Juan Carlos Gallegos — V00816131 — George Tzanetakis, Virtual Assistants, University of Victoria

CSC 490 Final Report

Project Introduction

Background

Currently, music streaming services like Spotify offer rich recommendation functionalities for finding new music based on general attributes and similarities to other music. Typically, recommendations are packaged into playlists and based on one of the following: a user's listening history (e.g. Spotify's *Discover Weekly*); a single song (e.g. Spotify's *song radio*); an artist, genre, activity, mood, etc (e.g. Spotify's *artist radio*). In general, the playlists are either curated or generated on a seed of the user's choice; there is a vast collection of playlists that represent an array of artists, genres, moods, time periods, and more.

These music recommendation interfaces are generally an effective way to offer a collection of songs that are similar in some opaque way; however, users cannot fine-tune the characteristics of the recommended music. Towards this end, Spotify recently added two buttons to their user interface for users to give feedback about the current song in a generated playlist; while this gives users slightly more influence on the details of a playlist's contents, it offers no direct access to details such as tempo, genre, popularity, or length. And so, an app capable of providing fine-grained recommendations would offer a novel way to explore and discover music.

Motivation

This project aims to develop an application for serving fine-grained music recommendations with the user's active participation. This application strives to allows users to continuously influence the provided recommendations by giving them explicit control of attributes such as tempo, danceability, happiness, and acousticness, to name a few. This approach contrasts with the mainstream approach of serving curated playlists or generating new ones based solely on a seed.

This report aims to describe the development process of this particular application to guide and motivate others in similar endeavours. We discuss the app architecture and general implementation details to highlight the challenges that arose and how they were faced.

Overview of Application

Use Cases

This project aims to satisfy one main use case: a user asks for a fine-grained recommendation in plain spoken English and the interface finds one and plays it automatically. For example, the user might say "Play a song like *thank u, next* but sadder"; in response, the system might find and play the song *needy* also by Ariana Grande. The system should be able to accomplish this for many songs and

for various attributes including **happiness**, **acousticness**, **danciness**, and **popularity**. Once the system implements song retrieval based on these various attributes, supporting more attributes would be relatively simple.

Requirements

From this preceding description, we can determine some key requirements:

1. The app interface must be capable of (a) recording the user's voice and (b) automatically playing music.
2. The system must have (a) access to a (b) music library with a (c) comprehensive set of song attributes (e.g. acousticness, happiness, etc.) and (d) information regarding similarity between artists and songs.
3. The system must be able to (a) processing the user's request and (b) match it to corresponding application logic.
    - This includes identifying and *extracting the parameters* — song name (e.g. *thank u, next*) and metadata parameter (e.g. sadness, tempo, acousticness, danceability, etc.) — that must be passed to the application logic.
4. At the end of the project, there must be a minimal but functional prototype that can be used as a proof-of-concept and to assess the potential value of a more complete version of the app.

In the face of (req. 4), I tried to rely on existing solutions wherever possible. This way, I could focus on developing end-to-end functionality, instead of spending time building out the underlying infrastructure. This bias explains why I relied heavily on existing services like Spotify and Dialogflow.

Requirement (req. 1a) allows for a range of platforms: desktop apps, mobile apps, and web apps all support audio recording. However, the other requirements are more restrictive. Streaming services offer the most practical way to achieve (req. 1b). In addition to APIs for retrieving music metadata, Spotify provides a Web Playback SDK that supports music streaming for their subscribers. Spotify also provides embeddable widgets, but they require the user to click play. I chose to use the Web Playback SDK so I could play songs in the background without being tied down to Spotify's widget.s

As part of a separate project, I developed a knowledge representation (KR) system capable of storing music metadata, relationships between entities (e.g. songs, artists, genres), and serving the info through a Python API — this component sufficed for (req. 2b) and (req. 2d). To satisfy (req. 2c), I adapted this KR system to store audio feature attributes (e.g. acousticness, instrumentalness, danceability, etc.). To satisfy (req. 2a), I developed a REST API in front of the existing KR system.

In the domain of Natural Language Understanding (NLU), there are various existing solutions: Google has Dialogflow and Google Assistant, Amazon has Alexa, Microsoft has LUIS/Cortana, Apple has Siri, and others smaller companies have their own (e.g. Wit AI). Of course, the chosen service must be flexible enough to tackle the target use cases.

Unfortunately, a lot of the existing NLU services do not integrate with music streaming services out of the box, which is important for (req. 1b). However, music streaming is certainly achievable in a web interface. Thus, I investigated NLU service capable of integrating with a web-based interface. Google's Dialogflow can naturally hook into a web interface through its Detect Intent REST API: sending queries and receiving responses is simply a matter of sending an HTTP request from the JavaScript client.

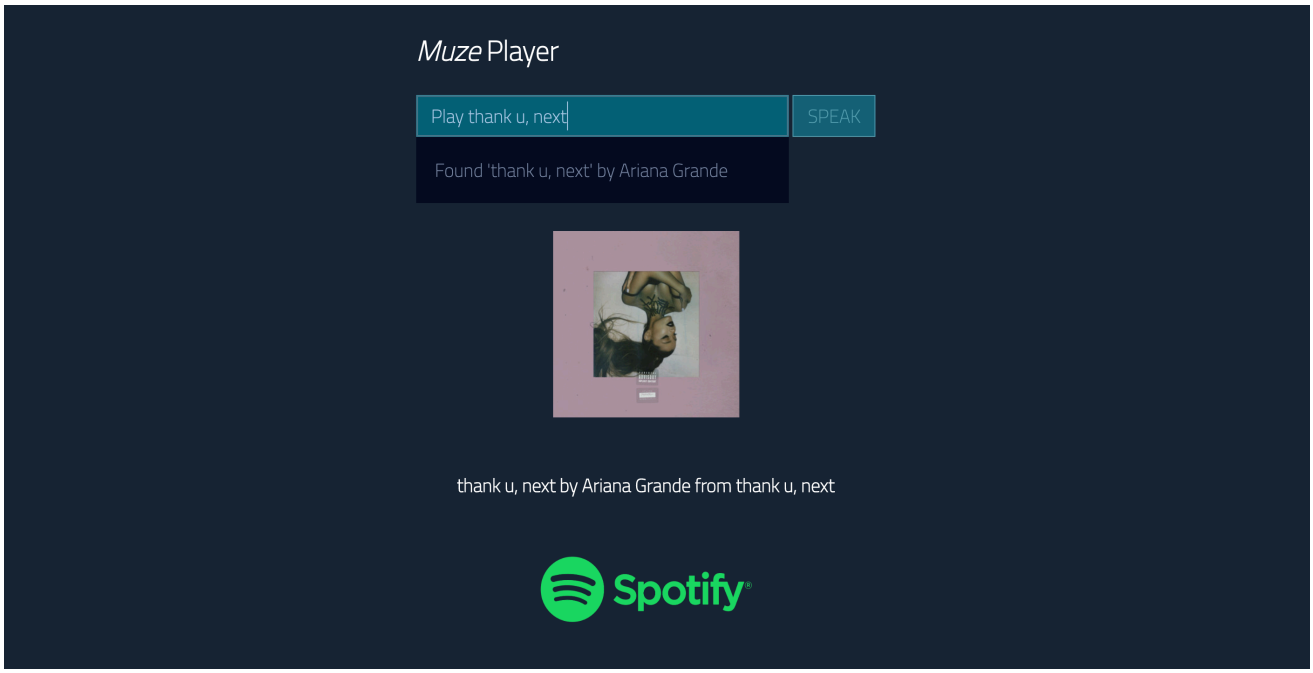
This app also required an NLU service that could integrate with the music knowledge base on the backend to retrieve a response to a user's query (req. 2a + 3b). Dialogflow supports this feature — which it calls *fulfillment* — out of the box. Integrating the agent with the backend service is achievable in two steps: supply Dialogflow with the service's URL and create a new endpoint in the backend service to serve requests from Dialogflow.

Architecture

Like many web apps, this app is composed of a client and a server.

The client has relatively little responsibility: it presents a user interface that accepts input, sends corresponding requests to the server, and presents the results. The user interface component is largely independent, so the app is not necessarily bound to a natural language user interface; and so, experimenting with other types of interfaces is certainly possible in the future.

The server does most of the work: it makes HTTP calls to Dialogflow to process natural language input provided by the client and houses the music knowledge base that it uses to find music recommendations. The music knowledge base contains information about musical 625 artists, 6249 songs, and 223 genres, stored in a relational database (SQLite); it was developed as part of separate project, the details of which are available in this report. Most notably, the knowledge base contains song audio features such as *acousticness*, *danceability*, *instrumentalness*, etc., for each song. This data is key to serving fine-grained recommendations, which is this project's main goal. All data was gathered from Spotify using the Spotify Web API.



A screenshot of the application in action. Deploying the application required complying with Spotify's Branding Guidelines

Usage

The app recognizes two main intents:

1. Play a song that is explicitly given
2. Play a song that is slightly different to an explicitly given song

The first intent can be triggered by various different phrases, including:

- `Play thank u, next by Ariana Grande`
- `Play thank u, next`
- `Play song thank u, next`
- `Play the song thank u, next`

The second intent is similar, but always requires a phrase with both a song name and an adjective (e.g. "more acoustic"):

- `Find something like thank u, next but more obscure`
- `Find a song like 7 rings but more acoustic`
- `Play a song like thank u, next but less popular`

The full set of supported audio feature adjectives, along with their recognized adjectives, are:

Audio Feature Attribute	**Adjectives**	**Adjectives (Opposite)**
acousticness	`more acoustic`, `less electric`	`more electric`, `less acoustic`
happiness	`happier`, `more happy`, `less depressed`, etc.	`sadder`, `more sad`, `less happy`, `angrier`, `more negative`, etc.
danceability	`dancier`, `more dancey`, `more danceable`, `less boring`	`less dancey`, `chiller`, `more relaxed`, `more lowkey`, etc
popularity	`more popular`, `more well known`, `less hipster`, etc.	`more obscure`, `more underground`, `more hipster`, etc.

Oftentimes, many songs satisfy the given comparison. The app randomly chooses one. Notice that this means that the recommendations are **non-deterministic**: the app will play various different songs even if you execute the same exact request repeatedly.

Below are a few of the 625 artists and 6249 songs, included to get started using the app.

Highly Popular Artists

Artist	Songs
Imagine Dragons	Bad Liar, Believer, Natural
Beyoncé	Halo, Love On Top
Taylor Swift	Delicate, Blank Space, Shake It Off
Ariana Grande	break up with your girlfriend, i'm bored, 7 rings, thank u, next
Nicki Minaj	MotorSport, Chun-Li, Bang Bang
Ed Sheeran	Perfect, Shape of You, Happier
One Direction	Story of My Life, Drag Me Down, Night Changes
Justin Bieber	No Brainer, Love Yourself, Sorry
Demi Lovato	Sorry Not Sorry, Sober, Échame La Culpa

Lesser Known Artists

Artist	Songs
Dead Posey	Don't Stop the Devil, Freak Show, Boogeyman, Holy Grail, God's Gonna Cut You Down
EZI	DaNcing in a RoOm, I'm Fine, Take My Breath Away, anxious., AFRAID OF THE DARK
Loote	Better When You're Gone, Your Side Of The Bed, Wish I Never Met You
Sonny Alven	All In My Head, Our Youth, Shallow Waters
Bryce Fox	Horns, Stomp Me Out, Chicago, Lucy, Punches
Clara Mae	Call Your Girlfriend, Drowning, Us, I'm Not Her, I Forgot
Willyecho	Welcome to the Fire, Monster
Valley Of Wolves	Lions Inside, Chosen One, Out For Blood, Rule The World, Born Bold
BBMak	Back Here, Out Of My Heart, Ghost Of You And Me, Next Time

Key Takeaways

The Spotify and Dialogflow platforms were key to the sucess of this project. Spotify's Web API allowed me to easily gather song audio feature data. Spotify's Web Playback SDK allowed me to fully stream songs from a vast selection. Dialogflow allowed me to leverage natural language understanding modules without having to implement them myself. Without platforms like these, it would have been impossible to deliver the app's end-to-end functionality as a part-time student in the span of a few months.

Furthermore, various unexpected challenges and implications arose during the development of this app:

1. **Dialogflow API Authentication**: Authenticating requests to the Dialogflow API
2. **Parameter Extraction**: Recognizing songs from user input
3. **Choosing a Suitable Platform**: Finding a suitable platform to satisfy the core requirements
4. **Deployment Logistics**: Deploying the application & complying with terms of service

(1) Dialogflow API Authentication

Requests to the Dialogflow and Spotify APIs must be authenticated with bearer tokens, which have a limited lifespan (typically of around one hour). Typically, these temporary bearer tokens are generated from permanent credentials that can be obtained through accounts with the Google Cloud Platform and the Spotify for Developers platform.

During the early stages of the project, it was practical to manually generate and paste in new bearer tokens to test the application. However, to deploy the application, it was important that the bearer tokens be generated on-the-fly by the application code itself. Getting a bearer token from Spotify was possible through a single HTTP request; however, I struggled to find an equivalent endpoint to get a Dialogflow bearer token. I finally settled on using the Dialogflow Python client to authenticate requests, which encapsulates the logic for requesting a bearer token using a Google Cloud Platform API Key JSON file.

However, this required that I move, from the client-side to the server-side, the request to the Dialogflow Detect Intent endpoint; waiting until later in the project to choose this way of generating the bearer token forced me to refactor my application significantly.

(2) Parameter Extraction

The app relies on Dialogflow to process raw user input. Specifically, the app uses the Dialogflow agent to recognize the user's intended action and the corresponding parameters. For example, when a user says "Play Truth by Ed Prosek", the Dialogflow agent extracts the song name, "Truth", and the artist name, "Ed Prosek", which are the key pieces of information necessary to fulfill the request.

Dialogflow requires the developer to define the **entity type** to be extracted, the type of the parameters to be extracted from the user input. Luckily, Dialogflow agents come with knowledge of musical artists, which is of the provided entity type `sys.music-artist`. Unfortunately, Dialogflow agents do not come with knowledge of songs; so, I had to create a `song` entity type. This is rather straightforward. However, if the Dialogflow agent is not provided examples of the `song` entity type, it fails to extract the parameter from the user input.

To work around this issue, I wrote a script to pull all the song names from the music knowledge representation system, so that I could upload them to my Dialogflow agent. Anyone looking to work with a virtual assistant service like Dialogflow should identify the entities they will need extracted and whether they will be able to provide them to Dialogflow. Moreover, relying on pre-provided entities is advisable.

(3) Choosing a Platform

Music streaming functionality was crucial to making this app actually valuable. Without it, the app would provide little more than a Google search. By choosing a virtual assistant service with a REST API for processing natural language input, I allowed myself to use any platform capable of sending HTTP requests. My choice of virtual assistant service could have instead confined me to a platform where music streaming was impractical or altogether impossible, like Google Assistant.

(4) Deployment Logistics

I deployed the application to share it and for it to be evaluated. I used Heroku because it provides a great free tier that sleeps the app when it is inactive. However, Heroku does not give the developer direct access to the server. Configuration must be provided either in the version control code repository or as environment variables. For my purposes, Heroku was sufficient.

I ran into one major snag during the deployment of the application. In its original form, the Muze service would make an HTTP request to the Dialogflow agent, which would make a separate HTTP request back to the Muze service before responding to the original request. And so, the Muze service would recieve an HTTP request from the Dialogflow agent while it waited for an HTTP response from the same Dialogflow agent. This worked fine when running the app locally, but seemed to cause problems when I deployed it to Heroku. I decided to turn off webhook fulfillment in my Dialogflow agent so that it would not make the request to the Muze service; instead, it would simply return a response containing the matched intent and extracted parameters, which were enough to fulfill the request within the Muze service.

In addition, since I was integrating with Spotify's platform, I ensured that my application met their terms of service. In particular, I had to add their logo to my app's interface, specify music metadata like song and artist names, and provide links to the current track in Spotify's service. Anyone looking to develop and eventually deploy an app using third-party services should investigate the terms of agreement early on.

Testing

User Testing

The application is available online. It requires a Spotify Premium account, since the Spotify Service is used for music streaming. The app satisfies the main use case described earlier. Below is a contrived example scenario in which a user begins with an arbitary song and repeatedly asks for a more acoustic song. It is meant to highlight two main features of the app's functionality: its abilities to find songs by similar artists and to find songs based on a fine-grained attribute (acousticness):

#	User Input	Message returned by app	Action taken by app	Info
1	`Play Greatest Love by Ciara`	`Found Greatest Love by Ciara`	Plays the song.	This song has a relatively low `acousticness` score.
2	`Play something like Greatest Love but more acoustic`	`Found 'Like a Boy' by Ciara`	Plays 'Like a Boy' by Ciara	This song is considered similar because it is also by Ciara; it has a slightly higher `acousticness` score than "Greatest Love".
3	`Play something like Like a Boy but more acoustic`	`Found 'Chun-Li' by Nicki Minaj`	Plays 'Chun-Li' by Nicki Minaj	This song is considered similar because it is by Nicki Minaj, a related artist to Ciara according to Spotify. Chun-Li has a higher `acousticness` score than Like a Boy.
4	`Play something like Chun-Li but more acoustic`	`Found '7 rings' by Ariana Grande`	Plays '7 Rings' by Ariana Grande	Like 3
5	`Play something like 7 rings but more acoustic`	`Found 'needy' by Ariana Grande`	Plays 'needy' by Ariana Grande	Like 2
6	`Play something like needy but more acoustic`	`Found 'Consequences' by Camila Cabello`	Plays 'Consequences' by Camila Cabello	This song is markedly more acoustic than "Greatest Love" by Ciara, the first song in this sequence of recommendations. According to Spotify, Ariana Grande and Camila Cabello are related and they so are considered similar by the Muze app.

Even at its prototypical state, the app proves an effective way to explore music based on arbitrary attributes.

Automated Testing of Voice Interface

In addition to plain text, Dialogflow accepts audio data as input. I wrote a script to gauge how accurately Dialogflow would match generated audio files to the appropriate intent. I generated virtual voice audio files for phrases like "Play Hello" and "Play something like Hello but happier", using the OS X command-line tool `say`. I did so with:

- a randomized batch of 100+ songs retrieved from the application database
- two different intents (discussed below)
- two distinct phrases per intent
- and, where relevant, three distinct adjectives corresponding to acousticness, valence, and popularity

The vast majority of errors occurred in *extracting the song name* from the input phrase; in contrast, errors caused by intent matching and adjective extraction were rare. Of the song name extraction errors, a large share were caused by song names containing dashes and parentheses. In general, they were responsible for about half of the song name extraction errors:

- `Find Song` intent: 18% out of the total 35% errors caused by song name extraction
- `Find Slightly Different Song` intent: 18% out of the total 40.6% errors caused by song name extraction

The test results follow in more detail.

Testing `Find Song` Intent

Tested with:

- 230 songs
- 2 different phrase formats: `Play {song}` and `Play song {song}`

Total tests	Perfect Pass	Song Name Extraction Errors	Intent Matching Errors	Ave Intent Matching Confidence
430	65%	35%	1.3%	95.6%

Testing `Find Slightly Different Song` Intent

While the Dialogflow agent consistently recognized the correct intent, it had trouble extracting the song name. Part of this is due to special characters in songs (like dashes and parentheses).

Tested with:

- 115 songs
- 2 different phrase formats:
  - `Play something like {song} but {audio_feature_adjective}`
  - `Find something like {song} but {audio_feature_adjective}`
- 3 different audio feature adjectives:
  - "more acoustic"
  - "more obscure"
  - "happier"

Total tests	Perfect Pass	Song Name Extraction Errors	Adjective Extraction Errors	Intent Matching Errors	Ave Intent Matching Confidence
690	57.7%	40.6%	3.8%	1.9%	86.8%

Future Work

With the foundation laid, this application provides an opportunity to explore different music exploration experiences:

- **Add buttons for instant request**: let users instantly request common recommendations (e.g. "more acoustic") with suggestion chips or another UI component of the like
- **Add interactive visualization controls**: let users visualize the current song's audio feature values (e.g. with intensity sliders) and request new songs by playing with them
- **Gather user interaction data**: set up application in the style of mechanical turk to gather common requests and their corresponding phrasings

In addition, there are many ways to improve the app's robustness:

- **Improve song name matching**: find songs despite minor differences (e.g. omitted/incorrect punctuation, accents, typos, etc)
- **Dynamically Find Songs**: if a user requests a song that is not in the database, retrieve it from Spotify dynamically

Furthermore, the app can be extended to offer many more features:

- **Onboard the user**: create an interactive user guide to using the app's interface so that users learn about its features
- **Stream from multiple providers**: e.g. Apple Music, YouTube, SoundCloud
- **Allow fine-grained control of existing attributes**: the app's database contains all attribtues found in Spotify's Audio Feature object; use them to allow users to find music based on more attributes (e.g. time signature, key signature, etc.)
- **Extend Song Feature Set**: gather or mine more attributes for fine-grained control (e.g. jazziness, experimentalness, rhythmicness, etc.)

Bibliography

Docs

- Dialogflow Detect Intent Endpoint
- Dialogflow Python Client
- Flask
- Flask Assistant
- Flask Socket IO
- Heroku Buildpack Python
- Heroku Config Vars
- Heroku: Deploying with Git
- jQuery API
- Spotify Developer Terms of Agreement
- Spotify Web Playback SDK
- Spotify Web API
- W3 Schools

Articles

- Web Sockets

Tutorials & Examples

- Spotify Implicit Grant Flow
- How to Build Your Own AI Assistant using API AI
- Using Google Cloud with Heroku