Roger Barber & Alex Brand

SI 206 – Data Oriented Programming

Dr. Ericson

11/29/22

REPO LINK - https://github.com/rogerbarber/final\_project206

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.

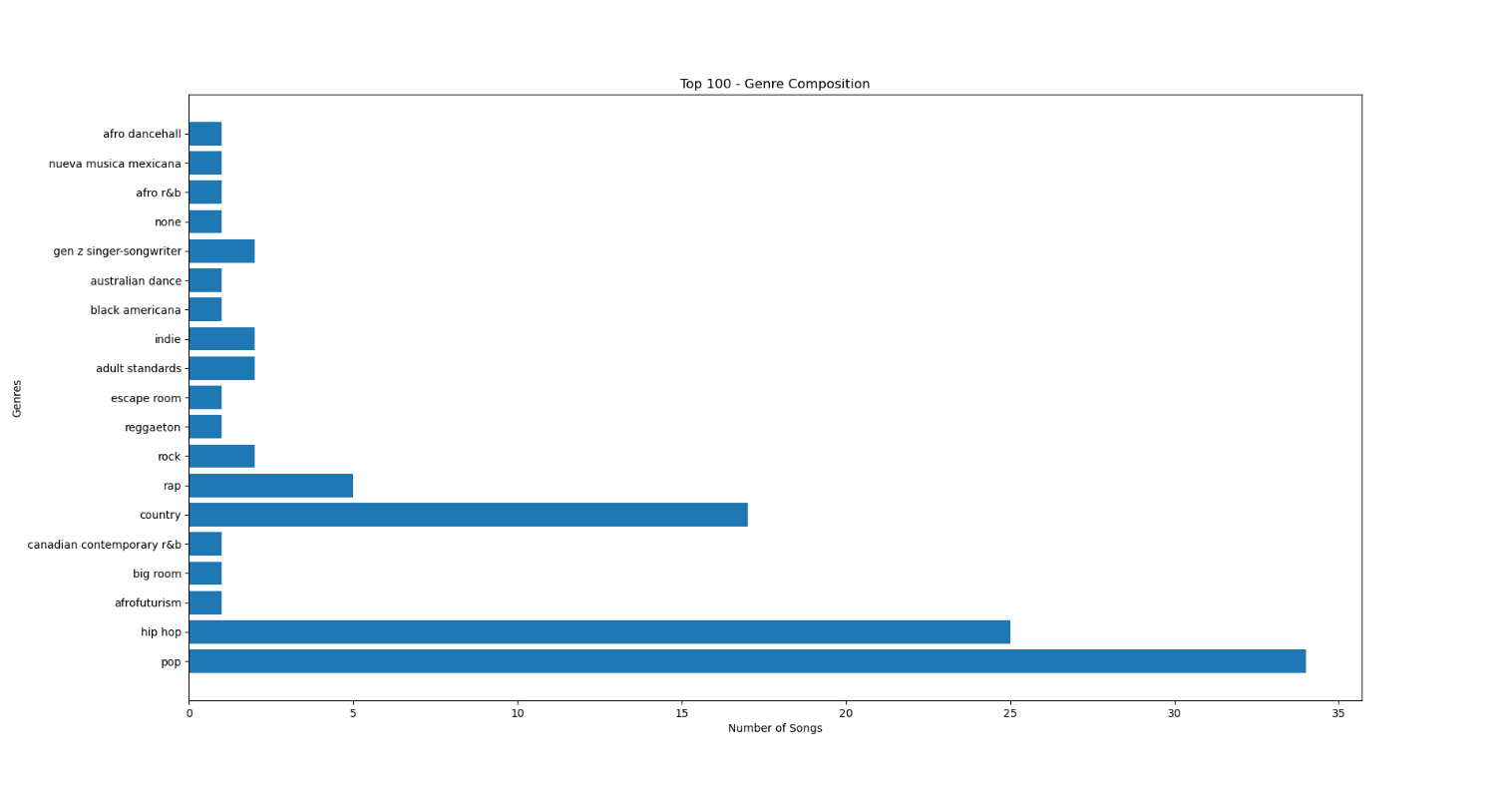
Our initial change was to construct our code in a way that would allow us to run our programs simultaneously for each 25 songs. The Spotify API would run and collect 25 songs and then the Genius API would use the artist and song information from the newly constructed database to conduct a search for 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.

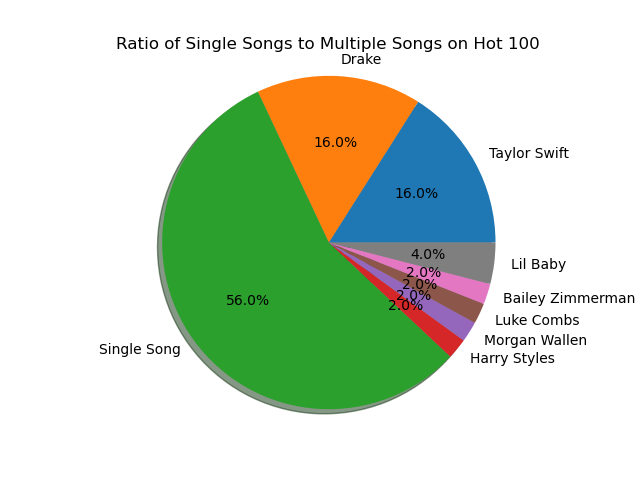
**Data Calculations & Visualizations**

For the Spotify API:

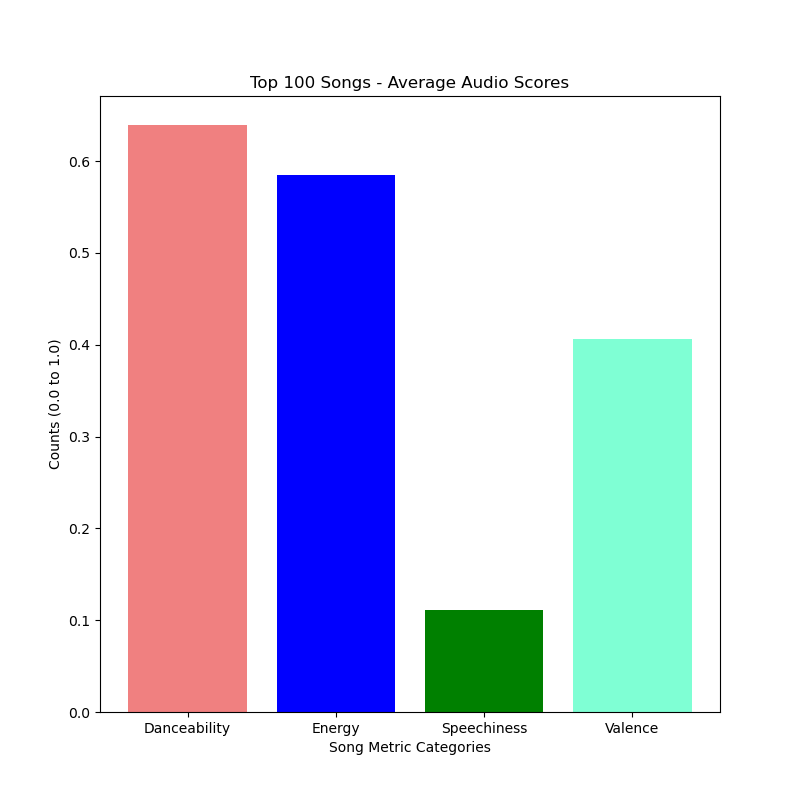
I calculated the count of genres (with pop, rock, country, rap, hip hop, r&b, dance, and indie as ‘conglomerated’ categories) within the data set first. The distribution is skewed towards certain genres, as shown below, but it is important to keep in mind that album releases from popular artists affect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



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.



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.



For the Genius API:

It was difficult to come to a conclusion for what data to plot for the Genius API, We knew what we wanted to find but the process of getting there had many hurdles. We ultimately decided to calculate the top ten words across the entire dataset. Unfortunately this information didn’t tell us much about the data we collected. But there’s a pretty chart to explain how this information is expressed.

Chart, bar chart, histogram

Description automatically generated

As shown, my was the most popular word expressed in all of the songs within a top 10 word subset of each song and was used 495 times within 100 songs.

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.

Trying to pin lyricism trends within the most used words across the top 100 was not a fruitful endeavor. We ended up with a lot of lyrical data that really didn’t explain much. The only real conclusions we can come to are that songs in the Top100 are often written from a first-person perspective, are usually about relationships, and that the most used words don’t describe any unique trends in lyricism.   
  
The most important data point that can help us explain this lack of data is the mean speechiness score which, across the entire Top100 was a .111. According to Spotify’s explanation of their audio feature metrics, songs usually range between a .3 and a .6 on the speechiness scale. Anything below a .3 is considered dance music, anything above a .6 is probably spoken word, a podcast, or an audio book.

**Instructions for Running Code**

First, you need to install our modules. The installation lines in a command prompt are:

1. pip install lyricsgenius
2. pip install spotipy

This will enable you to use the lyricsgenius and Spotipy modules. This was confirmed to be suitable by Holden in place of using the requests module, as they still require the generation of keys and processing of data.

The next steps are as follows:

1. Open main\_file1 and main\_file2.
2. Run main\_file1 to begin the API pulls and create the database.
   1. You will need to run the API at least 4 times to complete the database.
   2. You may run it more if you want; it will tell you that the data has ended and there is nothing more to add.
3. Run main\_file2 to begin writing files to the directory, performing calculations, and displaying and saving graphs and the associated images.
   1. This file only needs to be ran once.
   2. You will need to exit out of each figure as it displays to continue viewing all the figures in the code space.
   3. You’re all done!
4. The JSON files and images are saved in the same directory as the main files.

**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.
3. Main\_file1
   * Main3()
     + Runs imported main statements from the Genius API and Spotify API files.
     + Inputs: None.
     + Outputs: Database, tables.
4. Main\_file2
   * Main4()
     + Runs imported main statements from Genius API and Spotify API files to create JSON files and visualizations.
     + Inputs: None.
     + Outputs: Graphs and written files.

Alex // Genius Lyrics – lyricsgenius API

1. Genius\_lyrics\_pull.py
   * Pull\_songdata
     + Works in tandem with Roger’s first database function by pulling each set of 25 songs that were populated into the database and then constructing a dictionary containing the artist’s name and each song associated with that artist in the Top100. That dictionary is returned to be used in the lyric\_search function
   * Lyric\_search
     + Iterates through the song\_dict dictionary that was returned by the pull\_songdata function
     + Inputs a song and artist into the genius song search function to make api requests to the Genius api
     + Uses the .lyrics method to return the lyric information from the api song search and saves those lyrics to a dictionary containing the artist name, each song associated with that artist, and the lyrics for that song as a string
   * Clean\_and\_count\_lyrics
     + Establishes an ignore list of common words we did not want to count in our dataset
     + Establishes regex patterns to remove unwanted string data from the dataset
     + Iterations through the data packet (lyrics\_dict) returned by the lyric search to establish keys in the word\_counts dictionary associated with the songname. Then processes each lyric string by removing unwanted data using the .replace method and regex patterns.
     + Iterates through a cleaned list of lyrics for each song to produce dictionary information for word counts for each word in the list.
     + Returns a completed word count dictionary for each song.
   * Find\_top\_ten
     + Iterates through the word count dictionary to restructure the dictionary to be useable in the store top words function.
     + Uses lambda sorting and list slicing to compile lists of the top 10 words for each song.
     + Returns a dictionary with song names as key values with a list of the top 10 words for each song.
   * Store\_top\_words
     + Created the wordIndex table to track all of the words returned in the find\_top\_ten function
     + Creates a check object equal to the max id in wordIndex
     + Establishes the words from the first 25 songs in the database by checking if check[0] (the max id} is equal to None. If so, iterates through top\_ten\_words, stores words in an accumulation dictionary and iterates the count by one. Saves the index and word to the database for each word.
     + Establishes the words for songs 26+ by creating a comprehensive check dictionary and iterating through a copy of that dictionary to add unique words storing the count and unique word at the specified position in the database.
     + Returns top\_ten\_words as a new top\_ten variable to be used in the next database function
   * Make\_songwordrelation\_table
     + Creates the songWordRelation table which is a many to many relation between the songs and the word counts where each song\_song rank is tied to the Top100 table, each word\_id is tied to the word index table, and the counts are generated from the top\_ten dataset.
     + Matches word\_ids with their word position in the wordlist value in the the nested top\_ten dictionary and the song\_rank on the trackname in the Top100 dictionary
     + Stores the song\_rank, word\_id, and count in the database in their specified index position
   * Main
     + Establishes the genius API session
     + Establishes the connection and cursor for the database
     + Creates a song\_dict object by running pull\_songdata on the cursor and connection
     + Creates a lyrics object by running lyrics\_search on the genius session and the song\_dict object
     + Creates a word\_counts object by running clean\_and\_count\_lyrics on the lyrics object
     + Creates a top\_ten\_all\_songs dictionary by running find\_top\_ten on the word\_counts object
     + Stores the top\_ten words across the entire dataset in the word index table by running store\_top\_words on the Cursor, Connection, and the top\_ten\_all\_songs object
     + Establishes a many to many relational database for the entire word collection by running make\_songwordrelation\_table on the cursor, connection, and top\_ten object
2. Genius\_data\_collection.py
   * Get\_top\_ten\_of\_each
     + Joins and iterates over Top100 and songWordRelation to sort the data into a dictionary of the artist: song: [(word, count), …] to reconstruct a list of the top ten words for each song.
     + Returns the song\_word\_dict to be used in the get\_top\_ten\_of\_all function
   * Get\_top\_ten\_of\_all
     + Iterates over song\_word\_dict to count all the words in dataset
     + Iterates over the newly created word dict to sort the words in descending order
     + Creates a list of tuples with (word, count) as values for the top 10 words by slicing.
     + Returns the top ten word tuple list
   * Write\_to\_json
     + Writes the top ten word tuple list to a json format
   * Plot\_data
     + Iterates over the top ten word json file to plot the relevant data

**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. |
| 11/20/22 | Genius API | https://docs.genius.com/#/getting-started-h1 | No – Required additional support from the Lyrics Genius API module |
| 11/20/22 | Lyrics Genius API | https://lyricsgenius.readthedocs.io/en/master/index.html | Yes – allowed us to conduct searches for each song in the top 100. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |