Make Your Own Audience: Virtual Listeners Can Filter Generated Drum Programs

Amir Salimi and Abram Hindle

Department of Computing Science
University of Alberta
asalimi@ualberta.ca
hindle1@ualberta.ca

Abstract. Can we generate drum synthesizers automatically? We present an approach for the automatic generation of synthesizer programs for one-shot percussive sounds. Recent advancements in digital synthesis, heuristic search, and neural networks can be utilized for sound generation. Yet the need for data, the problem of open set recognition, and high computational costs persist as barriers towards the expansion of sound libraries using these techniques. We generate quick, scalable, percussion synthesizers using classical signal processing. We train drum classifiers to find and classify synthesizer programs that mimic percussive sounds. We use features from Fourier transformations and autoencoder embeddings to train machine learning classifiers. Manual listening tests of the generated sounds demonstrates the system can successfully generate drum synthesizers and categorize drum sounds. To facilitate future research, we share our curated dataset of free percussive sounds.

Keywords: Automatic Synthesizer design. Machine Listening. Sound Analysis. Novelty and Originality

1 Introduction

Digital recordings of novel, one-shot[1] drum sounds are commonly used in electronic music compositions. Yet unique drum sounds can be difficult or expensive to find. By relying on recordings of material drum sounds, artists are limited by what instruments exist in the material world and whether or not high-quality, one-shot recordings can be accessed. We believe automatic programming of virtual synthesizers for the creation of novel drum sounds can alleviate these limitations. To this end, we implement a programmable virtual synthesizer of audio, which we call the *virtual synthesizer*. We also implement machine learning classifiers for automatic separation of percussive sounds from non-percussive sounds. To effectively train the classifiers with small datasets, we simplify the representation of audio data by experimenting with fast Fourier transforms (FFT) and autoencoder embeddings. We call our system of feature extraction and classification of sounds the *virtual ear*. To create a dataset of synthetic drums, the

[1] A single hit on the drum that captures its capabilities

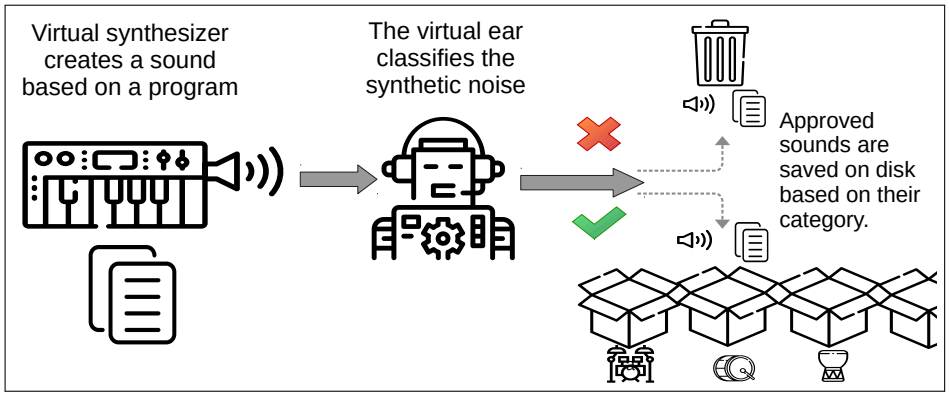


Fig. 1: A blueprint of our system which allows each component to be implemented in a number of ways. Our implementations of this pipeline allow for easy parallelization when needed. This implementation also allows for easy integration of heuristic search for future works.

Table 1: Curated databases

DB Name	Categories
FreeDB	Kicks:533 - Snares:372 - Claps:230 - Hats:105 - Other:281
RadarDB	Kicks:1054 - Snares:842 - Claps:353 Toms:349 - Hats:1561 - Rims:131 - Shakers:121
MixedDB	Kicks:533 - Snares:372 - Claps:230 - Hats:105 - Others:281

virtual synthesizer produces random programs and generates the corresponding audio. This audio is then categorized by the virtual ear. We save not just the desirable sounds, but also the programs which generated these sounds, which can be modified and experimented on by sound designers. In this work, only random search is used for program generation, but other heuristics can be integrated for an improved search algorithm in future works. We build generative systems using different implementations of the virtual ear, and conduct manual, blinded hearing tests of the generated sounds. Our results are promising as the majority of sounds generated from our systems are deemed percussive by human listeners. However, we cannot quantify the novelty of these sounds.

Furthermore, we provide a subset of our curated datasets to facilitate future research. We curated 3 datasets of one-shot drums which we use to train and validate drum classifiers. The number of samples per group in these databases is provided in Table 1. We provide unrestricted access to **FreeDB** via Zenodo[2]. **RadarDB** cannot be shared directly, we provide the script for its automated

[2] `https://zenodo.org/record/3994999`

creation[3]. We cannot provide a copy of **MixedDB**, which is curated from various sources and personal libraries.

2 Related Works

Numerous deep neural network models have been proposed for the purpose of signal generation in recent years (Engel et al., 2017; Yamamoto, Song, & Kim, 2020; Oord et al., 2017; Yee-King, Fedden, & d'Inverno, 2018; Ramires, Chandna, Favory, Gómez, & Serra, 2020). WaveGans and WaveNet have been subject to significant improvements and experiments since their proposal (Engel et al., 2017; Yamamoto et al., 2020; Oord et al., 2017). Particularly relevant works are the utilization of variational autoencoders (VAE's) for generation of percussive samples (Aouameur, Esling, & Hadjeres, 2019) and generation of percussive sounds by decoding a small set of latent features (Ramires et al., 2020). Automatic programming [4] of virtual synthesizers has long been a topic of interest. Genetic Algorithms (GA's) have been utilized for the generation of new sounds with various sound engines (Horner, Beauchamp, & Haken, 1993; Macret, Pasquier, & Smyth, 2012). A more recent work by Yee-King et al. (Yee-King et al., 2018) used Long short-Term Memory (LSTM) models and heuristic search to find the exact parameters used to create a group of sounds. The sounds approximated were made by the same virtual synthesizer and not with an external source. Esling et al. used over 10,000 VST synthesizer presets to learn a parameter space which can be sampled for creation of new audio (Esling, Masuda, Bardet, Despres, et al., 2019). Here, we work towards the approximation of percussive sounds with no prior knowledge about the parameter space of a synthesizer.

3 Virtual Synthesizer Design

To create sounds, we build digital synthesizers. We use classical DSP to build our synthesizer, which allows for quick, offline, and parallel generation of audio signals without the usage of GPUs. We made extensive use of Pippi[5] and SciPy (Jones, Oliphant, Peterson, et al., 2001) libraries. Our virtual synthesizer contains a set of one or more submodules. Each submodule is a self-contained noise making unit and creates signals by taking the steps depicted in figure 2. Submodules have identical sets of parameters, but widely different outputs can be achieved depending on the values assigned. The set of required parameters for each submodule is highlighted in Table 2. The sonic output of the virtual synthesizer is the normalized addition of the output of its submodules. Our implementation of a synthesizer can have any number of submodules. We call the number of submodules in each virtual synthesizer the *stack size*. We call the sets of parameter values that characterize a synthesizer's submodules a *program*(analogous to presets and submodules for a VST). Rather than directly

[3] `https://github.com/imilas/Synths_Stacks_Search`

[4] unsupervised generation of source code or programs towards a goal

[5] https://github.com/luvsound/pippi

Table 2: Synthesizer submodule parameters. 10^{15} unique programs are possible. Each synthesizer can contain any number of submodules

Parameters	Value Range	Notes and Constraints
Attack	0-3	A-D-S-R values relative
Decay	0-3	relative to A-S-R
Sustain	0-3	relative to A-D-R
Release	0-3	relative to A-D-S
OSC type	sine,square,saw	-
IsNoise	boolean	generate noise using cloud of waveform
Length	0-1 second	-
StartTime	0-1 second	Length+Start<1
Amplitude	0.1-1	1 = max amplitude
Pitches(notes)	list of pitches	range of C0(16.35hz) to B9
HP filter Cutoff	0-20000hz	-
LP filter Cutoff	20000hz-HP	never lower than HP cutoff
Filter Order	4,8,16	butterworth filter order

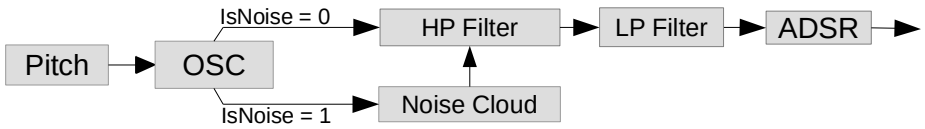


Fig. 2: High level representation of pre-rendering steps for each submodule. Each Synthesizer contains 1 or more these submodules. Synthesizer programs set the number of these submodules and their parameters. The output of a synthesizer is the normalized addition of all its submodule outputs.

using neural networks for sound synthesis, we generate programs for this virtual synthesizer. Our decision is based on the following factors: (i) *Novelty and Creativity*: The goal here is to work with the limitations of any tractable sound source to create its approximations of a given sound category. We seek to create novel sounds via artificial, exploratory creativity. Boden defines this concept as an emergent property of generative work within confined rule sets (Boden, 2009). An example is the perpetual popularity of 8-bit aesthetics (Collins, 2007). (ii) *Interpretability*: Neural networks are often described as black boxes with uninterruptible weights (Basheer & Hajmeer, 2000). Their highly recursive structure makes modern explanation methods such as saliency maps unreliable (Rudin, 2019). (iii) *Speed of Rendering*: Neural network synthesis is costly; Sub 24 khz sample rates are common in most relevant works (Yamamoto et al., 2020; Oord et al., 2017; Aouameur et al., 2019; Ramires et al., 2020). This is far below CD quality sampling rates (Reiss, 2016). At our fixed sampling rate of 48 khz, synthesizers with 8 submodules can create and save 1 second sounds to hard-disk

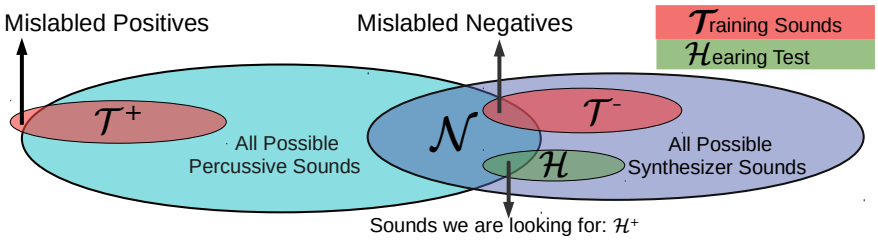


Fig. 3: An illustration of the discrepancy between the sounds we use to train our classifiers and the type of sounds the classifier is expected to classify. $\mathcal{N}$ is the set of percussive sounds a synthesizer is capable of making. The inclusion of sounds in this group may vary from person to person. Our positive samples, $\mathcal{T}^+$, is a small fraction of a wide variety of percussive sounds that are conceivable. For $\mathcal{T}^-$, we can generate any number of random samples. $\mathcal{H}$ is a series of sounds sent to the ear for classification.

with an average rendering time of 50 milliseconds[6]. (iv) *Flexibility and Scaling*: Probabilistic audio generation is often done sequentially. State of the art, parallel wave generation with GANs requires a fixed amount of rendering time for each time-step (Yamamoto et al., 2020). With our virtual synthesizer, the added footprint of increasing the length of rendered sounds or higher sampling rates is relatively minuscule.

4 Virtual Ear

The virtual ear receives sounds and discards those which do not resemble drums. It then classifies sounds by drum category. Our goal is to produce a virtual ear that accurately discriminates between percussive and non-percussive sounds, but is tolerant of novelty. The virtual ear makes two critical decisions:

Decision.1 Could the sound be used as a drum?
Decision.2 If it does sound like a drum, what type of drum should it be?

Decision.1 requires knowledge of what drums **do not** sound like, or knowledge of an infinitely large set, which cannot be fully represented via examples. An important consideration is that the source of sounds used for training the model (organic drum sounds) will be fundamentally different from the source of unlabeled sounds we wish to categorize (noise from a synthesizer). This issue is reflective of the open set recognition (OSR) problem (Geng, Huang, & Chen, 2020; Mundt, Pliushch, Majumder, & Ramesh, 2019). We use Figure 3 to highlight a number of caveats with our training approach. If the sound is deemed percussive, the virtual ear makes *Decision.2* by finding the best drum category

[6]Using a single process on a Macbook Air 2012 and Ubuntu 18.04

Table 3: Overview of survey model architectures. Further details in Appendix C

Model	Architecture Layout	Features
CNN(TPE)	CNN(2 channel convolution) -> LSTM(800 hidden states) -> Linear Layers (shape 400x5)	Spectrogram
FC(TPE)	Fully connected Linear layers with shape: 400x10x5x10x5	Spectrogram
E+F(TPE)	Fully connected network of size 50x10x2x5	Env. + Freq.
MEM	Encoder -> Embedded Spectrogram(size 64) -> Extra-Trees	Spectrogram

for the sound. The number of categories available is dependent on the database of drums used for training. To maximize the transference of knowledge gained from training the classifiers to evaluation of programs, we need to extract concise feature sets that capture fundamental characteristics of the data points.

In order to classify sounds, we need to summarize them as features. Various works have demonstrated effective reconstruction of signals given their Short-time Fourier Transforms (STFT) (Nawab, Quatieri, & Lim, 1983; Griffin & Lim, 1984). If the STFT of a signal can be used for its reconstruction, perhaps it can be utilized as a source of fundamental features (Lee, Pham, Largman, & Ng, 2009; Huzaifah, 2017). We defined 3 STFT (or FFT) transformation functions to capture important features of percussive sounds.

1. Envelope Transformation: Represents changes in loudness for the duration of the signal. Using STFT we generate a matrix $M_{i\times j}$ with rows i and columns j corresponding to time steps and frequency bins. Values $v_{i\times j}$ indicate the magnitude of frequency bin j at each time-step i. We approximate the envelope of the signal by summing the values of M at each time-step.
2. Frequency Transformation: Similar to envelope calculation, but the summation is done along the frequency axis. Shorter hop-sizes and wider windows were used to increase frequency resolution.
3. Spectrum Transformation: Mel Scaled STFT. Values normalized from 0-1.

Another set of features were extracted using the latent embeddings of autoencoder networks. Here, an autoencoder is a combination of an encoder which encodes spectrograms into a lower dimension and a decoder which tries to decode these embeddings into the original data. We trained a number of autoencoders with spectrogram transformations, and used the encoder from the autoencoder network with the lowest decoding loss as a feature extractor [7]. Latent embeddings at the bottleneck layer of our best spectrogram autoencoder are used as features for sound categorization.

To automatically learn from the extracted features, two groups of virtual ears are implemented: *two phased ears* (TPEs) and *Mixed Ear Models* (MEMs). For TPEs, we train multiple neural network architectures using different subsets of the FFT features to specialize in *Decision.1* or *Decision.2* and combine them to make decisions sequentially. To train TPEs for *Decision.1*, we use all

[7] Appendix D has further details on the hyperparameter search process

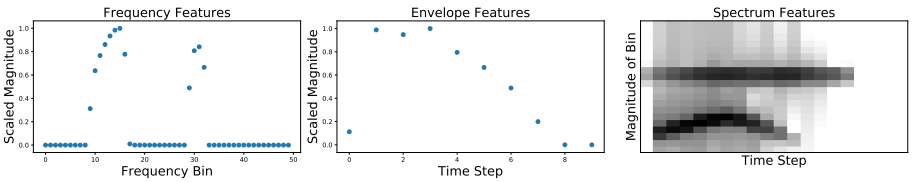


Fig. 4: Visualized representation of FFT features for a randomly synthesized noise. More examples in Appendix B.

drums in RadarDB and FreeDB and 6000 examples of virtual synthesizer noise. For *Decision.2*, we combine the two databases and merge toms into kicks and rims/shakers into "other". We trained the TPE models with 80% of this dataset. Using the remaining 20% of sounds, we achieved 98% accuracy in *Decision.1* and 82% accuracy in *Decision.2* with our best models. These accuracy numbers are weak as we did not account for category sizes or cross validate.

MEMs learn from autoencoder embeddings and envelope features and give simultaneous answers to both decisions. We use kicks, snares, claps, and hat sounds from RadarDB and FreeDB as well as 1000 examples of synthesized noise to train MEMs. Using 10-fold cross validation, our best MEM (an ExtraTrees Classifier) achieves the average f1-score of 92% in 5-way sound classification (4 drum categories + synthesized noise). Detailed measurements and model descriptions for TPEs and MEMs can be found in table 3 and Appendix C.

5 Building and Evaluating The System

High accuracy in detection of organic drums vs synthetic sounds does not necessarily make an ear more suitable for our generative system. Perfect detection would create a system which generates nothing. We cannot know the true size of $\mathcal{H}^+$ (see Figure 3) without exhaustive manual hearing tests. We conduct blinded hearing tests of two generative systems. For *Decision.1*, a network which learns from spectrogram data utilizing CNN and LSTM layers is used. For *Decision.2*, we categorize the drum types with 3 different models: FC (Fully-Connected architecture and spectrograms for learning), CNN (CNN and LSTM layers with spectrograms), and E/F (Fully-Connected architecture with frequency and envelope features). The MEM system uses our best MEM, which simultaneously classifies sounds as drums and categorizes them.

The TPE system produces samples in the following categories: "snare", "kick", "hat", "clap" and "other" (combination of rims, shakers and unusual percussive sounds). The MEM system does not output the "other" category, yet the option is available to surveyors when categorizing sounds manually. Both authors categorized a subset of the results without knowledge of the assigned labels. Additionally, each responder has the option of labeling samples as "bad" for samples that they deemed not percussive.

Table 4: Fleiss' kappa coefficients as a measurement of agreement between persons (H+H) and persons with models.

System's Ear Type	Drop Rule	Size	HvH	H+FC	H+CNN	H+E/F	H+MEM
TPE	No Drops	257	0.37	0.35	0.36	0.36	
	All Bad and Other	154	0.47	0.59	0.54	0.50	
MEM	No Drops	300	0.34				0.25
	All Bad and Other	120	0.62				0.59

The percentage of sounds labeled as "bad" by the authors is a measurement of success with regard to *Decision.1*. We measure success with regard to *Decision.2*—the reliability of agreement between persons and drum categorization models—via the Fleiss' kappa coefficient (Fleiss, 1971). The value of 0 or less for this coefficient indicates no agreement beyond random chance, and the value of 1 indicates perfect agreement. We re-measure this coefficient after dropping all "bad" and "other" samples.

The majority of sounds created by our system are deemed percussive by both surveyors. 30% of the outputs from the TPE system and 50% of the MEM system outputs are deemed as non-percussive by at least one person. This leaves much room for improvement with regards to *Decision.1*. Our performance with regards to *Decision.2* is promising as we achieve moderately high agree-ability scores after dropping "bad sounds" and "other". Despite the drawbacks, the MEM pipeline remains competitive while working with a fraction of the features to learn from and simultaneously making both decisions. Extended survey analysis is provided in Appendix E. Appendix A provides access to generated sounds.

6 Conclusion, Validity, and Future Work

Conclusion: We built a generative system for creation of percussive sounds via automatic programming of virtual synthesizers. We verified its results with human listeners. Our work enables not only the creation of new libraries of percussion sounds, but new synthesizer programs which can be modified and studied. Manual listening tests revealed much room for improvement, particularly with accurate separation of percussive sounds from the infinitely large set of non-percussive sounds. We had some success in our utilization of latent representations of autoencoder networks as low-dimensional representations of short sound files.

Threats to Validity: The lack of consistency in training and accuracy measurements makes comparisons between TPEs and MEMs difficult. Many arbitrary design decisions have been made, particularly in the design of our TPE classifiers and selection of datasets for training and testing. We cannot quantify the novelty of our results. Our requirement for training data remains high.

Future Work: Effective implementation of few-shot learning is a priority. Program generation may be improved by reinforcement learning and other heuristics.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th acm sigkdd international conference on knowledge discovery & data mining* (pp. 2623–2631).

Aouameur, C., Esling, P., & Hadjeres, G. (2019). Neural drum machine: An interactive system for real-time synthesis of drum sounds. *arXiv preprint arXiv:1907.02637*.

Basheer, I. A., & Hajmeer, M. (2000). Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, *43*(1), 3–31.

Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural computation*, *12*(8), 1889–1900.

Boden, M. A. (2009). Computer models of creativity. *AI Magazine*, *30*(3), 23–23.

Collins, K. (2007). In the loop: Creativity and constraint in 8-bit video game audio. *Twentieth-Century Music*, *4*(2), 209.

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Eck, D., Simonyan, K., & Norouzi, M. (2017). *Neural audio synthesis of musical notes with WaveNet autoencoders.*

Esling, P., Masuda, N., Bardet, A., Despres, R., et al. (2019). Universal audio synthesizer control with normalizing flows. *arXiv preprint arXiv:1907.00971*.

Fleiss, J. L. (1971). Measuring nominal scale agreement among many raters. *Psychological Bulletin*, *76*(5), 378.

Geng, C., Huang, S.-j., & Chen, S. (2020). Recent advances in open set recognition: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Griffin, D., & Lim, J. (1984). Signal estimation from modified short-time fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *32*(2), 236–243.

Horner, A., Beauchamp, J., & Haken, L. (1993). Machine Tongues XVI: Genetic algorithms and their application to FM matching synthesis. *Computer Music Journal*, *17*(4), 17-29.

Huzaifah, M. (2017). Comparison of time-frequency representations for environmental sound classification using convolutional neural networks. *arXiv preprint arXiv:1706.07156*.

Jones, E., Oliphant, T., Peterson, P., et al. (2001). Scipy: Open source scientific tools for python.

Lee, H., Pham, P., Largman, Y., & Ng, A. Y. (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems* (pp. 1096–1104).

Macret, M., Pasquier, P., & Smyth, T. (2012). Automatic calibration of modified fm synthesis to harmonic sounds using genetic algorithms. In *Proceedings of the 9th sound and music computing conference.* Copenhagen, Denmark.

Mundt, M., Pliushch, I., Majumder, S., & Ramesh, V. (2019). Open set recognition through deep neural network uncertainty: Does out-of-distribution detection require generative classifiers? In *Proceedings of the ieee international conference on computer vision workshops.*

Nawab, S., Quatieri, T., & Lim, J. (1983). Signal reconstruction from short-time fourier transform magnitude. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *31*(4), 986–998.

Oord, A. v. d., Li, Y., Babuschkin, I., Simonyan, K., Vinyals, O., Kavukcuoglu, K., ... others (2017). Parallel Wavenet: Fast high-fidelity speech synthesis. *arXiv preprint arXiv:1711.10433*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... others (2011). Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, *12*, 2825–2830.

Ramires, A., Chandna, P., Favory, X., Gómez, E., & Serra, X. (2020). Neural percussive synthesis parameterised by high-level timbral features. In *Icassp 2020-2020 ieee international conference on acoustics, speech and signal processing* (pp. 786–790).

Reiss, J. D. (2016). A meta-analysis of high resolution audio perceptual evaluation. *Journal of the Audio Engineering Society*.

Rudin, C. (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, *1*(5), 206–215.

Yamamoto, R., Song, E., & Kim, J.-M. (2020). Parallel WaveGAN: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram. In *ICASSP 2020-2020 IEEE international conference on acoustics, speech and signal processing* (pp. 6199–6203).

Yee-King, M. J., Fedden, L., & d'Inverno, M. (2018). Automatic programming of VST sound synthesizers using deep networks and other techniques. *IEEE Transactions on Emerging Topics in Computational Intelligence*, *2*(2), 150–159.

Appendices

A Dataset Details and Downloads

A.1 Download Instructions

FreeDB, survey data, and randomly generated sounds can be downloaded from:
`https://zenodo.org/record/3994999`

RadarDB requires lengthy downloads and processing. The script for its automated creation can be found under the "getting_data" directory of the project:
`https://github.com/imilas/Synths_Stacks_Search`
Also included on the project page is our full source-code, models, and extra output examples.

The latex source for this paper can be found at:
`https://github.com/imilas/CSMC-MuMe-2020`

A.2 Details

Our data sources are a large set of 2 second or shorter drum samples aggregated from personal libraries, free drum kits from the sample-swap project [8] which we organized into 5 groups, and a large set of drum sounds aggregated from royalty free sources such as music radar [9]. We put together 3 databases of drums using these sources. We also created a database of synthetic noise from 1, 3, and 5 stacked virtual synthesizers. Specifications about our datasets can be found in table 5.

We prioritize making generalizable tools which can learn from and produce a variety of different sounds. We utilize these databases depending on the task at hand. At times, we merged or purged drum groups to simplify tasks.

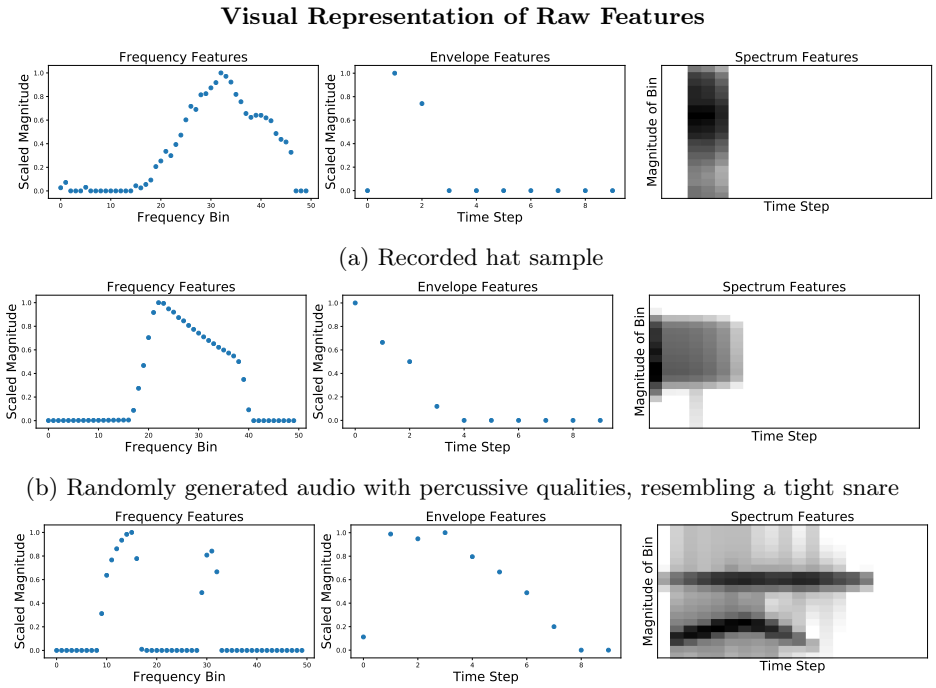
Table 5: Curated databases, including NoiseDB

DB Name	Categories
FreeDB	Kicks:533 - Snares:372 - Claps:230 - Hats:105 - Other:281
RadarDB	Kicks:1054 - Snares:842 - Claps:353 Toms:349 - Hats:1561 - Rims:131 - Shakers:121
MixedDB	Kicks:533 - Snares:372 - Claps:230 - Hats:105 - Others:281
NoiseDB	1 Stack:2000 - 3 Stacks:2000 - 5 Stacks:2000

[8] https://sampleswap.org/
[9] https://www.musicradar.com/

B Audio Transformation Functions

Figure 5 contains the graphed representation of features extracted for 3 different samples. Sample a is a recorded hat from our database. sample b is an example of randomly generated noise with percussive qualities that we found suitably similar to a snare sound. Sample c is an example of a randomly generated noise where the spectrum features are necessary for proper classification.



(a) Recorded hat sample

(b) Randomly generated audio with percussive qualities, resembling a tight snare

(c) A randomly generated noise with a percussive envelop but non-percussive frequency features (modulated pitch)

Fig. 5: Graphed representation of features extracted for 3 different samples.

C Virtual Ear Classifier Definitions

C.1 Two Phased models

Using the described features, we trained several neural network models for decision 1 and 2 with the pytorch library. The task of decision 1 is to separate drums from not-drums (DrumVsNotDrum, or DVN). The task of decision 2 is to categorize drums and percussion (DrumVsDrum, or DVD). We kept our feature space small, making it viable for feature selection and model design to be done on a trial and error basis. For all models, accuracy is calculated by prediction of all test dataset labels and the loss function and optimizer are Categorical Cross-Entropy and Adam respectively. Training continues until no reduction in loss and accuracy is observed in 10 epochs. All activation functions are PReLU:

1. FC-DVN: Fully connected network trained on Envelope features, reaching 97% accuracy on our test data for decision1. With size of 10x5x10.
2. CNNLSTM-DVN: A combination of CNN and LSTM models, where the CNN model extracts higher level features that are fed temporally to an LSTM cell. This model is trained on spectrum data and reaches 98% accuracy on our test set. Its structure is the combination of a CNN with 2 output channels and kernel size $(7, 3)$; Followed by an LSTM model of hidden size 800 and a fully connected layer of size 20x2.
3. E/F-DVD: A fully connected model trained on a concatenation of envelope and frequency features. Reaching 80% accuracy for 5-way drum categorization in decision2. Size of 50x10x2x6.
4. CNN-DVD: A CNN model trained on Spectrum features. Reaching 82% accuracy in a 5-way drum categorization in decision2. A combination of a CNN model with output channel size of 4, kernel of size of 5, another CNN model with output channel size of 8 and kernel of size 3. Followed by a fully connected network of shape 100x20x6.
5. FC-DVD: Fully connected 3 layer neural net with 78% accuracy for 5-way drum categorization in decision2. Size of 400x200x50.

Parameters are hand-picked and un-tuned.

C.2 Mixed Ear Models

We compare the performance of our models before and after the addition of envelope features (a vector of size 10) to the feature space. We reuse the model trained on MixedDB as our embedding feature extractor. We use a combination of RadarDB, FreeDB and NoiseDB for training. We only focus on clap,hat,kick,snares, and synthetic noise groups for measuring effectiveness to prevent class overlaps as much as possible. We also exclude samples longer than 1 second, to exclude potentially mislabeled data. Our final training database for mixed ear models is described in table 7.

Model name	Tuned Parameters†	Used Weights? ‡
Support Vector Classifier (SVC)	Gamma:0.001, C:100, kernel:rbf	Yes
LinearSVC	C:10	Yes
K Nearest Neighbors	Num. Neighbors:30	No
Random Forest Classifier	Num Estimators:500	Yes
Extra Trees Classifier	Num Estimators:1100	Yes

Table 6: Models implemented for comparison using envelope and embedded features.

† Tuned parameters values are based on grid-searching for best f-score. Parameters not mentioned have neither been tuned nor changed from scikit-learn's default values (as of version 0.23)

‡ Class weights are used unless not applicable to classifier.

Model Selection Using our encoding and envelope features to represent audio, five classification models were trained for the task of categorizing the five different sound groups. For hyper-parameter optimization, the models were trained using 5-fold cross validation and 80/20 train-to-test ratio. The F-Score result of each cross-validation is the unweighted average F-Score of all groups. For inter-model comparisons, the procedure is the same except 10-fold cross validations are used. Our models were derived from scikit-learn's implementations of these classifiers (Pedregosa et al., 2011). Before inter-model comparisons, we conducted a grid-search for each model on at least one of its possibly decisive hyper-parameters. The classifiers and other notable specifications are presented in Table 6.

DB Name	kick	snare	clap	hat	Synthetic Noise
MixedEarDB	1334	1035	401	1275	1000

Table 7: A database put together by combination of RadarDB, FreeDB and NoiseDB

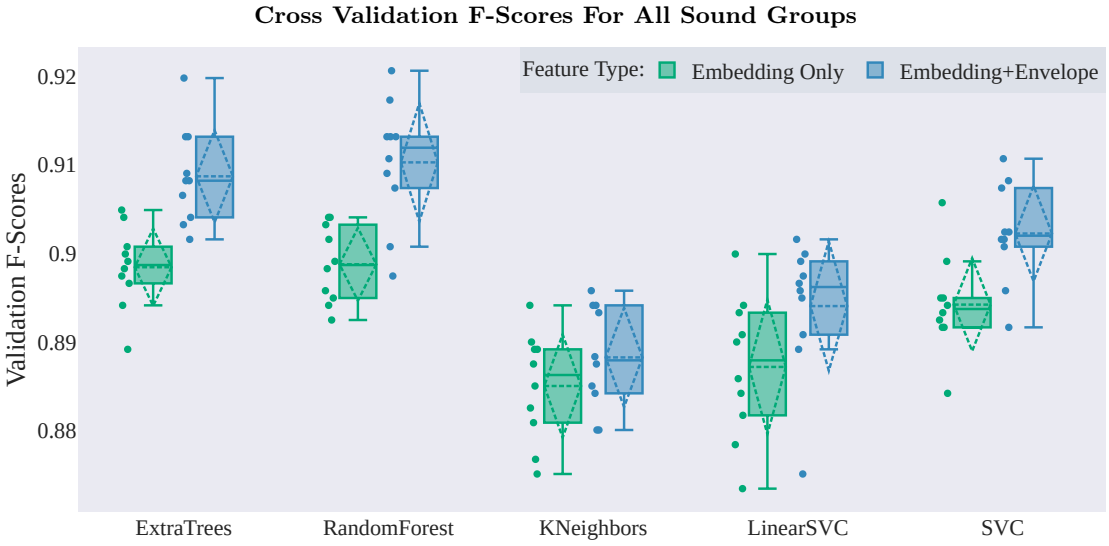


Fig. 6: Boxplots visualizing the F-Score results for each cross-validation. The individual scores, means, medians, standard-deviation and outliers are depicted. Envelope features improve classification accuracy.

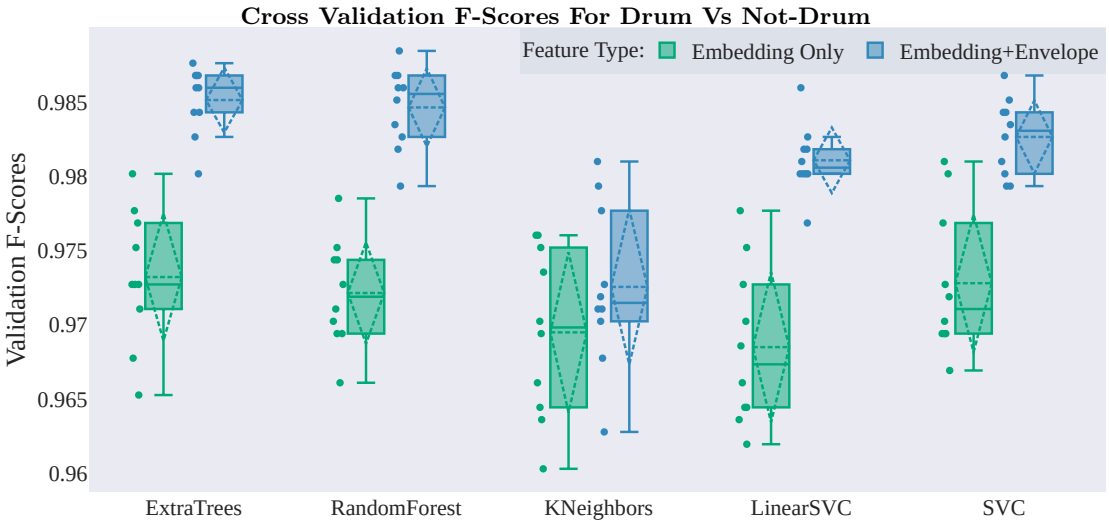
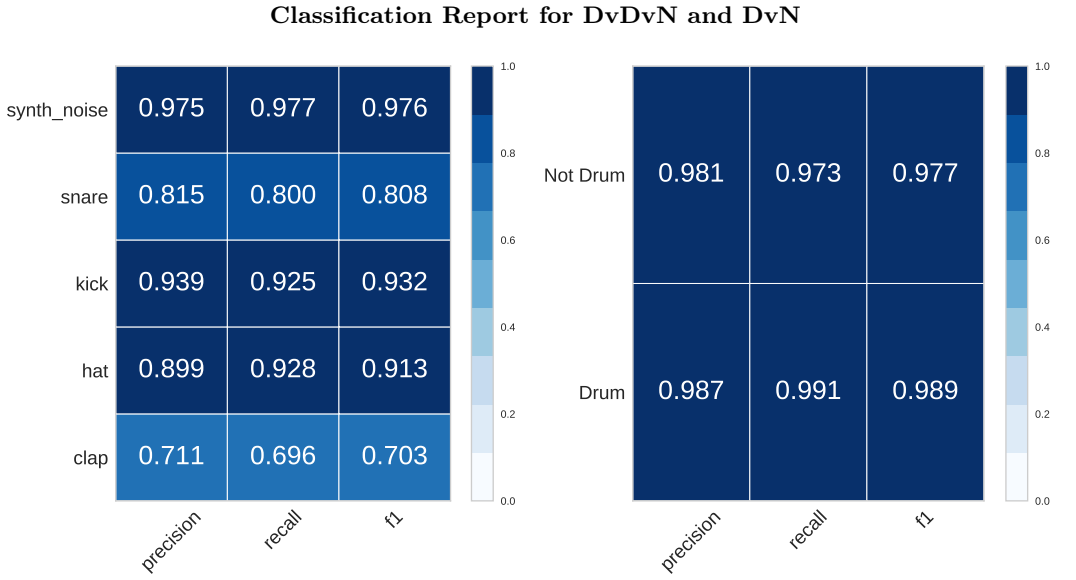
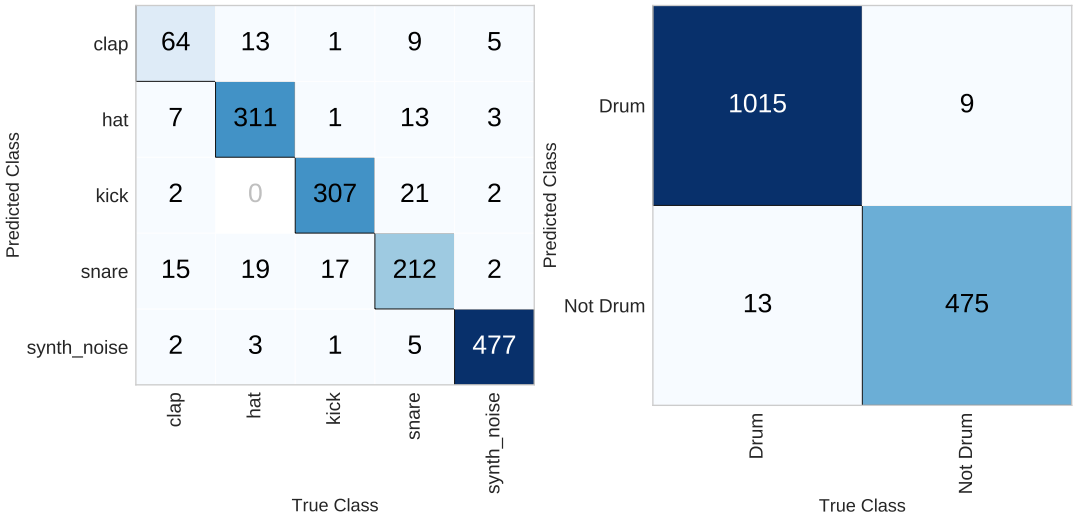


Fig. 7: F-Score results for each cross-validation. Models perform better as there are less categorization groups.



(a) Precision, recall, F1-Score, and number of supporting examples



(b) Confusion matrices

Fig. 8: F-Scores and confusion matrix of ExtraTrees model for both DvDvN and DvN.

D AutoEncoder Structures and Hyper-Parameter Optimization

Within the context of machine learning, a model's *hyper-parameters* are fixed parameters which are set before the training begins (e.g number of layers, size of layers, loss function) and are not learned during the minimization procedure (Bengio, 2000). To assist us with the construction of our model, we defined a number of possible choices for the architecture of our model and audio-transformers and used hyper-parameter optimization to extract promising sets of values 9.

The list of possible choices for the selected hyper-parameters can be found in table 9. We included not only model parameters but also spectrogram transformation parameters within this search space, as GPU accelerated FFT calculations allows ad hoc audio transformations to take place parallel to the training process. We implemented 3 base models which are affected by these hyper-parameters. The "Model Type" parameter dictates whether CNN or fully connected models are selected; If a "fully connected" model is selected, the "hidden layers" parameter selects between the two implementations. The specifications for these models can be found in tables 11, 12 and 10.

Using the optuna optimization tools (Akiba, Sano, Yanase, Ohta, & Koyama, 2019), we conducted 500 search trials. The trial's success is measured in their final loss value, calculated by applying the model to the test data-set. Each trial consisted of 20 epochs of training.

Hyper-Param.	Value
Model Type	CNN
Optimizer	Adam
Hidden Layers	Not Applicable
Learning Rate	0.001145
Frequency Bins	30
Time Steps	20
Latent Size	64
Regularization	3.25^{-6}
Dropout Rate	0.5

Table 8: Top performing hyper-parameter set

Hyper-Param.	Description	Values	Distribution
Model Type	Affects encoder's first hidden layer	CNN,FC	Categorical
Optimizer	Updates network's weights based on loss	Adam,SGD	Categorical
Hidden Layers	Extra hidden layer for the Encoder	True,False	Categorical
Time Steps	Temporal granularity of the spectrogram. Affects FFT windowing.	10,20	Categorical
Learning Rate	Optimizer's learning rate	1^{-4} ... 1^{-1}	Uniform
Frequency Bins	Number of spectrogram frequency bins	10, 30,60	Categorical
Regularization	L2 regularization parameter. Penalizes large weights to prevent overfitting	1^{-6} ... 1^{-1}	Uniform
Latent Size	Size of bottle neck layer or number encoded features	8,16,64	Categorical
Dropout Rate	Random zeroing of activations between layers to prevent over-fitting	0,0.5,0.1	Categorical

Table 9: The Hyper-Paramter space in which the optimization was conducted.

Layer-#	Out Shape	Param Num	Details
Conv2d-1	[-1, 8, 30, 20]	208	Encoder's input Num. Channels:8 kernel:5x5 stride:1 padding:2
ReLU-2	[-1, 8, 30, 20]	0	
MaxPool2d-3	[-1, 8, 15, 10]	0	kernel:5x5 stride:2
Dropout-4	[-1, 8, 15, 10]	0	
Linear-5	[-1, 8]	9,608	Encoder's output
Linear-6	[-1, 256]	2,304	Decoder's Input
Dropout-7	[-1, 256]	0	
Linear-8	[-1, 600]	154,200	Decoder's output

Table 10: CNN model design with latent size of 8. 30 and 20 are the assumed frequency bins and step size. Total number of parameters is 166,320.

Layer-#	Out Shape	Param Num	Details
Linear-1	[-1, 128]	76,928	Encoder's input
Dropout-2	[-1, 128]	0	
Linear-3	[-1, 8]	9,608	Encoder's output
Linear-4	[-1, 128]	2,304	Decoder's Input
Dropout-5	[-1, 128]	0	
Linear-6	[-1, 600]	77,400	Decoder's output

Table 11: Fully connected model with only 1 hidden dimension for encoder and decoder. Designed assumes latent size of 8. 30 and 20 are the assumed frequency-bins and step-size values. Total number of parameters is 156,512.

Layer-#	Out Shape	Param Num	Details
Linear-1	[-1, 128]	76,928	Encoder's input
Dropout-2	[-1, 128]	0	
Linear-3	[-1, 32]	4,128	
Dropout-4	[-1, 128]	0	
Linear-5	[-1, 8]	9,608	Encoder's output
Linear-4	[-1, 32]	2,304	Decoder's Input
Dropout-5	[-1, 32]	0	
Linear-4	[-1, 128]	2,304	
Dropout-5	[-1, 128]	0	
Linear-6	[-1, 600]	77,400	Decoder's output

Table 12: Fully connected model with 2 hidden dimensions for encoder and decoder. Designed assumes latent size of 8. 30 and 20 are the assumed frequency-bins and step-size values. Total number of parameters is 163,232.

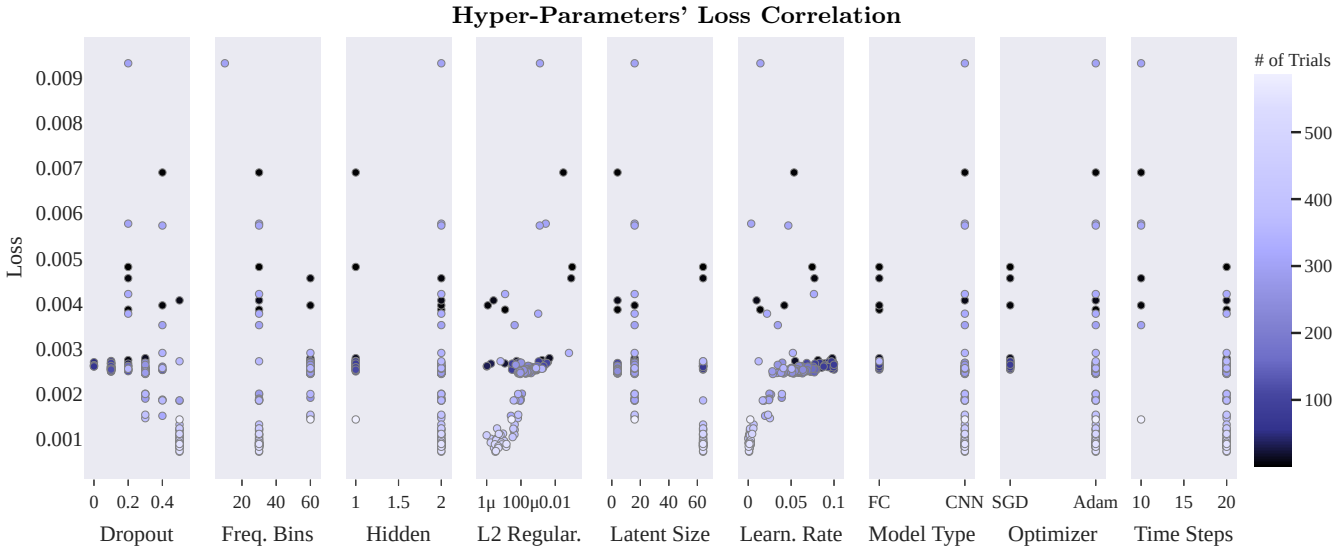


Fig. 9: Sliced plot depicting the correlation between hyper-parameters and loss values. The color-scale shows the number of times each parameters has been used in a trial. Our sampling algorithm aims to utilize spaces with higher potential more often.

E Survey Details

Details of survey responses are presented in figures 10 and 11. The surveys were done blinded with the two authors assigning categories to unlabled sounds.

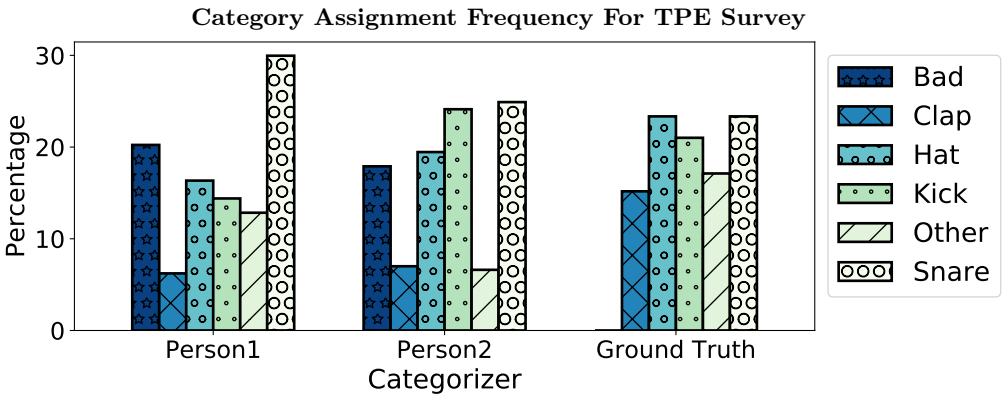


Fig. 10: Frequency of assigned labels by persons vs the true number of labels

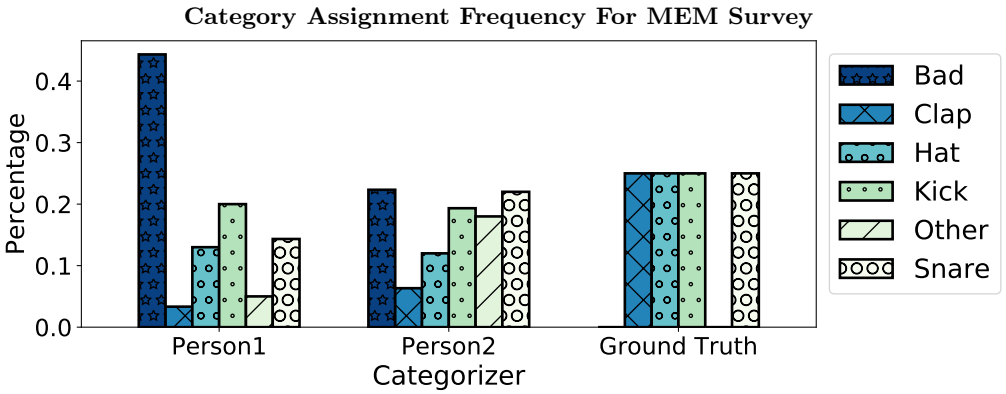


Fig. 11: Frequency of assigned labels by persons vs the true number of labels