Spotify Music Recommendation

Ali Shibli, Carina Wickstrom

February 2, 2022

1 Introduction

Music floods the soul with positive energy, and listening to ones favourite melodies only does it better. The topic of our project is music recommendation based on Spotify data. Our project stresses on the benefit of distributed computing for large-scale computing problems, where local hardware and memory are insufficient for such tasks. We run on the cloud with more than 32GB memory, for over 400,000 songs. The program displays a list of songs to the user, with a provided graphical user interface (GUI), gets some feedback of liked songs from the user, and finally recommends new songs based on what the user liked.

The program displays 10 songs to the user, and the user replies with which songs they like. Based on the user feedback, new recommendations are suggested iteratively. For recommendation, we implement 2 methods: one that computes closest distances between a user vector and all other song vectors in the dataset using standard deviation of each feature column, and the other methods uses kmeans with similar distance metric computations. More details will be presented in section 4.

2 Dataset

Spotify dataset consists of playlists of songs played by users on the Spotify app during an interval of time. It contains more than 1 million playlists, grouped as slices of json files, where each json file contains a bunch of meta data about each songs in the playlists. Each song in the playlist has a unique URI that identifies it. Since we need to retrieve data from Spotify Database, we retrieve the song URIs for more than 500,000 songs, scraping alongside the following set of features (or meta-data) using the Spotify API:

1. danceability
2. energy
3. key
4. loudness
5. mode
6. speechinesss
7. acousticness
8. instrumentalness
9. liveness
10. valence
11. tempo

Our dataset can be found here: https://drive.google.com/drive/folders/1A-lXC3y66_RJvui-uRj_qVriZi5g06jO?usp=sharing

3 Tools

We make use of the following tools to achieve our goal:

1. Spotify API: query songs from Spotify and build our dataset.
2. PySpark: use a SparkSession to read the input csv files of songs as a distributed dataframe to perform the computations on them.
3. Google Cloud Platform: for running our services and computations.
4. VNC viewer and VNC server: to connect to a GUI inside the Google Cloud Virtual Machine (VM) instance.
5. Flask: webserver to display a GUI consisting of playable iframes with Spotify songs, checkboxes for user input, and a graph displaying the current calculated user preference for each song feature.

4 Methods

Each song is represented as a vector of its corresponding features (11 dimensional vector). The user is represented as its own vector, starting with a zeros vector (vector with 0 entries). The goal is to find the closest vectors in the dataframe to the user vector:

1. $user\_vector = (user\_feature_i) \in [0,1]^{11}$
2. $song\_vector = (song\_feature_i) \in [0,1]^{11}$

for $user\_feature_i$ and $song\_feature_i$ in the 11 features defined in 2.

Note that we normalize values in the dataframe to the same scale: between 0 and 1.

At each iteration of the model, the user is presented a list of songs. They choose (select) the songs they like or simply ignore all if they don't like any, and press submit. Then the model will work in the backend to provide the user with new recommendations of songs based on his feedback (songs they liked or none if they didn't like any). This is done by updating the user vector after each iteration, and then filtering the closest vectors in the dataframe to the user vector.

4.1 Updating the user vector

At each iteration i, the user select or doesn't select any song. Initially, his vector is an 11 dimensional 0s vector:

$$user\_vector = [0] \text{ x } 11 \in R^{11} \tag{1}$$

If the user doesn't like any song when submitting, his vector remains fixed, and we update the list with new songs. If the user liked songs in the previous list, then their vector gets updated as the average of his previous entries and the new vector entries:

1. $user\_feature_i = user\_feature_i + \Sigma_j(song\_feature_i)_j$
2. $user_feature_i = user_feature_i/$total number of songs they liked

where j runs over all the songs the user liked.

To find the closest vectors in the dataset to the user vector, the following two methods were implemented:

4.2 Standard deviation

For this method, the standard deviation was calculated for each property (column) in the dataset. Thereafter, songs where queried with the criteria that the difference between user feature (preference) and row feature was less than 1 standard deviation away:

$$abs(user\_feature_i - song\_feature_i) < std\_feature_i \tag{2}$$

where *abs* refers to the absolute value and $std\_feature_i$ refers to the the standard deviation of the corresponding feature i (column in the dataframe).

From the songs that matched this condition, a sample of 10 random songs were selected and provided to the user.

4.3 K-Means

For this method, the data from the dataset was fitted into k=10 clusters using k-mean. Thereafter, songs were queried with the criteria that the difference between the center of each cluster and the user feature (preference) was less than 1 standard deviation away:

$$abs(cluster\_center\_feature_i - song\_feature_i) < std\_feature_i \tag{3}$$

where *abs* refers to the absolute value and $std\_feature_i$ are defined as before.

Finally, 10 random songs of those were provided to the user.

5 Results and Examples

The resulting program is a recommendation system for Spotify songs. Songs are queried from the dataset, at first randomly, thereafter based on prior user preference. The similarity of songs are based on 11 music properties, and proximity is calculated using either standard deviation method or k-means. An example condition for the query to the dataframe using Spark is shown below:

```
def recommend_by_std(self):
    # to check for feature importance: if previous feature value - new feature value >2 * std => not important

    # need all rows where difference between user feature and row feature is less than 1 std
    condition = ((self.df.danceability-self.user.danceability<self.std1) | (-self.df.danceability+self.user.danceability<self.std1)) \
            & ((self.df.energy-self.user.energy<self.std2) | (-self.df.energy+self.user.energy<self.std2)) \
            & ((self.df.key-self.user.key<self.std3) | (-self.df.key+self.user.key<self.std3)) \
            & ((self.df.loudness-self.user.loudness<self.std4) | (-self.df.loudness+self.user.loudness<self.std4)) \
            & ((self.df.mode-self.user.mode<self.std5) | (-self.df.mode+self.user.mode<self.std5)) \
            & ((self.df.speechiness-self.user.speechiness<self.std6) | (-self.df.speechiness+self.user.speechiness<self.std6)) \
            & ((self.df.acousticness-self.user.acousticness<self.std7) | (-self.df.acousticness+self.user.acousticness<self.std7)) \
            & ((self.df.instrumentalness-self.user.instrumentalness<self.std8) | (-self.df.instrumentalness+self.user.instrumentalness<self.std8)) \
            & ((self.df.liveness-self.user.liveness<self.std9) | (-self.df.liveness+self.user.liveness<self.std9)) \
            & ((self.df.valence-self.user.valence<self.std10) | (-self.df.valence+self.user.valence<self.std10)) \
            & ((self.df.tempo-self.user.tempo<self.std11) | (-self.df.tempo+self.user.tempo<self.std11))

    # self.df.where(condition).show(10)
    filtered_df = self.df.where(condition).head(10)

    links = ["https://open.spotify.com/embed/track/" + row.uri.split(":")[-1] for row in filtered_df]

    return random.sample(links, 10)
```

Figure 1: Example script for queries with standard deviation method

The user features vector are displayed in a bar chart which is updated at every iteration (every time the user presses submit), as ten new songs are recommended. This shows how the user preferences are being updated through time:

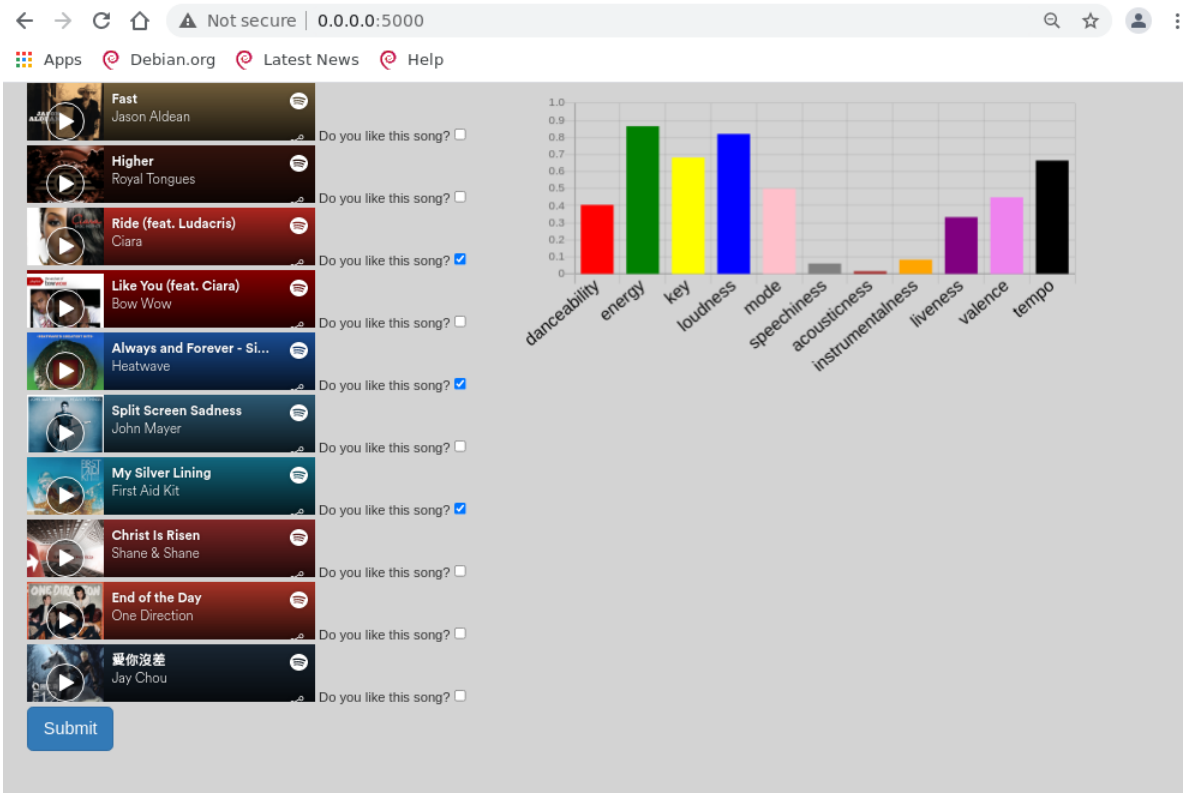

Figure 2: Basic GUI interface showing feature updates at each iteration

6 Analysis and Conclusion

We notice that kmeans is much more computationally expensive than pure standard method. The standard deviation method could initialize and run in less than 1 second, while kmeans needs at least 1 minute to initialize when the dataset is big (more than 50 thousand songs), even though inference is also fast.

The recommendation system was created with desired functionality, and queries within a big dataset was done using PySpark in adherence to the course material.