Bella Kressaty & Owen Sims
The Sum of Music
SI 206 - Final Project Report

Work in Progress Repo - https://github.com/kressb/bellaowen_si206_finproj
Final Files Repo - https://github.com/kressb/bellaowen_si206_finproj/tree/main/FINALFILES

In addition to your API activity results, you will be creating a report for your overall project. The report must include:

1. The goals for your project including what APIs/websites you planned to work with and what data you planned to gather (10 points)

Our initial goal was to look at Spotify and Apple Music's APIs and compare and contrast their stats, ie. which platform had more plays, what songs and artists were more popular on which platforms, if certain genres of music were more popular on one platform or the other, etc. We had to rethink the scope of the project when we were unable to access Apple Music's API due to their stingy developer requirements. It also came to light that Spotify does not provide the depth of information about their streaming demographics and genre breakdowns as we had hoped, so we instead had to focus more on top artists and songs.

This meant that our revised goal was to look at various music related APIs and websites that help us better understand the *now* in music. This meant looking at Spotify, SeatGeek, Ticketmaster, and Billboard (what we actually used of these changed a bit over the course of our project) in the hopes of understanding which artists were the most popular right now. Given the data available we wanted to look at a few different statistics such as how their popularity ranks on different platforms compared to one another, artist popularity vs quantity of their songs on the top of the charts, and artist popularity vs that artists average song length, as a few examples.

2. The goals that were achieved including what APIs/websites you actually worked with and what data you did gather (10 points)

Spotify Top 200 Artists Weekly, US, Website (using Beautiful Soup):

Downloaded a static version of the site to avoid illegal scraping, used beautiful soup to extract a list of 200 tuples with the artist name and their spotify URI.

Artist profiles via Spotify API:

For each of the artists in the top 200 weekly, we called their specific API for their top 10 most popular songs. From this we extracted the average popularity of their top 10 songs to get a singular popularity score for each artist, as well as the average length in seconds of each of these artists' 10 most popular songs.

SeatGeek API:

The SeatGeek API allows us to, when putting in an artist as a parameter, find out the location and date of their next five shows, as well as the price ranges for each concert. We wanted to use this information, specifically the pricing element and max prices, to see if spotify popularity correlates with higher ticket costs. SeatGeek also has its own popularity metric which we wanted to compare to Spotify's.

Billboard Charts via Spotify API:

Using the spotify API again, we were able to pull the Billboard Hot 100 playlist, which gave us the 100 most popular songs right now and the artists that sing them. This allows us to correlate which popular songs are sung by which popular artists.

3. The problems that you faced (10 points)

The Spotify API & Website:

The Spotify API (via Spotipy) has very specific requirements, to get any information about an artist you need their Spotify URI (Uniform Resource Indicator) which is an individualized 22 digit, alphanumeric string. To get this string for each artist, we had to download the static html page for the spotify weekly top 200 US artists chart for the week of april 7-13 (spotify explicitly says no scraping on their website, so we did the same thing you did for project 2 and pre downloaded the html and scraped that). We then had to pull out the URIs for each artist and use them as parameters in a function that makes the API call, which takes forever to run because the amount of data we are collecting is essentially 200 separate APIs.

SeatGeek API:

The seatgeek api follows a very basic structure with artist information and venue Information. It was probably the easiest data structure to pull information from, as you can specifically search for “performers” to filter out music artists, and every artist that you search by keyword is listed under “name”. We wanted to pull their metric of popularity score, which you can easily pull by referencing “score”. It is consistent. The only problem we ran into is that while seatgeek houses spotify URI data for artists, they do not let you search by this term, so we could not confirm that every dictionary of artist information we were pulling was the exact artist. For example, “Drake” is labeled as “Drake in Las Vegas!” and has a popularity score of 0 in seatgeek’s database.

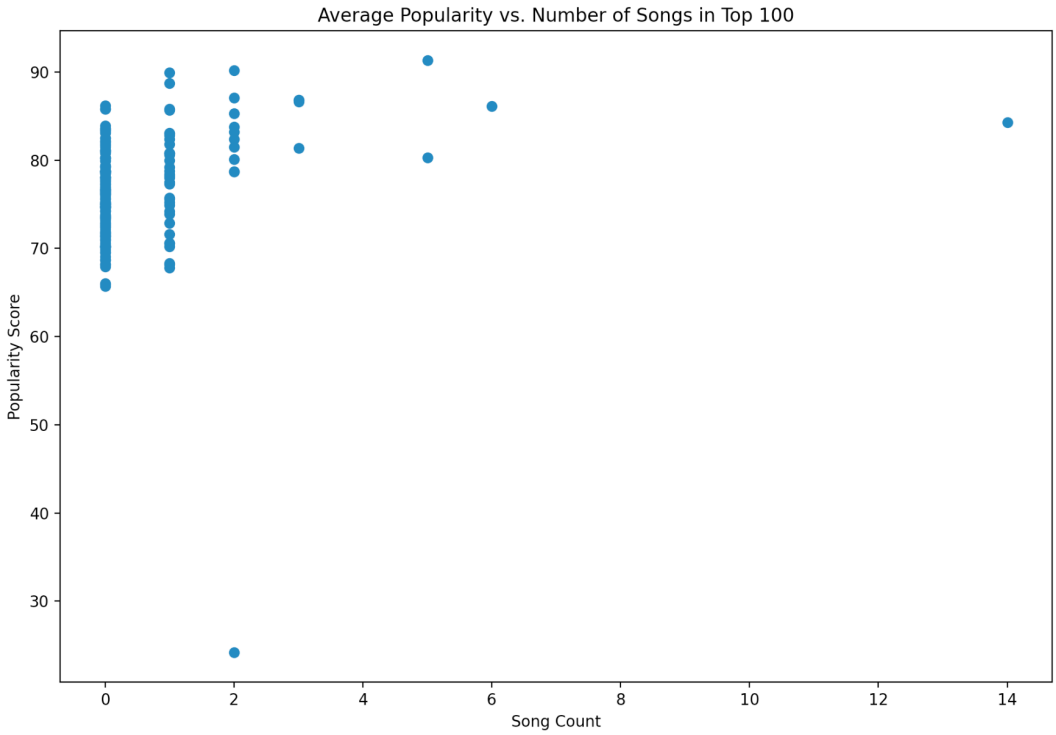
Adding to database 25 at a a time:

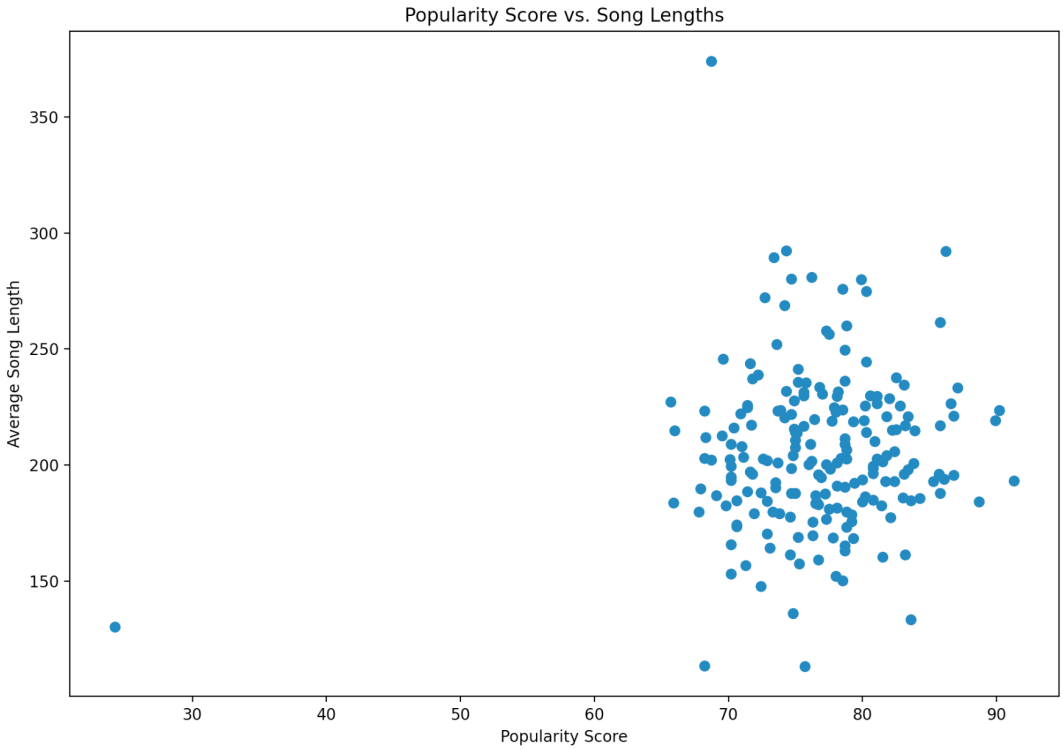
We frankly had no idea how to do this part, and trust us we spent more time on this one element than the whole assignment was probably supposed to take. We were very easily able to create a database in just a few lines that contained all 200 of our top artists with their relevant info, but the process of artificially breaking up this very easy function into multiple more convoluted ones escaped us. We did have a few questions about the practicality of this part of the assignment, why is this even relevant? Isn’t it more efficient to run the program one time? Also when did we/were we supposed to learn this?

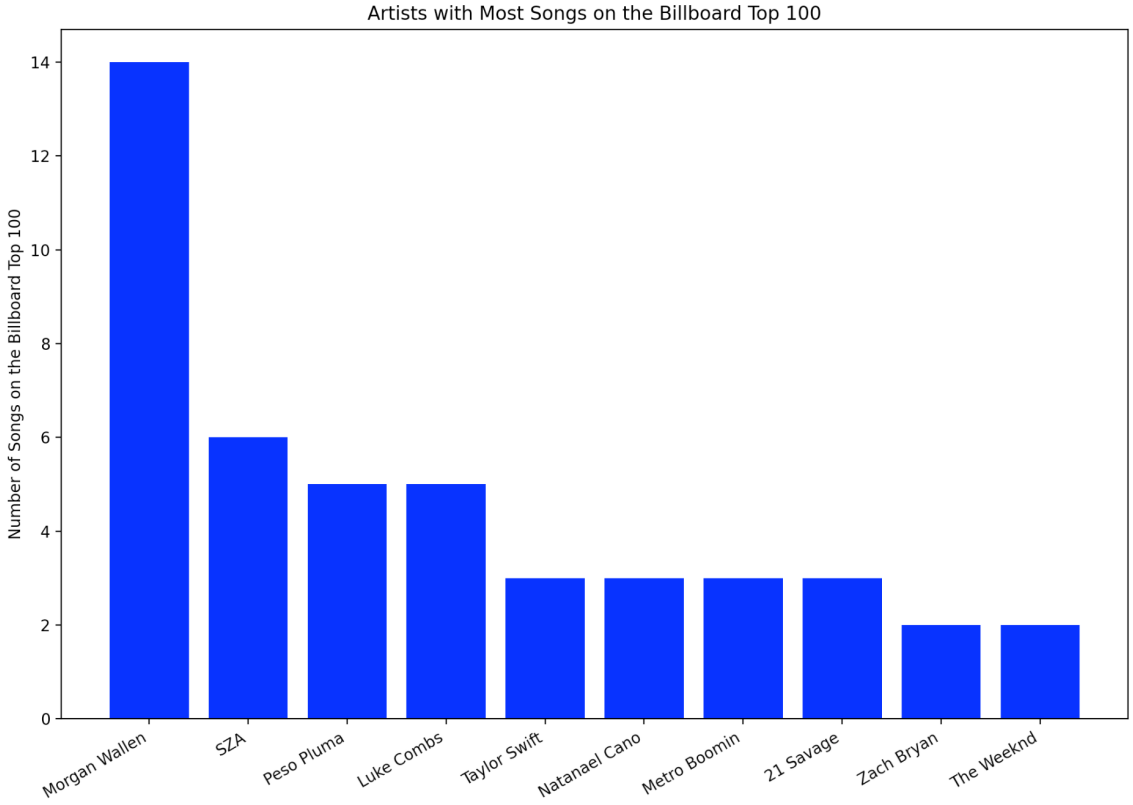
4. The calculations from the data in the database (i.e. a screen shot) (10 points)

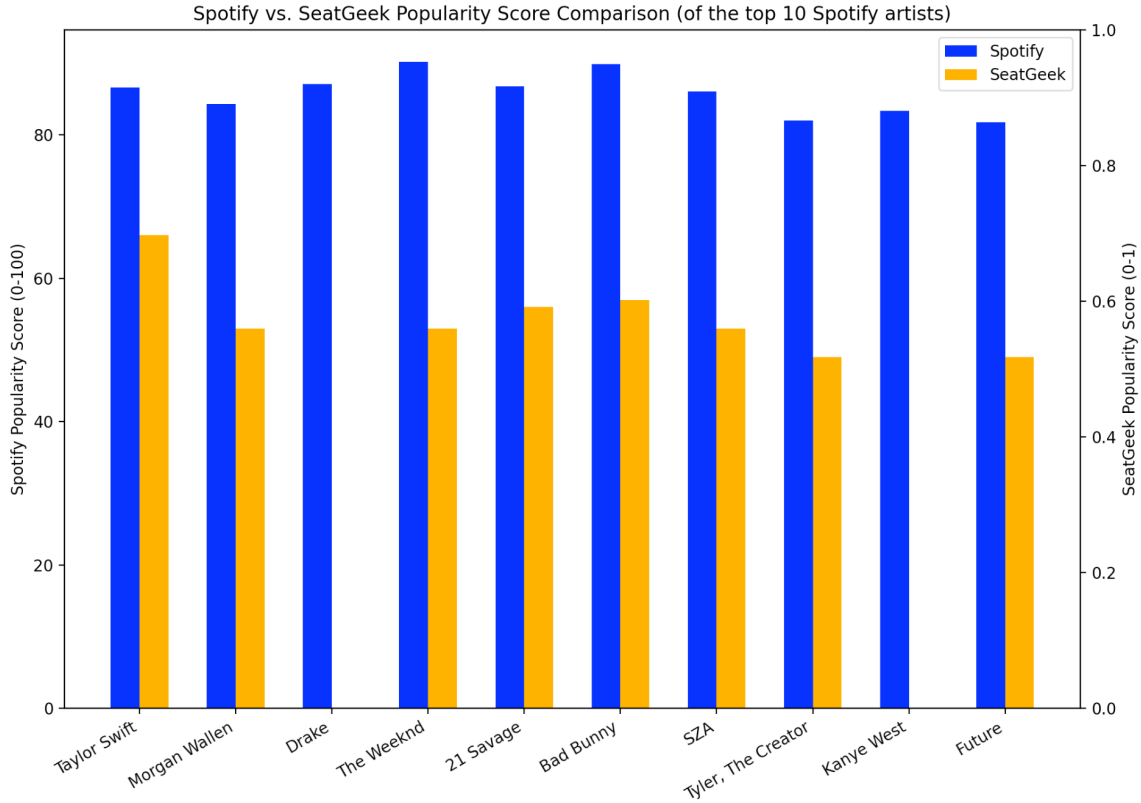
```
# Create a new table to hold artist counts
conn.execute('''
    CREATE TABLE artist_counts (
        artist_name TEXT,
        count INT
    );
''')

c.execute("SELECT artist_name FROM songs")
artists_list = [artist for row in c.fetchall() for artist in row[0].split(', ')]

# Create a dictionary to hold the artist names and their counts
artist_counts = {}

# Count the number of appearances for each artist
for artist in artists_list:
    if artist in artist_counts:
        artist_counts[artist] += 1
    else:
        artist_counts[artist] = 1

# Insert the artist names and counts into the artist_counts table
for artist, count in artist_counts.items():
    c.execute("INSERT INTO artist_counts (artist_name, count) VALUES (?, ?)", (artist, count))

# Commit the changes and close the connection
conn.commit()
conn.close()
```

5. The visualization that you created (i.e. screen shot or image file) (10 points)






Artists with Most Songs on the Billboard Top 100
Number of Songs on the Billboard Top 100
0
2
4
6
8
10
12
14
Morgan Wallen
SZA
Peso Pluma
Luke Combs
Taylor Swift
Natanael Cano
Metro Boomin
21 Savage
Zach Bryan
The Weeknd


Spotify vs. SeatGeek Popularity Score Comparison (of the top 10 Spotify artists)
Spotify
SeatGeek
Spotify Popularity Score (0-100)
SeatGeek Popularity Score (0-1)
0
20
40
60
80
0.0
0.2
0.4
0.6
0.8
1.0
Taylor Swift
Morgan Wallen
Drake
The Weeknd
21 Savage
Bad Bunny
SZA
Tyler, The Creator
Kanye West
Future

6. Instructions for running your code (10 points)

We have three python files, *finproj_spotify_scrape_and_api.py* and *seatgeek.py* serve as the data collection files from our three sources (spotify API, Spotify website, SeatGeek API) and *visualizations.py* which takes information from our databases and processes the visualizations.

The order in which the first two files are run is not important, only that both of them must be run before the visualization file. You should not need additional tokens or access keys to run them as the ones we generated are saved in the files.

It is worth noting that as of Friday 4/21 the spotify API was down and therefore the *finproj_spotify_scrape_and_api.py* will get hung and run indefinitely due to a 500 server side error if you try and run the whole thing, this error which was not elaborated upon in the Spotify API documentation. The file previously ran without error and we were able to save a successful database from this call, which is what we used to do our calculations and visualizations.

After running the first two python files, run the *visualizations.py* file which will return our four graphs (two scatter plots and two bar charts).

7. Documentation for each function that you wrote. This includes describing the input and output for each function (20 points)

File: *inproj_spotify_scrape_and_api.py*

Function: *top10(art)*

Inputs: an artists spotify UID

Outputs: a tuple the first item is a list of ten popularity scores correlating to an artist's ten most popular songs, and the second is a list of lengths in seconds of each of those songs.

Description: This is the function that calls the spotify API, it takes the UID given and sorts through the data the API returns to give just the list of popularity and lengths. The *spotify.artist_top_tracks()* portion of the code is where we have pinpointed the API not cooperating.

Function: *calc_artists_tuples(artists)*

Inputs: Takes in the list tuples (artist, UID) for the top 200 artists on spotify

Outputs: returns a list of tuples (artist name, artist rank, average popularity of their top 10 songs, average length of their top 10 songs).

Description: This function loops through each of the top 200 artists, calculates their rank based on their position in the list, calls the *top10(art)* function and then calculates the average of both of the outputted lists from the resulting in a list of 200 tuples each with four items.

Function: *get_top_200_artists(html_file)*

Inputs: the name of the static html file for the spotify charts page

Outputs: list of 200 tuples (artist name, artist spotify UID)

Description: This file scrapes the static html of spotify's top 200 artists page and returns their names and the UID needed to access the artist specific API.

Function: *get_top_200_artists_direct(url)*

Inputs: the url for the spotify top 200 artists weekly page

Outputs: list of 200 tuples (artist name, artist spotify UID)

Description: This file writes code to use requests.get to call the same html information as the above function. We do not use this function for two reasons: 1) spotify says not to scrape their website in the same way that airbnb does, so we followed the same procedure as project 2 in downloading static files and scraping those and 2) spotify requires you to be signed into your personal spotify account to access the charts page which means it won't run properly on someone else's device, hence the static html. We wrote this function to prove that we have the know-how to execute the beautiful soup web scrape as desired, but preferred to run the other version of this function.

Function: *db_setup(dbname)*

Inputs: a string for the desired database name

Outputs: a cursor and a connection to the database

Description: this sets up the database to be called in other functions

Function: *db_add_data(cur, conn, data)*

Inputs: a cursor and connection, as well as a list of tuples to be inserted into the database.
Outputs: none
Description: adds each tuple into the database one row at a time, does not add 25 rows at a time, but rather all in one go.

Functions:

check_rows_in_db(cur)
add_rows_to_db(cur, conn, data, row)
all_db_steps(db_name, data)
These functions were all part of an unsuccessful attempt to add 25 rows of data to the database at a time. We left them in to show that we tried and got close and in case there is partial credit.

File: *seatgeek.py*

Function: *top200artists(html_file)*

Inputs: Takes in a html file, specifically the static page downloaded from spotify's top 200 weekly artists page
Outputs: The names of the top 200 artists on spotify in the US
Description: Uses beautiful soup to parse through the html and extract the names of the 200 artists on the list

Function: *get_seatgeek_pop_score(artist)*

Inputs: An artists name
Outputs: A tuple of the artists name and their seatgeek performer score
Description: This function, for a given artist, calls the seatgeek API using the artist's name and produces their 'performer score' which is a metric seatgeek creates based on initial ticket price, secondary market ticket sale demand, resale value, as well as how many upcoming events an artist has.

Function: *main()*

Inputs: The only required input is the name of the html file which is necessary to call the function *top200artists()*.
Outputs: This returns a list of 100 tuples (artist name, seatgeek popularity score), representing the 100 most popular artists on spotify
Description: This function utilizes the list of 200 top artists on spotify to call the function *get_seatgeek_pop_score()*, but only loops through the first 100, as that is what we deemed most important. It ultimately returns a list of tuples that we subsequently add to a database.

File: *visualizations.py*

Function: *artist_counts_to_txt(file)*

Inputs: a filename
Outputs: none
Description: writes text information about our calculations to a txt file

Function: *makeScatterPlot()*

Inputs: none

Outputs: none
Description: uses a LEFT JOIN statement to creates a scatter plot from two of our databases comparing the spotify popularity of an artist with how many songs they have on the top 100 billboard

Function: *makeScatter2()*
Inputs: none
Outputs: none
Description: creates a scatter plot of each artists spotify popularity versus their average song length in seconds

Function: *make_grouped_bar_chart()*
Inputs: none
Outputs: none
Description: creates a grouped bar chart comparing an artist's spotify popularity (which is on a 1-100 scale) against their seatgeek popularity (0-1) for the top 10 artists on spotify currently. Do note that because the seatgeek API represents data on concerts and tour info, some artists don't have a seatgeek popularity score.

Function: *plot_artist_songs_chart()*
Inputs: none
Outputs: none
Description: offers a descending bar chart visualization of the 10 artists with the most songs on the billboard top 100, putting morgan wallen first with the largest bar of 14 songs, followed by SZA with a bar representing 6 songs, decreasing on and on for 10 artists with the most songs.

Function: *main()*
Inputs: none
Outputs: none
Description: runs all four visualizations and the text calculation functions

8. You must also clearly document all resources you used. The documentation should be of the following form (20 points)

Date	Issue Description	Location of Resource	Result
4/17	Apple Music API not working	The internet	We gave up, Apple has way too many hoops to jump through; to be considered an “apple developer” there seemed to be an approval process to get access an API key
4/17	Getting the spotify API to call correctly, initially struggled to get access keys	Deeper parts of spotify's website and & the Spotipy documentation guide	Was eventually able to.
4/17	Eventbrite API downloads	Eventbrite has very inconsistent information, because most events are created on the site on a per event basis by different people, there ends up being way too many variations of the same string data	We’re routed away from eventbrite and went with Ticketmaster and SeatGeek for the time being
4/18	Scraping the Spotify top charts website, spotify explicitly states not to do so	Followed the same procedure as for project 2 where all of the files were downloaded as static html	This is not an issue in terms of whether our data is accurate, as spotify only has a top artists chart on a weekly basis, updated every friday, so the next update is 4/21, the day the project is due
4/19	Using the spotify website data to call the spotify API	Trial and error	Got it working eventually
4/19	Adding only 25 rows at a time to a database	Watched the entirety of sentdex’s SQLite videos (he’s a youtube tutorial guy with a lot of followers)	We're unable to add only 25 at a time, it seems easier and more practical to add all 200 rows we are interested in in one go anyway.
4/19	Ticketmaster API has very unclean data	The internet & Ticketmaster API documents	Ticketmaster only returns events by keyword, and has a very inconsistent

			data structure. We tried to pull events by Future but it returned “Future the musical”. Both price ranges and venue names were also nested in the data structure by various and inconsistent labels, so accessing it was difficult. We decided to abandon this data since there was no way to make sure the events returned were specifically for Future the music artist.
4/19	Seatgeek only lets you search by artists name as a string, not their Spotify UID number, so there are some string name discrepancies	Platform API document	We’re unable to search by spotify URI, so the popularity score returned per artist search by keyword may or may not be fully accurate.
4/20	Spotify API broke, when we came back to work on the project after dinner, having made no changes to this particular function, the call to the spotify API would get hung and never terminate	We tried so many things: - Revisiting the spotify api documentation - Debugging the code line by line - One of our housemates friend who has done some 400 level python in the CS dept - Stack overflow - Reddit - God	We concluded that Spotifiy’s API was temporarily non-functioning, the internet seems to agree it’s not the most reliable.