Technische Universität Berlin

Fakultät I - Geisteswissenschaften
Fachgebiet Audiokommunikation
Audiokommunikation und -technologie M.Sc.

Self-Organizing Maps for Sound Corpus Organization

MASTER'S THESIS

Vorgelegt von: Jonas Margraf
Matrikelnummer: 372625
E-Mail: jonasmargraf@me.com

Erstgutachter: Prof. Dr. Stefan Weinzierl
Zweitgutachter: Dr. Diemo Schwarz
Datum: March 12, 2019

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.
Berlin, den March 12, 2019

. . . .
Jonas Margraf

Abstract An english abstract.

Zusammenfassung Die Zusammenfassung auch auf Deutsch.

Acknowledgements

This is where the thank yous go.

Contents

1 Introduction

This is the Introduction. Here's a citation about Self-Organizing Maps (SOMs)(Kohonen, 1990).

1.1 Motivation and Problem Description

1.2 Aims and Objectives

1.3 Previous Work

2 Background

The following chapter intends to provide a theoretical background for two key concepts underlying the work presented in this thesis, namely Audio Feature Extraction and the Self-Organizing Map.

2.1 Audio Feature Extraction

Audio Feature Extraction is the process of deriving *features* from a digital audio signal. A feature represents some sort of descriptive information about the audio data. According to Lerch (2012), this extraction process serves a dual purpose; that of dimensionality reduction as well as a more meaningful representation. A large variety of features for different purposes have been developed (refer to Peeters (2004) for an extensive list as well as Lerch (2012) for an in-depth look at the topic). The following subsections introduce the features used in this work, starting with the pre-processing required to prepare an audio signal for feature extraction and then moving into individual definitions for each feature. The equations presented here are based on the formal definitions given by Lerch (2012) along with the computational implementations of the features in the *Meyda* library for feature extraction in JavaScript (Rawlinson et al., 2015), which in turn adapted the *yaafe* library for Python (Mathieu et al., 2010).

2.1.1 Audio Pre-Processing

Consider a digital audio signal of the form $x[n]$, where n denotes the sample index and $x[n]$ the value of the individual sample at that index.

2.1.1.1 Normalization In order to have a standardized maximum amplitude of 1 across all audio signals, they are normalized such that

$$x_{norm}[n] = \frac{x[n]}{\max x[n]}. \tag{1}$$

2.1.1.2 Mono Conversion Spatial information, as contained in an audio file with more than one channel, is not deemed necessary in the presented work. For this reason, all audio signals are converted to mono by taking the average of all channels.

2.1.1.3 Frame-Based Feature Extraction Rather than performing feature extraction on the entirety of the audio signal, it is common practice to divide the signal into smaller chunks or *frames*, typically consisting of some 2^n samples (512, 1024, 2048 are often found values). The resulting feature values for each frame form a trajectory of the feature's evolution over time, which can either be used as such or can be averaged. In this work, audio signals are divided into frames with a length of 512 samples. In order to avoid computational errors (such as Not a Number (NaN) in JavaScript) during potentially silent portions of the audio signal, frames with $v_{RMS} < -60\,\text{dBFS}$ (see Root Mean Square (RMS) definition below) are omitted from the feature extraction. The following equations define each feature for a single frame.

2.1.2 Time Domain Features

Time domain features are features derived directly from the discrete-time signal $x[n]$.

2.1.2.1 Duration The overall duration of the signal $x[n]$ in seconds:

$$v_{DUR} = \frac{n}{fs} s, \tag{2}$$

where n is the number of samples and fs is the sampling rate.

2.1.2.2 Root Mean Square (RMS) measures the power of a signal (Lerch, 2012, p.73f). It describes sound intensity and is sometimes used as a simple measure for loudness (Rawlinson et al., 2019a) that does not take the nonlinearity of human hearing into account (Fletcher and Munson, 1933). It is calculated for an audio frame $x[n]$ consisting of n samples such that

$$v_{RMS} = \sqrt{\frac{\sum_{i=1}^{n} x(i)^2}{n}}. \tag{3}$$

2.1.2.3 Zero-Crossing Rate (ZCR) represents the rate of the number of sign changes in a signal. It can be used as a measure of the tonalness of a sound (Lykartsis, 2014) and as a simple pitch detection method for monophonic signals (de la Cuadra, 2019). It is defined as

$$v_{ZCR} = \frac{1}{2 \cdot n} \sum_{i=1}^{n} |sgn[x(i)] - sgn[x(i-1)]|. \tag{4}$$

2.1.3 Frequency Domain Features

Frequency domain features or *spectral* features are derived from the discrete complex spectrum $X(k)$, where k refers to the frequency bin number. $X(k)$ is calculated from $x[n]$ by performing an Fast Fourier Transform (FFT) (for information on the Fourier transform, refer to any signal processing textbook, such as Oppenheim and Schafer (2014)).

2.1.3.1 Spectral Centroid is a measure of the center of gravity of a spectrum. A higher value indicates a brighter, sharper sound (Lerch, 2012). The spectral centroid is defined as

$$v_{SC} = \frac{\sum\limits_{k=0}^{N_{FFT}/2-1} k \cdot |X(k)|^2}{\sum\limits_{k=0}^{N_{FFT}/2-1} |X(k)|^2}. \tag{5}$$

2.1.3.2 Spectral Flatness is a measure for the tonality or noisiness of a signal, defined as the ratio of the geometric and arithmetic means of its magnitude spectrum. Higher values indicate a flatter (and therefore noisier) spectrum, whereas lower values point towards more tonal spectral content. It is defined as

$$v_{SFL} = \frac{\sqrt[N_{FFT}/2]{\prod\limits_{k=0}^{N_{FFT}/2-1} |X(k)|}}{(2/N_{FFT}) \cdot \sum\limits_{k=0}^{N_{FFT}/2-1} |X(k)|}. \tag{6}$$

2.1.3.3 Spectral Kurtosis indicates whether a given magnitude spectrum's distribution is similar to a Gaussian distribution. Negative values result from a flatter distribution, whereas positive values indicate a peakier distribution. A Gaussian distribution would result in a value of 0. Spectral Kurtosis is defined as

$$v_{SKU} = \frac{2 \sum\limits_{k=0}^{N_{FFT}/2-1} (|X(k)| - \mu_{|X|})^4}{N_{FFT} \cdot \sigma_{|X|}^4} - 3, \tag{7}$$

where $\mu_{|X|}$ represents the mean and $\sigma_{|X|}$ the standard deviation of the magnitude spectrum $|X|$.

2.1.3.4 Spectral Skewness assesses the symmetry of a magnitude spectrum distribution. It is defined as

$$v_{SSK} = \frac{2 \sum\limits_{k=0}^{N_{FFT}/2-1} (|X(k)| - \mu_{|X|})^3}{N_{FFT} \cdot \sigma_{|X|}^3}. \tag{8}$$

2.1.3.5 Spectral Slope represents a measure of how sloped or inclined a given spectral distribution is. The spectral slope is calculated using a linear regression of the magnitude spectrum such that

$$v_{SSL} = \frac{\sum\limits_{k=0}^{N_{FFT}/2-1} (k - \mu_k)(|X(k)| - \mu_{|X|})}{\sum\limits_{k=0}^{N_{FFT}/2-1} (k - \mu_k)^2}. \tag{9}$$

2.1.3.6 Spectral Spread is a descriptor of the concentration of a magnitude spectrum around the Spectral Centroid and assesses the corresponding signal's bandwidth. It is defined as

$$v_{SSP} = \frac{\sum\limits_{k=0}^{N_{FFT}/2-1} (k - v_{SC})^2 \cdot |X(k)|^2}{\sum\limits_{k=0}^{N_{FFT}/2-1} |X(k)|^2}. \tag{10}$$

2.1.3.7 Spectral Rolloff measures the bandwidth of a given signal by calculating that frequency bin below which lie κ percent of the sum of magnitudes of $X(k)$. Common values for κ are 0.85, 0.95 (Lerch, 2012) or 0.99 (Rawlinson et al., 2019a). It is defined as

$$v_{SR} = i\Big|_{\sum\limits_{k=0}^{i} |X(k)| = \kappa \cdot \sum\limits_{k=0}^{N_{FFT}/2-1} |X(k)|}. \tag{11}$$

2.1.4 Perceptual Features

Both the time and frequency domain features introduced above are derived from raw audio samples without taking into account any concept of human sound perception. Perceptual features incorporate some sort of model that approximates this perception. While only a single perceptual feature is used in this work, more do exist (see Peeters (2004) for a list of some of them).

2.1.4.1 Total Loudness represents an algorithmic approximation of the human perception of a signal's loudness based on Moore et al. (1997), which uses the Bark scale as introduced by Zwicker (1961). The Total Loudness is the sum of all 24 bands' specific loudness coefficients, defined by Peeters (2004) as

$$v_{TL} = \sum_{i=1}^{24} v_{SL}(i), \tag{12}$$

where

$$v_{SL}(i) = E(i)^{0.23} \tag{13}$$

is the specific loudness of each Bark band (see Moore et al. (1997) for further details).

2.2 Self-Organizing Map

The *self-organizing map* (SOM) is a machine learning algorithm for dimensionality reduction, visualization and analysis of higher-dimensional data. Sometimes also referred to as *Kohonen map* or *network*, it was introduced in 1981 by Teuvo Kohonen (Kohonen, 1990).
The SOM is a variant of an *artificial neural network* that uses an unsupervised, competitive learning process to map a set of higher-dimensional observations (the *input vectors*) onto a regular, often two-dimensional grid or *map* of *neurons* or *nodes* that is easy to visualize. The SOM can be regarded as a nonlinear generalization of a principal component analysis (PCA) (Yin, 2007) or as a quantization of the input data, with the nodes along the map functioning as pointers into that higher-dimensional space. Each node has a position on the lower-dimensional grid as well as an associated position in the input space, which takes the form of a n-dimensional weight vector $m = [m_1, ..., m_n]$, where n is the number of dimensions of the input vectors. Nodes that are in close proximity to each other on the SOM will also have similar weight vectors (Vesanto et al., 2000), although the inverse (neighboring positions in the input space also mapping to neighboring nodes) is not necessarily true (Bauer et al., 1996).
For an in-depth look at the algorithm, its variants and applications, as well as an extensive survey of research on SOMs, the avid reader is referred to Kohonen (2001).

2.2.1 Algorithm Definition

The following definition is based on Kohonen (1990), Kohonen (2005), Kohonen and Honkela (2007) and Bauer et al. (1996).

Consider a space of input data in the form of n-dimensional vectors $x \in \mathbb{R}^n$ and an ordered set of nodes or model vectors $m_i \in \mathbb{R}^n$. A vector $x(t)$ is mapped to that node m_c with the shortest Euclidean distance from it:

$$||x(t) - m_c|| \leq ||x(t) - m_i|| \; \forall i. \tag{14}$$

This "winning" node m_c is referred to as the Best Matching Unit (BMU) for $x(t)$.

During the learning or adaptation phase of the algorithm, all nodes m_i are adjusted by a recursive regression process

$$m_i(t+1) = m_i(t) + h_{c(x),i}(x(t) - m_i(t)), \tag{15}$$

where t is the index of the current regression step, $x(t)$ is an input vector chosen randomly from the input data at this step, c is the index of the BMU for the current input vector $x(t)$ according to equation 14 and $h_{c(x),i}$ represents a so-called *neighborhood function.* The name-giving *neighborhood* is a subset N_c of nodes centered on m_c. At each learning step t, those nodes that are within N_c will be adjusted, whereas those outside of it will not. The reason for employing such a neighborhood function is so that the nodes "doing the learning are not affected independently of each other" (Kohonen, 1990, p.1467) and "the topography of the map is ensured" (Bauer et al., 1996, p.5). At its most basic, the neighborhood function is a decreasing distance function between neurons m_i and m_c. Its most common form, which is also employed in this work, is that of a Gaussian function with its peak at m_c such that

$$h_{c(x),i} = \alpha(t) \exp\left(- \frac{||r_i - r_c||^2}{2\sigma^2(t)} \right). \tag{16}$$

Here, α denotes a learning rate factor or adaptation "gain control" $0 < \alpha(t) < 1$, which decreases over the course of the regression, $r_i \in \mathbb{R}^2$ and $r_c \in \mathbb{R}^2$ are the locations of m_i and m_c on the SOM grid (the lower-dimensional output map, not the input space!), and $\sigma(t)$ is the width of the neighborhood function, which again decreases as the regression step index increases.

2.2.2 Node Initialization

Because of the iterative nature of the SOM algorithm, its outcome depends on the initial positions chosen for the nodes. The method implemented in this work uses random initialization, meaning the starting positions of the nodes are chosen randomly from within the bounds of the input space. An often employed alternative approach is to first perform a Principal Component Analysis (PCA) on the input data, select the largest d components, where d is the number of desired output dimensions for the SOM, and then distribute the nodes at equidistant intervals along those component vectors.

2.2.3 Input Data Scaling

Some consideration should be given to the dynamic range of the input data across its different dimensions. Are the dimensional ranges comparable in their limits? What about their variance? There does not appear to exist a clear consensus across the literature on whether or not normalization of input data is strictly necessary (Vesanto et al. (2000, p.34), Kohonen (1990, p.1470), Kohonen and Honkela (2007)).

Because the range of the data derived from the audio feature analysis used in this work varies considerably between features, the data for feature n is rescaled to have unit variance by dividing by the features' standard deviation σ_n:

$$x_n = \frac{x_n}{\sigma_n}. \tag{17}$$

2.2.4 Alternative Learning Rate Factors

2.2.4.1 Linear The traditional SOM algorithm uses a learning rate factor α that decreases linearly as a function of the regression step t:

$$\alpha(t) = \alpha_0 \left(\frac{1-t}{T} \right), \tag{18}$$

where α_0 is the initially chosen learning rate and T is the total training length or number of regression steps.

Two other approaches to the decreasing learning rate factor were implemented in this work:

2.2.4.2 Inverse The first is a reciprocally decreasing function where

$$\alpha(t) = \frac{\alpha_0}{\left(\frac{1+100t}{T} \right)}. \tag{19}$$

2.2.4.3 BDH The second alternative approach is that of an adaptive local learning rate as developed by Bauer et al. (1996) (BDH algorithm, also see Merenyi et al. (2007)):

$$\alpha(t) = \alpha_0 \left(\frac{1}{\Delta t_c} \left(\frac{1}{|x(t) - m_c|^n} \right) \right)^m, \tag{20}$$

where Δt_c represents the time since the current BMU for the current input vector was last selected as a BMU for any vector and m is a newly introduced, free control parameter. For a more complete review of the uses of this algorithm, the reader is referred to the original paper (Bauer et al., 1996) as well as Merenyi et al. (2007).

3 Implementation

After some theoretical background information was given in the previous chapter, the following sections aim to explain how the SOM algorithm was implemented in JavaScript. First, a smaller program was built to extend the existing software *CataRT*. Then a second, more fully fledged application called *SOM Browser* was developed. For both of these programs, we take a look at their functionality and features, give an overview of the code and program structure, and explain some concepts and considerations that were important for the development process.

3.1 Groundwork: CataRT Extension

For the purpose of laying the groundwork for a bigger standalone application (see section 3.2), a proof-of-concept implementation of the core SOM algorithm was written in JavaScript to serve as an extension to the *MuBu For Max* software package (Schnell et al. (2019a), Schnell et al. (2019b)) for the visual programming language Max (Cycling '74, 2019). *MuBu For Max* was developed by the Sound, Music, Movement, Interaction Team (ISMM) at Institut de recherche et coordination acoustique/musique (IRCAM) (Schnell et al., 2009). It contains the *catart-by-mubu* patch for realtime interactive corpus-based concatenative synthesis based on the original *CataRT* software (Schwarz et al., 2006). The developed extension is a Max patch called *mubu-SOM-js* (see Figure 1) and can be found on the digital resource included with this thesis in the directory XXX path to patch XXX.

Catart-by-mubu uses a two-dimensional scatter plot interface in which the user can select samples or grains from the loaded audio corpus (see Figure 2). The spatial position of these sounds in the interface is determined by two audio features, representing the horizontal and vertical axes, that can be selected by the user. The implemented SOM extension gives users the option to choose a two-dimensional SOM for the spatial organization of the corpus. This augments the interface in three ways: all analyzed audio features can be taken into account for the spatial positioning (as opposed to just two at a time), more of the available interface space is used and additionally the sounds are spaced in a more even fashion (see Figure 2).

3.1.1 Functionality

Mubu-SOM-js offers the user simple controls to influence the produced SOM. These can be set by sending the messages outlined in Table 1 to the `[js descriptor_som.js]` object.

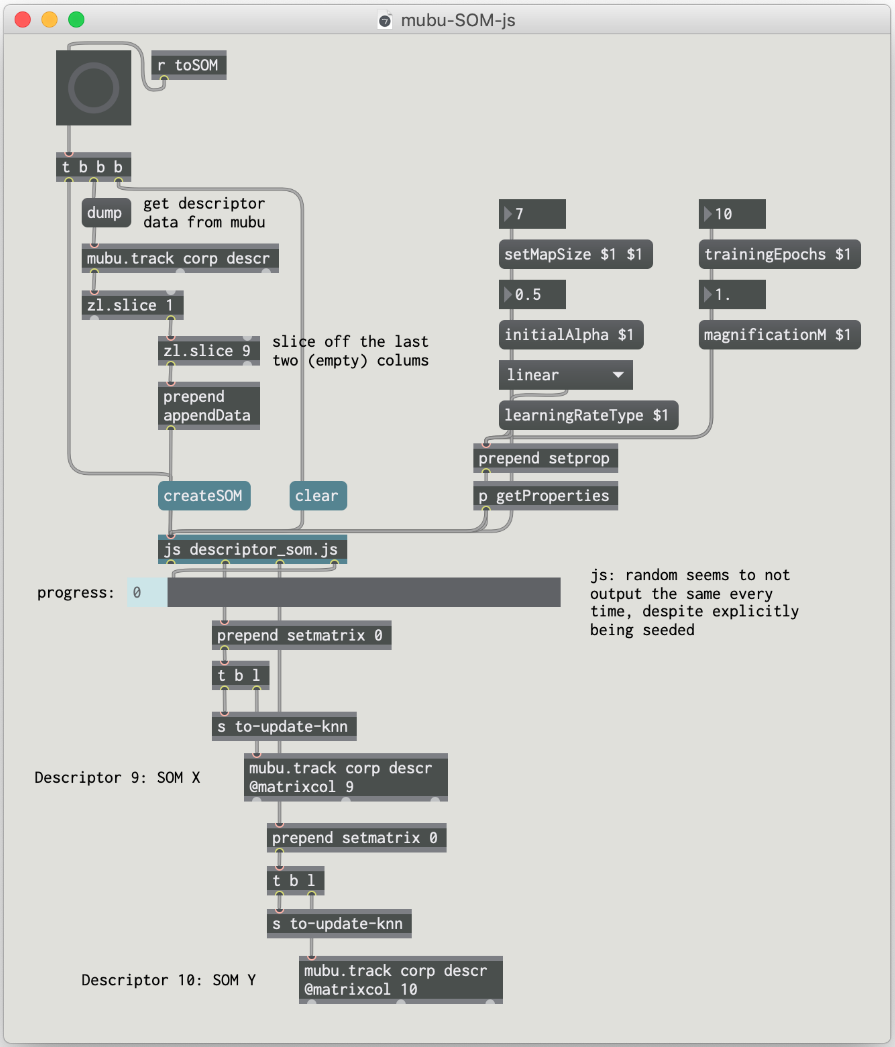
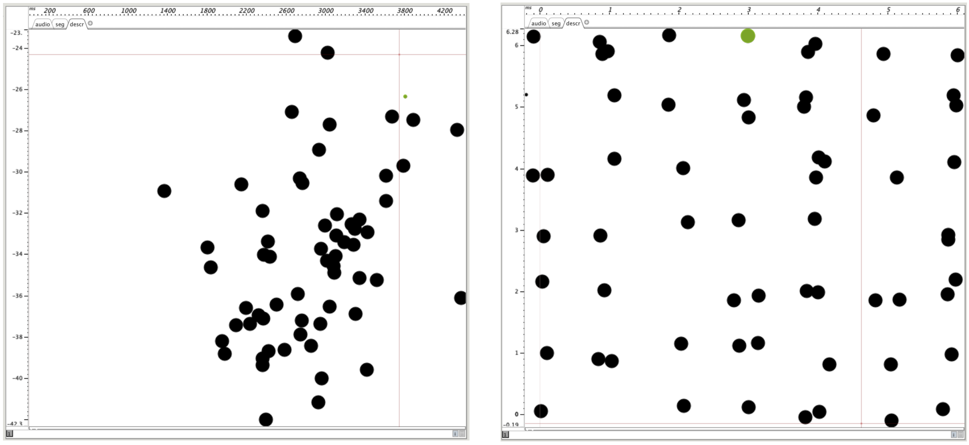


Fig. 1: mubu-SOM-js

3.1.2 Code Overview

The core of the *mubu-SOM-js* Max patch is a JavaScript program (see the file `mubu-som-js/descriptor_som.js`). The choice of programming language was determined by the fact that *CataRT* is a Max patch and JavaScript (via the built-in `[js]` object) can be used to script most aspects of the Max environment. This JavaScript version of the SOM is in some ways a port from



(a) (b)

Fig. 2: *CataRT* display of a corpus without SOM (2a, X axis shows spectral centroid, Y axis shows loudness) and with SOM extension (2b). Each circle represents a sample.

Message	**Type**	**Description**	**Example**
`createSOM`	n/a	Initiates SOM calculation.	`createSOM`
`setMapSize $1 $1`	Float	Sets size of map.	`setMapSize 7 7`
`trainingEpochs $1`	Int	Defines the length of the training in epochs. One epoch corresponds to n iterations of the training algorithm (see section 2.2.1), where n is the number of samples in the corpus.	`trainingEpochs 30`
`initialAlpha $1`	Float	Sets the starting value for the learning rate factor α.	`initialAlpha 0.5`
`learningRateType $1`	String	Sets the learning rate type (see section 2.2.4). It expects a string that is either `'linear'`, `'inverse'` or `'BDH'`.	`learningRateType ’linear’`
`magnificationM $1`	Float	Sets the magnification control factor m (see section 2.2.4.3). Only applies when `learningRateType === 'BDH'`.	`magnificationM 0.02`

Tab. 1: mubu-SOM-js: Messages for algorithm control

a first MATLAB implementation of the algorithm that was developed by the author during an internship at IRCAM in the fall of 2017. Some aspects of the structure of the presented program are based on the SOM Toolbox that was developed at Helsinki University of Technology by Vesanto et al. (2000).

The flow of the script is encapsulated in `createSOM()`. This function calls all other important functions that make up the program, as can be seen in Listing 1.

```
function createSOM()
{
  normalizeData();
  initializeMap();
  trainMap();
}
```

Listing 1: mubu-som-js/descriptor_som.js: `createSOM()`

After data normalization and map initialization, `trainMap()` is called, which executes the training procedure by repeatedly calling the function `training()` in an asynchronous background process (see Listing 2). For each step of the training phase, all calculations happen inside `trainingStep()`. The most important part, the updating of node positions on each iteration, is shown in Listing 3.

```
function training()
{
  if (t < trainingLength)
  {
    trainingStep(t, trainingLength, rStep, alpha, winTimeStamp);
    // Progress percentage on outlet 4
    outlet(3, math.ceil(100 * (t / trainingLength)));
    t++;
  }
  else
  {
    post('Training done.\n');
    findBestMatches();
    outputDataCoordinatesOnMap();
    arguments.callee.task.cancel();
  }
}
```

Listing 2: mubu-som-js/descriptor_som.js: `training()`

After the training phase is finished, the final map is populated by iterating over all vectors and finding their corresponding best matching units (meaning

```

    // For each neuron, get neighborhood function and update its position.
    neurons = neurons.map(function (neuron, index) {
      // Gaussian neighborhood function
      var h = alpha * math.exp(-(math.square(distances[index][bmu])
                               / (2 * math.square(r))));
      return math.subtract(neuron, math.multiply(h, differences[index]));
```

Listing 3: mubu-som-js/descriptor_som.js: neuron position updates inside `trainingStep()`

that node which is closest), as can be seen in Listing 4. In order to spatially differentiate between vectors that were assigned to the same node, a small amount of random noise is added to the position. This creates clusters around the exact node position and allows for the individual circles to be selected. An example of a map without added noise can be found in Figure 3.

```
function findBestMatches()
{
  bestMatches = normalizedData.map(function (vector) {
    var differences = [];
    var distancesFromVector = [];

    // Subtract chosen vector from each neuron / map unit, then calculate that
    // difference vector's magnitude.
    // In other words, calculate the Euclidean distance between each neuron and
    // the chosen vector.
    for (var n = 0; n < neuronCount; n++)
    {
      distancesFromVector.push(math.norm(math.subtract(neurons[n], vector)));
    }
    // Find best matching unit's distance and index:
    var bmuDistance = math.min(distancesFromVector);
    var bmu = distancesFromVector.indexOf(bmuDistance);
    return [bmu, bmuDistance];
  });
}
```

Listing 4: mubu-som-js/descriptor_som.js: `findBestMatches()`

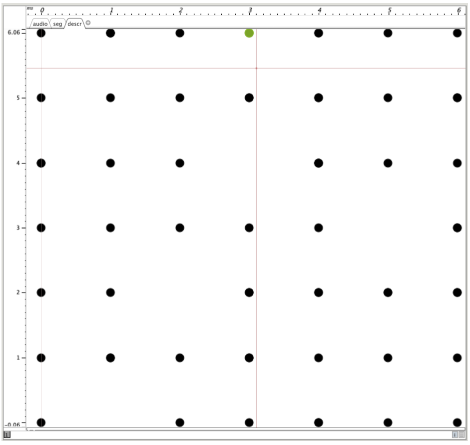


Fig. 3: *CataRT* display of a corpus with SOM extension, but without added noise to differentiate samples assigned to the same node. Each circle represents a sample.

3.2 SOM Browser

The majority of the work for this thesis consisted of the development of a standalone application for sample library exploration which we call *SOM Browser*. A screenshot of the program can be seen in Figure 4.

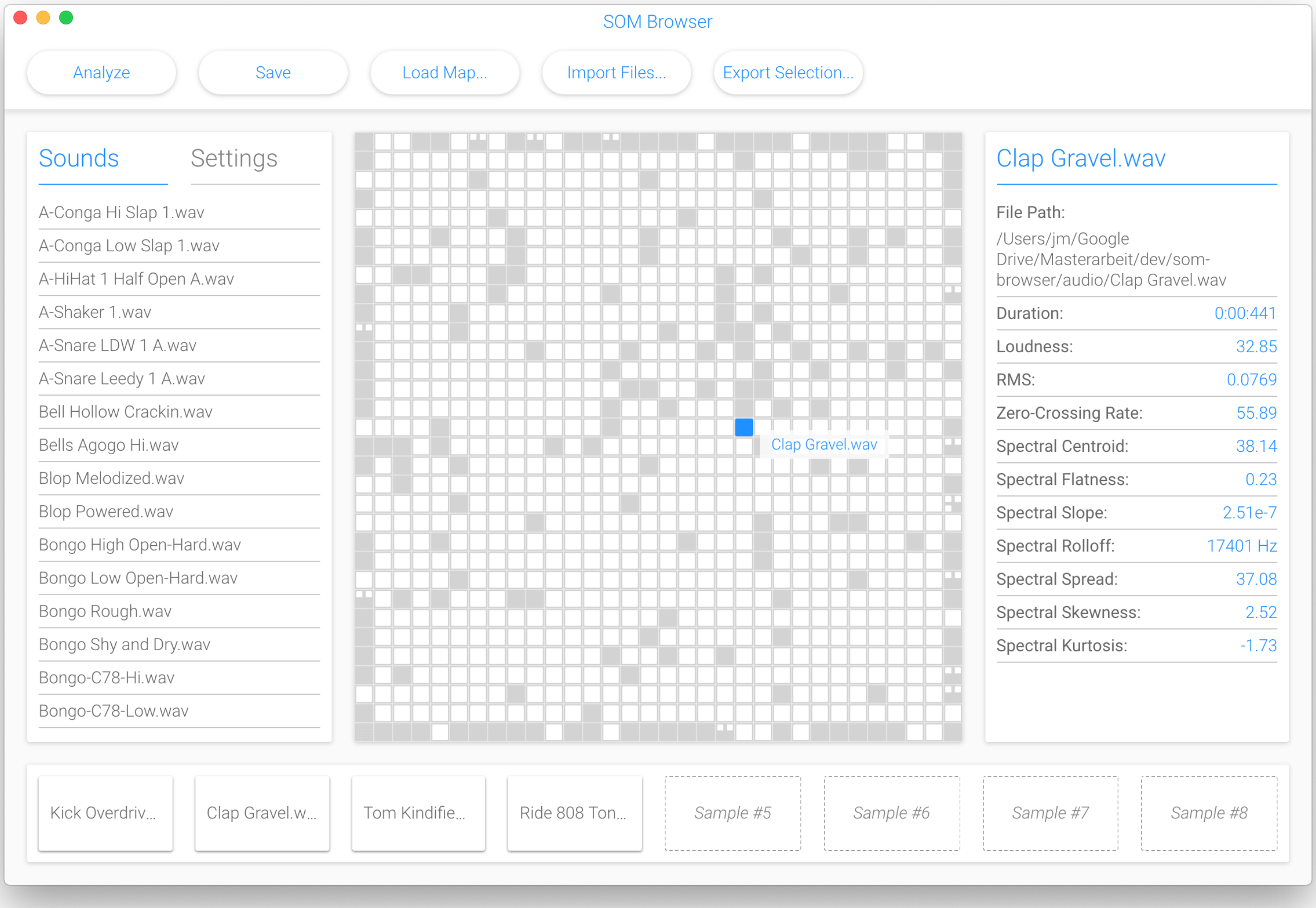


Fig. 4: *SOM Browser*

SOM Browser offers users an alternative interface for the interaction with a folder of audio samples. Instead of the traditional file browser interface consisting of an alphabetical list of file names, the presented application offers a spatial map layout of the samples, with the aim of allowing users a more direct interaction and giving them a quicker overview of the sounds.

3.2.1 Functionality

3.2.1.1 Loading Audio Files When launching *SOM Browser*, the application opens with no sounds or map loaded (see Figure 5). In order to create a map of a collection of sound files, the user can go to the menu bar at the top of the window and click the *"Import Files..."* button to load several audio files.

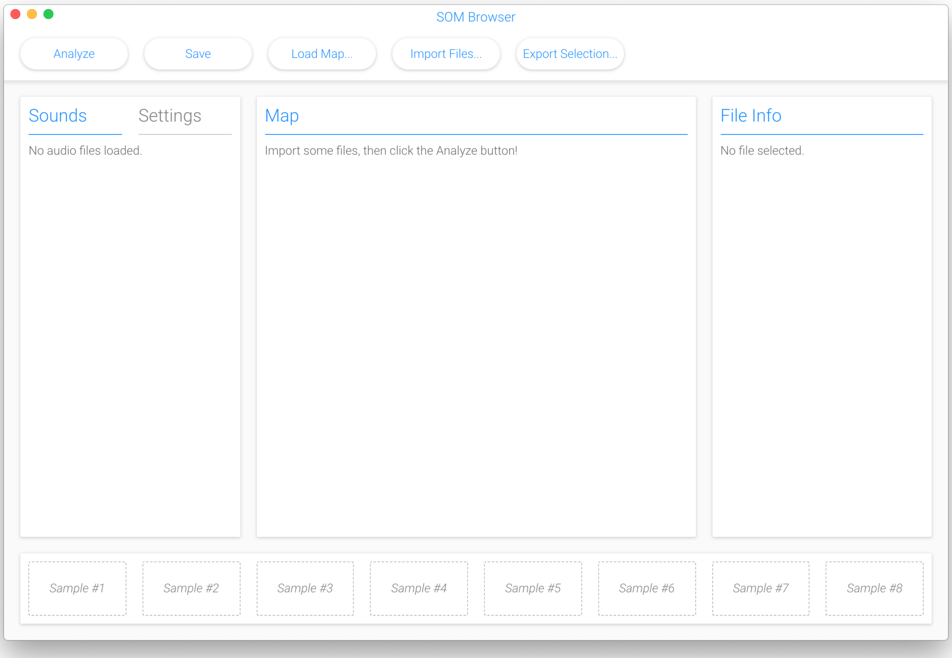


Fig. 5: *SOM Browser* without audio files loaded

3.2.1.2 Calculating a Map Once files are selected, the *Sounds* list on the left side of the application will be populated. Next, by clicking *"Analyze"*, the program will start to analyze the audio files in the background, first extracting audio features (see section 2.1) and then using this information to calculate a SOM using default settings. Alternatively, some SOM parameters can be altered by selecting the *Settings* field next to *Sounds* and adjusting the exposed parameters. Depending on the number of audio files to analyze and the selected training duration, the algorithm will take a while to process. Training progress is indicated as a percentage in the central *Map* panel.

3.2.1.3 Map Interaction Upon completion of the SOM calculation, the *Map* panel will be populated by a grid of white and grey squares. Each white square represents a single sound file. All files are loaded into the computer's Random-Access Memory (RAM) for quick access. Grey squares are empty nodes, meaning nodes to which no sound files were assigned. Sounds can be played by clicking on the white squares. They can also be played immediately by holding down the Shift key and hovering over them. This allows the user a very fast audition process and makes it possible to play back many files in fast succession, enabling very quick browsing of all loaded audio files. When hovering over a square, the corresponding file name is shown next to the mouse cursor. More detailed information about the file, including its full path, duration and audio feature values can be found in the *FileInfo* panel to the right of the map.

3.2.1.4 Selecting and Exporting Favorites The bottom of the window is taken up by the *Favorites* bar. If a sample is found on the map that the user would like to save for further usage, they can drag the square from the map down into one of the slots labeled *"Sample #1 - #8"*. Samples can also be also be played from the *Favorites* bar by clicking on them. If the user is satisfied with their selection of samples, they can export the selected *Favorites* (e.g. for further usage in a Digital Audio Workstation (DAW)) by clicking on *"Export Selection"* in the top menu bar. This will open a file dialog window to select a location where the files should be stored.

3.2.1.5 Saving and Loading Maps *SOM Browser* also offers the ability to save entire maps to disk for recall in a later session or import previously stored maps by clicking on the menu bar buttons *"Save"* and *"Load Map"*.

3.2.2 Libraries and Frameworks Used

Although a desktop application, *SOM Browser* was built entirely using web technologies, most importantly JavaScript. A vast variety of libraries and frameworks are available to use for all aspects of the development process. The following paragraphs outline the tools chosen for this application and their benefits.

3.2.2.1 Electron "is an open source library developed by GitHub for building cross-platform desktop applications with HTML, CSS, and JavaScript. Electron accomplishes this by combining Chromium and Node.js into a single runtime and apps can be packaged for Mac, Windows, and Linux" (GitHub, 2019). It offers a variety of Application Programming Interfaces (APIs) to offer native menus, interact with the file system and more. Its `ipcMain` and `ipcRenderer` APIs are used for asynchronous communication between the Graphical User Interface (GUI) and processes running in the background.

3.2.2.2 React is a JavaScript library for building user interfaces (Facebook, 2019). It breaks the GUI into smaller, self-contained units called *components* that can be independently updated and rendered.

3.2.2.3 Web Audio API enables audio processing and synthesis in (web) applications (World Wide Web Consortium (W3C), 2019). The use of this API makes it possible to write all audio processing code for the presented work in JavaScript. Its core concept is the *audio routing graph*, made up of *audio nodes* (simple building blocks such as an oscillator or a recording). This graph connects sources to other other nodes (e.g. effects or filters) and finally to an output destination.

3.2.2.4 Meyda "is a Javascript audio feature extraction library. Meyda supports both offline feature extraction as well as real-time feature extraction using the Web Audio API" (Rawlinson et al., 2019b). Its effectiveness has been validated by researchers at Queen Mary University ("Meyda [...] provide[s] excellent real time feature extraction tools", Moffat et al. (2015)).

3.2.3 Application Structure

SOM Browser is a *stateful* application, meaning it is designed to remember user interactions, save its internal data (the *state* of the application) between interaction steps and to allow the storing of state data between sessions.

3.2.3.1 System States Before the start of the development process, a set of system states was designed to represent the states through which the application is supposed to progress. These states and their order are shown in Figure 6, giving an abstract overview of the flow of the program. Each panel represents a state and consists of a title (shown in capitalized words at the top, e.g. *Map Created*), a method describing a state transition (underscored and in lower case, e.g. *show map*) and the next state to transition into (bottom right, marked by an arrow, e.g. $\rightarrow$ *File Audition*).

3.2.3.2 Code Overview *SOM Browser* was developed using the *git* version control system (Torvalds and Hamano, 2019) in a repository on GitHub

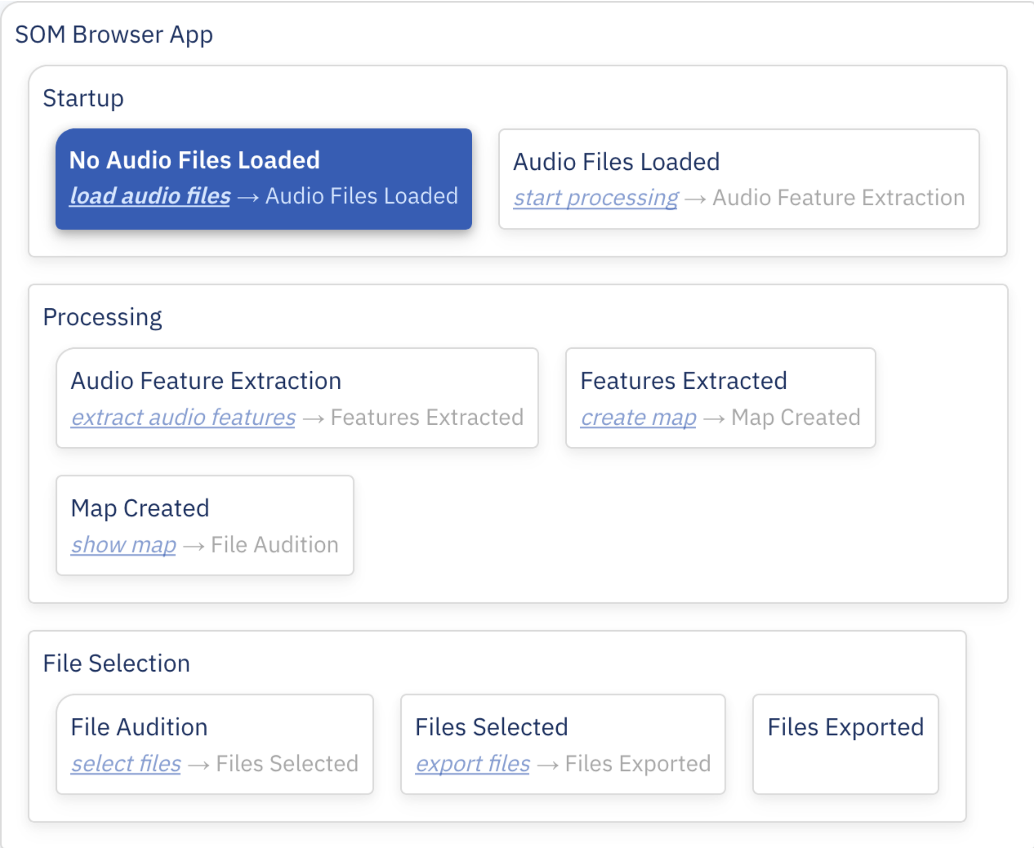


Fig. 6: *SOM Browser*: Mock-up outlining system states

[1]. The very basic structure of the application, in particular the way in which the Electron and React frameworks interact, is based on a boilerplate project by Phillip Barbiero (Barbiero, 2017).

The entry point of any Electron application is the `main.js` file, which in the presented work can be found in `som-browser/src/main.js`. This file creates an instance of the `BrowserWindow` class called `mainWindow` that serves as the single visible application window. `mainWindow` then loads `som-browser/src/index.js`, which imports the React library and uses the React function `render()` to create the `<App />` component, which is defined in `som-browser/src/components/App.js`. It serves as a container for the rest of the application logic and the entire GUI (see Listing 7 and Section 3.2.5 for more details). From here, the structure of the source files branches out into the individual GUI elements in `som-browser/src/components/` and a set of files in `som-browser/src/background/` containing the code for audio

[1] https://github.com/jonasmargraf/som-browser

feature extraction and SOM calculation.

3.2.4 Background Processing

Both audio feature extraction and SOM calculation are processing intensive tasks, therefore it was clear from the beginning of the development stage that these parts of the application must be separated from the GUI that the user interacts with. While it is not possible to built a truly multithreaded application (due to the fact that the fundamental Node.js framework is single threaded), one can create separate processes to run different tasks asynchronously. This is done by creating multiple `BrowserWindow` instances, as each window is running in its own process. These windows can have their `show` flag set to `false` in order to hide them, thereby creating an invisible window for a background process.

SOM Browser initiates two consecutive background processes when the user clicks on *"Analyze"*, one for feature extraction and one that runs the SOM algorithm. This is handled by the function `handleAnalyzeClick()` in *som-browser/src/components/App.js*, which passes the necessary data to these background processes by calling `processFiles(files)` and `createSOM(files, settings)` (see Listing 5). Note the chaining of commands using several `.then()` statements: *SOM Browser* performs asynchronous operations using the `Promise` feature of ECMAScript 2015 [2].

```
      processFiles(this.state.files)
      .then(files => this.setState({ files: files, loading: false
      ↪  }))
      .then(() => {
        console.log("Building map...")
        createSOM(this.state.files, this.state.settings)
        .then(som => {
          this.setState({ som: som })
          console.log(this.state)
        })
      })
```

Listing 5: som-browser/src/components/App.js: `handleAnalyzeClick()` [excerpt]

The crucial parts of the feature extraction code can be found in Listing 6. Since the SOM implementation in `som-browser/src/background/calculateSOM.js`

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

is logically identical to what was previously discussed, we refer to Section 3.1.2 rather than dissecting the contents of `calculateSOM()` separately.

```
        // Framewise loop over audio and extract features
        for (let start = 0; start < zeroPaddedSignal.length; start += bufferSize) {

          let signalFrame = zeroPaddedSignal.slice(start, start + bufferSize)
          let frameFeatures = Meyda.extract(featureList, signalFrame)
          // we only use total loudness, not per band
          frameFeatures.loudness = frameFeatures.loudness.total

          // Append this frame's features to array of feature frames
          for (let feature in frameFeatures) {
            // Only use frames that have RMS > -60dBFS
            (frameFeatures.rms >= 0.001) && features[feature].push(frameFeatures[feature])
          }
        }

        // Get feature average
        for (let feature in features) {
          features[feature] = math.mean(features[feature])
        }

        // Add file duration to features
        features.duration = decodedAudio.duration

        // Pass averaged features to parent file
        file.features = features
        resolve(file)
```

Listing 6: som-browser/src/background/extractFeatures.js: `extractFeatures()` [excerpt]

3.2.5 User Interface Components

The following paragraphs give an overview of the different React components that make up the GUI of *SOM Browser* (refer to Figure 4 for a screenshot of the program). An outline of the entire application interface is shown in Listing 7, which contains most of the `render()` function of `som-browser/src/components/App.js`, including all components, their properties and functions.

3.2.5.1 MenuBar is the horizontal element across the top of the application window. It holds five buttons to perform audio analysis, load and export samples and store and recall calculated maps.

3.2.5.2 FileList is one of two options for content to display in the left panel. It shows a list of all loaded audio files. Each list element is rendered

```
          <MenuBar
            files={files}
            onChange={this.handleFileListChange}
            onFileClick={this.handleFileClick}
            onAnalyzeClick={this.handleAnalyzeClick}
            onSaveClick={this.handleSaveClick}
            onLoadClick={this.handleLoadClick}
            onExportClick={this.handleExportClick}
            onPrintState={this.handlePrintStateClick}
            />

          <div className="leftPanel">
              <input id="tab1" type="radio" name="tabs" defaultChecked/>
              <label htmlFor="tab1">Sounds</label>
              <input id="tab2" type="radio" name="tabs"/>
              <label htmlFor="tab2">Settings</label>
            <div className="content">
              <div id="tabFileList">
                <FileList
                  loading={this.state.loading}
                  files={files}
                  selectedFile={file}
                  onChange={this.handleFileListChange}
                  onFileClick={this.handleFileClick}
                  onAnalyzeClick={this.handleAnalyzeClick}
                  onSaveClick={this.handleSaveClick}
                  onLoadClick={this.handleLoadClick}
                  />
              </div>
              <div id="tabSettings">
                <Settings
                  filesLength={files && files.length}
                  settings={this.state.settings}
                  onChangeSettings={this.handleChangeSettings}
                  />
              </div>
            </div>
          </div>

          <Map
            som={this.state.som}
            files={this.state.files}
            progress={this.state.progress}
            selectedFile={file}
            onMapClick={this.handleMapClick}
            onMouseLeave={this.handleMouseLeave}
            />

          <FileInfo file={file} />

          <UserSelection
            userSelection = {this.state.userSelection}
            onClick={this.handleMapClick}
            onUserSelectionUpdate={this.handleUserSelectionUpdate}
            />
```

Listing 7: som-browser/src/components/App.js: GUI Components by a nested `FileListItem` component. When clicking an item in the list, the corresponding map square will be highlighted and the `FileInfo` panel on the right will display information about the file.

3.2.5.3 Settings is the other display option for the left panel. Here, a number of SOM parameters can be set, similar to the Max messages listed in Table 1.

3.2.5.4 Map is the main focal point of the application. It displays a grid of grey and white squares, where each white square represents one sound. The `Map` component is made up of three nested components: `MapNode` (the grey background squares that are rendered first), `MapSubNode` (the white squares representing sounds) and `MapLabel` (the label displaying the name of the sound over which the user is currently hovering).

3.2.5.5 FileInfo is the panel on the right side of the application. It displays details about the currently selected file, including its path, duration and audio feature values.

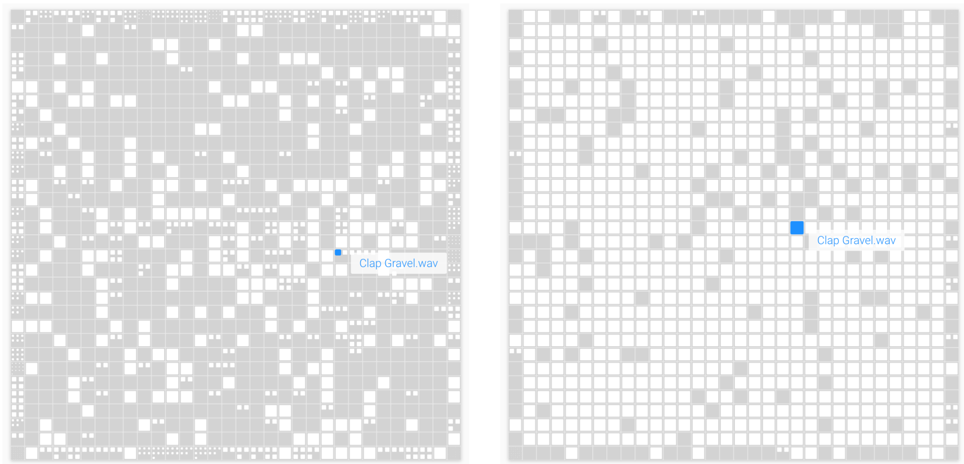
3.2.5.6 UserSelection is the horizontal element across the bottom of the application. It consists of eight instances of the nested component `UserSelectionSlot`, each representing a spot into which the user can drag a sound to keep as a reference or for later exporting.

3.2.6 Algorithm Extension: Forced Node Population

After the SOM algorithm was first implemented in *SOM Browser*, it became apparent that large parts of almost all created maps remained empty, which proved frustrating to interact with since these empty areas are essentially "dead" spots where no sounds are located and no interaction is possible. A method to counteract this phenomenon was deemed necessary.

In order to avoid "empty" nodes on the SOM - meaning nodes to which no input vectors are mapped - a post-processing extension was added to the original algorithm that inverts the mapping process, explicitly iterates over empty nodes and assigns each one that input vector which is closest. This algorithm extension, which we call Forced Node Population (Forced Node Population (FNP)), executes the following sequence after the regular SOM has been calculated (refer to Listing 8 to see the actual source code implementation):

1. Select a random empty node.
2. Find closest vector for that node and assign it to this node.
3. Remove this vector from the possible choices.
4. Repeat.



(a) *SOM Browser* without FNP (b) *SOM Browser* with FNP

Fig. 7: Comparison of *SOM Browser* maps with (7b) and without (7a) FNP

This process is only performed once for all nodes that are empty immediately after the initial SOM calculation. It is possible that the forced node population creates new empty nodes, but in order to minimize distortion introduced by this procedure (see Results in section 5), it is not repeated. The effect of the FNP extension be clearly seen in 7. For a closer look at the results of this algorithm extension, please refer to Results (section 5).

```
function populateEmptyNeurons(som) {

  let emptyNeuronIndeces = som.neuronAssignedFiles.map((e,i) => {
    return (e === null ? i : false)
  })
  .filter(e => e !== false)

  let tempVectors = som.normalizedData
  let tempVectorIndeces = tempVectors.map((e,i) => i)

  while (emptyNeuronIndeces.length >= 1) {
    // Get random empty neuron, then remove from possible choices
    let emptyNeuronIndex = math.pickRandom(emptyNeuronIndeces)
    emptyNeuronIndeces.splice(emptyNeuronIndeces.indexOf(emptyNeuronIndex), 1)
    let emptyNeuron = som.neurons[emptyNeuronIndex]

    let distancesFromNeuron = tempVectorIndeces.map((e,i) => {
        return math.norm(math.subtract(emptyNeuron, tempVectors[e]))
    })

    let nearestVectorIndex = tempVectorIndeces[distancesFromNeuron.indexOf(
      math.min(distancesFromNeuron))]
    // Remove the found closest vector from its previously assigned neuron
    // and instead assign it to the empty neuron.
    let oldAssignedNeuronIndex = som.neuronAssignedFiles.findIndex(e =>
      Array.isArray(e) && e.some(el => el === nearestVectorIndex))
    som.neuronAssignedFiles[oldAssignedNeuronIndex].splice(
      som.neuronAssignedFiles[oldAssignedNeuronIndex].findIndex(
        e => e === nearestVectorIndex), 1)
    som.neuronAssignedFiles[emptyNeuronIndex] = [nearestVectorIndex]

    tempVectorIndeces.splice(distancesFromNeuron.indexOf(
      math.min(distancesFromNeuron)), 1)
  }

  return som
}
```

Listing 8: som-browser/src/background/calculateSOM.js: `populateEmptyNeurons()` (FNP implementation)

4 Evaluation

In order to evaluate the SOM algorithm as implemented in this thesis, as well as the developed SOM Browser application as a whole, a two-part process was employed. First, a set of numerical metrics was selected to quantify aspects of the algorithm we deem salient when judging its effectiveness for sound corpus organization. Second, a series of five semi-structured interviews was designed, conducted and subsequently analyzed. The following sections go into detail about the selection of a data set of sound files for the evaluation, the metrics employed and the design of the interview.

4.1 Sound Corpus Selection

A crucial aspect for the evaluation of the work presented in this thesis is the choice of an appropriate data set of audio files to serve as a prototypical sound corpus. Ideally, two key conditions should be met by this corpus. It should be *ecologically valid*, meaning here that it should approximate a real-world sample library that would actually be used by contemporary music producers, and it should be a *well-established* data set which has been validated through use in other research, allowing for direct comparisons between results. Preferably, something akin to the Giant Steps data sets (Knees et al., 2015) for tempo and key detection should be used. In addition to identifying the aforementioned two conditions, the decision was made to only select "one-shot" drum and percussion sounds (meaning single instrument hits, no loops or other longer sounds) in order to evaluate a single, concrete use case and limit the scope of this evaluation.

Data sets used in previous research vary and it is often not possible to clearly establish provenance due to insufficient information being given by the authors (see for example Fried et al. (2014) and Shier et al. (2017), two papers which present important related work but fail to clearly identify the source of their employed sound files). Two established databases that have been cited in the literature are ENST-Drums (Gillet and Richard, 2006) and the RWC Music Database (Goto et al., 2002). However, neither of these data sets proved appropriate for this evaluation since they both contain only acoustic source material and, especially in the case of RWC, largely consist of longer musical passages instead of the single "one-shot" hits mentioned above.

For the reasons outlined above, the author decided to forego the condition that the selected sound corpus be a data set well-established through previous research. Because of this, more emphasis is placed on the requirement for ecological validity. In order to maximize real-world conditions, the sample library *Drum Essentials* (Ableton, 2019) was selected to serve as a sound corpus for this evaluation. It is a collection of samples created by the German music software company Ableton AG that is distributed to owners of the company's flagship product, the DAW *Ableton Live* (Ableton AG, 2019). As part of a commercially available product, this corpus of sound files does not just approximate a real-world sample library, it is an actual example of such a library and is, for the purpose of this thesis, considered representative of sample libraries used in a modern music production workflow. One additional benefit of using the selected sample library is the advantage of a single, clearly identifiable source of the data - it is made available as a professional product by Ableton AG. An alternative approach would have been to manually select sounds from places like *Freesound.org* (Font et al., 2013), where all files are licensed in a way that makes them free to use, but their quality is not guaranteed to be consistent, or to scour through sample libraries shared on various online forums, which brings along issues of copyright and expired links, making it hard to trace the files' origins.

The *Drum Essentials* collection as distributed by Ableton consists of 1181 one-shot samples, each in a separate audio file, as well as supplementary content, such as MIDI clips and effects presets. Only the raw audio files are used in the presented work. These sound files present a mixture of acoustic and electronic sounds stemming from a variety of drums and percussion instruments. The library is organized by instrument group, of which there are 17 in total. The names of these groups, as well as the number of sounds per group can be found in table 2. Some sound files appear in more than one group. These duplicates have been removed, so that every sound only appears once throughout the entire data set. The remaining number of sound files is 1081.

4.2 Metrics for SOM Analysis

In order to evaluate the SOMs created using the SOM Browser application and the *Drum Essentials* test data set, three core metrics are used: quantization induced by the SOM, map emptiness, and the ratio between nodes and their assigned vectors. These metrics and the motivation behind them are outlined further in the following paragraphs.

4.2.1 SOM-Induced Quantization

Fundamental to the SOM principle is the idea of mapping vectors to their corresponding BMUs, those nodes that are closest to them (see section 2.2). Several vectors can be assigned to one node - this can also be thought of as

Drum Essentials

Instrument Category	**Count**
Bell	19
Bongo	6
Clap	71
Conga	27
Cymbal	54
Electronic Percussion	49
FX Hit	64
Hihat	167
Kick	166
Misc. Percussion	64
Ride	40
Rim	65
Shaker	39
Snare	181
Tambourine	23
Tom	138
Wood	8

Tab. 2: Sound file counts per instrument category of the *Drum Essentials* sample library

a quantization process, where the absolute difference between the positions of vector x_t and node m_c is the quantization error Δ_t for that vector:

$$\Delta_t = |x_t - m_c| \tag{21}$$

As a metric for the SOM, the quantization errors for all vectors can be averaged, as well as their distribution examined. In order to maximize information preservation, quantization errors should be minimized.

4.2.2 Vector-Node Count

A second metric that was devised in order to quantify SOM quality is the count C_i of vectors $x_1, ..., x_n$ mapped to a node m_i and subsequently the distribution of those counts across the map. Ideally, this distribution should look like a single, narrow spike - meaning that (almost) all nodes have about the same number of vectors assigned to them, resulting in an even distribution of sounds across the SOM Browser map interface.

4.2.3 Map Emptiness

Another relevant aspect of the created SOMs, and the third metric employed here, is how much of the map remains "empty", meaning how many nodes were not assigned any vectors. We define this "map emptiness" metric ME as the number of nodes $m_1, ...m_n$ whose vector-node count $C_n = 0$ (see section 4.2.2), divided by the total number of nodes m_i. For the purpose of making optimal use of the space alloted to the map in the SOM Browser GUI, emptiness should be minimized so that users encounter the least amount of "blind spots" possible.

4.2.4 Influence of Forced Node Population

Since the concept of Forced Neuron Population is an addition to the SOM algorithm introduced in this work (see section 3.2.6), its influence on the SOM should also be evaluated. Therefore, the aforementioned metrics were calculated both with and without FNP.

4.3 Semistructured User Interviews

In order to evaluate the SOM Browser application prototype presented in this thesis, five semi-structured interviews with working audio professionals were conducted. These interviews were conducted by the author and consisted of a set of questions as well as observed user interaction with the prototype software. For this evaluation, a guide including questions outlining the structure of the interview as well a set of ratings scales was created. Subjects were asked about their experience with sample libraries and their current workflow, and to interact with a sample library in a file browser environment as well as using the SOM Browser software. Audio from the conversations was recorded and subsequently analyzed.

4.3.1 Motivation to Conduct Interviews

This evaluation procedure entails two aspects, namely a semi-structured interview series and a qualitative analysis of the collected responses. The decision to conduct qualitative interviews stems from the exploratory nature of the presented work. In order to assess the merit of the developed interface in its present state, direct feedback from potential users was sought, which Lazar et al. (2017) refers to as "fundamental to human-computer-interaction (HCI) research" (see Lazar et al. (2017, p.187)). But the motivation for a direct conversation with users was not only to evaluate the presented interface proposition, but also for these interviews to serve as an exploration of users' current situation, to hear about their own experience of it and to see what advantages and shortcomings they identify in their present workflows. In short, these interviews were motivated by a desire to gain some understanding of the complex situation that is sample library interaction in a music production environment and to gauge initial reactions to the developed prototype alternative. The semi-structured approach was chosen in order to be able to react to interviewees' responses more freely and allow the interviewer to ask follow up questions when deemed necessary. Naturally then, the gathered responses cannot simply be quantified, which makes a qualitative approach to their analysis a fitting choice.

There are of course downsides to the chosen approach. Conducting interviews is time-consuming, as it has to be done on a one-on-one basis and often (as in the case of this work) in person. After the interview is over, additional time and effort goes into transcribing and annotating the responses. This severely limits the number of participants that can feasible be recruited for a study, as is evident by the small number of five participants here. Lazar et al. (2017) identifies another disadvantage of interviews: "[...] data collection that is separated from the task and context under consideration [...] suffer[s] from problems of recall. [...] [I]t is, by definition, one step removed from reality" (Lazar et al., 2017, p.188ff.). Because of this, we follow the authors' suggestion of combining the interview with user observation.

4.3.2 Interview Subject Selection

The SOM Browser application is not aimed at the general population. Instead, it has been designed for specialized users that work in modern music production, as they constitute the potential future user base of an application like the one presented here.

In order to increase the validity and relevance of potential subjects' responses, the decision was made to interview only working professionals for this evaluation and to not include hobbyists or people without any experience in music production.

Subjects were recruited by inquiring about qualified candidates (in other words, people working professionally in modern music production) in the wider circle of acquaintances of the author. No compensation was offered and only sparse information about the nature of the research was given beforehand in order to minimize the possibility of instilling biases in subjects. Most importantly, subjects were asked to participate in an interview about sample library organization, but were not told that they would be shown software developed by the author.

4.3.3 Informed Consent Form

For the purpose of documenting participants agreement to be interviewed, an informed consent form was created for the interview series. This document outlines basic information about the purpose and content of the interview and its duration. It also lists all data that will be collected and explains the procedure used for data anonymization in order to protect subjects' privacy. Lastly, it informs participants of their rights to withdraw their consent to the usage of their data for research purposes and have it erased. This form was based on a template provided by the ethics commission of Technische Universität Berlin (TU Berlin) on their website (TU Berlin, 2019). The form used by the author can be found in XXX REF APPENDIX HERE XXX.

4.3.4 Test Subject Code Design

To ensure proper data anonymization, a test subject code was used. This code is comprised of a series of letters and numbers and was created at the beginning of the interview by the subjects themselves according to a set of instructions. All data and responses of the subjects were directly labelled with this code, so that individuals' names were never used. This code design procedure was again based on a template by the ethics commission of TU Berlin and can be found on the same website as the information concerning consent forms (TU Berlin, 2019). The instruction sheet that was distributed to subjects can be found in XXX REF APPENDIX HERE XXX.

4.3.5 Interview Structure

The guide developed for this interview can be found in XXX REF APPENDIX HERE XXX It outlines a three part structure: first, some general questions about subjects' usage of sample libraries. Second, some guided interaction with a predetermined sample library in a traditional file browser structure on a computer. In the third section, the SOM Browser application is finally introduced and subjects are asked to use it and describe their impression of it.

4.3.6 Question Design

The general composition employed for most questions is twofold, combining closed- and open-ended approaches: first, participants are asked to give a rating on a predefined scale (see 4.3.7 below). Then, participants are free to elaborate on their answer and explain their rating. If they don't initiate this themselves, a follow-up question along the lines of "Could you tell me why you chose this rating?" is asked.

4.3.7 Selection of Ratings Scales

In order to record subjects' ratings, 6 point Likert scales were used (as is common in Human-Computer Interaction (HCI) research, see Lazar et al. (2017, p.31, p.93)). The difference between even and uneven anchor counts in Likert scales lies in the presence (in the case of uneven anchor counts) or lack (for even counts) of a "neutral" middle option. Choosing scales without neutral mid-points was motivated by a desire to encourage subjects to make a definite choice with regard to their rating. For a short look at the effects of eliminating the mid-point, see Garland (1991). The scales presented to subjects were explicitly labeled textually instead of numerically. The anchor points were designed using two polar adjectives (such as "positive" and "negative") and a consistent, three-tiered set of adjective qualification with "very" marking the strongest option, followed by the adjective without qualifier and then "somewhat" as the weakest variant. The resulting scale for a positive/negative rating is composed of the following anchors: very positive, positive, somewhat positive, somewhat negative, negative, very negative. The selection of these qualifiers and appropriate anchors in general was inspired partially by Vagias (2006). The full set of scales used for the conducted interviews can be found in XXX REF APPENDIX HERE XXX.

4.3.8 Questions Used

In section 1, which serves as an introduction for the interviewee, general administrative requirements such as the signing of the consent form and a topical introduction of the research are taken care off. This is then followed by two simple Yes/No questions to establish whether the subject works with third-party and personally created sample libraries (see questions 1.1 and 1.2).

Section 2 begins with a presentation of the *Drum Essentials* sample library to the subject. This presentation includes the information that it is a library of drum samples that consists of around 1000 sound files which are organized in subfolders according to the respective instrument, such as kick drum, snare drum, hi-hat, and so forth. The interviewee is invited to explore the sample library using the laptop that it is being presented on.

Then, in question 2.1, subjects are asked to describe how to approach familiarizing themselves with the provided sample library in order to use its contents in a hypothetical work project of theirs.

Question 2.2 follows this up with a request for a rating of the subject's level of satisfaction with the workflow that they outlined.

In the third and final section of the interview, the SOM Browser software is introduced to participants. At first, a general overview of the interface is given, in which the interviewer mentions the map layout in the middle (without explaining the nature of its organization), the file list on the left, the file info panel on the right and the favorites bar at the bottom.

The subject is then asked to try out the software and explore its interface for a short period of time. Thereafter, they are asked to give a rating of their overall first impression of the software on a positive/negative scale (see question 3.1). Then, a follow-up question about their opinion on what does or does not work is posed.

In question 3.2, subjects are required to rate the interface's ease of use.

Question 3.3 inquires specifically about the understandability of the language used.

3.4 and 3.5 are open-ended questions aimed at subjects' interpretation of the organization of sounds in the map layout: 3.4. asks what subjects think about the organization, while 3.5 inquires specifically about a guess as to what the axes represent.

Question 3.6 then asks subjects to state whether or not they have a preference between the traditional file browser layout presented in section 2 or the SOM Browser interface shown in section 3.

The last ratings question of the interview, 3.7 requests interviewees to assess their level of comfortability with the software.

Finally, in 3.8 subjects are asked if they would consider using the presented software tool and what changes they would like to see.

The full interview guide including all questions can be found in XXX REF APPENDIX HERE XXX.

5 Results

This is the Results section.

6 Discussion

This is the Discussion.

6.1 Outlook

7 References

Ableton, AG (2019): *Drum Essentials*. Online. URL `https://www.ableton.com/en/packs/drum-essentials/`. Access 7.2.2019.

Ableton AG (2019): *Ableton Live 10*. Online. URL `https://www.ableton.com/en/live/`. Access 7.2.2019.

Barbiero, Phillip (2017): *pbarbiero/basic-electron-react-boilerplate: Modern and Minimal Electron + React Starter Kit*. Online. URL `https://github.com/pbarbiero/basic-electron-react-boilerplate`. Access 7.2.2019.

Bauer, H.-U.; Ralf Der; and Michael Herrmann (1996): "Controlling the magnification factor of self-organizing feature maps." In: *Neural computation*, **8**(4), pp. 757–771.

Cycling '74 (2019): *Max*. Online. URL `https://cycling74.com/`. Access 7.2.2019.

de la Cuadra, Patricio (2019): "Pitch Detection Methods Review." URL `https://ccrma.stanford.edu/~pdelac/154/m154paper.htm`.

Facebook (2019): *React*. Online. URL `https://reactjs.org/`. Access 7.2.2019.

Fletcher, Harvey and Wilden A Munson (1933): "Loudness, its definition, measurement and calculation." In: *Bell System Technical Journal*, **12**(4), pp. 377–430.

Font, Frederic; Gerard Roma; and Xavier Serra (2013): "Freesound technical demo." In: *Proceedings of the 21st ACM international conference on Multimedia*. ACM, pp. 411–412.

Fried, Ohad; Zeyu Jin; Reid Oda; and Adam Finkelstein (2014): "AudioQuilt: 2D Arrangements of Audio Samples using Metric Learning and Kernelized Sorting." In: *NIME*. pp. 281–286.

Garland, Ron (1991): "The mid-point on a rating scale: Is it desirable." In: *Marketing bulletin*, **2**(1), pp. 66–70.

Gillet, Olivier and Gaël Richard (2006): "ENST-Drums: an extensive audio-visual database for drum signals processing." In: *ISMIR*. pp. 156–159.

GitHub (2019): *Electron*. Online. URL `https://electronjs.org/`. Access 7.2.2019.

Goto, Masataka; Hiroki Hashiguchi; Takuichi Nishimura; and Ryuichi Oka (2002): "RWC Music Database: Popular, Classical and Jazz Music Databases." In: *ISMIR*, vol. 2. pp. 287–288.

Knees, Peter; et al. (2015): "Two Data Sets for Tempo Estimation and Key Detection in Electronic Dance Music Annotated from User Corrections." In: *ISMIR*. pp. 364–370.

Kohonen, Teuvo (1990): "The Self-Organizing Map." In: *Proceedings of the IEEE*, **78**(9), pp. 1464–1480.

Kohonen, Teuvo (2001): *Self-Organizing Maps*, vol. 30 of *Springer Series in Information Sciences*. Heidelberg: Springer.

Kohonen, Teuvo (2005): "The Self-Organizing Map (SOM)." URL `http://www.cis.hut.fi/somtoolbox/theory/somalgorithm.shtml`.

Kohonen, Teuvo and Timo Honkela (2007): "Kohonen network." URL `http://www.scholarpedia.org/article/Kohonen_network`.

Lazar, Jonathan; Jinjuan Heidi Feng; and Harry Hochheiser (2017): *Research methods in human-computer interaction*. Morgan Kaufmann.

Lerch, Alexander (2012): *An introduction to audio content analysis: Applications in signal processing and music informatics*. Wiley-IEEE Press.

Lykartsis, Athanasios (2014): *Evaluation of accent-based rhythmic descriptors for genre classification of musical signals*. Master's thesis, Master's thesis, Audio Communication Group, Technische Universität Berlin . . . .

Mathieu, Benoit; Slim Essid; Thomas Fillon; Jacques Prado; and Gaël Richard (2010): "YAAFE, an Easy to Use and Efficient Audio Feature Extraction Software." In: *ISMIR*. pp. 441–446.

Merenyi, Erzsbet; Abha Jain; and Thomas Villmann (2007): "Explicit magnification control of self-organizing maps for "forbidden" data." In: *IEEE Transactions on Neural Networks*, **18**(3), pp. 786–797.

Moffat, David; David Ronan; Joshua D Reiss; et al. (2015): "An evaluation of audio feature extraction toolboxes." In: .

Moore, Brian CJ; Brian R Glasberg; and Thomas Baer (1997): "A model for the prediction of thresholds, loudness, and partial loudness." In: *Journal of the Audio Engineering Society*, **45**(4), pp. 224–240.

Oppenheim, Alan V and Ronald W Schafer (2014): *Discrete-time signal processing*. Pearson Education.

Peeters, Geoffroy (2004): *A large set of audio features for sound description (similarity and classification) in the CUIDADO project*. Tech. rep., IRCAM.

Rawlinson, Hugh; Nevo Segal; and Jakub Fiala (2015): "Meyda: an audio feature extraction library for the web audio api." In: *The 1st Web Audio Conference (WAC). Paris, Fr.*

Rawlinson, Hugh; Nevo Segal; and Jakub Fiala (2019a): "Meyda: Audio feature extraction for JavaScript." URL `https://meyda.js.org/audio-features`.

Rawlinson, Hugh; Nevo Segal; and Jakub Fiala (2019b): "Meyda: Audio feature extraction for JavaScript." URL `https://github.com/meyda/meyda`.

Schnell, Norbert; et al. (2009): "MuBu and friends–assembling tools for content based real-time interactive audio processing in Max/MSP." In: *ICMC*.

Schnell, Norbert; et al. (2019a): *MuBu for Max - A toolbox for Multimodal Analysis of Sound and Motion, Interactive Sound Synthesis and Machine Learning*. Online. URL `http://ismm.ircam.fr/mubu/`. Access 7.2.2019.

Schnell, Norbert; et al. (2019b): *MuBu for Max - A toolbox for Multimodal Analysis of Sound and Motion, Interactive Sound Synthesis and Machine Learning*. Online. URL `http://forumnet.ircam.fr/product/mubu-en/`. Access 7.2.2019.

Schwarz, Diemo; Grégory Beller; Bruno Verbrugghe; and Sam Britton (2006): "Real-time corpus-based concatenative synthesis with catart." In: *9th International Conference on Digital Audio Effects (DAFx)*. pp. 279–282.

Shier, Jordie; Kirk McNally; and George Tzanetakis (2017): "Analysis of Drum Machine Kick and Snare Sounds." In: *Audio Engineering Society Convention 143*. Audio Engineering Society.

Torvalds, Linus and Junio Hamano (2019): *Git*. Online. URL `https://git-scm.com/`. Access 7.2.2019.

TU Berlin, Ethik-Kommission (2019): *Ethik-Kommission*. URL `https://www.ipa.tu-berlin.de/menue/einrichtungen/gremienkommissionen/ethik_kommission/`.

Vagias, Wade M (2006): "Likert-type Scale Response Anchors. Clemson International Institute for Tourism." In: *& Research Development, Department of Parks, Recreation and Tourism Management, Clemson University.*

Vesanto, Juha; Johan Himberg; Esa Alhoniemi; and Juha Parhankangas (2000): "SOM toolbox for Matlab 5." In: *Helsinki University of Technology, Finland*, p. 109.

World Wide Web Consortium (W3C) (2019): *Web Audio API.* Online. URL `https://www.w3.org/TR/webaudio/`. Access 7.2.2019.

Yin, Hujun (2007): "Nonlinear dimensionality reduction and data visualization: a review." In: *International Journal of Automation and Computing*, **4**(3), pp. 294–303.

Zwicker, Eberhard (1961): "Subdivision of the audible frequency range into critical bands (Frequenzgruppen)." In: *The Journal of the Acoustical Society of America*, **33**(2), pp. 248–248.

Appendices

A LaTeX Sources

The LaTeX sources for this work can be found in XXX.

B Thesis Bibliography

The references used in this work can be found in XXX.

Acronyms

API Application Programming Interface.

BMU Best Matching Unit.

DAW Digital Audio Workstation.

FFT Fast Fourier Transform.

FNP Forced Node Population.

GUI Graphical User Interface.

HCI Human-Computer Interaction.

IRCAM Institut de recherche et coordination acoustique/musique.

NaN Not a Number.

PCA Principal Component Analysis.

RAM Random-Access Memory.

SOM Self-Organizing Map.

TU Berlin Technische Universität Berlin.

List of Figures

List of Listings

List of Tables

Digital Resource

This page holds a data disk.