DOTIFY

MUSIC RECOMMENDATIONS
FOR EVERY MOOD

CONRAD STANEK
DEVAN SRINIVASAN
MAJDA LOJPUR
SALWA ABDALLA

CSC111 TERM PROJECT

Dotify: Music Recommendations

Conrad Stanek, Devan Srinivasan, Majda Lojpur, Salwa Abdalla

Friday April 16th 2021

Introduction

According to Streaming Machine, almost 3.1 million songs (Stein, 2020) are released every year on top of an already existing massive library of songs. With such a large selection of music it can be difficult for one to find new songs and artists that they enjoy in the midst of all the music that they find uninteresting. This got us thinking of graphs! The reason one likes a song is because it is similar to their current taste, but this is an abstract idea. Can this be quantified? Moreover, given that we are some of the people who don't appreciate having to make new playlists, **how can we create a customizable software that generates spotify playlists of new music that a user would like? Additionally, how can we adjust the software to nudge listeners into new genres to expand their music taste?** And so, this gave birth to our idea of using audio features of songs along with a uniquely connected graph to create a system that explores new music!

Dataset

The dataset we used for the project is a Kaggle Spotify Dataset collection of roughly 175000 songs from 1921-2020. The whole dataset is composed of 3 separate csv files:

- `data.csv` Main data file, contains all songs and their properties, i.e danceability, accoustincess etc. All of the headers are used besides specfic infromation releated headers, like explicit, year, duration_ms.
- `data_by_genres.csv` Contains all artists and their properties i.e danceability, accoustincess etc. but it omits some that do not make sense like release date. Also includes the genres that the artist makes. All of the headers are used
- `data_w_genres.csv` Contains all genres and their properties, i.e danceability accoustincess. Same properties as `data.csv` but it omits some that do not make sense like release date. Only the artists and genre headers are used.

Computational Overview

Data Types

The types of data that was used to represent our chosen domain were the following. All of these are found in `song_graph.py`

`Song`

This class represents a song. It contains all the relevant song attributes separated by auditory property (danceability, energy, ...) and then information (id, artists, ...). The `Song` class, when instantiated, represents a vertex in our `SongGraph`. Being that it is a vertex, it contains a neighbours dictionary that stores it's weighted edges. Each weight is a similarity score generating using the song's properties. Note that the way we calculated the similarity score means that the lower the score, the more similar the two vertices are. (It's more of a difference score)

`SongGraph`

This class is our weighted nested graph class. It is the graph of songs that corresponds to a particular genre, in which all songs share. The reason nested graphs were used, as opposed to having all songs in one graph, was to a) improve computational efficiency b) allow for more specific computations (having the songs already separated by genre)

`Genre`

This class represents a vertex in our weighted `GenreGraph`. The class has relevant attributes such as the genre's average properties, as well as its name and corresponding `SongGraph`. Being that the graph is weighted, this node also has a neighbour dictionary that stores its similar genres.

`GenreGraph`

This is the weighted graph that stores all `Genre` vertices, whom themselves contain the respective `SongGraph` graphs.

Major Computations

The bulk of our computations can be separated into three sections, loading, algorithms, and visualization. Graphs

are the backbone of our project, as they connect songs and genres together using quantitative measures. More specifically, songs are connected using a weighted edge, the weight being a similarity score calculated from the song's properties. Using this framework, we then developed unique algorithms and traversal patterns to take input and generate new songs we think the user may want to try.

Loading

To load the genre graph all of the genres and their average properties are loaded from `load_genres`, all artists and the genres they make are loaded from `load_artists_to_genres` and all songs and their properties from `load_songs`. Then a dictionary of genres to songs is created and is done by looking at each song's artist and then the genres the the artist makes and then comparing the average properties to select the most likely genre (This is because spotify does not store genre information about individual songs). Then each genre has a song graph created for it consisting of the songs in the genre it corresponds to. Instead of finding a similarity score between each song (because this takes to long) each song is given a rating and then the ratings are sorted and songs that are close to each other are connected. All of the song graphs are held in genre objects which are stored in a genre graph and connected in the same way.

Algorithms (`computations.py`)

We have 6 algorithms, all of which are unique graph methods that recommend songs in innovative ways. Below is a brief description of the computationally complex algorithms.

- `bfs_gen` This function traverses a song graph in a level based manner. It was inspired by breadth first search, which uses queues to explore in a level based manner as well. This is a method intended to stay closer to the user's existing taste in music.
- `artist_gen` This function is a function that uses recursion, to generate songs of the same artist (or a closely related one). It has a helper function that acts as the recursive step called `rec`.
- `explore_new_genres` Finds genres that are neighbours to inputted genres and calculates a biased similarity score between songs in the input list and songs in the neighbouring genres, that also takes into account preferences and then returns the 'best' 11 songs.
- `find_uniquely_connected` Finds songs by taking into account the degree of each song. Songs with a lower degree are preferred, since they are more uniquely' connected to other songs.

Filtering and Aggregation

We filter song choice by allowing the user to set custom parameters in the settings window, as well as utilize a visited set in our algorithms to prevent infinite loops/recursion, and ensure only **new** songs are recommended.

Spotify Methods (`spotify_methods`)

There are two major methods here.

`spot_song_to_vert`

This method takes in Spotipy search results, and formats it into a vertex. While this wasn't computationally taxing this function is critical as it converts large JSON formatted search results into our specific `Song` data type.

`song_to_genre_guess`

This method is critical to creating song vertices. Given a dummy vertex (one that hasn't been incorporated into the graph yet) this method guesses the best genre it should be assigned. When we receive songs from input, that don't exist in our graph already, this function determines which nested `SongGraph` it should be inserted into.

Visualization

Our visualization comes in two parts, that being the GUI and the Graph Visualization.

GUI (using Tkinter) (`main.py`) The GUI has many methods that set widgets in place, as well as header methods that call major methods from other files. No method here is very computationally complex.

Pygame Visualization (`pygame_visualization`) Displays any genre/song/(sub genre) graph. Also connects to plotly.

Interactive Visualization

Our software opens a GUI that contains everything the user needs to use our software.

Also the program outputs the playlists and automatically adds it to a tethered spotify account.

Libraries

Spotipy

This is Spotify's Web API, a major component to our project. The main functions that use this library are `find_track_options`, `pull_playlist`, and `generate_playlist`. Using this library allowed us to read data from spotify's database, use playlists and tracks from the user tethered to the project (the dummy account), and lastly generate playlists to load onto the user's account. Using Spotify's Web API also allowed us to use their song search engine which was helpful for allowing the user to find songs to add to the input list.

Pygame

We use pygame to give a preliminary visualization of the graph. Pygame was very helpful as its very versatile and unrestricted, and allowed users to select sub graphs and have them displayed which is complicated to do with plotly.

Plotly

Plotly was used to visualize the graphs formally. By using their software we allowed the user to zoom in and out, as well as examine specific portions of the graph using plotly's framework, i.e individual vertices.

Networkx

Networkx was used in a basic manner to organize our graph for visualization.

Tkinter

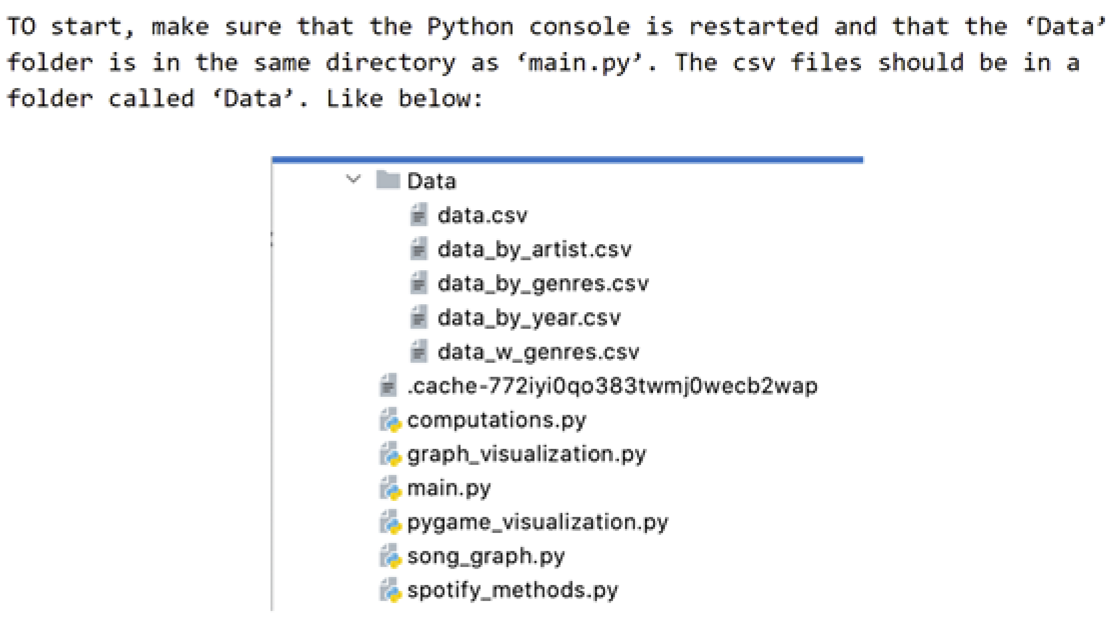
Tkinter was used to build the GUI for our project. We used their widgets to make all interactive components of the GUI including buttons, list views, and drop-down menus.

Instructions

Our program does not use any other programming language other than Python. The Python libraries required do not require any special installation instructions, and we have provided the marking team with a spotify account to use. All libraries that need to be externally installed are listed in requirements.txt.
To download the dataset use the link below.

`https://drive.google.com/drive/folders/1CeYJfHIhGTxMxZhMLqHww_oOD0p79r0b?usp=sharing`

Note you may need to configure your browser settings to allow this download, as we had to on Chrome. Google drive will instruct you on how to do this. Detailed (visual) instructions on how to use our software after running `main.py`: Note that you can't use the GUI and visualization simultaneously as you are in the pygame loop while using the visualization, and hence can't exit until the window's closed.



TO start, make sure that the Python console is restarted and that the ‘Data’ folder is in the same directory as ‘main.py’. The csv files should be in a folder called ‘Data’. Like below:

```
Data
    data.csv
    data_by_artist.csv
    data_by_genres.csv
    data_by_year.csv
    data_w_genres.csv
.cache-772iyi0qo383twmj0wecb2wap
computations.py
graph_visualization.py
main.py
pygame_visualization.py
song_graph.py
spotify_methods.py
```

STEP 1: Run ‘main.py’
It will prompt you for a threshold in the console, enter in a number between 0.001 to 0.5. The higher this value is the longer the program will take to start, but the more extensive the graph will be. We’d recommend around 0.2-0.3.

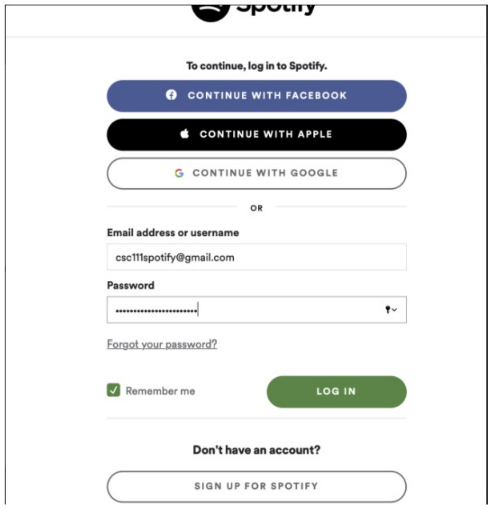
NOTE that it may take up to a minute for the entire graph to be loaded if the threshold is large, (How long it took on my macbookPro).

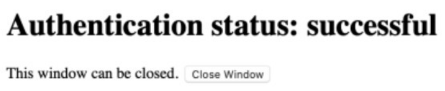
NOTE using the software for the first time, it will prompt you for authorization to use our dummy spotify account when the GUI is interacted with. This will open a window in your browser. BEFORE PRESSING AGREE, make sure you are LOGGED INTO THE RIGHT ACCOUNT because it may try to automatically log you into your personal account if you are logged in on google or on the spotify app on your computer. The account to authorize this with is:

Email: csc111spotify@gmail.com
Password: davidandmarioCSC111man$

The password is in no way meant to mock, it is out of respect for our professors and was chosen as a fun choice for the dummy spotify account.

When authorizing the account you should ensure that the authorization page recognizes the CSC111 account as the one authorizing it. For reference when you first run the file it should look like the photo on the left (without the details filled in) and then when you logged in it should look like the photo on the right.





Authentication status: successful

This window can be closed.

Login to Authorization Successful Authorization

If you accidentally authorize with the wrong account or something goes wrong, delete the .cache item in your project directory, sign out of spotify everywhere (the website and any downloaded spotify app), and run the project again you should be able to reauthorize it.

(This should work we tried it on a fresh install and deleting the .cache and signing out worked, but the actual way you are meant to reauthorize is too complicated for us to understand :/)

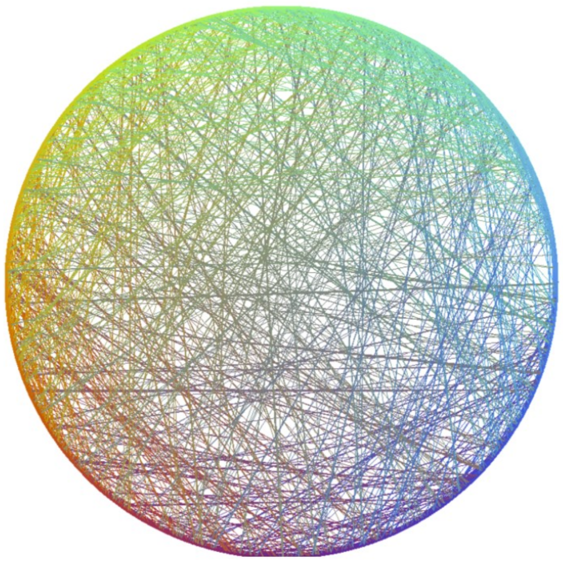
Note: The spotify account expires by May 15th

STEP 2: Input and Visualizing

The VISUALIZE BUTTON at the top right hand corner visualizes the genre graph, (it is just added functionality beyond the main playlist generation). It may take a few seconds to load fully depending on how large the graph is. In the interface you can do the following:

- Type in any genre we have listed in our graph, and press ENTER to view its SongGraph (if the genre doesn't exist nothing will happen). Some examples to try are: pop, rap, country, metal. Note that you can press the right shift to delete a query.
- The complete list of genres can be found in the 'data_by_genres.csv'
- While viewing the genre graph type out any genre in the genre graph, and click to show only that genre and its neighbours in the graph
- When viewing any graph, press TAB to open the graph in plotly. NOTE if the graph is too large (typically occurs with GenreGraphs of a high threshold) plotly won't be able to load. (For example, a threshold of greater than 0.1 will take too long if you try to view the genre graph. To visualize the graph on plotly we'd recommend trying thresholds such as 0.01, 0.001)

Note that the GUI window can be resized so if things are not appearing properly that should fix it.



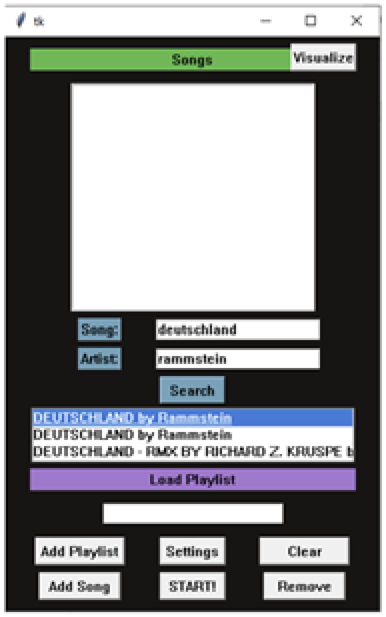
Example of a visualized graph on pygame

INPUTTING can be done two ways:

- SONG entry: Type in any song name and artist name into the entry fields, then press search to look for the song. When the desired song is found select it and press ADD SONG to add it to the input list. Any song can be removed by selecting it in the ListView and pressing REMOVE SONG.

- PLAYLIST entry: To load a whole playlist to use as input from the connected spotify account, type in playlist's name and press LOAD PLAYLIST. This will append the playlist to the ListView in addition to any preexisting input. There are 4 preloaded playlists for testing purposes: 'Rap Playlist', 'Pop Playlist', 'Country Playlist' and 'Metal Playlist'. Any playlist you wish to preload must be public on the connected spotify account.

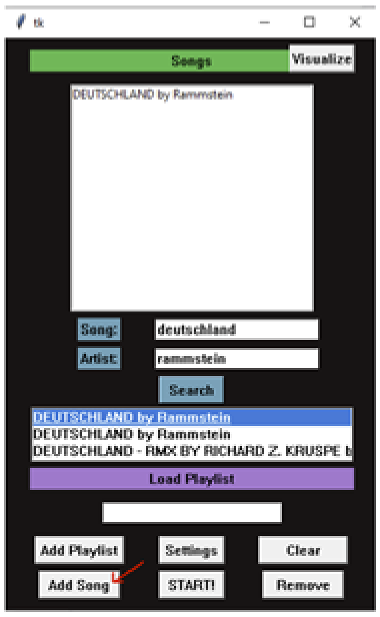
- CLEAR clears the list view



Song: deutschland
Artist: rammstein

DEUTSCHLAND by Rammstein
DEUTSCHLAND by Rammstein
DEUTSCHLAND - RMX BY RICHARD Z. KRUSPE b

Searching

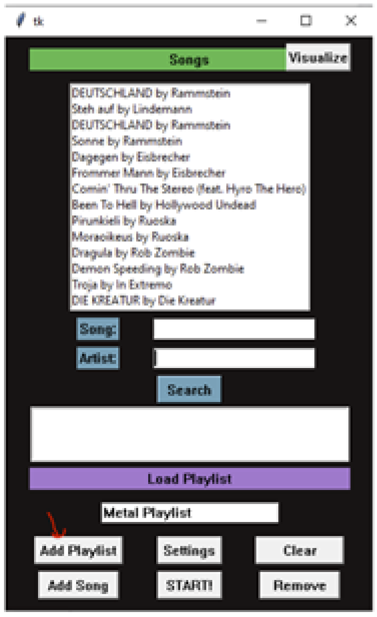


DEUTSCHLAND by Rammstein

Song: deutschland
Artist: rammstein

DEUTSCHLAND by Rammstein
DEUTSCHLAND by Rammstein
DEUTSCHLAND - RMX BY RICHARD Z. KRUSPE b

Adding the Song

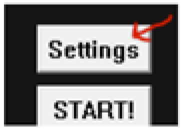


DEUTSCHLAND by Rammstein
Steh auf by Lindemann
DEUTSCHLAND by Rammstein
Sonne by Rammstein
Dagegen by Eisbrecher
Frommer Mann by Eisbrecher
Comin' Thru The Stereo (feat. Hyro The Hero)
Been To Hell by Hollywood Undead
Pirunkieli by Ruoska
Moraoikeus by Ruoska
Dragula by Rob Zombie
Demon Speeding by Rob Zombie
Troja by In Extremo
DIE KREATUR by Die Kreatur

Metal Playlist

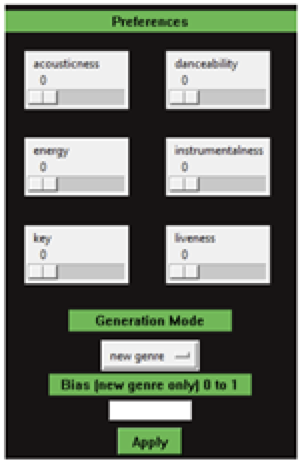
Add Playlist

STEP 3: Settings
Click the SETTINGS button to open the settings window. This window offers the following features:

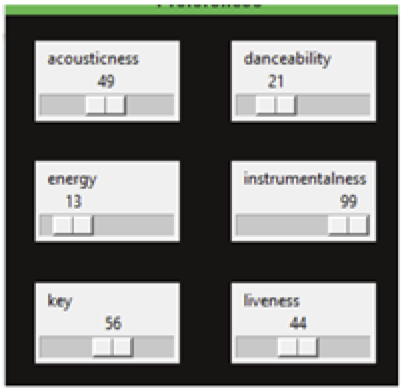


- Sliders:
    - There are 6 sliders that can be configured to your preference. These will be taken into account when generating songs using the following generation methods: custom gen, new genre, unique songs
- Generation Mode:
    - This dropdown menu allows you to select 1 of 6 generation modes, the descriptions of each mode are listed in the GENERATION MODE section of this document
    - The text entry for BIAS is only applicable to the generation mode: new genre. If entered with any other generation type, it will not be used.

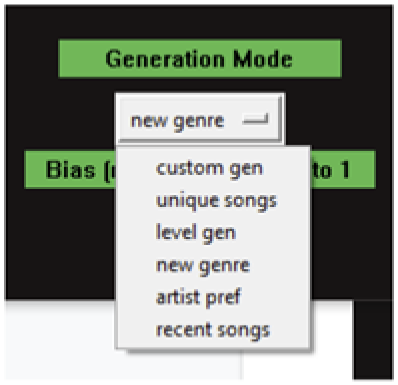
After changing something, press apply and you can close the window.



Initial State

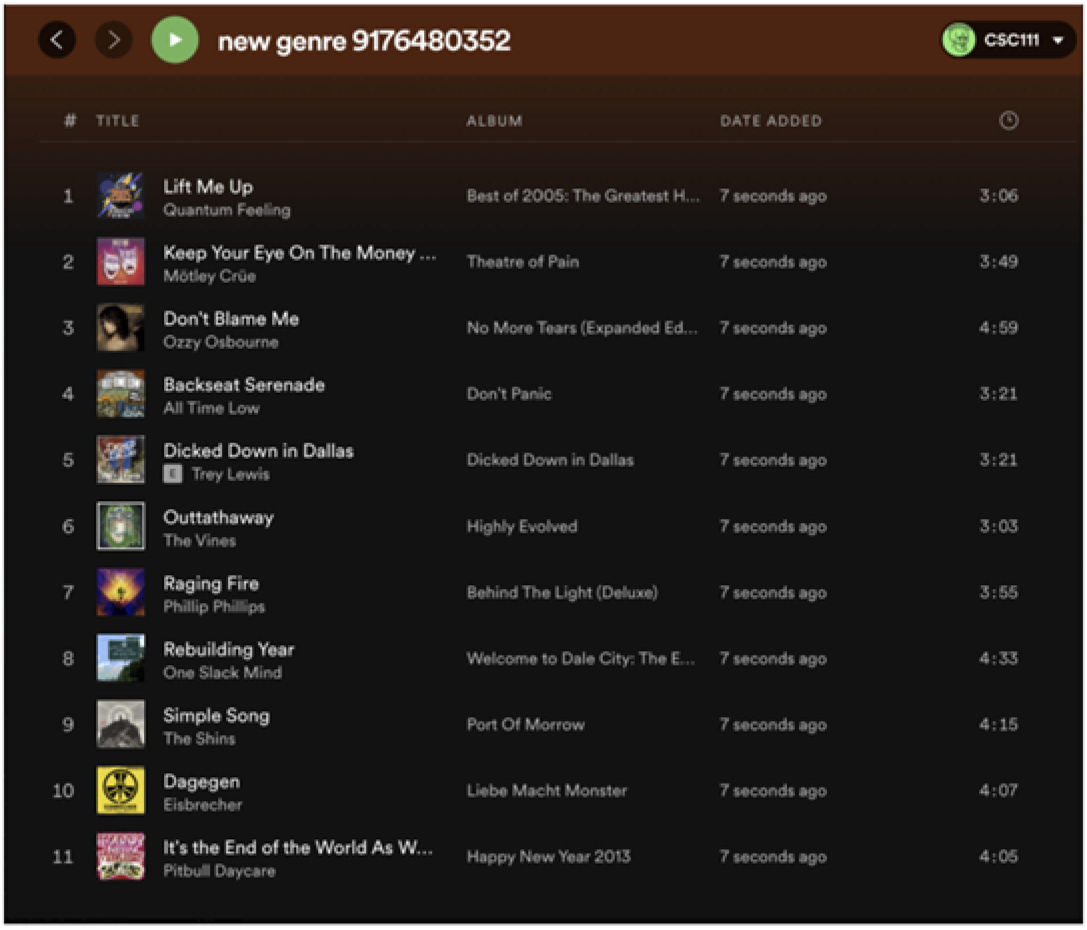


Customizing the Sliders



Customizing Genre

STEP 4: Start!
Click Start and check the spotify account to see the playlist generated. It will be named based on the generation mode selected, followed by a unique randomly generated number ID.



#	TITLE	ALBUM	DATE ADDED	🕒
1	Lift Me Up Quantum Feeling	Best of 2005: The Greatest H...	7 seconds ago	3:06
2	Keep Your Eye On The Money ... Mötley Crüe	Theatre of Pain	7 seconds ago	3:49
3	Don't Blame Me Ozzy Osbourne	No More Tears (Expanded Ed...	7 seconds ago	4:59
4	Backseat Serenade All Time Low	Don't Panic	7 seconds ago	3:21
5	Dicked Down in Dallas E Trey Lewis	Dicked Down in Dallas	7 seconds ago	3:21
6	Outtathaway The Vines	Highly Evolved	7 seconds ago	3:03
7	Raging Fire Phillip Phillips	Behind The Light (Deluxe)	7 seconds ago	3:55
8	Rebuilding Year One Slack Mind	Welcome to Dale City: The E...	7 seconds ago	4:33
9	Simple Song The Shins	Port Of Morrow	7 seconds ago	4:15
10	Dagegen Eisbrecher	Liebe Macht Monster	7 seconds ago	4:07
11	It's the End of the World As W... Pitbull Daycare	Happy New Year 2013	7 seconds ago	4:05

Generated Playlist
(using similar input and settings from above)

Generation Modes

Level Gen

This generation type uses techniques similar to ‘Breadth First Search’. It traverses the graph one level at a time, and uses similarity as well as preference score to recommend songs.

Custom Gen

This generation type traverses through vertex neighbours and optimizes preference score.

Artist Pref

This generation type uses recursion, to recursively look through neighbours and use similarity to score, while doing its best to only recommend songs of the same/similar artists.

New Genre

This generation type creates a playlist that biases songs that have a genre that is different from the input songs. If bias is 1, the songs that are recommended have a different genre than all of the songs in the ListView, if the bias is 0 it just recommends songs based purely on similarity.

Unique Songs

This generation type tries to recommend songs that are uniquely connected to the input list. Meaning the songs that are recommended are the ones who are neighbours to the input list and are not neighbors to many other songs.

Recent Songs

This generation type recommends songs but only ones that have been released recently. Recently is the oldest song that is in the input list, so if the oldest song you inputted was released in 2000, 10, 1 then all of the songs that are recommended will have been released after 2000, 10, 1.

Updates

There have been a few changes of our original idea from our proposal which come from both the feedback of the TAs, and our own fine tuning to while making our program:

- We finalized our data types (described earlier). This was not concrete in the proposal
- As the TA recommended, we used Tkinter for our main GUI as opposed to Pygame.
- We included visualization and used `networkx` and `plotly` as well as `pygame` for it.
- Contrary to the TA's recommendation and even our own plans, we structured our graph very differently. We separated our graph into subgraphs by genre. We also implemented **relative similarity score** in correspondence to the threshold entered, to drastically improve the computational efficiency of our project. We also did not create separate edges for each parameter between songs as per our original idea, but rather used just one similarity score. The TA recommended we use cosine similarity, and we were going to, but found that the solution we have was most efficient for our purposes.
- We did not use binary search trees as we thought we would need to. BST was originally intended to search our graph, but after separating our graph by genre and using dictionaries to store vertices (they have negligible search time complexity) the binary search tree was unnecessary.
- We did not include a way to filter out specific songs from being recommended since we changed the way the we traversed the graphs

Discussion

In making this project we hoped to create music recommending software that gave the user some control over the songs it recommended to them. We were able to do so, and from our testing, the playlists that were generated are fairly good, (I've unironically added some songs that were recommended to me to my liked songs on spotify). Obviously there are a lot of limitations however. One problem with the dataset is that spotify does not collect the genres of songs so we have to guess what they are which makes our graphs less accurate. Another problem is that the dataset itself is limited. It does contain 170000 songs but that still is a small number relative to the number of songs on spotify which leads to a lot of niche genres having very few songs to recommend from and as such the playlists that are generated wont vary too much. Another problem with the dataset is that it contains duplicate songs with different ID's. Meaning its the same song but the artist published it twice in different albums so our system can recommend the same song multiple times making the returned playlist mediocre. One last problem with the dataset is that it contains some songs that aren't playable in Canada, and the process to remove such songs would take a long time. The last problem really is that song recommendation is not an algorithmic process so in order to create an effective system it would require a lot of experimenting, and combinations of really complex algorithms and AI's to produce really good results.

The next steps to improve our project would first be to have some more extensive dataset with a more accurate way of connecting songs. Ideally we would be able to pre-load a large graph of songs that are connected by comparing every song instead of using the rating system described above to make the program run faster. More important then this however would be to improve the computations to make them more modular. Meaning that currently we have a few different algorithms that just take an input list and return a list of recommend songs. Instead however it would be more interesting to have a bunch of computations/functions that each do a specific portion of the song recommending process, (for instance determining what genre to recommend songs from) and have parameters that can be passed into each function that depending on what they are lead to different generation styles. We think that this would overall lead to better results (and it would be easier to add features if we want to change anything about the function) as more time could be spent on perfecting each part than simply making a whole algorithm that does the entire process from start to finish that has to implement each portion of the song recommending process within it.

Additionally, we would ideally have this type of software implemented along with Spotify itself. In doing so, each of the millions of users could contribute their songs and their playlists to our giant graph. This would improve the graph's knowledge as well as could be used in similarity score's. The last step we would like to note is the possible use of AIs. While we were happy with a lot of the playlists we generated (many of us actually use them now), we also realized that recommending songs isn't as mathematical as we hoped it was, but rather is very personal and specific. That being said, it can have many general patterns in terms of listening trends and songs that appear with eachother in many playlists, which is something only feasible on a global, multi user scale. This lead us to think that having an AI learn and study such patterns then make probabilistic recommendations, in concurrence with math and algorithms similar to ours, would really perfect the goal of computationally recommending songs. We learned a lot from this project and thoroughly enjoyed the whole process from working with API's for the first time to getting to expand our music taste.

References

Ay, Yamac Eren. "Spotify Dataset 1921-2020, 160k+ Tracks." *Kaggle*, 5 Apr. 2021, www.kaggle.com/yamaerenay/spotify-dataset-19212020-160k-tracks.

Lamere , Paul. "Welcome to Spotipy!" *Welcome to Spotipy! - Spotipy 2.0 Documentation*, 28 Feb. 2021, spotipy.readthedocs.io/en/2.18.0/.

Liu, David. *CSC111 Assignment 3: Graphs, Recommender Systems, and Clustering*, University of Toronto, Mar. 2021, www.teach.cs.toronto.edu/~csc111h/winter/assignments/a3/handout/.

"NetworkX Documentation." *NetworkX: Network Analysis in Python*, NetworkX, 2020, networkx.org/.

"Plotly Python Graphing Library." *Plotly Python Open Source Graphing Library*, Plotly, 2021, plotly.com/python/.

"Pygame Front Page." *Pygame Front Page - Pygame v2.0.1.dev1 Documentation*, Pygame Documentation, 2021, www.pygame.org/docs/.

Stein, Germano. "How Much Music Is Really Released per Year?" *Streaming Machinery,* 13 July 2020, streamingmachinerary.com/2020/07/13/how-much-music-is-really-released-per-year/#:~:test=That' s%20a%203.1%20million%20increase,but%20much%20less%20than%2040%2C00.

"Tkinter - Python Interface to Tcl/Tk." *Tkinter - Python Interface to Tcl/Tk - Python 3.9.4 Documentation*, The Python Software Foundation, 14 Apr. 2021, docs.python.org/3/library/tkinter.html.