Dynamic Classification Models for Human-Machine Improvisation and Composition

Master Thesis

Joakim Borg

Aalborg University Copenhagen
Sound and Music Computing

AALBORG UNIVERSITY
STUDENT REPORT

Sound and Music Computing
Aalborg University
http://www.aau.dk

Title:
Dynamic Classification Models for Human-Machine Improvisation and Composition

Theme:
Human-Machine Musical Improvisation

Project Period:
Spring Semester 2020

Project Group:
-

Participant(s):
Joakim Borg

Supervisor(s):
Stefania Serafin
Gérard Assayag

Copies: 1

Page Numbers: 77

Date of Completion:
May 27, 2020

Abstract:

This thesis presents a number of updates to an existing real-time system for human-computer improvisation, among them a modular framework for integrating new models for analysis and classification, an offline framework and scheduling solution to adapt the system to composition as well as its original purpose of real-time improvisation, and a framework for numerical evaluation of the system in terms of usability and quality. New harmonic classifiers have been added to the system and evaluated with a number of musical corpora, representing a number of genres and instrumental settings. The results indicate that the new classifiers outperform the original classifier in terms of usability on average, greatly so in more difficult cases. In terms of quality, the results indicate little difference between the classifiers, but further studies will be required to find models to quantify harmony based on the way it is used in the system.

AALBORG UNIVERSITET
STUDENTERRAPPORT

Sound and Music Computing
Aalborg Universitet
http://www.aau.dk

Titel:
Dynamiska klassifikationsmodeller för människa-datorimprovisation och -komposition

Tema:
Människa-datorimprovisation

Projektperiode:
Förårssemestern 2020

Projektgruppe:
-

Deltager(e):
Joakim Borg

Vejleder(e):
Stefania Serafin
Gérard Assayag

Oplagstal: 1

Sidetal: 77

Afleveringsdato:
27. maj 2020

Abstract:

Denna uppsats presenterar ett antal uppdateringar till ett existerande realtidssystem för människa-datorimprovisation, bland annat ett modulärt ramverk för att integrera nya analys- och klassifikations-modeller, ett ramverk för komposition, utöver dess ursprungliga användningsområde realtidsimprovisation, samt ett ramverk för numerisk utvärdering av systemet. Ett antal nya modeller för klassifikation har lagts till och utvärderats med ett antal musikstycken med syfte att representera ett flertal genrer och instrumentationer. Resultaten visar att de nya klassifikationsmodellerna har en högre användarvänlighet än systemets ursprungliga modeller, men indikerar på en svag skillnad i kvalitet.

Contents

Preface

This thesis was written as the final project of the Sound and Music Computing graduate programme at Aalborg University Copenhagen. The work was carried out between February 1st and May 27th. I would like to thank my supervisors Stefania Serafin and Gérard Assayag for their guidance during the course of this work.

Aalborg University, May 27, 2020

[illegible]

Joakim Borg
<jborg18@student.aau.dk>

Chapter 1

Introduction

Computer-generated music has a long history, starting with the Illiac Suite [18] from 1957, which is generally considered as one of the first pieces composed by a computer. But even long before the invention of computers, algorithmic approaches to composition have been used. Mozart's Musikalisches Würfelspiel from 1792 is commonly attributed as one of the first examples of the kind, but as Herremans points out, there are at least twenty examples of algorithmic compositions published between 1757 and 1812 [17], revolving around rules and/or pre-composed material. Algorithmic approaches to composition became more frequent in the postwar period with composers like John Cage and to some extent the entire serialism school, and in the past decades the field has grown significantly with the advances in technology and the availability of computers.

Today, there are a number of branches in the domain of music-generating systems. One branch of particular interest is that of corpus-based interactive music systems, which is a mode of human-computer music improvisation. Nika [27] gives the following description:

> The corpus-based interactive music systems create music from a musical memory constituted by offline corpora and / or live material. Sequences in this memory are searched, retrieved, transformed, and concatenated to generate the machine improvisation [...]

This thesis is based on a system called Somax, originally presented in [6], which is an example of a corpus-based interactive music system. In this work, the system is extended, generalized and evaluated, with the purpose of (a) adapting the system towards (offline) compositional practices, in parallel with its original domain of real-time improvisation, and (b) modularizing the system to allow new research to be integrated into the system.

In the following section, the context of the system and state of the art in the field will be presented in further detail. In chapter 2, the theoretical framework of the system will be presented, briefly reiterating the content of [6] and [7], where much of the original system was presented, but also adding a number of new aspects to the framework. In chapter 3, the realization and temporal context of the theoretical framework, with a detailed description of relevant aspects of the code, will be presented. Chapter 4 describes the procedure used to evaluate the system as well as the results of the evaluation, which will be further discussed in chapter 5.

1.1 Related Works

Many of the early digital systems for music generation were primarily rule-based, i.e. systems where the music was created by rules and/or internal logic. Among them are for example Lewis' Voyager system [22], David Cope's EMI (Experiments in Musical Intelligence) [10] as well as a number of systems and their related compositions by Iannis Xenakis [35]. Over time, this domain of music generation, which often goes by the name Computer-aided composition, has been extended with a number of concepts from the field of machine learning, for example Markov models and neural networks. For a more complete review of the different paradigms in computer-aided composition, see [26].

In the same domain, a number of libraries and framework aimed toward composers and musicians have been developed, often in a visual-programming environment. In this category, we see a number of applications developed at Ircam [2], where OpenMusic [8] is among the more famous examples. Another example is the Bach library [1], a toolbox for computer-aided composition for Max/MSP [11], which among other things adapts and integrates several concepts from OpenMusic into Max/MSP.

In recent years, with the progress in the field of deep learning, several systems have adapted deep learning into the domain of music generation, where the Magenta project [23] is among the more prominent ones. In most cases, systems based on deep learning rely on large datasets in the form of audio and/or other music representations, and are often able to generate longer sequences [30] or entire compositions [19] [32] [16] convincingly in the style of the dataset. For an exhaustive overview of the field of deep learning for music generation, see [9]. While many deep learning systems for music generation are able to produce convincing compositions, they lack usefulness as tools for composers since they (a) are limited to style constraints that require large amounts of data and (b) the user often has limited control over the output of the system. The latter is however not always true, there are a number of examples of parametric deep learning. Among them is for example the DeepBach system [16] which, on top of generating compositions in the form of four-part chorales in the style of J.S. Bach, also can harmonize user-defined melodies, and there are a number of systems in the Magenta Project [23] that allows some degree of user input and/or parametric control.

A third branch, in-between the rule-based approaches and the data dependent deep learning approaches, stems from the OMax system [3] [21], a corpus-based machine improvisation system which relies on a user-specified corpus (small dataset) and uses concatenative synthesis - producing output by recombining the order of small segments of the corpus, to some extent similar to the process of granular synthesis but with much larger grains - where the corpus role is similar to that of a user-controlled parameter, heavily influencing the generated result. Over the years, a number of systems have been developed using mechanics similar to or stemming from OMax, among them the PyOracle [33], Mimi4x [14], Somax [6][7], ImproteK [29] and the DYCI2 library [28]. For a complete taxonomy of these systems (and many others), refer to [17].

As can be seen in [17], most of these systems rely on some sort of classification of individual segments of the corpus as well as in some cases user input (from a musician or other sound sources) with regards to musical features, for example pitch, rhythm, timbre or harmony, and the output is largely determined by these values. The Somax system, which much of the work presented in this thesis is built on, relies on classification of pitch and harmony (quantified as chroma), and to some extent tempo/beat, but does not integrate any models for timbre, dynamics, time signature, etc. At the same time, there's an entire research field of music information retrieval (MIR), mainly represented by ISMIR [20] and its sub-scene MIREX [25], where a large portion of the research is related to estimating and classifying both high level features (for example genre, mood and key of an entire file) as well as low level features (for example chord analysis, onset detection and multi-f0 estimation) that would be relevant for the framework. As several of the corpus-based machine improvisation systems are written in and/or integrated with Python, the LibROSA [24] and Madmom [5] could be used to further integrate modes of MIR into the systems.

As mentioned, the work presented in this thesis is largely built on the Somax system, which was introduced in [6] and further developed in [7]. One of the main focuses of this work is to not necessarily add new features for classification, but design a modular framework where classifiers based on new (or existing) features can be integrated dynamically, allowing new research to be integrated as it becomes available. The thesis will also present a number of updates to bridge the gap between real-time human-machine improvisation and (offline) computer-aided composition, adapting Somax to be able to work in both contexts. Finally, a framework and procedure for evaluating the system and any newly implemented classifiers will be presented.

Chapter 2

Fundamentals

This chapter describes the theoretical foundation of the system, which will serve as a basis for chapter 3, where it will be further expanded. This chapter is largely based on [7], but describes the system in more detail, where a number of aspects were omitted in [7]. It also introduces a number of novelties presented for the first time in this thesis.

2.1 Overview

The main purpose of the system presented in [6] and [7] and described in this thesis is to generate musical material based on an external, already existing material. The principle through which it generates new material is called concatenative synthesis, and is (as briefly mentioned in section 1.1) to some extent alike that of granular synthesis, where very short pieces of audio are sampled from a preselected material and recombined. In this system, however, the grains (or "slices", as they will be denoted from here on) are much longer than in a granular synthesizer - either the duration of a beat or the length between two note onsets - and the output selection is based on listening to a second, external input, and the system is continuously selecting the most suitable slice using statistical machine learning. In other words, the system is improvising around a musical material and in real-time adapting to a musician (or any other sound source).

In practice, this behaviour is realized by learning the sampled material, which may be either audio or midi, in multiple layers, where each layer listens to a single feature (or "trait", as they will be denoted from here on) of the material, for example pitch, chroma, mfcc, velocity, etc. At runtime, each layer then listens to the corresponding trait of a second, external audio and/or midi source and continuously matches this to the learned material, generating activations in the memory where matches are found. The output is then selected from the point in the memory with the most activity, after the activities in all layers has been merged and scaled. The activities generated at earlier points in time also remain in the memory for some time, thus impacting future time steps and with that simulating a short-term memory with respect to the original material.

The process of learning the sampled material or constructing a corpus will be described in section 2.2. The listening, from here on denoted as influencing, is described in section 2.3, and finally the generation of output, which for practical reasons is decoupled from the influencing process, is described in section 2.4.

2.2 Corpus

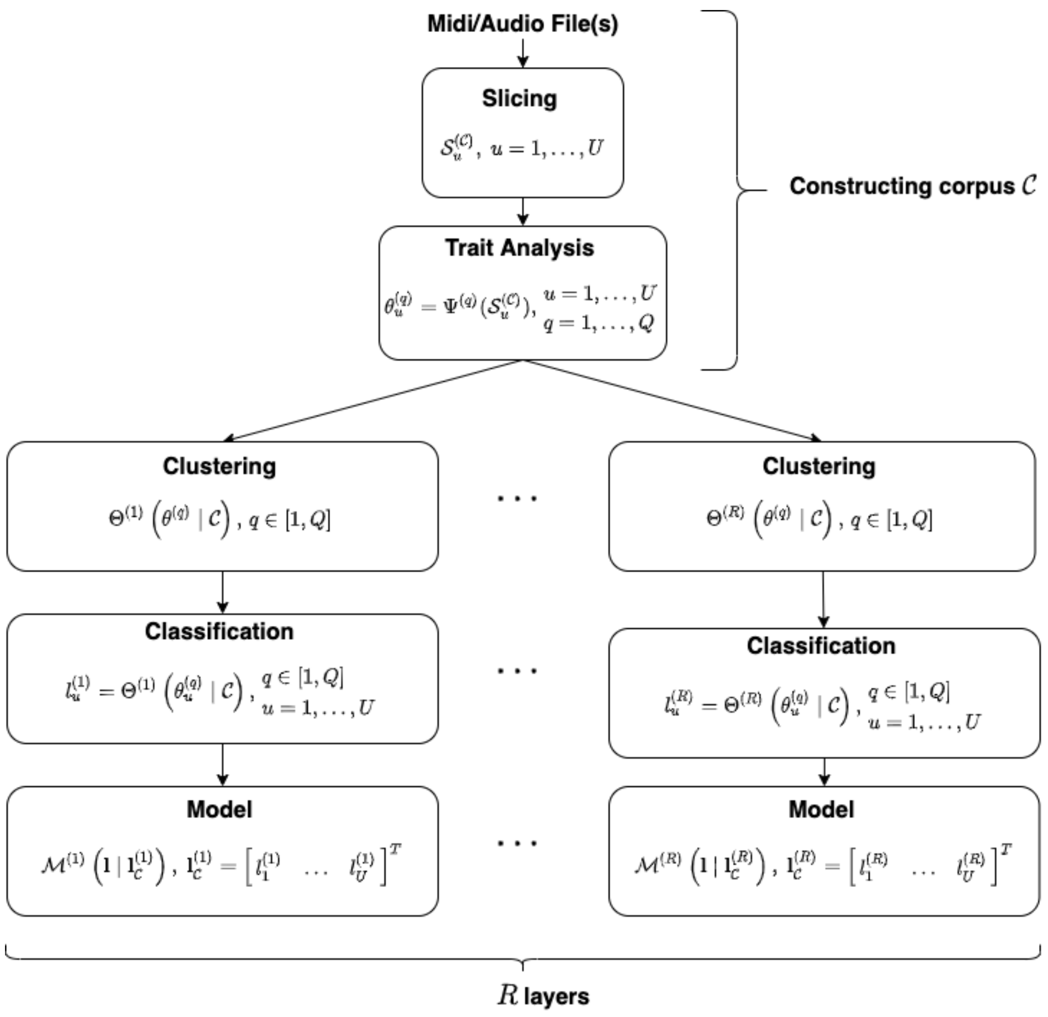

Midi/Audio File(s)
Slicing
$\mathcal{S}_u^{(\mathcal{C})}, u = 1, \ldots, U$
Trait Analysis
$\theta_u^{(q)} = \Psi^{(q)}(\mathcal{S}_u^{(\mathcal{C})}), u = 1, \ldots, U, q = 1, \ldots, Q$
Constructing corpus $\mathcal{C}$
Clustering
$\Theta^{(1)}\left(\theta^{(q)} \mid \mathcal{C}\right), q \in [1, Q]$
$\Theta^{(R)}\left(\theta^{(q)} \mid \mathcal{C}\right), q \in [1, Q]$
Classification
$l_u^{(1)} = \Theta^{(1)}\left(\theta_u^{(q)} \mid \mathcal{C}\right), q \in [1, Q], u = 1, \ldots, U$
$l_u^{(R)} = \Theta^{(R)}\left(\theta_u^{(q)} \mid \mathcal{C}\right), q \in [1, Q], u = 1, \ldots, U$
Model
$\mathcal{M}^{(1)}\left(1 \mid \mathbf{l}_{\mathcal{C}}^{(1)}\right), \mathbf{l}_{\mathcal{C}}^{(1)} = \left[l_1^{(1)} \ \ldots \ l_U^{(1)}\right]^T$
$\mathcal{M}^{(R)}\left(1 \mid \mathbf{l}_{\mathcal{C}}^{(R)}\right), \mathbf{l}_{\mathcal{C}}^{(R)} = \left[l_1^{(R)} \ \ldots \ l_U^{(R)}\right]^T$
R layers

Figure 2.1: The main steps in constructing and modelling a Corpus.

The corpus $\mathcal{C}$ is the basis of Somax from which all output material will be drawn. It is constructed from one or multiple audio and/or midi files, which are segmented into short slices $\mathcal{S}$, where each segment is analyzed with respect to a number of traits θ, clustered and classified in multiple layers - each layer with respect to a single trait - and finally modelled according to a specific data model. This procedure can be seen in figure 2.1, and each step will be described in detail in the following sections.

2.2.1 Slicing

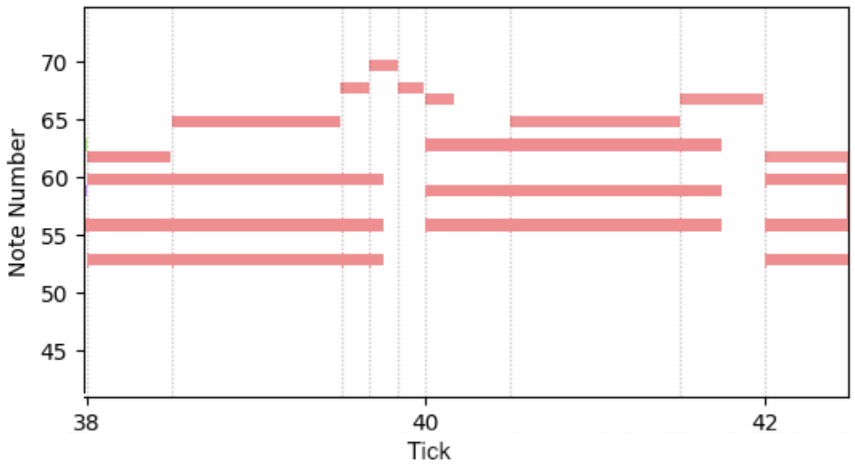


Figure 2.2: Example of slicing a short segment of a midi file, where the slices are represented by dashed vertical lines and notes represented by red horizontal bars.

The first step in constructing and modelling the corpus is to parse the midi/audio files and segment the content into slices along the time axis. The start of each slice is for midi file determined by each midi notes onset (see figure 2.2 and for audio files determined by each beat, which is estimated by either the dynamic programming algorithm [13] for audio files with a fixed tempo or using predominant local pulse estimation [15] for audio files with a dynamic tempo.

Each slice $\mathcal{S}^{(\mathcal{C})}$ is assigned an index u, an onset time $t^{(\mathcal{C})}$ (in ticks, where 1 tick correspond to one beat), a duration d (in ticks), a tempo ζ (in BPM) determined either by the midi tempo at that specific point in time or estimated from the inter-onset interval between two beats for audio files. The slice is also assigned an absolute onset time τ_u and an absolute duration δ_u, both in milliseconds, that will be relevant when determining the output position in audio files. Each slice is also analyzed with respect to a number of traits $\theta^{(q)}$, $q = 1, \ldots, Q$ which will be described in section 2.2.2. More formally, we have a corpus of length U defined as

$$\mathcal{C} = \left\{ S_1^{(\mathcal{C})}, S_2^{(\mathcal{C})}, \ldots, S_U^{(\mathcal{C})} \right\} \tag{2.1}$$

where

$$\mathcal{S}_u^{(\mathcal{C})} = \left\{ t_u^{(\mathcal{C})}, d_u, \zeta_u, \tau_u, \delta_u, \theta_u^{(1)}, \ldots \theta_u^{(Q)} \right\}, \quad u \in [1, U]. \tag{2.2}$$

2.2.2 Trait Analysis

Note that all parameters in equation 2.2 apart from $\theta^{(1)}, \ldots, \theta^{(Q)}$ are only related to timing, so the purpose of the traits θ is to store any other feature of the slice, for example pitch, velocity, harmonic content, etc. These traits will later be used for clustering, classification and modelling the corpus. Another purpose of modelling each slice in terms of traits is to make the model format-agnostic, i.e. independent of whether the corpus is based on midi or audio data. Formally, the procedure of calculating trait $\theta_u^{(q)}$ of slice $\mathcal{S}_u^{(\mathcal{C})}$ can be described as

$$\theta_u^{(q)} = \Psi^{(q)} \left(\mathcal{S}_u^{(\mathcal{C})}, \left[\mathcal{S}_{u-1}^{(\mathcal{C})}, \mathcal{S}_{u-2}^{(\mathcal{C})}, \ldots \right] \right), \quad u = 1, \ldots, U \tag{2.3}$$

where $\Psi^{(q)}$ denotes a function depending on $\mathcal{S}_u^{(\mathcal{C})}$ (and optionally previous slices) for calculating trait $\theta_u^{(q)}$. Once the trait analysis is completed, the process of constructing (but not modelling) the corpus is completed.

2.2.3 Clustering and Classification Modelling

The completed corpus $\mathcal{C}$ is modelled in R layers, where each layer $r \in [1, R]$ has its own clustering $\Theta^{(r)}$ and model $\mathcal{M}^{(r)}$. The clustering in layer r will be described as

$$\Theta^{(r)} \left(\theta^{(q)} \mid \mathcal{C} \right), \quad q \in [1, Q], \tag{2.4}$$

which in other words mean that each layer's clustering is constructed with regards to a single trait $\theta^{(q)}$ given a corpus $\mathcal{C}$. In practice, however, not all clusterings rely on $\mathcal{C}$ - there are both absolute and relative clusterings, which are described further in section 3.3.3. Once a clustering has been constructed in layer r, each slice $\mathcal{S}_u^{(\mathcal{C})}$, $u = 1, \ldots, U$ in the corpus will be classified with respect to its corresponding parameter $\theta_u^{(q)}$, i.e.

$$l_u^{(r)} = \Theta^{(r)} \left(\theta_u^{(q)} \mid \mathcal{C} \right), \quad u = 1, \ldots, U \tag{2.5}$$

where $l_u^{(r)} \in \mathbb{Z}$ denotes the label of slice $\mathcal{S}_u^{(\mathcal{C})}$ in layer r with respect to trait $\theta^{(q)}$.

Finally, a model $\mathcal{M}^{(r)}$ is constructed from the collected labels, i.e.

$$\mathcal{M}^{(r)} \left(\boldsymbol{l} \mid \boldsymbol{l}_{\mathcal{C}} \right), \quad \boldsymbol{l}_{\mathcal{C}} = \begin{bmatrix} l_1^{(r)} & \ldots & l_U^{(r)} \end{bmatrix}^T. \tag{2.6}$$

where $\mathcal{M}$ can be described as a mapping from a vector of labels $\boldsymbol{l}$ to a set of slices, i.e.

$$\mathcal{M}\colon \boldsymbol{l} \to \left\{\mathcal{S}_{u_1}^{(\mathcal{C})}, \ldots, \mathcal{S}_{u_j}^{(\mathcal{C})}\right\}, \quad u_1, \ldots, u_j \in [1, U]. \tag{2.7}$$

$\mathcal{M}$ can be described as a simplified, unweighted n-gram model which simply returns all slices that matches the given input. The process of constructing $\mathcal{M}$ is described in algorithm 2.1, where γ denotes the n-gram order, κ denotes the given input at each time step and $\mathcal{M}$ here is modelled as a map where $\mathcal{M}[\kappa]$ denotes the values at the map's index κ.

Algorithm 2.1 Constructing $\mathcal{M}$

```
for $u = \gamma$ to $U$ do
  $\kappa = [l_{u-\gamma}, \ldots, l_u]$
  if $\mathcal{M}[\kappa]$ exists then
    $\mathcal{M}[\kappa] := \mathcal{M}[\kappa] \cup \left\{\mathcal{S}_u^{(\mathcal{C})}\right\}$
  else
    $\mathcal{M}[\kappa] := \left\{\mathcal{S}_u^{(\mathcal{C})}\right\}$
  end if
end for
```

2.3 Influence

The influence process takes a continuous stream $\mathcal{K}$ of midi/audio data, segments it into slices, analyzes each slice with regards to its traits and determining where the corpus (or more specifically, at which temporal positions the models $\mathcal{M}$ constructed from the corpus) matches the incoming stream. These positions will later be used to determine the most suitable output. An overview of the influence process can be seen in figure 2.3.

One key aspect of Somax is that the influence process has a short-time memory, so that influences generated at previous time steps in the stream maintain an impact on the output for a certain amount of time. How this behaviour is simulated will be described in section 2.3.4.

2.3.1 Slicing

The slicing procedure of the influence process is almost identical to the procedure described in section 2.2.1, with the exception that it in the case of a real-time stream uses different algorithms for onset detection and beat estimation (see [7] for details). More formally, we will define the continuous stream $\mathcal{K}$

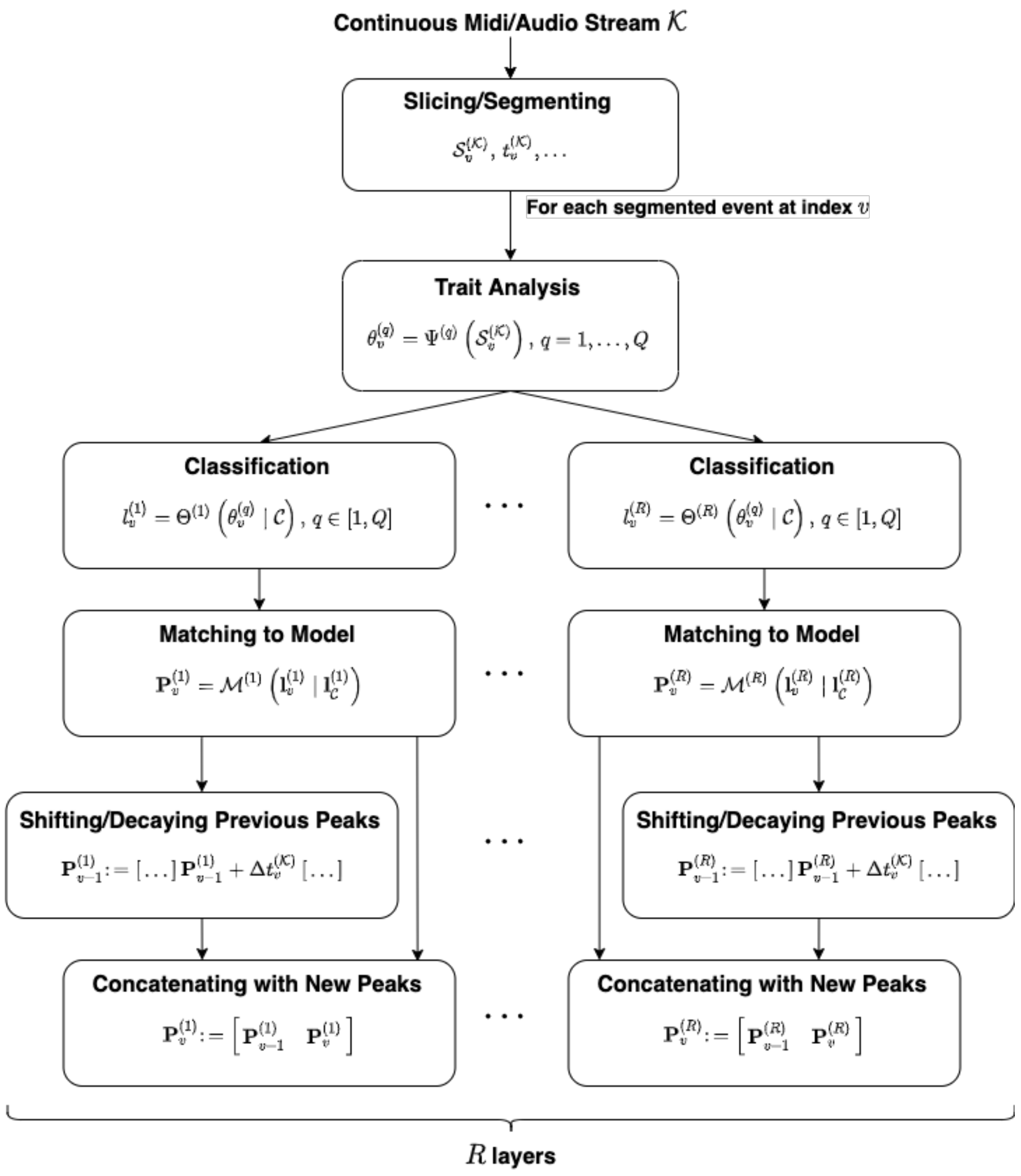

Continuous Midi/Audio Stream $\mathcal{K}$
Slicing/Segmenting
$\mathcal{S}_v^{(\mathcal{K})}, t_v^{(\mathcal{K})}, \ldots$
For each segmented event at index v
Trait Analysis
$\theta_v^{(q)} = \Psi^{(q)}\left(\mathcal{S}_v^{(\mathcal{K})}\right), q = 1, \ldots, Q$
Classification
$l_v^{(1)} = \Theta^{(1)}\left(\theta_v^{(q)} \mid \mathcal{C}\right), q \in [1, Q]$
Classification
$l_v^{(R)} = \Theta^{(R)}\left(\theta_v^{(q)} \mid \mathcal{C}\right), q \in [1, Q]$
Matching to Model
$\mathbf{P}_v^{(1)} = \mathcal{M}^{(1)}\left(\mathbf{l}_v^{(1)} \mid \mathbf{l}_{\mathcal{C}}^{(1)}\right)$
Matching to Model
$\mathbf{P}_v^{(R)} = \mathcal{M}^{(R)}\left(\mathbf{l}_v^{(R)} \mid \mathbf{l}_{\mathcal{C}}^{(R)}\right)$
Shifting/Decaying Previous Peaks
$\mathbf{P}_{v-1}^{(1)} := [\ldots]\, \mathbf{P}_{v-1}^{(1)} + \Delta t_v^{(\mathcal{K})}\, [\ldots]$
Shifting/Decaying Previous Peaks
$\mathbf{P}_{v-1}^{(R)} := [\ldots]\, \mathbf{P}_{v-1}^{(R)} + \Delta t_v^{(\mathcal{K})}\, [\ldots]$
Concatenating with New Peaks
$\mathbf{P}_v^{(1)} := \left[\mathbf{P}_{v-1}^{(1)} \quad \mathbf{P}_v^{(1)}\right]$
Concatenating with New Peaks
$\mathbf{P}_v^{(R)} := \left[\mathbf{P}_{v-1}^{(R)} \quad \mathbf{P}_v^{(R)}\right]$
R layers

Figure 2.3: The main steps in parsing a continuous midi/audio stream and influencing somax.

with an (in most cases undefined) length V as

$$\mathcal{K} = \left\{\mathcal{S}_1^{(\mathcal{K})}, \ldots, \mathcal{S}_V^{(\mathcal{K})}\right\} \tag{2.8}$$

where

$$\mathcal{S}_v^{(\mathcal{K})} = \left\{t_v^{(\mathcal{K})}, d_v, \zeta_v, \theta_v^{(1)}, \ldots \theta_v^{(Q)}\right\}, \quad v \in [1, V]. \tag{2.9}$$

While the differences between equations 2.2 and 2.9 seem to be that of replacing one index for another, there is one key difference that will be important in later sections: the difference between corpus time, $t^{(\mathcal{C})}$, and continuous time $t^{(\mathcal{K})}$. The corpus time $t_u^{(\mathcal{C})}$, $u = 1, \ldots, U$ denotes the position of slice $\mathcal{S}_u^{(\mathcal{C})}$ in the corpus which will always be fixed. The influence time $t_v^{(\mathcal{K})}$ denotes the time of the current state v in the process of generating and is closely related to scheduling, which will be described in section 3.4.1.

2.3.2 Trait Analysis, Classification and Matching to Model

Trait analysis and classification of a slice $\mathcal{S}_v^{(\mathcal{K})}$ is identical to the procedure described in section 2.2, more specifically to equation 2.3 and 2.5 respectively, with the only difference that it occurs in continuous time as soon as a slice $\mathcal{S}_v^{(\mathcal{K})}$ has been segmented. The system uses the R layers that were used model the corpus, and in each layer the clustering created from the corpus is used for classification. In other words, at time step $t_v^{(\mathcal{K})}$ corresponding to influence slice $\mathcal{S}_v^{(\mathcal{K})}$ with traits $\theta_v^{(1)}, \ldots, \theta_v^{(Q)}$, we get one label $l_v^{(r)}$ per layer $r = 1, \ldots, R$ so that

$$l_v^{(r)} = \Theta^{(r)} \left(\theta_v^{(q)} \mid \mathcal{C} \right) \tag{2.10}$$

where $\theta_v^{(q)}$ denotes the single trait used in layer r at the time step corresponding to index v.

2.3.3 Matching Slice to Model

In each layer $r = 1, \ldots R$, at time step $t_v^{(\mathcal{K})}$ corresponding to influence slice $\mathcal{S}_v^{(\mathcal{K})}$, a vector will be constructed from the previous $k \in \mathbb{Z}_*$ labels, i.e.

$$\boldsymbol{l}_v^{(r)} = \begin{bmatrix} l_{v-k}^{(r)} & \ldots & l_{v-1}^{(r)} & l_v^{(r)} \end{bmatrix}, \tag{2.11}$$

and in accordance with the definition in equation 2.7 generate a set of slices $\Sigma_v^{(r)} = \left\{ \mathcal{S}_{u_1}^{(\mathcal{C})}, \ldots, \mathcal{S}_{u_j}^{(\mathcal{C})} \right\}, \quad u_1, \ldots, u_j \in [1, U]$. From $\Sigma_v^{(r)}$, a matrix of peaks $\boldsymbol{P}_v^{(r)} \in \mathbb{R}^{2 \times j}$ is constructed so that

$$\boldsymbol{P}_v^{(r)} = \begin{bmatrix} \boldsymbol{p}_{u_1} & \ldots & \boldsymbol{p}_{u_j} \end{bmatrix} \tag{2.12}$$

where

$$\boldsymbol{p}_{u_m} = \begin{bmatrix} t_{u_m}^{(\mathcal{C})} \\ y_{u_m} \end{bmatrix}, \qquad m = 1, \ldots, j \tag{2.13}$$

where y_{u_m} is a score (height) designated to each peak by the model and $t_{u_m}^{(\mathcal{C})}$ is the temporal position of the corresponding slice's onset.

2.3.4 Peaks

The final step in the influence process is to in each layer $r = 1, \dots R$ add the peaks $\boldsymbol{P}_v^{(r)}$ generated at time step $t_v^{(\mathcal{K})}$ corresponding to index v to the previous set of peaks $\boldsymbol{P}_{v-1}^{(r)}$ generated at the previous time step $t_{v-1}^{(\mathcal{K})}$. The purpose of this behaviour is, as was described in section 2.3, to simulate a short-term memory, where previous influences maintain a degree of impact on the output.

To achieve this, the positions of the previous peaks are shifted (in corpus time $t^{(\mathcal{C})}$) corresponding to the influence time $t^{(\mathcal{K})}$ elapsed since the previous influence, and the scores are decayed exponentially corresponding to the same interval scaled by a factor $\tau \in \mathbb{R}$, i.e.

$$\boldsymbol{P}_{v-1}^{(r)} := \begin{bmatrix} 1 & 0 \\ 0 & \exp\left(\Delta t_v^{(\mathcal{K})} / \tau\right) \end{bmatrix} \boldsymbol{P}_{v-1}^{(r)} + \Delta t_v^{(\mathcal{K})} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 0 & 0 & \dots & 0 \end{bmatrix} \tag{2.14}$$

where

$$\Delta t_v^{(\mathcal{K})} = t_v^{(\mathcal{K})} - t_{v-1}^{(\mathcal{K})}. \tag{2.15}$$

Finally, the peak matrix $\boldsymbol{P}_v^{(r)}$ is updated by concatenating it with the shifted and decayed peaks from the previous time step, i.e.

$$\boldsymbol{P}_v^{(r)} := \begin{bmatrix} \boldsymbol{P}_{v-1}^{(r)} & \boldsymbol{P}_v^{(r)} \end{bmatrix}. \tag{2.16}$$

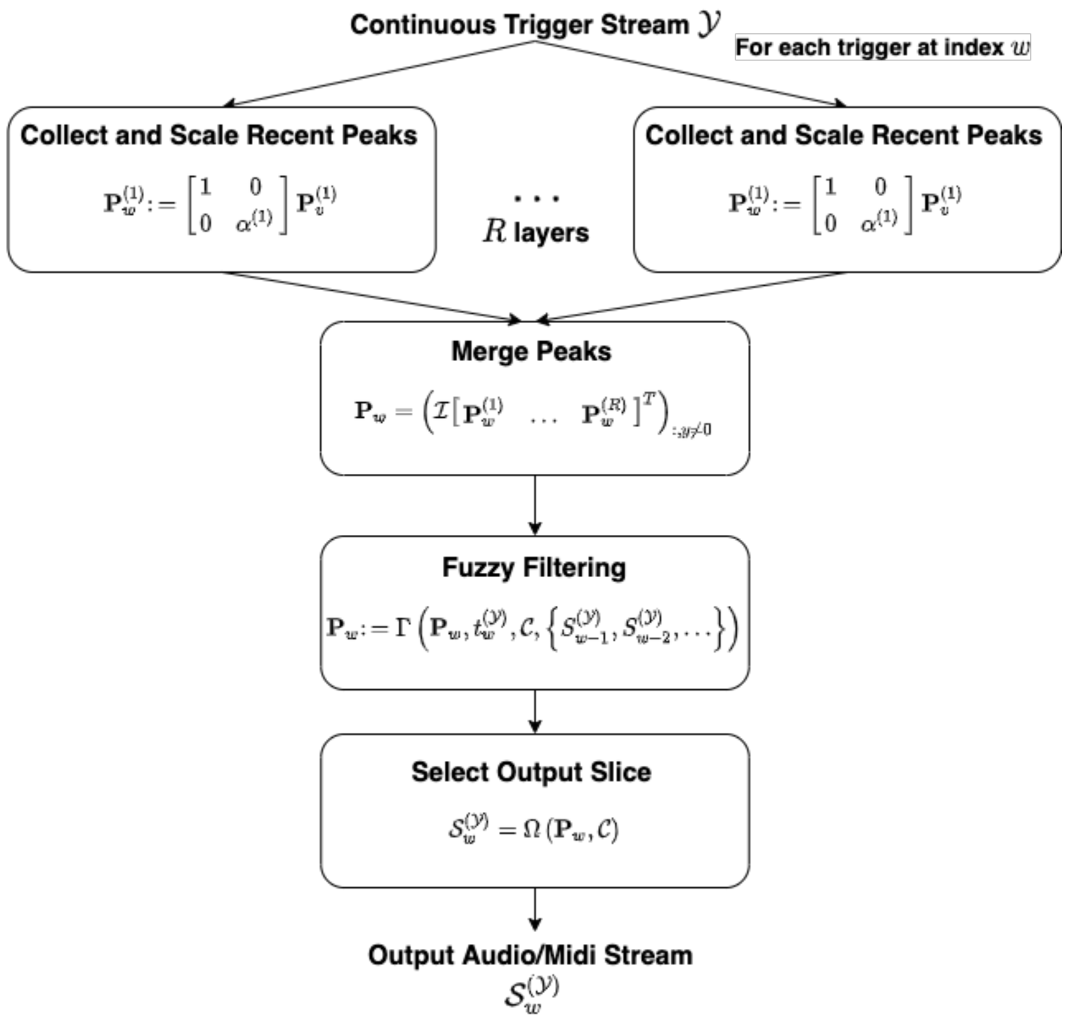
2.4 Generate

So far, both section 2.2 and 2.3 have described a number of procedures for updating the internal state of the system without generating any output. This section will describe the steps taken to generate output from a continuous trigger stream $\mathcal{Y}$ of (an often undefined) length W based on the state set by recent influences, for which the procedure is outlined in figure 2.4

2.4.1 Triggers

While the concept of a continuous audio/midi stream is rather self-explanatory, the concept of a continuous trigger stream might need some explanation. Similarly to a continuous midi stream, a trigger stream is a stream of discrete events occurring at a time $t_w^{(\mathcal{Y})}$, but without any external information. In other words, the behaviour is that of a signal telling the system when to generate output.

In practice, the trigger stream is often correlated to the influencing audio/midi stream, either by midi note onsets, beat onsets or pitch detection onsets (for more details, see [7]) depending on the mode of the system, but may also be



Continuous Trigger Stream $\mathcal{Y}$
For each trigger at index w
Collect and Scale Recent Peaks
$\mathbf{P}_w^{(1)} := \begin{bmatrix} 1 & 0 \\ 0 & \alpha^{(1)} \end{bmatrix} \mathbf{P}_v^{(1)}$
. . . R layers
Collect and Scale Recent Peaks
$\mathbf{P}_w^{(1)} := \begin{bmatrix} 1 & 0 \\ 0 & \alpha^{(1)} \end{bmatrix} \mathbf{P}_v^{(1)}$
Merge Peaks
$\mathbf{P}_w = \left(\mathcal{I}\left[\mathbf{P}_w^{(1)} \quad \dots \quad \mathbf{P}_w^{(R)}\right]^T\right)_{:,y \neq 0}$
Fuzzy Filtering
$\mathbf{P}_w := \Gamma\left(\mathbf{P}_w, t_w^{(\mathcal{Y})}, \mathcal{C}, \left\{\mathcal{S}_{w-1}^{(\mathcal{Y})}, \mathcal{S}_{w-2}^{(\mathcal{Y})}, \dots\right\}\right)$
Select Output Slice
$\mathcal{S}_w^{(\mathcal{Y})} = \Omega\left(\mathbf{P}_w, \mathcal{C}\right)$
Output Audio/Midi Stream
$\mathcal{S}_w^{(\mathcal{Y})}$

Figure 2.4: The main steps in generating output from a continuous trigger stream.

completely decorrelated to incoming influences. This behaviour depends on the mode of the system, which is discussed in section 3.4.1. In all cases, influencing and generating operate interleaved along the same time axis, so the behaviour of $t_w^{(\mathcal{Y})}$ is similar to the behaviour of $t_v^{(\mathcal{K})}$ (in chapter 3 we will see that they both operate on the scheduler's time).

2.4.2 Collecting, Scaling and Merging Peaks

When a trigger is received, the first step is to collect and scale the peaks from all layers $r = 1, \dots, R$. Each layer will have a designated weight $\alpha^{(r)}$ to scale the scores of each layer, i.e.

$$\boldsymbol{P}_w^{(r)} = \begin{bmatrix} 1 & 0 \\ 0 & \alpha^{(r)} \end{bmatrix} \boldsymbol{P}_v^{(r)}, \quad r = 1, \ldots, R \tag{2.17}$$

where $\boldsymbol{P}_v^{(r)}$ here denotes the peak matrix generated by the most recent influence step v.

The scaled peaks $\boldsymbol{P}_w^{(r)}$ from all layers $r = 1, \ldots R$ are then gathered into a single matrix where any peaks that occur sufficiently close to each other in corpus time $t^{(\mathcal{C})}$ are summed, i.e.

$$\boldsymbol{p}_i = \begin{bmatrix} t_i^{(\mathcal{C})} \\ y_i + y_j \end{bmatrix} \quad \text{if} \quad \left| t_i^{(\mathcal{C})} - t_j^{(\mathcal{C})} \right| < \varepsilon \quad \forall \boldsymbol{p}_i, \boldsymbol{p}_j \in \boldsymbol{P}_w, \tag{2.18}$$

for some interval ε, where

$$\boldsymbol{P}_w = \begin{bmatrix} \boldsymbol{P}_w^{(1)} & \ldots & \boldsymbol{P}_w^{(R)} \end{bmatrix}. \tag{2.19}$$

As the number of peaks n_w at the time at index w after v influences has a theoretical worst case of $n_w = \mathcal{O}(vRU)$, some optimization is required to solve equation 2.18, which by default has a time complexity of $\mathcal{O}\left(n_w^2\right)$. In practice, this is solved in linear time by multiplying the transposed peaks with a binary interpolation matrix $\mathcal{I} \in \mathbb{Z}_{[0,1]}^{m \times n}$ with $m = \lfloor 1/\varepsilon \rfloor$ rows and n_w columns, and selecting all non-zero columns from the transposed output, i.e.

$$\boldsymbol{\Pi}_w = \mathcal{I} \boldsymbol{P}_w^T \tag{2.20}$$

$$\boldsymbol{P}_w := \left(\boldsymbol{\Pi}_w^T \right)_{:, y \neq 0} \tag{2.21}$$

where

$$(\mathcal{I})_{i,j} = \begin{cases} 1 & \text{if} \quad \left\lfloor \frac{t_j^{(\mathcal{C})}}{\varepsilon t_U^{(\mathcal{C})}} \right\rfloor = i \\ 0 & \text{otherwise} \end{cases} \tag{2.22}$$

resulting in a single merged peak matrix $\boldsymbol{P}_w \in \mathbb{R}^{2 \times \hat{n}_w}$, $\hat{n}_w \leq n_w$ for the time step at index w.

2.4.3 Scaling Peaks Again - Fuzzy Filtering

In section 2.4.2, all peaks in each layer were scaled uniformly by a weight. Once merged, the peaks are scaled again with respect to a set of more elaborate algorithms $\left\{ \Gamma^{(1)}, \ldots, \Gamma^{(J)} \right\}$, based on the time of influence, each peak's corresponding corpus data and/or previously output slices, i.e.

$$\boldsymbol{P}_w := \Gamma^{(j)} \left(\boldsymbol{P}_w, t_w^{(\mathcal{Y})}, \mathcal{C}, \left\{ \mathcal{S}_{w-1}^{(\mathcal{Y})}, \mathcal{S}_{w-2}^{(\mathcal{Y})}, \ldots \right\} \right) \quad \forall j = 1, \ldots, J \tag{2.23}$$

where

$$\Gamma^{(j)}\colon \boldsymbol{P}_x \to \boldsymbol{P}_z, \quad \boldsymbol{P}_z \in \mathbb{R}^{m_x \times n_x}. \tag{2.24}$$

This behaviour can be seen as a sort of fuzzy filtering, as opposed to the binary matching provided by each classification/model-layer, to further emphasize or de-emphasize peaks with regards to parameters that may be difficult to classify in a meaningful manner.

2.4.4 Generating Output

Finally, the output for the time step at index w is selected from $\mathcal{C}$ as the slice closest to the peak $\boldsymbol{p}_i \in \boldsymbol{P}_w$ with the highest score y_i, i.e.

$$\hat{\boldsymbol{p}}_w = \begin{bmatrix} \hat{t}^{(\mathcal{C})} & \hat{y} \end{bmatrix}^T = \left\{ \boldsymbol{p}_j \mid \forall \boldsymbol{p}_i \in \boldsymbol{P}_w \colon y_j \geq y_i \right\}, \tag{2.25a}$$

$$\mathcal{S}_w^{(\mathcal{Y})} = \min_{\mathcal{S}_u^{(\mathcal{C})} \in \mathcal{C}} \left| \hat{t}^{(\mathcal{C})} - t_u^{(\mathcal{C})} \right|. \tag{2.25b}$$

If multiple vectors $\boldsymbol{p}_j$ fulfil the condition in equation 2.25a, a single one will be selected randomly. If on the opposite no vectors fulfil this condition, i.e. if the peak matrix $\boldsymbol{P}_w$ is empty, the slice $\mathcal{S}_{u+1}^{(\mathcal{C})}$ following the previously output slice will be used for output, i.e. given

$$\mathcal{S}_{w-1}^{(\mathcal{Y})} = \mathcal{S}_u^{(\mathcal{C})} \tag{2.26}$$

for some $u \in [1, U]$, we get

$$\mathcal{S}_w^{(\mathcal{Y})} = \mathcal{S}_{u+1}^{(\mathcal{C})}. \tag{2.27}$$

Chapter 3

Implementation

This chapter presents the realization of the theoretical framework as described in chapter 2. While section 3.3 to some extent reiterates certain aspects from [7], most of the content described here is either significantly rewritten or entirely new. In addition to the modules described in this chapter, there's also a front-end for the real-time human-machine improvisation mode of the system, written in Max/MSP [11], which is thoroughly described in [7] but will not be discussed in this chapter since the changes presented here do not provide any significant architectural changes to the front-end.

3.1 Overview

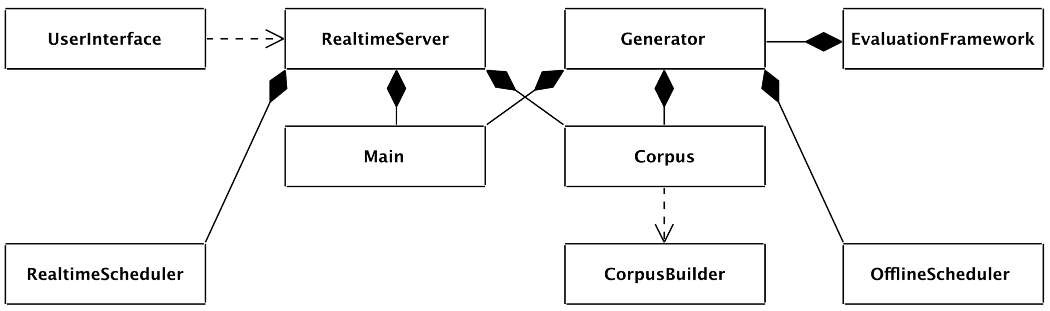


Figure 3.1: Module diagram over the main modules in the system and the relationship between them.

Figure 3.1 shows the different modules of the system and how they relate to each other. There are two main branches in this figure, one stemming from the `RealtimeServer` module, corresponding to the real-time (i.e. human-machine improvisation) framework, and one stemming from the `Generator` module, corresponding to the offline (i.e. composition-oriented) framework. Both of them share the `Corpus` module (and its related `CorpusBuilder`), which handles the construction of corpora and will be described in section 3.2, and the `Main` module, which handles all the internal (runtime) logic of the system and will be described in section 3.3. The `RealtimeServer` and its related `UserInterface` module will not be specifically described in this chapter, as they were thoroughly described in [7]. The logic of the `RealtimeScheduler` module, which handles the runtime scheduling of events over time, has however been significantly updated and will be presented in section 3.4.2.

In the branch stemming from the `Generator` module, the `Generator` itself will be described in section 3.4.4 along with its `OfflineScheduler`, which handles scheduling as an offline process and will be described in section 3.4.3. Finally, the `EvaluationFramework`, which is used to gather statistics about the generation process and its output, is described in chapter 4. All the code described here is written in Python 3 and is available at `https://github.com/DYCI2/Somax2`[1].

3.2 Corpus Builder

The purpose of the `CorpusBuilder` module is to construct the core of the system, the `Corpus`, from midi and/or audio files. It's an offline (as opposed to real-time) system that can be accessed both through a command-line build script as well as through the real-time user interface.

Another purpose is to achieve the format-agnostic behaviour of the runtime system as was described in section 2.2, i.e. to ensure that there are no differences in how the runtime system handles corpora built from audio files in comparison to corpora built on midi files. However, as we will see in section 3.2.2, this behaviour isn't fully achieved yet. For this reason, as well as due to the fact that the evaluation only uses midi, most of the behaviour described in this chapter will focus mainly on the midi implementation.

When building a corpus from midi files, the first step is to create a `NoteMatrix` class, which essentially is a matrix where each row correspond to a single midi note where each column correspond to pitch, velocity, channel, relative onset time (measured in ticks/beats since start of file), absolute onset time (measured in milliseconds since start of time), relative duration (measured in ticks), absolute duration (measured in milliseconds) and tempo. The rows are sorted by their relative onsets, i.e. their occurrences in time. This format is similar to the format used in [12] with the addition of tempo. The `NoteMatrix` class also stores any midi control changes and meta messages to allow reconstruction of the original midi file.

[1]As of May 27, 2020, all code related to this thesis can be found in the `evaluation` branch at `https://github.com/DYCI2/Somax2/tree/evaluation`, but will likely be merged into the `master` branch at a later point in time.

PITCH	VELOCITY	CHANNEL	ONSET (tick)	ONSET (msec)	DURATION (tick)	DURATION (msec)	TEMPO

Figure 3.2: Columns of the `MidiMatrix` class.

3.2.1 Slicing

Once the `NoteMatrix` is constructed, the next step is to determine the temporal boundaries for each slice, i.e. slicing. This is done differently for audio and midi as mentioned in section 2.2 - per beat for audio and per note for midi. In the case of midi file(s), to achieve the behaviour outlined in section 2.2.1 and specifically figure 2.2, algorithm 3.1 is used. In this algorithm, the `NoteMatrix` is denoted $\mathcal{N}$, where $\mathcal{N}_i$ denotes the note at row i and `.absolute_onset`, `.relative_onset`, etc. denote the columns corresponding to indices in figure 3.2. All other notation in the figure apart from ε is outlined in section 2.2.1. The algorithm is iterating over all midi notes and creating a new slice for any note whose onset is more than ε millseconds from the previous note, otherwise appending the note to the current slice. This ε is very important, as it will ensure that notes occurring sufficiently close (for example appoggiaturas and other articulation) are treated as part of the same slice, which both will be important for analysis later as well as maintaining the original rhythmic feel of the file without quantization. Another important aspect of the algorithm is line 14, which results in that any note in the previous slice that overlaps into the new slice will be added as well, and thus have an onset $\mathcal{N}_i$`.relative_onset` that may be earlier than the slice onset d_u. This will play an important role when scheduling slices, which is be described in section 3.4.1. The result of the slicing procedure is the `Corpus` class with all parameters set apart from its `Traits`.

3.2.2 Trait Analysis

In parallel with the slicing, two lowpass-filtered pseudo-spectrogram are computed from the `NoteMatrix` in the `Spectrogram` class, one for the foreground (or melodic) channels and one for the background (or harmonic) channels, which are both specified by the user. If no channels are specified, the foreground and background spectrogram will be computed from the entire set of channels and thus be identical. The procedure for computing the pseudo-spectrogram is the same as in [7], with the addition of the filter being interchangeable to allow different types of filtering, as well as no filtering at all. From these spectrogram, two pseudo-chromagram are computed in the `Chromagram` class, again one for the foreground channels and one for the background channels. For audio, these are computed directly from the audio data.

Once the `Spectrogram` and the `Chromagram` have been computed and the

Algorithm 3.1 Slicing a `NoteMatrix` $\mathcal{N}$ into a `Corpus` $\mathcal{C}$

```
 1: $u = 0$
 2: $\mathcal{C} = \{\}$
 3: $\theta_u^{(\mathcal{N})} = \mathcal{N}_0$
 4: $t_u^{(\mathcal{C})} = \mathcal{N}_0$.relative_onset
 5: $\zeta_u = \mathcal{N}_0$.tempo
 6: $\tau_u = \mathcal{N}_0$.absolute_onset
 7: for $i = 1$ to $|\mathcal{N}| - 1$ do
 8:     if $\mathcal{N}_i$.absolute_onset $> \tau_u + \varepsilon$ then
 9:         $d_u = \mathcal{N}_i$.relative_onset $- t_u^{(\mathcal{C})}$
10:         $\delta_u = \mathcal{N}_i$.absolute_onset $- \tau_u$
11:         $\mathcal{S}_u^{(\mathcal{C})} = \left\{ t_u^{(\mathcal{C})}, d_u, \zeta_u, \tau_u, \delta_u, \theta_u^{(\mathcal{N})} \right\}$
12:         $\mathcal{C} = \mathcal{C} \cup \left\{ \mathcal{S}_u^{(\mathcal{C})} \right\}$
13:         $u = u + 1$
14:         $\theta_u^{(\mathcal{N})} = \left\{ n \mid n \in \theta_{u-1}^{(\mathcal{N})}, n$.relative_onset $+ n$.relative_duration $> d_{u-1} \right\}$
15:         $t_u^{(\mathcal{C})} = \mathcal{N}_i$.relative_onset
16:         $\zeta_u = \mathcal{N}_i$.tempo
17:         $\tau_u = \mathcal{N}_i$.absoulute_onset
18:     else
19:         $\theta_u^{(\mathcal{N})} = \theta_u^{(\mathcal{N})} \cup \{\mathcal{N}_i\}$
20:     end if
21: end for
```

slicing procedure to create the `Corpus` is completed, the trait analysis begins. The trait analysis is dynamic, which means that it will import any class in the code base extending the `AbstractTrait` stereotype (see figure 3.3) and call the `analyze` function on each slice in the corpus.

```python
class AbstractTrait(ABC):
    @classmethod
    @abstractmethod
    def analyze(cls, event: CorpusEvent, audio_data: np.ndarray,
                fg_spectrogram: Spectrogram, bg_spectrogram: Spectrogram,
                fg_chromagram: Chromagram, bg_chromagram: Chromagram,
                **kwargs):
        pass
```

Figure 3.3: The `AbstractTrait` stereotype, which is used to analyze all traits.

In the current state of the system, the following traits have been implemented:

Notes $\theta^{(\mathcal{N})}$ The midi notes contained in the current slice, as defined in algorithm 3.1. Note that at the moment, no corresponding values exist for audio data, which in other words means that the model isn't truly format-agnostic yet. Ideally, this could be solved by estimating these values with a polyphonic f0-estimator, for example [31].

Top Note $\theta^{(P_T)} \in \mathbb{Z}_{[0,127]}$ This value is simply the note number of the highest note in each slice, i.e.

$$\theta_u^{(P_T)} = \left\{ \mathcal{N}_i.\texttt{pitch} \mid \mathcal{N}_i \in \theta_u^{(\mathcal{N})} \wedge \mathcal{N}_j \in \theta_u^{(\mathcal{N})} : \mathcal{N}_i.\texttt{pitch} \geq \mathcal{N}_j.\texttt{pitch} \right\}. \tag{3.1}$$

Onset Chroma $\boldsymbol{\theta}^{(C)} \in \mathbb{R}^{12\times 2}$, which is the column in the `Chromagram` class (both foreground and background) at the index corresponding to the absolute onset of the slice, i.e.

$$\boldsymbol{\theta}_u^{(C)} = \begin{bmatrix} \boldsymbol{C}^{(\text{fg})}_{:\,,\tau_u} & \boldsymbol{C}^{(\text{bg})}_{:\,,\tau_u} \end{bmatrix} \tag{3.2}$$

where $\boldsymbol{C}$ denotes the `Chromagram` class as constructed in the previous step.

3.2.3 The Corpus

With the trait analysis completed, the `Corpus` class is finalized. Once again, it's important to emphasize the two purposes of the `Corpus` class: to (a) abstract the data of the audio/midi file(s) into high-level data that can be used for classification and (b) to create a format-agnostic object. The latter means that from this point, all raw midi and audio data as well as the spectrogram and chromagram will be thrown away. For midi data, this isn't a problem, as the `Notes` trait $\theta^{(\mathcal{N})}$ contains all the midi data - in fact, the `Corpus` and `NoteMatrix` are interchangeable, thus allowing re-export of the `Corpus` back to midi. For audio data, only a reference to the original file will be kept, thus the raw audio data corresponding each slice $\mathcal{S}_u^{(\mathcal{C})}$ can be reproduced by its absolute onset τ_u and its absolute duration δ_u. This compact data format also allows exporting the corpus as a JSON-file for quickly reloading previously built corpora.

Finally, while the `CorpusBuilder` module is the main way to construct a `Corpus`, it's not the only way. As we will see in section 3.3, a `Corpus` can be constructed from another `Corpus` during a real-time performance, and as we will see in section 3.4, it can also be generated from other corpora offline.

3.3 Runtime Architecture

The runtime architecture handles all the influencing and output generation, it's basically the core of the system. While most of it already has been explained in [7], some key aspects have been left out due to the article's condensed format, as well as some critical changes made since the article was written. For this reason, the design of the architecture and the relation between the algorithms described in chapter 2 and the components of the system will be reiterated in the following section.

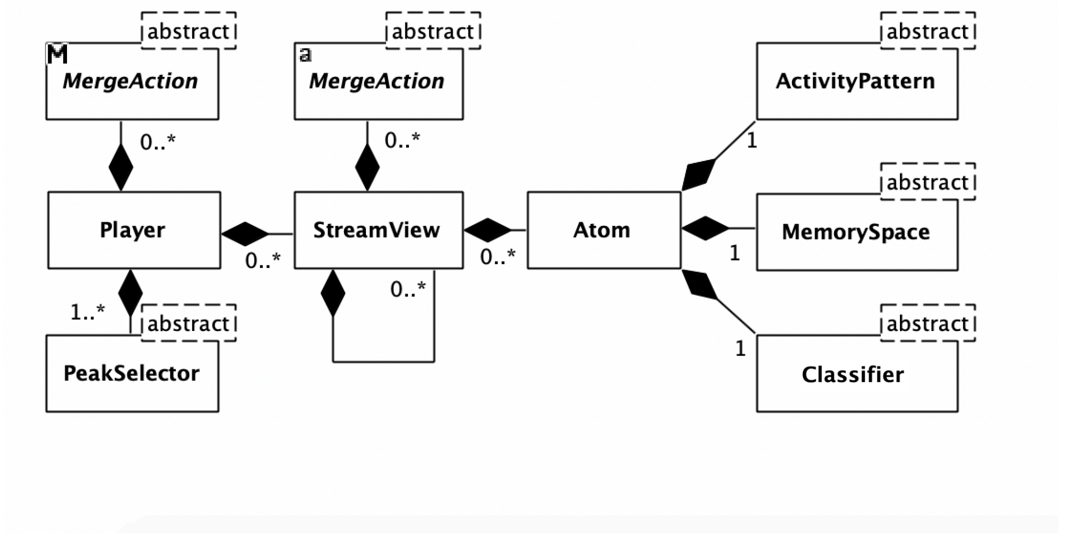


Figure 3.4: Simplified class diagram over the main components of the runtime architecture.

3.3.1 The Runtime System's Components

The architecture of the system can be seen in figure 3.4. At the root of the system is the `Player` class, through which all interaction with the system occurs. At the opposite end, at what could be considered the core of the system, is the `Atom`, where each `Atom` corresponds to one of the $r = 1, \ldots, R$ layers described in chapter 2. The `Atom` contains one `Classifier` instance, corresponding to a classifier $\Theta^{(r)}$, one `MemorySpace` instance, corresponding to a model $\mathcal{M}^{(r)}$ and one `ActivityPattern` instance, which handles storing, shifting, decaying and concatenation of peaks $\boldsymbol{P}^{(r)}$ as described in section 2.3.4.

Inbetween the `Player` and the `Atoms` is the `StreamView` class, an element that wasn't mentioned in chapter 2. Each `Player` contains any number of `StreamViews`, which in turn is a recursive structure containing any number of `StreamViews` and any number of `Atoms`, effectively forming a tree structure where the `Player` correspond to the root of the tree, each `StreamView` correspond to a branch and each `Atom` correspond to a leaf of the tree. Each `Atom` and `StreamView` is assigned a (by the user controlled) weight $\alpha^{(r)}$ and at each branch in the tree, merging (as described in section 2.4.2) and user-controlled fuzzy filtering Γ (as described in section 2.4.3) are performed by the `MergeAction` class. Finally, once all peaks have been merged up through the tree to the `Player` class, a final set of (user-defined) `MergeActions` are performed, and from that set of peaks, the output is selected as described in section 2.4.4 by the `PeakSelector` class.

There are three main states in the runtime architecture, each roughly corresponding to the three main sections of chapter 2: initialization, influence and output. In the initialization state, which is only performed once, the user defines the runtime architecture, i.e. the tree structure, which `Classifiers` (and `MemorySpaces` and `ActivityPatterns`) to use in each layer, as well as the `MergeActions` to use at each branch. This is also where the `Corpus` is built (or loaded from a previously built `Corpus`), clustered, classified and modelled in each of the `Atoms`. Note that while this step is mandatory for the initialization, the clustering, classification and modelling can be recomputed with different parameters in each of the `Atoms` while the system is running.

The two other states, influence and output, takes turns continuously while the system is running and operate in opposite directions, where the influence state flows from the `Player` through the architecture, computing each of the steps defined in section 2.3, ending in each of the `ActivityPatterns` where the generated peaks are stored, and the output state gathers all the generated peaks in each of the `ActivityPatterns` and merge them towards the `Player`, generating the output according to the steps in section 2.4.

3.3.2 Modularity and Dynamicity

Similarly to the `AbstractTrait` stereotype defined in section 3.2.2, each of the components labelled with the `abstract` keyword in figure 3.4 can be substituted using each class' corresponding stereotype. For the `MemorySpace`, `ActivityPattern` and `PeakSelector` classes, this behaviour is simply for future use and their stereotypes will not be discussed in detail. In this work, the stereotypes for the `Classifier` class and the `MergeAction` class are of greater importance, as a number of each have been implemented and will be described in section 3.3.3 and section 3.3.4 respectively.

Finally, among the novelties added to the system are the `Parametric` and `Parameter` classes (these are not displayed in figure 3.4, but all of the classes in the figure extends the `Parametric` class). From a software engineering perspec-

```
class Parameter(HasParameterDict):
    def __init__(self, default_value: Ranged, min_value: Ranged,
                 max_value: Ranged, type_str: str,
                 description: str, setter: Optional[Callable]):
        # ...
```

Figure 3.5: The constructor for the `Parameter` class.

tive, addressing a parameter somewhere inside dynamic tree can be difficult, especially when communicating with an external client over a string-based protocol. The purpose of the `Parametric` class is to expose any `Parameter` to the user interface. In practice, this means that any class that extends the `Parametric` class can declare any of its user-controlled parameters as a `Parameter` class, and it will be immediately available in the user interface with a name, type, range, description and optional setter function. The constructor for the `Parameter class` can be seen in figure 3.5.

3.3.3 Clustering and Classification: the `Classifier` class

Clustering and classification is handled by the `Classifier` class. Each classifier is implemented by extending the `AbstractClassifier` stereotype shown in figure 3.6, implementing the functions `cluster`, corresponding to equation 2.4, `classify_corpus`, corresponding to equation 2.5 and `classify_influence`, corresponding to equation 2.10. In practice, not all `Classifiers` rely on the `Corpus` for clustering - in fact some `Classifiers` don't implement the `cluster` function at all.

In the current state of the system, these classifiers have been implemented:

Top Note Classifier $\Theta^{(P_T)}$, which, as the trait $\theta^{(P_T)}$ already is a discrete parameter, simply is an identity classifier, defined so that

$$l = \Theta^{(P_T)}\left(\theta^{(P_T)} \mid \mathcal{C}\right) = \theta^{(P_T)}, \quad l \in \mathbb{Z}_{[0,127]}, \tag{3.3}$$

in other words, a `Classifier` without any clustering and thus independent of $\mathcal{C}$.

Pitch Class Classifier $\Theta^{(P_{12})}$, defined so that

$$l = \Theta^{(P_{12})}\left(\theta^{(P_T)} \mid \mathcal{C}\right) = \theta^{(P_T)} \bmod 12, \quad l \in \mathbb{Z}_{[0,11]}. \tag{3.4}$$

Again, a `Classifier` without any clustering and thus independent of $\mathcal{C}$.

```
class AbstractClassifier(ABC):
    @abstractmethod
    def cluster(self, corpus: Corpus) -> None:
        pass

    @abstractmethod
    def classify_corpus(self, corpus: Corpus) -> List[AbstractLabel]:
        pass

    @abstractmethod
    def classify_influence(self, influence: AbstractInfluence) -> AbstractLabel:
        pass
```

Figure 3.6: Stereotype for implementing a `Classifier`.

SOM Chroma Classifier $\Theta^{(C_{\text{SOM}})}$ A classifier of onset chroma vectors based on the original Somax implementation as defined in [6]. The clustering was computed using a self-organizing map on a matrix $\boldsymbol{X}$ of 3600 chroma vectors, i.e. $\boldsymbol{X} \in \mathbb{R}^{3600\times 12}$, returning a set of labels $\boldsymbol{l}^{(\boldsymbol{X})} \in \mathbb{Z}^{3600}_{[0,121]}$. The origin of these 3600 chroma vectors, as well as the exact parameters for the self-organizing map has unfortunately been lost, but this classifier will serve as an important base case when comparing different chroma classifiers.

As the self-organizing map itself can't be used for classifying corpora or influences, this classifier will simply select the label of the row in $\boldsymbol{X}$ minimizing the distance to the chroma vector $\boldsymbol{\theta}^{(C)} \in \mathbb{R}^{12}$ to classify, i.e.

$$l = \Theta^{(C_{\text{SOM}})}\left(\theta^{(C)} \mid \mathcal{C}\right) = l_i^{(\boldsymbol{X})} \tag{3.5}$$

where

$$i = \underset{\boldsymbol{x}\in\boldsymbol{X}}{\operatorname{argmin}} \|\boldsymbol{x} - \boldsymbol{\theta}^{(C)}\|_2 \tag{3.6}$$

and $l_i^{(\boldsymbol{X})}$ denotes the label in $\boldsymbol{l}^{(\boldsymbol{X})}$ at index i.

Absolute GMM Chroma Classifier $\Theta^{(C_{\text{AGMM}})}$ A classifier of onset chroma vectors based on a Gaussian Mixture Model clustering, as described in appendix A. The classifier uses the same matrix $\boldsymbol{X} \in \mathbb{R}^{3600\times 12}$ as the SOM Chroma Classifier for clustering, but with a user-defined number of clusters K. The initial clustering $\Theta_{i=0}^{(C_{|\text{GMM}|})}$ is computed using K-means [4] with K clusters and the EM-algorithm iterated for (user-defined) I iterations. The classification is defined as in equation A.17, i.e. (with adapted notation)

$$l = \Theta_I^{(\mathcal{C}_{\text{AGMM}})}\left(\boldsymbol{\theta}^{(\mathcal{C})} \mid \mathcal{C}\right) = \underset{k \in 1 \ldots K}{\operatorname{argmax}}\, p\left(C_I^{(k)} \mid \boldsymbol{\theta}^{(\mathcal{C})}\right) \tag{3.7}$$

where $C_I^{(k)}$ denotes cluster k after I iterations. Compared to the SOM Chroma Classifier, the main benefit with this is the variable number of clusters. Having a variable number of matches means that the precision of the classifier can be adjusted (where a higher number of clusters would mean a higher precision) at the cost of number of matches in the corpus (where a high number of clusters in most cases will result in less matches). This means that each performance can be parametrically tuned with regards to how well the corpus matches the input.

Relative GMM Chroma Classifier $\Theta^{(\mathcal{C}_{\text{RGMM}})}$ This classifier is identical to the Absolute GMM Chroma Classifier, but uses the data in corpus $\mathcal{C}$ to construct the matrix $\boldsymbol{X}$, and thus uses $\mathcal{C}$ for both clustering and initial classification. In other words, we have

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{\theta}_1^{(\mathcal{C})} \\ \vdots \\ \boldsymbol{\theta}_U^{(\mathcal{C})} \end{bmatrix}, \tag{3.8}$$

$\boldsymbol{X} \in \mathbb{R}^{U \times 12}$. This is the first classifier where the clustering is input dependent. Compared to the Absolute GMM Chroma Classifier, having a clustering dependent on the corpus can potentially result in very poor matches if the corpus is harmonically dissimilar to the input, as the classification algorithm will simply select the match with the highest probability (which then may be very low). But on the other hand, if the corpus and the input are harmonically similar, the precision in the matches may be much higher, even with a low number of classes, thus (ideally) resulting in a high number of matches with high precision.

3.3.4 Fuzzy Filtering: the `MergeAction` class

The scaling of individual peaks with regards to parameters of the peaks or their related slices $\mathcal{S}_u^{(\mathcal{C})}$ as described in section 2.4.3 is handled by the `MergeAction` class, which only requires implementation of the `merge` function. The stereotype for this class is shown in figure 3.7, and there are currently two such fuzzy filters that the system makes use of:

Phase Modulation $\Gamma^{(\phi)}$ which scales the peaks with regards to their current phase/position in the beat so that peaks occurring at phase close to the

```
class AbstractMergeAction(Parametric):
        @abstractmethod
        def merge(self, peaks: Peaks, time: float,
                  history: ImprovisationMemory,
                  corpus: Corpus = None, **kwargs) -> Peaks:
            pass
```

Figure 3.7: Stereotype for implementing a `MergeAction`.

current phase of the output time $t^{(\mathcal{Y})}$ are emphasized and vice versa,

$$\Gamma^{(\phi)}\left(\boldsymbol{p}_i\right) = \begin{bmatrix} t_i^{(\mathcal{C})} & \phi_i y_i \end{bmatrix}^T \quad \forall \boldsymbol{p}_i \in \boldsymbol{P}_w \tag{3.9}$$

where

$$\phi_i = \exp\left[\cos\left(2\pi\left(t_w^{(\mathcal{Y})} - t_i^{(\mathcal{C})}\right)\right) - 1\right], \quad \phi \in \mathbb{R} \tag{3.10}$$

Next State Modulation $\Gamma^{(+)}$ which scales peaks close in time to the previously output slice $\mathcal{S}_{w-1}^{(\mathcal{Y})}$ by a constant α, i.e. for some $\varepsilon \in \mathbb{R}$

$$\Gamma^{(+)}(\boldsymbol{p}_i) = \begin{cases} \begin{bmatrix} t_i^{(\mathcal{C})} & \alpha y_i \end{bmatrix}^T & \text{if } \left|t_i^{(\mathcal{C})} - t_{w-1}^{(\mathcal{C})}\right| < \varepsilon \\ \boldsymbol{p}_i & \text{otherwise} \end{cases} \quad \forall \boldsymbol{p}_i \in \boldsymbol{P}_w. \tag{3.11}$$

3.4 Scheduling and the `Generator` Module

While the internal algorithms of the system has been quite thoroughly described by now, it has not yet been presented in context as a key aspect is missing - how input and output is handled over time, i.e. scheduling. The following sections describe the `Scheduler` module, which determines how influences and triggers are scheduled to generate actual midi/audio output and the modes that the scheduler operate under. Section 3.4.4 describes the `Generator` module, which is using the scheduler to generate new corpora offline, which in turn is the module that enables the statistical evaluation of the system, which chapter 4 will describe.

3.4.1 Scheduling

The main role of the `Scheduler` is to handle triggers to appropriately queue and output slices $\mathcal{S}_w^{(\mathcal{Y})}$ as they unfold over time, similar to a timeline in a DAW but where the events in the timeline are continuously generated by the system itself. The `Scheduler` has a running tick $t^{(\mathcal{Q})}$, a tempo $\zeta^{(\mathcal{Q})}$ and a queue of scheduled events $\mathcal{E}$, where each event has timestamp $t_w^{(\mathcal{Y})}$ and a predefined behaviour upon triggering, which depends on the event type. An event will be triggered when its tick $t_w^{(\mathcal{Y})}$ is greater than or equal to the scheduler tick $t^{(\mathcal{Q})}$. There are currently seven types of scheduled events:

`TempoEvent`: Sets the tempo of the scheduler to its value when triggered.

`MidiEvent`: Outputs a stored midi note on or note off message when triggered.

`AudioEvent`: Outputs an interval $[\tau_{\text{start}}, \tau_{\text{end}}]$ (in milliseconds) in the audio file to play over a duration determined by a tempo factor f_ζ, defined as

$$f_\zeta = \frac{\zeta_w^{(\mathcal{Y})}}{\zeta^{(\mathcal{Q})}} \tag{3.12}$$

where $\zeta_w^{(\mathcal{Y})}$ denotes the tempo of the audio event's corresponding slice $\mathcal{S}_w^{(\mathcal{Y})}$.

`CorpusEvent`: Outputs a slice $\mathcal{S}_w^{(\mathcal{Y})}$ when triggered. As we will see in sections 3.4.2 and 3.4.3, this behaviour seems to overlaps with the behaviour of midi and audio events, but they are never used in combination.

`InfluenceEvent`: Calls the influence process as described in section 2.3 for a given `Player` with its stored value.

`TriggerEvent`: Corresponding to an event in the trigger stream $\mathcal{Y}$, which when triggered calls the generate process as defined in section 2.4. The output of the generate process is sent back to the scheduler and queued, either as a `CorpusEvent` or as a `MidiEvent`/`AudioEvent`, depending on the type of scheduler, as we will see in the following two sections. In practice, this means that all `MidiEvent`s, `AudioEvent`s and `CorpusEvent`s are queued only through a `TriggerEvent`.

There are two different ways to add `TriggerEvent`s to the scheduler, which in turn depends on the scheduler's mode, which may be either `Automatic` or `Manual`. The `Manual` mode means that `TriggerEvent`s are added manually, which in practice means that they are added by the system after every influence call. This is useful to create a note-by-note interaction between the system and the input. The `Automatic` mode means that new `TriggerEvent`s are automatically queued after a duration corresponding to the generated output slice $\mathcal{S}_w^{(\mathcal{Y})}$, i.e. for a trigger $\mathcal{Y}_i$, a new trigger $\mathcal{Y}_{i+1}$ is added at $t_{i+1}^{(\mathcal{Y})}$ defined as

$$t_{i+1}^{(\mathcal{Y})} = t_i^{(\mathcal{Y})} + d_w^{(\mathcal{Y})}, \tag{3.13}$$

where $d_w^{(\mathcal{Y})}$ denotes the duration of the generated slice. Note that the retriggering uses the time of the trigger $t^{(\mathcal{Y})}$ rather than the time of the scheduler $t^{(\mathcal{Q})}$ when queueing new triggers to avoid drifting.

In practice, the scheduler is divided into two different classes, the `RealTime-Scheduler`, which will be described in section 3.4.2, and the `OfflineScheduler`, which will be described in section 3.4.3. As we will see, these two have very little in common apart from the handling of `TempoEvents` and `TriggerEvents`.

3.4.2 Real-time Scheduling

The behaviour of the `RealTimeScheduler` is in many ways similar to the behaviour of the audio thread in an audio plugin. It's a high-priority thread that's continuously polled at a millisecond interval, at each poll i updating its tick so that

$$t_i^{(\mathcal{Q})} = t_{i-1}^{(\mathcal{Q})} + \Delta\tau_i \frac{\zeta^{(\mathcal{Q})}}{60}, \tag{3.14}$$

where $\Delta\tau_i$ denotes the number of milliseconds that have passed since the last poll, and triggering any event w whose tick $t_w^{(\mathcal{Y})} \geq t_i^{(\mathcal{Q})}$.

When the system is used as a real-time framework, the scheduler is based on the `asyncio` Python module, where influencing and setting parameters, as well as queueing new `TriggerEvents` and `TempoEvents`, is handled by a different thread corresponding to the ui thread in an audio plugin. The `asyncio` module is however not truly multithreaded, but rather handles ui calls in-between polling scheduler. These ui calls are blocking the thread and completes its operation before the next poll is called, hence eliminating any risk of tearing. While such a solution would not be acceptable for a real audio thread as the ui calls may delay the audio thread up to a few milliseconds, it's not a problem when handling an event-based stream as these delays are too small to be perceivable.

Once a `TriggerEvent` has generated an output slice $\mathcal{S}_w^{(\mathcal{Y})}$, the real-time scheduler will extract its content as an `AudioEvent` for audio corpora or as a set of `MidiEvents` for midi corpora. For midi notes $\mathcal{N}_w$, great care must be taken when determining note ons and note offs, since we according to the slicing procedure determined in section 2.2.1 would add a single note to multiple slices. In practice, we generate note ons at $t_w^{(\mathcal{Y})}$ for any note $n_i \in \mathcal{N}_w^{(\text{on})}$, where the latter is defined as

$$\mathcal{N}_w^{(\text{on})} = \mathcal{N}_w^{(\mathcal{Y})} \setminus \mathcal{N}_{w-1}^{(\text{from})} \tag{3.15}$$

and note offs at $t_w^{(\mathcal{W})} + n_j$`.duration` for any note $n_j \in \mathcal{N}_w^{(\text{off})}$, where the latter is defined as

$$\mathcal{N}_w^{(\text{off})} = \left(\mathcal{N}_w^{(\mathcal{Y})} \setminus \mathcal{N}_w^{(\text{from})}\right) \cup \left(\mathcal{N}_{w-1}^{(\text{from})} \setminus \mathcal{N}_w^{(\text{to})}\right) \tag{3.16}$$

where

$$\mathcal{N}_w^{(\text{to})} = \left\{ n \mid n \in \mathcal{N}_w^{(\mathcal{Y})} : n.\texttt{onset} < t_w^{(\mathcal{Y})} \right\} \tag{3.17}$$

and

$$\mathcal{N}_w^{(\text{from})} = \left\{ n \mid n \in \mathcal{N}_w^{(\mathcal{Y})} : n.\texttt{onset} + n.\texttt{duration} > t_w^{(\mathcal{Y})} + d_w^{(\mathcal{Y})} \right\}. \tag{3.18}$$

When using the system in real-time, the `RealTimeScheduler` doesn't handle `InfluenceEvents` or `CorpusEvents`. The former are handled directly by the ui thread and the latter are converted to `MidiEvents` or `AudioEvents`.

3.4.3 Offline Scheduling

The `OfflineScheduler` is unlike the `RealTimeScheduler` designed for a single thread, where any operation stems from the scheduler itself while running. It is also not continuously polled, but iterating over all the events $\mathcal{E}$ in the scheduler in order (where the iterator is being updated after each cycle to allow requeueing of `TriggerEvents`) until the queue is empty. At each step i in the iteration, the tick $t_i^{(\mathcal{Q})}$ is updated so that

$$t_i^{(\mathcal{Q})} = \min_{t^{(\mathcal{Y})} \in \mathcal{E}} t^{(\mathcal{Y})} \tag{3.19}$$

where once again all events w whose tick $t_w^{(\mathcal{Y})} \geq t_i^{(\mathcal{Q})}$ are triggered in order, sorted by tick position as the first axis and type by the second, to ensure that `InfluenceEvents` are triggered before `TriggerEvents`, should they occur simultaneously.

Unlike the `RealTimeScheduler`, the `OfflineScheduler` will not handle `MidiEvents` or `AudioEvents` at all - it will output the slice $\mathcal{S}_w^{(\mathcal{Y})}$ corresponding to the `CorpusEvent` directly, effectively producing a new `Corpus`. But since the `Corpus` class is interchangeable with its midi and/or audio data, the generated result could easily be converted to a midi/audio file.

3.4.4 The `Generator` Module

The `Generator` is a separate module completely detached from the MaxMSP environment and the real-time system, and is designed around the `OfflineScheduler` to quickly generate new corpora. Similarly to section 3.3, the first steps when creating a `Generator` are to define and initialize the architecture and load a `Corpus`, which we from here on will call the source corpus. The `Generator` itself is an abstract class where the definition of the architecture is done by extending the class and implementing the `initialize` function, as can be seen in figure 3.8. But as the figure shows, there `Generator` requires two corpora in the

```
class Generator(ABC):
    def __init__(self, source_corpus: Corpus, influence_corpus: Corpus,
                 use_optimization: bool, gather_peak_statistics: bool,
                 name: Optional[str], **kwargs):
        # ...

    def run(self) -> Tuple[Corpus, Optional[PeaksStatistics]]:
        # ...

    @abstractmethod
    def initialize(self, **kwargs) -> None:
        pass
```

Figure 3.8: The signature of the constructor and `run` function as well as the `initialize` stereotype of the `Generator` class

constructor and returns a third corpus from the `run` function. These three correspond to the three main sections in chapter 2: a source corpus $\mathcal{C}$ similar to its definition in section 2.2, an influence corpus $\mathcal{K}$ (instead of an influence stream as we defined it in section 2.3) and an output corpus $\mathcal{O} = \left\{\mathcal{S}_1^{(\mathcal{Y})}, \dots \mathcal{S}_W^{(\mathcal{Y})}\right\}$, constructed from the output slices $\mathcal{S}_w^{(\mathcal{Y})}$ as defined in section 2.4. In other words, it will build the architecture and source corpus $\mathcal{C}$ as usual, but it will also build an influence corpus $\mathcal{K}$ using the same procedure. Each of the slices $\mathcal{S}_v^{(\mathcal{K})}$ of the latter are then queued as `InfluenceEvents` in the `OfflineScheduler` at their corresponding ticks $t_v^{(\mathcal{K})}$, together with a `TempoEvent` constructed from the slice's tempo $\zeta_v^{(\mathcal{K})}$ and (if the mode is set to `Manual`) a `TriggerEvent`.

When the `run` function is called, the `Generator` will iterate over all events in the `OfflineScheduler` for as long as there are `InfluenceEvents` left in the queue and then stop, effectively producing a corpus $\mathcal{O}$ with the same duration (and hopefully same traits) as the influence corpus $\mathcal{K}$, using the slices of the source corpus $\mathcal{C}$.

The `Generator` module also allows the user to gather statistics about the peaks at each step in the iteration, which as we will see in chapter 4 is very useful for evaluating the usefulness of the architecture from a performer's perspective.

Chapter 4

Evaluation

Evaluating large real-time systems with multiple parameters and complex output is by no means a trivial task. In the case of the system described in this text, most parameters are to some extent non-direct in a sense that the the change occurring when modifying a parameter will often not be immediately perceivable, and even if when it is, that change might be subtle enough to require special training to hear. It's also greatly dependent on the architecture used and which source corpus $\mathcal{C}$ the system is trained on, as well as the style of the influence $\mathcal{K}$ and how well it matches the source.

While usability tests of the actual system and/or listening tests of the system's output likely is the ideal way to evaluate a system designed for composition and improvisation, a quantitative study can serve as a means to evaluate individual components of the system, for finding reasonable parameters and designing good source corpora for matching different types of influences. For this reason, the evaluation presented in this chapter is designed as a semi-quantitative pre-study where a number of corpora from different periods and genres are evaluated in terms of usability, quantified as a number of measurements related to the generation of peaks, and quality of the output, quantified as a number of parameters related to the output.

4.1 Evaluation Procedure

Based on the `Generator` module described in section 3.4.4, each of the seven midi corpora described in table B.1 will act as both source $\mathcal{C}$ and influence $\mathcal{K}$ in relation to every other corpus, where the output $\mathcal{O}$ of each pair will be evaluated in relation to its influence $\mathcal{K}$, including information about its generated peaks, that will be gathered after each influence $v = 1, \ldots, V$. As one may recall, the influence $\mathcal{K}$ corresponds to the input of a real-time improviser, so this evaluation simulates the behaviour of the system if it's for example listening to (and being

Abbreviation	**Classifier**	**Parameters**
PT	Top Note Classifier	-
P12	Pitch Class Classifier	-
CSOM	SOM Chroma Classifier	-
AGMM50	Absolute GMM Chroma Classifier	Clusters: $K = 50$
AGMM100	Absolute GMM Chroma Classifier	Clusters: $K = 100$
AGMM150	Absolute GMM Chroma Classifier	Clusters: $K = 150$
RGMM50	Relative GMM Chroma Classifier	Clusters: $K = 50$
RGMM100	Relative GMM Chroma Classifier	Clusters: $K = 100$
RGMM150	Relative GMM Chroma Classifier	Clusters: $K = 150$

Figure 4.1: Evaluated classifiers Θ and their corresponding parameters.

influenced by) César Franck's Sonata for Violin and Piano while being trained on Arnold Schönberg's Drei Klavierstücke (and every other ordered pair of the items in table B.1). Formally, we will denote each tuple $\left(\mathcal{C}_i, \mathcal{K}_j, \mathcal{O}_{i,j}\right)$, $i, j = 1, \ldots, 7$ as a generator $\mathcal{G}_{i,j}$, with in total 49 generators. All generators will be constructed using the architecture described in section 4.1.2 and evaluated with regards to each classifier in figure 4.1, measuring each of the measurements described in section 4.1.1.

Note that the specific case where the same corpus is used as source and influence, i.e. all generators $\mathcal{G}_{i,j}$ where $i = j$, should for any well-designed classifier reproduce its influence $\mathcal{K}_j$ as its output $\mathcal{O}_{i,j}$ so that $\mathcal{S}_v^{(\mathcal{K}_i)} = \mathcal{S}_w^{(\mathcal{Y}_{i,j})}$ $\forall v = w = 1, \ldots, V$. For this reason, these generators will only be evaluated with regards to the measurement self-similarity described in section 4.1.1 and discarded for every other measurement, leaving in total 42 generators to evaluate.

4.1.1 Measurements

This section presents the measurements used to evaluate the usability and quality of each classifier. Most measurements are calculated either as a mean $\bar{x}$ and a standard deviation s or as a ratio η for a single classifier over an entire generator $\mathcal{G}_{i,j}$. The equations for calculating these measurements are presented on a form $\lambda_k = f(x)$ where λ_k denotes the value of the measurement at a specific time index k. Note that this time index varies depending on if the measurement is related to influence ($v = 1, \ldots, V$), output ($w = 1, \ldots, W$) or if it's discretized in some other way ($n = 1, \ldots, N$). Finally, the sample mean and standard deviation are estimated using the normal formulas:

$$\bar{x} = \frac{1}{K}\sum_{k=1}^{K} \lambda_k \tag{4.1}$$

$$s = \sqrt{\frac{1}{K-1}\sum_{k=1}^{K} (\lambda_k - \bar{x})^2} \tag{4.2}$$

Number of Peaks $\left(\bar{x}^{(\mathrm{card})}, s^{(\mathrm{card})}\right)$**:** The total number of peaks existing in all layers after each influence with index $v = 1 \ldots V$, i.e.

$$\lambda_v = \sum_{r=1}^{R} m_v^{(r)}, \quad \lambda_v \in \mathbb{Z}_{[0,\infty)} \tag{4.3}$$

where $m_v^{(r)}$ denotes the number of columns in the peak matrix $\boldsymbol{P}_v^{(r)}$ in layer $r = 1, \ldots, R$ and R denotes the total number of layers. Note that this measures peaks before merge, so it may contain duplicates. A high value for $\bar{x}^{(\mathrm{card})}$ with a low standard deviation $s^{(\mathrm{card})}$ indicates a consistent large number of matches over time and thus a high responsiveness of the system in general and the classifier in particular. A too high value could indicate problems with the decay rate of the peaks, as well as issues with the precision of the classifier.

Number of Generated Peaks $\left(\bar{x}^{(\mathrm{gen})}, s^{(\mathrm{gen})}\right)$**:** Number of new peaks generated by the evaluated classifier in layer r from the influence at time step $v = 1, \ldots, V$, i.e.

$$\lambda_v = m_v^{(r)}, \quad \lambda_v \in \mathbb{Z}_{[0,\infty)} \tag{4.4}$$

where $m_v^{(r)}$ here denotes the number of rows in the peak matrix $\boldsymbol{P}_v^{(r)}$ in the evaluated layer $r \in [1, R]$ before the concatenation with the previous time step's peaks performed in equation 2.10. Similar to $\bar{x}^{(\mathrm{gen})}$, this value is correlated to both the responsiveness and the precision of the system.

Score Selected Peak $\left(\bar{x}^{(\mathrm{mag})}, s^{(\mathrm{mag})}\right)$**:** The score (magnitude) of the by equation 2.25a selected peak $\hat{\boldsymbol{p}}_w$ at index $w = 1, \ldots, W$, i.e.

$$\lambda_w = \begin{bmatrix} 0 & 1 \end{bmatrix} \bar{\boldsymbol{p}}_w, \quad \lambda_w \in \mathbb{R}_{[0,\infty)}. \tag{4.5}$$

A high value (above 1) would indicate accumulation of consecutive peaks, which would mean that the selected segment over time matches the influence well, but at the same time also mean that new peaks generated (with default score of 1) will not be selected unless they have prior history (which is the intended behaviour of the system, but may sometimes result in poor matches).

Non-generating Influence Ratio $\left(\eta^{(\text{ng})}\right)$**:** The case where there are no new generated peaks at influence index $v = 1, \ldots, V$ is of particular interest, as this means that the classifier fails to find matches between the source and influence. Like most other measurements, there is no ideal value for this - a good classifier should fail to find matches between materials that are too far apart - but a high value will severly impact the performance of the system. This ratio is calculated as

$$\eta^{(\text{ng})} = \frac{1}{V} \sum_{v=1}^{V} \lambda_v, \quad \eta \in \mathbb{R}_{[0,1]} \tag{4.6}$$

where

$$\lambda_v = \begin{cases} 1 & \text{if } m_v^{(r)} = 0 \\ 0 & \text{otherwise} \end{cases}, \quad \lambda_v \in \mathbb{Z}_{[0,1]} \tag{4.7}$$

and $m_v^{(r)}$ once again denotes the number of rows in the peak matrix $\boldsymbol{P}_v^{(r)}$ of the evaluated layer $r \in [1, R]$ before the concatenation with previous peaks.

Chain Length $\left(\bar{x}^{(\text{len})}, s^{(\text{len})}\right)$**:** The number of from the source $\mathcal{C}_i$ consecutive slices in output $\mathcal{O}_{i,j}$, i.e. the average length of the segments sampled directly from $\mathcal{C}$. More specifically, each output slice $\mathcal{S}_w^{(\mathcal{Y})}$ temporarily stores its source index u, which we will denote $u_w^{(\mathcal{Y})}$ when assigned to the output $\mathcal{O}_{i,j}$. The average chain length is calculated so that

$$\lambda_w = \begin{cases} \lambda_{w-1} + 1 & \text{if } u_w^{(\mathcal{Y})} = u_{w-1}^{(\mathcal{Y})} + 1 \\ 0 & \text{otherwise} \end{cases}, \quad \lambda_w \in \mathbb{Z}_{[0,1]}. \tag{4.8}$$

A high value for this measurement indicates that large chunks are sampled directly from the source $\mathcal{C}_i$ and thus likely resulting in a higher degree of coherency at a beat/bar level of the composition/improvisation, but a too high value indicates in less originality of the output. Similarly, a high standard deviation will indicate one or a few long chains, which could be the result of either a long sequence of very good and/or accumulated matches or the opposite - a long sequence of no matches at all (which would cause the system to output consecutive slices as per equation 2.27).

Root Mean Square $\left(\bar{x}^{(\text{rms})}, s^{(\text{rms})}\right)$**:** The root mean square is the main measurement for evaluating the quality of each match, and is individually defined for each trait θ. It measures how much the output $\mathcal{O}_{i,j}$ differs from its influence $\mathcal{K}_j$ over time. As the slices of $\mathcal{O}_{i,j}$ and $\mathcal{K}_j$ are not necessarily are aligned, the comparison is done continuously over the entire duration of $\mathcal{K}_j$

(in ticks), denotes $T^{(\mathcal{K}_j)} \in \mathbb{R}_{[0,\infty)}$, discretized into $N \in \mathbb{Z}_+$ steps. For this purpose, we define two signals $\theta^{(\mathcal{K}_j)}[n]$ and $\theta^{(\mathcal{O}_{i,j})}[n]$ of length N so that the value at $\theta^{(\cdot)}[n]$ correspond to the evaluated trait value at time t_n,

$$t_n = \frac{nT^{(\cdot)}}{N}, \tag{4.9}$$

of corpus $(\cdot)$. The root mean square $\bar{x}^{(\mathrm{rms})}$ and standard deviation $s^{(\mathrm{rms})}$ are then calculated in accordance with equations 4.1 and 4.1.1 for a signal measurement $\lambda[n]$ of length N so that

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} \lambda[n] \tag{4.10}$$

$$s = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N} (\lambda[n] - \bar{x})^2}. \tag{4.11}$$

For each trait, the signal measurements $\lambda[n]$ are are defined as:

Top Note: For the signals $\theta^{(\mathcal{K}_j)}[n], \theta^{(\mathcal{O}_{i,j})}[n] \in \mathbb{Z}^N_{[0,127]}$,

$$\lambda[n] = \begin{cases} 0 & \text{if } \theta^{(\mathcal{K}_j)}[n] = \theta^{(\mathcal{O}_{i,j})}[n] \\ 1 & \text{otherwise} \end{cases}, \quad \lambda[n] \in \mathbb{Z}^N_{[0,1]} \tag{4.12}$$

Pitch Class: For the signals $\theta^{(\mathcal{K}_j)}[n], \theta^{(\mathcal{O}_{i,j})}[n] \in \mathbb{Z}^N_{[0,11]}$,

$$\lambda[n] = \begin{cases} 0 & \text{if } \theta^{(\mathcal{K}_j)}[n] \equiv_{12} \theta^{(\mathcal{O}_{i,j})}[n] \\ 1 & \text{otherwise} \end{cases}, \quad \lambda[n] \in \mathbb{Z}^N_{[0,1]} \tag{4.13}$$

Chroma: For the vector signals $\boldsymbol{\theta}^{(\mathcal{K}_j)}[n], \boldsymbol{\theta}^{(\mathcal{O}_{i,j})}[n] \in \mathbb{R}^{Nx12}_{[0,1]}$,

$$\lambda[n] = \|\hat{\boldsymbol{\theta}}^{(\mathcal{K}_j)}[n] - \hat{\boldsymbol{\theta}}^{(\mathcal{O}_{i,j})}[n]\|_2, \quad \lambda[n] \in \mathbb{R}^N_{[0,1]} \tag{4.14}$$

where

$$\hat{\boldsymbol{\theta}}^{(\cdot)}[n] = \frac{\boldsymbol{\theta}^{(\cdot)}[n]}{\max_{\theta \in \boldsymbol{\theta}^{(\cdot)}[n]} \theta}. \tag{4.15}$$

Self-similarity $\left(\eta^{(\mathrm{id})}\right)$: The final evaluation measurement is self-similarity, which as was mentioned above is only evaluated for generators $\mathcal{G}_{i,j}$ where $i = j$, and basically compared how well the output $\mathcal{O}_{i,j}$ of the generator reproduces the original influence $\mathcal{K}_j$. Similarly to root mean square, this is measured continuously over the duration $T^{(\mathcal{K})}$ discretized in N steps. It utilizes the fact that $\mathcal{C}_i = \mathcal{K}_j$, and thus indices v correspond to indices u, so we can use the previously defined notation $u_w^{(\mathcal{Y})}$ for the source index u as stored in the output slice $\mathcal{S}_w^{(\mathcal{Y})}$ for output index w. We once again define discrete vectors $u^{(\mathcal{C})}[n]$ and $u^{(\mathcal{Y})}[n]$ of length N and define the self-similarity measure $\lambda[n]$ as

$$\lambda[n] = \begin{cases} 1 & \text{if } u^{(\mathcal{C})}[n] = u^{(\mathcal{Y})}[n] \\ 0 & \text{otherwise} \end{cases}, \quad \lambda[n] \in \mathbb{R}^N_{[0,1]}, \tag{4.16}$$

from which we calculate self-similarity ratio $\eta^{(\mathrm{id})}$ as

$$\eta^{(\mathrm{id})} = \frac{1}{N}\sum_{n=1}^{N} \lambda[n], \quad \eta^{(\mathrm{id})} \in \mathbb{R}_{[0,1]}. \tag{4.17}$$

This measurement simulates an identity operator and works as a sanity check - for any well-designed classifier, given a corpus $\mathcal{C}_i = \mathcal{K}_j$, we expect $\eta^{(\mathrm{id})}$ to be close to 1, with the only exception being if the corpus contains identical repetition of longer sections.

For each classifier k in figure 4.1, each measurement will be evaluated with regards to every generator $\mathcal{G}_{i,j}$, $i, j = 1, \ldots, 7$. This produces a set $\mathcal{G}^{(k)}$ for every classifier k so that

$$\mathcal{G}^{(k)} = \left\{ \mathcal{G}_{i,j}^{(k)} \mid i = 1, \ldots 7, j = 1, \ldots 7, i \neq j \right\}, \tag{4.18}$$

each containing a generator with all of the above measurements. For each measurement, section 4.2 will present the generator $\mathcal{G}_{i,j}^{(k)} \in \mathcal{G}^{(k)}$ that produces (a) the minimum value, (b) the maximum value and (c) the unweighted mean of all generators in $\mathcal{G}^{(k)}$, as well as their corresponding standard deviations.

4.1.2 Architecture and Parameters

The architecture used for evaluation consists of a single `Player`, a single `StreamView` and a single `Atom`. The `Atom` uses the `Classifier` that is currently being evaluated as presented in figure 4.1 with, if applicable, the parameters as specified in the figure. The `Player` uses each of the `MergeActions` presented in section 3.3.4, and the `Scheduler` only operates in `Manual` mode. The values of all other user-controlled parameters are the same for all classifiers and can be found in figure 4.2.

Parameter	**Value**
n-gram order (γ)	2
weight (α, `Atom` & `StreamView`)	1.0
next state modulation ($\varepsilon_{\Gamma(+)}$)	0.5
peak decay ($\tau_{\Gamma(\phi)}$)	4.6

Figure 4.2: User-controlled parameters used in the evaluated architecture.

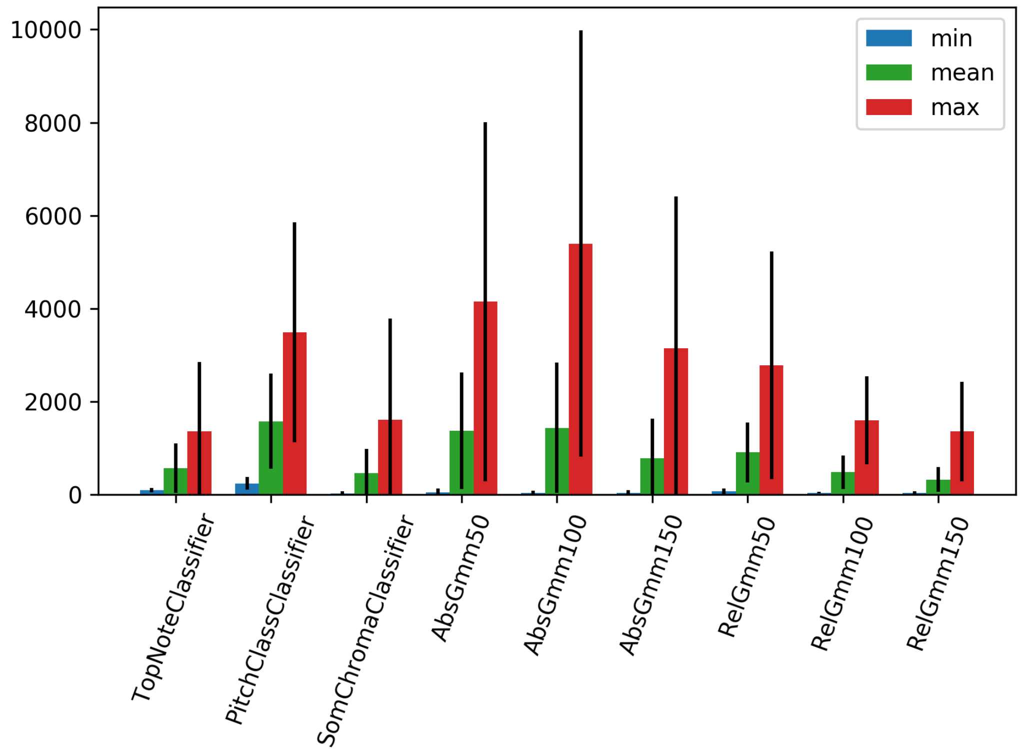


Figure 4.3: Number of peaks $\bar{x}^{(\text{card})}$ ordered by classifier, with error bars indicating the standard deviation $s^{(\text{card})}$.

4.2 Evaluation Results

4.2.1 Number of Peaks

For pitch classifiers, the number of peaks $\bar{x}^{(\text{card})}$ range from 93.98, in the worst ("min") case of the Top Note classifier, to 3491, in the best ("max") case of the Pitch Class classifier. For chroma classifiers, the range is between 29.43, in the worst case of the SOM Chroma classifier, to 5395 in the best case of the Absolute GMM classifier ($K = 100$). An overview of the data is presented in figure 4.3 and more detailed information can be found in figure C.2, along with information about which corpora are being used in each of the worst and best cases.

4.2.2 Number of Generated Peaks

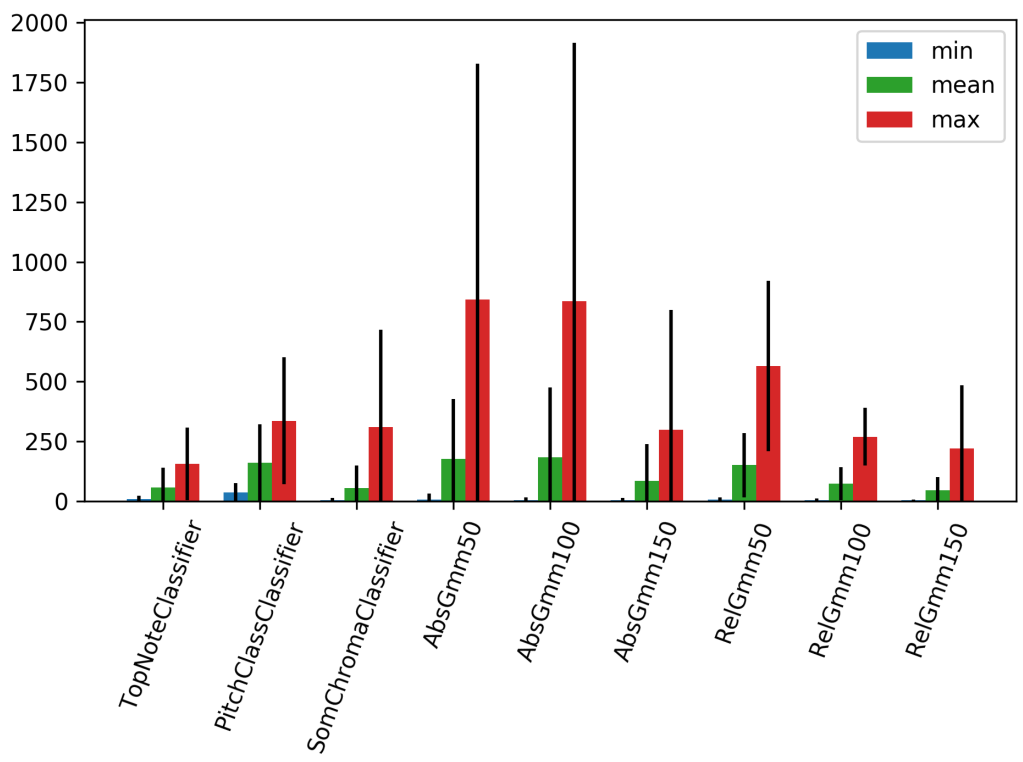


Figure 4.4: Number of generated peaks $\bar{x}^{(\text{gen})}$ ordered by classifier, with error bars indicating the standard deviation $s^{(\text{gen})}$.

For pitch classifiers, the number of generated peaks $\bar{x}^{(\text{gen})}$ range from 9.366, in the worst ("min") case of the Top Note classifier, to 336.3, in the best ("max") case of the Pitch Class classifier. For chroma classifiers, the range is between 3.634, in the worst case of the Relative GMM Chroma classifier ($K = 150$), to 842.2 in the best case of the Absolute GMM classifier ($K = 50$). An overview of the data is presented in figure 4.4 and more detailed information can be found in figure C.4, along with information about which corpora are being used in each of the worst and best cases. Also note that the number of peaks generated has a theoretical limit of the length of the source corpus, i.e. $\bar{x}^{(\text{gen})} \leq U$, and will thus vary for corpora of different lengths.

4.2.3 Score Selected Peak

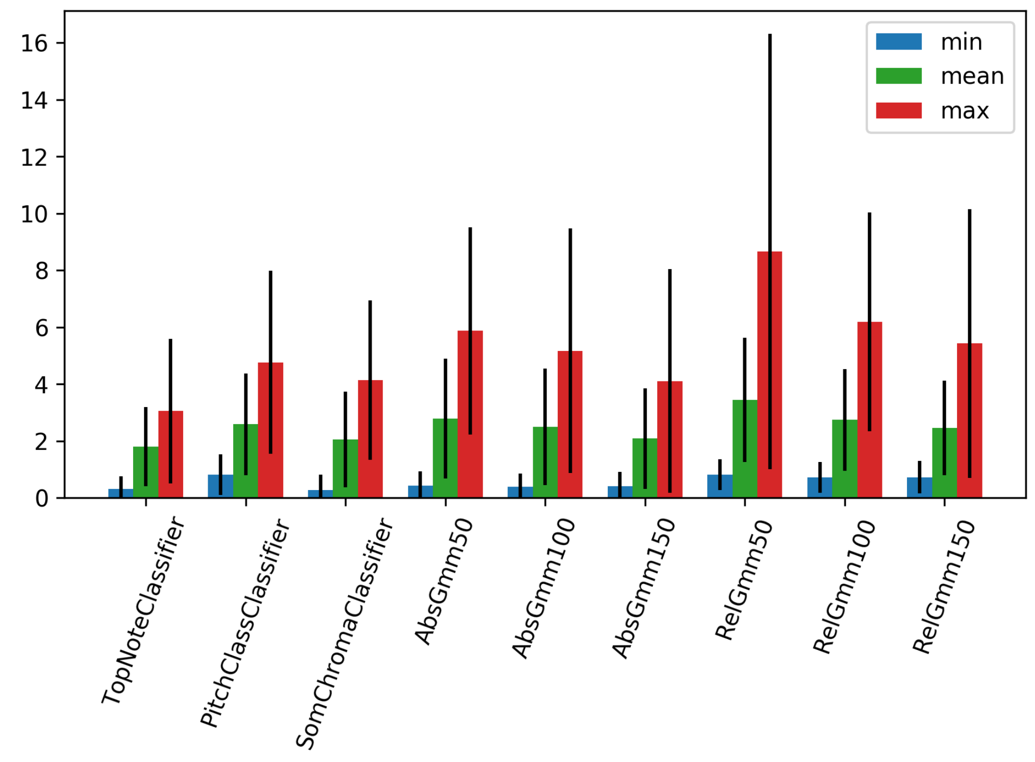


Figure 4.5: Score selected peak $\bar{x}^{(\text{mag})}$ ordered by classifier, with error bars indicating the standard deviation $s^{(\text{mag})}$.

For pitch classifiers, the score of the selected peak $\bar{x}^{(\text{mag})}$ range from 0.3058, in the worst ("min") case of the Top Note classifier, to 4.765, in the best ("max") case of the Pitch Class classifier. For chroma classifiers, the range is between 0.2691, in the worst case of the SOM Chroma classifier, to 8.661 in the best case of the Relative GMM classifier ($K = 50$). An overview of the data is presented in figure 4.5 and more detailed information can be found in figure C.3, along with information about which corpora are being used in each of the worst and best cases.

4.2.4 Non-generating Influence Ratio

For pitch classifiers, the non-generating influence ratio $\eta^{(\text{ng})}$ range from 0.0001, in the best ("min") case of the Pitch Class classifier, to 0.7535, in the worst ("max") case of the Top Note classifier. For chroma classifiers, the range is between 0.0073, in the best case of the Relative GMM Chroma classifier ($K =$

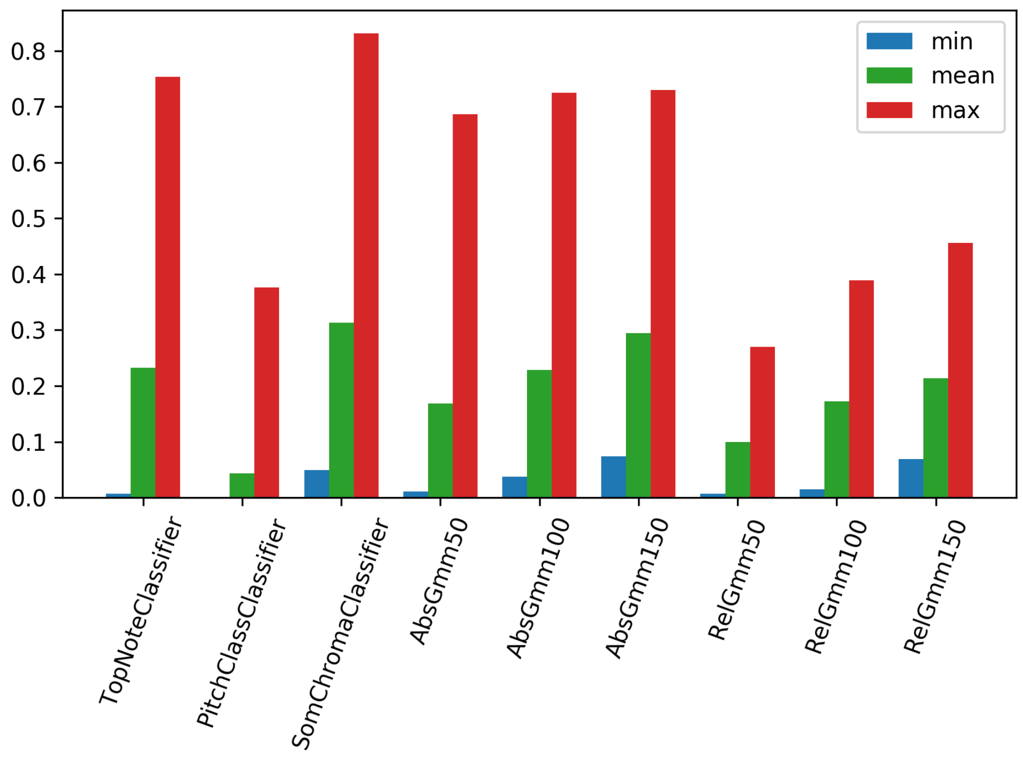


Figure 4.6: Non-generating influence ratio $\eta^{(\mathrm{ng})}$ ordered by classifier.

50), to 0.7299 in the worst case of the Absolute GMM classifier ($K = 150$). An overview of the data is presented in figure 4.6 and more detailed information can be found in figure C.5, along with information about which corpora are being used in each of the worst and best cases.

4.2.5 Chain Length

For pitch classifiers, the chain lengths mean $\bar{x}^{(\mathrm{len})}$ range from 0.0278, in the best ("min" - we will see in chapter 5 why the "min" case is labelled as the "best" case) case of the Pitch Class classifier, to 1.930, in the worst ("max") case of the Top Note classifier, and in this case more importantly, standard deviations ranging from 0.1776, in the best case of the Pitch Class classifier to 8.443 in the worst case of the Top Note classifier. For chroma classifiers, the range is between 0.0181, in the best case of the SOM Chroma classifier, to 2.986 in the worst case of the SOM Chroma classifier, with corresponding standard deviations ranging from 0.1404 in the best case of the SOM Chroma classifier to 21.53 in the worst case of the SOM Chroma classifier. An overview of the data is presented in figure 4.3 and more detailed information can be found in figure C.2, along with

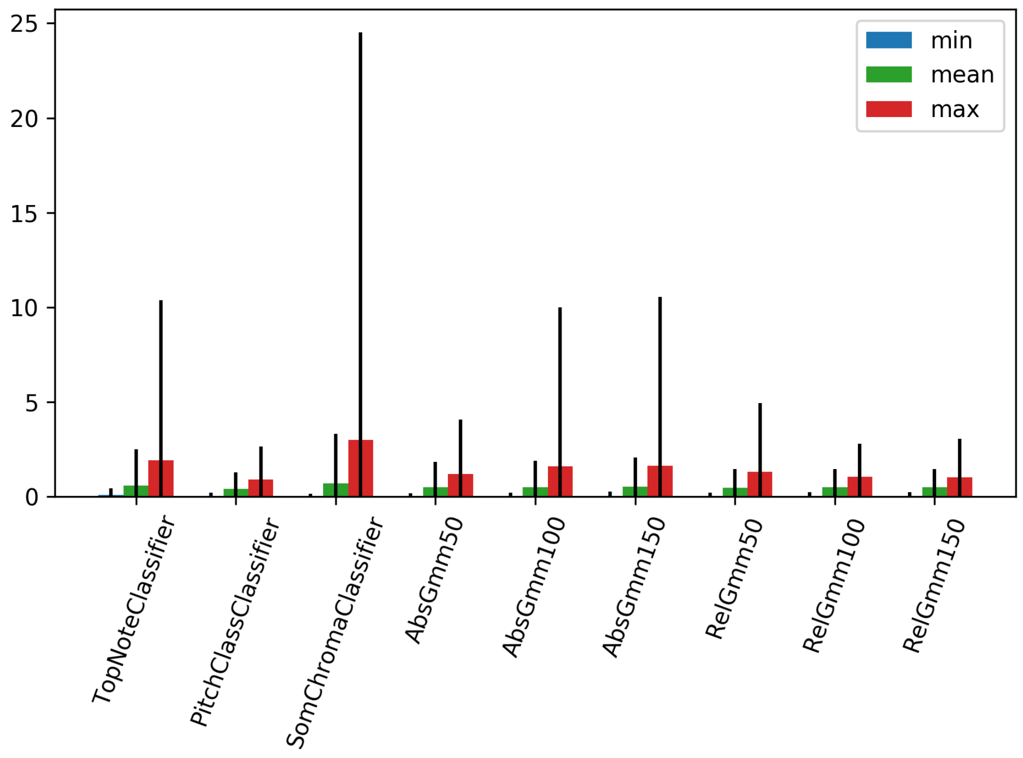


Figure 4.7: Chain length $\bar{x}^{(\text{len})}$ ordered by classifier, with error bars indicating the standard deviation $s^{(\text{len})}$.

information about which corpora are being used in each of the worst and best cases.

4.2.6 Root Mean Square

For pitch classifiers, the root mean square $\bar{x}^{(\text{rms})}$ range from 0.1707, in the best ("min") case of the Pitch Class classifier, to 0.7900, in the worst ("max") case of the Top Note classifier. For chroma classifiers, the range is between 0.2081, in the best case of the SOM Chroma classifier, to 0.4569 in the worst case of the Absolute GMM Classifier ($K = 100$). An overview of the data is presented in figure 4.3 and more detailed information can be found in figure C.2, along with information about which corpora are being used in each of the worst and best cases.

For comparison, another test was run to compute root mean square values for all traits in a completely random case. This test was performed so that each of the 42 corpus pairs $(\mathcal{C}_i, \mathcal{K}_j)$ corresponding to generators $\mathcal{G}_{i,j}$, $i \neq j$ were evaluated, where in each pair, the influence corpus $\mathcal{K}_j$ was randomly shuffled and the

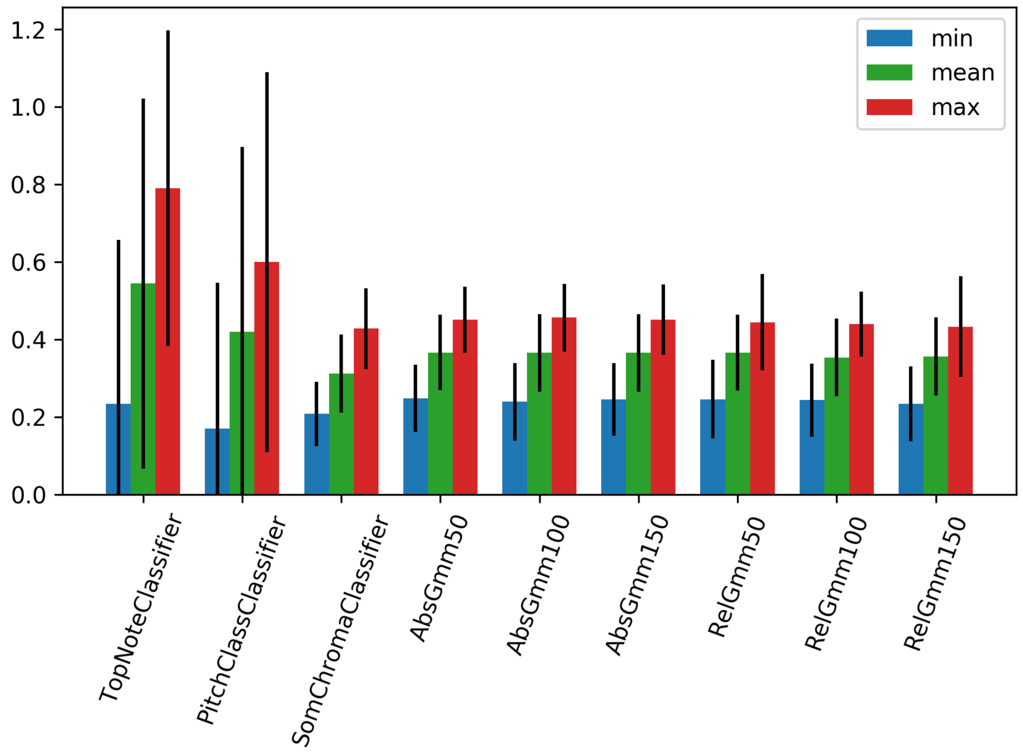


Figure 4.8: Root mean square $\bar{x}^{(\mathrm{rms})}$ ordered by classifier as defined by the trait corresponding to the evaluated classifier, with error bars indicating the standard deviation $s^{(\mathrm{rms})}$.

two corpora were compared using the root mean square procedure as defined in section 4.1. This procedure was repeated 100 times for each corpus pair, resulting in a total of 4200 evaluations, from which a mean and a standard deviation were calculated. The results were:

Top Note: $\bar{x}^{(\mathrm{rms})} = 0.9650$, $s^{(\mathrm{rms})} = 0.1833$,

Pitch Class: $\bar{x}^{(\mathrm{rms})} = 0.9085$, $s^{(\mathrm{rms})} = 0.2879$,

Chroma: $\bar{x}^{(\mathrm{rms})} = 0.4395$, $s^{(\mathrm{rms})} = 0.09094$.

4.2.7 Self-similarity

For all classifiers, the self-similarity $\eta^{(\mathrm{id})}$ range from 0.7240, in the worst ("min") case of the Pitch Class classifier, to 1.000, in the best ("max") case (all classifiers). An overview of the data is presented in figure 4.3 and more detailed information can be found in figure C.2, along with information about which corpora are being used in each of the worst and best cases.

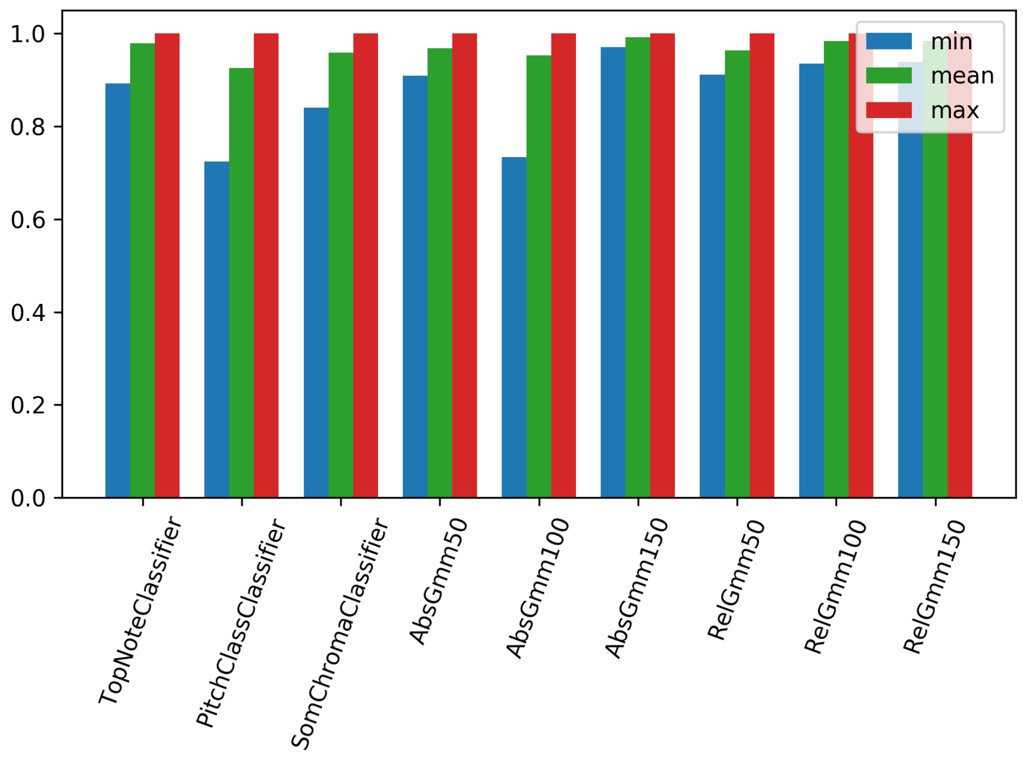


Figure 4.9: Self-similarity $\eta^{(\mathrm{id})}$ ordered by classifier.

As the self-similarity in some of the worst cases were surprisingly low, a second test was performed where the root mean square in these specific cases (i.e. generators $\mathcal{G}_{i,j}$, $i = j$) were computed. These are presented in figure C.8 and range from 0.0000 (all classifiers) in the best case to 0.0203 in the worst case (Relative GMM Chroma classifier, $K = 50$).

Chapter 5

Discussion

In this chapter, the results presented in chapter 4 will be discussed in relation to each classifier. This discussion will focus on the two main goals of the evaluation: usability and quality, as described in chapter 4. The former is in this chapter further subdivided into two areas: peak generation and peak accumulation, where peak generation focus on measurements related to the number of peaks existing in the system at each point in time as well as the non-generating influence ratio $\bar{x}^{(\mathrm{ng})}$, while peak accumulation focus on measurements regarding how the peaks relate to and merge with each other.

5.1 Top Note Classifier (PT)

For the Top Note classifier, we can see that we have a rather low value $\eta^{(\mathrm{ng})}$ of 0.2326 for the mean case, which means that on average 76.74% of the influences generate new peaks. The minimum value of 0.0067 indicates that in the best case out of the 42 generators, almost every influence generates a peak, while in the worst case, 75.35% of the influences will not generate any peaks. Regarding the two measures for number of peaks generated, $\bar{x}^{(\mathrm{card})}$ and $\bar{x}^{(\mathrm{gen})}$, we see that there is, even in the worst case, on average 93.98 peaks existing at a given time with a standard deviation of $s^{(\mathrm{card})} = 48.71$, while in the average case we have $\bar{x}^{(\mathrm{card})} = 567.1$ peaks before merge with a standard deviation $s^{(\mathrm{card})} = 534.9$. While the comparatively large standard deviations indicate that there could occasionally be a critically low number of peaks, it will in most cases be a decent number of peaks. In the worst case, the low number of peaks generated per influence, $\bar{x}^{(\mathrm{gen})} = 9.366$ with a standard deviation of $s^{(\mathrm{gen})} = 13.51$, is however problematic.

For the score of the peaks, we have in the mean case $\bar{x}^{(\mathrm{mag})} = 1.801$ and $s^{(\mathrm{mag})} = 1.394$, which indicates some degree of peak accumulation (as new peaks are created with score 1.0), while the minimum case has a $\bar{x}^{(\mathrm{mag})} = 0.3058$ with a standard deviation of $s^{(\mathrm{mag})} = 0.4574$, in other words no or very little peak accumulation. Regarding the chain length measurement, as described in section 4.1, this measurement was originally intended as a quality measure, but manual inspection of the minimum and maximum cases of the raw data has shown that there's in all of these cases, there's in fact a strong relationship between high chain length, high non-generating influence ratio and high root mean square. There are theoretically two disparate cases when the chain length would be high:

1. When multiple consecutive influences trigger consecutive slices as output. This is the case we see in generators $\mathcal{G}_{i,j}, i = j$, where the chain length often is several orders of magnitude higher than in figure 4.7, while root mean square and non-generating ratio are close to zero, as the generator perfectly matches the influence to the corpus.

2. When no peaks $\boldsymbol{P}_w$ exist at all for multiple consecutive outputs $w \in [1, W]$. In this case, the system will output the next event as per equation 2.27 since there are no peaks to base the output on. In this case, we will have a high non-generating ratio $\eta^{(\mathrm{ng})}$ and (most likely) a high root mean square. We will for these scenarios often see high standard deviations $s^{(\mathrm{len})}$, indicating that there are patches of very long chains where no matches are found at all.

As we will see in the following sections, the cases among the 42 generators $\mathcal{G}_{i,j}, i \neq j$ with the highest chain lengths are in fact always cases with high non-generating ratio and high root mean square, thus high chain length is in this evaluation related to aspects regarding peak generation and accumulation (i.e. usability) rather than the quality of the output. Or perhaps even worse - a high chain length is (in most cases) an indication that the system is not listening to the performer but rather generating results solely based on a previous state.

For the Top Note classifier, we have a chain length $\bar{x}^{(\mathrm{len})} = 0.5741$ with a comparatively high standard deviation $s^{(\mathrm{len})} = 1.927$ and in the worst case $\bar{x}^{(\mathrm{len})} = 1.930$, $s^{(\mathrm{len})} = 8.443$. These high standard deviations indicate that there are quite a few patches of longer chains where no peaks are present at all.

Regarding the quality of the output, we can see from figure 4.8 that the root mean square $\bar{x}^{(\mathrm{rms})}$ is surprisingly high in all cases. For the average case, we have $\bar{x}^{(\mathrm{rms})} = 0.5446$, $s^{(\mathrm{rms})} = 0.4775$, indicating that in 54.46% of the slices of the output, the top note of the output does not match the top note of the influence. Even in the best case, almost 25% of the output slices will not match the influence. In this case, we have a non-generating ratio close to 0, so we can with some certainty say that the the fact that the system, despite finding matches with the correct top note, will due to either peak accumulation and/or fuzzy filtering still in the average case output something else than the match more than half of the time. This is of course part of the design and intention with the system - if the purpose was to match the input note-by-note, a much simpler system without memory, merge actions or accumulation of peaks would suffice and perform better - but it's interesting to see how high the ratio of non-matches in fact is.

Finally, regarding the self-similarity $\eta^{(\mathrm{id})}$, we can see that it in the average case reproduces the original to a 97.92% degree, and in the worst case, produced by the Brahms corpus (see figure C.7), where the somewhat low value of $\eta^{(\mathrm{id})} = 0.8929$ is achieved, the root mean square (as per figure C.8) is as low as 0.0089, indicating that the differences in self-similarity are caused by locating identical places in the corpora, as both the used movements in that corpus follows a sonata form.

In summary, we can see that in the average case, the Top Note classifier is performing quite well when it comes to generation and accumulation of peaks. It is however quite context-dependent, and will in some cases, as shown by the worst cases in the study, have problems with both generation and accumulation of peaks. Also, while it generally will be able to find peaks, it will not necessarily output these in a single atom architecture like the one evaluated due to accumuluation of peaks and/or as a result of fuzzy filtering, thus resulting in a rather bad quality even in the mean case when comparing the top note of the output to the top note of the influence.

5.2 Pitch Class Classifier (P12)

Compared to the Top Note classifier, the Pitch Class classifier has a much lower $\eta^{(\mathrm{ng})}$: in the mean case 0.0430 and in the best case 0.0001. This is to be expected, as the alphabet over which the Pitch Class classifier is defined only contains 12^{γ} symbols for an n-gram order of γ, compared to the Top Note classifier whose alphabet contains 127^{γ} possible symbols. The number of peaks generated $\bar{x}^{(\mathrm{gen})}$ is much higher than the Top Note classifier with a stable value even in the worst case, and the number of non-merged peaks $\bar{x}^{(\mathrm{card})}$ is good in all cases. This indicates that the system in general will have a lot more peaks to use in comparison to the Top Note classifier, and that the Pitch Class classifier is much less context sensitive, even though there are still quite a few influences that won't generate peaks in this case.

The score of the peaks, $\bar{x}^{(\mathrm{mag})}$ is 2.585 in the mean case and 4.765 in the best case, indicating quite some extent of peak accumulation, while in the worst case there's little, but still a more stable extent of accumulation compared to the Top Note classifier, with an average of $\bar{x}^{(\mathrm{mag})} = 0.8139$ and $s^{(\mathrm{mag})} = 0.7091$. The chain length $\bar{x}^{(\mathrm{len})}$ is lower than the Top Note classifier in all cases and, more importantly, the standard deviation is much lower in the mean and worst cases, both relative and absolute, indicating that there still may be occasional sequences of no peaks occurring, but at a much lower rate than in the Top Note classifier.

While the root mean square has a less strict definition for the Pitch Class classifier than the Top Note classifier, the values are still comparatively large in the mean and worst cases with $\bar{x}^{(\mathrm{rms})} = 0.4195$ and 0.5997 correspondingly. It's also interesting to note that this case, as well as the worst non-generating influence ratio, occurs between the Franck corpus and the Palestrina corpus, where most other such worst-cases occur between the Schönberg corpus and some tonal (or modal) corpus. While the used sample is much too small to draw any definitive conclusions from, this could potentially indicate that the shape of the melodic lines in the Schönberg corpus is a bigger problem than the differences in tonality, when compared to the Top Note classifier. For the self-similarity, we see the lowest value of all classifiers with $\eta^{(\mathrm{id})} = 0.7240$, but this seems to once again indicate that it's finding duplicate slices, as the root mean square value in this case is lower than 0.0164, as indicated by figure C.8.

Overall, we see that (not surprisingly) the Pitch Class classifier is less context-sensitive than the Top Note classifier, given its smaller alphabet. It's a good option for disparate corpora and will almost always have peaks, but may in some cases have a rather large non-generating influence ratio. The root mean square is in most cases still rather high, most likely due to peak accumulation and the intended design of the system as was explained in section 5.1.

5.3 SOM Chroma Classifier (CSOM)

For the SOM Chroma classifier, we have a non-generating ratio $\eta^{(\mathrm{ng})}$ ranging from 0.0496 in the best case up to 0.8310 in the worst case, with a mean case of 0.3135. The number of peaks in the system $\bar{x}^{(\mathrm{card})}$ are on average 462.3 with a standard deviation of $s^{(\mathrm{card})} = 518.6$, thus a number of peaks almost on par with the Top Note classifier, but a rather problematic number of peaks in the worst case, $\bar{x}^{(\mathrm{card})} = 29.43$ with $s^{(\mathrm{card})} = 39.29$. For the number of peaks generated at each influence, $\bar{x}^{(\mathrm{gen})}$, we can see in figure 4.4 that the standard deviations are rather large compared to the means in all cases, indicating that there will be some problems with peak generation, especially for min and mean case, where the number of peaks are low.

The score of the peaks $\bar{x}^{(\mathrm{mag})}$ are for the mean and best cases 2.045 and 4.132 respectively, indicating some degree of peak accumulation, while for the worst case 0.2691, indicating very little peak accumulation. The chain lengths are one of the highest of all classifiers, $\bar{x}^{(\mathrm{len})} = 0.7091$ in the average case and 2.9855 in the worst case, with corresponding standard deviations $s^{(\mathrm{len})} = 2.600$ and 21.53 respectively. Especially the latter indicates that there are several long segments of no peaks at all.

For the quality of the output, we can see that the classifier successfully reproduces its influence for generators $\mathcal{G}_{i,j}, i = j$ where, once again, the worst case of $\bar{x} = 0.8401$ is caused by the Mozart corpus, but where the root mean square in this case is 0.0141 (as per figure C.8), thus once again likely selecting the identical repetitions in the sonata form. For the second quality measurement, the root mean square, we have $\bar{x}^{(\mathrm{rms})} = 0.2081$, 0.3125 and 0.4279 for the best, average and worst case, with rather uniform standard deviations $s^{(\mathrm{rms})} = 0.0830$, 0.1012 and 0.1049 respectively. Of all the chroma classifiers, these are the lowest values for root mean square, and would thus indicate the best performance out of all the chroma classifiers. There are however a number of problems with this measurement, the first one becomes evident when we inspect figure 4.8, where all chroma classifiers (apart from the SOM chroma classifier, which is slightly lower in all values) independent of number of clusters have almost identical values for their corresponding best, average and worst cases. This isn't necessarily an issue, but it could indicate that the measurement is rather coarse for the purpose. The second issue is related to the human perception of chroma. For while we with a high degree of certainty can tell that the implementation of root mean square in the Top Note classifier and Pitch Class classifier roughly models (while ignoring aspects such as timbre and intonation) the human perception of whether two notes are of the same pitch and whether two notes are of the same pitch class respectively, the perception of chroma is more complex. While we do have certain boundaries for the root mean square in this case - the root mean square when comparing a corpus to itself is 0, while as presented in section 4.2, the root mean square of a randomized comparison between two corpora is on average 0.4394 with a standard deviation of 0.009094 - but we cannot necessarily assume that this measurement within these boundaries linearly corresponds to the human perception of chroma.

Do bear in mind that there was a very specific reason why this measurement for root mean square was chosen for chroma, namely the fact that this is how the chroma classification was done in the original implementation [6] and still is with regards to the SOM Chroma classifier, as per equation 3.5. If this measurement proves to be problematic, that would in turn indicate further problems with the original implementation. Ideally, the solution to this problem would be to develop better measurements for (filtered) chroma vectors, but as this is out of the scope for this thesis, all further discussion regarding quality of chroma will treat these two disparate assumptions separately:

1. The root mean square $\bar{x}^{(\mathrm{rms})}$ as defined in equation 4.14 is a valid measurement corresponding to the perception of differences between two chroma vectors,

2. The root mean square $\bar{x}^{(\mathrm{rms})}$ as defined in equation 4.14 is insufficient as a measurement for the perception of differences between chromas, with everything that implies for the SOM Chroma classifier.

In the former case, we can make some assumptions regarding the quality of the output in terms of chroma based on the $\bar{x}^{(\mathrm{rms})}$, but in the latter, we cannot - we can only determine how well the classifier performs with regards to peak generation, accumulation and self-similarity.

For the SOM Chroma classifier, we can safely under both premises say that the worst case is very close the random case, hence the quality with regards to chroma is very low, but for the best and mean cases, we can under the first premise also say that out of the three chroma classifiers, the SOM Chroma classifier has the best quality.

In summary, we see that the SOM Chroma classifier is one of the most context dependent classifiers, spanning a range from 0.0496 to 0.8310 for the non-generating influence ratio, and while performing well in most best cases, the mean case is generally less performant and the worst case is often terrible in terms of both usability and quality. While it has the best quality under the first of the root mean square measurement for the best and mean cases, it's still close to random in the worst case.

5.4 Absolute GMM Chroma Classifier (AGMM)

For the Absolute GMM Chroma classifier, the non-generating influence ratio $\eta^{(\mathrm{ng})}$ is lower than the SOM Chroma classifier for all cases and all cluster sizes except in the best case for $K = 150$ clusters, The values in the best and mean cases seem to be linearly increasing with the number of clusters, from $\eta^{(\mathrm{ng})} = 0.1685$ in the mean case for $K = 50$ up to 0.2949 for $K = 150$, while the worst cases are fixed around 0.7 in all cases. The number of peaks $\bar{x}^{(\mathrm{card})}$ and number of generated peaks $\bar{x}^{(\mathrm{gen})}$ are much higher than in the SOM Chroma classifier, indicating a much more stable amount of peaks, except in the worst case where the number of generated peaks $\bar{x}^{(\mathrm{gen})}$ is close to the same as the SOM Chroma classifier but with slightly higher standard deviations. In the two lower cluster sizes $K = 50$ and $K = 100$, we do however see a worrying "best" case, where on average 842.2 peaks and 835.0 peaks are generated respectively. This would in this case mean that for the Brahms corpus consisting of 7040 slices, around 12% of these slices matches the input every influence. This seems to indicate problems with the precision of the classifier. So, while the classifier in the mean case performs better than the SOM Chroma classifier in terms of peak generation, the sheer number of peaks generated indicates some problems with the accuracy in the worst case.

Regarding the score of the peaks $\bar{x}^{(\text{mag})}$, we see an accumulation that is approximately of the same magnitude as the SOM Chroma classifier with slightly better results for $K = 50$ and $K = 100$, which seems to be linearly decreasing to the same level as the SOM Chroma for $K = 150$. The chain length $\bar{x}^{(\text{len})}$ is comparatively lower in all the mean cases, but indicates quite some sequences of no peaks in the worst case for clusters sizes $K = 100$ and $K = 150$.

As for the quality of the classifier, the interpretation of the results varies depending on which of the two assumptions that were presented in section 5.3 we choose. If the first assumption is valid, one peculiarity is the fact that all cluster sizes K result in almost identical values for their best, mean and worst cases respectively. We also see that the classifier performs slightly worse than the SOM Chroma classifier in the best and mean case, while the worst case, similarly to the SOM Chroma classifier, is close to random. Under the second assumption, we can draw few conclusions about the quality from this measurement. For the self-similarity measurement, it's very high in all cases but for $K = 100$, where we have a similar situation as in section 5.2, where we due to the low root mean square value of 0.0172 in this case (as indicated in figure C.8) can safely assume that it's finding repetitions and properly reproduces the influence corpus.

In summary, we see that apart from the slightly higher root mean square, the Absolute GMM classifier outperforms the SOM Chroma classifier in all measurements. There are some cases of a problematically high amount of generated peaks $\bar{x}^{(\text{gen})}$ for lower numbers of clusters K that could indicate problems with the quality of the output in the worst case, but the fact that K is a user-controlled parameter should allow the user to easily adapt the cluster size in real-time based on the number of peaks generated.

5.5 Relative GMM Chroma Classifier (RGMM)

The final classifier, the Relative GMM Chroma classifier, is the only classifier that doesn't base its clustering on the matrix $\boldsymbol{X}$ that was described in section 3.3.3. It's therefore compared to the Absolute GMM classifier free of any bias caused by clustering on $\boldsymbol{X}$. We see that in terms of peak generation, this is by far the most balanced classifier. It has the lowest best, mean and worst cases in terms of non-generating influence ratio $\eta^{(\text{ng})}$ and is the only chroma classifier with reasonable values even in the worst case, seemingly linearly increasing from $\eta^{(\text{ng})} = 0.2695$ for $K = 50$ to 0.4561 for $K = 150$. It has a much lower mean case for $\bar{x}^{(\text{card})}$ than the corresponding Absolute GMM classifiers, but never to the point of being problematic, except for the worst case, but even there the standard deviation is much lower, with a promising $\bar{x}^{(\text{card})} = 38.57$, $s^{(\text{card})} = 18.19$ for $K = 100$, suggesting that there rarely is a lack of peaks even in the worst case.

Despite the in comparison to the Absolute GMM Chroma classifier lower number of peaks, the score of the peaks $\bar{x}^{(\mathrm{mag})}$ is much higher in all cases, suggesting a rather high degree of peak accumulation. Similarly, while the chain length $\bar{x}^{(\mathrm{len})}$ is similar to the Absolute GMM classifier in the mean and best case, we see much lower standard deviations for $K = 100$ and $K = 150$, $s^{(\mathrm{len})} = 1.718$ and 2.052 respectively, indicating a low number of sequences without peaks.

Regarding the quality of the Relative GMM Chroma classifier, the same discussion as in section 5.4 holds true as the results are almost identical, with all the complications that arises with regards to the root mean square measurement. For the self-similarity, we see very high values in all cases, with the lowest once again being the corpora containing sonata form repetitions.

In summary, the Relative GMM Chroma classifier is much better than the Absolute GMM and SOM Chroma classifiers when it comes to generating and accumulating peaks, suggesting a high degree usability, especially for $K = 100$. Due to the issues presented in previous sections regarding the root mean square implementation for chroma, it's however difficult to evaluate the quality of this output, and further evaluation in terms of developing better measurements or listening tests, which in either case would be out of the scope for this thesis, would be needed. What we can see is however that the Relative GMM Chroma classifier due to its high usability in multiple contexts could serve as a good starting point for experimenting in unknown contexts, where if problems with the quality of the output arises despite a good number of peaks, changing to Absolute GMM or SOM Chroma classifiers could serve as an option.

5.6 Corpora

So far, little focus has been on the corpora used in the evaluation, despite their importance for the results. Many of the classifiers have been designed around the idea of genre agnosticism, or at least around an idea that it the classifier shouldn't be limited to tonal music (by for example using chroma instead of chord analysis), but such an idea is difficult to evaluate with a small selection of corpora. Of the corpora used for evaluation, the only atonal corpus is Schönberg's Drei Klavierstücke, Op. 11 for piano, which, as we can see in appendix C is overrepresented in many of the worst cases. There's obviously no point in trying to draw any statistical inferences from a single sample, which was never the point with this study, especially not for a concept such as "atonality" which ranges from dodecaphony to noise music and everything in between, but can in this case merely serve an indication whether a classifier can find similarities in very disparate pieces.

On the other side, we see many tonal corpora, particularly the pairs Mozart-Palestrina and Brahms-Palestrina overrepresented in many of the best cases, which could suggest a bias towards the other end of the tonality spectrum. But to evaluate this, a much larger set of corpora, preferably labelled by genre, would be required. The reason for choosing a small set of corpora was (a) that this was never intended as a large-scale quantitative study but a pre-study for determining the scope of the system and (b) it wouldn't technically possible with the current implementation within the given time constraints. Previous tests of the real-time architecture indicate that an influence-generate cycle can be handled within a worst case of 10 ms. Given 7 corpora with an average length of 5988 influences, we get a worst evaluation time of around 48 minutes per classifier. With a sample size of 100 corpora with the same length, the worst case evaluation time would be 166 hours or close to 7 days per classifier. In other words, while the system currently works well for real-time generation and offline generation of a few corpora, it would need a lot of optimization (and likely a change of programming language) to fully support large-scale quantitative evaluation.

Chapter 6

Conclusion

This thesis presented a number of changes to the Somax framework, a system originally designed for human-machine real-time improvisation, but with this work extending the domain of the system towards composition as well. The most important changes presented in this thesis are:

- A new procedure for constructing and labelling corpora, where the old system's static construction and labelling has been replaced with a modular trait (feature) analysis framework and a modular classification framework. This allows the system to handle the labelling and clustering dynamically at runtime so that it can be parametrically controlled by the user. Another purpose of the trait analysis framework is to abstract the data to a level where the runtime framework can operate independently of the format (midi/audio) from which the corpus was originally constructed.
- The module that handles the timing and scheduling of the system has been redesigned and separated into a real-time component, designed for human-machine improvisation, and an offline component, designed for offline generation of entire compositions. The latter can be used with the `Generator` module where the user can define a configuration of the system and control the form of the composition by controlling the parameters of the system over time.
- Based on the latter module, a module for evaluation was designed, which allows the user to gather statistics about the performance of a configuration, a set of corpora and/or a classifier, in terms of usability, quantified as a number of measurements related to peak generation in the system, and quality of the output, quantified as the differences in terms of traits between the input and output of the system.

To evaluate the performance of the system, a number of new classifiers were implemented, using the new modular classification framework. A semi-quantitative study was carried out using the evaluation module based on a small set of corpora intended to cover a number of different genres and formats, and the evaluation was performed on each of the classifiers in the system. The results indicate that of the two pitch classifiers, Top Note classifier and Pitch Class classifier, the Top Note classifier is quite sensitive to the context it's being used in, in terms of usability, while the Pitch Class classifier performs rather well in terms of usability in most contexts. In terms of quality, both classifiers had a surprisingly large differences between the input and the output in terms of trait values. For the three chroma classifiers, we see that the new Absolute GMM Chroma classifier outperforms the original SOM Chroma classifier in terms of usability in most contexts, but they are both quite context-sensitive. In contexts where the Absolute GMM classifier performs poorly, the Relative GMM classifier seems to be a good option. Moreover, both the Absolute GMM classifier and the Relative GMM classifiers are unlike the SOM Chroma classifier parametric, which allows the user to further tune the clustering algorithm in real-time. In terms of quality, the results show little difference between the different classifiers as well as between different cluster sizes within each parametric classifier, indicating problems with the measurement used. Therefore few quantitative conclusions can be drawn from the results and further studies will be required to evaluate this aspect.

One of the main limitations of the system is that it doesn't fully support audio in its current state, and the system is therefore only in theory format agnostic. For future work, this would be one of the top priorities. Another key issue is that the system would require a large amount of optimization to be able perform a truly quantitative study. But what's presented here is in a very early stage of development. It's a framework with a highly modular design but with very little content in each of the modules. Fortunately, this framework will serve as a basis for the ongoing MERCI[1] project for the next three years, and there will as such be plenty of time implement the missing components and fill the modules with content. With this, the framework can hopefully grow from its current state to a truly useful tool for human-machine improvisation as well as composition.

[1] https://www.ircam.fr/projects/pages/merci/

Bibliography

[1] Andrea Agostini and Daniele Ghisi. “Bach: An environment for computer-aided composition in max”. In: ICMC. 2012.

[2] Gérard Assayag et al. “Computer-assisted composition at IRCAM: From PatchWork to OpenMusic”. In: Computer Music Journal 23.3 (1999), pp. 59–72.

[3] Gérard Assayag et al. “Omax brothers: a dynamic topology of agents for improvization learning”. In: Proceedings of the 1st ACM workshop on Audio and music computing multimedia. 2006, pp. 125–132.

[4] Christopher M Bishop. Pattern recognition and machine learning. springer, 2006.

[5] Sebastian Böck et al. “Madmom: A new python audio and music signal processing library”. In: Proceedings of the 24th ACM international conference on Multimedia. 2016, pp. 1174–1178.

[6] Laurent Bonnasse-Gahot. “An update on the SOMax project”. In: Ircam-STMS, Tech. Rep (2014).

[7] Joakim Borg. “Somax 2: A Real-time Framework for Human-Machine Improvisation”. In: Internal Report - Aalborg University Copenhagen (2019).

[8] Jean Bresson, Carlos Agon, and Gérard Assayag. “OpenMusic: visual programming environment for music composition, analysis and research”. In: Proceedings of the 19th ACM international conference on Multimedia. 2011, pp. 743–746.

[9] Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation. Vol. 10. Springer, 2019.

[10] David Cope. “Experiments in musical intelligence (EMI): Non-linear linguistic-based composition”. In: Journal of New Music Research 18.1-2 (1989), pp. 117–139.

[11] Cycling74 - Max. `https://cycling74.com/products/max/`. Accessed: 2020-05-24.

[12] Tuomas Eerola and Petri Toiviainen. “MIDI toolbox: MATLAB tools for music research”. In: Department of Music, University of Jyvaskyla (2004).

[13] Daniel PW Ellis. “Beat tracking by dynamic programming”. In: Journal of New Music Research 36.1 (2007), pp. 51–60.

[14] Alexandre RJ François, Isaac Schankler, and Elaine Chew. “Mimi4x: An interactive audio-visual installation for high-level structural improvisation”. In: International Journal of Arts and Technology 6.2 (2013), pp. 138–151.

[15] Peter Grosche and Meinard Muller. “Extracting predominant local pulse information from music recordings”. In: IEEE Transactions on Audio, Speech, and Language Processing 19.6 (2010), pp. 1688–1701.

[16] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. “Deepbach: a steerable model for bach chorales generation”. In: Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org. 2017, pp. 1362–1371.

[17] Dorien Herremans, Ching-Hua Chuan, and Elaine Chew. “A functional taxonomy of music generation systems”. In: ACM Computing Surveys (CSUR) 50.5 (2017), pp. 1–30.

[18] Lejaren A Hiller Jr and Leonard M Isaacson. “Musical composition with a high speed digital computer”. In: Audio Engineering Society Convention 9. Audio Engineering Society. 1957.

[19] Cheng-Zhi Anna Huang et al. “Music transformer: Generating music with long-term structure”. In: arXiv preprint arXiv:1809.04281v3 (2018).

[20] ISMIR. `https://ismir.net/`. Accessed: 2020-05-24.

[21] Benjamin Lévy, Georges Bloch, and Gérard Assayag. “OMaxist dialectics”. In: New Interfaces for Musical Expression. Ann Arbor, United States, 2012, pp. 137–140.

[22] George E Lewis. “Too many notes: Computers, complexity and culture in voyager”. In: Leonardo Music Journal (2000), pp. 33–39.

[23] Magenta. `https://magenta.tensorflow.org/`. Accessed: 2020-05-24.

[24] Brian McFee et al. “librosa: Audio and music signal analysis in python”. In: Proceedings of the 14th python in science conference. Vol. 8. 2015.

[25] MIREX. `https://www.music-ir.org/mirex/`. Accessed: 2020-05-24.

[26] Gerhard Nierhaus. Algorithmic composition: paradigms of automated music generation. Springer Science & Business Media, 2009.

[27] Jérôme Nika. “Guiding human-computer music improvisation: introducing authoring and control with temporal scenarios”. PhD thesis. Paris 6, 2016.

[28] Jérôme Nika et al. “DYCI2 agents: merging the "free", "reactive", and "scenario-based" music generation paradigms”. In: International Computer Music Conference. Shangai, China, 2017.

[29] Jérôme Nika et al. “Guided improvisation as dynamic calls to an offline model”. In: Sound and Music Computing (SMC). Maynooth, Ireland, 2015.

[30] Adam Roberts et al. “A hierarchical latent vector model for learning long-term structure in music”. In: arXiv preprint arXiv:1803.05428 (2018).

[31] Justin Salamon and Emilia Gómez. “Melody extraction from polyphonic music signals using pitch contour characteristics”. In: IEEE Transactions on Audio, Speech, and Language Processing 20.6 (2012), pp. 1759–1770.

[32] Bob L Sturm et al. “Music transcription modelling and composition using deep learning”. In: arXiv preprint arXiv:1604.08723 (2016).

[33] Greg Surges and Shlomo Dubnov. “Feature selection and composition using PyOracle”. In: Ninth artificial intelligence and interactive digital entertainment conference. 2013.

[34] Sergios Theodoridis and Konstantinos Koutroumbas. Pattern recognition. Elsevier, 2003.

[35] Iannis Xenakis. Formalized music: thought and mathematics in composition. 6. Pendragon Press, 1992.

Appendix A

Gaussian Mixture Models

A Gaussian Mixture Model (GMM) is method for modelling an unknown distribution as a linear combination of weighted, normal distributed probability density functions (for more details, see for example [4] or [34]). Several `Classifiers` in this work utilizes some sort of GMM for clustering as well as classification. This appendix describes the procedure for clustering and classification using a GMM.

A.0.1 Clustering

Given a data set $\boldsymbol{X} \in \mathbb{R}^{N \times M}$ with N samples and M features, i.e.

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}, \quad \boldsymbol{x}_n \in R^{M \times 1}, \tag{A.1}$$

a clustering $\Theta = \left\{ C^{(1)}, \ldots C^{(K)} \right\}$ consisting of K clusters $C^{(k)}$, where each cluster is modelled as a weighted gaussian probability density function

$$p^{(k)}(\boldsymbol{x}) = \pi^{(k)} \mathcal{N} \left(\boldsymbol{x} \mid \bar{\boldsymbol{x}}^{(k)}, \boldsymbol{S}^{(k)} \right), \tag{A.2}$$

$\pi^{(k)}$ is a weight specific for cluster $C^{(k)}$, and $\mathcal{N}$ the multivariate gaussian probability density function determined the mean $\bar{\boldsymbol{x}}^{(k)}$ and covariance matrix $\boldsymbol{S}^{(k)}$ of the vectors $\{\boldsymbol{x}\}^{(k)}$ assigned to cluster $C^{(k)}$, the clustering is performed over I iterations or until convergence.

Initialization

Each vector $\boldsymbol{x} \in \boldsymbol{X}$ is initially assigned to a cluster (the method for initial assignment is described in each classifier), from here on denoted as $C_i^{(k)}$ where the subscript denotes the current iteration $i = 0, \dots, I-1$. The initial weights $\boldsymbol{\pi}_0 = \left[\pi_0^{(1)} \quad \dots \pi_0^{(K)}\right]^T$ are computed so that

$$\boldsymbol{\pi}_0 = \frac{1}{N}\left[|C_0^{(1)}| \quad \dots \quad |C_0^{(k)}|\right] \tag{A.3}$$

where $|C^{(k)}|$ denotes the number of vectors assigned to cluster $C^{(k)}$, $\bar{\boldsymbol{X}}_0$ denoting the matrix of means of each cluster so that

$$\bar{\boldsymbol{X}}_0 = \begin{bmatrix} \bar{\boldsymbol{x}}_0^{(1)} \\ \vdots \\ \bar{\boldsymbol{x}}_0^{(K)} \end{bmatrix}, \quad \bar{\boldsymbol{X}}_0 \in \mathbb{R}^{K\times M} \tag{A.4}$$

where

$$\bar{\boldsymbol{x}}_0^{(k)} = \frac{1}{|C_0^{(k)}|}\sum_{\boldsymbol{x}\in C_0^{(k)}} \boldsymbol{x} \tag{A.5}$$

and

$$\boldsymbol{S}_0^{(k)} = \operatorname{cov}\left(\{\boldsymbol{x}\}^{(k)}\right), \quad \boldsymbol{S}_0^{(k)} \in \mathbb{R}^{M\times M}. \tag{A.6}$$

Iteration: EM-algorithm

Once $\boldsymbol{\pi}_0$, $\bar{\boldsymbol{X}}_0$ and $\boldsymbol{S}_0^{(k)}$ have been estimated, the expectation-maximization algorithm is used to find the maximum a posteriori estimates for $\boldsymbol{\pi}_I$, $\bar{\boldsymbol{X}}_I$ and $\boldsymbol{S}_I^{(k)}$, iterating over I steps or until convergence. The two steps in the algorithm are:

E-step: Calculate the probability that $\boldsymbol{x}$ belongs to cluster $C_i^{(k)}$ at iteration $i \in [1, I]$ for all $\boldsymbol{x} \in \boldsymbol{X}$:

$$\boldsymbol{P}_i = \left[\boldsymbol{p}_i^{(1)} \quad \dots \quad \boldsymbol{p}_i^{(K)}\right] = \begin{bmatrix} p\left(C_i^{(1)} \mid \boldsymbol{x}_1\right) & \dots & p\left(C_i^{(K)} \mid \boldsymbol{x}_1\right) \\ \vdots & \vdots & \vdots \\ p\left(C_i^{(1)} \mid \boldsymbol{x}_N\right) & \dots & p\left(C_i^{(K)} \mid \boldsymbol{x}_N\right) \end{bmatrix}, \tag{A.7}$$

where $\boldsymbol{P}_i \in \mathbb{R}^{N\times K}$ is the posteriori matrix at iteration i and

$$p\left(C_i^{(k)} \mid \boldsymbol{x}_n\right) = \pi_{i-1}^{(k)}\mathcal{N}\left(\boldsymbol{x}_n \mid \bar{\boldsymbol{x}}_{i-1}^{(k)}, \boldsymbol{S}_{i-1}^{(k)}\right), \tag{A.8}$$

where $\mathcal{N}\left(\boldsymbol{x} \mid \bar{\boldsymbol{x}}^{(k)}, \boldsymbol{S}^{(k)}\right) \in \mathbb{R}_{[0,1]}$ denotes the multivariate gaussian

$$\mathcal{N}\left(\boldsymbol{x} \mid \bar{\boldsymbol{x}}^{(k)}, \boldsymbol{S}^{(k)}\right) = \frac{1}{\sqrt{(2\pi)^M \left|\boldsymbol{S}^{(k)}\right|}} \exp\left[-\frac{1}{2}\left(\boldsymbol{x} - \bar{\boldsymbol{x}}^{(k)}\right)^T \left(\boldsymbol{S}^{(k)}\right)^{-1}\left(\boldsymbol{x} - \bar{\boldsymbol{x}}^{(k)}\right)\right]. \tag{A.9}$$

M-step: Calculate weights, means and covariance matrices for step i. Note that unlike in the initialization step, these values are estimated from the soft-count $\boldsymbol{\eta}_i$ of each cluster, defined as

$$\boldsymbol{\eta}_i = \begin{bmatrix} \eta_i^{(1)} & \dots & \eta_i^{(K)} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \dots & 1 \end{bmatrix}}_{N} \boldsymbol{P}_i, \quad \boldsymbol{\eta}_i \in \mathbb{R}^{1 \times K} \tag{A.10}$$

instead of the hard count $\left| C^{(k)} \right|$ used in the initialization. We get

$$\boldsymbol{\pi}_i = \frac{1}{N} \boldsymbol{\eta}_i, \quad \boldsymbol{\pi}_i \in \mathbb{R}^{1 \times K}, \tag{A.11}$$

$$\bar{\boldsymbol{X}}_i = \boldsymbol{P}_i^T \boldsymbol{X} \operatorname{diag}_K^{-1}(\boldsymbol{\eta}_i), \quad \bar{\boldsymbol{X}}_i \in \mathbb{R}^{K \times M}, \tag{A.12}$$

where $\operatorname{diag}_K^{-1}(\boldsymbol{v})$ denotes the element-wise inversion of the diagonal $K \times K$ matrix constructed from vector $\boldsymbol{v}$, and

$$\boldsymbol{S}_i^{(k)} = \frac{1}{\eta_i^{(k)}} \left(\operatorname{diag}_N \left(\boldsymbol{p}_i^{(k)} \right) \boldsymbol{D}_i^{(k)} \right)^T \boldsymbol{D}_i^{(k)} \tag{A.13}$$

where

$$\boldsymbol{D}_i^{(k)} = \boldsymbol{X} - \underbrace{\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}}_{N} \boldsymbol{x}_i^{(k)}, \quad \boldsymbol{D}_i^{(k)} \in \mathbb{R}^{N \times M}. \tag{A.14}$$

Convergence: The EM-algorithm is as mentioned iterated over for I steps or until convergence, which is defined as

$$|E_i - E_{i-1}| < \varepsilon \tag{A.15}$$

for some tolerance ε, where $E_i \in \mathbb{R}$ denotes the accumulated log likelihood over $\boldsymbol{X}$, at step i, i.e.

$$E_i = \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_i^{(k)} \mathcal{N} \left(\boldsymbol{x}_n \mid \bar{\boldsymbol{x}}_i^{(k)}, \boldsymbol{S}_i^{(k)} \right) \right). \tag{A.16}$$

A.0.2 Classification

Given the final weights $\boldsymbol{\pi}_I$, means $\bar{\boldsymbol{X}}_I$ and covariance matrices $\boldsymbol{S}_I^{(k)}$, $k = 1, \dots, K$, a vector $\boldsymbol{x} \in \mathbb{R}^{M \times 1}$ is classified as belonging to cluster $C_I^{(j)}$, $j \in [1, K]$ if

$$j = \underset{k \in 1, \dots K}{\operatorname{argmax}} \, p \left(C_I^{(k)} \mid \boldsymbol{x} \right). \tag{A.17}$$

Appendix B

Evaluation Corpora

Index	**Composer**	**Title**	**Excerpt**	**Year**	**Instrumentation**	**Length**
(f)	Palestrina	Missa Papae Marcelli	all mvmt:s	1562(?)	Choir (SSAATBB)	3147
(e)	Mozart	Symphony No. 36 In C Major, K. 425	all mvmt:s	1783	Orchestra (0200 2200 timp str)	7654
(a)	Brahms	Symphony No. 3 In F Major, Op. 90	mvmt 1 + 2	1883	Orchestra (2222+cfg 4230 timp str)	7040
(c)	Franck	Sonata in A Major for Violin and Piano	all mvmt:s	1886	Chamber (Violin & Piano)	6904
(b)	Debussy	String Quartet in G Minor	all mvmt:s	1893	Chamber (String Quartet)	7966
(g)	Schönberg	Drei Klavier-stücke, Op. 11	all mvmt:s	1909	Solo (Piano)	3302
(d)	Jarret	The Köln Concert	all	1975	Solo (Piano)	5900

Table B.1: The corpora used for evaluation. The "Excerpt" column indicates which movements were used for the evaluation if not all of the movements were used. The "Length" column indicates the number of slices in the corpus.

Appendix C

Detailed Evaluation Results

classifier	bar	$\bar{x}$	s	source	influence
PT	min	0.0934	0.3545	(a)	(f)
	mean	0.5741	1.9272	(n/a)	(n/a)
	max	1.9297	8.4429	(f)	(g)
P12	min	0.0278	0.1776	(a)	(f)
	mean	0.4012	0.8844	(n/a)	(n/a)
	max	0.9072	1.7257	(b)	(d)
CSOM	min	0.0181	0.1404	(a)	(f)
	mean	0.7091	2.6005	(n/a)	(n/a)
	max	2.9855	21.5295	(f)	(b)
AGMM50	min	0.0194	0.1638	(a)	(f)
	mean	0.4921	1.3481	(n/a)	(n/a)
	max	1.2062	2.8731	(f)	(g)
AGMM100	min	0.0281	0.1821	(a)	(f)
	mean	0.4948	1.4086	(n/a)	(n/a)
	max	1.5973	8.4021	(f)	(b)
AGMM150	min	0.0376	0.2192	(a)	(f)
	mean	0.5389	1.5149	(n/a)	(n/a)
	max	1.6147	8.9232	(f)	(b)
RGMM50	min	0.0331	0.1827	(a)	(f)
	mean	0.4628	0.9839	(n/a)	(n/a)
	max	1.3052	3.6474	(b)	(c)
RGMM100	min	0.0338	0.1931	(a)	(f)
	mean	0.4915	0.9581	(n/a)	(n/a)
	max	1.0597	1.7175	(e)	(b)
RGMM150	min	0.0321	0.2039	(a)	(f)
	mean	0.4915	0.9500	(n/a)	(n/a)
	max	1.0093	2.0522	(c)	(b)

Figure C.1: Chain length $\bar{x}^{(\mathrm{len})}$ and its corresponding standard deviation $s^{(\mathrm{len})}$ by classifier with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B.

classifier	bar	$\bar{x}$	s	source	influence
PT	min	93.9803	48.7093	(g)	(f)
	mean	567.0502	534.9195	(n/a)	(n/a)
	max	1365.4058	1483.1599	(e)	(a)
P12	min	241.5438	133.6169	(g)	(f)
	mean	1577.1498	1024.6000	(n/a)	(n/a)
	max	3490.8465	2361.6722	(e)	(a)
CSOM	min	29.4320	39.2903	(g)	(f)
	mean	462.2816	518.5547	(n/a)	(n/a)
	max	1608.9841	2178.8250	(e)	(a)
AGMM50	min	48.3151	82.9972	(g)	(f)
	mean	1370.8170	1256.0073	(n/a)	(n/a)
	max	4147.1343	3856.6018	(e)	(d)
AGMM100	min	35.2957	44.1210	(g)	(f)
	mean	1436.3921	1404.3004	(n/a)	(n/a)
	max	5394.7881	4583.6579	(e)	(g)
AGMM150	min	42.0137	47.5429	(g)	(f)
	mean	781.8305	848.1519	(n/a)	(n/a)
	max	3145.1872	3266.7072	(e)	(a)
RGMM50	min	69.8099	66.0801	(f)	(g)
	mean	907.2567	645.4078	(n/a)	(n/a)
	max	2780.2911	2448.8219	(e)	(d)
RGMM100	min	38.5702	18.1903	(g)	(f)
	mean	482.6139	363.5348	(n/a)	(n/a)
	max	1600.1041	948.2913	(b)	(g)
RGMM150	min	33.7639	33.9166	(f)	(g)
	mean	323.9137	268.5948	(n/a)	(n/a)
	max	1355.0847	1075.2837	(c)	(d)

Figure C.2: Number of peaks $\bar{x}^{(\mathrm{card})}$ and its corresponding standard deviation $s^{(\mathrm{card})}$ by classifier with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B.

classifier	bar	$\bar{x}$	s	source	influence
PT	min	0.3058	0.4574	(f)	(g)
	mean	1.8013	1.3944	(n/a)	(n/a)
	max	3.0500	2.5461	(b)	(a)
P12	min	0.8139	0.7091	(f)	(g)
	mean	2.5850	1.7898	(n/a)	(n/a)
	max	4.7646	3.2218	(a)	(d)
CSOM	min	0.2691	0.5374	(f)	(g)
	mean	2.0445	1.6831	(n/a)	(n/a)
	max	4.1322	2.8051	(a)	(e)
AGMM50	min	0.4273	0.5088	(g)	(f)
	mean	2.7806	2.1070	(n/a)	(n/a)
	max	5.8766	3.6431	(a)	(d)
AGMM100	min	0.3823	0.4702	(f)	(g)
	mean	2.4994	2.0472	(n/a)	(n/a)
	max	5.1716	4.2920	(a)	(d)
AGMM150	min	0.4008	0.5129	(f)	(g)
	mean	2.0848	1.7687	(n/a)	(n/a)
	max	4.1052	3.9278	(a)	(d)
RGMM50	min	0.8143	0.5362	(f)	(g)
	mean	3.4408	2.1776	(n/a)	(n/a)
	max	8.6614	7.6503	(a)	(d)
RGMM100	min	0.7104	0.5402	(f)	(g)
	mean	2.7367	1.7866	(n/a)	(n/a)
	max	6.1864	3.8481	(b)	(d)
RGMM150	min	0.7229	0.5731	(f)	(g)
	mean	2.4610	1.6584	(n/a)	(n/a)
	max	5.4260	4.7234	(a)	(d)

Figure C.3: Selected peak score $\bar{x}^{(\mathrm{mag})}$ and its corresponding standard deviation $s^{(\mathrm{mag})}$ by classifier with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B.

classifier	bar	$\bar{x}$	s	source	influence
PT	min	9.3660	13.5142	(g)	(c)
	mean	57.3204	82.0839	(n/a)	(n/a)
	max	155.7201	152.0477	(d)	(f)
P12	min	36.5241	38.4281	(g)	(e)
	mean	160.4490	160.5064	(n/a)	(n/a)
	max	336.2672	265.6539	(e)	(f)
CSOM	min	4.7036	8.5269	(g)	(b)
	mean	55.0656	93.3381	(n/a)	(n/a)
	max	309.2882	406.9802	(e)	(f)
AGMM50	min	7.9139	24.6414	(g)	(f)
	mean	177.1874	251.2320	(n/a)	(n/a)
	max	842.2050	986.9866	(a)	(f)
AGMM100	min	5.5446	11.2040	(g)	(f)
	mean	182.9100	292.3782	(n/a)	(n/a)
	max	834.9655	1080.9673	(e)	(g)
AGMM150	min	4.1965	10.3206	(g)	(d)
	mean	85.8593	154.1919	(n/a)	(n/a)
	max	299.6466	499.8891	(e)	(a)
RGMM50	min	7.8534	9.5179	(f)	(g)
	mean	150.9822	134.6780	(n/a)	(n/a)
	max	565.4995	356.1842	(a)	(f)
RGMM100	min	5.6432	6.4621	(f)	(g)
	mean	72.8901	68.8736	(n/a)	(n/a)
	max	269.4897	120.9738	(b)	(f)
RGMM150	min	3.6336	4.3155	(f)	(g)
	mean	47.0705	54.1489	(n/a)	(n/a)
	max	221.7073	263.1990	(e)	(f)

Figure C.4: Number of peaks generated $\bar{x}^{(\text{gen})}$ and its corresponding standard deviation $s^{(\text{gen})}$ by classifier with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B.

classifier	bar	η	—	source	influence
PT	min	0.0067	n/a	(d)	(f)
	mean	0.2326	n/a	(n/a)	(n/a)
	max	0.7535	n/a	(f)	(g)
P12	min	0.0001	n/a	(a)	(b)
	mean	0.0430	n/a	(n/a)	(n/a)
	max	0.3759	n/a	(f)	(c)
CSOM	min	0.0496	n/a	(e)	(f)
	mean	0.3135	n/a	(n/a)	(n/a)
	max	0.8310	n/a	(f)	(g)
AGMM50	min	0.0105	n/a	(e)	(f)
	mean	0.1685	n/a	(n/a)	(n/a)
	max	0.6867	n/a	(g)	(f)
AGMM100	min	0.0375	n/a	(d)	(f)
	mean	0.2281	n/a	(n/a)	(n/a)
	max	0.7250	n/a	(f)	(g)
AGMM150	min	0.0740	n/a	(e)	(f)
	mean	0.2949	n/a	(n/a)	(n/a)
	max	0.7299	n/a	(f)	(g)
RGMM50	min	0.0073	n/a	(a)	(f)
	mean	0.0991	n/a	(n/a)	(n/a)
	max	0.2695	n/a	(f)	(g)
RGMM100	min	0.0145	n/a	(b)	(g)
	mean	0.1721	n/a	(n/a)	(n/a)
	max	0.3889	n/a	(a)	(g)
RGMM150	min	0.0686	n/a	(e)	(f)
	mean	0.2136	n/a	(n/a)	(n/a)
	max	0.4561	n/a	(c)	(g)

Figure C.5: Non-generating influence ratio $\eta^{(\mathrm{ng})}$ by classifier with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B.

classifier	bar	$\bar{x}$	s	source	influence
PT	min	0.2336	0.4231	(e)	(f)
	mean	0.5446	0.4775	(n/a)	(n/a)
	max	0.7900	0.4073	(f)	(g)
P12	min	0.1707	0.3762	(e)	(f)
	mean	0.4195	0.4771	(n/a)	(n/a)
	max	0.5997	0.4900	(f)	(c)
CSOM	min	0.2081	0.0830	(e)	(f)
	mean	0.3125	0.1012	(n/a)	(n/a)
	max	0.4279	0.1049	(f)	(g)
AGMM50	min	0.2481	0.0864	(e)	(f)
	mean	0.3666	0.0968	(n/a)	(n/a)
	max	0.4516	0.0852	(f)	(g)
AGMM100	min	0.2397	0.0999	(e)	(f)
	mean	0.3657	0.0998	(n/a)	(n/a)
	max	0.4569	0.0872	(e)	(g)
AGMM150	min	0.2456	0.0939	(e)	(f)
	mean	0.3656	0.1004	(n/a)	(n/a)
	max	0.4511	0.0908	(d)	(g)
RGMM50	min	0.2462	0.1010	(e)	(f)
	mean	0.3659	0.0978	(n/a)	(n/a)
	max	0.4447	0.1241	(f)	(b)
RGMM100	min	0.2436	0.0943	(a)	(f)
	mean	0.3540	0.1000	(n/a)	(n/a)
	max	0.4401	0.0839	(b)	(g)
RGMM150	min	0.2339	0.0969	(e)	(f)
	mean	0.3559	0.1010	(n/a)	(n/a)
	max	0.4333	0.1300	(f)	(b)

Figure C.6: Root mean square $\bar{x}^{(\mathrm{rms})}$ and its corresponding standard deviation $s^{(\mathrm{rms})}$ by classifier with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B.

classifier	bar	η	—	source	influence
PT	min	0.8929	n/a	(a)	(a)
	mean	0.9792	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
P12	min	0.7240	n/a	(e)	(e)
	mean	0.9253	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
CSOM	min	0.8401	n/a	(e)	(e)
	mean	0.9592	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
AGMM50	min	0.9095	n/a	(e)	(e)
	mean	0.9677	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
AGMM100	min	0.7340	n/a	(e)	(e)
	mean	0.9526	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
AGMM150	min	0.9707	n/a	(a)	(a)
	mean	0.9920	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
RGMM50	min	0.9115	n/a	(e)	(e)
	mean	0.9640	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
RGMM100	min	0.9348	n/a	(a)	(a)
	mean	0.9840	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)
RGMM150	min	0.9383	n/a	(e)	(e)
	mean	0.9833	n/a	(n/a)	(n/a)
	max	1.0000	n/a	(b)	(b)

Figure C.7: Self-similarity $\eta^{(\mathrm{id})}$ by classifier over the seven generators $\mathcal{G}_{i,j}$, $i = j$ with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B. self similarity self

classifier	bar	$\bar{x}$	s	source	influence
PT	min	0.0000	n/a	(b)	(b)
	mean	0.0020	0.0270	(n/a)	(n/a)
	max	0.0089	0.0942	(a)	(a)
P12	min	0.0000	n/a	(b)	(b)
	mean	0.0033	0.0346	(n/a)	(n/a)
	max	0.0164	0.1271	(a)	(a)
CSOM	min	0.0000	n/a	(b)	(b)
	mean	0.0045	0.0193	(n/a)	(n/a)
	max	0.0141	0.0494	(e)	(e)
AGMM50	min	0.0000	n/a	(b)	(b)
	mean	0.0058	0.0290	(n/a)	(n/a)
	max	0.0177	0.0789	(d)	(d)
AGMM100	min	0.0000	n/a	(b)	(b)
	mean	0.0046	0.0274	(n/a)	(n/a)
	max	0.0172	0.0700	(e)	(e)
AGMM150	min	0.0000	n/a	(b)	(b)
	mean	0.0015	0.0118	(n/a)	(n/a)
	max	0.0069	0.0436	(a)	(a)
RGMM50	min	0.0000	n/a	(b)	(b)
	mean	0.0066	0.0290	(n/a)	(n/a)
	max	0.0203	0.0777	(a)	(a)
RGMM100	min	0.0000	n/a	(b)	(b)
	mean	0.0015	0.0094	(n/a)	(n/a)
	max	0.0077	0.0398	(a)	(a)
RGMM150	min	0.0000	n/a	(b)	(b)
	mean	0.0011	0.0086	(n/a)	(n/a)
	max	0.0035	0.0221	(e)	(e)

Figure C.8: Measurement of root mean square over the seven generators $\mathcal{G}_{i,j}$, $i = j$ with its mean $\bar{x}$ and its corresponding standard deviation s by classifier with corresponding corpora used for source and influence indicated in the two last columns, where the letter correspond to their indices assigned in appendix B. This measurement is not presented in chapter 4, but briefly discussed in chapter 5.