AUTOMATIC SUPPORT FOR DIGITAL MASHUPS

Giancarlo Camilo
University of Victoria
gkc@uvic.ca

Rafael Valle
University of California
jrafaelvalle@gmail.com

Thiago Lima
University of Victoria
trcl@uvic.ca

ABSTRACT

This paper describes the whole process revolving the creation of our automated music mashup creating software. Called Automatic System for Digital Audio Mashup, or ASDiM, the software is still a work in progress and a subject of continuous study in different areas of Music Information Retrieval, such as song segmentation, music similarity, instrument recognition and audio representation.

1. INTRODUCTION

The art of mixing songs or elements within songs started getting notoriety the 1990s and originated the Mashup style. Mashups are now very popular in music production and remixing. However, the process for mashup creation depended on professionals with access to a large audio database, mixing tools and time to compare songs to find the mixable ones.

Our first motivation was to bring this ability to a regular user, so anyone could input their own songs or their favourite songs and then listen to new ones. Although some comercial software already exist to assist in mashup creation, the process still relies mainly on human work. *Harmonic Mixing* [1] and *DJ Mix Generator* [2], for example, are tools for identifying compatible songs based on specific features, such as keys and tempo.

On the other hand, *Mashup* [3] is a software that automatically finds harmonically compatible songs for a track given as input, matches the beats and then lets the user adjust the result. Davies *et al* [2] proposed the *AutoMashUpper* system for automatically creation of mashups, which focuses on a new masure of what they call "mashability" – the ability of two songs to fit.

There is still very little exploration of the automatic mashup creation in both commercial and research areas and that motivates even more to develop our own ASDiM and take part on what has been called one of the "grand challanges" of Music Information retrieval [9].

[1] http://harmonic-mixing.com/
[2] http://www.djprince.no/site/DMG.aspx
[3] http://mashup.mixedinkey.com/

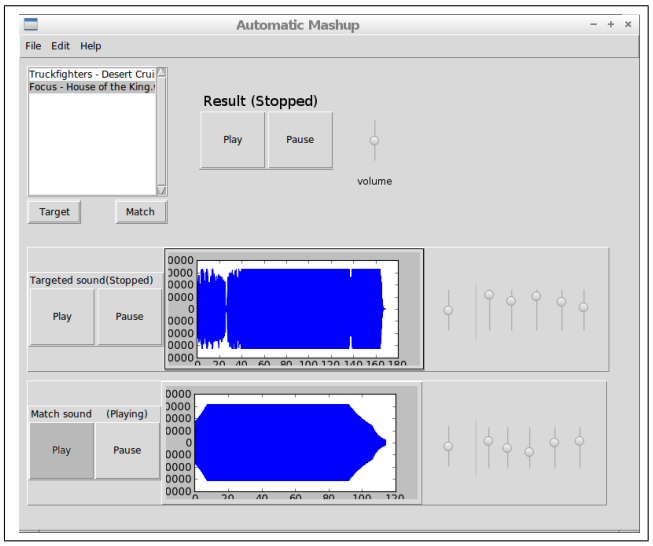

Figure 1. A sample for the user interface

2. INITIAL CONCEPT

As previously stated, our motivation is to have a system that would automatically mix up two songs to create a third one. A standard use-case scenario would consist of a user selecting an input music file and providing a directory containing a song database. The software would then compare the input song with the given dataset to find the most compatible song that was different from the original one. Finally, the user would have the option to choose which instruments from which song would be heard in the final version.

The user would be presented with an interactive interface, such as the one in Figure 1. In this case there is a list of songs selected by the user, a visualization for the selected track and another one for the matched song found by the program. There are sliders to provide equalization of the tracks to control the intensity of different range of frequencies or instruments. There are also play/stop buttons to preview any track at any time.

This gave us three distinct areas to work on: graphical user interface, song matching and audio processing/equalization.

3. PLANNING

Since we have three members in our group and three major areas to work with, we could have assigned each area to a member. However, we decided to participate on all fields. For that matter, we decided to start off with a sketch of

each area so we have a standard for the whole program and later optimize them as needed. We will be holding online meetings every two weeks.

3.1 Graphical User Interface

The GUI is so far the least concerned area. A raw interface with basic usability is enough for the developing and prior testing stages. A real user-friendly interface, with modern design and animations would only be necessary for a commercial release, which is not in our plans at the moment.

3.2 Song Matching

The process of finding the best matching song is the core of the system. To accomplish this, we decided to study and test different approaches. The first step was to choose which features we would extract from the pieces of music to compare them. The next problem is how to compare them.
Foote [8] extracts Mel-Frequency Cepstrum Coefficients and trains a quantization tree. Aucouturier and Pachet [13] suggest a similarity measure based on a timbre signature for a whole song. They use Gaussian Mixture Models and MFCC. Spectra calculated from beat-tracking has also been used as a similarity measure [7].
For the first version we decided to focus on three features: MFCC, tempo and chromagrams. The members responsible for them are, respectively, Giancarlo, Thiago and Rafael. We will then analyze the efficiency of different methods of representing and calculating the distance between feature vectors. The first choices included euclidean Distance, Edit Distance and GMM.

3.3 Audio Segmentation and Equalization

The ideal system would be able to identify the instruments in each track and give the user the ability to control each of their volumes. For the sake of full automation, a random combination may be suggested.
A problem that arises is that just as the original songs may have several segments, so might have the resulting track. This would require separate controls for them and could also need different matchings for each segment, which result in an even harder search process in a big database.
In literature there is some proposed query optimizations. Schnitzer *et al* [19] propose a filter-and-refine method based on FastMap, which projects the songs into higher dimensions. An indexing technique based on *Factor Oracle*, used for search in text, has been proposed for songs: Audio Oracle [5] [6].
For the first stages a simple equalizer (for ranges of frequencies) may be implemented for each of the original tracks.

3.4 Material

We discussed about the tools we would use during the development of the project. The chosen programming language was Python, given there is already a solid base of modules freely available for statistics, machine learning and even music retrieval. Our code will be maintained on GitHub and a shared document on Google Drive will be used for brainstorming and update reports.
For the GUI, we started using Python's standard package: Tkinter. For the song matching part we are still to decide between using *MARSYAS*[4] or the *Bregman Audio-Visual Information Toolbox*[5], a package with Python modules designed for MIR. *jAudio*, which can output data in format accepted by Weka [16], was also considered, but then discarded.

3.4.1 Papers

For further research we have already picked some papers that may help us along the way, aside from some that have already been cited. We selected [1] and [14] for audio segmentation, [3] and [4] for beat detection, [18] and [15] for MFCC.

4. OBJECTIVES

1. Set up basic Python code:
   - Open WAV files.
   - Create GUI.
2. Compile and explore Marsyas.
3. Check Bregman Toolbox's documentation.
4. Decide between Marsyas and Begman Toolbox.
5. Study the features to be extracted:
   - MFCC - Giancarlo.
   - Tempo - Thiago.
   - Chromagram - Rafael.
6. Analyze impact with different algorithms.
7. Combine information to create first version of song similarity area.
8. Study music segmentation detection.
9. Create equalizer for both tracks.
10. Update GUI and finish first version of audio equalization.
11. Optimize.

[4] http://marsyas.info/
[5] http://bregman.dartmouth.edu/bregman/

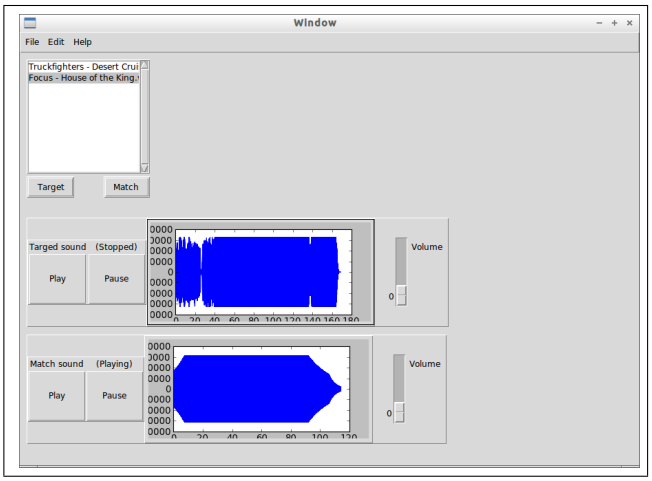

Figure 2. The first version of the GUI

5. PROGRESS

5.1 Tools

We have tested both Marsyas and Begman Toolbox. It was decided to work mainly with the latter because it is easier to use within our Python code and there are already some functions implemented that we need for feature extraction. We may use some tools from Marsyas if it becomes necessary and Begman Toolbox cannot supply a solution, such as the *tempo* command.
Reading the documentation from Begman Toolbox we learned there are modules for chromagram and MFCC extraction. It has a function that searches a database of songs given some range of tempo as key and also has a classifier based on multivariate Guassian models.

5.2 GUI

A simple interface has been done using Python's default GUI module – *Tkinter* – and *PyGame* [6] to operate the WAV files. Begman Toolbox has support to manage the files, but there were some problems with the communication between the audio device and the module in some machines. We plan to fix this soon. Figure 2 shows the screen so far (upon which, Figure 1 was based).

5.3 Beat Detection

Dixon *et al* [4] proposes a beat tracking algorithm that analyzes musical data, checks for onsets of rhythmic changes and determines the tempo using a multiple hypothesis search. They achieved good results, only with minor ocasional phase errors.
Davies and Plumbley [3] use a complex spectral difference onset detection function and retrieve the beat times by passing its autocorrelation function through a shift-invariant comb filter. This is usd to identify the phase of the beats.

[6] http://pygame.org/

5.4 MFCC

MFCCs are commonly used in speech recognition, but recently they also have been used for music modeling. The main objective here was to find if they can be used as similarity measures and, if so, how can we use them in our program.
According to Beth Logan [15], the process of obtaining MFCCs has 5 main steps:

1. Divide signal into frames
2. Obtain the amplitude spectrum
3. Take the logarithm
4. Convert to Mel spectrum
5. Take the Discrete Cosine Transform (DCT)

And although the 4th step is not sure to be the optimal scale for modeling music, it is not harmful. That is, it may not improve the results, but it will not worsen it. All the other steps are appropriate for music spectra.
Loughran *et al* [18] show that, for musical instrument recognition, the use of MFCCs yields good results. It also shows that for music modeling, at least 10 MFCCs should be used.
We decided to use Bregman Toolbox for the the aforementioned reasons. We plan on using 25 MFCCs as a starting point, with the possibility of reducing the number depending on the results achieved with different combinations. It is possible that not much more than 10 MFCCs are necessary, considering we plan on using other features aswell.

6. DEVELOPMENT

We have studied some of the required components for the proposed system to start working on the objectives that were previously defined. Each member worked on their own assignments but also helped all other areas of development. Giancarlo was mainly focused on the GUI and MFCC extraction, Thiago concentrated on tempo, equalization and similarity measure. Rafael was responsible for the chromagram extraction.
The following subsections describe the achievements related to each previously stated goal.

6.1 Set up basic Python code

Currently, the system only loads audio files encoded in *WAV* format. Bregman's modules were used to load and write the audio files. However, to play the audio inside the program, Bregman Toolbox was not suitable because it was unable to play some specific files. Thus, PyGame modules were used to reproduce the *WAV* files.
As for the graphical user interface, Tkinter was used and the initial GUI was upgraded to fit new functionalities. The code was reestructured to follow a object-opriented paradigm, which results in a more scalable system. A new section was added on the GUI for the mash-up itself. The user has access to information regarding the target sound,

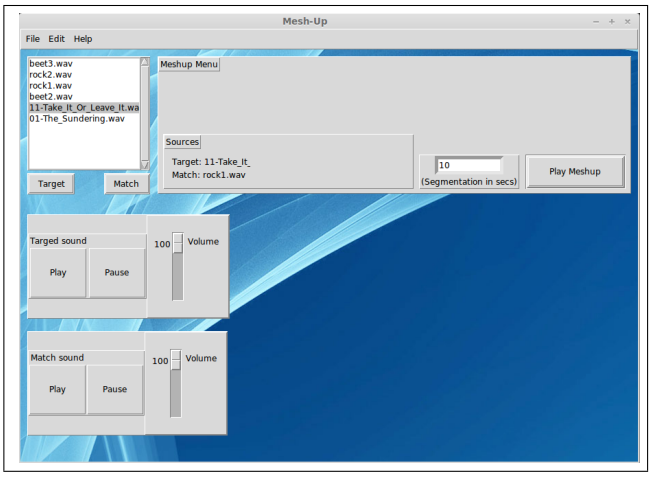

Figure 3. The GUI so far

the matched song and to mash-up creation settings. So far, the user can choose a duration time for each segment of the original songs, that are then concatenated to produce the mash-up. The last version of the interface is shown on Figure 3.

6.2 Marsyas and Bregman Toolbox

After reviewing Bregman's documentation, we decided to use the modules to extract MFCC and Chromagram features, besides the loading and writing functions for WAV files. However, we realized it lacked a function to extract the tempo.
We were able to compile and test Marsyas under different operating systems. Among the several tools that were available, we decided to use the INESC-Porto Beat Tracker (IBT) to retrieve the tempo from the songs that were to be analysed.
A third resource that was considered was LibROSA [7]. It also has tempo extraction modules, along with other functionalities, such as feature extraction and sound source extraction.

6.3 Study the features to be extracted

The features that are extracted from the songs are stored in a text file that works as the database for our system. Each time an audio file is loaded, the system checks the known database for a matching entry. If it is the case of a new file, the features are extracted. The audio information is stored in the following way:

- File name
- List main features (MFCC and Chromagram so far)
- Median tempo

Considering extracting MFCC takes some time, this method saves time when the user wants to load the same collection of songs again for future usage.

[7] https://pypi.python.org/pypi/librosa

6.3.1 MFCC

We used modules from the Bregman Toolbox to extract the MFCC information from the loaded file. As previously stated, the recommended minimum number of bands to be considered is 10 and thus we decided to run tests using 10, 12, 18 and 20. Given that we are using MFCCs to measure the similarity between songs, which is greatly subject to human taste, we didn't find much improvement from 12 bands to 20 bands. On the other hand, there was a considerable delay increase with higher amount of bands. Thus, we decided to use a 12 band MFCC.
Bregman's module *LogFrequencyCepstrum()* returns an object containing a matrix with 12 rows and n columns, where n is the length of the audio file. The matrix yields the frequency intensity for the corresponding band at a given time. As saving these values would result in feature vectors of different sizes depending on the audio file length and it would require too much memory space and processing time, we decided to calculate and store the mean value for each band. Nevertheless, we realized that the average value is not very accurate when representing such a long set of numbers, since two very dissimilar sets may yield the same mean value. We then decided to also store the standard deviation for the bands.

6.3.2 Tempo

The tempo feature was the most controversal. At first, matching an audio track with another one with similar tempo seemed like the fastest way to filter out songs that were completely off. We then realized that two songs with same tempo may not necessarily fit well together, whereas some songs with different tempos can still sound harmonious.
Our python code calls Marsyas's *ibt* binary file, which extracts both beat timings and the median tempo. The values retrieved from the beat timings could be of help when trying to synchronize two songs. But, for now, our code only uses the median tempo, which *ibt* writes into a file that is then opened and read by our script.

6.3.3 Chromagram

The Chromagram information was initially extracted using Bregman's *Chromagram()* function, which also had 12 bands. So far the system also saves the mean and standard deviation values, beside the MFCC features. New tests were executed by extracting the chromagram from just the harmonic part of a song, by first excluding the percursion. This was implemented using LibROSA's functions. However, it has not yet been integrated into the main system.

6.4 Algorithms and Similarity Measurement

As previously stated, we tested different matchings by prefiltering the results by the tempo difference between them. But this has proved not to be a successful approach. So we decided to calculate the euclidean distance using the feature vectors.
Once the user chooses the target song, the program traverse the list of loaded files to calculate and return the smallest

euclidean distance. The euclidean distance between songs a and b with n features is given by the following equation:

$$\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

6.5 Sound mixing

Our system is divided in two main steps:

1. Find the most harmonically suitable match.

2. Automatically generate a mix of both samples.

The second one can be done in several ways. We basically tried two approaches. The first one we considered was to simultaneously play both songs, while allowing the user to control which bands of frequencies or instruments were to be heard from one song or another. This, however, has proved to be harder than initially expected and may be then considered for a future improvement. The second and current method simply concatenates segments of both songs alternately, with a duration specified by the user. When the duration is set to a single unit, the songs seem to match in a more harmonically way, although the reproduction speed is halved.

7. ISSUES

Throughout the study and development processes, some tasks turned out to be much more challenging than initially expected. The first problem was related to memory usage. It seems like Bregman's modules keep consuming memory, even when the audio file is closed and the object, that supposedly contained it, is destroyed. This resulted in 100% memory usage when extracting the features of 15 songs or more. Following the way *ibt* was used, the memory problem was solved by calling another instance of python to deal with the extraction of every single song. the resulting features are also stored in a file for our main script to open, read and add to the database.
Another problem happened when dealing with the Discrete Fourier transform. The idea was to separate the audible frequencies from a song and give the user the ability to increase or decrease their magnitude. We used NumPy's *fft.fft()* and *fft.ifft()* to decompose and recreate the audio signal, respectively. However, this always ended up in Python printing a *Memory Error* and exiting. Then, several workarounds were tried, but there was no success.
Finally, the automatic music segmentation had to be discarded for the first version of the system. The first idea was to break the song in an arbitrary number of segments and extract the features, store in the database and find then find a matching segment of another song. This would require much more space and processing time, so it was disregarded for the time being.

8. FUTURE IMPROVEMENT

8.1 Chromagram

Although the structure of the chromagram – that extracts the notes considering only the harmonic part – was implemented, it has not yet been added to the main system. The extraction chromagram that currently lies within our main code is raw and considers the whole audio file. Ideally, the new chromagram feature vector could improve the music similarity measure, given that it represents the keys played across the song. So, if the keys of two songs could be synchronized, alternating the sound of each song, would produce a very harmonious mash-up, with just a varying timbre. Jensen *et al* [11] [12] suggest this kind of extraction for MFCC too.

8.2 Similarity

So far, our system only returns one match, the most similar according to the euclidean distance using the selected features. An improvement would be to return a small list of best matches, and so the user could decide which song to use.
Another idea is to give the user the freedom of choosing which features to consider when searching for the best match. This way, songs could be matched by either harmonic or tempo similarity.

8.3 Music Segmentation

Currently, our system stores the features for the whole song because saving the data for each segment of every song would slow down the system and consume too much memory. To use music segmentation efficiently on mash-ups, we plan on segmenting the audio after the list of matches has been returned. This way the system may only extract features from the segments of one song and find the most suitable candidate (from within the returned list) for each segment. If the candidate list is small enough, these songs may also be segmented, so the distance is calculated between two segments. For this case, each segment would represent a piece of music with no abrupt changes in rhythm and harmony within itself.

8.4 Equalizer

As previously stated, a sound equalization could be used as a tool to generate more clean results after the mash-up. When songs were played simultaneously with no sound editing, the resulting audio had a lot of noise and no synchrony. Isolating the percursion of only one song at a time and the harmonic part of any other matching song, the resulting sound would be better.

8.5 GUI and I/O

We are still planning on using another library to play the songs within the system. PyGame library is too heavy for this task.
The interface may be upgraded to allow the user to select which segments of a song need to be considered for the

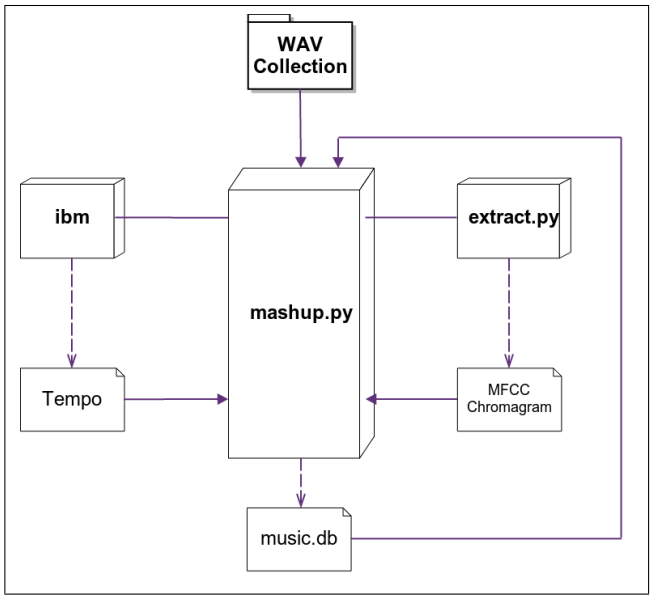

Figure 4. System diagram

similarity measure and mash-up. The user may also need more control over the music reproduction [10], allowing he or she to select a start point to hear the resulting audio and play/pause buttons for the mash-up. Music visualization of the input and output songs may also be added to give a better user experience [17].

9. CONCLUSION

In this paper, we have presented a system for automatic assistance on the creation of audio mash-up. This has been just the first iteration on the production cycle. The system is still very raw, as we did not have much time to study every area deeply, but we did work on all of them.
The diagram in Figure 4 represents the main course of our system. The user loads a collection of *WAV* files into the main script, which then calls *ibt* and the MFC and Chromagram extracting script. The output files of both these programs are loaded and added to the database, which will be later consulted in case the same files are loaded in a future usage.

For this project, we have studied different kinds of features and their respective relations with each area of our system. We realized the process of finding a mashable song for a given target is harder than initially thought. It is needed to find songs with matching tempo and harmony, and then synchronize them using beat tracking and audio segmentation. A common event was matching a song with a long introduction with another that had almost none, which resulted in many abrupt changes in rhythm. Still, the matched songs really sounded alike and did not produce such bad noisy sounds as a random mixture would do. We believe working on the proposed improvements could result on the production of songs with a decent quality and clean sound.

10. REFERENCES

[1] Patrick Marmaroli Dalia El Badawy and Herv Lissek. Audio novelty-based segmentation of music concerts, 2013.

[2] Matthew E. P. Davies, Philippe Hamel, Kazuyoshi Yoshii, and Masataka Goto. Automashupper: An automatic multi-song mashup system. In *ISMIR*, pages 575–580, 2013.

[3] Matthew E. P. Davies and Mark D. Plumbley. A spectral difference approach to extracting downbeats in musical audio. In *Proceedings of 14th European Signal Processing Conference (EUSIPCO)*, 2006.

[4] Simon Dixon. Automatic extraction of tempo and beat from expressive performances. *Journal of New Music Research*, 30:39–58, 2001.

[5] Shlomo Dubnov, Grard Assayag, and Arshia Cont. Audio oracle: A new algorithm for fast learning of audio structures. In *In Proceedings of International Computer Music Conference (ICMC)*, 2007.

[6] Shlomo Dubnov, Grard Assayag, and Arshia Cont. Audio oracle analysis of musical information rate, 2011.

[7] Jonathan Foote, Matthew L. Cooper, and Unjung Nam. Audio retrieval by rhythmic similarity. In *ISMIR*, 2002.

[8] Jonathan T. Foote. Content-based retrieval of music and audio. In *MULTIMEDIA STORAGE AND ARCHIVING SYSTEMS II, PROC. OF SPIE*, pages 138–147, 1997.

[9] Masataka Goto. Grand Challenges in Music Information Research. In Meinard Müller, Masataka Goto, and Markus Schedl, editors, *Multimodal Music Processing*, volume 3 of *Dagstuhl Follow-Ups*, pages 217–226. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2012.

[10] Jarmo Hiipakka and Lorho Gaetan. A spatial audio user interface for generating music playlists. In *Proceedings of the 9th International Conference on Auditory Display (ICAD)*.

[11] D. P. W. Ellis J. H. Jensen, M. G. Christensen and S. H. Jensen. Quantitative analysis of a common audio similarity measure. *IEEE Tr. Audio, Speech, Lang. Proc.*, 17:693–703, 2009.

[12] J. H. Jensen, M. G. Christensen, M. Murthi, and S. H. Jensen. Evaluation of mfcc estimation techniques for music similarity. In *European Signal Processing Conference, (EUSIPCO)*, 2006.

[13] Jean julien Aucouturier and Francois Pachet. Music similarity measures: What's the use ?, 2002.

[14] Ari Lazier and Perry Cook. Mosievius: Feature driven interactive audio mosaicing, 2003.

[15] Beth Logan. Mel frequency cepstral coefficients for music modeling. In *In International Symposium on Music Information Retrieval*, 2000.

[16] Cory Mckay. Automatic music classification and similarity analysis.

[17] Elias Pampalk and Masataka Goto. Musicrainbow: A new user interface to discover artists using audio-based similarity and web-based labeling. In *Labeling, in the Proceedings of the ISMIR International Conference on Music Information Retrieval*, pages 367–370, 2006.

[18] Michael ONeill OFarrell Risn Loughran, Jacqueline Walker. The use of mel-frequency cepstral coefficients in musical instrument identification. In *Routes/Roots*, 2008.

[19] Dominik Schnitzer, Arthur Flexer, and Gerhard Widmer. A filter-and-refine indexing method for fast similarity search in millions of music tracks, 2010.