PACE THE MUSIC: ACTIVITY-BASED PLAYLIST GENERATOR

James Gray
University of Victoria
grayj@uvic.ca

Kaileen McCulloch
University of Victoria
kaileenm@uvic.ca

Nick Warwick
University of Victoria
nwarwick@uvic.ca

ABSTRACT

The following design specification outlines our plan to create a physical activity based playlist generation utility. It specifies the goals of our project, and provides a development timeline along with a list of tools required for development. This document also lists the roles of each team member, and provides links to any resources that we use during development.

1. INTRODUCTION

Many people enjoy listening to music while they exercise, and in many cases music itself can motivate the listener to put in even more effort than they would otherwise. [1] However, some types of exercise require strict adherence to a pre-existing training plan - for example, runners and cyclists often train at different paces for different portions of their activity. In these instances, the influence of music on an athlete's pace or performance could be detrimental; a very fast song might subconsciously cause a runner to run at a faster pace than they intended to.

To address this problem, we plan to create a playlist generation utility with a focus on creating playlists for various types of physical activity such as running, cycling, or strength training. The user of this utility will input an activity plan, specifying features such as running pace or step frequency over the course of the activity. Given the plan as input, the playlist generation utility will create a playlist from an existing set of songs, basing the playlist on audio features such as tempo, genre, mood, and subjective energy level.

As an example, a user could create a running plan which specifies that they will run at a slow pace for fifteen minutes, increasing to a medium pace for twenty minutes, fast for ten minutes, and finally cooling down at a slow pace for five minutes. Given this input, the utility would generate a playlist containing songs at a slower tempo for the first fifteen minutes, moving to faster music as the pace of the run increases, following the user-created plan as closely as possible.

2. RELATED WORK

Playlist generation is a popular topic in the field of MIR. Creating the right playlist to supplement an activity is a complex problem, and automating the process is even more so. There are a wide variety of external factors that can be used for playlist generation. For example, Oliver and Kreger-Stickles created a generator which bases the decision process on the user's physiological response [2]. Similarly, Pauws and Eggen studied the correlation between the environment and the type of music a given user would listen to in that environment [3]. Environment-based playlists are also present in streaming music applications such as Google Play Music and Spotify, which allow users to choose playlists based on their location - for example, in the shower or at a coffee shop. Additional factors and methods in playlist generation include random shuffle, content [4], frequency spectrum, and skipping behaviour [5].

Our idea of generating playlists based on user-specified activity plans is similar to the work done by Masahiro et al. [6] and Chen et al. [7]. Masahiro et al. worked on the development of an automatic music selection system based on a runner's step frequency, and Chen et al. developed a music assisted run trainer which also uses step frequency to slow down or speed up the tempo of the music to match the user's pace. We plan on using step frequency as one of the decision features in our algorithm; however, the step frequency we plan on using will not be a real-time measurement, but rather a calculation based on the user-specified activity plan.

Tempo tracking and audio feature analysis are other key components of our research. Musical tempo has been identified as an important component of music classification [8] [9] [10]. Other audio feature extraction techniques which have been explored are intensity, timbre, rhythm, and mood tracking [11] which may also play a role in our playlist generator's selection algorithm.

3. TIMELINE

The following development timeline outlines the major stages of the project and their estimated completion dates.

1. **Determine goals:** [Complete] During this phase we determined the goals for the end product and what capabilities it should have. We also decided on a platform for the application. (*February 22nd to 23rd*)

2. **Requirements gathering:** [Complete] We developed functional and non-functional requirements for the

project. These outline our functionality goals for the final product. (*February 23rd to 26th*)

3. **Design:** [Complete] We decided on a system layout. After a system layout was determined we began development on the UI and started structuring the backend of the application. (*February 26th to March 1st*)

4. **Prototyping:** [Complete] We are in the progress of completing a rough version of the application with basic functionality. (*March 1st to 21st*)

5. **Testing and revision:** [Complete] Throughout development we have repeatedly tested the prototype to ensure that it is working as intended and that each requirement has been met. Once a final draft has been created, a thorough testing set well be run to ensure no final issues need to be addressed. (*March 17th to April 1st*)

6. **Release:** [Complete] Complete development of the final version of the application. (*April 1st to 5th*)

4. TOOLS AND RESOURCES

The project will require a range of tools and technologies, from audio feature extraction and analysis tools to database utilities, application and UI logic.

4.1 Backend

We intend to program the core application logic with Python. This code will interface between the various layers of the utility, such as the core playlist generation logic, the database, the audio feature extraction logic, and the UI code.

4.2 Database

To create a playlist, the utility will need access to a set of songs. The song files will be stored on disk, and any additional metadata or audio features will be stored alongside the files in a database. We intend to use a MySQL database alongside a Python database interaction utility such as SQLAlchemy [12]. Additionally we intend to build a corpus of song files using Music21 [13].

4.3 Audio Feature Extraction

The audio files themselves will need to be analyzed in order to extract the necessary features from the music, such as BPM information. We plan to use Marsyas and the associated Python bindings to extract this information, which will then be stored in the database in order to refer to it later.

4.4 GUI

Our current plan is to create a standalone desktop utility or application. The GUI for this application will be created using a Python UI library or framework, such as PyJamas [14], PyQT [15], or TkInter [16].

5. PROGRESS REPORT

5.1 Completed Work

As of March 19, we have made significant progress in developing the foundation of our playlist generation utility. Most of the base data model has been completed, with core object classes implemented, and the associated table schemas and relationships have been defined using SQLAlchemy. These classes range from music-related classes, such as Artist, Song, and SongMeta, to activity-related classes, with definitions for ActivityPlan, Pace, and Segment objects. Additionally, basic database setup logic is in place to both create and drop the database and its tables, allowing the utility to be installed and run quickly after downloading. The base layout for the UI has been completed, in terms of where the buttons and widgets will be placed, with some basic functionality in place as well. Some exploration into audio feature extraction using EchoNest has been done, with a proof of concept established using API calls to query the service for tempo and energy information for different songs; however, we decided to eschew the service in favour of using Marsyas to perform feature extraction locally, in order to avoid expensive API calls. Core app configuration logic has been implemented, as well as some basic documentation describing the dependencies and installation instructions.

5.2 Remaining Work

With the base layout of the UI complete, we can start working on adding functionality to the UI. This will involve taking user input to create objects which represent "segments" of a given playlist. These objects will be stored in a list, which will be repeatedly updated as segments are added or removed. This list will then be displayed to the user in the UI. Once functionality for the UI has been finished, we can begin to tie the different parts of the application together by adding interaction between the front-end and back-end of the application. This will involve persisting user actions to the database; for example, if a user removes a segment from a playlist by performing an action on the UI, that segment should, at some point, be removed from the corresponding playlist in the database by committing the changes. Once the functionality has been tied together, we will have a working prototype available for testing. The testing phase will likely uncover bugs that will require minor changes to be implemented. These changes could involve modifying the layout of the UI, as the base version has not been tested with any functionality. Although the back-end of the application has been lightly tested, more extensive testing will be required once the different areas of the application have been tied together.

5.3 Revisions

Once we started working on the project, it quickly became clear what aspects of the original proposal would be suitable and which aspects would need to be changed from the original plan. We specified in the project proposal that we would generate a music dataset using Music21; instead, we decided for simplicity's sake to use a dataset composed of a variety of music from our own libraries. Once we had our dataset, we needed to determine exactly what audio features to extract in order to match the songs up with exercise pace. In order to do this, we tried using EchoNest to classify the music. EchoNest provides a list of many different audio features, such as energy, danceability, and BPM. After trying out EchoNest, we decided to use only BPM and to extract it using Marsyas, for reasons outlined previously. In terms of front-end development, the UI framework is being implemented using PyQT due to it's cross-platform support. In terms of back-end functionality we chose to use SQLite [17] instead of MySQL because it is simpler and does not require the user to install a large database system. MySQL is a good choice on a server-based web application; however, it is not intended for use on applications that run directly on a client's machine. SQLite is much more lightweight and serves our purposes better with regards to keeping the application small. We initially started our back-end implementation using Python 3; however, the front-end development had been started using Python 2.7, so we collectively decided to switch all development to Python 2.7 due to our familiarity with that version of the language. Despite these revisions, we have still met all of our timeline goals, with the exception of the prototype deadline. However, due to the release schedule being extended to April 13th, we intentionally pushed the remaining deadlines back a week to allow ourselves more time to perfect the prototype.

6. FINAL REPORT

6.1 Completed Work

6.1.1 Motivation

As a personal experiment, one day after a long run, Kaileen decided to compare her running cadence against the BPM of her music. While she was out running she noticed that when a fast-paced song that she liked started playing, her cadence would unintentionally change to match the pace of the song. Since she had the playlist readily available after her run, tools to perform BPM estimations, and a heart rate monitor tracking her cadence, she was able to plot the song BPM against her running cadence to see if there was a correlation. Figure 1 shows the outcome of this experiment.

Even at face value a correlation between running cadence and song BPM can be observed. However, to be able to find stronger correlation there are a few other things that need to be taken into consideration. The most important of these is elevation; running cadence up a steep hill is not going to be as fast, regardless of what song the runner is listening to. Secondly, fatigue levels will play an important role as well, because the more tired a runner is, the slower, on average, their cadence will be. Additionally, the runner's emotional response to the song influences cadence - if the runner likes the song, they will be more likely to get excited about it and match the pace.

We believe that this experiment strongly supports the

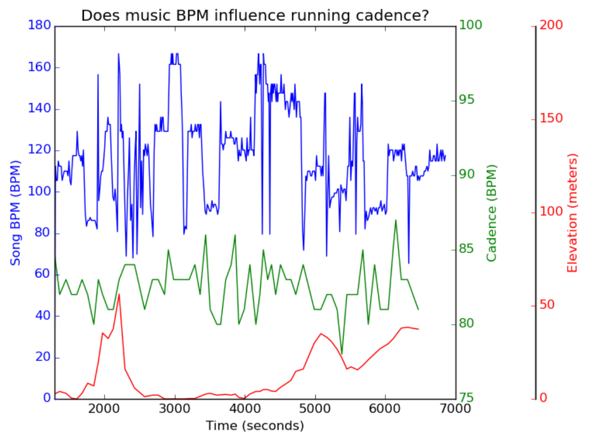


Figure 1: Song BPM plotted again running cadence and elevation over time.

usefulness of this kind of application. As a result, we decided to first implement a playlist generator based on cadence (BPM) alone, and that other audio features could be considered in future iterations of the application.

6.1.2 Feature Extraction/Audio Analysis

For implementation and testing purposes, a dataset of 245 songs was established. Songs in the dataset were chosen in order to cover a wide range of features: short and long songs; fast and slow songs; house, techno, metal, folk, punk, rock, and country genres; and different file formats such as FLAC, WAV, MP4, and M4A.

Each file type stores its metadata differently, so each type needed to be read differently. Once the file was read in, the artist, song title, and song length for each song was extracted. Next, each audio file was converted to WAV format, because Marsyas' Tempo estimation only reads WAV files; finally, we analyzed the average BPM over the full length of each song. The song metadata and the estimated BPM for each song were next stored in a CSV document for the entire dataset. During the setup of the application, the contents of the CSV are imported into the database. In the future, the feature extraction will be done as part of the application itself, once the application supports the ability for a user to choose a folder from which to import songs.

In general, Marsyas' Tempo estimations are relatively accurate, however, BPM estimation is a hard problem so there are certain instances in which the BPM estimation may be inaccurate. Techno and house songs tend to be accurately quantized, and therefore have more regular beats, which results in a high BPM estimation accuracy. On the other hand, songs played by humans on traditional instruments that are prone to human error and temporal inaccuracies - especially those which have unusual time signatures, or a lot of variation in speed - tend to have a much lower BPM estimation accuracy.

6.1.3 Data Model

The data model for the application consists of two main layers: the persistence layer, which consists of the tables and data in a SQLite database, and the data class definitions, which are written in Python using the SQLAlchemy ORM (Object-Relational Mapper). Communication between these layers is accomplished by way of the SQLAlchemy Core, an SQL abstraction toolkit which abstracts away the default Python DBAPI paradigm of interfacing with relational databases. Specifically, we used the Engine and DBSession constructs of the Core to achieve this communication.

6.1.3.1 Persistence Layer

All application data is stored in a single SQLite database file, with the file extension ".db". The persistence layer requires little direct developer intervention when using the SQLAlchemy ORM; data and relationships are defined via Python classes using SQLAlchemy's rich class system, which is then used by SQLAlchemy to automatically generate SQL statements to create the tables in the database. The data model itself is defined exclusively in the data class definitions.

6.1.3.2 Data Class Definitions

Using the SQLAlchemy ORM, we defined the required application objects in Python classes. The objects fit into one of two main categories: "Music" objects and "Activity" objects. The Music category contains five classes: Song, SongMeta, Artist, Playlist, and PlaylistSong. Each class represents a single conceptual entity in our application, with the exception of PlaylistSong, which contains attributes that reference other Playlist and Song objects and represents a many-to-many association table in the database that maps Playlists to their Songs, and vice versa. The Activity category contains three classes: ActivityPlan, Pace and Segment. The Segment class is similar to the PlaylistSong class in function, as there exists a one-to-many mapping of ActivityPlan objects to Segments, and each Segment ties the ActivityPlan to a Pace object. However, the Segment class also contains an additional length field which specifies the length of the segment in seconds. Entity-Relationship diagrams for the two categories can be seen in Figure 2 and Figure 3.

As mentioned earlier, the PlaylistSong class acts as a mapping between Playlist objects and their Songs; a Playlist object alone in the database is fairly simplistic, containing only a name and an ID. However, the Playlist class in the data class definitions contains a significant amount of code for writing out playlist files, as well as the main algorithm for playlist generation.

It should be noted that our application pre-populates the database with a set of four Pace objects, with the names "Slow", "Steady", "Fast", and "Sprint". Similar to Playlists, ActivityPlans simply act as a vessel for Segments, and only have a simple 'name' property of their own.

6.1.4 Playlist Algorithm

At the heart of the application is the playlist generation algorithm. Before running the algorithm, the corpus of songs is divided into four subsets based on the BPM range of

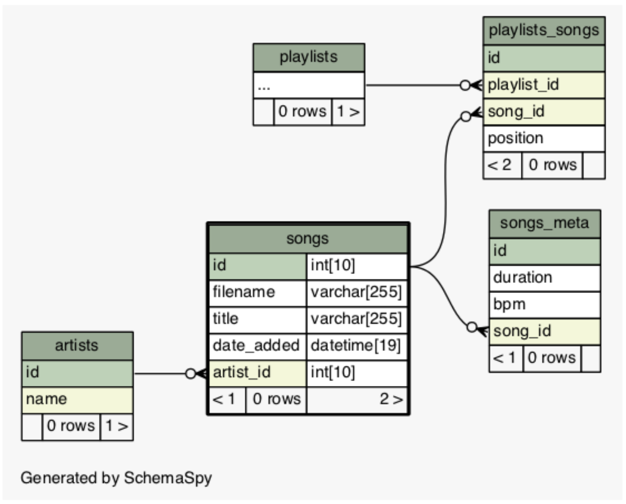


Figure 2: ER diagram for Music classes. Relationships are shown via crows-feet arrows between class blocks.

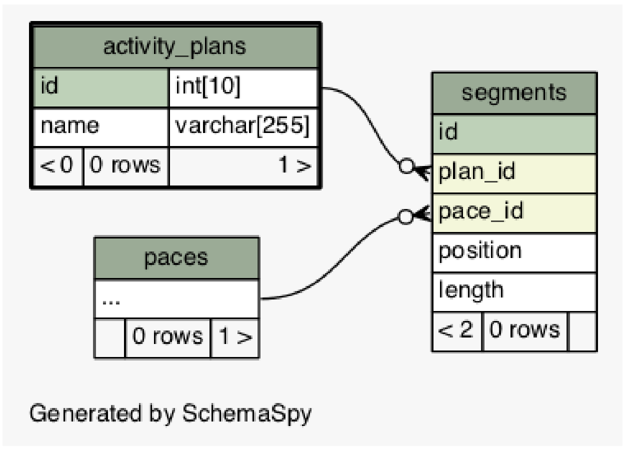


Figure 3: ER diagram for Activity classes.

songs in the corpus. First, the corpus is sorted by BPM, before being divided in a 30/30/30/10 split, such that the bottom 30% of the songs in the list are the "Slow" set, the next 30% are the "Steady" set, the third 30% are the "Fast" set, and the fastest 10% of songs are the "Sprint" set. If the corpus of songs were to change, this effectively means that the BPM ranges that are considered "Slow", "Steady", etc. will change based on the range of BPMs in the corpus of songs. We opted for this weighting as we felt that most people would be sprinting for significantly less time than they would be running at a slower pace.

The algorithm itself can be summarized using the following pseudocode:

```
• For each segment at pace $p_1$:
    – Add random songs from $p_1$_set to the playlist,
      until no more full songs at pace $p_1$ can fit without overlapping the next segment
    – If next segment is faster than current segment
      and the shortest song in $p_1$_set overlaps the next segment by more than 50% of the song's time:
        ∗ Choose the shortest song from $p_2$_set, where $p_2$ is the pace of the next segment
    – Else:
```

        * Choose the shortest song from p_1_set
    - Calculate the overlap of the last song from the previous segment with the current segment and account for this in the remaining time counter
        * If the current segment is completely overlapped by the last song from p_1_set, skip the segment entirely

Essentially, each segment is filled with random songs that match the segment's pace, until no more songs at that pace can fit into the segment without overlapping the next - at this point, some logic is in place to determine whether to choose a song at the current pace or at the next segment's pace. Our playlist algorithm has been setup to favour the playlist speeding up early rather than staying slow for too long. This is because we believe a slow song overlapping into a faster segment is more detrimental than a faster song overlapping into a slow segment; people are more likely to be slow even when the music tells them to be fast than they are to be fast when the music tells them to be slow.

If one of the subsets of songs is empty due to each song from the subset having already been selected, the generation algorithm implementation contains logic to choose from the next non-empty subset, in increasing speed order. For example, if the "slow" set is empty, the algorithm will first attempt to pull from the "Steady" set - if the "Steady" set is also empty, it will next try the "Fast" set, and so on. In the case of the "Sprint" set being empty, the algorithm will "wrap around" to the slow set.

The algorithm will continue to add songs to the playlist until one of two end conditions is reached: either we have reached the last segment and there is no more time remaining on the counter, or we have run out of songs from all of the sets. In either case, the playlist will then be written out to a file and the function will terminate.

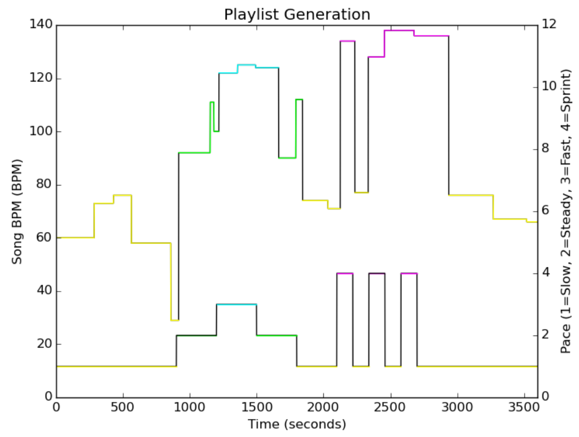


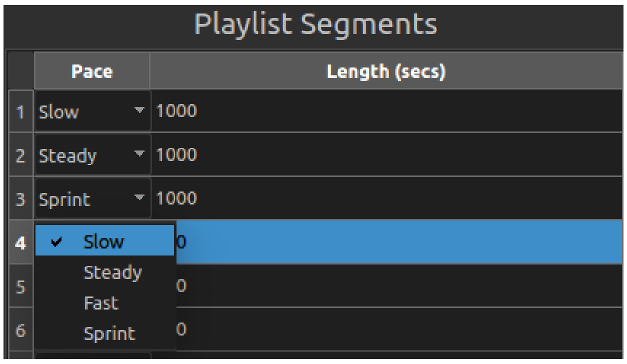
Figure 4: An example of a playlist generated by our algorithm plotted against the input workout plan (colour coded to the pace: Yellow = Slow, Green = Steady, Blue = Fast, Purple = Sprint).

6.1.5 Playlist Format

There are a variety of playlist formats available [18] and each has their own advantages and disadvantages. We opted to use the PLS file format. This choice was for two main reasons; first, the format is widely supported by a range of media players, most notably iTunes and VLC; second, the format is essentially a type of INI file, which are easy to read and write in Python using the "ConfigParser" module. By specifying the title, file path, and length of songs in the playlist file, the file can be simply opened in Mac or Linux in either VLC or iTunes, and will automatically locate the songs and load them into the player.

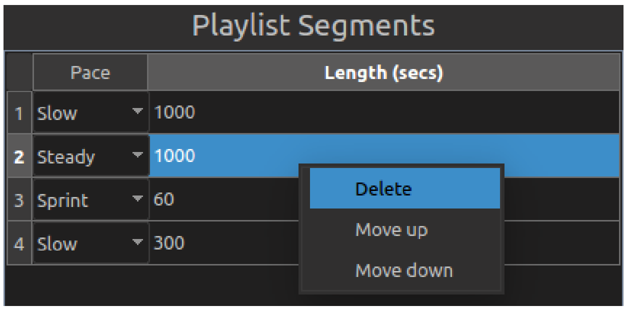
6.1.6 UI

The front end of the application has been completed and tied together with the database. Functionality has been added to allow users to add, move, edit, and delete segments from the interface. Functionality for playlist generation has also been completed, allowing users to generate a playlist corresponding to a list of user defined segments. Users now have the ability to create multiple playlists within a plan, and define names for these playlists. Various error prompts and input windows have also been added to provide feedback to users and make the application easier to navigate and understand. The following figures give an overview of the UI [19] [20].


Playlist Segments

	Pace	Length (secs)
1	Slow	1000
2	Steady	1000
3	Sprint	1000
4	✓ Slow	0
5	Steady	0
6	Sprint	0

Figure 5: The new format for the table containing the segments, along with the option to edit the pace of a segment within the table.

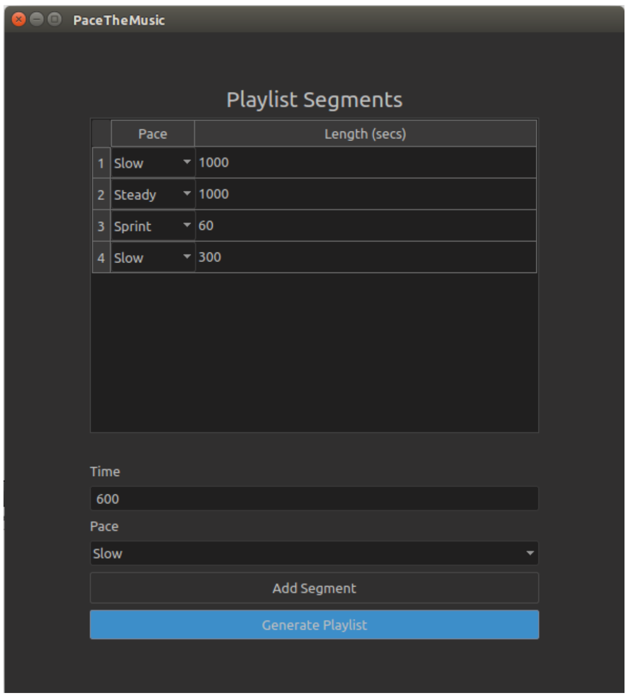

Playlist Segments

	Pace	Length (secs)
1	Slow	1000
2	Steady	1000
3	Sprint	60
4	Slow	300

Figure 6: The right-click menu for moving and deleting segments.

The above figure shows an overview of the completed UI. Users define the length and pace of segments using the controls on the bottom, and click the "Add Segment" button to add the segments to the above table. Once a desired list of segments has been chosen, users can click the "Generate Playlist" button to create a playlist.

The base structure of the UI layout has also been updated, allowing the UI to scale to different resolutions. An



	Pace	Length (secs)
1	Slow	1000
2	Steady	1000
3	Sprint	60
4	Slow	300

Time: 600
Pace: Slow

Figure 7: An overview of the completed UI. Users define the length and pace of segments using the controls on the bottom, and click the "Add Segment" button to add the segments to the above table. Once a desired list of segments has been chosen, users can click the "Generate Playlist" button to create a playlist.

additional change to the layout of widgets was also required due to issues with UI elements scaling differently on machines running OS X. This update was done using Qt Creator. The issue was that the central and input layout widgets of the UI were in a grid format. Switching to a vertical layout widget resolved the issue.

6.2 Remaining Work

Now that a basic version of the application has been completed, we can look towards adding additional features as well as expanding the application to cover other areas and platforms. Adding support for user defined paces is an easy addition that would add another layer of customization to the application. Support for MP3 files as well as different formats for playlists would be another useful addition to the application. As mentioned previously, the ability for users to specify a folder of songs for the application to use is a feature that needs to be added to the front-end, but would not require much work to implement. This addition would result in a fully-functional end-to-end application, without the need to separate the feature extraction process from the rest of the application. Another feature that would be useful to improve user experience is to base playlist generation off of additional features (in addition to BPM) - for example, a user-specified rating system. Developing a mobile version of the application is another area which could be explored. Expanding the application to cover mobile platforms would require a lot of effort, but it would potentially result in a much larger audience. It would also make sense for the application to be available in a mobile format, as that is where the majority of people would store and use their playlists.

7. TEAM ROLES

The roles of each team member are loosely defined as follows; however, they are subject to change, and team members will contribute in other areas of the project as needed. Nick will be in charge of GUI design and front-end development, James will handle back-end design and database logic, and Kaileen will be in charge of developing feature extraction logic.

8. CONCLUSION

Although there is some work to be done to expand and perfect the application, we feel that we have developed a solid first iteration of the playlist generation utility that we set out to create for this project. After talking to various potential users, we believe that this application could prove to have multiple real-world uses such as generating playlists for running as well as spin classes. This has motivated us to continue development and release an improved version of the application to the public. We look forward to learning more about the field of music information retrieval and playlist generation, with the goal of making the world a better place, one segment at a time.

9. REFERENCES

[1] N. Shivar. "How I Cut 1:57 Off My Average 5k Time By Tweaking My Playlist." http://www.nateshivar.com/1182/how-i-cut-157-off-my-average-5k-time-by-tweaking-my-playlist/, 2015.

[2] N. Oliver and L. Kreger-Stickles. "PAPA: Physiology and Purpose-Aware Automatic Playlist Generation." In *Proceedings of the International Symposium on Music Information Retrieval.* http://ismir2006.ismir.net/PAPERS/ISMIR06162_Paper.pdf, 2006.

[3] S. Pauws and B. Eggen. "PATS: Realization and User Evaluation of an Automatic Playlist Generator." In *Proceedings of the International Symposium on Music Information Retrieval*. http://www.ismir2002.ismir.net/proceedings/02FP074.pdf, 2002.

[4] B. Logan. "Content-Based Playlist Generation: Exploratory Experiments." In *Proceedings of the International Symposium on Music Information Retrieval.* http://www.ismir2002.ismir.net/proceedings/03SP052.pdf, 2002.

[5] E. Pampalk, T. Pohle and G. Widmer. "Dynamic Playlist Generation Based on Skipping Behavior." In *Proceedings of the International Symposium on Music Information Retrieval.* http://cis.ofai.at/~elias.pampalk/publications/pam_ismir05b.pdf, 2005.

[6] N. Masahiro, H. Takaesu, H. Demachi, M. Oono and H. Saito. “Development of an Automatic Music Selection System Based on Runner’s Step Frequency.” In *Proceedings of the International Symposium on Music Information Retrieval.* 2008.

[7] L. Chen, Y. Tung and J. R. Jang. “MART: Music Assisted Run Trainer.” In *Proceedings of the International Symposium on Music Information Retrieval.* http://www.terasoft.com.tw/conf/ismir2014/LBD/LBD24.pdf, 2014.

[8] M. McKinney and J. Breebaart. “Features for Audio and Music Classification.” In *Proceedings of the International Symposium on Music Information Retrieval.* https://jscholarship.library.jhu.edu/handle/1774.2/22, 2003.

[9] A. Pikrakis, I. Antonopoulos and S. Theodoridis. “Music Meter and Tempo Tracking from Raw Polyphonic Audio.” In *Proceedings of the International Symposium on Music Information Retrieval.* http://www.ee.columbia.edu/~dpwe/ismir2004/CRFILES/paper160.pdf, 2004.

[10] M. Alonso, B. David and G. Richard. “Tempo and Beat Estimation of Musical Signals.” In *Proceedings of the International Symposium on Music Information Retrieval.* http://www.ee.columbia.edu/~dpwe/ismir2004/CRFILES/paper191.pdf, 2004.

[11] D. Liu, L. Lu and H. Zhang. “Automatic Mood Detection from Acoustic Music Data.” In *Proceedings of the International Symposium on Music Information Retrieval.* https://jscholarship.library.jhu.edu/handle/1774.2/14, 2003.

[12] “SQLAlchemy.” http://www.sqlalchemy.org, 2016.

[13] “Music21.corpus.” *Music21 Module Reference.* http://web.mit.edu/music21/doc/moduleReference/moduleCorpus.html, 2015.

[14] “Pyjs.” http://pyjs.org/Overview.html, 2016.

[15] “PyQT.” https://www.riverbankcomputing.com/software/pyqt, 2015.

[16] “TkInter.” http://tkinter.unpythonic.net, 2014.

[17] “SQLite.” https://www.sqlite.org/, 2016.

[18] L. Gonze. “A survey of playlist formats.” http://gonze.com/playlists/playlist-format-survey.html, 2003.

[19] “PyQT tutorial.” https://nikolak.com/pyqt-qt-designer-getting-started/, 2015.

[20] “PyQT guide.” https://www.safaribooksonline.com/blog/2014/01/22/create-basic-gui-using-pyqt/, 2014.