Roger Barber & Alex Brand

SI 206 – Data Oriented Programming

Dr. Ericson

11/29/22

Spotify & Genius API – Relationships Between Word Usage & Song Metrics

**Goals**

1. To see if there is a relationship between word usage and popularity in music.
2. To see if popular songs have higher or lower of certain Spotify-specific song metrics.
3. Otherwise, to see if there are any correlating patterns in the data we have collected.

**Problems / Limitations**

Some major issues we ran into with the Spotify API were the nature of the Billboard Hot 100 that we decided to work with. For one, the data set changes every week, so collecting 100 items proved to be difficult, as the set we were trying to examine changed regularly and revealed new issues. If I were to go back and change our project, it would have been collecting data every week on the Top 100 and storing that into a database – but that would get a bit large pretty quickly.

Secondly, we noticed that there is a lot of seasonal and timing aspects that change the composition of the Hot 100. Taylor Swift and Drake both released new albums before our project, so a large proportion of the top 100 songs were from their albums. This significantly skews the genre data we collected and most likely has a great effect on the music feature averages as well. And since this is the holiday season, there are a few Christmas songs that have crept back onto the charts, which is not representative of what the chart contains for the rest of the year. I think the goal of our project is a great idea, but to properly do this work would require a long period of data collection and careful examination of external happenings such as album releases, seasonal music, and viral trends.

With the Genius Lyrics API, the root Genius API does not allow its users to pull lyrics directly from its database claiming that the lyrics are proprietary information. Fortunately, there is a lyricsgenius package that uses the Genius Lyrics API to search the song based on song name and artist and then auto screen scrapes the lyrics. This allowed us to search songs in the top 100 and then return a full lyric string.

From here, the idea was to count all the words in each song across the whole dataset and then generate a top ten words list. The intention was to calculate what percentage of the top 100 songs contained at least one of the top 10 words. Unfortunately, the top ten list was full of common words like “I, me, we, you, etc.” and didn’t really describe anything interesting about the data. Additionally, many of the songs in the Top100 didn’t contain any of the top 10 words so it was difficult to make any meaningful calculations. On top of this, we also tried to construct our code in a way that the Spotify API code would pull all of the data and then the Genius code would do word counts for 25 songs at a time. This created issues because the top ten words were different as each group of 25 songs got added to the database. These issues required us to pivot how we were collecting and processing the data.

Our initial change was to construct our code in a way that would allow us to run our programs simultaneously for each 25 songs. The Spotify API would run and collect 25 songs and then the Genius API would use the artist and song information from the newly constructed database to conduct a search for the lyrics of each song, process the lyrics by cleaning and counting, and then ultimately processing a list of the 10 ten words for each song. The Genius program then constructed a word index for the top ten words across all 100 songs resulting in a table of over 340 items, and then a word to song relation table which was a many to many relationship with the song rank, workd id, and count for each word as seen in that particular song. Coding these tables posed many hurdles because keeping track of the word index and the song rank required some dictionary magic we didn’t know we could do.

The last minor issue was that in some cases, the Lyrics Genius Api would not be able to find the lyrics for a song in the top 100. This mostly happened for Christmas music; the API call would time out on these specific songs in most cases. This required an except: skip clause in the lyrics search function. My hypothesis was that these songs may be getting hit with a higher volume of lyric searching due to the holidays.

**Data Calculations & Visualizations**

For the Spotify API:

I calculated the count of genres (with pop, rock, country, rap, hip hop, r&b, dance, and indie as ‘conglomerated’ categories) within the data set first. The distribution is skewed towards certain genres, as shown below, but it is important to keep in mind that album releases from popular artists affect this list. Also, seasonality affects this – think Christmas music.

< Placeholder for Genre Graph>

Next, I calculated the number of tracks that belong to a certain artist on the Hot 100. If the artist only had a single track, they were counted in the “Single Track” category, while anyone with greater than 2 tracks had a separate area on our pie chart. Again, this seems to be heavily affected by album releases and the season in which you are collecting data.

<Placeholder for Artist Pie Chart>

Finally, I calculated the average of four specific metrics that Spotify uses to categorize their tracks, most likely for their search algorithms: danceability, energy, speechiness, and valence. Danceability is an aggregate score of several categories including tempo and beats per minute and it describes how easy it is to dance to a song on a scale of 0.0 to 1.0, with anything above 0.6 being easy to dance to. Energy is another combined score that generally describes the level of noise, the perceived speed, and the feeling of ‘energy’ from a track, scored similarly as danceability. Speechiness is the number of words or lyrics present in the song, scored similarly to the last two; anything over a 0.6 is wordy, or is something like spoken word poetry. Finally, we have valence; simply put, it’s how positive a song is. Songs high in valence tend to have themes and lyrics and melodies that invoke good times, happy memories, or positive experiences, while songs low in valence sound brooding, dark, and melancholic.

< Placeholder for average scores chart>

For the Genius API:

Conclusion:

The Billboard Hot 100 tends to favor songs that favor song metrics in this order: danceability, energy, valence, and speechiness. Speechiness tends to run significantly lower than the other three metrics, so songs in the Top 100 seem to have minimal lyricism. Danceability and energy are similar, while danceability is higher, so the most popular songs tend to be strongly rhythmic and possess strong, loud, and driving beats.

**Instructions for Running Code**

First – run the Spotify API to collect data for usage by the Genius API and to create visualizations and tables for the Spotify API.

For the Spotify / Spotipy API:

1. Open up both spotifyAPI\_1.py and spotifyAPI\_2.py.
2. Both of these files are structured to have a main function call at the end. There is no need to enter in any external variables, and there are no user inputs required beyond running the code.
3. Run spotifyAPI\_1.py first:
   * The Spotipy module creates an Oauth2 instance using my client secret and client ID that accesses the Spotify API using an object named ‘sp’ – all the call methods are ran on that object using methods native to the module.
   * If you want to check if API pull works, be sure the database file is empty. We will do so before submitting.
   * Run the program until the third table indicates it is full. The longest table pull will be 100 items which will take 4 runs of 25 maximum items. Approximately 8 runs should be the maximum you need to populate all three tables.
   * The program will indicate whether or not the table has been created from scratch.
   * It will also indicate whether the table has existing items, and it will also indicate that the function has ran out of items to parse from the API data.
   * Finally, it has a failsafe method that skips operation if you are attempting to add data to a table and there is no new data to add.
4. Run spotifyAPI\_2.py second:
   * Again, there should be no text files besides the reports and instructions after we submit, but double checking for old files ensures the data being used in the table matches the JSON files stored.
   * This program only needs to be ran once.
   * The program will populate three visualizations while running, for three different measurements: an average of the four musical scores we were testing for, a count of songs by genre, and a count of songs by artist.
   * You will need to exit out of the graph visualizations for the program to continue running, and for it to end.

**Function Documentation**

Roger // Spotify API

1. spotifyAPI\_1:
   * artistIndex(track\_list)
     + This function creates an artist index file for use in a table.
     + Inputs: a “track\_list” item that is a dictionary object created from the Spotipy module by using a predetermined track ID for the Billboard Hot 100.
     + Outputs: a dictionary that contains an integer as a key and an artist name as a value.
   * genreIndex(track\_list, sp)
     + This function creates a genre index file for use in a table.
     + Inputs: a “track\_list” item that is a dictionary object created from the Spotipy module by using a predetermined track ID for the Billboard Hot 100. Also, a Spotipy object (“sp”), an instance of the Spotipy module.
     + Outputs: a dictionary that contains an integer as a key and a genre name as a value.
   * spotipyScouring(track\_list, artist\_index, genre\_index, sp)
     + This function creates a multi-level dictionary that contains the top 100 songs as keys, with their artist index, genre index, and song features as keys in a nested dictionary.
     + Inputs: The same above track\_list dictionary, the previously generated artist\_index dictionary, the previously generated genre\_index dictionary, and a Spotipy module object (‘sp’).
     + Outputs: the same as described in the first statement.
   * createSongTable25(song\_dict)
     + Creates a Top 100 song table.
     + Inputs: a dictionary, the output of spotipyScouring.
     + Outputs: creates a table or adds to a table, 25 items at a time.
   * createGenreTable25(genre\_dict)
     + Creates a Genre Index table.
     + Inputs: a dictionary, the output of genreIndex.
     + Outputs: creates a table or adds to a table, 25 items at a time. Returns True if no items are left to be added.
   * createArtistTable25(artist\_dict)
     + Creates an Artist Index table.
     + Inputs: a dictionary, the output of artistIndex.
     + Outputs: creates a table or adds to a table, 25 items at a time. Returns True if no items are left to be added.
   * tableWriter25(artist\_index, genre\_index, track\_features)
     + Consolidates three table creation functions to run in sequence and only 25 items per run overall.
     + Inputs: 3 dictionaries; one for artist index, one for genre index, and one for track features.
     + Outputs: Three tables, one for each dictionary. Nothing returned to program space.
   * main()
     + Runs all of the above functions sequentially to add items to all three tables at the same time, 25 items per run, per table.
     + Inputs: none.
     + Outputs: none.
2. spotifyAPI\_2:
   * scoreAverage(output)
     + Writes a text file in JSON format that averages the four scores amongst all 100 songs.
     + Input: the filename you want for the stored data, including file extension.
     + Output: a text file with the average scores for all songs.
   * genreCount(output)
     + Writes a text file in JSON format that counts the number of songs in each genre amongst all 100 songs.
     + Input: the filename you want for the stored data, including file extension.
     + Output: a text file with the counts for all genres present in the data.
   * artistCount(output)
     + Writes a text file in JSON format that counts the number of songs for each artist amongst all 100 songs.
     + Input: the filename you want for the stored data, including file extension.
     + Output: a text file with the counts for all artists present in the data.
   * bar\_graph(filename)
     + Creates and displays a bar graph of the data for scoreAverage.
     + Inputs: a filename for a text file in JSON format to pull data from to generate a graph.
     + Outputs: displays the graph within the program and saves a picture in your file directory.
   * barh\_chart(filename)
     + Creates and displays a horizontal bar graph for genreCount.
     + Inputs: a filename for a text file in JSON format to pull data from to generate a graph.
     + Outputs: displays the graph within the program and saves a picture in your file directory.
   * pie\_chart(filename)
     + Creates and displays a pie chart for artistCount.
     + Inputs: a filename for a text file in JSON format to pull data from to generate a graph.
     + Outputs: displays the graph within the program and saves a picture in your file directory.

**Works Cited / Outside Resources Used**

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Issue | Resource Used | Did it Work / Results |
| 11/20/22 | Learning to use Spotipy | https://towardsdatascience.com/extracting-song-data-from-the-spotify-api-using-python-b1e79388d50 | Yes – I successfully learned how to use the API and gave me examples on how to use it to pull information out. |
| 11/20/22 | SQL Help | https://www.w3schools.com/sql/ | Helped a little, referred to our notes for SQLite specific syntax. |
| 11/20/22 | Spotify Documentation / API Structure | https://developer.spotify.com/documentation/web-api/reference/#/ | Yes – helped me understand the structure of my API pulls so I could extract data. |
| 11/20/22 | Spotipy Documentation | https://spotipy.readthedocs.io/en/2.21.0/ | Yes – introduced me to the methods in the module. |
|  | Genius API |  | No – Required additional support from the Lyrics Genius API module |
|  | Lyrics Genius API |  | Yes – allowed us to conduct searches for each song in the top 100. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |