ECE 180D

Final Project Report

(Winter 2021)

Project Smartify: Intelligent Music Player

Team 9
Hyounjun Chang
Karunesh Sachanandani
Gerald Ko
Justin Suh

Abstract

Smartify is an intelligent interactive player that plays the best smacking tunes at the right time in the right place for the users. Smartify can detect the listener's activity, mood, and gestures in order to play the most suitable songs to get the user in the groove. You can find our recorded product demo here.

Project Description

Objectives

- To create music playlist suggestions based upon user inputs (camera, microphone, gestures).
- Create an interactive and engaging way to listen to music between multiple people.
- Create an easy environment for users to share/discover music from other people

Features

- **Personalization**. Provide a personal touch and specific feel to the music that are suggested based on the user's emotions
- **Collaboration**. Share music suggestions with friends that are subscribed to the same channel, in-real time!
- **Voice Activation**. Use voice to control specific actions (e.g. next song, pause, search a song online, skip 30 seconds).
- **Gesture Control**. Hand gesture using an IMU to control (e.g. next song, pause current song).
- **Activity Detection**. Recognize user activities with the IMU used as a remote as an alternative to gesture control.

Use Cases

- Control the playlist (skip, pause, play) from a distance (half a globe away) with gestures
- Feeling gloomy, create a playlist of happy music to cheer you up!
- Listen to your friends playlist, live, without hassle of looking songs up manually!

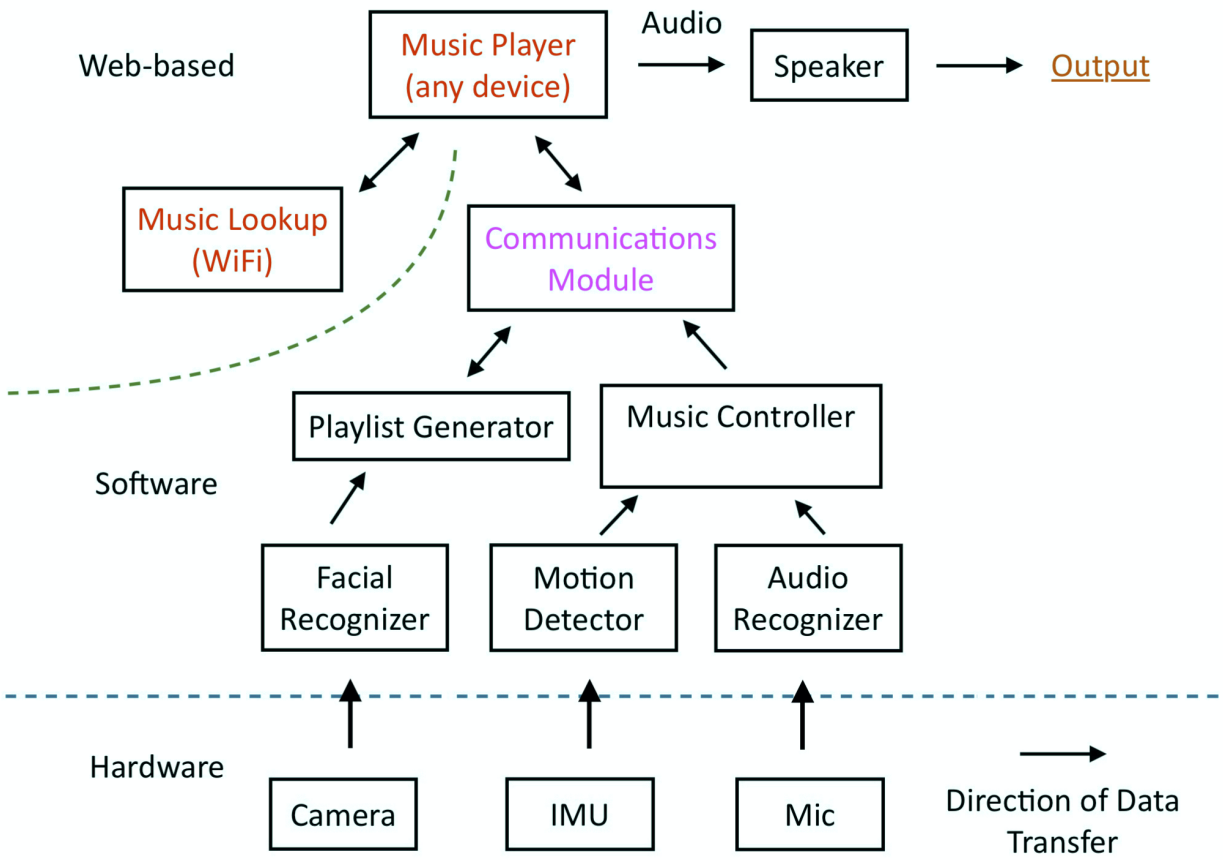
Requirements

- IMU, Raspberry Pi (if using a ,motion controller)
- Microphone
- Camera
- Hardware capable of installing Smartify Player
- Speakers

System requirements for installing Smartify Player is listed on the Github page. Smartify Player was only tested to work as desired on Python 3.8 or higher.

High-Level System

Architecture and Block Diagram



High-Level Task

Design

- Hardware Level
    - *Camera, Gyroscope/Accelerometer, Microphone*: Input hardware from user, sends data stream to Raspberry Pi/Hardware
- Software Level
    - *Facial Recognizer (Emotion Detection):* Takes image stream as input, outputs features that may be helpful to identify emotional state of users.
    - *Motion Detector:* Takes input from Gyro/Accel controller, outputs the corresponding player command (play pause)

- *Audio Recognizer:* Looks for registered voice commands, and sends the commands to be parsed by the music player.
- *Playlist Generator:* Generates a playlist given inputs (emotion of user)
- *Music Controller:* Checks results from lower level modules and determines command to send to Communications Module (if sending command via MQTT)
- *User Database (Within Playlist Generator and Music Player)*: Stores relevant information regarding users (song metadata, directory, emotion associated with the song)
- *Communications Module*: sends data between the controller and music player (or locally within player) to send playlist/player commands
- *Music Player (Within Playlist Generator)*: Plays music from output device, through communicating with Communications Module (with GUI interface)
- *Music Lookup*: Finds music that is not within the music player's local directory via WiFi, and return an audio stream to be played on the player

Implementation

- Hardware Level
  - *Camera, Microphone, Gyro/Accel* - physical hardware to get raw input data from users.
- Software Level
  - *Facial Recognizer:* Takes image stream as input, processes it via Computer Vision Library (OpenCV), using a trained Neural Network (Tensorflow) to detect faces and emotions of users.
  - *Motion Detector:* Uses a Threshold Value for Accel/Gyro for certain motions, and outputs the corresponding command.
  - *Audio Recognizer:* Use python SpeechRecognition library to parse inputs from users.
  - *Playlist Generator:* Uses features selected from input modules to generate a playlist. When not specified, creates a random playlist of songs in the directory. When using "Smartify" features, suggests songs that match the perceived mood of the user. (Song mood could be specified by user)
  - *Music Controller:* Polls response from other modules (Motion detector, Audio Recognizer), sends it to communication module (for MQTT or interprocess communication)
  - *User Database:* Pandas dataframe containing relevant metrics (metadata, file location, emotion associated with song, etc.)
  - *Communications Module:* Communicates with other devices by MQTT, or guides interprocess communication between the same devices.

- *Music Player (Within Playlist Generator)*: Run python-vlc to play a variety of music files/audio streams, given the path/audio stream link. GUI interface will be through TkInter.
- *Music Lookup*: Send a query to YouTube, get audio stream link via youtube-dl API to be played by the Music Player

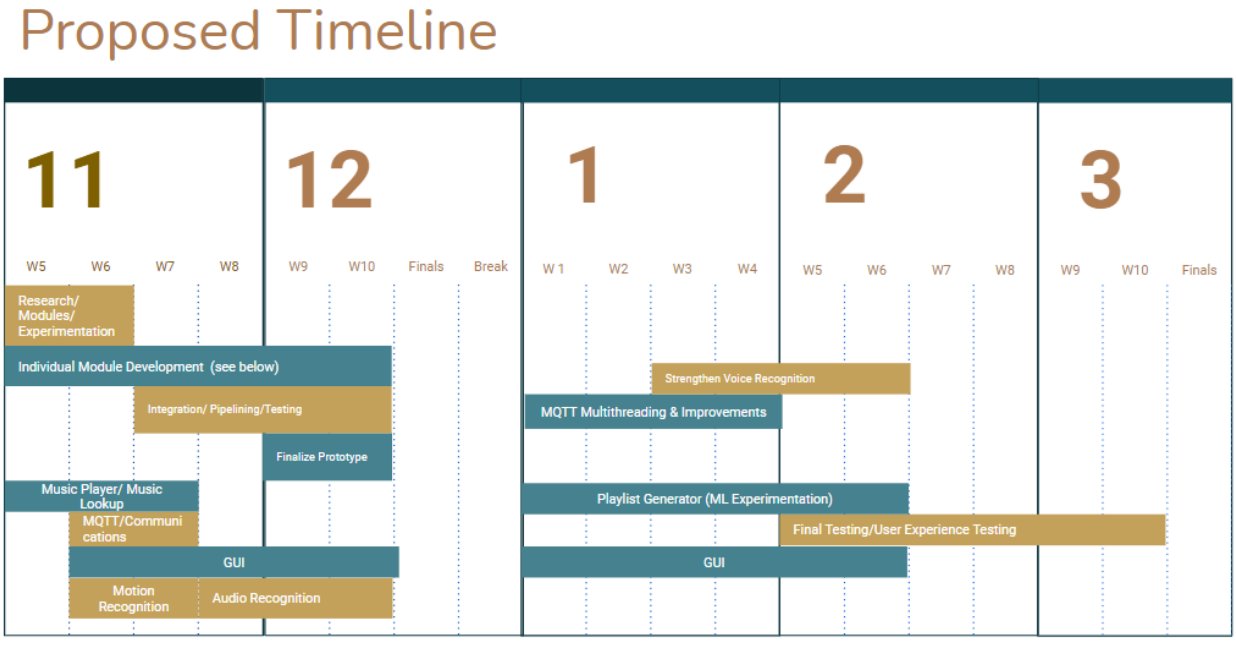
System Design Choices

There were some design constraints we wanted to meet for our product.

1. Free Product: This meant that we couldn't use some paid software like Microsoft Azure. We have used open-source libraries for our project as a result.
2. Cross-Platform Support: We wanted our product to be functioning on different operating systems. Since our development time was limited, we have decided to use a scripted language, Python. Any device that can run python and our libraries can run the program.
3. Scalability of the Project: We wanted our player to be able to grow with more features. When choosing libraries, we mostly chose open-source libraries that have been around for a long time for stability and functionalities. For the music player, vlc-python was our library choice, as it had multiple functionalities (like audio-streams) that could be added when adding more features to our player.

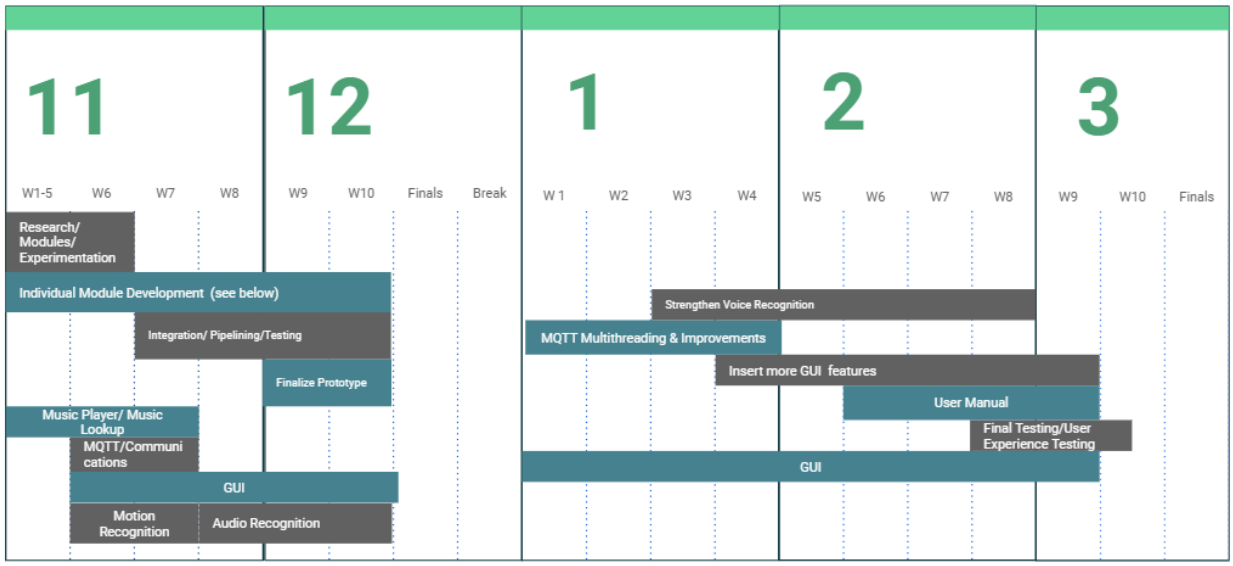
After researching which libraries could be used for our modules (after system design), we have worked on the modules to be as modular as possible so it would be much easier to debug. The only part of the project where it wasn't as much scalable when adding functionalities to our player was from the GUI Music player, as it had to integrate all different modules.

Progress to Final Product



Our Proposed Timeline from Week 5, Fall Quarter

Project Timeline



Our Actual Timeline

As you can see above, the initial proposed timeline and the actual timeline, most of the goals on the timeline were completed. We spent the entirety of the first quarter creating modules that would be crucial to the smartify player later on, just like the Emotion Detection Module or Communications modules. First 5 weeks of the Fall quarter were used to research how to build our modules and testing with different libraries. By the end of Fall quarter, we had finished the first prototype of our modules, and created our “prototype product” that integrated the most basic modules.

There turned out to be quite some mess when we were integrating the modules together during winter quarter; we had issues with the GUI library, audio library, and multi-threading. Our initial prototype from the first quarter did not have all the core features we wanted (like multiple room sharing or searching songs online), nor the stability (Emotion Detection module would freeze the player). After winter break, our group has decided to focus more on the stability aspect of the player to improve the user experience. We fixed the most crucial integration/pipelining issues by Week 3 of Winter Quarter.

The second quarter has mostly been about focusing on user experience. We were trying to figure out what additional functionalities would be useful for the average user. For example, allowing much more audio commands in the Audio Recognition module would help the most average user, alongside easier GUI redesign to guide users to different functionalities. As seen by the two pictures above, our actual timeline wasn’t too far off from our predicted timeline. We believe that we did great job finishing tasks as a team, fulfilling good two-thirds of the functionalities we proposed in the Winter Project Proposal (the remaining one-third were mostly less trivial functionalities).

Most of our goals from the first quarter were met by the end of this quarter. The list below lists some few features we had planned to finish that were finished..

- Working music player that could be controlled by IMU
- A music player that allows sharing between multiple users
- A music sharing system that saves bandwidth by sending minimal amount of messages rather than audio streams
- An emotion detection module that can be used to detect emotion of our user
- A playlist generator that creates a playlist based upon the user's emotions
- An easy, intuitive GUI that users won't need to read 10 pages of manual for most features
- A database system that can support different directories for music lookup
- A simple audio-based command system that can control our music player (like Alexa) and much more…

However, there was one major feature that ended up not being finished due to our lack of resources and expertise; it was the Playlist Generation with ML (and signal processing), as seen on the timeline graph. While researching how to classify songs based upon emotions, we have found out that it was actually computationally expensive, as it requires fast fourier transformations and model design for suggestion algorithms. Since our team did not have any resources (song licenses or AWS for cloud servers), we could not store the "musical values" such as "intensity" or "pitch", meaning that we had to locally compute the values and create a suggestion algorithm accordingly. This was a problem as these operations are computationally expensive, and would make the codebase much harder to maintain, taking away most of our time. We as a group decided to not implement this feature due to our constraints, and have decided to take user inputs from a .csv file instead. Our player was redesigned with this in our minds.

It was sad to see one major aspect of our Smartify Player not being finished, but we believe that it was a reasonable choice to create a complete product within the 20 week time-frame.

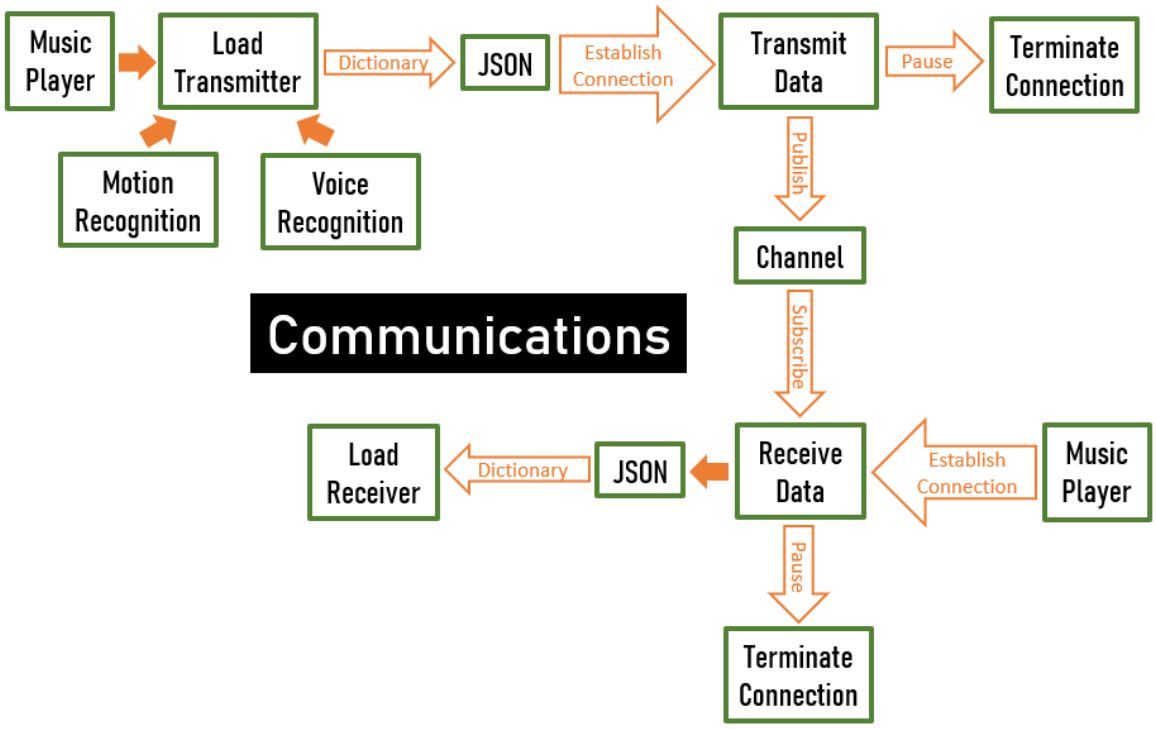
Detailed Experimental Verification and Testing (Data)

MQTT Communication (Karunesh)

The initial goal for the MQTT module was to design a system that allows users of the music player to share their current playlist with user's around the globe, and for them to be able to

tune in to the same playlist. As time progressed, the goals of the MQTT module grew as it became important to be able to transmit and receive the commands inputted from the Motion Detection and Voice Recognition modules via MQTT. This was because our current RPi setup does not have a microphone or a speaker and in order to realize the full features of the application, we decided to make the distinction between the player and the speaker. The two can of course be running from the same device, but they are not constrained to be so.

In order to achieve this functionality, much of the focus last quarter was on systematic design of the MQTT module and its integration with the other modules. In doing so, I was not able to effectively multi-thread the MQTT process with the music player, which would have allowed "live" sharing of the playlists.



However, this quarter my focus became to improve the communication capabilities of our music player and implement live playlist sharing. This was given a higher priority partly because of the fact that other modules (Voice and IMU) depend on the MQTT channels for intra player communication. Even more importantly was the focus on strengthening the multi user capabilities of our product, as part of improving user experience and preparing a more finished product that would be easy for another team to use and review. To start off, we established separate threads so that the transmitter and receiver could run parallel to the music player. In this way, we enabled live sharing of playlists between users.

In its final state, the music player now has instances of the transmitter and receivers, and separate threads for each. The threads are set “alive” on pressing the button to Toggle the transmitter/receiver ON.

```
# MQTT Transmitter (from MQTT module)
self.transmitter = MQTTTransmitter()
self.transmitter_client = self.transmitter.connect_mqtt()
self.transmit_msg = False

self.transmitter_thread = Thread(target=self.transmit)
```

```
if self.transmit_msg == True:
    self.transmit_msg = False
    self.button_transmit.configure(text="Trasmitter: OFF")
    print("Transmitter Turned Off")

else:
    self.transmit_msg = True

    if not self.transmitter_thread.is_alive():
        # only start thread if thread is not alive
        self.transmitter_thread = Thread(target=self.transmit)
        self.transmitter_thread.start()

    self.button_transmit.configure(text="Trasmitter: ON")
    print("Transmitter Turned On")
```

Once the transmitter is turned on, it continues to gather the current song information from the user’s player, and broadcast the relevant information over the channel that the user has set. This data is sent in JSON format via MQTT, and the process is updated real time so that changes on the user’s music player would be replicated on any of the listeners as well (fast forward 30 seconds, pause, change song, etc). However, transmitting a signal “continuously” isn’t possible. The fastest that our version of MQTT could handle without errors would be approximately 0.2 seconds as seen from the tests last quarter. However, when using a rudimentary form of communication such as MQTT, it does not make sense to transmit that often as you would only crowd up the user’s channel with constant messages of updated songs. And so, we decided to transmit a message at an interval of 3 seconds as a middle ground choice between live updates and convenience.

```
while self.transmit_msg == True:
    (song_metadata, songtime) = self.get_info_current_song()

    if song_metadata is not None:
        songname = song_metadata.title
        artistname = song_metadata.artist

        self.transmitter.setSongname(songname)
        self.transmitter.setArtistname(artistname)
        self.transmitter.setSongtime(songtime)

        if self.player.is_playing():
            self.transmitter.setCommand("INPUTSONG")
        else:
            self.transmitter.setCommand("PAUSE")

        self.transmitter.publish(self.transmitter_client)

    time.sleep(3)  # Interval to sleep between each message
```

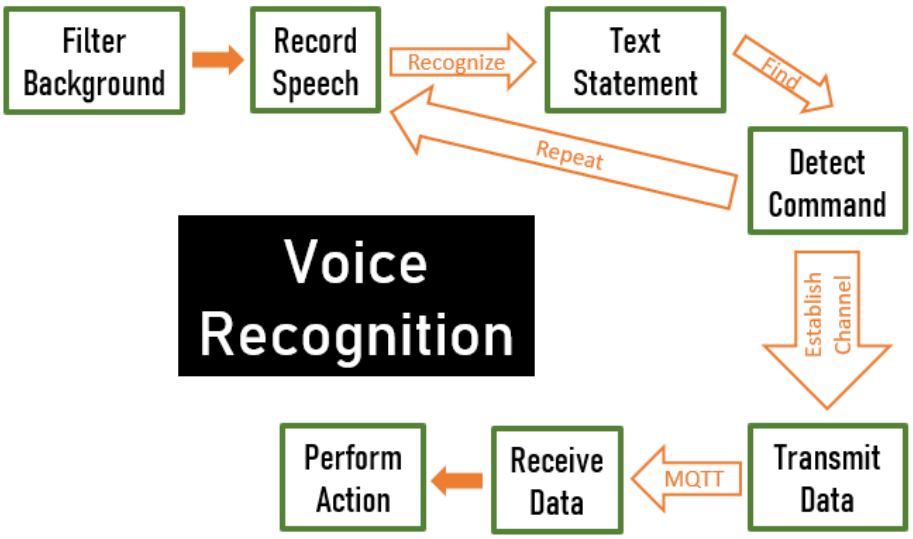
On the receiver end, the process is similar. A thread is active for as long as the receiver is turned on. In this state, song information from the user’s channel is picked up and played. If the song is not present in the user’s current library, Smarify searches for it online and plays the first result on youtube. As new updates are received from the channel, the receiver’s player changes the song if the timestamp of the current song, and that of the incoming message differ by over a

second. This is to ensure that noises during communication do not result in a choppy stream on the receiver's end.

This quarter, I also worked on setting up different rooms for each user so that multiple users around the globe could use the music player independently and share their playlists with their followers, without interference. This was definitely useful when we had members from other teams testing our product as Smartify could now be used by various groups of people concurrently. This also involved working on integration with the music player and adding these features on the GUI. Finally, the MQTT module can also now be used by the Voice and IMU to send commands over specific defined channels.

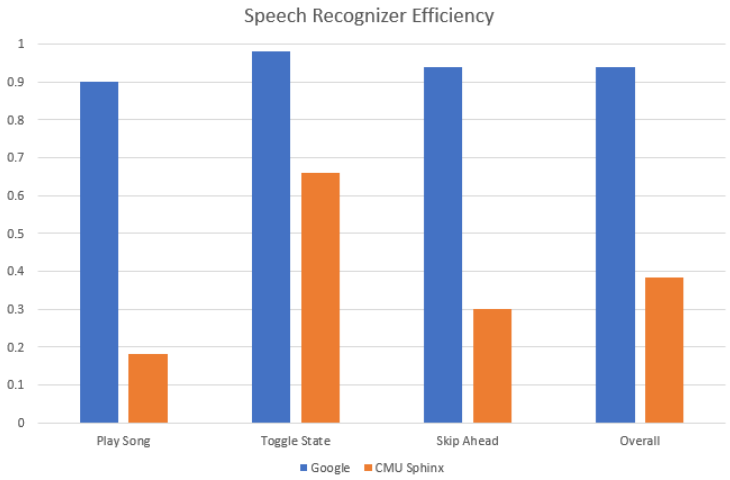
Audio Recognition (Karunesh)

Currently, Smarify uses a laptop's microphone to accept voice commands from the user. As of last quarter, the recognized commands would be transmitted through MQTT via a controller that had to be run separately. As improving user experience was the primary goal for this quarter, I modified the structure of this module to do away with this unnecessary inconvenience. Now, voice commands can be sent directly through the GUI of the player itself, resulting in a seamless user experience. Furthermore, another controller, that is too primitive to run the music player, could run just the voice recognition module using a separate script, and this would transmit the commands to a custom MQTT channel.



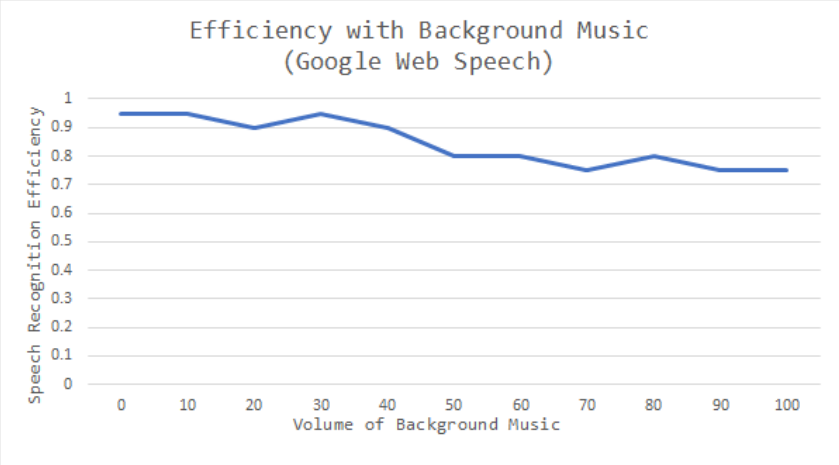
The Voice Recognition module allows Smartify's users to input voice commands to interact with the player and control it with certain commands. We have implemented the Python library of Speech Recognition in order to accept voice commands, parse the audio input from the user in order to extract specific commands, and communicate those commands with the music player.

Currently, Google's web speech API is used to recognize the commands. An implementation of the CMU Sphinx library was tested for the Voice Recognition module. This library had the advantage of being able to be used offline, and had a greater degree of support for mobile applications. Unfortunately, after testing with a variety of possible inputs, it appeared that the CMU Sphinx recognizer was a lot less accurate than the current Google recognizer. The following types of commands are currently able to be processed: Play a song by an artist, toggle the state of the music player (play/pause), and skip ahead a certain amount of time in the current song. For each of these commands, the accuracy of the two recognizers was compared over 50 commands of each type (per recognizer).



As we can see here, there is a very large difference in performance. To note, in particular the CMU Sphinx library performed very poorly in recognizing song/artist names that were not very common words/names (< 10%). After testing Team 8's MEAT Repo, which used an implementation of Sphinx, we found that the library itself seems to have really poor performance for anything other than the most primitive phrases. The Google API matches our needs well and so we decided to continue with it.

I decided to use Python's ambient noise filtering module to filter out the background noise. The module records the background sound for a short period (set as 1 second here), and adjusts the remaining audio using that. Ideally, a user would be able to use the voice recognition module with music playing in the background. After some tests (see figure below), it was found that the noise filtering works rather well and yields a negligible performance drop at moderate levels of background music. Even at maximum volume, the command recognition accuracy is still > 75%.

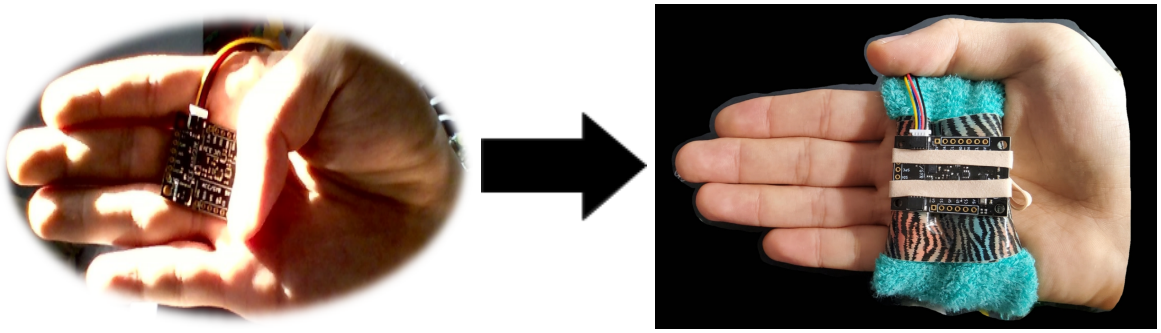


This quarter I worked on increasing the flexibility of the possible accepted commands, so that the user could enter the same command using different phrases. Also additional capabilities were added so users could start their desired song at any specific time, or skip ahead to any time as well, allowing for an even greater degree of control just through voice commands.

```
# if the command also says at ____ seconds (start at this time)
if secondsind - atind == 2 and atind > byind + 1:
    songtime = int(wordsinput[atind+1])*1000
if minutesind - atind == 2 and atind > byind + 1:
    if wordsinput[atind+1] == "one":
        songtime = 1000*60
    else:
        songtime = int(wordsinput[atind+1])*1000*60
self.songtime = songtime
```

IMU Motion Detection (Justin)

The IMU will be used to recognize command gestures to control the mp3 player. There will be commands such as toggle(play or pause), next, and previous. Previously, the IMU was placed on the palm of the hand as shown on the left below, but this was changed due to the user friendly issues where the user would have to hold the IMU. Now the IMU is attached to a wristband to allow ease of use for the user.



After testing different gestures, it was decided to do the following.

Toggle	Lift up and down to original position
Next	Tilt Right and back to original position
Previous	Tilt Left and back to original position

These motions are fairly intuitive especially for next and previous. I asked family members to do a gesture they think is good for each task except for toggle and they performed the gesture declared. The play and pause gesture was combined as a song will only need to take in one or the other for play and pause. This command is called toggle as stated before.

Bird's eye view of the hand. The black arrow represents the movement from initial hand position and the red back to original position

The figure on the left is for the tilt to the right to play the next song output. The flip of this would be for the previous song.

The initial gesture recognition used a simple threshold filter to determine the command. However, this method was too simple and had many false positives and negatives. The new filter uses dynamic thresholding, where the threshold changes depending on the difference of the previous reading. This gesture recognition was created by first taking the difference between the read GYRx/GYRz value and its previous value. Pass that through the first filter, then receive a time sleep to take another reading to check for the return movement. If the return movement meets the second modular threshold, the command will be sent and pause the reading for 0.3 seconds. The thresholds for all the gestures are listed in the table below.

Command		First diff read	Second diff read
Next	GYRx	>1800	<-0.9*(pastGYRx - GYRx)
Previous	GYRx	<-2000	>-0.9*(pastGYRx - GYRx)
Toggle	GYRz	>2000	<-0.9*(pastGYRz - GYRz)

The initial and final accuracy of this implementation was tested over 20 repeated commands and 5 repeats. The results averaged and rounded are listed below:

- Toggle - 95% -> Toggle - 95%

- Next - 85% -> Next - 95%
- Previous - 90% -> Previous - 95%

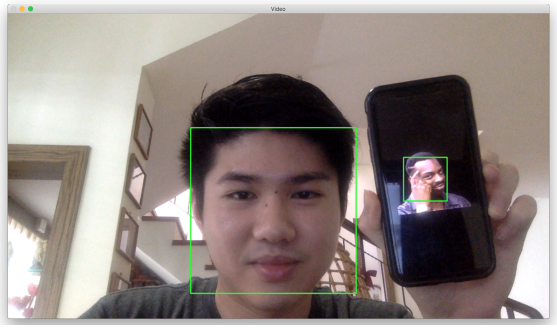
The IMU code uses MQTT to send the read command to the mp3 player to be used. There is latency of about 0.5 sec due to the time delay implemented and ping. Initially, the IMU only sent commands to a main room. Now, the IMU can be sent to a specific room to control the music player. Overall, the IMU's goal was fully completed, but to further improve the system, a new controller which can be easily attached to the hand and have a better connection system.

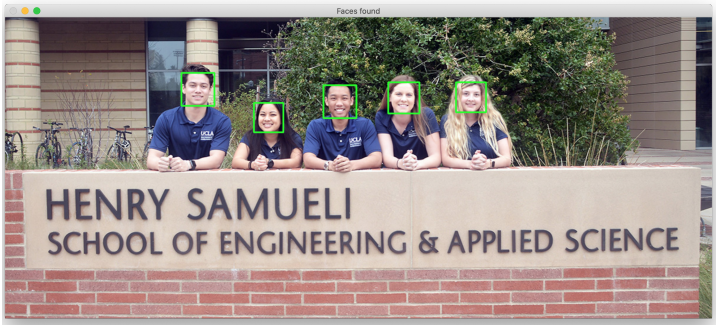
Emotion Detection (Gerald)

Emotion detection is a feature of Smartify whereby users are able to use the webcam on their computers to identify their moods based on their facial expressions. The detected emotion is then used by the player to intelligently play a song that matches the atmosphere of the room.

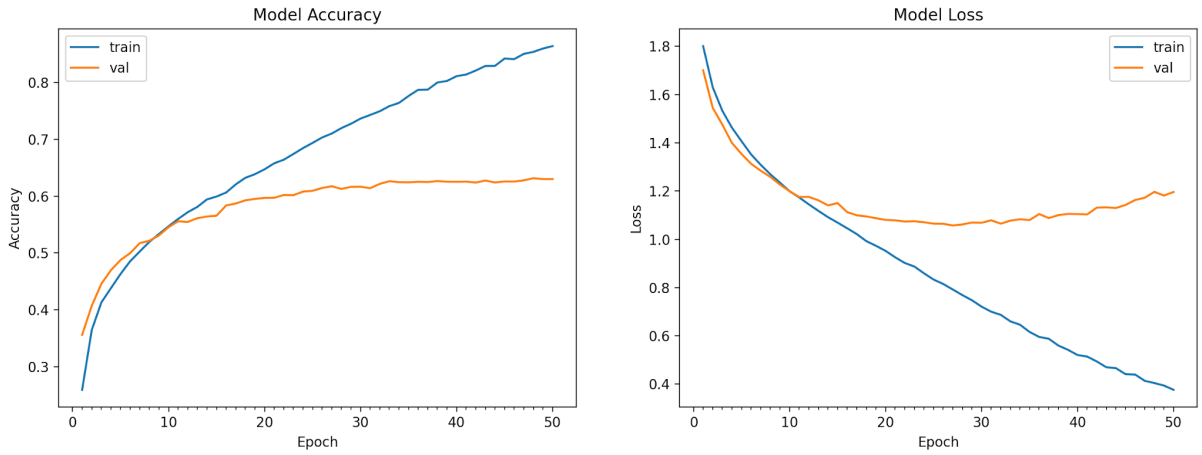
On a high level, this feature takes the camera input feed and first uses Haar Cascade, a machine learning object detection algorithm to identify the user's face. The Haar Cascade is used due to its high response rate and appreciable accuracy. Then, a 4-layer convolutional neural network is used to classify the user's emotions into one of seven different categories: neutral, sad, happy, angry, disgusted, surprised, and sad.

Attached below is the result of facial detection that we implemented:



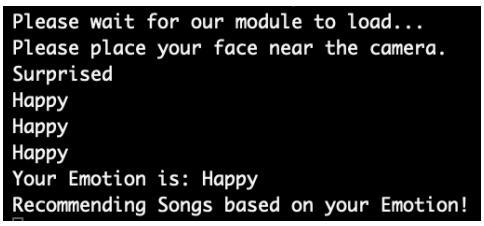


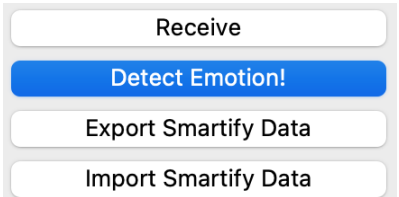
As for the emotion detection neural network, the model is trained using the **FER-2013** dataset which was published at the International Conference on Machine Learning (ICML), containing 35887 grayscale, 48x48 sized face images. For my specific model (which you can find here), it is trained using Adam Optimization and for 50 epochs with an accuracy of 63.2% which is not fantastic but appreciable.

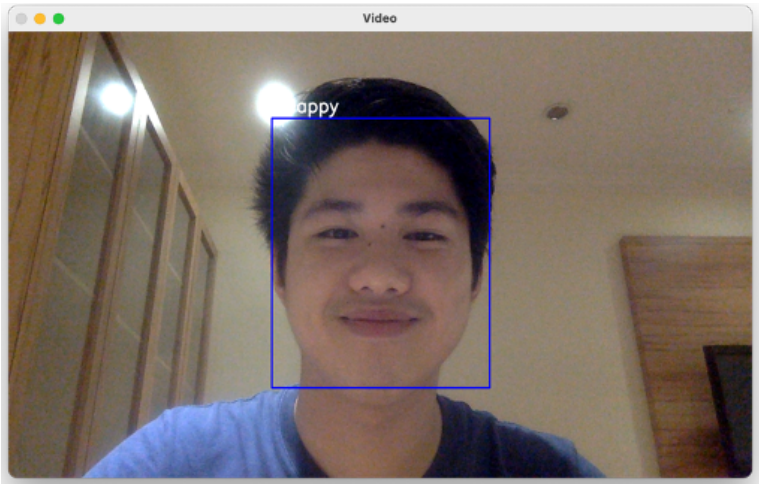


As we can see from the Model Loss graph above, the loss of the validation set (orange) ticks upwards as the number of epochs approaches 50, showing us that the model starts to overfit there. Therefore, I decided to not train my model with an even higher number of epochs, given that the 50 epochs model took an entire morning to train on my local computer.

Now that we have the emotions detected with an accuracy of about 60%, I explored various ways to use the detection in a more meaningful and accurate way. After trying multiple approaches, I decided to implement an emotion array to keep track of the emotions detected by the model. The emotion module only successfully registers an emotion once three consecutive identical emotions are detected, and this confirmed consistent emotion value is used as a reference to play the right song.







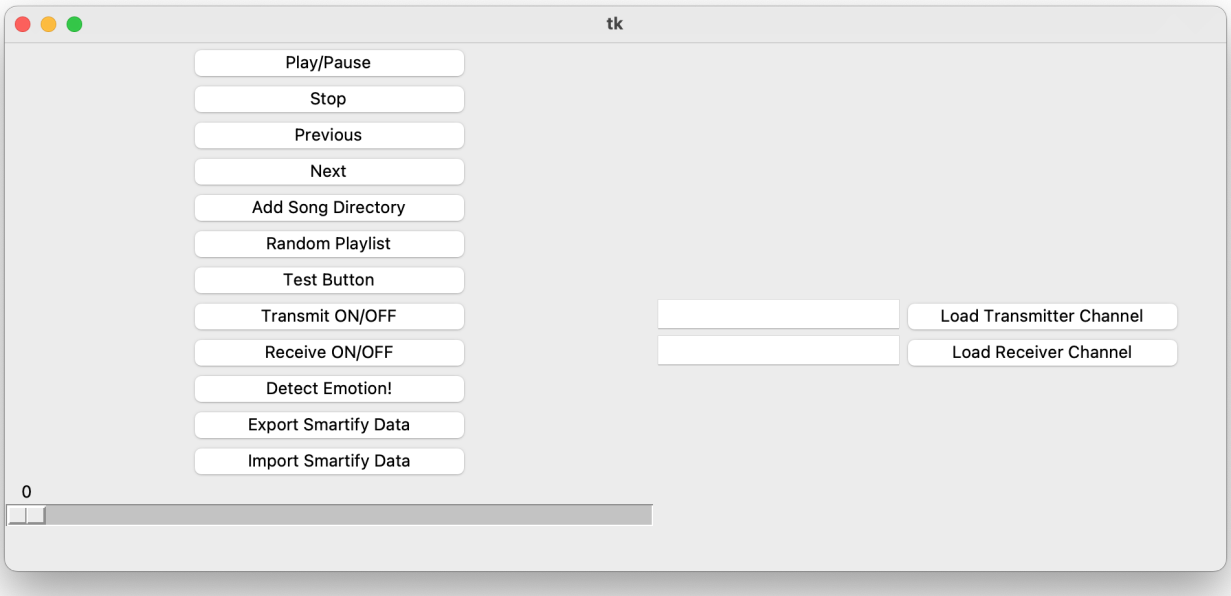
Just a side note, the module is run as a sub-process when the Emotion Detection function is toggled on, and the process terminates with the emotion value as exit code before sending it to the main player process. Here are some screenshots of how the module works:

Finally, the last part of the emotion detection module is the identification of the emotions of various songs. Currently, for the working prototype, songs are pre-tagged with one of the seven emotion values. Users can first “Export Smartify Data” which is a CSV file containing a list of songs in the music player’s directory. Users are then free to change the corresponding emotion values for each song as they please before they use the “Import Smartify Data” function. This mapping will be used by the player to play the right tunes. Moving forward beyond this class, there is a potential to implement our own model to detect emotions of different songs instead of the manual approach we currently have.

Graphical User Interface (Gerald)

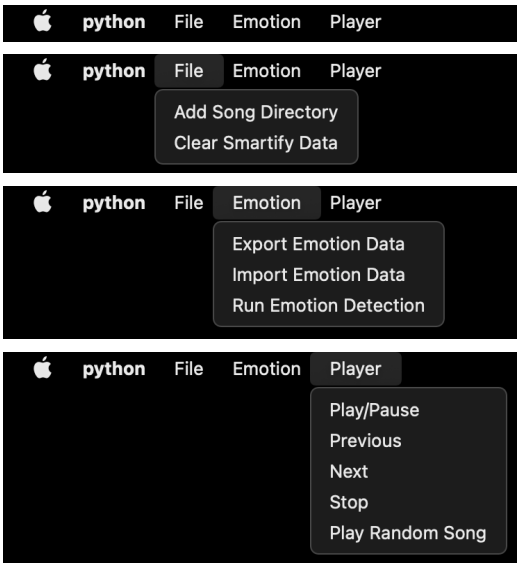
We spent the first quarter of this class implementing the different modules of our intelligent music player. This quarter I spent most of my time implementing the graphical user interface of the player to ensure our users have a seamless and intuitive experience interacting with our product.

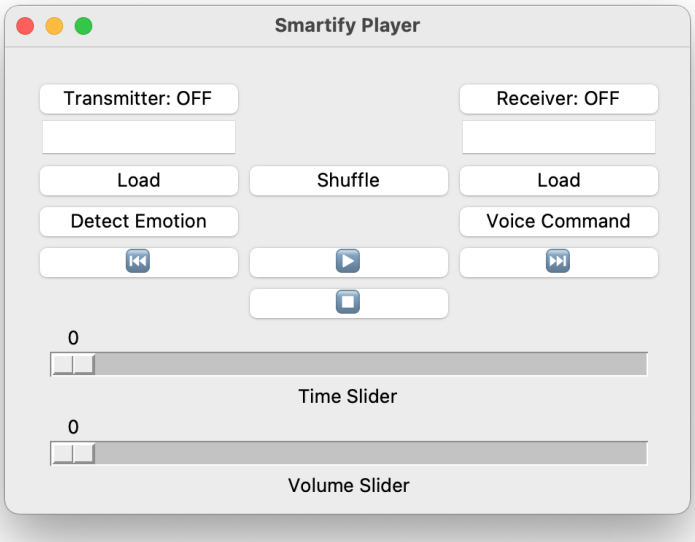
Attached below is a screenshot of our player in Fall quarter:



As you can see, the initial GUI of the player is very basic and does not necessarily have the most elegant or intuitive interface. That being said, we iterated through different variations of the player and finally landed on a much more simplified and simplistic design.

Smartify Final GUI





To create a more intuitive user interface, we decided to include the importing and exporting functionalities into the menu bar, such as “Add Song Directory” and “Import/Export Emotion Data” features. This allows us to have only relevant features and buttons of the player to be on the player window.

Music Player/Integration (Hyounjun)

My part of the project as Integration manager and working on the Music Player was based more upon using existing system design principles and debugging, so there is less “data” to provide. However, since the music player module was supposed to be able to deal with a variety of outputs from modules done by my groupmates, certain requirements were needed to orchestrate a jumble of modules into one fully functioning player. I will talk about some design constraints I had to deal with to make Smartify Player possible.

First thing we had to decide on the player was to decide how we would be sending messages from one device to another (when listening from another player or controller). After talking with Karunesh, in charge of MQTT, we have decided to send a json string formatted like below:

```
Send {"command": "INPUTSONG", "songname": "IDGAF", "artistname": "Dua Lipa", "songtime": 3319} to topic /ECE180DA/Team9/
```

Side Note: we are not sending the “emotion” of the song since emotion of the song can be different for each user.

However, there was the issue with there being two different devices with different directories (and different songs). How do we find where the song is on our device given a song name and artist name? The solution to the “different directories'' problem was to use a dataframe (which

stores the data as csv). To find a song in a new device, we would just need to start a query in the local machine (as long as the .csv or the directories were loaded).

```
path,artist,title,emotion
C:\Users\Hyounjun\Desktop\UCLA\EE180D\Smartify\drive-download-20201211T001434Z-001\Angry - IDGAF.mp3,Dua Lipa,IDGAF,0
C:\Users\Hyounjun\Desktop\UCLA\EE180D\Smartify\drive-download-20201211T001434Z-001\Disgusted - Boys Ain_t Shit.mp3,SAYGRACE,Boys Ain't Shit,1
C:\Users\Hyounjun\Desktop\UCLA\EE180D\Smartify\drive-download-20201211T001434Z-001\Fearful - Not Afraid.mp3,Eminem,Not Afraid,2
C:\Users\Hyounjun\Desktop\UCLA\EE180D\Smartify\drive-download-20201211T001434Z-001\Happy - Watermelon Sugar.mp3,Harry Styles,Watermelon Sugar,3
C:\Users\Hyounjun\Desktop\UCLA\EE180D\Smartify\drive-download-20201211T001434Z-001\Neutral - Rain.mp3,Lucy Park,Rain,4
C:\Users\Hyounjun\Desktop\UCLA\EE180D\Smartify\drive-download-20201211T001434Z-001\Sad - Say Something.mp3,A Great Big World,Say Something,5
C:\Users\Hyounjun\Desktop\UCLA\EE180D\Smartify\drive-download-20201211T001434Z-001\Surprised - It Wasn_t Me.mp3,Shaggy,It Wasn't Me,6
```

.csv file used for the Demo (“path” is different for each devices)
path (location of music file), artist name, title of song, and emotion related to the song are stored in .csv files

```python
#Python has no switch statements, I could use a dict, but we can talk about this later
if command == "INPUTSONG":
    self.play_song(songname, artist=artistname, start_time=int(songtime))
elif command == "PLAY":
    self.play()
elif command == "PAUSE":
    self.pause()
elif command == "TOGGLE":
    self.play_pause_music()
elif command == "SKIPTIME":
    self.skip_time(int(songtime)*1000) #time to skip is in ms
elif command == "NEXT":
    self.next_song()
elif command == "PREV":
    self.previous_song()
else: #command not recognized
    print("Command not Recognized!")
```

Currently accepted commands in the Music Player - More could be added in the future.

The result - after receiving commands via MQTT and parsing - is the same as what was demonstrated in the project demo; we were able to play the song remotely in just mere seconds even on different directories.

To solve the last problem of “can’t find songs in another directory”, I had to add a feature to lookup songs online via Youtube-dl. Since we have the song name and artist name that we want to look up (received from MQTT or from voice command), we simply need to pass it onto a youtube query; it will fetch the audio stream which the vlc-player can play. On a successful youtube-query and internet connection, the command line will show something similar to below. Some audio errors (actually warnings) will appear since we will need to fetch the audio stream online (which is slower than playing music from a local directory).

```
Unrecognized input. Please enter your voice command again.
We think you said: play polaris by aimer
Got it. We're on it now.
{'command': 'INPUTSONG', 'songname': 'polaris', 'artistname': 'aimer', 'songtime': 0}
[00000260ca4a5b60] cache_read stream error: cannot pre fill buffer
[00000260ca4dfde0] mjpeg demux error: cannot peek
```

On a successful attempt at online music look-up

We are still using information from the command format (same one we send via MQTT) to decide what to query on youtube. However, on some instances, we had issues accessing audio stream, due to copyright issues like this one with Back in the Black by AC/DC

```
We think you said: play back in the black by acdc
Got it. We're on it now.
{'command': 'INPUTSONG', 'songname': 'back in the black', 'artistname': 'acdc', 'songtime': 0}
ERROR: pAgnJDJN4VA: YouTube said: Unable to extract video data
Error occured playing song online; Possible copyright issues
```

On a failed attempt at online music look-up

Youtube-dl is not able to extract the audio stream for some songs due to copyright issues. Our group does not plan to circumvent this, and we simply exit. This may adversely affect the user experience if the listener does not have a song and the song is copyrighted on Youtube. The system will attempt to query the song when a message is received (and most likely will continuously fail), but it will not crash.

For the Emotion-based Playlist, we simply find all songs associated with the same emotion as the EmotionDetection unit.. Since making the player identify the emotion was time consuming (it would involve signal processing on all songs), the player was designed to take user-input on the .csv file that could be imported. If a user does not specify an input, emotion of the song is assigned **randomly**.

The current algorithm simply finds songs with matching emotions, as seen by code below.

```
def play_emotion_playlist(self, num_songs=20):
    """
    Creates a random playlist of a song matching the emotions
    With songs with matching emotion
    If matching songs < num_songs, playlist will contain all matching songs
    """

    emotion_playlist = self.df_songs.find_emotion_songs(self.emotion)
    random.shuffle(emotion_playlist)

    if len(emotion_playlist) > num_songs:
        emotion_playlist = emotion_playlist[:num_songs]

    if len(emotion_playlist) == 0:
        print("No songs matching your current emotion! Try adding more songs!")

    self.player.addPlaylist(emotion_playlist)
    self.player.play()
```

To add randomness for the suggestions, the playlist only consists of 20 songs. This will change the playlist each time (so the playlist isn't the same every time). More improvements could be made to the suggestion algorithm with more data and signal processing in the future. The algorithm was simplified due to the fact that we did not decide to use a signal processing model of music analysis (which would have complicated the player on the developmental end).

User Testing/Feedback

To test Smartify, we asked Team 8 about setup difficulties, and intuitiveness of our GUI.

Team 8 had some issues with the download of the libraries and basic setup of the system. Once Nate sent us his Conda environment and some log files of the errors he was getting, we were able to deduce which libraries were causing problems with setup.

As a result, we were able to create a conda environment for both Windows and Mac that could run our Smartify Player. (PyAudio needed to be manually installed due to pip not finding the most recent version, and this was noted in our Github repo)

Later on during the project, we asked Team 8 about how intuitive our GUI was, and whether they were able to use some of the “smart” functionalities of the player. They reached out and initially said that the user manual did not explain that system very well. After receiving that feedback, the user manual was edited to clarify how to use some features on the player.

After some revisions, Team 8 did not have a hard time using audio commands, or listen to another’s “music channel”. Some more complicated commands like emotion-based playlist were not thoroughly tested, however. While using our player, Team 8 has reported some errors that were from some of our libraries, so we revised our code to gracefully transition on the library failures such as failed Youtube queries or copyright issues.

Results and Conclusion

Overall, we are very satisfied with our project. We have finished most of the tasks we planned out fall quarter and early winter quarter. We made the Smartify Player much more stable and user friendly during the second quarter. We made improvements after feedback from Team 8 for better UI/UX. Our schedule mostly went according to plan, with just a few hiccups along the way. After 20 weeks, our final end-product is free, cross-platform supported, and running on any device that can run python and our libraries.

There are many features we could still add, such as a song-statistic based suggestion algorithm, custom user playlists, or suggesting additional songs based upon Spotify API. We did not end up fully finishing the “Emotion Playlist Generation with Signal Processing”, but we were able to find a work-around for the time being (user input of emotions through .csv file). However, if we were to list every feature we could have added or things we could have improved upon, the list would go on for a long time.

We have chosen our system design and libraries with project scalability in our mind. Most of the features that we can think of can be added without changing the design of our player (except perhaps maybe GUI redesign). We found this project to be a great learning experience, and we are still satisfied about finishing the main core features of Smartify, just like our original plans.

References and Materials

Emotion Detection

1. Emotion Songs Library with Emotion Mapping CSV File

https://drive.google.com/drive/folders/1y39AhN18e-TeKhVm2jMLDValoCtHx5-3?usp=sharing

2. Emotion Detection CNN Model
   https://drive.google.com/file/d/1rJ4bVUzoUg3HG5RPyLTOotPzl5SxYSZz/view?usp=sharing
3. Emotion Detection Training Data
   https://drive.google.com/file/d/1X60B-uR3NtqPd4oosdotpbDgy8KOfUdr/view?usp=sharing
4. Code Reference
   https://github.com/atulapra/Emotion-detection

Music Player

Most of the code was self-written, but GUI was influenced by:

1. Tkinter and VLC Demo: https://forum.videolan.org/viewtopic.php?f=5&t=128595

MQTT

The prototype MQTT code followed the structure of:
https://github.com/pholur/180D_sample

Voice Recognition

To setup the python speech recognition library, the following was used:
https://github.com/Uberi/speech_recognition

Appendix

Smartify Demo:
https://drive.google.com/file/d/1FcLh_3RdWc11luPGlq_0B6kOkrFysZDV/view?usp=sharing

Smartify Gesture Control Demo: (our IMUs broke or went missing :/)
https://drive.google.com/file/d/19oV5f7zHHLx1okzwKKpX6sJfB-XTfCzk/view?usp=sharing

Team 8 Demo:
https://drive.google.com/file/d/1-jwMs8vKmWh_DcMST5sI8EL4JvZxTfUq/view?usp=sharing