Groover, A Song Recommendation Web Application

CS 121: Software Development
Department of Computer Science
Harvey Mudd College
Spring 2019

Trevor Walker
Siena Guerrero
Jocelyn Chen
Walker Quinn
Teresa Ibarra

Contents

1 Introduction

Groover is a web application that allows users to find lyrically-similar song recommendations given an input song and the songs in our database. Users can input a song that they want recommendations for and the application will then display recommended songs to the user with supplemental information for each. The supplemental information may include the recommendation's song title, song artist, album art, genre, preview, and a link to the song on Spotify, if available.

Our project is currently live on the internet [1]. The code for our project is available on Teresa Ibarra's Github [2].

1.1 Background

In brainstorming how to design and build our web application, we consulted computer science research papers that focused on musical content and lyrical analysis to generate recommendations. We primarily explored two papers for this project to understand how we could build a model that will accomplish our goal of generating lyrically-similar song recommendations.

In Yang and Lee's paper "Music Emotion Identification from Lyrics" [3], the group used song lyrics to classify songs based on a "psychological model of emotion." They found in their research that unique words are helpful in classifying a song's mood. They took 1032 random song lyrics from allmusic.com and trained a model to classify the songs based on the 23 PANAS-X psychological emotions. They created a numerical representation for each song as a feature vector.

For our project, this style of approach would mean classifying a large corpus of songs by their mood and then recommending songs to the user based on the mood of the songs that the user had input. While we decided to not pursue this type of recommendation system, we did decide that it would be beneficial to have a large corpus of songs to perform training on, leading us to use Kaggle for our song dataset.

In Sanford and Skryzalin's paper "Deep Learning for Semantic Similarity," [4] the authors compared using two different structures of neural networks to compute the semantic similarity of two sentences (i.e., how similar is the content of the sentences). This concept of evaluating similarity formed the foundation of our project.

We utilized a model that, given an input song's lyrics, can generate a list of songs whose lyrics are the most similar to the input song's. These vectors are then used to find the corresponding songs stored in a static database.

By using lyric-based recommendations, we hope to be able to offer users recommendations that are similar in theme but not limited by musical genre. To accomplish this, we built a model that analyzes lyrical content and finds other songs with similar content, no matter how different they may be in rhythm or genre.

2 Functionality

2.1 Overview

Groover generates music recommendations intelligently by using a neural network to find songs in our database with the most similar lyrics to the input song. A user can enter a song's artist and title and within seconds get 20 different recommendations. These recommendations are guaranteed fresh, as our database contains many different genres like Country, Hip Hop, and even Christmas. Our web application displays the album covers, artists, names, and genres of our recommended songs. If available, the display will provide users with a 30 second preview clip of the song within the browser as well as a link that will direct them to the song on the Spotify web player.

2.2 Song Input

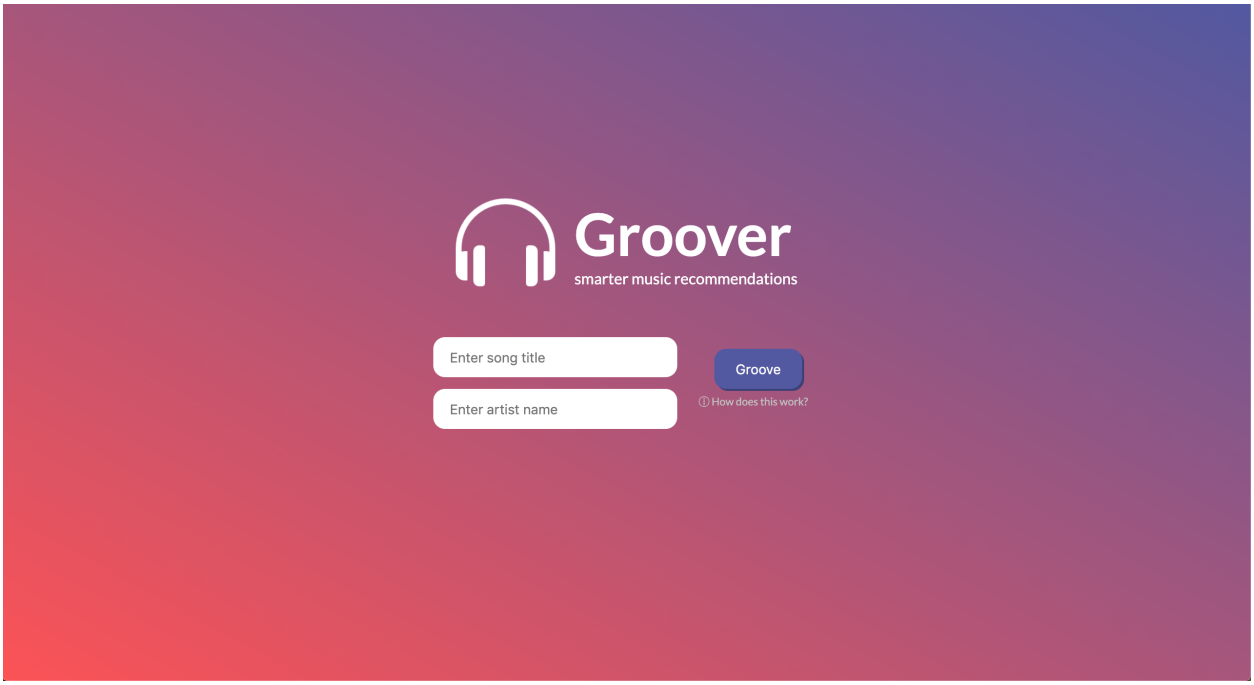

Figure 1: Landing page. Users can enter a song name and artist, then click "Groove" to generate recommendations.

Users can input the artist and title of the song that they want recommendations for. Then, they can click the "Groove" button (or hit the "Enter" key on the keyboard) which will toggle the generation of lyrically similar recommendations. Figure 1 shows the landing page, which contains the input boxes for song name and artist and the "Groove" button. Below the button, there is a hyperlink with the text "How does this work?," which, if clicked, links to a page which explains the song recommendation generations.

2.3 Recommendation Generation

After clicking the "Groove" button, users will then be able to view 20 of the top recommendations produced by our model. The application will display album covers of each recommendation. If an album cover is clicked, it will display genres, artists, music previews, and Spotify links for that song, if available. Clicking the arrows below the album covers will cycle through the 20 recommendations. If the user would like to generate recommendations for a new song, they can click the "Groove Again" button.

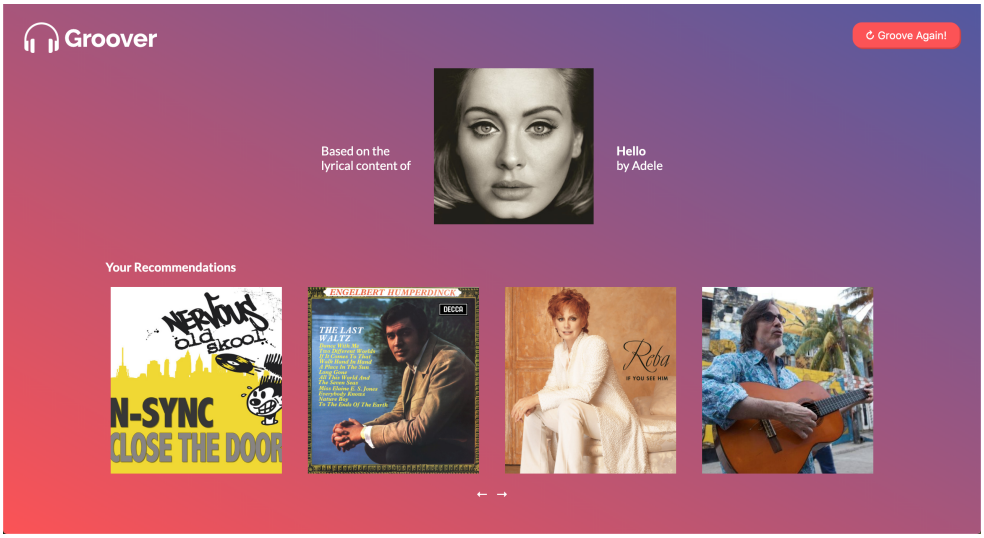

Figure 2: Recommendation Page. Users can browse the album covers of each song or click the "Groove Again" button to generate more recommendations.

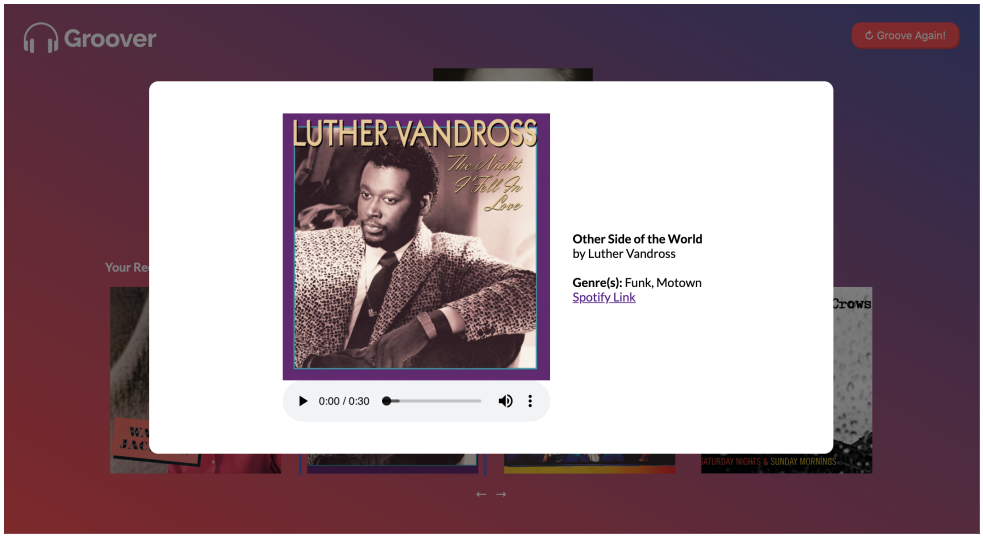

Figure 3: Recommendation Detail Overlay. After an album cover is clicked, users can read more information about the song, preview the song with a short clip, view the song's genres, and click a link to find the song on Spotify.

3 Architecture

3.1 Introduction

Our architecture uses Flask, a Python microframework, to communicate with the Spotify [6] and Musixmatch [5] APIs, our document similarity model, and our song database. Our document similarity model and static JSON database is loaded and used within a Python script. The web application is hosted on Heroku, which automatically deploys changes to our master branch to production. Each of the major components of our application will be discussed in further detail in their own section.

3.2 Model-View-Controller Diagram

The goal of a Model-View-Controller (MVC) diagram is to aid in the understanding of our application's structure. In our application, the model is our pre-trained document similarity model and the database. The view is our application's frontend. The controller is our Flask application. Figure 4 shows the relationships between each of these components.

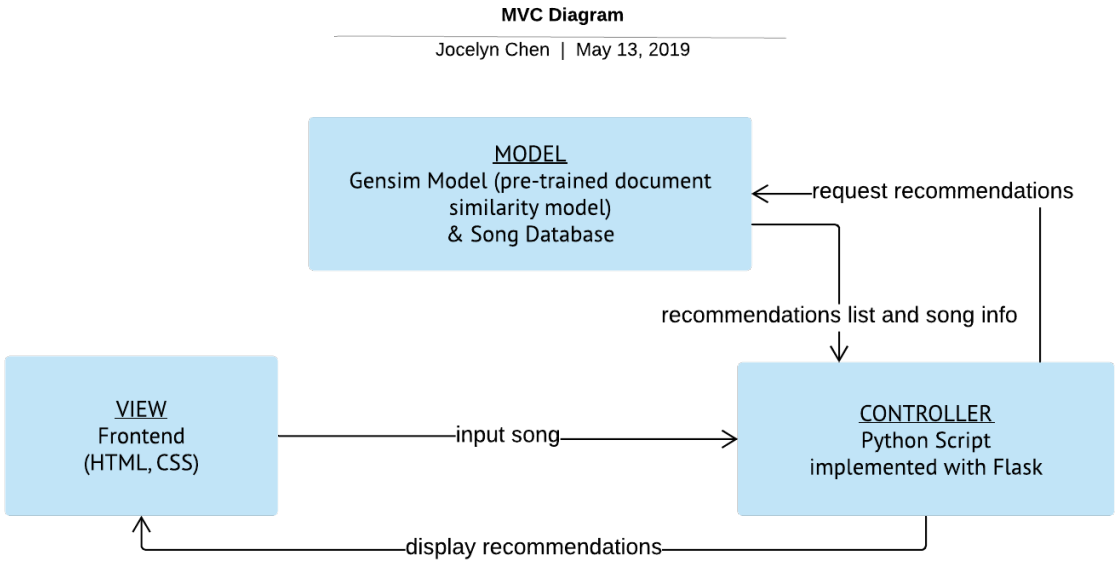

Figure 4: MVC Diagram. Displays the relationships between the structures of the web application.

3.3 Sequence Diagram

In this section, the sequence diagram will detail the flow between the view, controller, and model structures and how the application communicates with APIs, the model, and the database to generate and display recommendations for the user.

Figure 5 shows a UML Sequence Diagram that highlights the Model-View-Controller structure of our application. Our application receives the user's input and makes a call to the Musixmatch API, which returns the song's lyrics. The controller separately makes a call to the Spotify API, which retrieves additional information about the input song such as album art, song genre, Spotify link, and 30-second preview. The controller then loads the pre-trained document similarity model to retrieve similar songs by using the input song's lyrics. The output is given in the form of indices, which we then use to access our song database, or

list, of 57,000 songs. Information about the input song and the generated recommendations is then shown to the user.

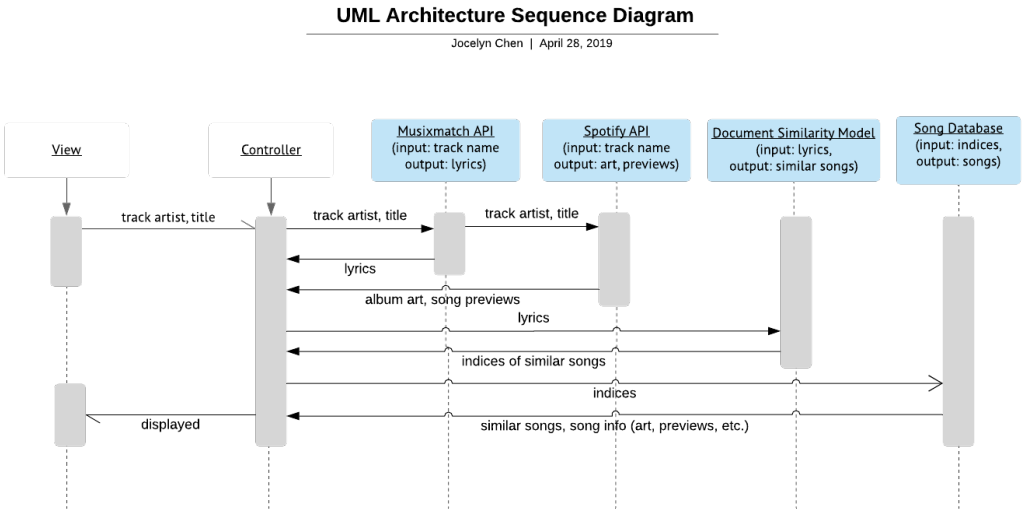

Figure 5: Architecture Diagram. Details the major components of the application and shows detailed processes between components. Diagram can be read from top to bottom, left to right.

3.4 Model

We found that the simplest and easiest to use solution for a document similarity model would be a Gensim [7] Doc2Vec model. Gensim is a popular Python library used for topic modelling and Doc2Vec is a class within Gensim. Doc2Vec modifies the Word2Vec algorithm to create vector representations for large blocks of text (or documents). In order to create a model that determines recommendations based on lyrical similarity, we first trained our Gensim model in Python with our lyrics database of 57,000 songs with 50 epochs. At run-time, we pass in the input song lyrics as a document to this model in order to generate recommendations.

The model infers the vector of the input song's lyrics, and then calculates the vectors that are the most similar. We then find the songs in our database that correspond with the most similar vectors, resulting in a list of songs with lyrics that are the most similar to our input song.

We have found that if we query a song that is already in our database, the same song is returned as the most similar result. In effect, this functions as a test that our model works on the most basic level, given that it makes sense that the same song would be lyrically similar to itself. Judging the quality of our recommendations beyond this point was difficult due to the subjective nature of song recommendations. However, we found the lyrical content of the song recommendations to always be similar to the content of input song.

3.5 Database

The major persistent data object that we store is a "database" of 57,000 songs with supplemental information for each song. We first selected a Kaggle dataset [8] that we previously used to train experimental models, for which we then removed all songs that were not found in Musixmatch's database. For every song, we made a call to the Spotify and Musixmatch API and retrieved the lyrics, mp3 preview URL, album cover, and genres. We stored this information as a list of dictionaries, in which each song has a dictionary of

information. We stored this data structure as a JSON, which we load into our Python code for generating recommendations when needed.

3.6 Hosting

Our web application is hosted on Heroku, a cloud platform. We have linked Heroku to our GitHub repository such that every time there is a change to the master branch of the Groover repository, a new version of our web application is automatically deployed.

4 Discussion

4.1 Database

In choosing our database format, we needed to be able to easily retrieve our database and find new song recommendations quickly. We also decided that we did not want to update our database constantly with new songs. Thus, we decided to store our database as a static JSON file, particularly as a list of dictionaries. This was easier to implement than using a data structure like Redis or Firebase's database capabilities.

4.2 Dataset

We recognize limitations to this dataset as it clearly contains a small portion of all the music that exists and it cannot automatically update with new music releases. Our decision to choose this dataset is based on the fact that it contains a relatively large sample compared with those others that are available and is also fairly current, in that it contains songs from 2018 and prior.

There is also no known database currently available that is able to update with new music. Although our generated recommendations are limited to songs in our dataset, our implementation of the Musixmatch API allows users to generate recommendations for almost any song.

4.3 Heroku

We found that it was easiest to deploy our application with Heroku instead of manually setting up with Amazon Web Services' EC2. We had spent several weeks attempting to set up our server on EC2, but we struggled to connect this with GitHub. Heroku can automatically deploy to our website once a change has pushed to our master branch on GitHub, which simplified our deployment process.

If we detect a bug in our application after a change to the master branch, Heroku allows us to roll-back to a previous build of our application so users will not encounter any issues when using Groover. We did not have this capability in EC2.

We are able to generate a DNS target name for our Heroku instance. We then have a web domain name point to it, which we purchased from Amazon Web Services. This was easier to set up in Heroku than in EC2. By using Heroku for hosting, we are able to store our API keys and environment variables securely.

This was an overall better solution to hosting Groover than setting up an EC2 instance to serve the application.

4.4 APIs

We decided to use Musixmatch over other APIs because it was the only API that had "fuzzy matching" of a song's title and artist. Musixmatch's database is also larger than any other API available, which gives us access to as many lyrics as possible and allows us to accommodate users wanting recommendations for almost any song, old or new.

We decided to use Spotify because it was the API that had the most supplemental information about songs with minimal API setup and authentication. We decided to add this API call in order to display more useful information to users and have better visuals for the recommendations page.

5 Limitations

5.1 Song Title Verification

In order to prevent the application from crashing or suffering from malicious intent, we needed to implement a basic method for sanitizing text. We decided to omit special characters that can be input, such as '%' or '&', when passing information to our Python script using our web form.[1] As such, our application has difficulty handling cases with input songs that contain a lot of special characters.

5.2 Lack of Testing

Due to time constraints, we were unable to set up unit testing for this project. To address this, we spent time manually testing our application to find possible bugs, leading us to develop a document for found bugs and their fix status.

6 Future Work

6.1 Multiple Streaming Platform Integration

Spotify is just one of the many music streaming platforms available. For our recommendations, we could offer a link to the track on Apple Music, TIDAL music, and others. This would be relatively simple to implement with Apple Music's API, but we would have to make at least 57,000 API calls for each song in our database, which has taken several hours to perform in the past. Other music platforms do not necessarily have an API.

6.2 Youtube Music

If a recommendation is not available on any music streaming platform, we could offer a link to the track on YouTube. This would be relatively simple to implement with YouTube's API, but we would again have to make at least 57,000 API calls for each song in our database, which has taken several hours to perform in the past.

6.3 Better Recommendations

We would like to make better recommendations based on the audio features of each input song. We found that users were puzzled by the wildly different genres of the recommendations given by our model. As a stretch goal, we hoped to factor in tempo, dance-ability, acousticness, and other audio features. However, because a large fraction of the songs in our dataset were not available through Spotify's API, implementing this would require a new database of songs, all of which are available on Spotify, or the use of a new API that gives information on all of our database's songs, which, to our knowledge, does not exist.

6.4 Spotify Playlist Generation

For this feature, users could choose to create a Spotify playlist containing their generated recommendations. The technical implementation of this would need user authentication to use the playlist creation feature of the Spotify API. But because our database is not fully on Spotify, we would be unable to create a playlist of all our generated recommendations. Implementing this would require a database fully available on Spotify.

[1] For example, "H?e!l&l=0" would change to "Hello"

7 Acknowledgements

We would like to thank Radon Rosborough and Ian Taylor for their help in setting up our web application. We would also like to thank our Writing Center consultant, Blake Larkin, for assisting in the write-up for this final report.

References

[1] Groover. 2019. Smarter music recommendations. http://www.groovermusic.net/

[2] GitHub. 2019. Groover GitHub Repository. github.com/teresaibarra/groover/.

[3] D. Yang and W. Lee. 2009. "Music Emotion Identification from Lyrics." http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5363083&isnumber=5362509/.

[4] A. Sanborn and J. Skryzalin. 2015. "Deep Learning for Semantic Similarity." https://cs224d.stanford.edu/reports/SanbornAdrian.pdf.

[5] Musixmatch. 2019. Song Lyrics and Translations. https://www.musixmatch.com/.

[6] Spotify. 2019. Digital Music Service. https://www.spotify.com/us/.

[7] Gensim. 2010. Software Framework for Topic Modelling with Large Corpora. https://radimrehurek.com/gensim/.

[8] Kaggle. 2017. "55000+ Song Lyrics" (Kaggle Database). https://www.kaggle.com/mousehead/songlyrics/.