Personal and Small Group Music Suggestion Application – Final Paper (December 2020)

Joseph Hackman, Sean Tillman, William Smith, Eli Ruckle

***Abstract*— The purpose of this project is to create an application that is capable of making effective and accurate music suggestions. While many applications can make music suggestions, we plan on implementing a unique combination of features including extracting features from audio of the user's environment, matching with accurate music metadata, and integrating local music libraries. While many streaming services offer music suggestions, local music suggestions have fallen flat. An application that incorporates these features on a mobile device would allow users to have a better experience with their music, adjusting itself to surroundings and allowing the user to focus on their experience. Our application utilizes the phone's accelerometer to update the selection of songs based on the user's movement.**

I. Introduction

THERE is a widespread collection of information about music listening habits, song and artist preferences, and music suggestion concepts built into commercial streaming services. For example, Spotify suggests playlists based on your previous listening, as well as having a multitude of playlists sorted by genre. However, there is a lack of software that accepts an input of a device's surroundings, such as movement and extraneous sounds. These could factor into an individual's choice of song, album, artist, playlist, etc. which a device such as a mobile phone has the hardware capabilities to gather. We believe that incorporating such additional parameters could result in better music suggestions and smoother listening experience, and in the case of our application, accelerometer data.

II. Motivation and problem statement

Since much of music can be classified into a variety of moods and genres, it is intuitive to develop a product that can make use of these classifications while considering factors that can be gathered by the device. The device's accelerometer could gather information about how a person is moving; if they are still or being active. This information could then influence music suggestions, ideally leading to more accurate suggestions. While most commercial streaming platforms have some form of feature calculation and suggestion methods, they do not have any contextual elements to them. They only use the information that can be obtained from the audio content of the song. Programs that operate with local music libraries lack any suggestion method and are also not context-aware. Our methods would be tested on and apply to local libraries. This would allow the user to listen to their library in a context-aware manner, based on their movement.

III. Applications

This would be primarily useful for individuals or small groups that want to focus more on their activities than providing a soundtrack. There could be several cases where this may prove useful. For example, if a group of people were running around or engaged in physical activity and the pace were to change from vigorous to a state of relative rest, the music on a device would still be playing loud and energetic music which could cause some to become uncomfortable. The application would detect this change, and adjust the song selection accordingly, better matching the current state of activity. Also, if a person were doing a sedentary activity, the accelerometer would also be helpful. Since no major changes would be detected, the playlist would remain consistent, instead of inserting a song that may disrupt the user. The application could also be applied to larger gatherings, but for this particular project, we focused on more personal settings.

IV. Related Works

Several applications were interacted with in order to determine how similar products and applications behave, what features they have that might be useful to include, and any issues that might hinder their functionality of their product or potentially our application. The observed programs were Last.fm, MusicBrainz & AcousticBrainz, and Fit Radio. These were chosen with two purposes in mind: to learn what works well from a metadata standpoint, and what practical utilization can or has been made with the accelerometer in mobile devices.

Last.fm is a website and desktop application that aggregates song and music metadata from across multiple sources, creating a platform for song suggestion, information tracking, and song playback. This was observed for the purpose of gaining insight into how metadata for local libraries could be obtained. During observation, the website had suggested a Schumann piece and given that a Grieg piano piece was in playback locally, that suggestion seems within the realm of possibilities. One unique feature of Last.fm was that it could gather information from a variety of playback applications, including both local libraries and commercial streaming platforms such as Spotify. It accomplishes this as an application that can be installed locally

This project proposal is for Group 2 of Project Studio MUSI-3770, submitted on September 10, 2020.

Authors:
Hackman, Joseph; Tillman, Sean; Smith, William; Ruckle, Eli

on a device and is linked to – presumably – the same user's Spotify account. Testing took place in the web application, and it played songs it identified to be local through YouTube. This proved to be inaccurate for this version of the website; "In the Stone" by Earth, Wind & Fire was selected, but Last.fm played something completely different. While the integration used in Last.fm may not be ideal for our situation, we saw how mismatches in metadata could be problematic and rather than playing through YouTube or another service, we chose to play the songs directly from the local library.

Fit Radio is a mobile application geared towards individuals looking to improve their song selection during workouts and activity. Upon opening the application for the first time, it asked several questions that might be made to aid in the selection and suggestion of music. Some of those questions gathered gender, one of three modes for the app (The answer "I'd like to run to the beat" was chosen to increase the likelihood of utilizing the accelerometer.), and what activities in which the user would most likely engage. "Cycling", "walking" and "running" were selected to increase the likelihood of utilizing the accelerometer. Although a questionnaire may be helpful for establishing some constants about the user, its scope could be limited in order to prevent pigeon-holing the application into only being used during certain times. A unique feature about this application was that it tapped into accelerometer data to select music based on the level of activity in which a user is engaged. The application has a wide library of popular music available which are divided by genre, era, mood, ideal activity, and more. There are also "DJ pages", which have evolving playlists that play certain styles of music. For example, in the classic rock DJ page, there was one DJ that featured "high energy" songs ranging from KISS, Dire Straits, ABBA, and more, with the important distinction that the songs were over 120 beats per minute. While this application provides useful examples of what can be accomplished, optimizing itself for fitness has limited it for general use and periods of rest.

Ultimately, we aimed for our application to include and build upon useful features in the observed products such as clear categorization, efficient use of mobile device hardware, and local library integration. We also aimed to avoid limiting the application to only being used for a specific set of activities or simply not playing the right music. Correctly balancing hardware input and matching local libraries to online metadata prevented these issues.

V. Goals and Research Statement

The goal for our final product was to make a mobile application that would take in parameters and other inputs from the user and select songs from the user's personal library accordingly. Initially our goal for the inputs was to use recorded audio of the user's environment, but eventually transitioned into using motion data from the accelerometer. Parameters for the music selection would draw from questions asked to the user about what they were doing, what sort of genres they would like to hear, what sort of mood they were in, and what speed (tempo) of music they would like. Since the songs did not have this information available in the file metadata, the songs must first be processed through MusicBrainz and AcousticBrainz to extract any missing title or artist information, and the low-level & high-level data that contain mood, genre, etc. This information is formatted and written in a text file. The formatting to the text file allows the information to be read easily by an algorithm which determines what songs should be played. This algorithm takes the user input from the UI and filters the songs that best match that criteria. This algorithm is also passed accelerometer data, which is used to measure user movement. Once the algorithm returned the list, they would be passed into the device's library and played back through the device's built-in media player. The playlist would also be able to adjust according to sudden changes in the user's movement, and could also be changed if the user updated their preferred parameters.

VI. Proposed Method

A. Tasks and Timeline

In order to complete tasks efficiently and stay on schedule, we applied the principles of Scrum. Communication is critical to having a well-functioning team; therefore, we met at least twice a week. During these meetings, the team divided tasks, discussed any issues a member may be encountering, built upon ideas that may need clarification, etc. Notes of all major meetings were scribed for the purpose of task management and to document any project changes. Another key principle of Scrum that we employed is breaking down large tasks and logically solving them. In a similar manner to breaking down our goals, the tasks followed a path from the beginning of the application to the end; starting with input data and ending with effective suggestions and playback. We did find it useful to start on either end of the project, resulting in more specialized perspectives for each team member. Once smaller tasks were complete, we began to merge teams based on how our programs interacted with each other. By implementing these systems, we aim to become an effective team.

For our implementation, we have a procedure that must be completed on a desktop system before the mobile system can function as intended. This procedure gathers all the low-level and high-level data and generates a text file that the application will use to filter and choose songs. We utilize the data available on MusicBrainz and AcousticBrainz to build this file. Since MusicBrainz IDs are unique to MusicBrianz & AcousticBrainz and are the unique identifier with which correct recording can be accessed, it is important that the correct MBID can be found. A process that is crucial to obtaining MBIDs is fingerprinting, which creates a unique identifier that can be passed to MusicBrainz and return not only a MBID, but also populate the data for an audio file which has empty fields. To accomplish this, we utilized a tool called Chromaprint (via the package pyacoustid) provided by AcoustID, which is the same open source project as MusicBrainz and AcousticBrainz. Chromaprint obtains the fingerprint for the song and compares it to the MusicBrainz database and returns a list of MBID matches. In the extraction process, Chromaprint reads through the audio files that are provided and returns MBIDs, which can be used to gather additional information about a recording. These MBIDs are used to query AcousticBrainz and retrieve the needed data. Since not all the data is necessary for our application, we include data that our application uses such as

mood & genre probabilities, and exclude data that would be unused such as the key of the song & beat count. This allows faster write times, easier debugging, and quicker access within the application. Once the utilized data has been gathered and sorted, the text file is written and ready to be used in the application.

The mapping algorithm is a step by step filtering process. It begins by reading information from a text file and restructuring the data into multiple arrays for each data type. Once the library is read into the algorithm, the filters can sort based on the user inputs. The first filter sorts the songs based on the moods chosen by the user. Mood types come in pairs: Aggressive vs Relaxed, Electronic vs Acoustic, Happy vs Sad, and Party vs Chill. Each song has AcousticBrainz data for each mood in each pair and the filter will consider the most significant probability to determine the most representative mood categorization. For example, if a song has an Aggressive probability of 90% and a Relaxed probability of 60%, then the song will be matched with the Aggressive value, since the probability is higher. Then each song's mood value will be normalized to a range of 0-1. From there, moods will be relabeled based on the midpoint of this range. This allows the mood categorization to be relative to the playlist. For example, once the Aggressive value of each song is determined, all values are normalized from 0-1. Songs above the midpoint will be labeled as Aggressive and songs below the midpoint will be labeled as Relaxed. Once each song is labeled with a mood profile, such as ["Aggressive", "Acoustic", "Sad", "Chill"], the profiles are then compared to the user input. If songs meet the exact criteria of the input, they are chosen to be considered by the following filters. In some cases, few songs meet the criteria, so a fallback function will then compare mood profiles that match three out of four of the mood criteria to guarantee a significant output length.

The following filters are less complex in design, however they significantly narrow down the song selection to meet the criteria of the input. The filters are in this order: tempo, RMS, and genre. The tempo filter receives the selected songs from the mood filter. It compares song tempos to the input tempo; if the song tempo is within a range of the input, then it is passed to the RMS filter. The RMS values of all songs are normalized to a range of 0 - 1. Depending on the activity input, the accelerometer input operates on a certain range. For example, if the activity is "Working Out", the accelerometer input will range from 0.8 - 1. Or if the activity is "Studying", it will range from 0 - 0.2. This means, in the case of the "Working Out" activity, a low accelerometer input will yield an output of songs whose RMS values are around 0.8 in the normalized 0 - 1 range and a high input will output songs with values around 1 in the range. Similarly, for the "Studying" activity, a low accelerometer reading will output songs with RMS values around 0 and a high reading will output songs with value around .2. These songs are passed to the final filter, the genre filter. This takes each song's genre and compares it to see if it is in the list of selected genres. If a song's genre is in that list, then it is added to the output list. In rare cases, the size of the output may be too small. A fallback algorithm will ignore the final filter to ensure a sizable output size. If that filter still has a small length, then it will output from the filter before.

The two biggest tasks on the application side of development was the UI creation and data input. Sean worked on most of these tasks, with Eli helping experiment with the audio and making the initial accelerometer functions. The UI for the application is laid out in a tabbed view format, with respective pages for the parameter selection, playback controls, and extra information and instruction. Each page contains the application name and the bright, clean feel of the application. The preset page was our first page (Figure 1) and was drawn up extremely early in the building process, as we needed to figure out the metadata side of things before creating the parameters. The first thing featured was an activity selector, followed by a list of genres of which multiple could be selected. The next section was the mood selection, which was designed as a way to pick between two options for four different categories by using a slider to "point" at which mood the user was feeling more. Below that was a tempo slider; we made it a slider so that those unfamiliar with concepts of tempo would have an easy visual reference for speed. We also included the BPM below for those more musically inclined. Below that was the button to process the selected options. When pressed, the application will first collect a small sample of accelerometer data, then "export" the states of each selection method to the data class which would then be passed into the mapping algorithm. This mapping algorithm would return the list of songs that were most fitting, which would then be queued by the built-in music player. The button would update its text to inform the user of the progress. As seen in Figure 1, it starts by saying "Set my choices!". After being pressed, it displays "Processing…" while analyzing accelerometer data and then "Done!" once it has processed the mapping. The button then changes to "Update my choices!" so that the user is easily aware of how to change the settings they entered. For the playback menu (Figure 2), there is of course the standard play/pause, forward, and backward buttons that almost every single media player uses. They are oriented towards the button and are large and defining so that the user has no confusion of how to skip a song they don't want to hear. There is also a very simple text area to display the name and artist of the current piece that is playing. Not pictured in any figure is a very simple "about" page, which explains the app's purpose, basic instructions, and direction to our team's GitHub page.

The other task on the application side of development was our input of motion data. The accelerometer data provided by an Apple device can be processed in many different ways, but we used the "UserAcceleration" attribute, which focused purely on the movement of the device without factoring in direction. The data was used in two main places: at the initialization of a playlist, and in a loop to check for changes. For the initialization, a specific timer function is run when the user presses the button at the bottom of the presets page. This timer method contains a large timer that looks at incoming accelerometer data for 4 seconds. There is also a small timer which takes the specific acceleration magnitude every 200 milliseconds and appends it to an array. At the end of the large timer, the max value of the appended array is chosen as the value to be passed on to the mapping function. This function is called every time the user sets or resets their parameters and is very quick as to not take too long for the user. The other function of the accelerometer is run constantly throughout the entire usage of the application to check for major changes in magnitude. After initialization, a one-minute timer is looped

that checks the magnitude in 10 second chunks. If within these 10 second intervals a large change in magnitude over the value processed by the mapping algorithm is detected, then the function triggers an immediate recalling of the function to accommodate for the change. Otherwise, the function will look again in another minute. The function will not trigger if the user has just recently triggered the manual evaluation method of repressing the button. You can see the exact process of this monitoring function in Figure 3, which displays the values in the console. The top value was the initial set magnitude for this instance. After one minute, the second value was found, but it was determined to be not a large enough distance to recall anything. After another minute, the final value was recorded, which was different enough from the original to warrant a reanalysis of the mapping function.

B. Roles

Since we had many tasks to cover in this project, multiple roles were required to effectively divide work. Each role was not necessarily be filled by a single person, rather we had a member lead efforts in that particular section. Over the course of the project, each member wore each hat at some point.

1) *App Developer*
   The app developer was the primary Swift coder and was in charge of the large-scale user interactions. This concerned the main uses of the app itself UI design, track selection, pre-selection inputs, and accelerometer data. This role is primarily concerned with front-end functionality and user experience.
2) *API Implementation*
   The API implementer researched and created a script and txt file which contained any necessary information from MusicBrainz & AcousticBrainz. This included the extraction of metadata of all songs in the user's local library from these databases.
3) *Algorithm Developer*
   The algorithm developer was the link between the UI of the application and information stored in the text file. This role ensured that the data would be read in correctly and was able to return appropriate song suggestions based on the information from the pre-selection process and accelerometer.

C. Challenges

One of our biggest challenges for development of the application itself was getting used to the coding itself. None of our team had coded in Swift or Objective C before this project, and we were all forced to get acquainted with the style of the language quickly. This, combined with Apple's lack-luster documentation on swift development, made it hard to solve bugs and errors. As an example, most UI organizer objects such as a VStack (vertical stack) have a built-in limit to the amount of UI elements that can be held within. However, XCode's error messages do not indicate this, and we were only able to figure out this was why we were getting errors through consulting outside sources.

Another challenge that ended up being a major turning point in our development was making our round recording feature work. From the start we had many issues with recording background audio to be analyzed. Recording itself could be accomplished, but Apple's extremely tight privacy measures prevented us from accessing the recordings. We eventually elected to try using audio buffers instead but ran into a bigger problem soon after. We were made aware some time into our progress that the audio may not record during the same time as the built-in Apple playback, and this was confirmed after some testing. We theorized ways to get around this issue, by perhaps letting the user select when they would want to record, or stopping the music between songs to record a short audio sample. Eventually we decided to stop using the audio recording feature as a whole and focus on purely using the accelerometer as means for live updating. Given more time, and perhaps with prior knowledge of Swift development, we would've more heavily pursued the idea of audio input and we would hope in the future that with future development of the application this would be implemented.

The last major challenge on the application side was that of the music player. Apple's MediaPlayer Framework is horribly under-documented from a newcomer's perspective, and that meant that several issues took a long time to overcome. Simply testing code was the music player was incredibly hard at first, since the built-in simulator for XCode does not contain a music library. We therefore had to run it on an actual phone for the remainder of our project, however only Sean's phone and computer had the recent operating systems needed to run the code required. Like everything else in Swift, the MediaPlayer framework was also extremely buggy, with one error coming extremely late in the process coming with the queueing process. Lots of searching on developer sites and web for RMS lead to no solutions to this problem, which was troublesome for so late in our process. We eventually rewrote the code to try to limit the bug, but up until the writing of this paper it has still caused issues from time to time and is something we'd like to figure out even past the deadline, just to make the overall product smoother.

One particular challenge that arose with the extraction process was the fact that a given song could have multiple MBID matches. This is due to the fact that there are different versions of that song that could exist, ranging from live recordings to bootlegs to demos. Sometimes, these versions that do not match the local file may end up being the top result returned by MusicBrainz sorted by a match percentage. These differing versions – both in MusicBrainz and AcousticBrainz – usually lack much of the needed metadata and information needed for the application. For example, a local file may be a studio recording of a song, but the first returned MBID is a bootleg of the song. While this will return a MBID, using that MBID to query AcousticBrainz will result in empty fields for the low-level and high-level data since it is not present. Since the information queried from AcousticBrainz is reliant upon the proper MBID, it is crucial that the most likely MBID that is returned is a match to the local file. To solve this, rather than taking the first MBID, a list of MBIDs is retrieved for each song, and each MBID is used to query AcousticBrainz. The first MBID to return valid AcousticBrainz data is the data which the application uses for the song.

One of the issues that made reading the text file into Swift difficult was the formatting. String formatting for both Swift and Python are quite different, so reading lines into Swift directly would cause an error without proper formatting

adjustments. By default, python surrounds strings with single quotes, which is not an acceptable format for Swift. To avoid this issue, lines had to be manually adjusted with a replacingOccurrences object to replace the quotes as well as remove extra characters, such as \n. Since lines are read as a string, Swift could not directly interpret the string arrays as actual arrays. At first, the strings were split by ", " to isolate each element of the string array to append to the real array, but this caused many issues. Some songs have commas in them, so song names would be split into parts, causing a longer song length and creating mismatches in title/artist pairs. To ensure that every title, artist, and song information values were left exact, we tagged each element inside of a bracket: <# element #>. By splitting the elements around the tags, the original element value was left exactly the same when translated to an actual Swift array.

The algorithm sorted moods and other filters relative to the information of the song library. This would ensure that songs would be output regardless of any input that doesn't exactly meet the AcousticBrainz data. For example, if all songs were interpreted as "happy" by AcousticBrainz, then based on the AcousticBrainz data, no songs would be output if the user selected a "sad" output. By tailoring the filter to the context of the library, then songs with lower "happy" probabilities would be output for the "sad" selection. This helped guarantee that songs would be selected regardless of the user input, however certain settings would still render an output with very few songs. To prevent a small playlist, we implemented filters that would select songs that may not exactly meet the criteria of the input, but would come close. For example, if the mood filter could only match a small number of songs that meet the criteria for the input, then it would try to select songs that meet three out of four mood parameters. Although uncommon, the final output could still be fairly short even when the mood filter generates a large selection. To ensure output length, we used another fallback algorithm for the filters following the mood sorting. If the last filter generates too few selections, then the filter would be ignored and the final output would be the selection of the second-to-last filter, and so on.

The final issue of the mapping algorithm was that it would generate a similar playlist every time it was run (given that the UI input parameters are the same), however we wanted to prevent the same song from playing twice within a given amount of time. We needed to figure out a way to prevent the same song from being generated twice. To fix this, we put the mapping code into two main functions: filter function and file reading function. When the file reading function is read, it outputs song information into nine arrays for each info type. When the filter function is called, it removes the indices of the chosen songs from the arrays, reducing the number of songs that are left to be chosen. If the remaining song list gets too low, the filter function will call the file function to restock the preselection playlist.

VII. Proposed Measure of Success

Our initial goal of this app is to return songs based on settings and movement input, but there are a few things we were looking for more specifically. We wanted our mappings to not only give us the most fitting songs, but also give us additional songs if there weren't many that satisfied the user's inputs. We also wanted to make sure that the mapping would not constantly repeat the same songs given a likewise parameter input, because a user would likely not want to listen to the same three songs or so over and over again. When testing, we used songs from many different genres and would sometimes get crossovers where they wouldn't seem likely. We were okay with this however, since that is mainly due to Accousticbrainz's data and therefore did not factor into our methods of success. Another thing that we looked for when running our app was to make sure that a large change in accelerometer data would trigger a change in the songs. We wanted to make sure if the user gets up to do something else, the music will change even if they do not trigger it themselves. Finally, we wanted to make sure that the playback was smooth for a user experience, and we wanted to make sure it was not a complicated task for the user to activate playback. We set up alerts that help direct a confused user and made the playback UI as normal as we could to help make sure a user could easily navigate and operate the app. Overall, we of course wanted to limit bugs and glitches from interrupting the user experience, and we ran countless tests to try and identify and fix all known bugs. Crucial to our testing was trying to make the app break by doing things that a user would normally not do to see how our app would hold up.

VIII. Resources

A. Tools

The initial code for the mapping algorithm was done with Python in Jupyter Notebook. This allowed us to experiment with a language we were already familiar with and the ability to run individual cells allowed for the manipulation of individual chunks of code at a time. If we had created the initial algorithm in Swift, it would have been much more difficult to effectively test its functionality and reliability because we would have had to accommodate our lack of Swift knowledge simultaneously. By having the whole algorithm already completed in Python, we just needed to restructure the code based on the new format in Swift. Additional changes were made while the algorithm was implemented in Swift, however by this point our knowledge of Swift was enough to ensure that these changes would not be too complicated.

For the application side of our project, the main tool used was XCode, Apple's default IDE for iOS application development. XCode's built in Simulator was used for basic UI testing, however for full application testing we had to use an actual iPhone, as it was the only way to receive accurate accelerometer data and a built-in music library application.

For code management, we continued to use GitHub throughout the semester. GitHub was especially important to Sean's work on the application side to keep track of changes made over time, as well as merging in other parts of the project into the application.

B. Languages

The data extraction process was written in Python. The ability to make changes quickly and easily made it ideal for

testing scripts and writing the information file which would eventually be in the application.

The application itself was composed entirely in Swift. It proved to be extremely buggy and hard to understand due to poor documentation but was a well-designed and convenient language for application development.

C. *Libraries and Frameworks*

PyacoustID is a Python tool provided by AcoustID which allows metadata to be obtained based on the audio content of a file through fingerprinting. This process returns metadata which can be populated in the file information, as well as a unique MusicBrainz ID (MBID), which can be used to query the MusicBrainz & AcousticBrainz Web APIs.

MusicBrainz & AcousticBrainz were particularly crucial in obtaining data about the local library. Once the MBIDs were obtained, they were then used to query AcousticBrainz to retrieve the low-level & high-level information. This contained data such as mood probabilities, genre probabilities, BPM, RMS values, song title, artist name, and other data that would be used in the app. Since MusicBrainz, AcousticBrainz, and AcoustID are all part of the same network, the data coming from each of those sources largely matched.

Several Swift libraries and frameworks were used for the application itself. We used the relatively new SwiftUI kit to construct the UI, rather than the old and less intuitive UIKit. For accelerometer input collection we used the built in CoreMotion library to draw information from the device. Finally, we added Apple's MediaPlayer framework so that we could access and playback the media stored in the library of the phone. All of these libraries were provided by Apple through XCode - the MediaPlayer framework had to be added to the project manually - and no third-party products were needed on the application side.

IX. Deliverables

Our main product contains two portions: the python script used to generate the text file, and the swift project that contains the app itself. The python script is meant to run on a desktop system, which will generate the text file that can then be transferred to the application. The Swift project contains several key components, which are the selection algorithm, code to capture the accelerometer data, and a UI system which will keep track of the user pre-selection. When combined, this will allow the application to generate a playlist of songs for the user depending on their mood and accelerometer data. For our purposes, generating smooth, effective local playback is key to satisfying the user and having a successful application.

X. Novelty

One aspect that is novel is the fact that our application incorporates accelerometer inputs from a mobile device combined with user pre-selection to make music suggestions, whereas other applications have only used accelerometer data. The applications that have done that tend to focus on fitness and movement, unlike our application which could ideally apply to any movement scenario. Another unique aspect our application possesses is well-functioning local library inclusion. Local library and streaming library integration are present in Last.fm, but playback is limited and confusing. Since we applied consistent metadata from MusicBrainz & AcousticBrainz, we ensure accurate matches to existing files. Our application aimed to combine the aspects of hardware input, previous interactions, existing metadata, and more, to provide a smooth and easy experience for the user.

XI. References

1.AcousticBrainz: (n.d.). Retrieved December 01, 2020, from https://acousticbrainz.org/

2. AcoustID: (n.d.). Retrieved December 01, 2020, from https://acoustid.org/

3. Last.fm: (n.d.). Retrieved December 01, 2020, from https://www.last.fm/

4. MusicBrainz: (n.d.). Retrieved December 01, 2020, from https://musicbrainz.org/

5. Fit Radio Workout: (n.d.). Retrieved December 01, 2020, from https://play.google.com/store/apps/details?id=com.fitradio

XII. Figures

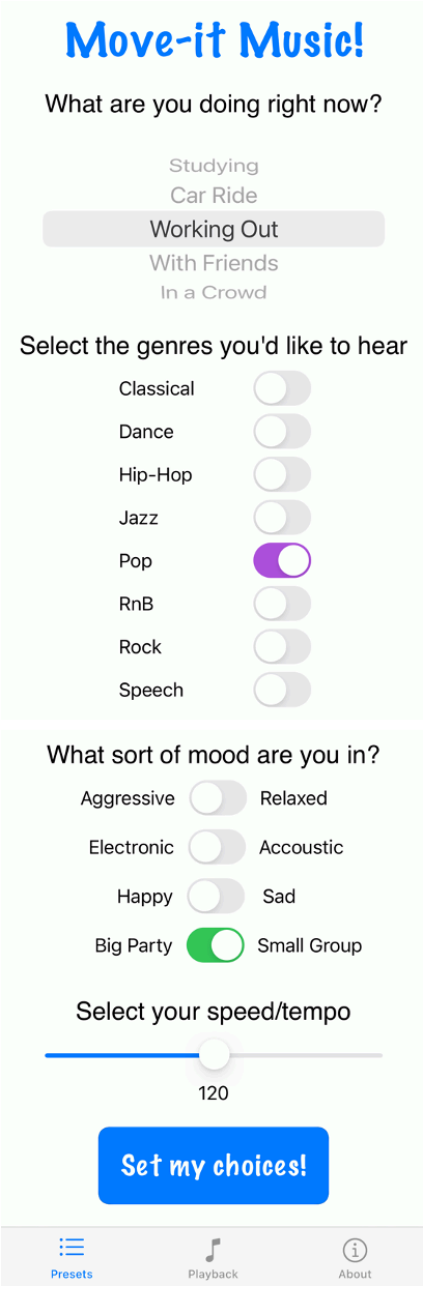

Figure 1. Main page used for user pre-selection process.

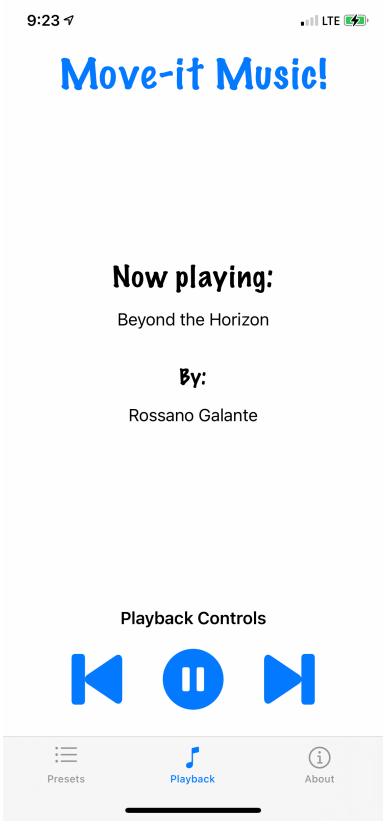

Figure 2. Playback page displays title & artist and has controls for the playback.

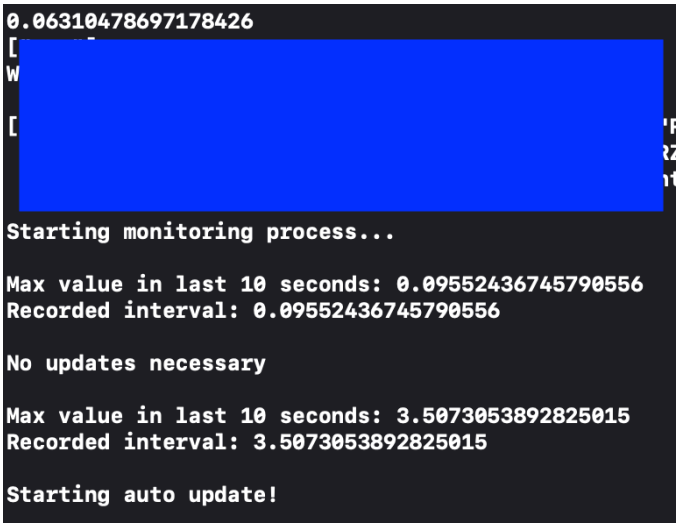

Figure 3. Console display of the accelerometer being measured and automatically updated.