Spotify Recommendation System Using Distributed Computing System

Guanchen Liu
College of Computer, Mathematical, and Natural Sciences
University of Maryland – College Park
College Park, MD
gliu0529@umd.edu

***Abstract*— The project aimed to develop a song recommendation system using the Spotify Million Playlist Dataset [1]. We compared the computational differences between a standard personal computer and cluster networks that use Spark as the framework. Our initial model employed simple collaborative filtering techniques on the dataset. Subsequently, we used various preprocessing techniques to reduce the dimensionality and size of the data. We enhanced our collaborative filtering model by integrating a clustering algorithm that utilized graph network. This approach created models that significantly outperformed our baseline while having the smallest overhead minimizing computational time and memory usage. Finally, we accessed the benefits of using Databricks Spark on a cluster network in comparison to running the same model on a personal computer.**

Keywords—recommendation system, spark, distributed computing, collaborative filtering, graph network

PAPER CODE

The source code for the research presented in this paper in available on GitHub. You can access the code here: https://github.com/hliu88/spotify_recommendation_pyspark

I. INTRODUCTION

In the age of rapid evolving content streaming services, content recommendation have become a cornerstone of any successful platform. A recommendation system can make and break a service, with a effective recommendation system enabling users to discover new content. This, in turn, captures user engagement, leading to increased time spent on the platform and better user retention, generating more revenue through subscriptions or advertisements. This project aims to develop a recommendation system, using the Spotify Million Playlist Dataset. The dataset was sampled of Spotify users in the United States, comprises one million playlists consisting of 300,000 artists with two million unique tracks. We will explore ideas and strategies for creating a recommendation system of this big data problem with data size of this magnitude.

As this project focuses primarily focuses on handling big data than employing different machine learning techniques, we will solely utilize the available dataset without additional data gathering through Spotify API for metadata information, such as song attributes(e.g., acousticness, danceability, liveness, loudness, etc.). This data can be used to create neural network models to recommend songs similar to those in a user's playlist. Additionally, we will not gather song information, such as lyrics, which would be used to create word embedding models and cluster songs in high-dimensional space for recommendations. Instead, we will use only the features available to use to create a collaborative filtering model, a fundamental yet effective technique.

An essential aspect of this study involves the exploration and comparison of computational capabilities between a stand personal computer and a cluster of computers. By using a cluster, we gain insights into the best practices and results of what an enterprise-level architecture and assess the performance benefits it offers compared to a single compute unit.

II. METHODOLOGY

A. Dataset

The dataset that is used in this study was sourced from the 2018 RecSys Chellenge, where the Spotify Million Playlist Dataset was sampled from over 4 billion playlists on Spofiy, consists of over two million unique tracks by 300,000 artists. The playlists in the dataset were created between the time of January 2010 and November 2017.

The entire dataset is organized in JSON format, comprising of a total of 1000 JSON files with a combined size of 33 gigabytes (GB). The JSON files follow a playlist structure, with each JSON entry containing information such as the playlist name, creation date, number of albums in the playlist, the total number of tracks within a playlist, and the number of artists in the playlists. Additionally, each playlist entry includes a list of tracks, where each track is associated with its respective artist, track, and album unique identifiers.

B. Preprocessing

Given the substantial size of this dataset, our first task was to reduce the file sizes to enable processing on a single computing unit. Attempting to load all 33GB of data simultaneously would inevitably exhaust memory resources and lead to runtime crashes. Since our objective did not require preserving the dataset's semi-structured format, which includes numerous symbols and strings used for structural purposes, we perform simplification of the data. Additionally, we decided to exclude irrelevant features, such as the number of tracks in a playlist or the number of followers for a playlist. Our primary goal was to create a collaborative filtering model, which necessitated simplifying the data to establish a straightforward relationship between playlists and the songs within them.

To achieve this, we began by converting the JSON files into CSV format. Essentially, we flattened each playlist so that each row contained the unique identifiers of a track and one of the tracks within that playlist. We performed this conversion iteratively, processing each JSON file and converting it into a CSV file. By the time we had converted all 1000 files, we had successfully reduced the dataset's size to a more manageable 5GB in total.

With the dataset processed and simplified, we will conduct exploratory analysis to gain a better understanding of its characteristics. While we lack individual song features to conduct in depth analysis, we need to know the distribution of the dataset.

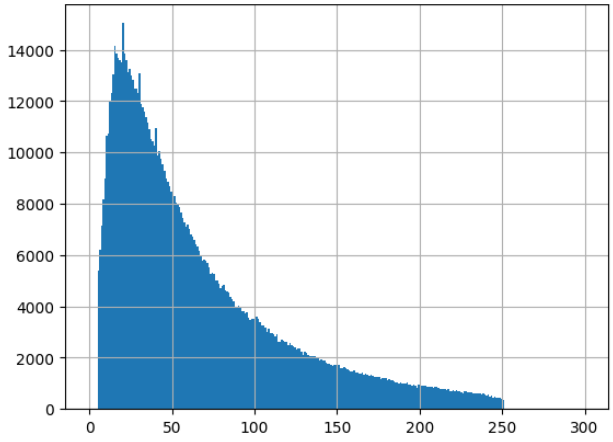

Fig. 1. Distribution of playlist length

We grouped the dataset by playlists and generated a graph depicting the distribution of playlist lengths. The graph revealed that the dataset comprised playlists with lengths ranging from 5 to 250 songs. Notably, there were numerous playlists with fewer than 25 songs. Short playlists, such as those with only 5 songs, do not provide sufficient information for meaningful recommendations. Moreover, evaluating our recommendation system with such short playlists would be impractical, as it would require inputting 4 songs to generate recommendations for the remaining 1 song, not enough samples to gauge the performance of the model.

To address this, we further processed the data by excluding playlists below a certain length and removing songs that appeared in fewer than a specified number of playlists. This operation had to be executed recursively since removing playlists might introduce songs appearing in fewer than 25 playlists, which needed to be eliminated as well. To achieve this, we constructed an undirected graph using the dataset, with vertices representing unique playlists and tracks. We then applied a k-core operation to the graph.

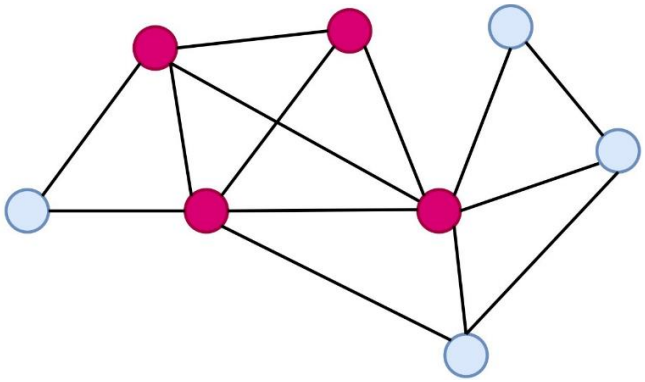

Fig. 2. 3-core of a graph

The k-core of a graph is the largest subgraph in which each vertex has at least k edges. In this paper, we generated two k-core graphs for evaluation: a 25-core and a 50-core graph. These graphs aimed to provide a dataset with robust inter-relationships among vertices. As Spark GraphFrames [2] lacked supported algorithms for k-core, we utilized NetworkX [3] to process the graph and compute the k-core.

TABLE I. DATASET SIZE

	Vertices	**Edges**
Original	2,482,051	65,031,793
25-core	874,684	56,153,746
50-core	506,623	42,990,995

From Table I, it is evident that the number of vertices has significantly decreased by a factor of at least 3, while the number of edges remains relatively consistent. This aligns with our goal of eliminating unused vertices, resulting in a graph that emphasizes closely related and interconnected relationships.

Once the k-core graphs were computed, we stored the edges into new files. These edges represent the relationships between playlists and tracks, precisely what is needed for collaborative filtering.

C. Baseline

For our baseline model, we used Spark's Alternating Least Squares (ALS) [4] to train our collaborative filtering model.

This Spark collaborative filtering method utilizes the matrix factorization technique, which is more effective than the traditional nearest-neighbor approach. This method enables the discovery of latent features that capture the underlying relationships between playlists and songs. What sets it apart from the classical approach is its ability to handle implicit feedback, which aligns perfectly with our problem.

Song recommendation differs from movie rating predictions, as each song in a user's playlist is not explicitly rated. There are no numerical ratings assigned to each track. Instead, we consider each song in the playlist as a song that the user likes, implicitly associating it with the user's preferences. Consequently, we introduced a column of 1s in our dataset to represent this implicit feedback problem, allowing us to model user preferences effectively.

D. Clustering with Graph

Even after reducing the dataset for our collaborative filtering method, it still contains a considerable number of entries, which can lead to significant computation time. Additionally, the current model is limited in its ability to handle new playlists or songs efficiently. When a new playlist is introduced or new songs are added, the existing model becomes obsolete, requiring complete retraining from scratch.

Collaborative filtering models typically struggle to adapt to new data. However, can we find a way to reduce the amount of data required for training the model so that we can generate recommendations swiftly when new playlists or songs are introduced?

To explore this concept, we initially visualized the distribution of tracks within the 50-core dataset as a graph to identify any clustering patterns among edges.

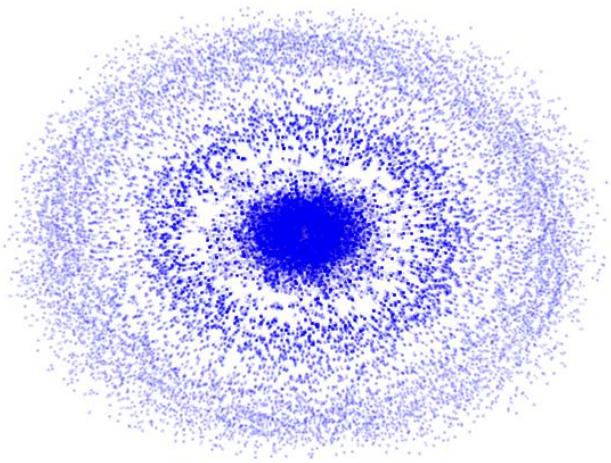

Fig. 2. Distribution of 500,000 random edges

Fig. 2 illustrates the distribution of 500,000 random edges. It indicates that the graph exhibits clustering. However, this graph alone does not reveal whether the clusters are formed around playlists.

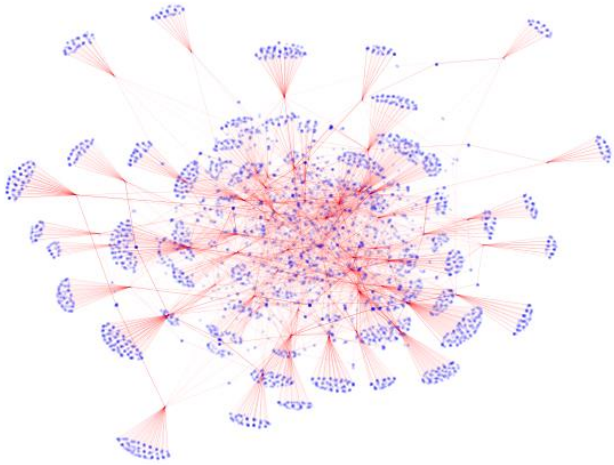

Fig. 3. Distribution of 10,000 edges by random playlists

Fig. 3, generated from 10,000 edges selected randomly, clearly demonstrates that the graph is indeed clustered by playlists. Playlists connected by one or two tracks appear on the outer edges, while those in the middle share more connections.

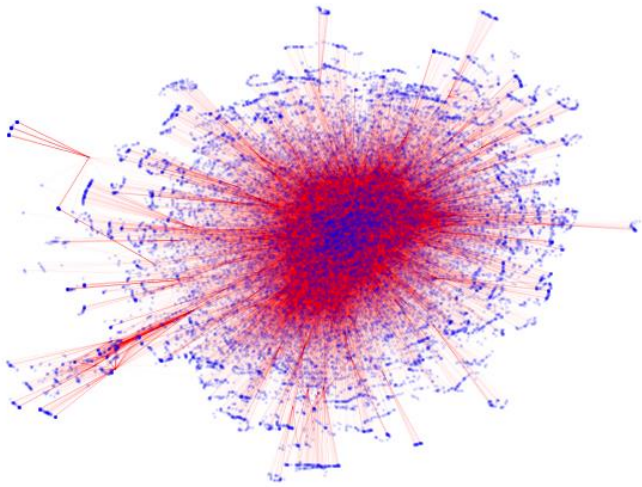

Fig. 4. Distribution of 100,000 edges by random playlists

In Fig. 4, using 100,000 edges from playlists, each vertex now represents a group of tracks forming a playlist. Although each vertex appears relatively small, we can still observe playlist clustering, with a majority situated in the middle and smaller clusters forming on the outer periphery. We don't need to be concerned about playlists with few connections to others, as the k-core operation has ensured that each track in a playlist has at least k edges.

Rather than training the collaborative filtering model on the entire database, we opt for a different approach to make predicting for a specific playlist. We cluster relevant playlists by identifying the nearest neighbors of all the songs in the playlist we intend to recommend. Since the nearest neighbor of a track is another playlist, we aggregate these neighboring playlists

together. However, this step alone may only reduce the number of playlists by up to 50 percent at worst. To further decrease the number of playlists, we count the number of times a unique playlist is in our aggregated list. The same neighbor playlist may appear multiple times in our list, as the playlist in question and the neighbor playlist might share more than one common song. We then set a threshold to consider only those playlists that share a specified minimum number of songs. This significantly reduces the amount of data used to train the collaborative filtering model.

E. Evaluation

To assess the performance of our model, we will use two metrics. The first metric is the root mean square error (RMSE) of the model. We will split the dataset into an 80-20 ratio for training and testing purposes. The RMSE value, which ranges between 0 and 1, will indicate how well the model predicts songs. Since our model addresses an implicit feedback problem, a smaller RMSE value signifies superior results in the recommendation system.

However, RMSE alone does not provide a comprehensive picture of model performance. Instead, we aim to directly evaluate the model's performance by generating predictions for playlists. To achieve this, we will generate 500 song recommendations for a specific playlist and determine how many of the test songs are correctly recommended from the list of 500 song recommendations. We will calculate the hit rate using the formula (1).

$$Hit\ Rate = \frac{Songs\ Hit}{Songs\ Hit + Songs\ Missed} \quad (1)$$

F. Comparison

This study extends beyond model performance evaluation; it also investigates the performance differences between working on a single compute unit and a web cluster. We will evaluate the runtime results of key operations in both environments to understand the advantages of parallel computing in the realm of big data.

III. RESULTS

A. Baseline

We assessed the baseline model's performance using both the 25-core and 50-core datasets. To optimize the model's performance, we conducted hyperparameter tuning via grid search on a random portion of the dataset, employing the parameters listed in Table II.

TABLE II. BASELINE HYPERPARAMETERS

maxIter	25
rank	50
regParam	0.1

The results of the baseline model for both datasets are presented in Table III. It is evident from the results that the performance of our baseline model falls short of our expectations. The hit rate for all playlists hovers around half, even when considering 500 recommended songs generated for each playlist.

TABLE III. RESULTS

	Baseline		**Graph Clustering**	
	RMSE	**Hit Rate (500)**	**RMSE**	**Hit Rate (50)**
25-core	0.891	0.435	0.0767	1
50-core	0.832	0.526	0.212	0.8

B. Clustering with Graph

After testing various threshold values for clustering playlists with the playlist in question, we found that setting the threshold to 15 shared songs yielded the best results. A small number of shared songs led to marginal performance improvements compared to the baseline, while increasing the threshold resulted in fewer matched playlists and, consequently, fewer unique songs in the model, leading to poorer recommendation results.

During the execution of the code with graph clustered data, we encountered significant computation cost in data preparation taken place before training the collaborative filtering model. Upon investigation, we identified that the outliers in the dataset, specifically playlists with 100 or more tracks, were causing substantial delays. The clustering algorithm produces a list of clustered playlists, but retrieving the tracks within these playlists from the original dataset proved to be time-consuming. The filtering operation used for this purpose took an excessive amount of time to process entries in large playlists. While there were significantly less long playlists compared to shorter ones, the combined number of songs in these larger playlists was substantial. To address this, in addition to the k-core operation, we decided to remove all playlists with a length of 75 tracks or more.

This adjustment significantly reduced our computation time and, notably, improved our model's performance. The performance boost was so substantial that evaluating the clustering models using a hit rate with 500 recommendations was no longer appropriate. Instead, we will evaluate the models with 50 generated recommendations. For 100 randomly selected playlists, the average hit rate for the 25-core dataset reached 100%, while the 50-core dataset achieved 80%. Additionally, the RMSE values of both models outperformed the baseline significantly. The 25-core model, in particular, achieved an outstanding score of 0.212, representing a significant improvement over the baseline's 0.8 and a great improvement from clustering 25-core result of 0.7.

C. Computation Comparison

The computational experiments were conducted on two different computing setups. The results from a single computer device were obtained using a MacBook Pro equipped with an Apple ARM M1 Max CPU and 32GB of RAM. In contrast, the cluster network results were computed using AWS EC2, utilizing a driver instance of i3.2xlarge with 8 cores and 61GB

of memory, along with six concurrent worker instances of i3.xlarge, each equipped with 8 cores and 30.5GB of memory.

The baseline ALS model benefited significantly from parallel computing due to the dataset's size. Having concurrent workers markedly reduced the training time.

TABLE IV. BASELINE ALS MODEL TRAING TIME

	Personal Computer	**Cluster**
25-core	90m 24s	12m 50s
50-core	79m 40s	10m 43s

The clustering model demands additional computational resources prior to training the collaborative filtering model. For the case of neighbor playlists, the "populate playlist" function is invoked, initiating queries that filter the entire dataset to identify entries matching the playlist ID. This process requires continuous memory access, and the ability to concurrently access and process data significantly accelerates the operation.

TABLE V. POPULATE PLAYLIST COMPUTE TIME

	Personal Computer	**Cluster**
Time	38m 52s	58s

Table V demonstrates that when performed on a single compute cluster, the entire operation can be completed in under one minute. However, when using a single compute unit, the operation took approximately 40 minutes to finish. It's important to note that this time was recorded after filtering out playlists with 75 songs or more in the 25-core dataset. Prior to the filtering step, executing the same operation on a compute cluster took over 10 minutes, while the personal computer took an impractical amount of time, leading to a manual cancellation of the operation.

Test

Comparing Table IV and Table VI may appear counterintuitive at first glance. In Table IV, the baseline training time is shorter for the 50-core dataset, while in Table VI, the clustered training time is shorter for the 25-core dataset. Upon closer examination, this discrepancy can be justified.

For the baseline model, the 25-core dataset contains a larger number of entries. The 25-core dataset includes playlists ranging from 25 to 200 tracks. However, in the case of the graph clustering data, the 25-core dataset comprises playlists with 25 to 75 tracks that share 15 tracks with our playlist of interest.

In contrast, for the 50-core dataset, there are more playlists that can be matched. The larger playlists in the 50-core dataset make it easier to match playlists that have 15 shared songs compared to the 25-core dataset. This helps explain why the computational time for the graph clustering model is longer for the 50-core dataset.

Adjusting the 50-core data with the specific aim of increasing the threshold value beyond 15 shared songs revealed an interesting outcome. Contrary to expectations, increasing the threshold led to a decrease in performance. Moreover, it resulted in a dramatic reduction in the number of playlists, adversely affecting the model's performance due to the limited number of available samples.

The graph clustered processed data, being considerably smaller compared to the baseline dataset, significantly reduces computation time when executed on a personal computer. However, the computation time on the cloud is even faster. When provided with a playlist for recommendation, the entire pipeline processes and generates relevant playlists, initiates the model, and produces recommendations in less than 2 minutes. With horizontal and vertical scaling, it is possible for the entire pipeline to complete within a few seconds. Regardless of the specific setup, the performance increase observed was substantial, especially when compared to the baseline training time.

TABLE VI. GRAPH CLUSTERING ALS MODEL TRAING TIME

	Personal Computer	**Cluster**
25-core	4m 22s	43s
50-core	11m 8s	1m 52s

IV. CONCLUSION

In conclusion, this project successfully developed a recommendation system capable of accurately predicting songs based on a given playlist. Not only did it improve upon the baseline collaborative filtering method, but it also reduced the amount of data required for model training, significantly enhanced model performance, and created a generalized solution that can be applied to up-to-date data without modifications.

Furthermore, the collaborative filtering model designed in this project is lightweight and requires minimal training time. In a production environment, it only necessitates a graph-structured database containing playlists and associated tracks. With the aid of a powerful compute cluster, the entire recommendation pipeline can generate personalized song recommendations within seconds. This advantage is particularly significant in scenarios where new data entries are regularly added to the database, as the lightweight nature of the model minimizes the overhead associated with updating recommendations.

The project's algorithm can be seen as a combination of nearest-neighbor collaborative filtering and matrix factorization collaborative filtering. It implicitly identifies similar playlists by seeking playlists with shared songs and subsequently employs matrix factorization to improve song predictions.

Beyond the development of the recommendation system, this project highlighted the importance of runtime optimization and time complexity when dealing with big data operations where parallel computing may not be applicable. The difference between a linear solution and an exponential one becomes crucial when dealing with large-scale operations. For instance, the pairwise reduce function in the graph clustering model demonstrated the significance of optimizing time complexity. Instead of iteratively combining dataframes in a nested fashion, a more efficient approach was found by storing dataframes in an array and merging them in a way that reduced computational complexity.

$$((((\big((a+b)+c\big)+d)+e)+f)\ldots \quad (2)$$

$$(a+b)+(c+d)+(e+f)\ldots \quad (3)$$

Instead of combining iteratively each dataframe in the format of (2), it is better to merge the lists in the format of (3).

Overall, this project demonstrates how careful consideration of data processing techniques, algorithm design, and efficient use of computational resources can lead to the development of a high-performance recommendation system suitable for real-world applications.

REFERENCES

[1] "AIcrowd | Spotify Million Playlist Dataset Challenge | Challenges," *AIcrowd | Spotify Million Playlist Dataset Challenge | Challenges*. https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge#references (accessed Dec. 15, 2023).

[2] [1]"Overview - GraphFrames 0.8.0 Documentation," graphframes.github.io. https://graphframes.github.io/graphframes/docs/_site/index.html

[3] "k_core — NetworkX 3.1 documentation," *networkx.org*. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.core.k_core.html

[4] "Collaborative Filtering - Spark 2.2.0 Documentation," *Apache.org*, 2019. https://spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html

[5] [1]"Spark SQL — PySpark 3.5.0 documentation," spark.apache.org. https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html