****

**Spotify Playlist Ranking and Recommendation System**

CMPE 256 - Large Scale Analytics

Summer 2019

**Project Group# 10**

Zaid Laffta <[zaid.laffta@sjsu.edu](mailto:zaid.laffta@sjsu.edu)> - 009689433

Mario Yepez <[mario.yepez@sjsu.edu](mailto:mario.yepez@sjsu.edu)> - 012556050

Kunj Parikh <[kunj.parikh@sjsu.edu](mailto:kunj.parikh@sjsu.edu)> - 012532065

Srinivasa Prasad Sunnapu <[srinivasaprasad.sunnapu@sjsu.edu](mailto:srinivasaprasad.sunnapu@sjsu.edu)> - 012526241

Miguel Covarrubias <[miguel.covarrubias@sjsu.edu](mailto:miguel.covarrubias@sjsu.edu)> - 008809892

**Table of Contents**

[**Abstract**](#_af2nya4rnwow) **3**

[**Introduction**](#_7sjys3ramps5) **4**

[**System Overview**](#_9yut6q1j1a97) **5**

[Spotify Recommendation and Ranking System](#_5gyd3892p729) 5

[Challenges](#_ybnd2e7ns3a7) 6

[**Data Acquisition and Preprocessing**](#_u71odeufje3d) **7**

[Data Collection](#_uau34ttok2nz) 7

[Data Processing](#_ji1sc6c25kev) 8

[**Implementation**](#_6ty9u11xv20b) **10**

[Application Workflow](#_5gsickh9en1m) 10

[Modeling Data and Feature Engineering](#_tf15jfjr1a6x) 11

[Model-1 Playlist Based Cosine Similarities](#_1lgnmnczj7rm) 12

[Model-2 Nearest Neighbor Based Similarities (Euclidean)](#_jtc2og4ixhrj) 13

[**Deployment**](#_iwj6smhwfe6z) **14**

[**Conclusions**](#_p8mq0edu9ta6) **18**

[**References**](#_st1einhuw1qm) **19**

[**Appendix**](#_hh4hgrt1k37g) **20**

**Table of Images**

Figure 1: Recommendation results with ranking 6

Figure 2: Simple Application Architecture 10

Figure 3: Recommendation Engine and Application flow 12

Figure 4: Cosine similarity between playlist vectors 12

Figure 5: Cosine Recommendation Output 12

Figure 6: Distribution and Grouping of Neighbors 13

Figure 7: Nearest Neighbor Recommendation Output 14

Figure 8: Architecture of Cloud Deployment 15

Figure 9: Landing Page 16

Figure 10: Recommendation Page 17

**Table of Tables**

Table 1: Acoustic features of tracks 9

Table 2: Sample Playlist data before processing (Viva Latino!) 11

Table 3: Sample Playlist data before processing (Viva Latino!) 12

# Abstract

The recommendation systems are everywhere, helping organizations to provide better value to its users by offering content that best represents users interest. When a user login into Spotify, Amazon, or Netflix, content in the home page is generated by complex recommendation systems specific to that user based on previous interests and usage history. The intention is to engage users with software applications / platforms which often improve customer subscription rate, revenue generated through ads, order conversation rate etc. In other words, recommendation systems are revenue generators for a modern day applications. Moreover, The rise of online streaming servers whether it is audio or video streaming exploded in the last decade or so. This rise of online streaming in general has dramatically altered the media habits of Americans, especially young adults. This trend in the industry and the high number of available options made it hard to pick something that will work with your taste. For example, in Spotify there are more than 2 billion playlists which makes it really hard to know what will suit user taste the most. This project addresses this problem through recommendation system that can recommend playlists and rank by match score to user preferred playlist.

# Introduction

Taking the challenge of building recommendation system that can recommend playlists and rank them was something of interest from day one. Our work in this project is inspired by noticing that there isn’t a recommendation system for playlists in Spotify which makes it different compared to its competitors like Apple, Amazon, Pandora etc. Spotify’ recommendation system isn’t based on a single model / algorithms, instead it’s a combination of different recommendation models based on collaborative filtering, natural language processing (NLP), audio data analysis etc. Not only it does a good job in recommending songs as per users interest, but also scale really well across 140 million active users, 30 million tracks and 2 billion playlists. There are different machine learning and deep learning algorithms also used at Spotify for different use cases. For example, Spotify’s default home screen is powered by Multi-armed bandit algorithm to surface music and podcasts for every situation, personalized playlists, new releases, old favorites, and undiscovered gems.

This project focuses on recommending top playlists matching a given playlist, then those playlists will be ranked. The similarity between the playlists is determined by the kind of songs it is composed of. A playlist composed of more songs similar in comparison to input playlist will be ranked higher over the other playlists. The similarity between songs can be determined by features / attributes / tags associated with the song. Fortunately Spotify offers easy to use APIs to extract such features of songs included in a playlist. However, Spotify doesn’t share list of playlist available in its platform. This required for alternative ways to collect playlists, like scraping data from social network platforms like Twitter, Reddit etc. More details about dataset preparation, implementation and findings are covered in covered in further sections.

This application is made available as an easy to use website for user evaluation. Refer to appendix for more details. The website is hosted in AWS to help out with security, scalability, and ease of deployment and modification. This ranking application made available to be used by anyone without the need to register or pay any type of fees. Finally, the feedback system was designed and put in place to help get feedback from users based on their experience using the website. This feedback will be used to improve the recommendation engine and increase user satisfaction.

# System Overview

## Spotify Recommendation and Ranking System

The main target behind this project is to have a user friendly GUI supported by a strong recommendation system that will help the users to find playlists from Spotify that would match their taste in music. Also, ranking those lists from top match to lowest match will show option to the end user and make the process more interactive. The user will need to copy a URL for a playlist that he likes and the system will show him a list of ten different playlist that matches the list entered in a ranked fashion. Figure 1 shows the result.

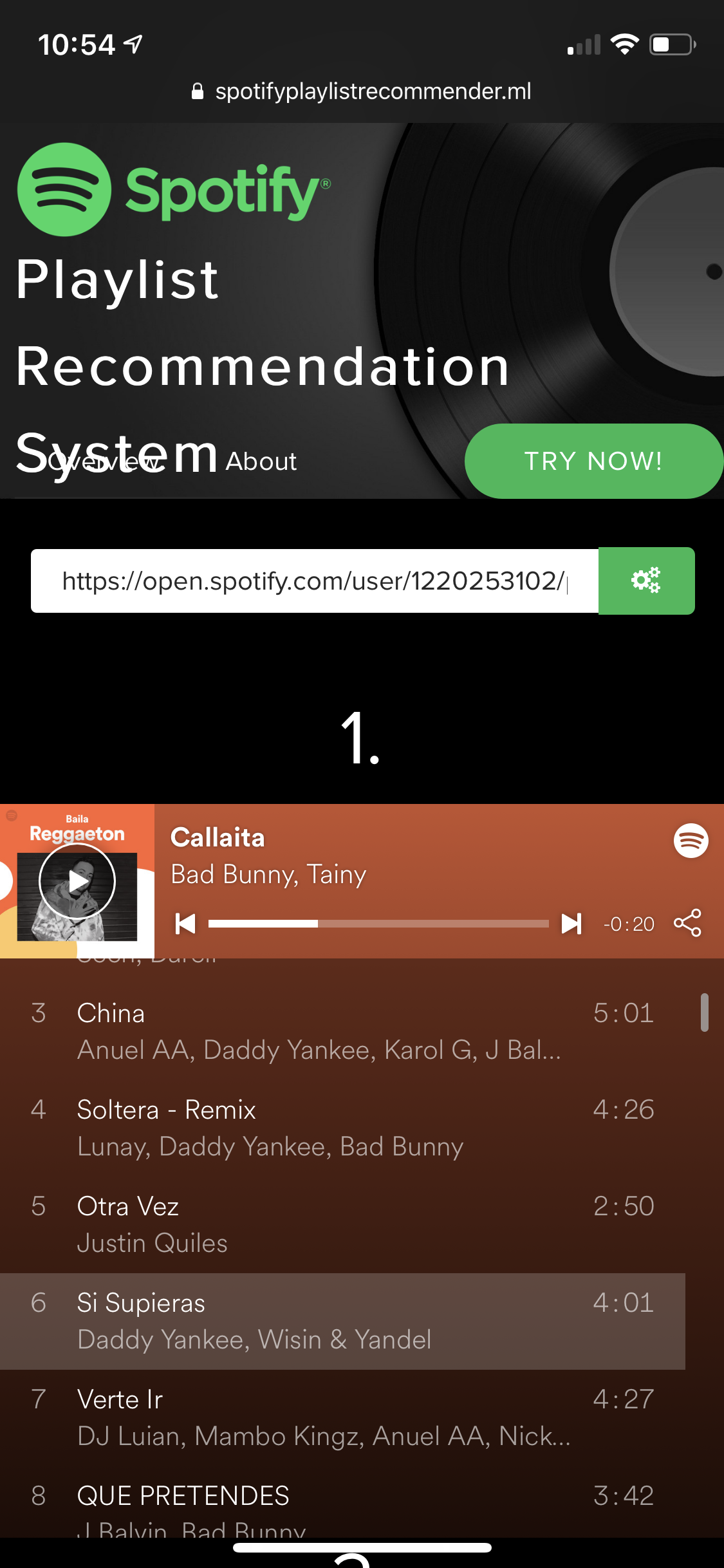


Figure 1: Recommendation results with ranking

Just like any recommendation system projects, we had some challenges like data collection, modeling and others which will be touched upon later. Moreover, this project was executed in phases which helped to organize the workflow. The first phase was started off with finding the right data collection, modeling, execution, testing, and optimizing.

## Challenges

Most of the times recommender System project has a lot of challenges and stop at achieving some satisfactory accuracy depending on the model, data set and other factors. This is the case with this project also. After modeling the system and we start running different types of testing in a controlled environment and with a limited dataset yielded very impressive results but deploying an algorithm into real world and scaling wasn’t as expected. The above example shows one of the challenges that we faced. Other challenges was observed can be listed below:

1. Cold start problem - some songs and playlists doesn’t have rating because they are new in the system
2. Feature extraction - some songs doesn't have certain features like danceability, instruments, and loudness just to name a few
3. Size of the dataset - the more playlist we have the longer it takes to process them
4. Scaling the application which will result in more traffic and eventually more server resources will be required
5. Security where we will need a firewall in the VPC to protect it from different types of attacks
6. Synonymy - where something songs has the same name
7. Not accurate rating for the evaluation system that may affect the overall results

# Data Acquisition and Preprocessing

## Data Collection

The very first step in every recommender system project is collecting data and in most cases it is the hardest step. Data can make or break the recommendation system. In this project case, data collection was challenging due to the nature of the data we want. Typically in most of the cases we would ask people about their opinion regarding some types of product. In our case we ask people about their Spotify playlist, which makes people a bit hesitant to give it to us.In this project we got out of the basic definition of data which is a large and complex information like such as rating or financial history. In this project our data set is playlist. And any mistake in the playlist url will cause a problem where the playlist will not be considered and eventually it will affect the overall system accuracy. Finally, most of Spotify playlist links belong to Spotify, so any use of this data that can modify it will have legal implications. .

We follow multiply ways to collect data, it can be listed as the following,

1- Live survey at school and other people we new like social media friends

2- Survey Monkey to get date by using social media and group email every present in my company, friends and family

Below is the link for the paid survey I sent out on Survey Monkey.

https://www.surveymonkey.com/r/CY7DL9D

The result where great especially from the live survey, but not so much from the online survey. In total I got 45 people to take my online survey and we got 39 valid playlist. On the other hand, 223 to take in person survey between school.

Next we looked for playlists available publicly. We found that people usually post their favorite playlists, or what they are listening to on Twitter, Reddit, other platforms. So we took a look at REST APIs for Twitter and Reddit. We used them to scrape posts on Twitter and Reddit for strings like “[https://open.spotify.com/playlist](https://open.spotify.com/playlist/0q3bHAqkQrarbaCnkjDQR7)”. We collected more than 11k playlists from Reddit and even more from Twitter.

After collecting data, we started the process of cleaning data and make sure all playlists are valid. Finally, we merged the data into a single data set to have it ready to process.

## Data Processing

This phase can be summarized as the following:

1. Playlist validation: making sure that the playlist is valid and still accessible in Spotify
2. Adding the playlist to our dataset
3. We used Spotify’s REST API to extract features for each track within each playlist. The track’s acoustic features were used to compute two playlists’s similarity. Feature details are below. Their usage is described in detail in the following sections.

Table 1: Acoustic features of tracks

|  |  |  |
| --- | --- | --- |
| duration\_ms | int | Duration of tracks in ms |
| key | int | Overall track : Pitch -> Int mapping eg. C# = 1 |
| mode | 0,1 | Modality : Major or minor |
| time\_signature | int | Meter of track : #beats in each bar |
| acousticness | [0.0,1.0] | Confidence measure whether track is acoustic |
| danceability | [0.0,1.0] | Combination of tempo, rhythm stability, beat strength, overall regularity |
| energy | [0.0,1.0] | Perceptual measure of intensity and activity |
| instrumentalness | [0.0,1.0] | Predicts whether track contains vocals or not. |

We also collected others features like Liveliness (audience), loudness (dB), speechiness. valence (positiveness), tempo(bpm). Spotify provides other analysis features like - pitches, timbre, bars, beats, tactum - at the start, overall, end of the track, however it would have exploded our dataset size so we didn’t consider them.

The collected data is broken us into compressed json.zip files. Each file containing 1000 playlists, its tracks and track features. JSON provided us with easy read-in method in the form of python dicts.

# Implementation

The Spotify recommendation application focuses on recommending playlists based on a given user playlist. The idea is to calculate the similarity between user playlist features and the features from the playlists in our dataset. In this section we will cover the application workflow, feature selection and the different model implementations.

## Application Workflow

The application starts by loading the data from the file system for processing. Once the data processed the extracted data will be loaded into memory as a pandas dataframe. This dataframe will later be instantiated by the models to create recommendations. Loading data into memory could be an issue if not enough memory is available in the server. Now, when users submit a playlist to find recommendation the API call payload will contain both the preferred model they would like to use and the name of the playlist. Once the server side receives the payload it will then issue a call to the Spotify API for information about the input playlist features. Then the features will be processed and they will be used as one of the inputs for the model. Once the model finishes it will return back to the user N recommended playlists. Figure 2 shows a simple architecture of the application.

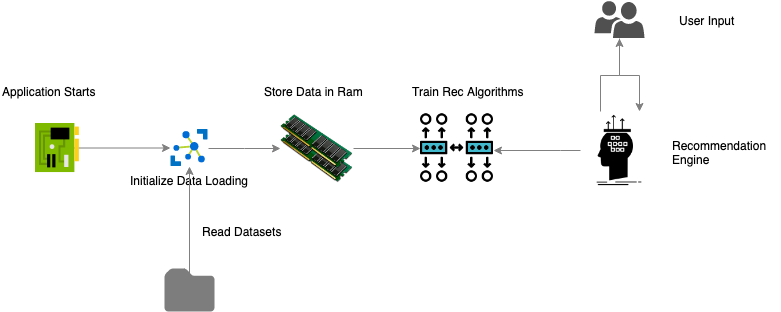


Figure 2: Simple Application Architecture

## Modeling Data and Feature Engineering

When the data is loaded from the file system it must be processed. Every playlist has feature information for each of the songs in the playlist which means that all the songs features must be aggregate in such a way where each playlist has one vector of features. Table 2 shows the data for a playlist before processing which includes the features for each of the songs in a given playlist. This data format is not ready to be used in the modeling. Table 3 shows the data after processing which contains only one row record per playlist. The way the data was processed was by averaging the features in the songs for each of the playlist. Taking the average may not be the best way to do this but it gives enough meaningful results. Later we will use another approach instead of averaging the feature’s data for each of the playlists. At this point the data is ready to be used by the models which will calculate the similarities. We did not see differences in results if the data was normalized and having non normalized data allowed us to append the user given playlist record to the model dataset. Furthermore, the features shown in tables 2.2 and 2.3 were picked as the best features to represent a songs since they had ranges of values instead of binary numbers.

Table 2: Sample Playlist data before processing (Viva Latino!)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| track name | danceability | energy | loudness | speechiness | acousticness | liveness | valence | tempo |
| Callaita | 0.62 | 0.624 | -4.773 | 0.309 | 0.6 | 0.243 | 0.244 | 176.169 |
| Soltera Remix | 0.783 | 0.783 | -2.2471 | 0.04332 | 0.361 | 0.43 | 0.8 | 92.016 |

Table 3: Sample Playlist data before processing (Viva Latino!)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| playlist id | danceability | energy | loudness | speechiness | acousticness | liveness | valence | tempo |
| 37i9dQ | 0.23 | 0.934 | -34.564 | 0.353 | 0.7 | 0.943 | 0.234 | 160.541 |
| 2t2aNg | 0.783 | 0.783 | -33.928 | 0.0647 | 0.425 | 0.53 | 0.9 | 100.943 |

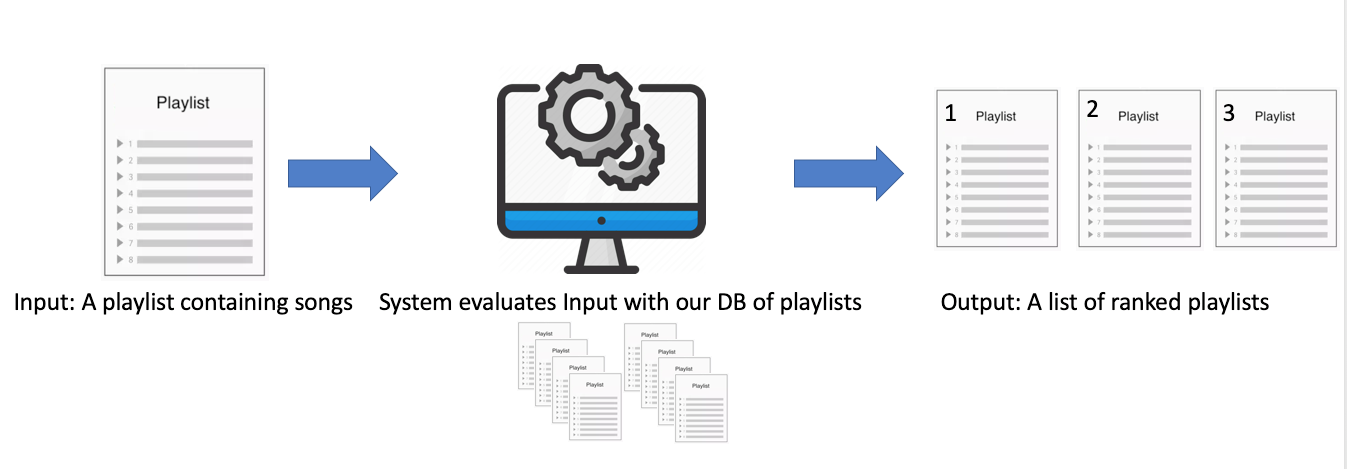


Figure 3: Recommendation Engine and Application flow

### Model-1 Playlist Based Cosine Similarities

In this model we focused on using cosine similarity between the input playlist given by the user and the playlist datasets stored in memory. The cosine similarity measures the similarity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction.

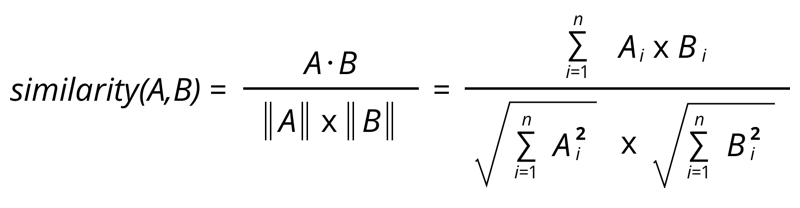


Figure 4: Cosine similarity between playlist vectors

**Result Example 1:**

User input playlist with features:

*features = {playlist\_uri, danceability, energy, loudness, speechiness, acousticness, liveness, valence, tempo}*

*user\_input\_vector = {0, 37i9dQZF1DX10zKzsJ2jva, 0.73936, 0.69842, -4.51284, 0.146688, 0.261872, 0.17367, 0.62518, 124.31074}*

*Recommendation Output:*

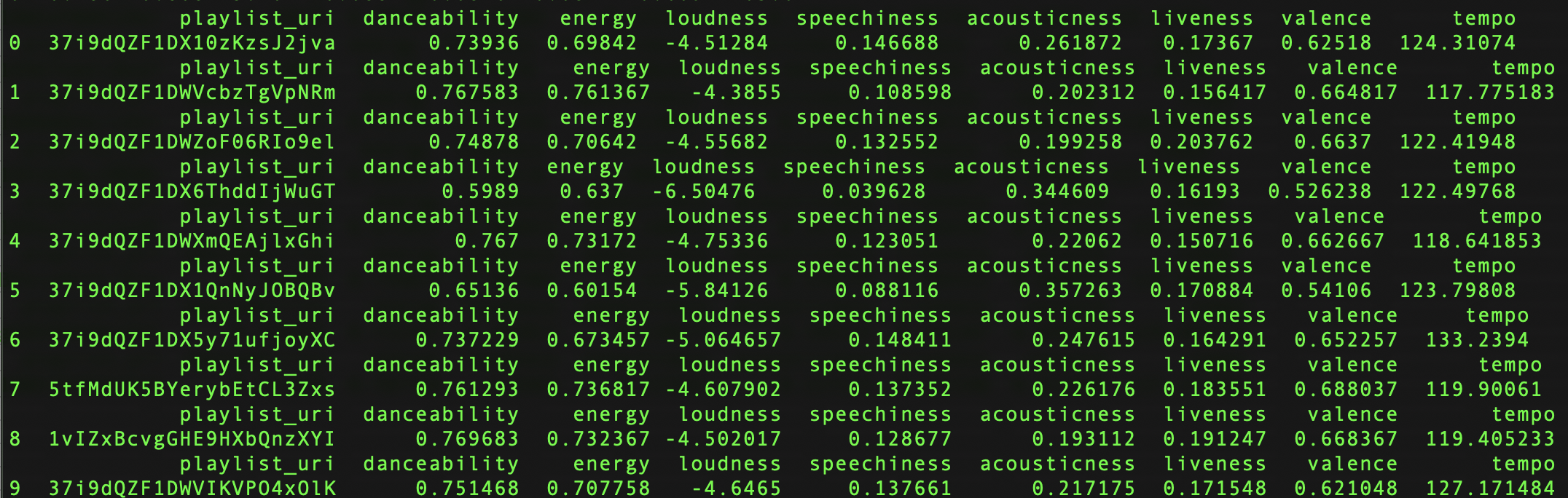
**

Figure 5: Cosine Recommendation Output

In the example above notice that the input playlist is already part of the model dataset this it is the first playlist that the recommendation engine suggests. This means that the cosine distance between the user input playlist and the top recommended playlist are very close together (they are the same. We can restrict our model from recommending the same playlist the user inputted but that is not too important because this allows us to make sure the model is working properly.

### Model-2 Nearest Neighbor Based Similarities (Euclidean)Figure 6: Distribution and Grouping of Neighbors

In this model we focused on using Nearest Neighbor algorithm between the input playlist given by the user and the playlist datasets stored in memory. The K Nearest Neighbour is a simple algorithm that stores all the available cases and classifies the new data or case based on a similarity measure. It is mostly used to classify a data point based on how its neighbours are classified. In our case we used the euclidean distance metric that can also be cosine and we used euclidean to explore how this metric can perform.

Figure 6: Distribution and Grouping of Neighbors

As seen in the example in Figure 7, the Nearest Neighbor algorithm performed well using the euclidean distance although not as good as the cosine similarity. We made multiple testings and changed the hyper parameters of the NN function to increase the recommendation accuracy. Highlighted in blue is the same playlist as the user inputted and it is not the number 1 recommendation as expected because the playlist which is queried live from Spotify was recently updated (new songs were added) and the features changes slightly. This means that the recommendations is still good enough even when features in the playlist changes.

**Result Example 1:**

User input playlist with features:

*features = {playlist\_uri, danceability, energy, loudness, speechiness, acousticness, liveness, valence, tempo}*

*user\_input\_vector = {37i9dQZF1DWVcbzTgVpNRm, 0.737183, 0.801633, -5.463283, 0.095895, 0.191847, 0.183595, 0.788333, 118.398683}*

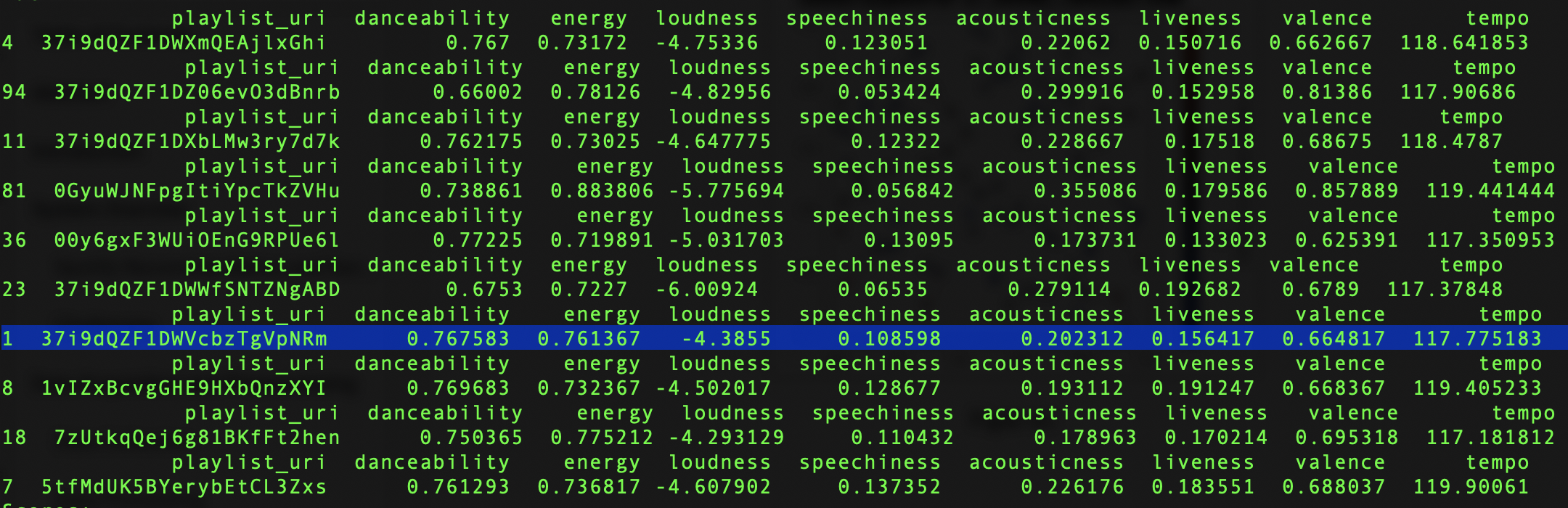


Figure 7: Nearest Neighbor Recommendation Output

# Deployment

Once we had a working model that we’ve trained and tested the next step was deploying it so that others can access it. Up until this point we’ve gathered our data, processed it, and implemented our model in Python and to make integration easy we decided to use Python once more for the Web-Application. For this we opted to use the Flask framework to build our Web-Application and deployed it on an Elastic Cloud Compute (EC2) Instance on Amazon Web Services. Using an EC2 instance allowed to easily allocate more resources as needed for the Web-Application on the fly. At one point we needed to upgrade our instance to something more powerful in order to effectively run the recommendation algorithm.

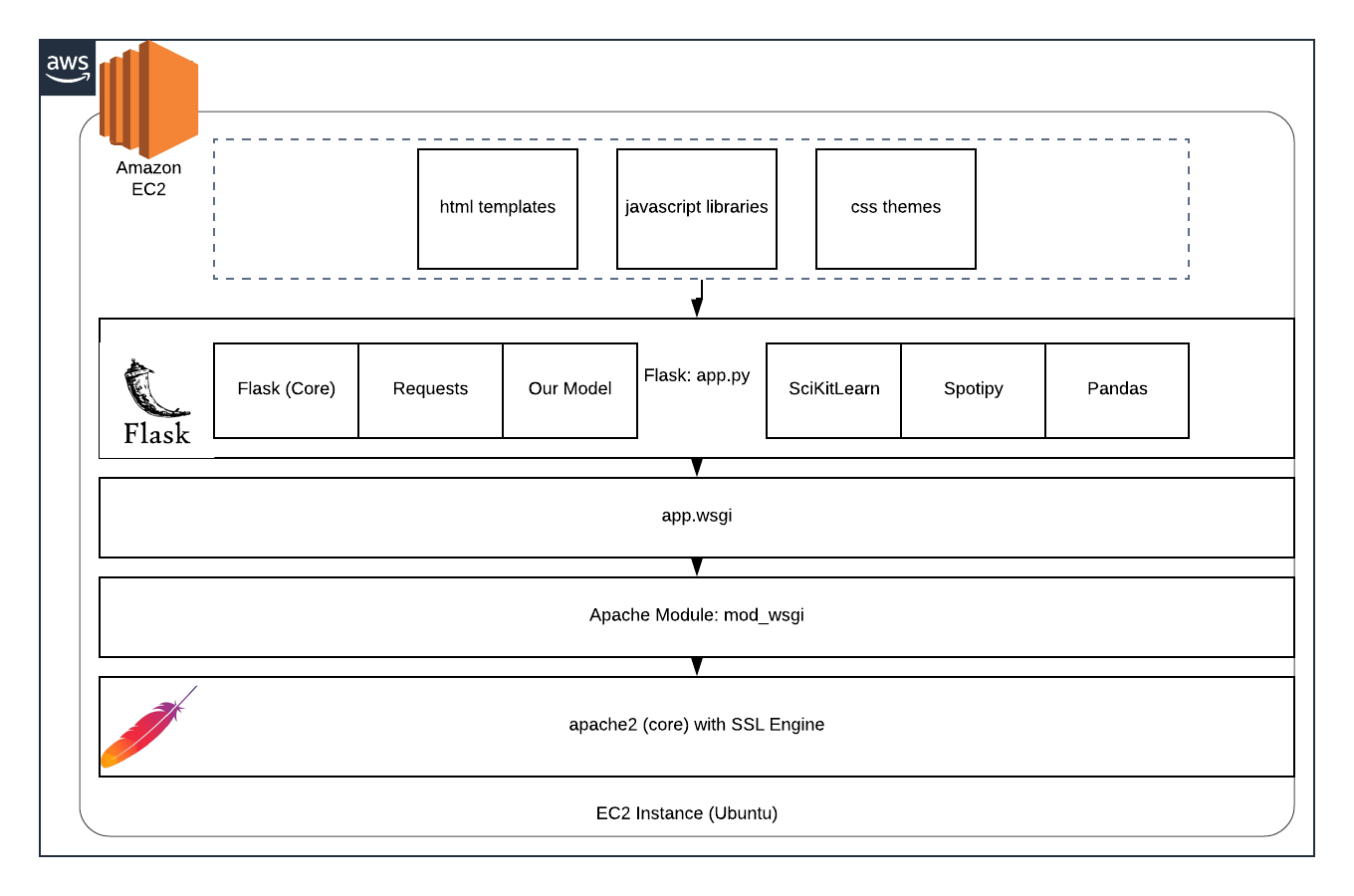


Figure 8: Architecture of Cloud Deployment

As far as how our Web-Application was developed and ran we decided to keep things relatively lightweight and simple. Our Web-Application runs using Apache2 for the webserver. We’re using the two Apache Modules ‘ssl’ and ‘mod\_wsgi’. Using the ‘ssl’ module we are able to serve https connections. This was accomplished by acquiring a domain for our Web-Application and getting an SSL certificate from Let’s Encrypt. The ‘mod\_wsgi’ module is what enabled Apache2 to run our Python Flask application on Apache. The ‘mod\_wsgi’ module executes the code in ‘app.wsgi’ that then runs the code from ‘app.py’ (the Python Flask project). For the Front-End we used jQuery and Bootstrap 4 to create a responsive webapp that conforms to the size of the screen and utilized Ajax to create a dynamic page that displays the recommendations. We send an Ajax request to the Web Server with the parameters for the recommendation model. The parameters include the Input Playlist URL and the method (either knn or cosine).

Our website can be reached at the following link <https://www.spotifyplaylistrecommender.ml/>.

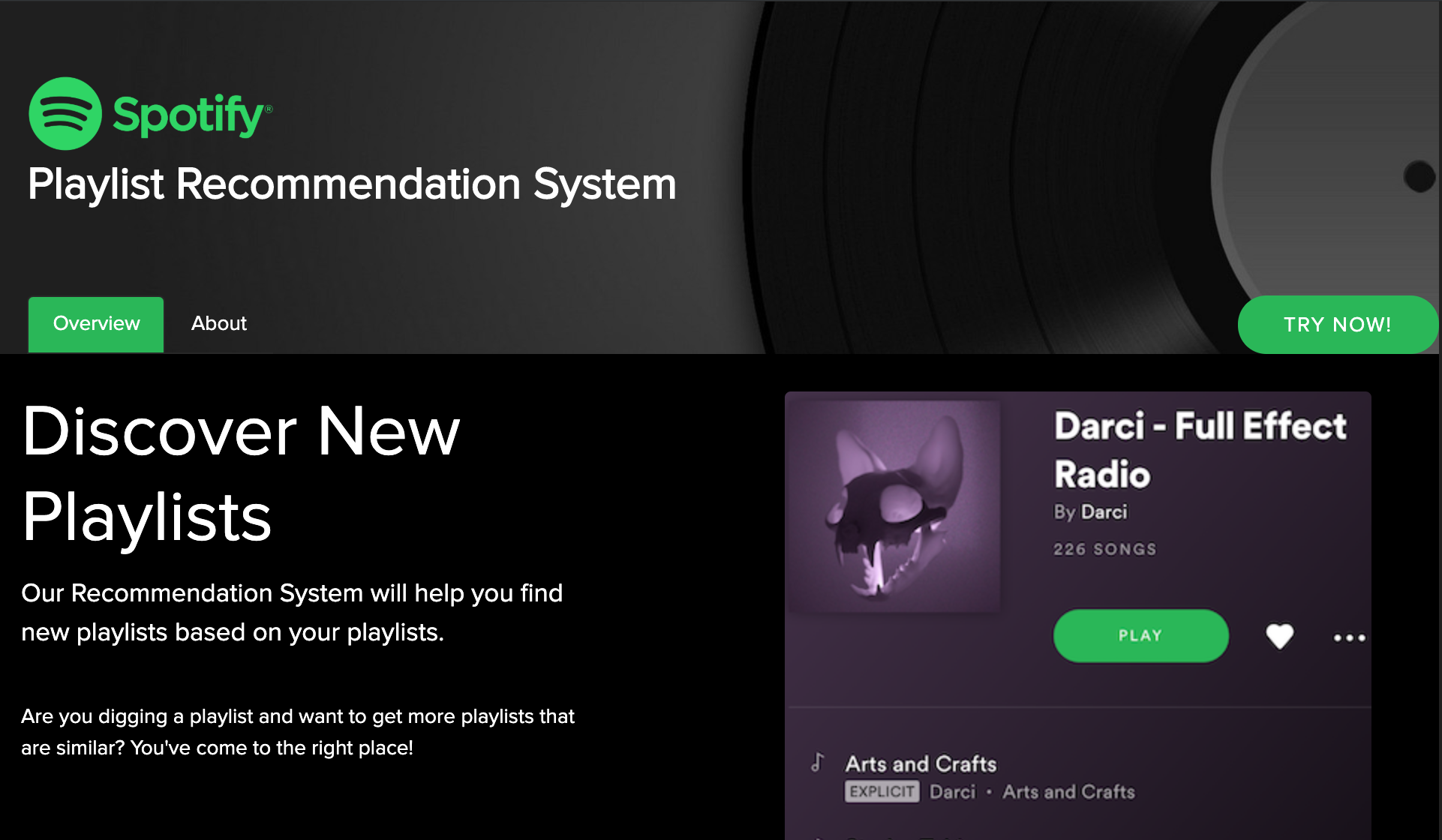


Figure 9: Landing Page

On the landing page for our webpage we briefly explain how our recommendation system works in terms of inputs and outputs. At first glance, one of the things you’ll notice is how we stylized the website’s fonts and color schemes to match that of Spotify’s User Interface. The design decision was made so that users would have a sense of familiarity between our Web-Application and the actual Spotify application. Spotify provides branding guidelines for developers to make sure that we respect Spotify’s brand and legal restrictions. The guidelines include assets for the logo and the color palettes that we then used on our website.

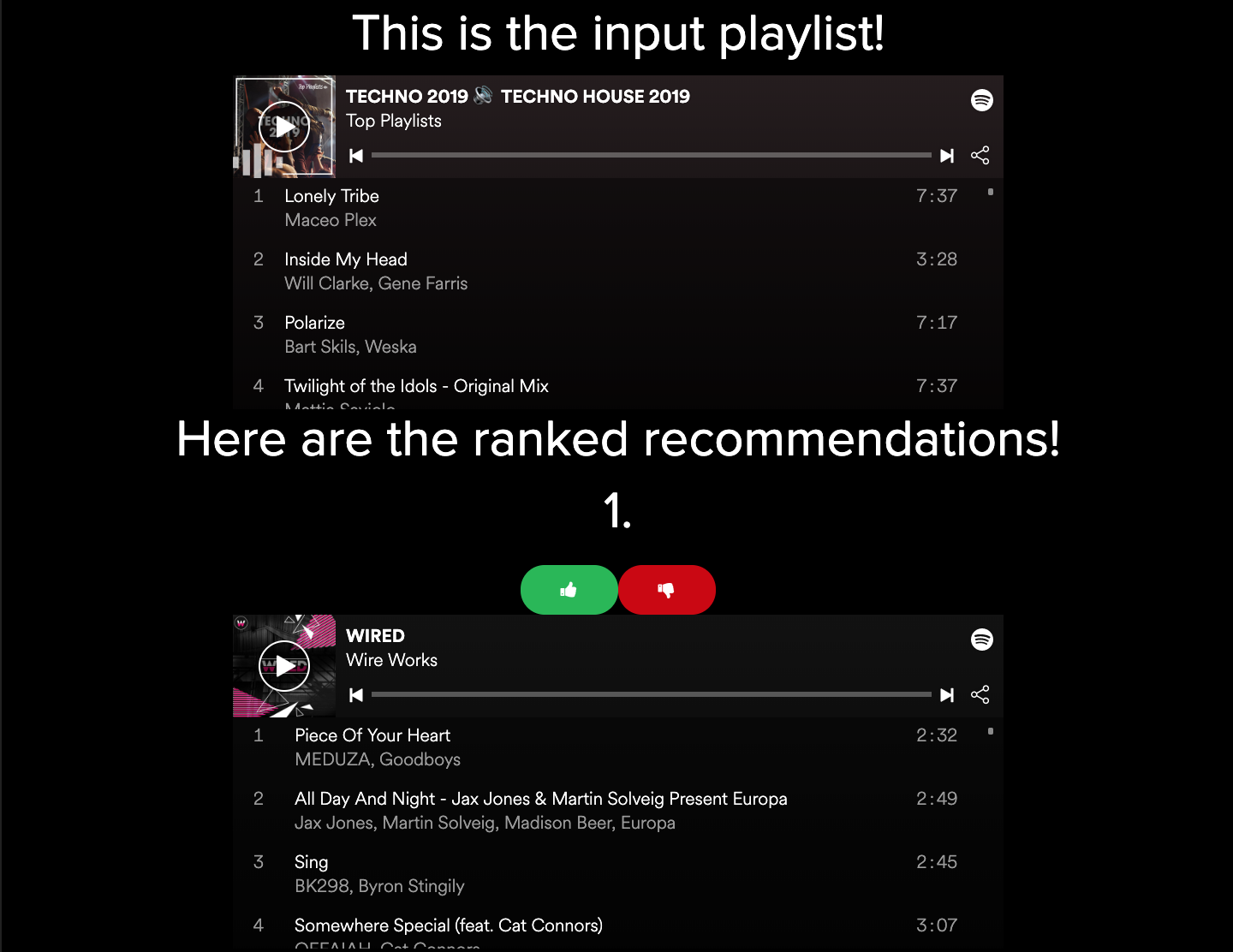


Figure 10: Recommendation Page

On the Recommendation page users can provide us with the input, a Spotify playlist url, that we then process and provide recommendations for. On the Web-Application we integrated Spotify Playlist Widgets to allow users to listen to the recommendations directly on the page without the need to navigate to another webpage or even have a Spotify account. The Playlist Widgets allow users to go through the individual tracks on the playlists and listen to a 30 second sample. If the user likes the playlist they can then click on the white Spotify logo on the top right corner of the Playlist Widget and it’ll open the playlist on the actual Spotify application so that the user can then add the playlist to their accounts.

We also added a feedback mechanism so that users can give us either a thumbs up or thumbs down depending on whether or not the user liked the recommendation or not. The feedback isn’t being directly used in the model, but it does allow us to gauge how our system is doing and gain some insights. We capture the rating and then store the result in a CSV file that we can do our own analysis on. The format for the csv is as follows.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date + Time** | **Input Playlist** | **Recommended Playlist** | **Score** | **Method** |
| dd-mm-yyyy-hh-mm-ss | <playlist url> | <playlist url> | 1 for like  -1 for dislike | knn or cosine depending on the method used |

# 

# Conclusions

Through this project we were able to apply key learning from this course, like data collection, cleansing, analysis, use of various models available to build recommendation systems for different kinds of use cases, and measure their performance through various evaluation methods. In spite of non existence of prepared datasets, we came up with an innovative way to collect playlist data by scraping social media platforms. Leveraged Spotify APIs to retrieve playlist characteristics, which helped us to build the playlist recommendation and ranking system that yielded more than satisfactory results for the dataset in use. This application is made available as website with easy to use interface for which we have received good feedback from test users. We have also provided feedback gathering system for recommendations which could help to build hybrid recommendation system as users engagement with platform grows.

# 

# References

* Reddit REST API : <https://praw.readthedocs.io/en/latest/index.html>
* Spotify REST API : <https://github.com/plamere/spotipy>
* Twython Twitter Rest API Library: <https://twython.readthedocs.io/en/latest/>
* Flask: <https://www.fullstackpython.com/flask.html>

# Appendix

* Project Github Link: <https://github.com/miguelcovarrubias/cmpe-256-group-project>
* Website Link: <https://www.spotifyplaylistrecommender.ml/>