INTEGRATING MACHINE LEARNING WITH DSP FRAMEWORKS FOR TRANSCRIPTION & SYNTHESIS IN COMPUTER-AIDED COMPOSITION

Brandon Lincoln Snyder and Marlon Schumacher
Institut für Musikinformatik und Musikwissenschaft
der Hochschule für Musik Karlsruhe
Karlsruhe, Germany
`brandon.snyder@stud.hmdk-stuttgart.de`
`schumacher@hfm-karlsruhe.de`

ABSTRACT

In this paper we present applications integrating two classic machine learning methods into a Computer-aided Composition environment with the specific purpose of notating, organizing and synthesizing audio from large sets of sound data. We present a modular sample replacement engine driven by a classification method, and a texture synthesis application employing a clustering method. The applications are designed and presented with a particular focus on modularity and extensibility, with the goal of providing flexible options for integration into existing *OpenMusic* projects. Therefore, in addition to presenting the metholodgy behind our applications, we also highlight the modular aspects of their structure along with several functions for performing transient detection, Mel-frequency cepstrum analysis, and probability vector calculation.

1. INTRODUCTION

The development of computer software for compositional applications has a long tradition and today's computer-aided composition (CAC) environments provide users with powerful frameworks in which multiple types of media (instrumental, electronic parts, spatialization, gesture, etc.) can be created, represented and manipulated in an integrated fashion [1]. Machine learning (ML) methods open up a large number of possibilities for such environments, particularly because these methods can create multi-dimensional networks of relationships across large amounts of data [2]. In this paper we outline two applications for sound analysis, notation and synthesis driven by ML. The ML methods involved here are classic amongst current literature for ML-based audio applications, and are designed with clear access points for altering and extending them. We were particularly interested in these applications being modular. That is, for them to be easy to take-apart, build-up, and integrate into existing CAC projects. Built using a combination of the *OM-SoX* external library as well as custom functions within the *OpenMusic* environment, these applications are able to easily be incorporated into a user's larger CAC workflow.

In this paper, we first situate these applications within the wider context of machine learning-based and corpus-based synthesis and notation tools. Highlighting the modular aspects of the project, we show several custom functions that are implemented at different stages of the sound analysis, notation, and synthesis process. Then, we walk through the two applications, describing the techniques and algorithms involved, as well as show audio examples of the applications' output (available on the accompanying website). [1] Finally, we discuss the benefit, potential impact, and limitations of these applications, and speculate on further directions this project could take.

1.1 Related Works

Our interest in analyzing and querying large amounts of audio brought us near several methods in corpus-based concatenative analysis and synthesis. Tools such as the Caterpillar system [3], Audioguide [4], and OM-Pursuit [5] [2] are significant examples from this field, and the research behind those tools has informed our own decisions around what audio features, distance functions, and software to use. Though these aforementioned tools are effective and flexible, we have found in our experiences with students and professionals that they can be perceived as a bit of a 'black box' to artists with beginner to intermediate knowledge in these methods. The input and output of these tools are clear, while the internal algorithm is not immediately so. Thus, we sought to provide a modular toolbox within a popular visual composition environment, in a way that not only makes clear their internal processes, but also that is easy to adjust and integrate into existing *OpenMusic* projects.

[1] `https://edu.marlonschumacher.de/audio-ml-4-cac/`
[2] `http://www.music.mcgill.ca/marlonschumacher/software-contributions/om-pursuit/`

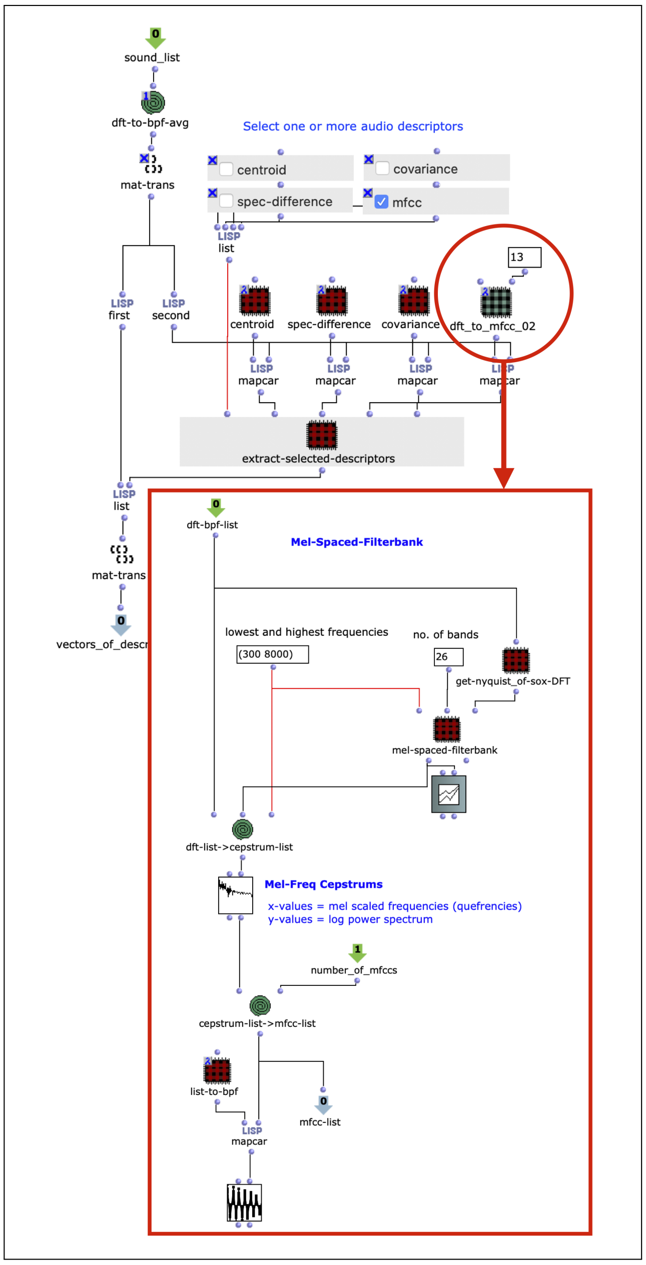

Figure 1: Interchangeable audio descriptors are set as patches in lambda mode. Here, a patch extracting 13 MFCCs is being used.

In terms of ML methods in CAC, our project shares similarities with OM-AI [6], an external library for *OM#* [3] that also seeks to provide tools to composers interested in ML. In order to maintain the modular nature of our applications, the ML algorithms and audio descriptor analysis functions are built as abstractions consisting of *OpenMusic* objects and simple custom LISP functions. Consequently, a user may change or even switch out these abstractions to suit their needs. Figure 1 shows our audio descriptor analysis engine with each descriptor as a patch in lambda mode. Featured in our application are Mel-frequency cepstrum coefficients (MFCCs), as well as spectral centroid, spectral difference, and covariance descriptors. Any number of these descriptors can be chained together to be incorporated into the ML process.

Likewise with our ML algorithms, Figure 2 shows a clustering algorithm in the form of an *OpenMusic* program, with the distance function implemented as a patch in lambda mode. This was an important structural decision to the project, as it not only gives a user a clear view of how the ML functions operate within the *OpenMusic* patch, it also allows for modularity within the actual ML algorithms. This would be useful for a user who might need to use a different distance function to evaluate their data, or who may need to pre-process or clean up their audio sets in some way before entering certain stage of the ML algorithm.

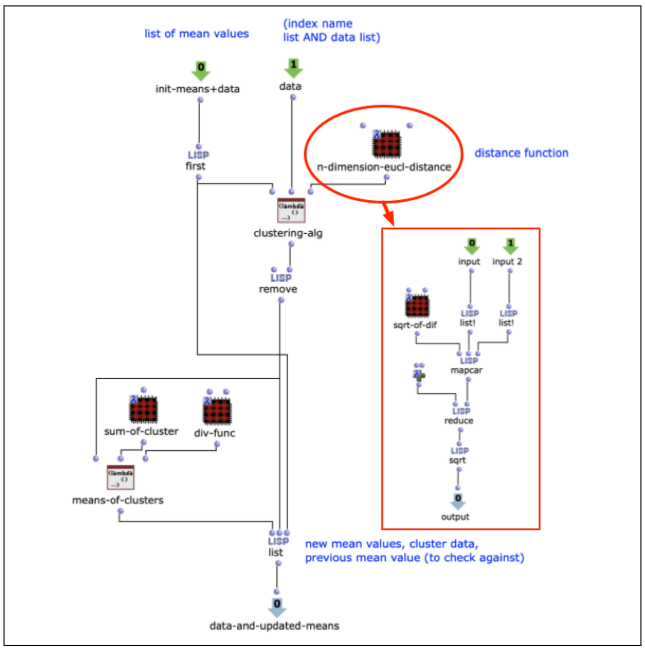

Figure 2: Interchangeable distance function. For certain audio sets, special functions such as a distance matrix or tree distance may be more suitable than Euclidean distance [3].

In order to provide the required DSP functionalities within *OpenMusic* and for flexible integration into existing workflows we employed *OM-SoX*, a framework for audio analysis and synthesis, bundled as an external library to be loaded into *OpenMusic*.

1.2 *OM-SoX*

OM-SoX is a programmable, modular analysis-synthesis framework for algorithmic audio processing in *OpenMusic*, developed by the second author. We used its analysis functions to derive audio descriptors, and detect and classify transients. We also used its processing functions to synthesize sound from that analysis data. While *OM-SoX* has been a popular tool used for audio processing [7], sound synthesis [4] and notation [5] (see Figure3), there so far

[3] https://github.com/cac-t-u-s/om-sharp

[4] http://www.music.mcgill.ca/marlonschumacher/arts/6-fragments-on-one-act-of-cleaning-a-piano

[5] http://notation.afim-asso.org/doku.php/evenements/2013-06-27-etude-notation1

seem to have been less notable applications for incorporating its analysis functionalities into compositional workflows. Included in our applications are functions that use *OM-SoX* to extract cepstrums, Mel-frequnecy cepstrum coefficients, spectral centroid, spectral difference, covariance, and detect transients. We hope this project might provide inspiration for possible musical applications of the analysis engine in *OM-SoX*.

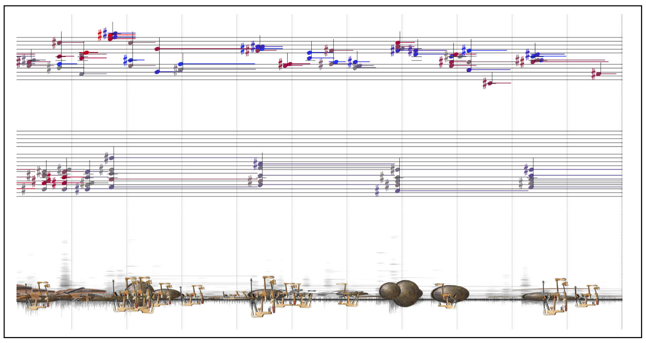

Figure 3: Figure shows the use of *OpenMusic* & *OM-SoX* for the creation of a mixed-paradigm notation in a *Maquette* object.

2. TWO MACHINE LEARNING APPLICATIONS FOR COMPUTER-AIDED COMPOSITION

With the overall scope of this project introduced, we will now present the two applications and the ML methods that drive them.

2.1 Modular Drum Sample Replacement

Our first technique uses a supervised classification method to create a modular sample replacement engine. In our example *OpenMusic* patch, we use three objects for mediating the sample replacement process: a *multi-seq*[6] notates the detection and classification of transients in the audio, a *BPF-library* notates the multi-sound-set sample replacement process, and a *maquette*-based sample sequencer transcribes new audio with the replacement samples.

The multi-seq, an object that notates data in the form of traditional western musical staves, sorts the segments of a given instrument-type (i.e. snare, kick, hi-hat) into its own staff (see Figure 4). This 'score' serves firstly as a transcription of the transients detected and classified by our *OM-SoX* functions. The score's second purpose is that of a script, dictating, in partnership with the BPF-library, the construction of a new sequence of samples.

In the BPF-library, each BPF corresponds to a transient which will be reconstructed from a pool of possible replacement samples. The specific replacement sample used is determined by a probability vector, which is sampled from the BPF.

In our accompanying audio examples there are three separate sample replacement libraries used: an acoustic

[6] https://support.ircam.fr/docs/om/om6-manual/co/Poly-Multi-Editor.html

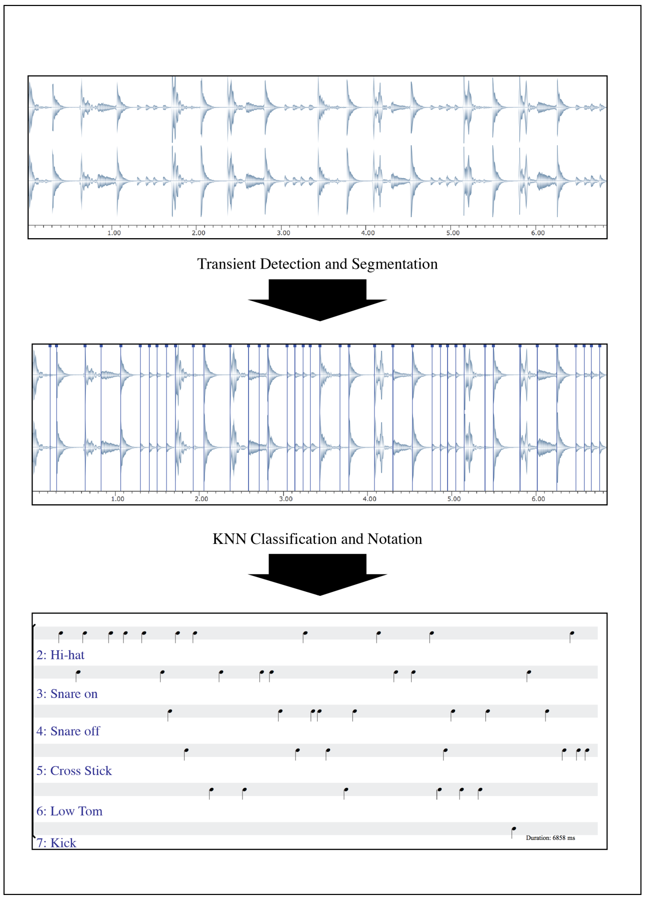

Figure 4: A sound is trimmed into individual segments, which are then classified and notated by instrument-type on a *multi-seq* object.

drumkit, an electronic drumkit, and a set of field recordings. Thus, each BPF called for each note from the multi-seq is sampled into three points. If the number of sample libraries were four, then four would be the sample-value of the BPF weighted vector. As illustrated in Figure5 the weighted vector is calculated from a three-point and four-point sampling. In the accompanying example audio, an interpolation between two BPFs is executed such that the beginning of the resynthesized drumloop consists exclusively of acoustic drum samples (sample replacement library 1), see Figure 6 for the specific interpolation. Gradually, this balance shifts so that at the end of the loop, there is an equal likelihood that a note from the multi-seq would call the electric drumkit or the field recordings. (sample replacement libraries 2 and 3). This BPF interpolation is a simple technique employing a high level visual interface for controlling the complexity of large sets of audio samples.

The multi-seq and BPF interfaces then lead to a maquette-based sequencer (see Figure 7), where the classified audio is represented again, this time with the final replacement samples (designated by color). Here, the transients notated and classified in the multi-seq can now be handled as actual audio samples, and can controlled for even further sound synthesis.

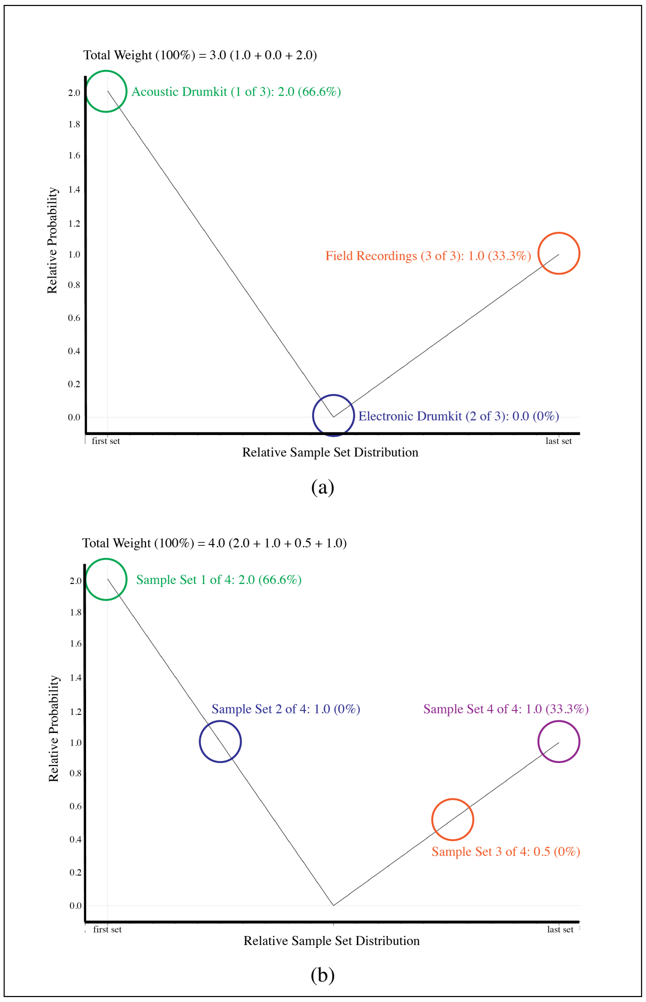

Figure 5: The x- and y-axis of the probability vector BPFs are in relative units. The probability values (y-axis) are relative to the total sum of all the indices values, and the distribution of proability values along the function is evenly distributed across the x-axis.

2.1.1 *Preparing Audio for Classification*

In this example, audio of live-performed drums is segmented into its individual attacks according to a transient detection method in *OM-SoX*. We track changes in amplitude between small grains to detect transients in the audio (see Figure 8. In our examples, we take the average amplitude of the audio. However, it is possible here to also detect transients along a specific band of the frequency spectrum.

Each segment is compared against a training set of already-classified drum hits using a k-nearest neighbors classification algorithm. The algorithm assigns each segment a class-ID, designating what drum or cymbal the segment contains. The class-ID of this segment sorts it into one of seven staves in a multi-seq (seven, per the number of class-IDs learned by the training set), altogether constructing a rough[7] transcription of the drum loop.

[7] The multi-seq in its current state notates only a class-ID, an onset time, and a duration for seven possible class-IDs. A higher level AMT tool would likely employ hundreds, accounting for dozens of different parameters including dynamics and timbre.

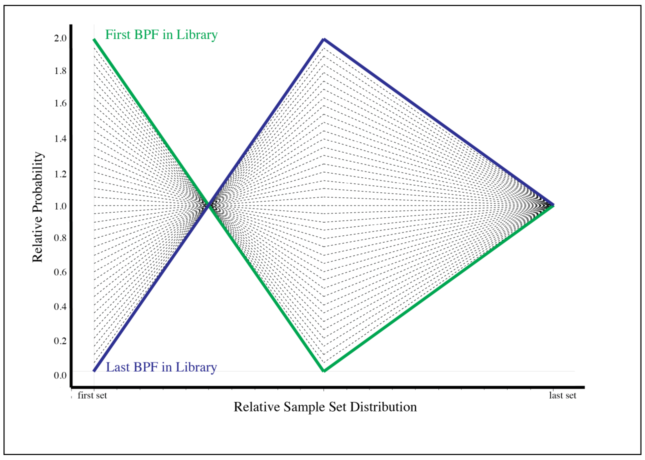

Figure 6: The gradual interpolation from the first BPF to the last is a simple technique that results in a complex mixture of many different sample libraries.

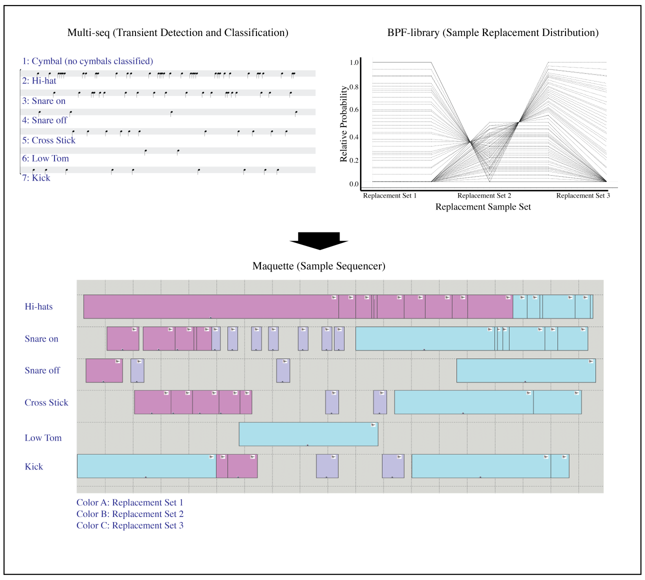
Multi-seq (Transient Detection and Classification)
BPF-library (Sample Replacement Distribution)
1: Cymbal (no cymbals classified)
2: Hi-hat
3: Snare on
4: Snare off
5: Cross Stick
6: Low Tom
7: Kick
Relative Probability
Replacement Set 1
Replacement Set 2
Replacement Set 3
Replacement Sample Set
Maquette (Sample Sequencer)
Hi-hats
Snare on
Snare off
Cross Stick
Low Tom
Kick
Color A: Replacement Set 1
Color B: Replacement Set 2
Color C: Replacement Set 3

Figure 7: The maquette notates the results of the sample replacement process controlled by the multi-seq and BPF-library.

2.1.2 *Classification Method*

The method of transient classification used is a k-nearest neighbors algorithm. This is a classic supervised ML classification method, one that has been applied already for many years to audio applications. For readers who may be experienced in *OpenMusic* but unfamiliar with ML methods, this section (and the later section on k-means clustering) dives into the specifics of how this ML algorithm works.

First, a library of already-labeled drum samples has 'trained' the *OpenMusic* patch to classify a sound in one of seven different sound types (this is the 'training set'). Then, a segment of input audio (from the 'testing set') is analyzed (converted from audio signal to a vector of numbers) and compared against the data in the training set (see Figure 9). Here, the classification algorithm assigns a class-ID to the testing set vector according to

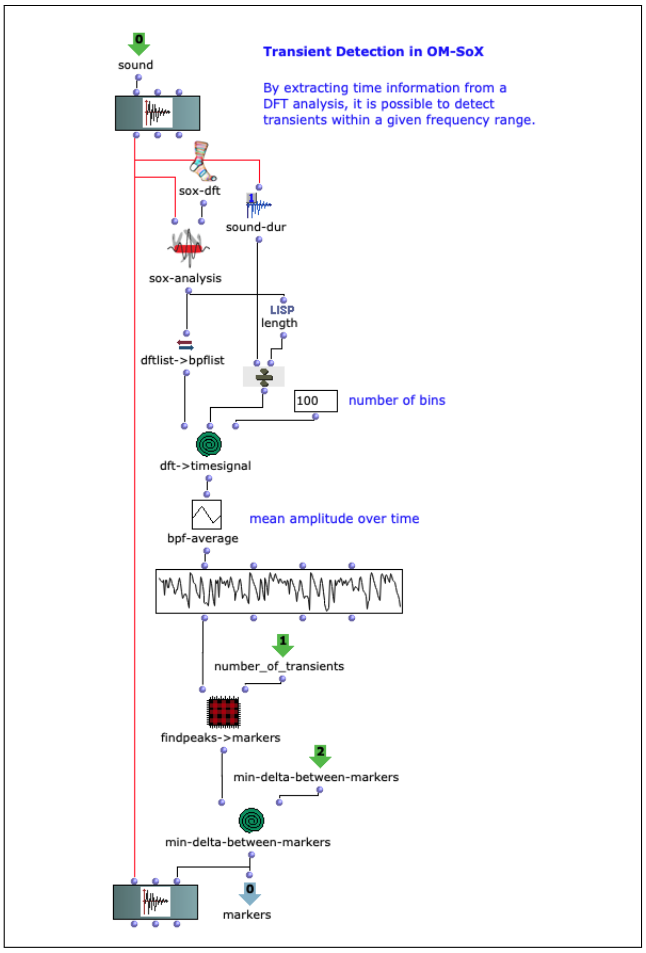
Transient Detection in OM-SoX
By extracting time information from a DFT analysis, it is possible to detect transients within a given frequency range.
sound
sox-dft
sound-dur
sox-analysis
LISP length
dftlist->bpflist
100
number of bins
dft->timesignal
bpf-average
mean amplitude over time
number_of_transients
findpeaks->markers
min-delta-between-markers
min-delta-between-markers
markers

Figure 8: The amplitude is derived from a DFT analysis, making it possible to detect transients within specific frequency bands.

the class-ID's of the nearest vectors from the training set. This method is considered 'supervised' because there is a training set which the author of the algorithm has arbitrarily designated what class-IDs correspond to what data. An important corollary to this is that the act of classifying means that the algorithm is returning a discrete value (the class-ID. For this method there is no result that is "in-between" one sound type and another.

This algorithm does not handle any actual audio signal. Before any data is processed, it is first analyzed and assigned a number of values that describe it. These are called audio descriptors, and they are often derived from spectral information in the signal. In our project, we used the discrete Fourier transform (DFT) analysis to extract spectral centroid, spectral difference, covariance, and Mel-frequency cepstrums of the audio signal. Figure 10 compares a sound to its vector and eventual classification.

2.1.3 *Classification Training Phase*

We train our *OpenMusic* patch on a relatively small set of audio: 105 classified drum samples (fifteen samples of seven different drum sound-types), assigning MFCC data to each of them. The audio used in the testing set does not overlap with the training set. However, it does come

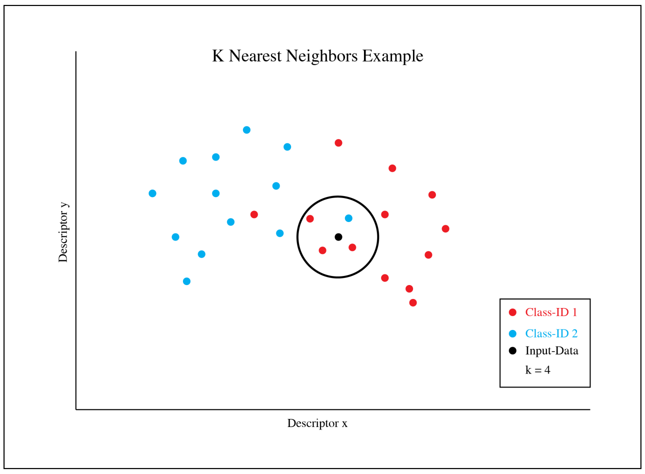

Figure 9: A data point is tested against a training set of labeled data to determine what class it belongs to.

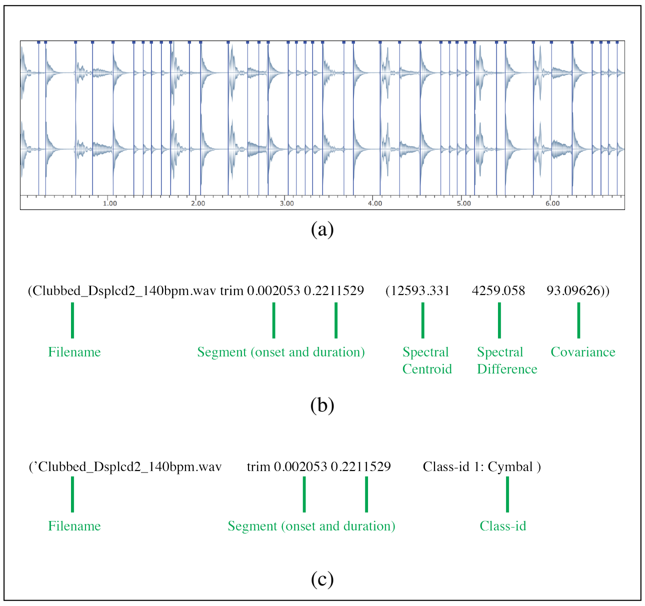
1.00
2.00
3.00
4.00
5.00
6.00
(a)
(Clubbed_Dsplcd2_140bpm.wav trim 0.002053 0.2211529 (12593.331 4259.058 93.09626))
Filename
Segment (onset and duration)
Spectral Centroid
Spectral Difference
Covariance
(b)
('Clubbed_Dsplcd2_140bpm.wav trim 0.002053 0.2211529 Class-id 1: Cymbal)
Filename
Segment (onset and duration)
Class-id
(c)

Figure 10: (a) The segmented sound before analysis. (b) The vector that references the sound segment. (c) The assigned class-ID after classifying the vector.

from the same recording session. Having a common source instruments and microphones between training and testing sets is an important necessity when the training set is small. This helps ensure that the classification algorithm is not influenced by characteristics of the audio signal not relevant to the application. For example, the frequency response of a microphone, the spectral content of the background noise, and the specific timbre and tuning of the drums recorded all influence the audio's signal and consequential MFCC data.

Our decision to use a small training set aligns with our interest in keeping a low barrier of entry for users new to ML methods. While these applications also function with large training sets [8] , such sets typically carry their own sets of concerns. Every data set brings different results to an

[8] such as MedleyDB, OpenMIC-2018, URBAN-SED, Urban Sound Datasets, for example.

algorithm, and thus it becomes especially important to interrogate the audio features each of these sets are curated around and annotated with [8].

2.1.4 *Classification Testing Phase*

Audio from the testing set is analyzed and converted to vectors of audio descriptor data in the same manner as the training set. The testing set data is processed with a k-nearest neighbors (knn) classification algorithm. In summary, a knn algorithm compares the distance between the given testing set vector and a number (k) of the nearest vectors from the training set. The effectiveness of a knn algorithm depends on how this distance is calculated.

We used an n-dimension euclidean distance function for our knn algorithm (as well as for our other, unsupervised clustering algorithm). Drawing from research on corpus-based concatenative synthesis [3], we determined that an n-dimension euclidean distance function was the most appropriate function for our specific goals of classifying and querying audio based on DFT analysis. Our *OpenMusic* patches include functions for extracting four audio features (MFCCs, spectral centroid, spectral difference, and covariance). However, it is possible for any number of audio descriptor functions to be incorporated. For example, any of *OM-SoX*'s 33 built-in audio descriptor analysis functions can also be used. Referencing classification methods used by Artemi-Maria Gioti [9], we used MFCCs in our examples because of their usefulness in classifying the timbre of audio.

2.2 Texture Synthesis

Our second technique uses an unsupervised clustering method to synthesize sound textures from a large library of unclassified audio (we refer to texture synthesis as the reconstitution of certain sound characteristics via smaller preexisting sound elements, see e.g. [10] for an overview of different approaches). A sound library of any size is analyzed, clustering the individual sounds into groups based on one or more audio descriptors. These clusters are presented in a *class-array* and a *3DC-library*, two *OpenMusic* objects that can provide alternative representations of multiple parameters of a sound all at once (see Figure 11). This allows a user to view and organize large amount of data for texture synthesis. The synthesis process is executed through a collection of sorting and processing functions in which a user is capable of both orchestrating broad gestures involving many sounds, and arranging individual sounds.

2.2.1 *Curating Sounds for Clustering*

The results of this clustering method are influenced primarily by what audio is curated at the start. This method is 'unsupervised', meaning that the input of the audio is compared against itself. There is no external 'training-set' of data by which the processing of the input data is supervised. Rather, the clustering method analyzes a set of data and makes a network of connections between each sound.

This clustering method analyzes a three-dimensional vector of audio descriptors (spectral centroid, spectral difference, and covariance). We found that grouping multiple different descriptors such as these can lead to unexpected and interesting clusters. The idea here is that the sounds are not simply sorted along a single parameter (pitch, noise, harmonicity). But rather are grouped along a metric that combines multiple parameters.

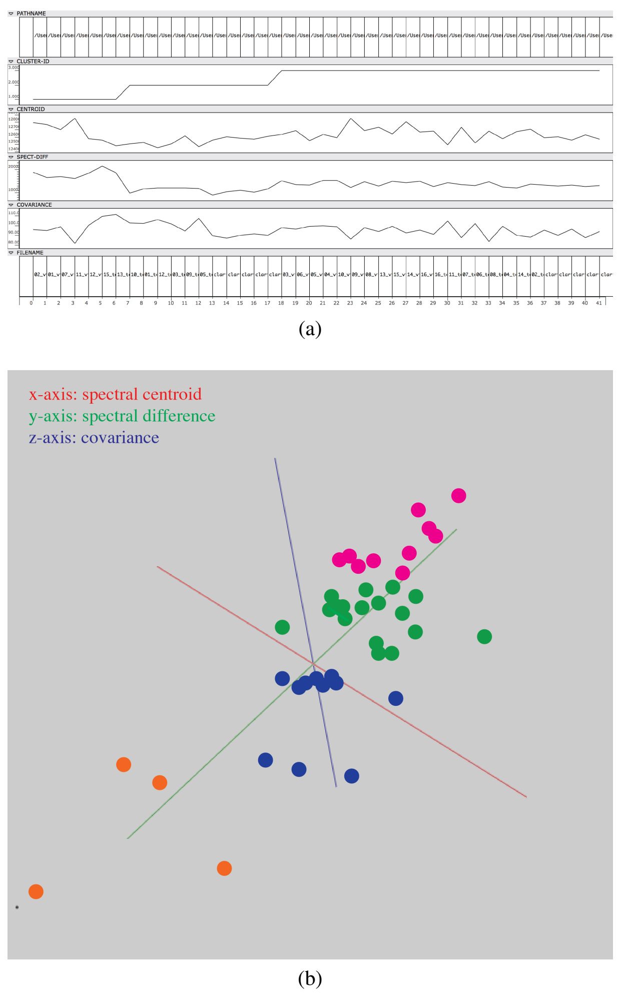

Figure 11: (a) A *class-array* object illustrating the pathname, cluster-id, audio features, and segmentation information of the input audio library. (b) A 3DC-library visualizing the four clusters of sounds in a three-dimensional space.

We found that the most interesting results of this method often come when the designated number of cluster groups is different than the number sound-types in the input library. This "encourages" the algorithm to find similarities between sounds that are not from the same sound type. For example, with a 3-mean cluster method (i.e. the audio is clustered into three groups), an audio library consisting of cymbals, clarinet multiphonics, and field recordings clearly groups itself according to its sound type. Sound-type (or, timbre) has such a strong influence over our three audio features. However, this same three-mean cluster method, when applied to only clarinet multiphonics suggests for us an unorthodox, but interesting, way of grouping this relatively homogeneous set of sounds into three groups.

2.2.2 *Clustering Method*

The k-means clustering algorithm is a classic method for unsupervised machine learning. In our context, a clustering algorithm is used to group the input sounds. A 'mean' in this context refers to the mean value of a given cluster. For example, a 3-mean clustering algorithm will return 3 clusters (see Figure 12). The k-means clustering method is an iterative 4-step process in which 1) a number (k) of random 'means' are generated and each sound from our input is assigned to its nearest mean value. 2) This forms a number (k) of groups. The 'centroid' of a given group is calculated and that becomes a new mean value of that group. 3) step 1 is repeated with the new mean. The vectors are grouped anew, according to their distance from the closest of the new means. And then step 2 is done, evaluating the centroid of these new groups and generating a new set of k means from that. 4) After multiple iterations, the groups eventually settle into an unchanging set. Further iterations only produce the same clusters and means. This indicates that the process is complete.

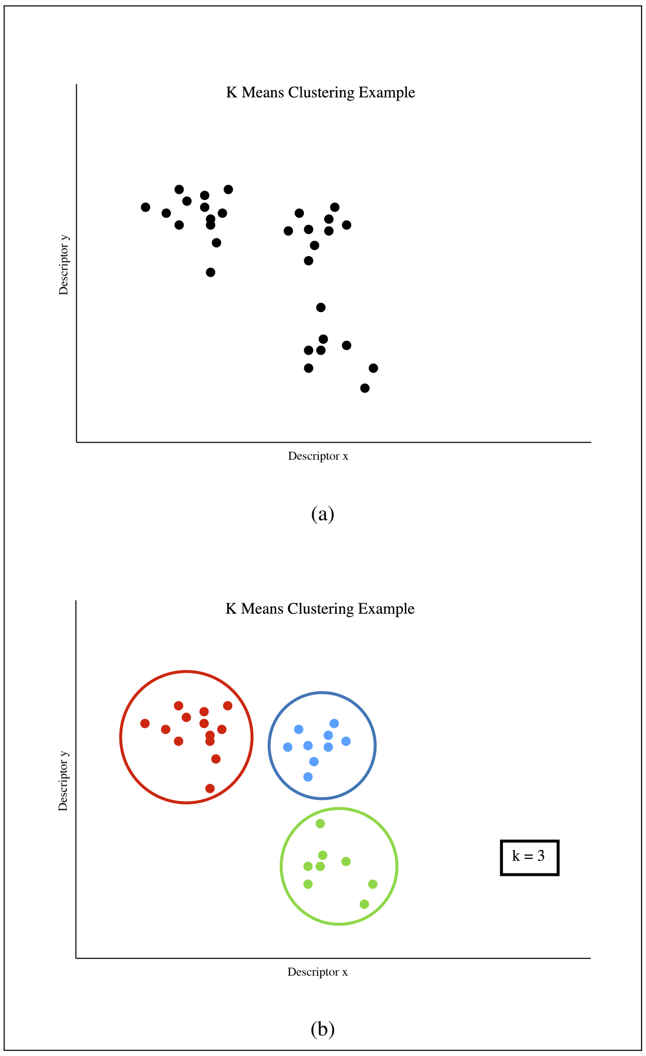

Figure 12: (a) A collection of unlabeled data (b) That same data clustered around three 'means'.

This algorithm may return different results each time it is executed, even when using the same input data. This is due to the random means generated in step 1. For our purpose of organizing sound against multiple parameters at once, we found it preferable to have an algorithm that provided varying outputs upon each return. Similar to using seeds in pseudo-random functions, this k-means clustering algorithm could potentially be controlled more tightly through the use of seeds for the randomly generated mean. Despite that, by representing and indexing the returned clusters in a 3DC-library and class-array object, a user is able to easily evaluate the clusters, allowing them to make adjustments to their input library and k-value, and iterate the clustering algorithm further until a desired grouping is found. Any need to save or replicate a particular output from the clustering algorithm is able to be saved by exporting the class-array as an OM-instance.

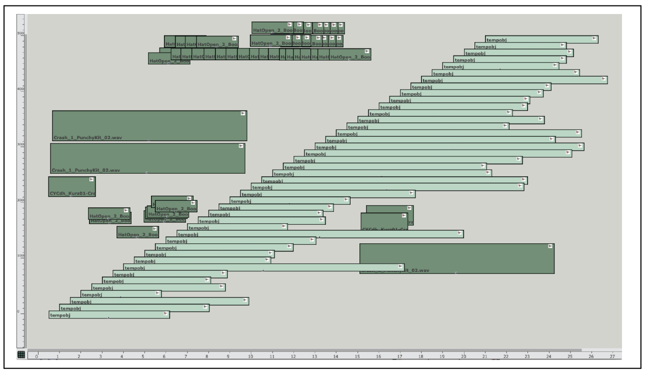

Figure 13: Texture synthesis with a *maquette* object. The cluster-sorted pitched-instrument sounds are the temp boxes in the maquette, while the sound boxes denote hand-placed cymbal samples. In this example, the y-axis does not influence the sound.

2.2.3 *Sorting and Calling Sounds for Synthesis*

Once clustered and stored in the class-array object, the sounds are selected and organized in a maquette, where they can be subsequently edited and synthesized. There are a number of functions for organizing audio in the maquette. Firstly, the order of the clustered sounds can be sorted by cluster-id, as well as by any other dimension in the class-array (e.g. covariance). Once sorted, any number of cluster groups can be called from the class-array to be placed in the maquette. The selected groups of sounds are then arranged in the maquette according to a list of onsets. Finally, it is also possible to hand-pick and arrange individual sounds in the texture, via a pop-up menu that indexes all the sounds in the patch. This is a useful tool for balancing out broad algorithmic gestures with micro, 'hand-written' ones. Figure 13 illustrates the accompanying example output audio: a collection of saxophone, clarinet, and violin sounds are sorted into three cluster groups. This creates a gradual nuanced harmonic texture that ebbs between noise and harmony. This homogeneous texture is then punctuated by individually placed cymbal sounds called by the pop-up menu.

3. CONCLUSIONS

Our hope is that these applications inspire and lower the barrier of entry for composers interested in incorporating ML methods into their own CAC workflows. Particularly, we hope that the modularity in these applications helps users develop new aesthetic approaches to ML methods. A variety of interfaces common in the *OpenMusic* environment were incorporated into the applications as forms of higher-level, complementary representations for controlling synthesis processes. These interfaces can be seen as a form of notation not only in a symbolic format (i.e. the multi-seq and maquette), the class-array can be understood as a tabulated notation, and the 3DC- and BPF-libraries as a form of geometrical notation. This variety aligns with the modular intentions of this project.

This project is still in its early stages, with several possible directions for further development. One current limitation of these applications is that they do not break out of the medium of sound synthesis. While there are notation-based parts in these applications, they are limited to mediating the journey of beginning with input sounds and ending with new sounds. Functions for score synthesis would be a clear step forward and would provide one further level of integration for a composer's CAC environment. A score synthesis function would be particularly useful for the simultaneous notation and synthesis of electronic parts.

If the applications were evaluated purely on their accuracy of timbre classification, then the use of classic ML methods can also be considered a limitation. For example, incorporating few-shot object detection into the classification application would likely increase its accuracy and efficiency in transcribing drum hits.

Acknowledgments

The first author would like to thank the second author, who has not only provided generous advisement, support, and resources for this research, but also developed several of the software tools used as well as theoretical models that supported the foundation of this project. The first author also thanks the students he works with at the Institute for Music Informatics at the Hochschule für Musik Karlsruhe. Their interest has provided fresh insight in developing the modular nature of this project's software design. The applications presented in this paper are available to the community for download, along with accompanying audio. [9]

The sounds for training, testing, and clustering were curated from: Mark Guiliana Drums by Loop Loft (drums) [10], Musicradar.com (drums) [11], Freesound.org (field recordings) [12] [13] [14] [15] [16], Marcus Weiss (tenor saxophone) [11], and Gerhard Krassnitzer (clarinet) [12]. The violin harmonics were produced by the first author.

[9] `https://edu.marlonschumacher.de/audio-ml-4-cac/`

[10] `https://www.thelooploft.com/products/mark-guiliana-drums?variant=959418489`

[11] `https://www.musicradar.com/news/drums/1000-free-drum-samples`

4. REFERENCES

[1] M. Schumacher, "A Framework for Computer-Aided Composition of Space, Gesture, and Sound. Conception, Design, and Applications." Ph.D. dissertation, McGill University, Music Technology Area, Schulich School of Music, Jun. 2016.

[2] R. A. Fiebrink and B. Caramiaux, *The Oxford Handbook of Algorithmic Music*. Oxford University Press, 2018, ch. The Machine Learning Algorithm as Creative Musical Tool.

[3] D. Schwarz, "Corpus-Based Concatenative Synthesis," *Signal Processing Magazine*, pp. 92–104, 2007.

[4] B. Hackbarth, N. Schnell, and D. Schwarz, "Audioguide: a Framework for Creative Exploration of Concatenative Sound Synthesis," Tech. Rep., Mar. 2011.

[5] M. Schumacher, "Ab-Tasten: Atomic Sound Modeling with a Computer-Controlled Grand Piano," in *The OM Composer's Book: Volume 3*, J. Bresson, G. Assayag, and C. Agon, Eds. Éditions Delatour France / IRCAM—Centre Pompidou, 2016, pp. 341–359.

[6] A. Vinjar, "OMAI: An AI Toolkit For OM#," in *Linux Audio Conference*, Bordeaux, France, November 2020, pp. 12–15.

[7] J. Garcia, J. Bresson, M. Schumacher, T. Carpentier, and X. Favory, "Tools and Applications for Interactive-Algorithmic Control of Sound Spatialization in OpenMusic," https://hal.archives-ouvertes.fr/hal-01226263, Karlsruhe, Germany, 2015, [Online; accessed 5-January-2022].

[8] J. Salamon, "What's Broken in Music Informatics Research? Three Uncomfortable Statements," https://slideslive.com/38917449, Long Beach, California, USA, June 2019, [Online; accessed 5-January-2022].

[9] A. M. Gioti, "Imitation Game: Real-time Decision-making in an Interactive Composition for Human and Robotic Percussionist," https://www.researchgate.net/publication/340979353_Imitation_Game_Real-time_Decision-making_in_an_Interactive_Composition_for_Human_and_Robotic_Percussionist, New York, USA, June 2019, [Online; accessed 5-January-2022].

[12] `https://freesound.org/people/Jack_Master/sounds/384466/`

[13] `https://freesound.org/people/Grupo2SONIDO/sounds/255214/`

[14] `https://freesound.org/people/doxent/sounds/386009/`

[15] `https://freesound.org/people/man/sounds/29580/`

[16] `https://freesound.org/people/laribum/sounds/353025/`

[10] D. Schwarz, “State of the Art in Sound Texture Synthesis,” in *International Conference on Digital Audio Effects*, Paris, France, 2011, pp. 221–231.

[11] M. Weiss and G. Netti, *The Techniques of Saxophone Playing*. Bärenreiter, 2010.

[12] G. Krassnitzer, *Multiphonics für Klarinette mit deutschem System*. ebenos, 2002.