Exploring Music Listening Data

DS 2000 Spring 2020 Final Project

April 3rd, 2020

Katie Conner

conner.kat@husky.neu.edu

Serena Luo

luo.se@husky.neu.edu

Anna Repp

repp.a@husky.neu.edu

Project code, notebook, data, and report can also be found on Github

Problem Statement and Background

Spotify, the well-known music streaming service, is particularly well-loved for many reasons. From access to millions of songs, artists, and playlists to the social element of seeing the music a user's friends are listening to, there's a lot of value in the service. Perhaps the feature that most distinguishes Spotify from the competition is the customized playlists it produces, whether through Daily Mixes, Discover Weekly, or the yearly Wrapped playlist. These generated playlists recommend new songs and reflect previous user favorites.

But there's one thing the streaming service doesn't do: Spotify doesn't have an option to combine the music taste of different users into one playlist. Suppose you're hosting an event or you're in the car with someone and want to play music that everyone will enjoy. Spotify doesn't curate playlists across users, so it's up to the user to do that work. And there's a certain stress that accompanies being the designated DJ, trying to please everyone.

We wanted to pursue a solution to this problem because it is so widely felt by so many people our age. Spotify users have been looking for an answer to this question for all the years the service has been around – why can't we combine playlists, why isn't there a function to generate specific and unique playlists? The closeness to home of the problem made finding a viable solution that much more important.

The team was brimming with ideas for the approach. Weeks of bouncing ideas off each other at all stages of code development ensured a thorough and practical answer. Our solution was to build a program that would find similarities in song data among playlists, find other songs that reflected those similarities, and return all of that into a playlist accessible by both users.

Introduction and Description of the Data

Our main data source was the Spotify API. We used it to get data surrounding individual songs, albums, and artists in addition to user data, such as public playlists and saved songs. This gave us access to the playlist we would eventually focus on the most: the yearly Spotify Wrapped playlist, which is customized for every user and reflects the top 100 songs the user listened to most frequently, sorted by most to least listens within the year. We believe the Wrapped playlist is a good representation of the music a user likes and is a reliable model for our program, since it provides a large sample of songs to work with. Additionally, it standardizes the number of songs in each user playlist, making playlist-to-playlist comparisons more fair.

We interacted with the API using the Spotipy package, which provided all the functions available in the API as easily callable Python functions. It included functions to get top artists or songs played by a certain user, get audio attributes (which became very important later on), and the list of genres that Spotify uses when they make the individually curated playlists for users.

The source dataset for the results and visualizations in this report is the result of running our "get user data" code on a group of nine friends (names anonymized in submitted csv). Each row is a track appearing in a given Wrapped playlist – there can be duplicate track IDs, but there are no duplicate combinations of track ID and playlist ID. For each track, we got the track name, a list of artists, a list of corresponding artist IDs, a list of artist genres, and the six audio feature scores (danceability, energy, acousticness, instrumentalness, speechiness, and valence). Audio features are a score provided by Spotify describing on a scale of 0 to 1. Features such as acousticness and instrumentalness represent a confidence score or likelihood that the track is acoustic/instrumental, while scores like danceability and energy are calculated based on the song tempo, key, etc.

We also stored playlist length and user information in each row, even though this data was duplicated for all playlists in a track. The columns for playlist ID, playlist length, and username were later converted into attributes for the Playlist class that we created.

Methods

When approaching the methodology of the project, we worked backwards from our goal to find our starting point. We knew what we wanted to present at the end of the project – a customized playlist – so all of our functions had this end result in mind.

Processing Data

Although the Spotify API provides a function to get playlist tracks, we found that the data provided was not detailed enough. The API's get-playlist function returns playlist-level information such as playlist name, creation date, owner, etc. However, data for each track in the playlist was limited to information such as song name, album name, artist name, available markets, and popularity. To get more detailed track information, such as danceability, energy, and acousticness, we used audio-features.

To store playlist data, we created a Playlist class. It contains a username (str), playlist id (str), playlist length (int), and track data (pandas DataFrame). One of the main benefits of creating the Playlist class was that it made playlist comparison easier.

Additionally, we wrote a flexible method for the class called avg_attr, which takes an audio feature as a parameter and returns the average value of that audio feature across a whole playlist's tracks. This was very useful for shortening code for similarity score calculations.

Calculating Similarity

Creating a similarity score between two playlists took several iterations. First, we tried to count the number of songs or artists that were common to a pair of playlists. However, these counts tended to be low (and/or 0), and not very informative. As a result, we started to incorporate audio features. Our reasoning was that audio features give a better overview of each song's attributes.

For each user, we calculated an average audio feature score for each of the six features we explored (danceability, energy, acousticness, instrumentalness, speechiness, valence). We also calculated what percentage of a pair's combined songs/artists/genres were not shared (a pair with 5/10 artists in common would have 0.5 artist difference, while a pair with 1/10 artists in common would have 0.9 artist difference). Using these average audio feature and difference percentage values, we used euclidean distance to calculate similarity between the user pair:

$$dance = avg\_danceability\_1 - avg\_danceability\_2$$
$$energy = avg\_energy\_1 - avg\_energy\_2$$
(...repeated for all audio features)
$$sum\_squares = dance^2 + energy^2 + acoust^2 + instru^2 + speech^2 + val^2 + artist^2 + song^2 + genre^2$$
$$eucl\_dist = \sqrt{sum\_squares}$$

We believe Euclidean distance is an accurate way of calculating similarity between playlists, because playlists with lower differences in their average scores or lower percentages of non-shared artists will have a smaller euclidean distance.

Results, Conclusions and Future Work

Visualizations

For our nine-person group, we plotted each user's average audio feature score, for all six features evaluated (Fig 1). This shows us what the group's average tastes look like, and gives us a sense of the ideal audio attribute range for a song that could be enjoyed by every member of the group (discussed in the next section).

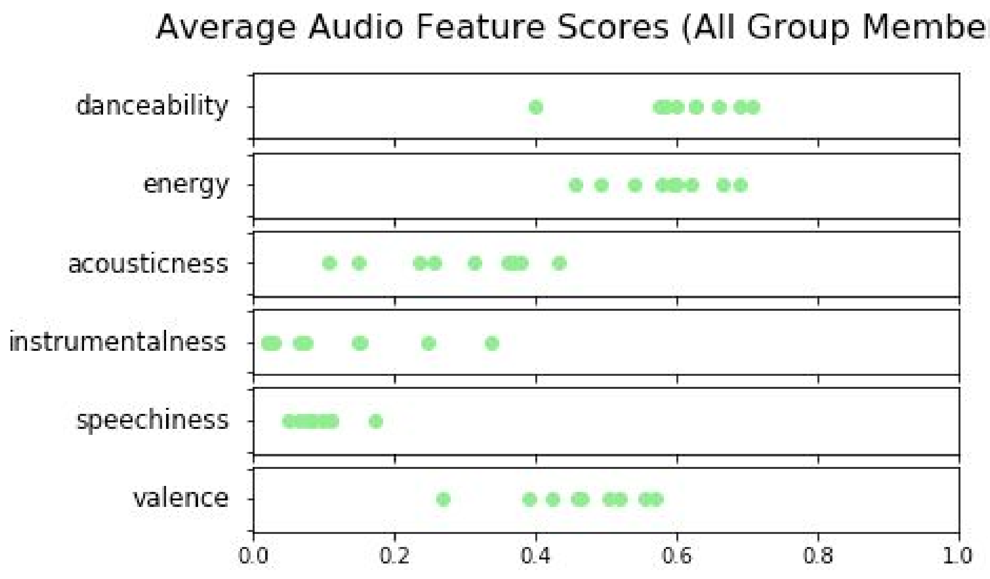
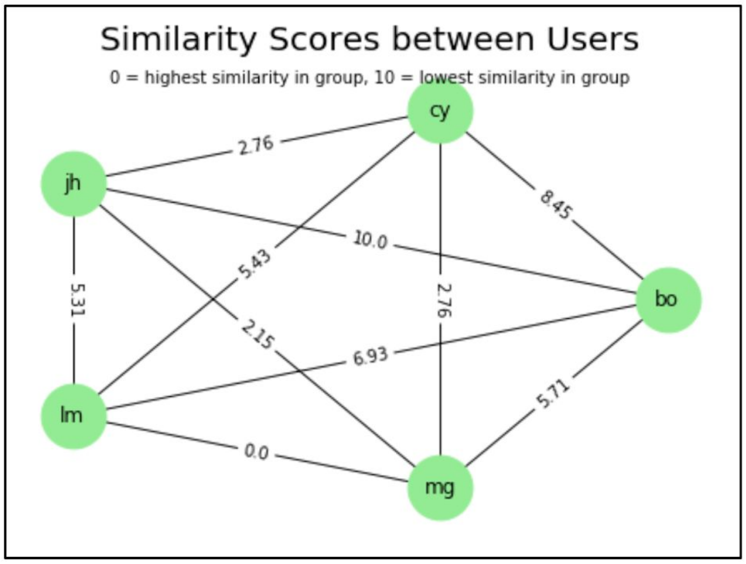
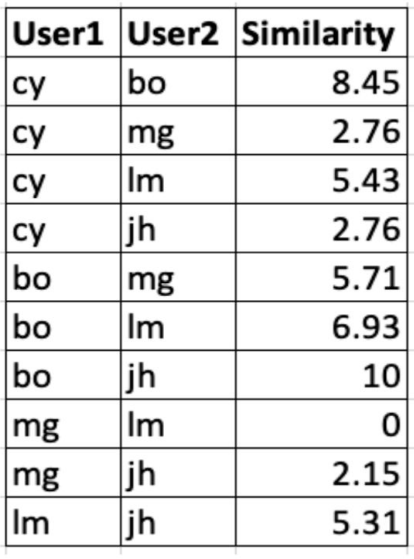


Figure 1: Average audio feature scores

Next, we created a network chart showing the similarity between every pair of users in the group (Fig 2). Because similarity scores were numerically quite close, they tended to fall in the range of 1.5-1.9, the scores were re-distributed on a scale of 0 to 10 (0 being the minimum/most similar pair in the group, and 10 being the maximum/least similar). It made the most sense to present our calculated scores in a network, because it reduces the amount of effort needed to find and compare users (as opposed to a table of scores, Fig 3).


Similarity Scores between Users
0 = highest similarity in group, 10 = lowest similarity in group
cy
jh
bo
lm
mg
2.76
8.45
10.0
5.43
2.76
5.31
2.15
6.93
5.71
0.0



User1	User2	Similarity
cy	bo	8.45
cy	mg	2.76
cy	lm	5.43
cy	jh	2.76
bo	mg	5.71
bo	lm	6.93
bo	jh	10
mg	lm	0
mg	jh	2.15
lm	jh	5.31

Figure 2: Network chart *Figure 3: Underlying data table*

We also generated a wordcloud of the most frequently appearing genres across the group of playlists (Fig 4). Although this is not a standard graph (we could have created a bar chart of the top genres plotted against the number of playlists each genre appeared in), we thought that this would be a more interesting way of visualizing the result.

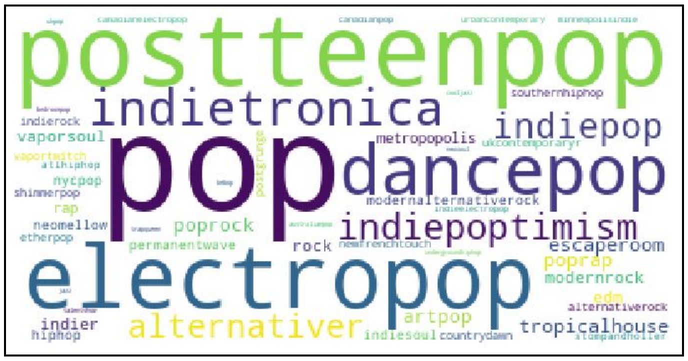


Figure 4: Music genre word cloud

Generating Playlist Recommendations

The final part of our analysis involved generating a playlist recommendation for the group. To do this, we first determined what genres were listened to by more than one user in the group. Then we took all songs from every group playlist, and filtered out songs that did not fall in the list of shared genres. We also filtered on audio features by selecting only songs that had a danceability value between the minimum and maximum group average scores. Although we could have filtered on more audio features, this led to short or nonexistent playlists when generating playlists for small groups. Danceability was chosen as the audio feature to filter on because we assume that people getting together in a group are choosing to socialize, and we wanted to set the mood accordingly.

Conclusions

In conclusion, by focusing on the use of classes and euclidean distance throughout the code development, we were able to streamline a big idea into a workable, real-life product. We calculated the similarity across tracks, analyzed their audio features, created an output and wrote it into a csv, and finally, translated that output to a playlist that appears directly within Spotify. Our final product solved the problem we initially identified in a comprehensive and widely usable way, and our data visualization tells the story in a way that supports and reflects the data gathered and used throughout the project.

Future Work

Given more time, we could modify many aspects of our project. Three key areas of improvement are outlined below:

Data: We knew this coming into the project, but one of our biggest data collection roadblocks was getting accurate genre information for each track. Spotify only provides genre information on the artist or album level, not for individual songs. We tried using the last.fm API to get song tag information, but were ultimately unsuccessful. As a workaround using Spotify data, we made the assumption that all of an artist's genres could apply to a song; however, it would be good to improve this process in the future.

Additionally, the data collection process is somewhat slow due to the fact that we are calling the API for every track. If possible, it would be useful to send track information requests in batches to a) speed up the data collection process and b) prevent rate limiting from Spotify.

Playlist generator: To improve the playlist generator, we could bias the track selection towards genres that appeared more frequently in the group. For example, if every person in a five-member group listened to pop but only two listened to rock, we could select more pop songs in the output playlist (5 pop songs to 2 rock songs).

Another improvement would be incorporating new music into the playlist. Because the group playlist is pulled from everyone's concatenated tracks, that means that every song in the generated playlist has been listened to by at least one person in the group. If we had a method of pulling songs from Spotify by genre, we could add new music (roughly aligned to listener tastes) into the playlist.

Single user song recommender: One of our initial goals for the project was to create a separate playlist recommender for single users. It would take a user's playlist and their friends' playlists as inputs, and create a list of song recommendations for the main user by drawing songs from the playlists of friends with closer similarity in music taste. We did not complete this goal due to lack of time, but it would be an interesting next step for the project.

References

"Euclidean Distance." *Wikipedia*. Wikimedia Foundation, 18 Mar. 2020. Web. 3 Apr. 2020. https://en.wikipedia.org/wiki/Euclidean_distance

Provided a guide for how to do euclidean distance calculation over more than 2 planes.

NetworkX developers. "NetworkX." *NetworkX*, 2019, networkx.github.io/.

Used as a reference to draw a network graph from a list of nodes. We also learned how to add edge labels using draw_networkx_edge_labels.

Plamere. "Plamere/Spotipy." *GitHub*, GitHub, 2 Apr. 2020, github.com/plamere/spotipy.

Used as a reference to write the authorization and data pulling process in python, instead of using requests.

SumedhKadam. "Generating Word Cloud in Python." *GeeksforGeeks*, 11 May 2018, www.geeksforgeeks.org/generating-word-cloud-python/.

Used as a reference to generate a word cloud from strings (third visualization output).

"User Guide." *User Guide - Pandas 1.0.3 Documentation*, 2014, pandas.pydata.org/pandas-docs/stable/user_guide/index.html#user-guide.

We frequently referenced Pandas documentation to join playlists together, convert lists to dataframe (and vice versa) depending on function needs, and apply functions to DataFrame columns (instead of iterating through a list of lists and changing/appending values one by one). We also got syntax for altering DataFrame indices.

"Web API." *Spotify for Developers*, developer.spotify.com/documentation/web-api/.

We relied heavily on the API documentation to figure out which functions to use and what our data meant. In addition, the web API generated key credentials allowing access to spotify data and user information. The website explained the user authentication process and the necessary permissions from a user required to gather information.