# SIPY

This project was created to satisfy the final project requirements for the SI206 class in Winter 2021.

This project utilizes Billboard's top 100 Artist list and pathfinding algorithms to create 100 related artists for each of the artists on the site. By using each root artist to generate a list or related artist, it was possible to see a correlation between the artists that were recommended and the genres that they were known to create in.

## Modules that you will need

We will mainly be utilizing many of python's built-in default libraries including, requests, sqlite3, beautifulSoup, matplotlib, networkx for the code portion of this project. External technologies that will be utilized include Billboard's hot 100 artists website data, as well as the spotifyAPI

Some libraries will need to be installed, by entering the following commands in your terminal

```

pip install beautifulsoup4

pip install plotly

pip install seaborn

pip install networkx

```

This program makes use of the following modules for the following tasks:

- [Seaborn](https://seaborn.pydata.org) is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. We used Seaborn to plot some data in the calculations part followers/ranking vs genre recommendations.

- [Networkx](https://networkx.org) is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. We used NetworkX to build each artist directed graph network.

- [Beautiful Soup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/) is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work. We used Beautiful Soup to scrape the Billboard Top 100 Artist's website.

- The [PPrint](https://github.com/python/cpython/blob/3.9/Lib/pprint.py) module provides a capability to “pretty-print” arbitrary Python data structures in a form which can be used as input to the interpreter. We used pprint to easily decipher JSON coming from requests.

- [Pandas](https://pandas.pydata.org/) is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. We used pandas to wrap data in our dataframe objects.

- [Secrets](https://pypi.org/project/python-secrets/) Generate unique values for variables, or use a single value per type to simplify use of secrets in access control of services while supporting a “break-glass” process to quickly regenerate secrets when needed. We used secrets to grab a theme for the artist network graphs.

## Goals

Our original goals before we had started any work or even meeting for more than a few minute were as follows

THREE API’s: Spotify, Soundcloud, Apple Music

We will be collecting song rankings per year for the last 3 years.

We will calculate the difference in genre types between the top songs on each platform

We will be using Plotly

We will create a simple mapping of all genres and total streams and mark each platform

After some further brainstorming we refined our vision and modified our original goals:

A program using Spotify’s API and also was capable of scraping [Billboard Artist 100](https://www.billboard.com/charts/artist-100)

Our second goal was to utilize Spotify’s API artist recommendation feature to analyze recommendations by genre.

Our third goal was to use a Directed Graph Network to model the recommendations by genre for each artist on the Billboard Artist 100 list.

Our fourth goal was to create an aggregate visualizations of all genre networks for the top 100 artists the was both readable and understandable.

Our fifth goal was to create code that was reusable and multipurpose with sufficient documentation.

Our sixth goal was to develop a write-up that sufficiently explains all the aspects of our code and all decisions made.

In consideration of the goals we set out to accomplish we achieved almost all of them in holistic fashion.

## running instructions

Clone the github repository and used the compiler of your choice to run the code, the version we used was the 64 bit version of python 3.8.5.

In your computer's terminal or powershell run the above pip install commands to gather the required python libraries, in addition to the ones provided by the condo package manager.

The program requires no input during run and the generated files (the path database, the aggregate image, the directed genre recommendations per artist) are stored in the program. directory and the graphs folder within the program directory.

If you would like to manually look through the path database, [DB Browser for Sqlite]() is recommended.

## sample run

The program will start out by constructing a list of existing databases in `path.db` and compare it to a list gathered by scraping Billboard's (link) website using Beautiful Soup 4. By comparing these two lists

1. The list generated from Billboard and the local Path database are compared

2. For all of the missing artist within the database, the program will create a new table and append these new artist names to them

3. The program will then generate 100 `pathed` recommendations for each new artist added to the root database

4. For each new artist a genre dictionary will be constructed and saved to the graphs folder highlighting Graph network of Spotify genres for that artist

5. A cumulative Graph network will be created for all artists using that subdictionary details that were discovered per artist

6. This final plot will be displayed

## Methods

With this program we aimed to determine how genres are recommended to users on Spotify. To accomplish this, we utilized SpotifyAPI's recommended artist functionality to generate a series of related artists scraped from the `Billboard Artists 100` list most popular artist list to generate a series of starting points to begin pathfinding from.

The `Billboard Artists 100` list is scraped with `BeautifulSoup4` and the most relevant artist on the date the program is accessed is generated and stored into the `root` (labeled part 1 in the program) . Using each row in `root`:

| artist | id | followers | popularity | genres | parent |

|---|---|---|---|---| ---|

| Kanye West | 5K4W6rqBFWDnAN6FQUkS6x | 13761535 | 92 | chicago rap,rap | null |

| J. Cole | 6l3HvQ5sa6mXTsMTB19rO5 | 13121708 | 88 | conscious hip hop,hip hop,north carolina hip hop,rap | null |

| blackbear | 2cFrymmkijnjDg9SS92EPM | 4099377 | 87 | electropop,pop,pop rap | null |

a call is made to SpotifyAPI's related artist feature and a dictionary of up to 20 related artists is returned. The related artist dictionary is parsed in order and compared to the list of discovered artists for stemming from this artist and appended to this artists 'SUBPATH table if the recommended artist is newly discovered. If the topmost artist returned from the API already exists within the `discovered artist` list then the subsequent item is chosen from the API call list and is counted as a miss. The pathfinding concludes when the number of misses is equivalent to the number of related artists returned by the SpotifyAPI or the path reaches 100 Entries. For the completion of each generated path a completion message is printed as which states the artist's name and the length of the generated list:

```

>>>>>> {completed: Khalid -> 100}

>>>>>> {completed: Imagine Dragons -> 87}

>>>>>> {completed: Tom Petty and the Heartbreakers -> 100}

```

After each artist within the `root` table has a path generated for them (labeled part 2 in the program) , the next portion of the program begins. For each artist path, the genres are parsed and mapped to one another using the following logic and assumption that successive entry in an artist's path call is directly related to the one preceding it.

1. A set of items found in SUBPATH entry n and a set for items SUBPATH n + 1 are created

2. By taking the difference of these two sets, set (n) - set (n + 1) and add every entry appearing in this set to a dictionary of lists into the following format {n : a, b, c , d}

3. The dictionary is then converted into a series of edges in the format of a directed graph network as follows {(n, a), (n, b), (n,C), (n,d)}. In this case n, a, b, c , d all function as nodes and the paths (n, a), (n, b), (n,C), (n,d) are the edges that connect these nodes.

4. Using the created edgelist from above and the `NetworkX` python module, a directed graph is created with a scalar colormap, nodesize dependent on the number of paths to that node. While a directed graph is created the arrows are omitted for the visuals.

After each graph is created, it is stored in the graphs folder using the naming convention <artist\_name>.png. After all of the graphs are created, part 3 of the code is complete.

The next part of the program aggregates all of the genre data from all of the artists and constructs a treemap using `plotly.express` of the relationship between artist's popularity, recommendation-parent, follower count using the genre to group related artist by types of music that they are associated with.

The data is parsed from every file with the following sqlite commands:

```

a) cur.execute("SELECT name FROM sqlite\_master WHERE type='table';")

b) cur.execute('SELECT \* FROM ' + table + ' WHERE genre!=""')

```

The first query gathers a list of all of the database names within the `path.db` database and passes them to the second query prompt which selects all rows which have a non-empty genre from the table it is parsing. Together these two functions gather all of the data stored from the SpotifyAPI

With the help of a `pandas` dataframe wrapper the data is converted into a form that works well with the plotly express treemap module. For the generated graph, the size of each artist's square is relative to the their popularity scores and the coloring is related to the number of followers that they have on instagram. When looking at the categories, a couple duplicate artists might be observed, due to the fact that each artist generated path was generated irrespective of any other artists path meaning that duplicates were tabulated in this process.

While this generated graph is robust, it is a simplification of the true nature of the data collected. It would be nearly impossible to effectively graph each artist within each genre they were associated with as can be seen in the J. Cole example above. He is associated with "conscious hip hop,hip hop,north carolina hip hop,rap". Using the python `secrets` module, just one of these genres is picked for the creation of the graph which makes the final graph an imperfect representation of the data. However, allowing duplicate J. Coles allows for the chance for him to be picked for multiple genres creating a truer representation of the aggregated data.

## code documentation

The following are the documentations for what each function does

```

# creates a new database

def initialize\_database(db\_name):

# creates a table with artist data

def create\_table(cur, conn, table\_name):

# appends the table with new data

def append\_table(cur, conn, table\_name, data):

# builds the url from the artist name or id

def build\_url(artist\_tag, mode):

# gets data from a singular artist URL via SPotify API

def get\_artist\_data(req\_url):

# Processes all of the root requests

# returns what it appends -> a list of root artist data

def make\_append\_root\_requests(cur, conn, table\_name, urls):

# gets a set of up to 20 related artists from a artist recommendation

def get\_related\_artists(req\_url, parent):

# generates a single artist path with a stack pathfinding algorithm

def generate\_path(cur, conn, table\_name, url, name):

# scrapes billboards website

def scrape\_billboard\_artists(url):

# build a directed graph network using genres and nodes and edges as relations

# the graph is unweighted for visual comfort but is calculated in a weighted fashion

def build\_graph\_network(cur, conn, table\_name, theme, time\_stamp):

```

The breakdown of the parts of the code do the following

Part 1:

1. The Billboard top 100 artists list is scraped and stored into a list

2. The new artist list is compared to the existing artists to determine which to add to the ROOT if any

3. If a ROOT table does not exist in the path database, the ROOT file is created to store the path heads in path.db

4. The ROOT table is appended with the new artist list (name, id, followers, popularity, genres, NULL)

Part 2:

1. Sets the start position of the recently appended potion of the root as "start" to avoid path regeneration for paths that already exist

2. Iterates from "start" to the end of ROOT and builds a SpotifyAPI call using the artistID gathered from the ROOT

3. Creates a new table for each new artist in ROOT that is compatible with sqlite3's formatting regulations

4. Builds a 100 item long path per new artist within ROOT using linear pathfinding and appends data to each new artist's table

Part 3:

1. Creates a directory called graphs to store all the graphs generated in the following part

2. Using python secrets, a random theme is selected and pushed to the graph builder function

3. A graph is created by mapping recomended genres stemming from ROOT artist on to one another and generated with the networkx module

4. The graphs are saved in .graphs/ directory

Part 4: UPDATE

1. This creates a cumulative graph network of all artist's genres interconnection using the data provided by the SpotifyAPI recommendations feature

2. The graph shows the weightage of each recommendation and the frequency with which its recommended

3. The graph is displayed and stored in the project directory

4. A runtime summary message is printed

## calculations

For the calculations, we determined the relationship between the followers and the rankings provided by the SpotifyAPI related artist’s functionality and normalized this data by dividing the followers (followers refers to the number of followers that a user has the ranking is the Secret Sauce Score that Spotify assigns to a person by their payability and their relevance) by the rankings per artists. Through our project we were able to calculate genre relevance and assign numerical scores to each genre according to the frequency in which they appeared, using these scores we calculated an average genre score per artist. We then plotted this data.

## problems encountered

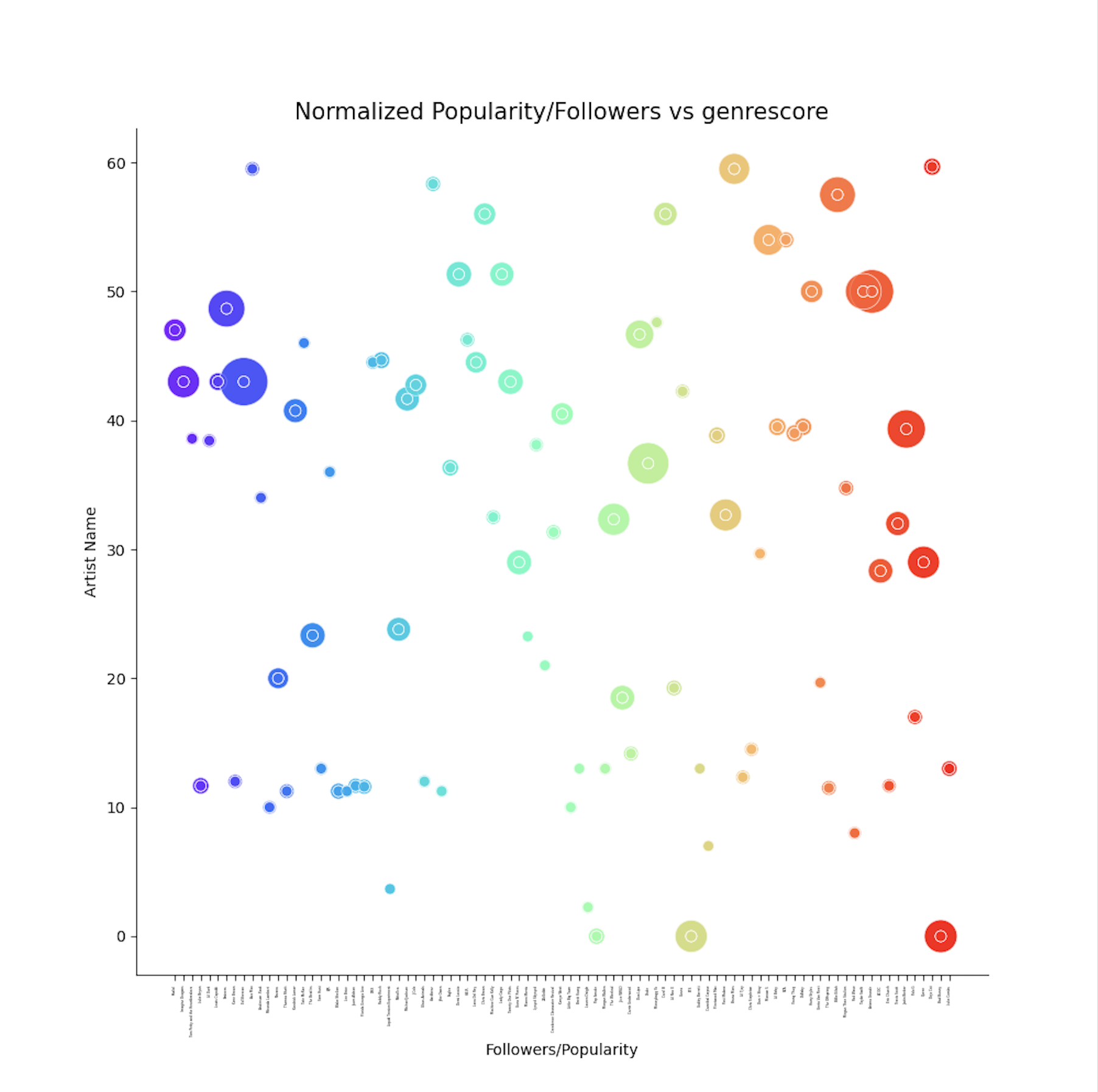
Problems we faced included extensive run times- with a total length of about 15 minutes for two of the 3 portions of our project (database generation 8900 api calls, and the data visualization 89 directed graph networks). Due to the batch processing nature of our program, testing the functionality of one aspect meant running tests on the entire set of Artist via Billboard often making if difficult to complete portions of code very fast. This coupled with poor Python documentation for many libraries including Seaborn and Networkx led to hours of research and problematic halts. Thankfully, these issues were offset by the nature of our working relationship; tasks could easily be split between members and separately implemented reducing total project work time. Overall the large datasets that we created ad hoc were difficult to parse with SQLite (we needed to learn obscure portions of the module to achieve desired functionality) , additionally, by constructing a huge database (8900 API Calls, each taking approximately a hundredth of a second) . Furthermore, the nature of the rubric handcuffed our ability to create a truly interesting project that mirrored exactly what we wanted and led to early feelings of burnout and slow progress, leading us to completely change the nature of our project (see updated goals above). Aspects including the JOIN command and the length of the database conflicted with the goals we hoped to achieve and the portions of our project that address these tasks feels unnecessary .

#### contributors

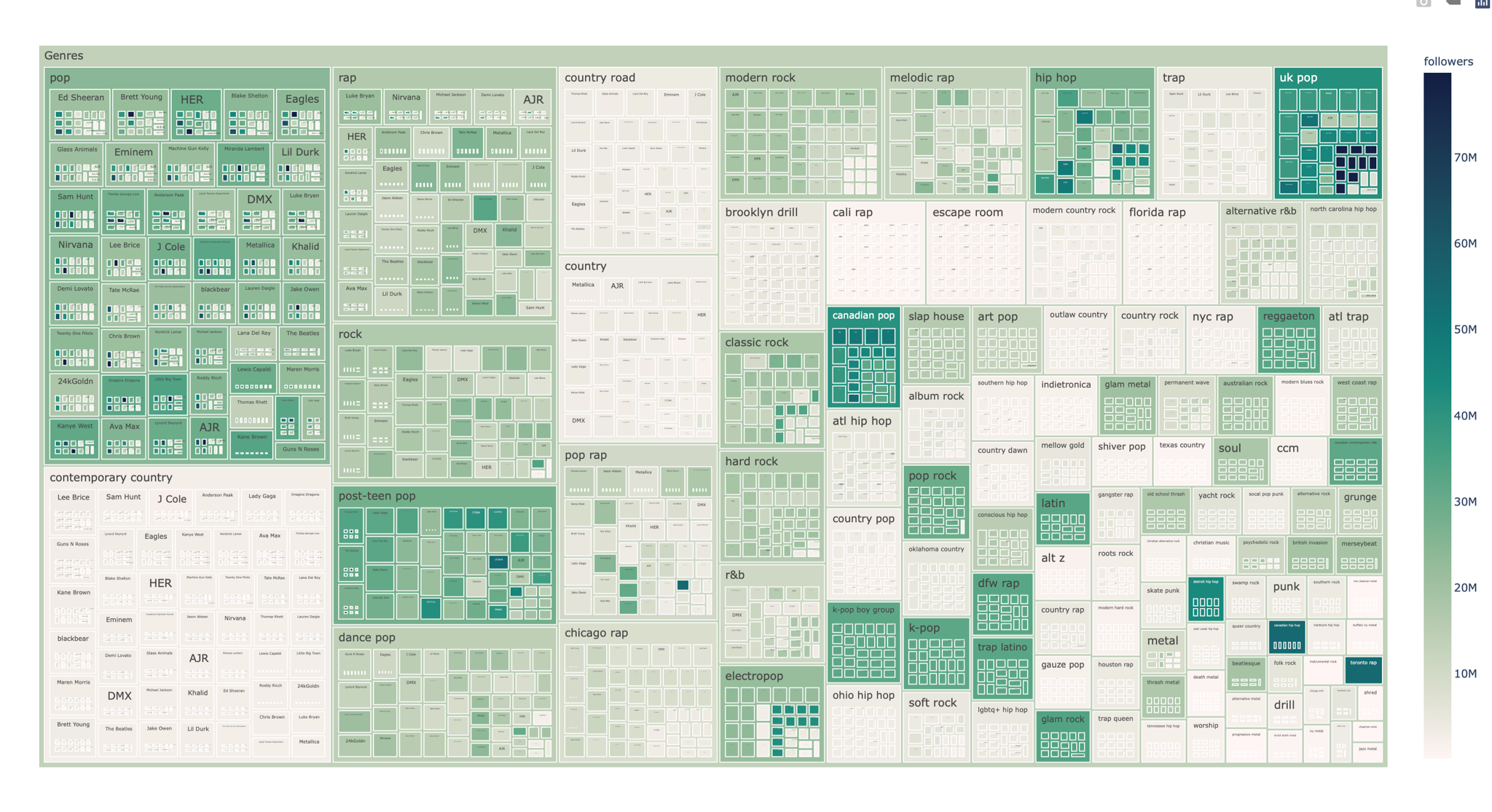
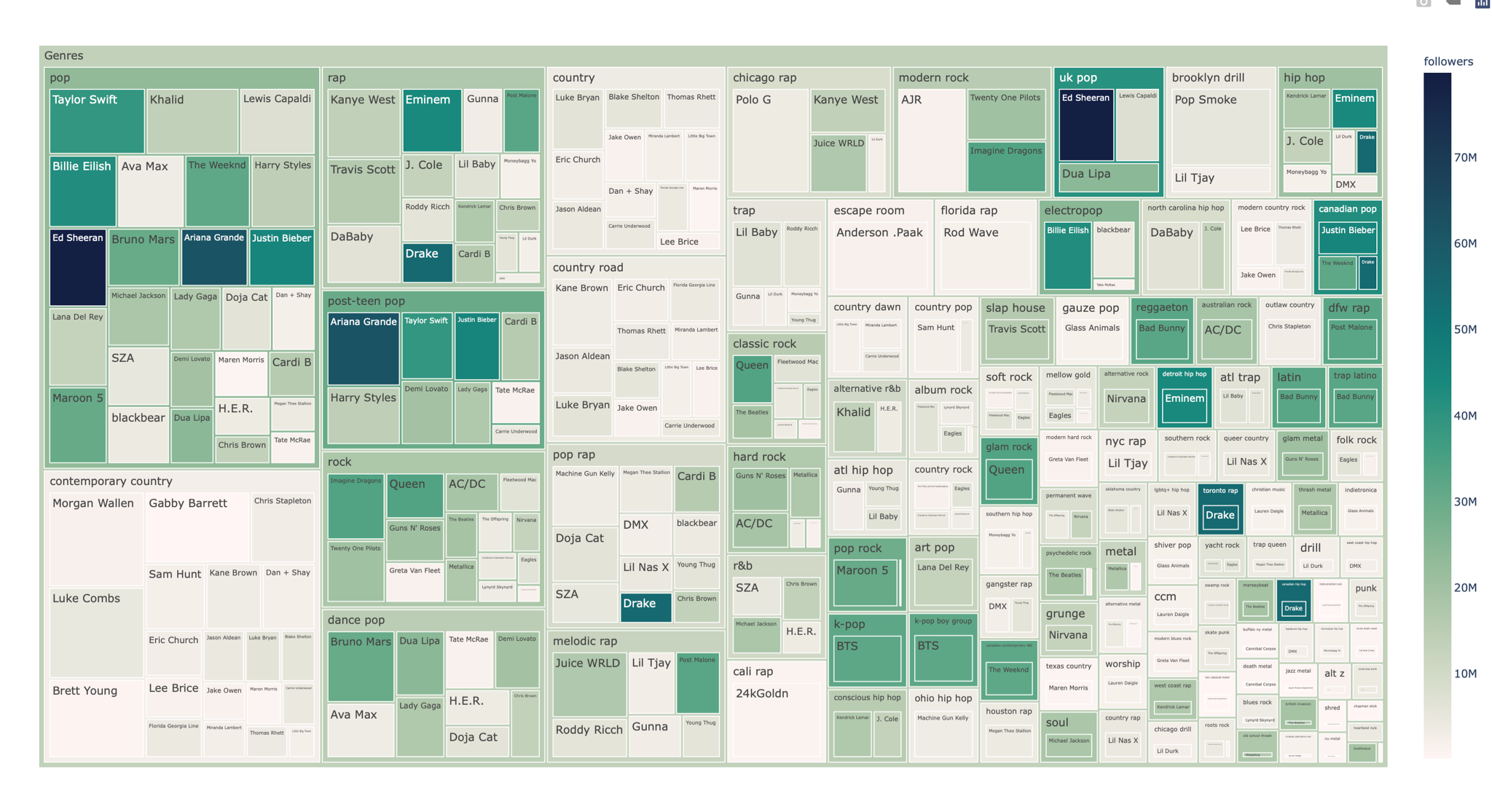
@Suneeth Torke

@ Brandon Huggard

EXTRA CREDIT GRAPH 1:



EXTRA CREDIT GRAPH #2:

1. 
2. 

Extra Viz - 1

* This representation is for [...] it includes a connected network graph of the recommendations stemming from this artists genre. We define genre as the genre information derived from the Spotify API, using this information we built a network of recommendations using linear pathfinding. What you see above is an unweighted graph of the connected recommendations. Each circle represents a node, each edge connected two nodes represents a recommendation pathway made by Spotify.

Extra Viz - 2

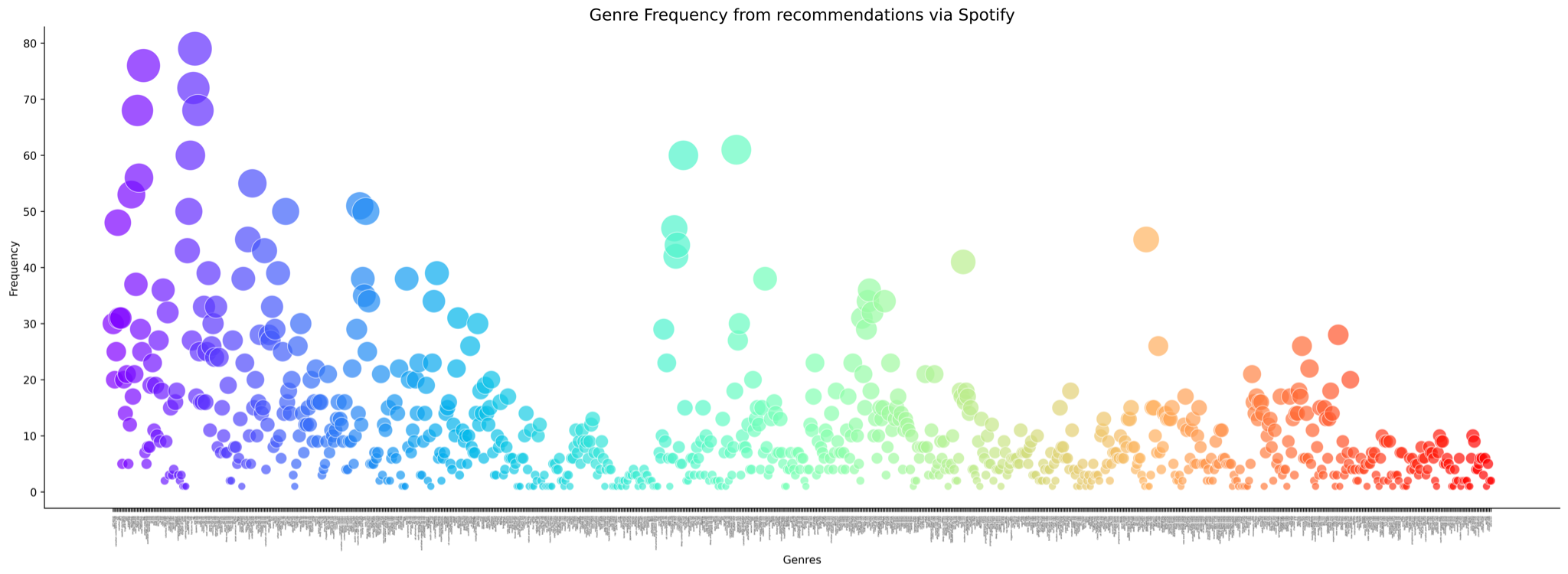
* After creating our calculations by normalizing the followers by the rankings assigned by Spotify we calculated a genre score based on the frequency of which a genre is recommended and plotted the relation between these two categories.

Visualization 1:

* + 1. The graph illustrates a connected network graph of the recommendations stemming from this artist’s genre. We define genre as the genre information derived from the Spotify API, using this information we built a network of recommendations using linear pathfinding. What you see above is an unweighted graph of the connected recommendations. Each circle represents a node, each edge connected two nodes represents a recommendation pathway made by Spotify.

Visualization 2:

* + 1. The cumulative relation treemap illustrates the relationship between four major aspects of the data we collected. It plots artists in relation to their genre, in relation to their followers, in relation to their popularity; the way it does this is by representing the popularity based on the size of the boxes the followers based on the color of the box, the genre based on the nested boxes, and the artists based on the named boxes.

Visualization 3Visualization 1 is below, Visualization 2 is the bottom visualization.