Spotify Song Recommender

Jong Inn Park park2838@umn.edu
Jooyong Lee lee02628@umn.edu
Omavi Collison coll1396@umn.edu

1 Introduction

Revenue in the Music Streaming segment is projected to reach US$ 13.66 billion in 2022. In global comparison, most revenue will be generated in the United States (US$ 5,456.00 million in 2022)'.[1] As we can see from this sentence, a music streaming service business, such as Spotify, Youtube Music, and Apple Music, is a vast market. Especially, Spotify ranks 2nd among the podcast platforms in the United States, with 40% of people using it regularly. Among many different parts of their business, the song recommendation system is critical to the music streaming service business. Therefore, data consistency and efficient data flow are essential to offer a better recommendation to streaming service subscribers, and, to have a deeper understanding of this process, our group decided to implement such data flow in this project. More specifically, our group will structure a database containing a music dataset scraped using Spotify Application Programming Interface (API) and train a recommendation model with such data. Finally, we will recommend adequate tracks in terms of user input.

2 Dataset Construction

2.1 Kaggle Dataset

First, we downloaded the Spotify playlists dataset from Kaggle[2], an online community of data scientists and machine learning practitioners. It consist of 10,000,000 playlists which contain 2,262,292 unique tracks' id, 734,684 unique albums' id, and 295,860 unique artists' id.

2.2 Web Scraping using Spotify API

There are several steps for scraping track data using Spotify API.

Step 1. Extract track information from the raw playlist data. We need the keys for Spotify API, which are track_uri, artist_uri, and album_uri. And they are 22-length combinations of characters and numbers. As shown in Figure 1 The extracted data's structure will be the dictionary, and the key will be track_ids.

Step 2. Set up a Spotify account and get an authentication token, which is a credential key when access to Spotify's API.

Step 3. Make mini-batches of each type (track, artist, album), request them to Spotify API, and get responses.

Step 4. Save the whole JSON data with a dictionary structure. As shown in Figure 1, the track_id is also used as a key in this data. [3]

2.3 Raw Data

Using Spotify API, we scraped all the details of tracks, albums, and artists listed in the Kaggle playlist dataset. As shown in Table 1, we constructed four different raw datasets: Tracks, Albums, Artists, and Audio Features, and each dataset is structured with several levels of attributes. Attributes listed in the table are only part of all.

We proceeded with exploratory data analysis for all datasets and observed some characteristics of each dataset. As shown in Figure 2, the popularity distribution of tracks (the very left graph) follows an exponential distribution; we can tell that there are more tracks that have lower popularity scores than tracks that have high popularity scores. From the graph of the released year of albums, we observe that most of the albums in the dataset were released during the 2000s. The popularity distribution of artists follows the normal distribution.

[1] https://www.statista.com/outlook/dmo/digital-media/digital-music/music-streaming/worldwide
[2] https://www.kaggle.com/discussions/general/60644
[3] https://developer.spotify.com/documentation/web-api/

Figure 1: Extracted track information, scraped track information, and Spotify Web API documents

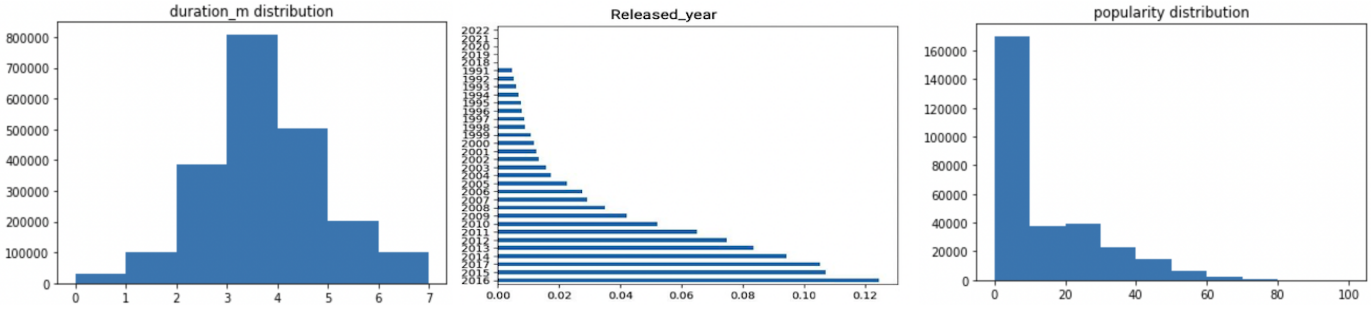


Figure 2: Popularity of tracks, released year of albums, and popularity of artists

As shown in Figure 3, we drew a bar graph of genres in descending order, and we observe that there are many subgenres in genres, for example, there are 'country rock,' 'pop rock,' 'soft rock,' 'indie rock' in a big boundary genre 'rock.'

Datasets	1st-level attributrs	2nd-level attributes
Tracks	album artists linked_from disc_number, duration_ms, external_ids, external_urls, ...	album-type, total_tracks, available_markets, id, name, release_date, artists ... genres, id, name, popularity, type, imaged, ... album, artists, disc_number, duration_ms, id, ...
Albums	album_type artists copyrights tracks external_ids, genres, name, popularity, release_date, total_tracks, ...	 external_urls, id, name, type, uri, ... text, type, ... artists, disc_number, duration_ms, id, name, ...
Artists	external_urls followers genres, id, name, popularity, type, url, ...	spotify href, total
Audio_features	acousticness, analysis_url, danceability, duration_ms, energy, id, instrumentalness, ...	

Table 1: Details of a raw data.

2.4 Data Pre-Processing

Two main parts of data pre-processing are feature selection and flattening of each sample in datasets which include nested dictionaries. In the feature selection part, we first ensured that each entity (track, album, artist, audio feature) contains other entities' ids as attributes. Then those ids can be foreign keys connecting different tables. Next, we considered which features would be helpful for the training recommendation model, and we decided to keep most of the numerical features such as disc_number, duration_ms, total_tracks, popularity, acousticness, and instrumentalness.

2.5 Final Dataset

Final datasets before converting to table contain the same number of tracks, albums, and artists as raw datasets. As shown in the table2, each dataset was flattened, and track, album, artist, and audio feature datasets contain 9, 4, 4, and 10 attributes, respectively.

Since there are too much track data, we needed to cut off the number of tracks. There are many sampling standards, but we chose the release year and the album popularity of a track, considering recommending well-known tracks to users. In the release year, the 2010-2016

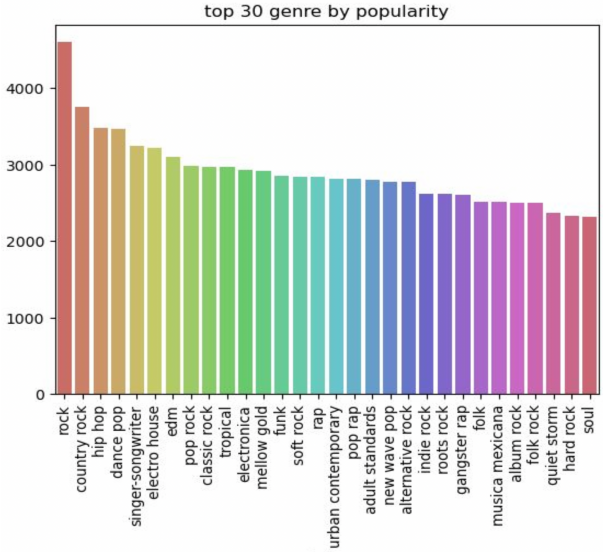


Figure 3: Popularity of genres

Datasets	attributrs
Tracks	track_id, name genres, disc_number, album_id, popularity, duration_ms, time_signature, track_number, explicit
Albums	album_id, album_popularity, release_date, genres
Artists	artist_id, name, genres, artist_popularity
Audio_features	track_id, tempo, valence, danceability, acousticness, energy, instrumentalness, liveness, loudness, speechiness

Table 2: Details of final datasets

bin still has more than 50% songs, which was enough to recommend a bunch of songs. And we dropped songs with an album popularity of less than 30, leading to 89,307 sampled tracks.

In Figure 4, overall, there is no significant difference in distribution between raw and sampled data. However, the distribution of artists' popularity changed from exponential to normal distribution. This is because we only took relatively popular tracks. Artists with low album popularity tend to have low artist popularity. Thus, sampling removed many artists with low artist popularity and balanced the number of artists on both ends.

3 Database Construction

3.1 Entity-Relationship Diagram (ERD)

Before building and overall coding our Database schema for our project, we created an ER Diagram5 in order to first visualize our database schema and accordingly plan what we are going to do. It helped us properly plan the structure of our project; by building the ER Diagram, we were able to perfectly declare our main entities and tables as well as declare their keys and relationships with each other.

With this, we declared our four main entities, Artist, Album, Track, and Audio Feature. As well as relationship Releases, Saves_ID, Contain and Constitutes. Our final draft of our ER Diagram went as such: The *Artist* entity *Releases Album* which *Contain Track* that *Constitute Audio Features*. Furthermore, in our visualization, we decided to give the relationships *Releases* and *Saves_ID* their own attributes theorizing that these would be made into tables that connect the album and artist as well as the track and artist tables together.

This perfectly encapsulated the data we scraped that contained items of artists who own items of tracks and albums which share arrays of tracks with audio features. Other intricacies include the *genre* attribute being multi-valued as artists, tracks, and albums may have numerous types of genres represented. Furthermore, after deliberation, we decided to make *Audio Feature* a weak entity as we concluded that it only exists because of the inclusion of the Track entity in the ER Diagram. The ER Diagram is shown in Figure 5 below.

3.2 ERD to Schema

With our ER Diagram plotted, we began mapping the visualization to our SQL code and began to create our database. To create our tables, we utilized the MySQL workbench interface. In creating our tables, we renamed the *Releases* and *Saves_ID* relationships to *album_artist*

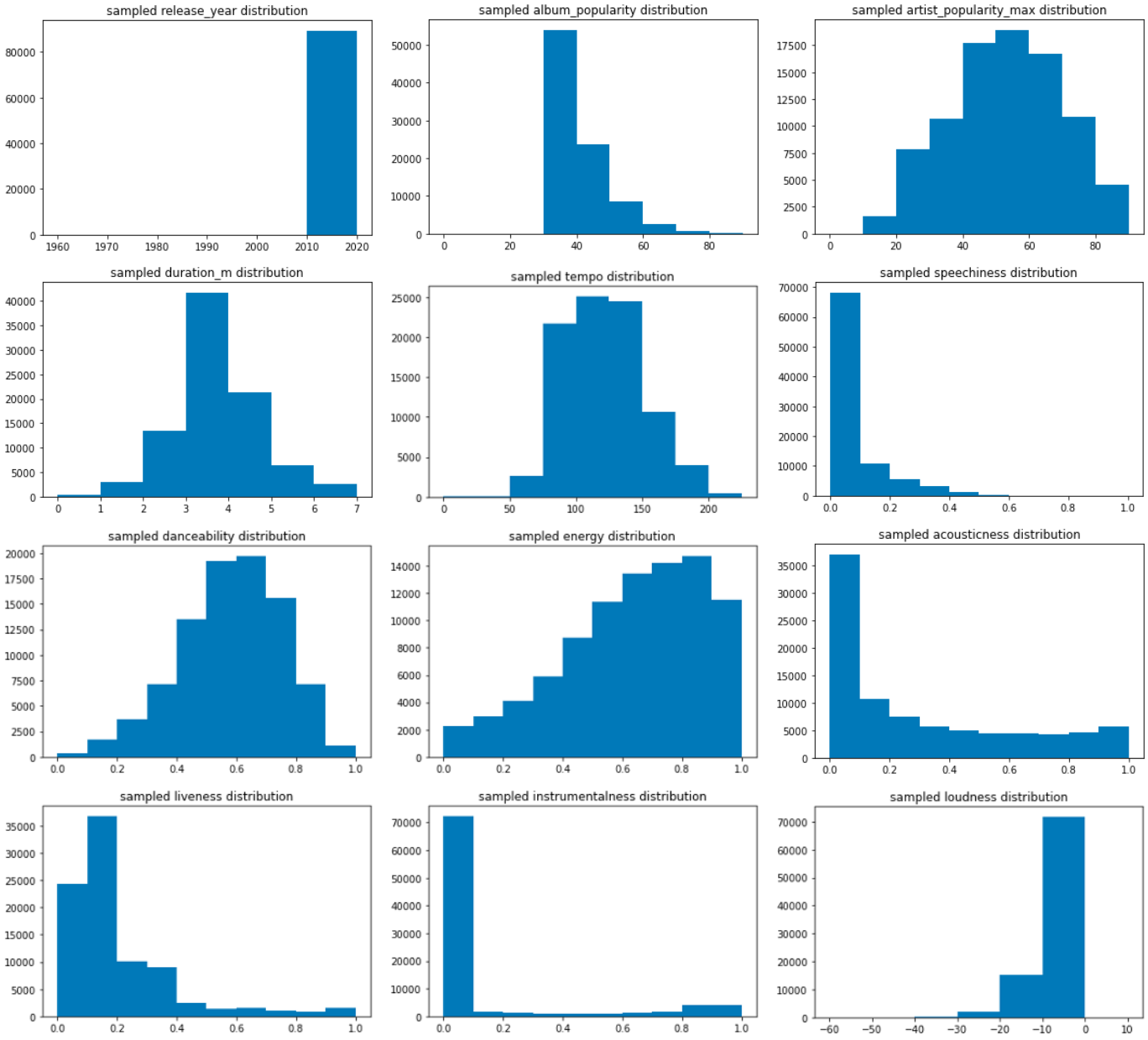
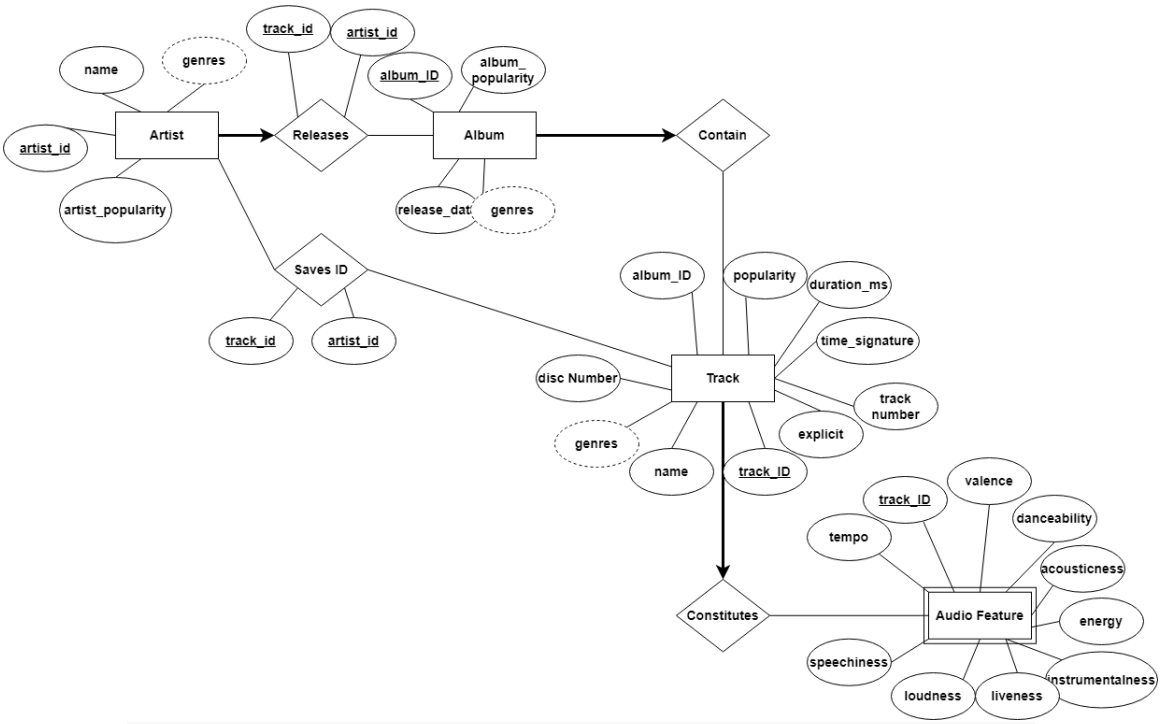


Figure 4: Distributions of attributes in sampled dataset



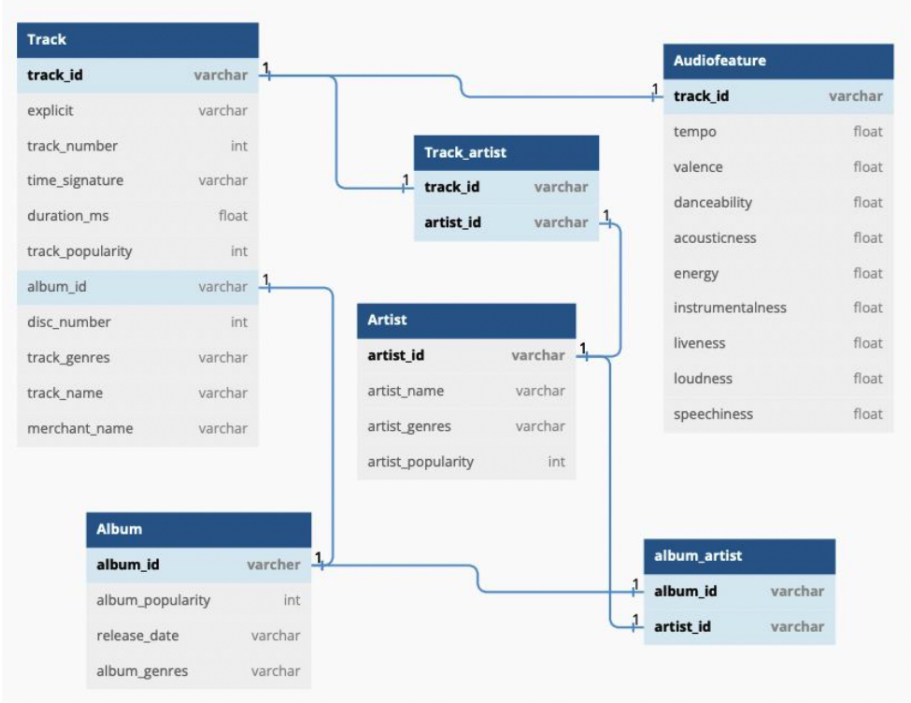


Figure 5: ER Diagram and Schema

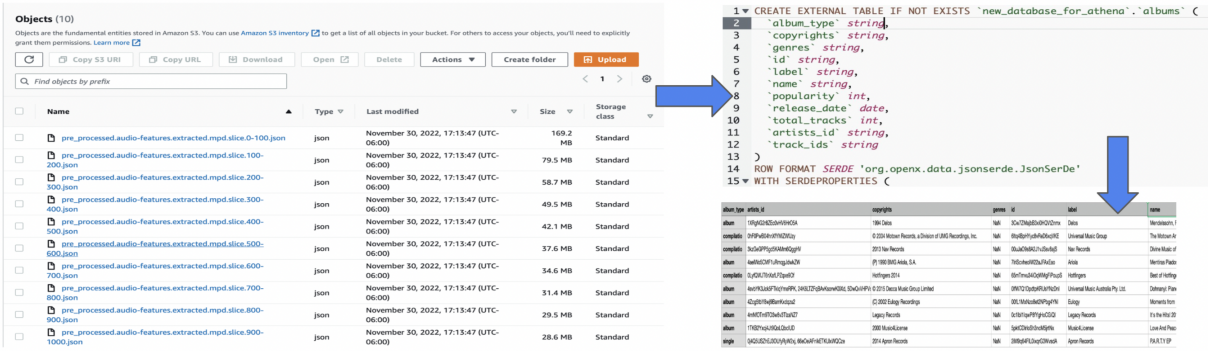


Figure 6: Process to create a table of album data using S3 and Athena

and *track_artist* accordingly. These tables then held the foreign keys for artist, album, and track as they connected the two pairs of tables together. A snippet of the schema can be seen in Figure 5.

3.3 Creating Tables with AWS S3 and AWS Athena

As a shareable data storage for final datasets, we decided to use AWS S3. The biggest reason we decided to use S3 is that it is easy to extract data from there using AWS Athena as seen in Figure 6. As long as the format of each sample is flattened, we can make the desiring table using the MYSQL query fast (it took less than 6 seconds to convert 424.6 Mb of JSON file to table in a CSV file).

3.4 Relational Data Base: Local MySQL server

In the midst of formatting our MySQL Databases utilizing AWS, we realized were being charged an exorbitant amount for their database hosting services. So, in the interest of finances, we decided to go with a Local MySQL Server which would be hosted on one of our Laptops.

Connecting to a local database was synonymous with connecting to a database hosted by AWS. The only changes we made were to the Database object's parameters, where the hostname, user, pass, and name were changed. Afterward, the connection was seamless, and we were able to interact with the Database in our Python environment with the cursor object and commit method.

3.5 MySQL connector: Python IDE

Python was the main programming language used throughout this project. Therefore we had to find a way to connect our MySQL database to our python environment, so we can write SQL code to our database from python, grabbing data from our database to incorporate into our code.

In order to do this, we had to install and the my-SQL-connector for python. Firstly we used the *pip install* command in our command prompt to install the packages, and then we imported the packages to our code and began to link our database.

We connected and created our Database object using the parameters of our database, inclusive of the: host, user, password, port, and database name. Once the database object was initialized, the cursor object was created through it. With the cursor object, we can write SQL code to be executed to our database within our Python Environment. Finally, we can use the commit method of our Database object to actually commit the changes, or SQL queries ran through the cursor directly to our database. This process is exemplified in Figure 7 below.

4 Recommendation

4.1 K-Means Clustering

K-Means clustering is one of the unsupervised algorithms that is mostly used. We can label the observations into K clusters with the nearest mean by this method.[4] It is very useful when we don't have labels in the training dataset.

Since we don't have any users' log data, we cannot recommend songs based on their preferences. So instead, we intended to recommend tracks with only their metadata and audio features. One method recommends songs that are geometrically close to another one with

[4]https://en.wikipedia.org/wiki/K-means_clustering

```
import mysql.connector

myDatabase = mqsql.connector.connect(
    host='localhost',
    user='admin',
    password='pass',
    port='3306',
    database='dbName'
)
```

```
cursor = myDatabase.cursor()
query = "SELECT * " \
        "FROM a " \
        "WHERE release_date-'2016-11-25' " \
        "ORDER BY album_popularity " \
        "DESC limit 10"
cursor.execute(query)
myDatabase.commit()
```

Figure 7: Connecting Our MySQL Database to our Python IDE

euclidean distances. However, in this system, we can only input songs that are in the prepared dataset. Also, it only gives us fixed results, which can be felt boring to users. On the other hand, using the K-Means clustering algorithm can provide users with randomness, which can prevent users from leaving.

Furthermore, we built the model considering the genres of artists. In other words, we created K-Means clustering models by each genre. Without genres, the recommendation would be too random. We assumed that users at least wanted to get freshness within the genres of that songs. However, Spotify provides us only an artist's genres, not ones of the track. It is reasonable because it is hard to define the genre of songs these days. So instead, we used the artists' genres, and there can be several artists in one track. Also, one artist can be categorized into several genres. Thus, one track can have many genres, meaning one track can be used more than once to build the models. In addition, Spotify provides a lot of subgenres, so we must group them into larger categories. We selected 15 genres: rock, hip hop, pop, jazz, blues, country, metal, reggae, folk, soul, EDM, dance, Latin, funk, and others. And we categorized all genres, for example, putting k-pop into the pop category. Thus, using K-Means clustering, we've trained the model.

4.2 Model Architecture

As shown in Figure 8, we built models by each genre. That means we should use different scales in each model. Also, we have to decide on the hyperparameter K, the number of clusterings, in each model. To do this, we used the elbow method, which picks the point whose direction changes rapidly as K by plotting the error against K. [5] According to Figure 10, all graphs shows elbow points are in the range from 60 to 110. Thus, we picked 60 as the K for the datasets with less than 10,000 rows and 110 for the datasets with more than 10,000 rows. We used Python's scikit-learn package, which is the popular one, and the building process was like the one below Figure 9.

After training, we can label all tracks with specific clustering numbers in each model. Therefore, when new input songs come into the system's entrance, it gives them to corresponding models. Then, we get predicted clustering numbers and can call related datasets. Finally, we select one random sample from datasets, as shown in Figure 8. In addition, the example in Figure 8 recommends only one track, but if the input song has several genres, then the system can recommend several tracks.

4.3 Test

We have to select test inputs first. Since we don't have any performance measurements, our performance depends on users' listening. Thus, we picked songs that are well-known or recently released. In Figure 12, we chose Taylor Swift's Midnight Rain and Jung Kook's Dreamers, famous artists who recently released albums. Also, we decided on Eminem, Coldplay, and Drake's songs, expecting the system to recommend tracks with similar moods because their music styles are very bold.

During prediction, we assumed we already knew the track_ids, but we need information on those new inputs. Thus, we requested the data to Spotify API first, like Figure 11. Also, we called the Python scaler object and rescaled the test set's features. Finally, we got eight tracks recommended. Due to randomness, we got songs that have different language lyrics. Among recommendations, we found that three songs have similar moods: Taylor Swift's Midnight Rain and Adam Lambert's Rumors, Eminem's Lose Yourself and Yelawolf's Get Mine, and Jung Kook's Dreamers and Lionel Garcia's Te Bese.

5 Conclusion And Future Work

Ultimately, we achieved what we sought to do and were able to make a sound Spotify song recommendation system that recommends music of similar taste when given a track. There wasn't any clear-cut conclusion we sought out to make from this project but we gained many insights into Database, Spotify, Spotify API, and SQL. This workflow was as such:

[5] https://en.wikipedia.org/wiki/Elbow_method_(clustering)

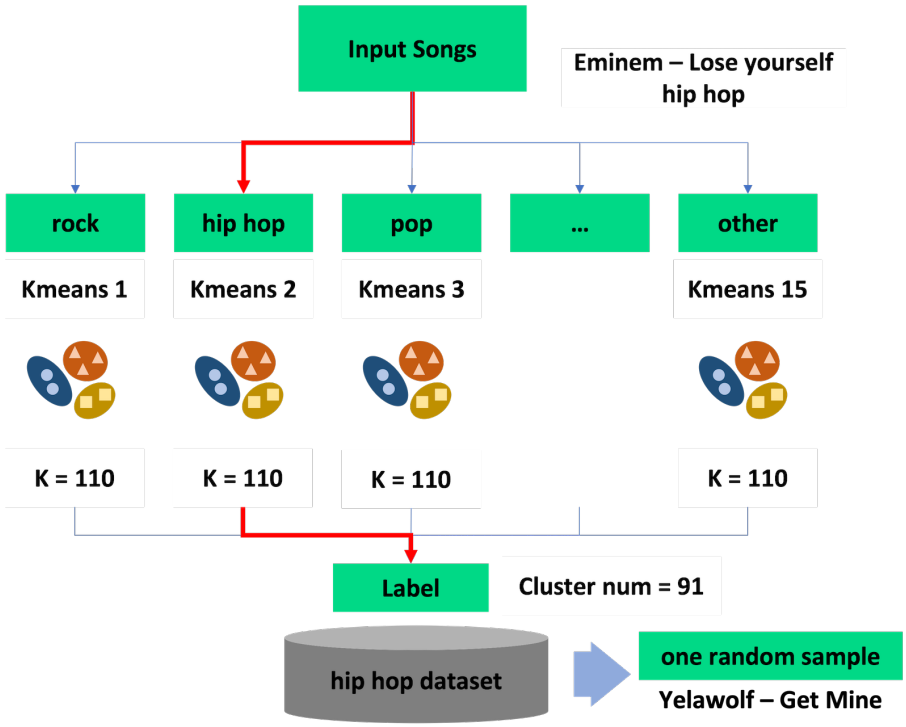


Figure 8: Recommendation Model Architecture

Firstly we got raw data from an online Kaggle dataset, and then scraped all the other data we needed using the python algorithm made possible with Spotify Web API. We received the data as JSON files which we pre-processed removing unnecessary items from the JSON arrays as well as formatting the data in a way so that it can be processed into AWS Athena. After we had our pre-processed JSON files, we stored them in an AWS S3, exporting them from the S3 to AWS Athena where the data was converted into CSV files. With CSV files, we were able to use MySQL's built-in function to convert properly formatted csv files into tables. The biggest takeaway insight of this project was the existence of the Audio Features within tracks being discovered as we scraped data.

The Audio Features of the track are used to classify exactly what type of song you are listening to from a musical standpoint. It wasn't available to the average Spotify user, only to developers who actively seek the data. It also became the crux of our Spotify Song Recommendation algorithm.

Still, we believe that there are more rooms that we can improve the project by proceeding with it more, For example, we can systematize the following process: 1) Scraping music data, 2) Storing it in the Database, 3) Query data for the training model, 4) Recommend tracks in terms of user input. To achieve this, building a data pipeline that can implement efficient data flow is essential, as we mentioned in the introduction.

Accordingly, in the future, we plan to continue working on this project in order to streamline the aforementioned progress so we can ultimately make our own personal Spotify Song Recommendation system that we can use anytime!

```python
kmeans_kwargs = {
    "init": "random",
    "n_init": 10,
    "max_iter": 300,
    "random_state": 2022,
}

kmeans_model_list = []

for genre, k in k_parameter_list:
    print(f"genre: {genre}, k: {k}")
    for df_name, scaler, scaled_features in scaled_features_list:
        if genre == df_name:

            kmeans = KMeans(n_clusters=k, **kmeans_kwargs)
            kmeans.fit(scaled_features)
            kmeans_model_list.append((genre, copy.deepcopy(kmeans)))
```

✓ 10.9s

```
genre: rock, k: 110
genre: hip hop, k: 110
genre: pop, k: 110
genre: jazz, k: 60
genre: blues, k: 60
genre: country, k: 60
genre: metal, k: 60
genre: reggae, k: 60
genre: folk, k: 60
genre: soul, k: 60
genre: edm, k: 60
genre: dance, k: 60
genre: latin, k: 60
genre: funk, k: 60
genre: others, k: 110
```

Figure 9: Building the model with scikit-learn package

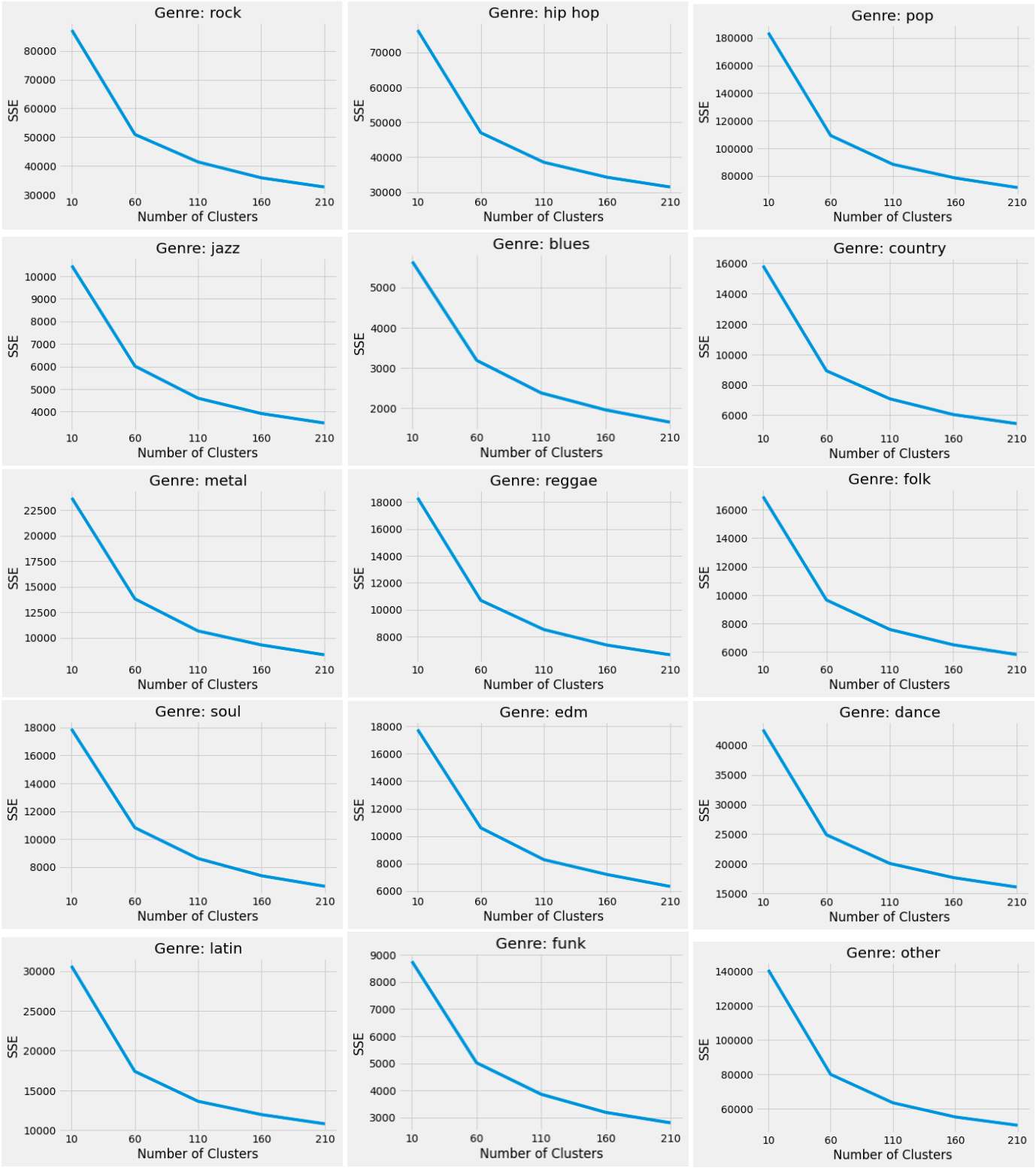


Figure 10: SSE plots of each genre

```python
ids_list = [
    ("coldplay", "viva la vida", "1mea3bSkSGXuIRvnydlB5b"),
    ("taylor swift", "midnight rain", "1DAHLigfUqSLrU3RLG1EKR"),
    ("eminem", "8 mile", "2552neNWWQ2i2XZwPi5B3G"),
    ("eminem", "lose yourself", "77Ft1RJngppZlq59B6uP0z"),
    ("drake", "hotline bling", "0wwPcA6wtMf6HUMpIRdeP7"),
    ("jung kook", "dreamers", "1RDvyOk4WtPCtoqciJwVn8")
]

spotify = SpotifyAPI()
spotify.get_token()


response_tracks = spotify.get_query_by_ids("tracks", [tup[2] for tup in ids_list])
response_audio_features = spotify.get_query_by_ids("audio-features", [tup[2] for tup in ids_list])
artists_list = [track["artists"][0]["id"] for track in response_tracks["tracks"]]
response_artists = spotify.get_query_by_ids("artists", artists_list)


test_row_list = []

for tup, response_dict in zip(ids_list, response_audio_features["audio_features"]):
    tmp_dict = {}
    tmp_dict.update({"track_id": tup[-1], "track_name": tup[1], "artist": tup[0]})
    tmp_dict.update(response_dict)

    test_row_list.append(tmp_dict.copy())

test_df = pd.DataFrame(test_row_list)
test_df = test_df[["track_id", "track_name", "artist"] + col_list]
```

✓ 0.4s

```
response status code: 200
```

```python
artist_row_list = []

for tup, response_dict in zip(ids_list, response_artists["artists"]):
    tmp_dict = {}
    tmp_dict.update({"track_id": tup[-1]})
    tmp_dict.update({"genres_combined": ",".join(response_dict["genres"])})

    artist_row_list.append(tmp_dict.copy())

tmp_df = pd.DataFrame(artist_row_list)
test_df = pd.merge(test_df, tmp_df, how="left", on="track_id")

test_df["genre_list"] = test_df.genres_combined.str.findall("(" + "|".join(genre_list) + ")").apply(set)

display(test_df)
```

✓ 0.6s

	track_id	track_name	artist	acousticness	danceability	energy	instrumentalness	liveness
0	1mea3bSkSGXuIRvnydlB5b	viva la vida	coldplay	0.09540	0.486	0.617	0.000003	0.1090
1	1DAHLigfUqSLrU3RLG1EKR	midnight rain	taylor swift	0.72300	0.638	0.369	0.000053	0.1150
2	2552neNWWQ2i2XZwPi5B3G	8 mile	eminem	0.12500	0.740	0.940	0.000083	0.1320
3	77Ft1RJngppZlq59B6uP0z	lose yourself	eminem	0.00922	0.689	0.735	0.000720	0.3650
4	0wwPcA6wtMf6HUMpIRdeP7	hotline bling	drake	0.00258	0.891	0.628	0.000190	0.0504
5	1RDvyOk4WtPCtoqciJwVn8	dreamers	jung kook	0.15800	0.710	0.879	0.001420	0.4390

Figure 11: Scraping test set information

Inputs

Artist	Track	Track_id
Coldplay	Viva La Vida	1mea3bSkSGXuIRvnydlB5b
Taylor Swift	Midnight Rain	1DAHLigfUqSLrU3RLG1EKR
Eminem	8 mile	2552neNWWQ2i2XZwPi5B3G
Eminem	Lose Yourself	77Ft1RJngppZlq59B6uP0z
Drake	Hotline Bling	0wwPcA6wtMf6HUMpIRdeP7
Jung Kook	Dreamers	1RDvyOk4WtPCtoqciJwVn8

Outputs

Artist	Track
A.R. Rahman	Magudi Magudi
Adam Lambert	Rumors
Gilli	Penge Kommer Går
Yelawolf	Get Mine
Jack Harlow	Magazine
Leonel Garcia	Te Bese
The Girl and The Dreamcatcher	Gladiator
Erica Kane	Speaker Knockerz

Figure 12: Test sets and recommendation results