocals.
- *Liveness*: Detects the presence of an audience in the recording.
- *Valence*: A measure describing the musical positiveness conveyed by a track. A sentiment analysis based on audio components.
- *Tempo*: The overall estimated tempo of a track in beats per minute (BPM).Tempo is the speed or pace of a given piece and derives directly from the average beat duration.
- *Time signature*: The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).

Track features used are:

- *Popularity*: A measurement of popularity ranging from 0 to 100 based on number of plays.
- *Release year*: Release year of the track's album. A user listening to old songs would prefer recommendations of old tracks and a user listening to old songs would not want to listen to old tracks.
- *Explicit*: Whether or not the track has explicit lyrics. A user input having explicit lyrics should be recommended some songs having explicit lyrics, however an input with no explicit lyrics should not be recommended any tracks with explicit lyrics. This is ensured by using this feature as a Boolean rather than a continuous value.

Preprocessing of track embeddings includes normalising all individual features in a 0 to 1 scale before creating track embeddings. After creating embeddings for all tracks embeddings are centered to zero mean by subtracting the mean of all tracks. The audio and track features used for creating track embeddings are collected using the open source Spotify Web API. The preprocessing was handled in python. The general flow of how we will use these track embeddings is described in below flowchart.

C. *Vector Similarity Search*

Once we have created embedding vectors for all tracks, we need to recommend tracks similar to the user input track. For this we need to first define what we mean similarity. We can say 2 tracks are similar if they have similar features, further 2 tracks are similar if the vectors representing them are similar. So we need a similarity measure of vectors. For the size of dataset we are using, current practices suggest the use of cosine similarity, and we used the same for this task. For a user input track, we recommend tracks that have the top 20 highest values of cosine similarity with this input track. To implement this we need a method to efficiently search the entire database of track embedding vectors and find the most similar tracks by cosine similarity.

1) Approach 1 - Using MongoDB: Our first approach was to used MongoDB to store all our vectors. To recommend tracks similar to an input vector, the entire database would be iterated over and cosine similarity would be computed between every vector in the database and the input vector. The vectors resulting in the top 20 highest cosine similarities are recommended to the user. This method accurately determines the most similar tracks, however this method is extremely slow. An initial test setup that used a MongoDB database containing 35K vectors took roughly 6 seconds to compute recommendations for a single user query. This means that delivering recommendations to a user while utilising one-tenth of the original dataset(660k tracks) would take over 2 minutes per query which is impractical.

2) Approach 2 - Using Pinecone Vector Database: To improve our search speed, we decided to use a vector database. The current SOTA method for retrieval of high dimensional vectors is to use Approximate Nearest Neighbour search methods. Vector databases like Pinecone are based on Approximate Nearest Neighbor methods like FAISS (Facebook AI's similarity search algorithm) which create vector indexes to efficiently and quickly find similar vectors by cosine similarity.

The performance improvement achieved by this method is what makes content based recommendation practical. Using pinecone, a database containing 100k vectors can return top 20 similar vectors to an input vector in under 1 second. The pinecone database used is hosted on the cloud and accessible using pinecone API which is open source.

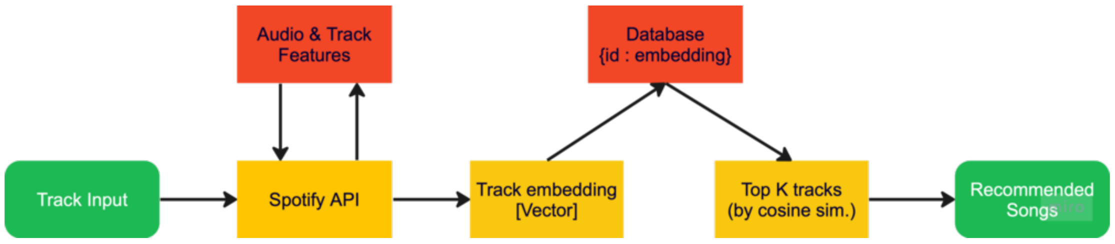

Fig. 1. Flowchart describing recommendation done using track embeddings

V. Frequent Itemset Mining based Recommendation

A. Introduction to Frequent Itemset Mining

Frequent itemset mining is a crucial technique in data mining that focuses on uncovering patterns of items that occur together frequently within a dataset. The process begins by examining a large number of data transactions, where each transaction is comprised of a set of individual items. By analyzing these transactions, the aim is to identify combinations of items that appear more frequently than a predetermined threshold.

The significance of an itemset's frequency is determined by a parameter known as the support threshold. The support of an itemset is calculated as the percentage of all transactions in the dataset that include this particular combination of items. Frequent itemset mining is tasked with the discovery of all the itemsets that meet or exceed this minimum support threshold, thereby revealing the most common item associations within the dataset.
To summarise this method, frequent itemset mining discovers all combinations of items that frequently co-occur in the same collection over a database containing a large number of collections of items. A set of items (itemset) is said to frequently co-occur if this combination of items is found in more than a threshold fraction of collections of the database.

B. Application in Track recommendation

To extend frequent itemset mining to track recommendation, we need to define what the items and transactions/collections are in this application. The spotify dataset contains 1 million playlists, and each playlist is a user created customised collection of tracks that the user wants to listen to in one session. Here, individual tracks are the items and playlists are the collections/transactions of items that users create. The goal of frequent itemset mining here is to find what set of tracks frequently co-occur in atleast a threshold number of playlists in the dataset.

Intuitively, this can be used as a recommendation method because when a user creates a playlist, they are choosing tracks that are related in a way that the user would like to listen to a track of the same playlist after every track in that playlist. The task of recommendation is related to this, here the goal is to recommend a user a track they would like to listen to based on their previous track. If track A and track B co-occur in a large number of playlists, that indicates that users find the tracks to be related such that they would like to listen to track A and track B in the same session and hence track B can be recommended after A and vice versa.

Frequent itemset mining can provide a popularity based recommendation of sorts. If a large itemset occurs in a many playlists, that means many users like to listen to these tracks together. These large itemsets can act as trending playlists of the time as they represent a commonly chosen collection of songs by people. For this reason, we have implemented a trending today section in the frontend of the project which is the largest frequent itemset of the database. This is the largest set of tracks that co-occur in 0.5% of the 100k playlists of the database used.

VI. Implementation of Frequent Itemset Mining

This section goes over experimentation in implementation methods of frequent itemset mining used in this project. Frequent itemset mining cannot be brute-forced as there exists 2^N possible itemsets, where N is the number of unique tracks. This computation is unfeasible.

For many frequent itemset algorithms, main-memory is the critical resource. As we read baskets, we need to count something, e.g., occurrences of pairs of items, and the number of different things we can count is limited by main memory. If we have to load a page from disk every time to update a count the amount of I/O operations needed are extremely large. In this project, we implemented the A-priori algorithm for frequent itemset mining.

A. The A-priori Algorithm

The A-priori algorithm employs an iterative approach, where it first finds all single-item frequent itemsets (items that meet a minimum support threshold), then pairs of items that appear frequently together, and so on, increasing the size of the itemsets with each iteration.

The core principle behind the A-priori algorithm is the Apriori property, which states that all subsets of a frequent itemset must also be frequent. This property is used to reduce the search space significantly. In each iteration, the algorithm generates candidate itemsets of a particular size from the itemsets found to be frequent in the previous iteration. It then scans the database to count the occurrences of these candidate itemsets. If a candidate itemset does not meet the minimum support threshold, it is discarded, as are its supersets, due to the Apriori property.
This pruning step is crucial as it improves the efficiency of the algorithm by avoiding the generation of itemsets that are unlikely to be frequent. After the process concludes, the algorithm outputs all the itemsets that meet the specified support threshold.

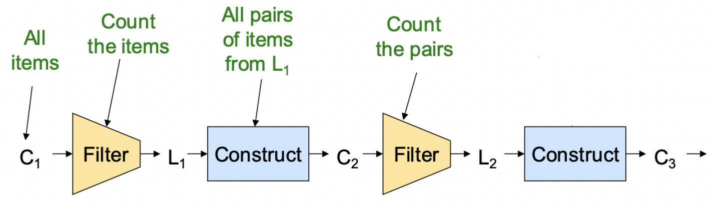

Fig. 2. Two pass process of A-priori algorithm

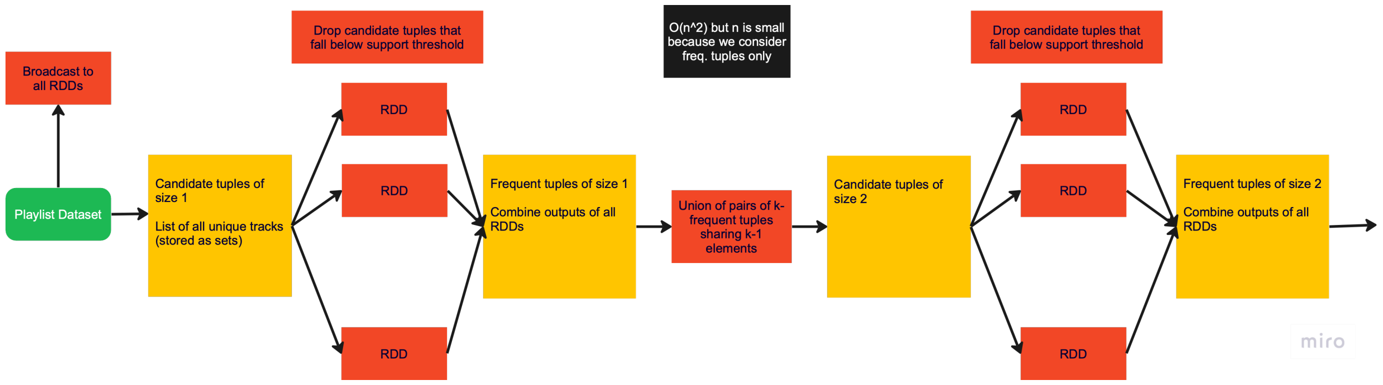
Broadcast to all RDDs
Playlist Dataset
Candidate tuples of size 1
List of all unique tracks (stored as sets)
Drop candidate tuples that fall below support threshold
RDD
RDD
RDD
Frequent tuples of size 1
Combine outputs of all RDDs
O(n^2) but n is small because we consider freq. tuples only
Union of pairs of k-frequent tuples sharing k-1 elements
Candidate tuples of size 2
Drop candidate tuples that fall below support threshold
RDD
RDD
RDD
Frequent tuples of size 2
Combine outputs of all RDDs
miro

Fig. 3. Implementation of A-priori Algorithm in PySpark framework. RDDs refer to Resilient Distributed Datasets of PySpark over which the computation is parallelised.

As described in Fig.2, A-priori algorithm does two passes for computing frequent itemsets of each size. For getting frequent itemsets of size k, in first pass over the data we create candidate itemsets C_{k} from frequent itemsets $L_{\text{k-1}}$ of size k-1. The exact steps of this are discussed in the next subsection. Following this, in the second pass of the data, we filter all candidate itemsets C_{k} based on whether they are actually frequent and above the minimum support threshold. This gives us L_{k}.

B. A-priori Algorithm in Pyspark

- The first step is to compute candidate tuples of size 1. This is basically all unique tracks across all playlists. Following this we need to prune all candidate tuples that fall below support threshold.
- The pruning step at each itemset size can be parallelised. The candidate tuples of size 1 are split into chunks and distributed over RDDs, each RDD calculates the support of each candidate tuple in its chunk and returns the candidate tuples that are above the threshold.
- This step remains the same at all itemset sizes (k $>$ 1), candidates are distributed over RDDs and those below the support threshold are pruned. The candidates returned are the frequent tuples of size k.
- Once we have frequent itemsets of size k, we need to construct candidates of size k+1. This is done based on the property that any subset of a frequent tuple is also a frequent tuple.
- Thus, if we have two frequent tuples of size k, which share exactly k-1 elements(and thus have a union of k+1 elements), then the union must be a frequent tuple of size k+1 as all its subsets are frequent.
- To implement this we check every pair of the frequent tuples of size k, and if they share k-1 elements, then their union is a candidate tuple of size k+1. This step has a time complexity of O(n^2). However, here n is the number of frequent tuples, which is usually less that 1000 in number which makes this feasible.
- Following this we repeat the pruning process to get the frequent tuples and then repeat with the candidate tuple generation.

C. FPGrowth

FPGrowth, or Frequent Pattern Growth, is a powerful algorithm used for mining frequent itemsets in large datasets. Developed by Jiawei Han and Jian Pei in 2000, FPGrowth efficiently discovers frequent patterns by employing a divide-and-conquer strategy. The algorithm builds a compact data structure called an FP-tree (Frequent Pattern tree) to represent the dataset and its frequent patterns. The FP-tree structure allows for efficient and recursive mining of frequent itemsets without the need to generate candidate sets, which is a common step in other algorithms like Apriori. By eliminating the generation of candidate sets and leveraging the FP-tree structure, FPGrowth significantly reduces the computational overhead associated with traditional methods, making it particularly well-suited for large-scale datasets.

The FPGrowth algorithm operates in two main steps. In the first step, it scans the dataset to construct an FP-tree, a compressed representation of frequent patterns along with their support information. In the second step, it recursively mines the FP-tree to extract frequent itemsets by traversing the tree structure in a depth-first manner. The algorithm's efficiency stems from the fact that it avoids generating candidate itemsets, which can be computationally expensive, especially when dealing with large datasets. FPGrowth is highly efficient and scalable algorithm.

Implementing this algorithm for this project, allows us to perform frequent itemset mining on a much larger subset of the original data in a much smaller amount of time. Using this, we were able to find frequent itemsets on 100k playlists. The frequent itemsets that contain more than 15 tracks can be thought of as a popularity based playlist which can be recommended to users as trending songs of the time.

VII. Integrating the Two Approaches

We have integrated content based recommendation and frequent itemset mining in different ways to improve the user experience.

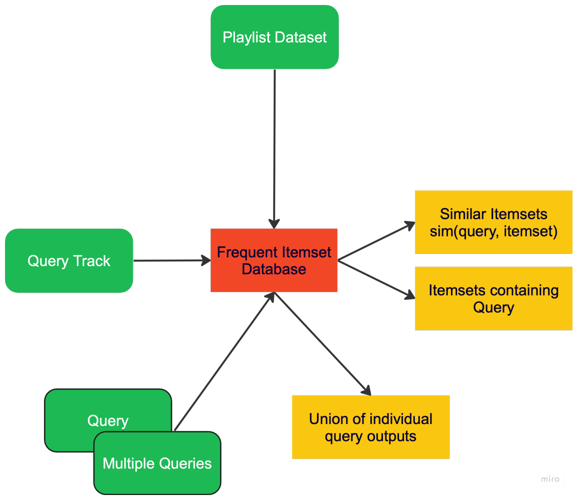
Playlist Dataset
Query Track
Frequent Itemset Database
Similar Itemsets sim(query, itemset)
Itemsets containing Query
Query
Multiple Queries
Union of individual query outputs
miro

Fig. 4. Various methods of using frequent itemset mining for recommendation

Firstly, using frequent itemset mining we have created a trending today section for the user, which acts a suggestion of tracks based on popularity among playlists. This recommendation is the largest frequent itemset of the database. This itemset is the largest set of songs that co-occurs in more than 0.5% of the 100k playlists used in the database.

Secondly, if a user input is a track which exists in a frequent itemset, then the entire frequent itemset is also returned as a recommendation along with the tracks recommended by the content based method. This allows to diversify recommendations and also include recommendations that are created by user based choices and not just embeddings.

VIII. Conclusion

In summary, our project accomplished the successful implementation of a content-based recommendation system tailored for music tracks. Leveraging audio and track features, we devised a sophisticated approach to create embeddings that were efficiently stored in Pinecone vector database. This greatly enhanced the performance of the content based recommendation.

Moreover, we significantly elevated the recommendation experience by incorporating frequent itemset mining techniques, specifically using the A-priori and FPGrowth algorithms. This integration allowed us to incorporate user patterns within the dataset based on their choice of playlist creations.

We successfully implemented A-priori algorithm from scratch in PySpark, ensuring parallelization across RDDs to optimize computational efficiency. Along with this implementation of frequent itemset mining, we also used PySpark's integrated FPGrowth algorithm implementation which allowed us to compute frequent itemsets on 100k playlists in a signifiicatly reduced time, allowing to make use of a large number of frequent itemsets.

We also developed a user-friendly frontend for our recommendation system. The frontend is based on streamlit and utilises all required algorithms and APIs in under one second. The pre-computation of frequent itemsets also meant that the frequent itemset algorithms are not time consuming and reducing user satisfaction. The front end also has a fully functional dynamic word search bar, that allows users to search song based on track, album or even artist name. The fully functional front-end with a robust back-end, makes this project a successful recommendation system.

IX. Deliverables

Project deliverables other than this report are listed below,

- Presentation & demonstration video:youtube-link
- Code - Uploaded on GitHub and Teams
- Presentation slide - Uploaded on Github and Teams This report is also uploaded to GitHub and Teams