# On-the-fly Synthesizer Programming with Rule Learning

**Alejandro Iván Paz Ortiz**

Thesis Presented for the
PhD degree in Computing

Soft Computing Research Group at Intelligent Data Science and
Artificial Intelligence Research Center
Computer Sciences Department
Universitat Politècnica de Catalunya - BarcelonaTech
Thesis director: Àngela Nebot and Francisco Mugica

2021

CONACYT

Scholarship Number: 240514

UPC

HANGAR.
ORG

## Abstract

This manuscript explores automatic programming of sound synthesis algorithms within the context of the performative artistic practice known as live coding.

Writing source code in an improvised way to create music or visuals became an instrument the moment affordable computers were able to perform real-time sound synthesis with languages that keep their interpreter running. Ever since, live coding has dealt with real time programming of synthesis algorithms.

For that purpose, one possibility is an algorithm that automatically creates variations out of a few presets selected by the user. However, the need for real-time feedback and the small size of the data sets (which can even be collected mid-performance) are constraints that make existing automatic sound synthesizer programmers and learning algorithms unfeasible. Also, the design of such algorithms is not oriented to create variations of a sound but rather to find the synthesizer parameters that match a given one.

Other approaches create representations of the space of possible sounds, allowing the user to explore it by means of interactive evolution. Even though these systems are exploratory-oriented, they require longer run-times.

This thesis investigates inductive rule learning for *on-the-fly* synthesizer programming. This approach is conceptually different from those found in both synthesizer programming and live coding literature. Rule models offer interpretability and allow working with the parameter values of the synthesis algorithms (even with symbolic data), making preprocessing unnecessary.

RuLer, the proposed learning algorithm, receives a dataset containing user labeled combinations of parameter values of a synthesis algorithm. Among those combinations sharing the same label, it analyses the patterns based on dissimilarity. These patterns are described as an IF-THEN rule model.

The algorithm parameters provide control to define what is considered a pattern. As patterns are the base for inducting new parameter settings, the algorithm parameters control the degree of *consistency* of the inducted settings respect to the original input data.

An algorithm (named FuzzyRuLer) able to extend IF-THEN rules to hyperrectangles, which in turn are used as the cores of membership functions, is presented. The resulting fuzzy rule model creates a map of the entire input feature space. For such a pursuit, the algorithm generalizes the logical rules solving the contradictions by following a maximum volume heuristics.

Across the manuscript it is discussed how, when machine learning algorithms are used as creative tools, glitches, *errors* or inaccuracies produced by the resulting models are sometimes desirable as they might offer novel, unpredictable results.

The evaluation of the algorithms follows two paths. The first focuses on user tests. The second responds to the fact that this work was carried out within the computer science department and is intended to provide a broader, nonspecific domain evaluation of the algorithms performance using extrinsic benchmarks (i.e not belonging to a synthesizer's domain) for cross validation and minority oversampling. In oversampling tasks, using imbalanced datasets, the algorithm yields state-of-the-art results. Moreover, the synthetic points produced are significantly different from those created by the other algorithms and perform (controlled) exploration of more distant regions.

Finally, accompanying the research, various performances, concerts and an album were produced with the algorithms and examples of this thesis. The reviews received and collections where the album has been featured show a positive reception within the community. Together, these evaluations suggest that rule learning is both an effective method and a promising path for further research.

## Resum

Aquest manuscrit explora la programació automàtica d'algorismes de síntesi de so dins del context de la pràctica artística performativa coneguda com a *live coding*.

L'escriptura improvisada de codi font per crear música o visuals es va convertir en un instrument en el moment en què els ordinadors assequibles van poder realitzar síntesis de so en temps real amb llenguatges que mantenien el seu intèrpret en funcionament. D'aleshores ençà, el *live coding* comporta la programació en temps real d'algorismes de síntesi de so.

Per a aquest propòsit, una possibilitat és tenir un algorisme que creï automàticament variacions a partir d'alguns *presets* seleccionats. No obstant, la necessitat de retroalimentació en temps real i la petita mida dels conjunts de dades (que fins i tot es poden recollir al mateix *performance*) són restriccions que fan que els programadors automàtics de sintetitzadors de so i els algorismes d'aprenentatge no siguin factibles d'utilitzar. A més, el disseny d'aquests algorismes no està orientat a crear variacions d'un so, sinó a trobar els paràmetres del sintetitzador que aplicats a l'algorisme de síntesi produeixen un so determinat (*target*).

Altres enfocaments creen representacions de l'espai de sons possibles, per permetre a l'usuari explorar-lo mitjançant l'evolució interactiva. Tot i que aquests sistemes estan orientats a l'exploració, requereixen temps més llargs.

Aquesta tesi investiga l'aprenentatge inductiu de regles per a la programació *on-the-fly* de sintetitzadors. Aquest enfocament és conceptualment diferent dels que es troben a la literatura tant de programació de sintetitzadors com de *live coding*. Els models de regles ofereixen interpretabilitat i permeten treballar amb els valors dels paràmetres dels algorismes de síntesi (fins i tot amb dades simbòliques), fent innecessari el processament previ.

RuLer, l'algorisme d'aprenentatge proposat, rep un conjunt de dades que conté combinacions etiquetades per l'usuari dels valors dels paràmetres d'un algorisme de síntesi. A continuació, analitza els patrons, basats en la dissimilitud, entre les combinacions de cada etiqueta. Aquests patrons es descriuen com un model de regles IF-THEN.

Els paràmetres de l'algorisme proporcionen el control per definir el que es considera un patró. Llavors, com que els patrons són la base per induir nous paràmetres, els paràmetres de l'algorisme controlen el grau de *consistència* dels paràmetres induïts respecte a les dades d'entrada originals.

A continuació, es presenta un algorisme (anomenat FuzzyRuLer) capaç d'estendre les regles IF-THEN a hiperrectangles, que al seu torn s'utilitzen com a nuclis de funcions de pertinença. El model de regles difuses resultant crea un mapa complet de l'espai de la funció d'entrada. Per a aquesta recerca, l'algorisme generalitza les regles lògiques que resolen les contradiccions seguint una heurística de volum màxim.

Al llarg del manuscrit es discuteix com, quan s'utilitzen algorismes d'aprenentatge automàtic com a eines creatives, de vegades són desitjables *glitches*, *errors* o imprecisions produïdes pels models resultants, ja que poden oferir nous resultats imprevisibles.

L'avaluació dels algorismes segueix dos camins. El primer es centra en proves d'usuari. El segon, que respon al fet que aquest treball es va dur a terme dins del departament de ciències de la computació, pretén proporcionar una avaluació més àmplia, no específica d'un domini, del rendiment dels algorismes mitjançant *benchmarks* extrinsecs utilitzats per *cross-validation* i *minority oversampling.* En tasques d'*oversampling*, mitjançant *imbalanced data sets*, l'algorisme proporciona resultats equiparables als de l'estat de l'art. A més, els punts sintètics produïts són significativament diferents als creats pels altres algorismes i realitzen exploracions (controlades) de regions més llunyanes.

Finalment, acompanyant la investigació, es van produir diverses presentacions, concerts i un àlbum amb els algorismes i exemples d'aquesta tesi. Les ressenyes rebudes i les col·leccions on s'ha presentat l'àlbum mostren una bona acollida de la comunitat. Aquestes avaluacions suggereixen que l'aprenentatge de regles és alhora, un mètode eficaç i un camí prometedor per a recerca futura.

## Resumen

Este manuscrito explora la programación automática de algoritmos de síntesis de sonido dentro del contexto de la práctica artística performativa conocida como *live coding*.

La escritura de código fuente de forma improvisada para crear música o imágenes, se convirtió en un instrumento en el momento en que las computadoras asequibles pudieron realizar síntesis de sonido en tiempo real con lenguajes que mantuvieron su intérprete en funcionamiento. Desde entonces, el *live coding* ha implicado la programación en tiempo real de algoritmos de síntesis.

Para ese propósito, una posibilidad es tener un algoritmo que cree automáticamente variaciones a partir de unos pocos *presets* seleccionados. Sin embargo, la necesidad de retroalimentación en tiempo real y el pequeño tamaño de los conjuntos de datos (que incluso pueden recopilarse durante la misma actuación), limitan el uso de los algoritmos existentes, tanto de programación automática de sintetizadores como de aprendizaje de máquina. Además, el diseño de dichos algoritmos no está orientado a crear variaciones de un sonido, sino a encontrar los parámetros del sintetizador que coincidan con un sonido dado.

Otros enfoques crean representaciones del espacio de posibles sonidos, para permitir al usuario explorarlo mediante evolución interactiva. Aunque estos sistemas están orientados a la exploración, requieren tiempos más largos.

Esta tesis investiga el aprendizaje inductivo de reglas para la programación de sintetizadores *on-the-fly*. Este enfoque es conceptualmente diferente de los que se encuentran en la literatura, tanto de programación de sintetizadores como de *live coding*. Los modelos de reglas ofrecen interpretabilidad y permiten trabajar con los valores de los parámetros de los algoritmos de síntesis (incluso con datos simbólicos), haciendo innecesario el preprocesamiento.

RuLer, el algoritmo de aprendizaje propuesto, recibe un conjunto de datos que contiene combinaciones, etiquetadas por el usuario, de valores de parámetros de un algoritmo de síntesis. Luego, analiza los patrones, en función de la disimilitud, entre las combinaciones de cada etiqueta. Estos patrones se describen como un modelo de reglas lógicas IF-THEN.

Los parámetros del algoritmo proporcionan el control para definir qué se considera un patrón. Como los patrones son la base para inducir nuevas configuraciones de parámetros, los parámetros del algoritmo controlan también el grado de *consistencia* de las configuraciones inducidas con respecto a los datos de entrada originales.

Luego, se presenta un algoritmo (llamado FuzzyRuLer) capaz de extender las reglas lógicas tipo IF-THEN a hiperrectángulos, que a su vez se utilizan como núcleos de funciones de pertenencia. El modelo de reglas difusas resultante crea un mapa completo del espacio de las clases de entrada. Para tal fin, el algoritmo generaliza las reglas lógicas resolviendo las contradicciones utilizando una heurística de máximo volumen.

A lo largo del manuscrito se analiza cómo, cuando los algoritmos de aprendizaje automático se utilizan como herramientas creativas, los *glitches*, *errores* o inexactitudes producidas por los modelos resultantes son a veces deseables, ya que pueden ofrecer resultados novedosos e impredecibles.

La evaluación de los algoritmos sigue dos caminos. El primero se centra en pruebas de usuario. El segundo, responde al hecho de que este trabajo se llevó a cabo dentro del departamento de ciencias de la computación y está destinado a proporcionar una evaluación más amplia, no de dominio específica, del rendimiento de los algoritmos utilizando *beanchmarks* extrínsecos para *cross-validation* y *oversampling.* En estas últimas pruebas, utilizando conjuntos de datos no balanceados, el algoritmo produce resultados equiparables a los del estado del arte. Además, los puntos sintéticos producidos son significativamente diferentes de los creados por los otros algoritmos y realizan una exploración (controlada) de regiones más distantes.

Finalmente, acompañando la investigación, realicé diversas presentaciones, conciertos y un álbum utilizando los algoritmos y ejemplos de esta tesis. Las críticas recibidas y las listas donde se ha presentado el álbum muestran una recepción positiva de la comunidad. En conjunto, estas evaluaciones sugieren que el aprendizaje de reglas es al mismo tiempo un método eficaz y un camino prometedor para futuras investigaciones.

## Acknowledgements

There is no such thing as thinking in solitude. Every piece of work involves a series of experiences, exchanges and relationships that remain unreported, especially in scientific, academic writing. True to this unwritten tradition, I mention here only those who, in one way or another, have collaborated in the development of this work. This of course leaves out many of whom I love, but who fortunately have no purpose in reading this work. The text acknowledges the persons almost in order of appearance so it also tells an implicit evolution of my ideas as being collaborating with such amazing humans.

First, I have to mention Sam Roig, not only because I know he would love to appear first, but also, because he is probably more aware of the potentialities of some algorithms presented here, as we have been discussing ideas from the beginning to the 2021 MIRLCa workshop we organized together with TOPLAP_Barcelola and Anna Xambó.

At the forefront, I thank my supervisors Àngela Nebot and Francisco Mugica, because of their special care of the technical formalism and for the freedom to direct my creative explorations, which have allowed for me to forge this work into an equilibrium of technical and artistic balance. I would also like to acknowledge Enrique Romero, whose mathematical thinking was essential for the algorithm discussions. He participated from the very beginning to the last oversampling experiments.

I thank Josep Manuel Berenguer for introducing me to the Electroacustic scene of Barcelona. He was probably the first to listen to my sound experiments and even then, he still encouraged me to continue.

I deeply thank the Hangar people, especially Tere Badia, who supported me during the Artistic Research Residency that resulted in the album Visions of Space. Hangar was also my entry point to an artistic community of free and open principles, many of which are implicit in this work.

I thank Julian Rohrhuber, from whom I learned more from in a month watching him writing code in SuperCollider, than in the last years combined. He also taught me a lot of critical thinking so as not to make the naive mistakes we do as scientists when we engage in artistic practices.

My deepest thanks to Julia Mugica, who sat with me and reviewed the algorithms in those moments (of maybe every work) in which everything seemed senseless and gray. Through those revisions I visualized some light at the end of the tunnel. She has always been my main sonic critic, so I knew something was right every time I saw her smile.

I thank Francis Knights, Pablo Padilla and Dan Tidhar, for the month we spent together in Cambridge thinking about entropy and mathematical models of music. Pablo Padilla together with Jaime Lobato have been discussing computing, music and mathematics with me before this work began so they are in part responsible for it.

A special acknowledgement to Lina Bautista, with whom I started the live coding collective TOPLAP_Barcelona. Living within a live coding community helps a lot when it comes to writing about live coding. She also always reminds me that this is about music and that tools are not necessarily artistic artifacts by themselves.

Within the extended TOPLAP community I want to thank Niklas Reppel, Hernani Villaseñor, Roger Pibernat, Thor Magnusson, Francisco Bernardo, Chris Kiefer, and Shelly Knotts. With them I have written code, papers, and discussed ideas about the possibilities of machine learning within live coding, especially lately with Shelly during our last ICLC paper.

Thanks to David McAndrews and Rebekah Breding for the proof reading of this manuscript, for all the collaborative writing and fun.

I thank the anonymous reviewers for reading this manuscript and help to improve it.

I thank my family Camilo and Amelia (the exception in these acknowledgments) who have always been there.

Finally, I thank the people who participate in the on-the-fly project, especially the Hangar team, Lluis Nacenta, Ludovica Michellin and LinaLab, and the Virtual

Research group, Anne Veinberg, Patrick Borgeat, Luka Frelih, Dare Pejić. Heading a project dedicated to live coding practice and research is probably one of the results that cannot be mentioned in a thesis, but that continually reminds us that live coding has not only a human-machine feedback loop, but also a community one which is probably the one that best contextualizes this practice.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This manuscript investigates the problem of automatic sound synthesizer programming (Yee-King et al., 2018; Macret and Pasquier, 2014; Mitchell, 2012; Stoll, 2014; Tatar et al., 2016) in the specific case where a human improvisationally writes source code in real-time to create music[1]. This activity is known as Live Coding (Collins et al., 2003; Collins, 2011b; Magnusson, 2015). In particular, this problem is approached from the perspective of generative algorithms, that is, when algorithms are used to automatically generate material, in this case: in interaction with a performer.

This pursuit requires investigating sound synthesizer programming, live coding and machine learning algorithms applied to live coding systems (objects that will be described when the time comes).

Around the start of the millennium, personal computers became capable of performing sound synthesis in real time (Dean and McLean, 2018; Collins et al., 2003). This opened new creative and research possibilities for sound programming, which evolved into the technique termed on-the-fly programming, live coding, or conversational programming (Rohrhuber and De Campo, 2009). This technique incorporates the act of programming as an essential part of the running program, as it is written or modified while it is running. For that, the livecoder (the person who is coding) uses the text

[1]Although the algorithms developed are designed to be used in real-time in interaction with a performer, offline applications were also explored.

as the main interface to interact with the running program, usually using interactive programming[2].

As live coding evolved, the new computing developments were incorporated into it. This is the case with machine learning, which experienced a boom almost parallel to live coding and is currently finding its path to integrate into the live coding practice (As an example, see the Sema project from 2019, described in Section 2.4.6).

Generative algorithms have been used in live coding to automate material generation, freeing the performer to prepare code for the sections to follow or control the high level evolution of the performance (Brown and Sorensen, 2009; Magnusson, 2011; Brown and Sorensen, 2007; Collins, 2003).

Preset programming techniques have been explored since the advent of electronic synthesizers (Schroeder, 1962; Schroeder, 1970; Strange and Mumma, 1972). The general problem is, given a synthesizer, finding parameter settings producing desired results, e.g matching a given sound or producing completely novel but interesting results. However, programming sound synthesizers is a complex task, given the huge size of its parameter spaces, the nonlinear response of the sound to changes in the parameter values, and the possible interdependencies among them (Yee-King, 2011b). It has been addressed in literature with two main approaches: automatic and non-automatic.

Non-automatic approaches (Collins, 2002b; Collins, 2002a; Dahlstedt, 2001b; Dahlstedt, 2009) provide representations of the parametric space, which are used as interfaces for its exploration. This is mainly done by interactive evolution, which can be traced back to Dawkins' biomorphs (Dawkins, 1986). Interactive evolution is a good approach for searching for solutions that are not known in advance, i.e. as an exploratory technique that helps us to find interesting sounds that we might not have known existed until we found them (Dahlstedt, 2009).

Automatic approaches (Yee-King et al., 2018; Macret and Pasquier, 2014; Mitchell,

[2]There are some exceptions for instance, during a live coding session held at the Barcelonian Artistic Research Centre **Hangar**, at the end of May 2018, Niklas Reppel performed by iteratively modifying, compiling and excecuting a C program.

2012; Stoll, 2014; Tatar et al., 2016) use feature extraction for instance, Mel-frequency cepstral coefficients (MFCC) feature vectors, which are widely used for timbre instrument recognition (Davis and Mermelstein, 1980; Bhalke et al., 2016; Chakraborty and Parekh, 2018). Then, they use optimization algorithms such as genetic algorithms, neural networks, hill climbers or data-driven approaches, to conduct a search approaching a target. These approaches are dominated by evolutionary computation (Yee-King, 2011b).

However, live coding constrains, such as the need for real-time feedback and the small size of the data sets (which can even be collected mid-performance), make existent automatic sound synthesizer programmers and learning algorithms unfeasible. In the case of automatic approaches, its design is not oriented to create variations of a sound, but rather to find the synthesizer parameters that match a given one. Algorithms using interactive evolution provide an exploratory-oriented approach, however, they require longer times to select the material[3].

Writing an algorithm to automatically produce new material (ideally allowing the control of the novelty-consistency balance), while providing real-time feedback and being able to work with small data sets, motivated the development of the present investigation.

## 1.1 Research Context and Personal Motivation

Before diving into formal descriptions, I will present here a few lines about my personal motivation since I have always found it interesting as a reader. The research questions of this thesis emerged mainly from my experience performing as a livecoder. I started writing code in real-time as a performative and exploratory practice in 2009. My main programming language was (and still is) SuperCollider (McCartney, 1996), originally designed for sound synthesis and algorithmic composition, both performed

[3]There have been some experiments using real time interactive evolution. They are commented in Section 2.1.3.

mainly offline.[4] The amount of code to be written and the restricted amount of time constrain the performer's possibilities. For example, in the *from scratch* practice, the performer starts from a white screen -without written code- and has only nine minutes to play (Villaseñor and Paz, 2020). Throughout the evolution of the live coding practice, higher levels of abstraction, e.g Just in time library (Rohrhuber et al., 2005), began to be implemented for such pursuit. Later, new programming languages, e.g Tidal Cycles (McLean and Wiggins, 2010a), were designed specifically for live coding. Extending the language is another possibility. See, for example, the SuperCollider classes for stochastic synthesis (Collins, 2011a; Luque, 2009). This includes using generative algorithms to automatically produce musical material (Brown and Sorensen, 2009). This process resembles my personal path, constantly looking for tools and strategies to increase the expressiveness of the performance by automating some tasks and reducing the amount of code without losing its readability. For a more detailed description of the live coding practice the reader is referred to (Collins et al., 2003; TOPLAP, 2004a).

The idea of performing through on-the-fly sound synthesizer programming came from the following observation (which is not new, but at which I arrived through live programming synthesizers in SuperCollider): Live coding conducts the sound by real-time intervention of parametric devices (such as synthesizers). Coding a piece on-the-fly requires a bridging of the cognitive gap associated with devices' huge parameter spaces and its possible nonlinear variations. Then, one possible approach is to have some pre-selected parameter combinations, of which the aural result is known, as a starting point for the performance. However, collecting or memorizing many combinations is time consuming, and using only a few can be perceived as monotonous. The algorithms presented in this research seek to contribute in that direction. They provide a tool that allows the performer to start from some pre-selected material (it can be collected in the

[4]The possibility of real-time source code writing as an instrument, as Rohrhuber and De Campo, 2009 point out, "was not self-evident at first; for instance, the moment when the author of the SuperCollider language decided to keep the interpreter running during sound synthesis was not specifically mentioned in the release notes, even though he was certainly aware of the new possibilities."

moment), which is explored during the performance while the piece is unfolding.

The material of the thesis is presented from the creative perspective involving and motivating the development of the algorithms. It focuses on how the algorithms help composers to explore new creative spaces, by learning patterns that can be used in real-time composition and performance. It also engages with post-organic approaches that conceive technology as an extension of the human capacities (Sibilia, 2012)[5]. The text starts with a brief (not exhaustive) historical review of computer music and the use of combinatorics for creative exploration from which the aforementioned perspectives are introduced.[6] When exploring machine learning algorithms in live coding, since it is an emerging field, the text reviews the existing documented examples. Some of the examples do not have as of yet published documentation and are based on personal communications with the authors.

As it will be discussed, the use of machine learning in live coding faces difficulties such as model training or collecting data on-the-fly. Other times, the data sets are small, but not because of the impossibility of collecting data, but rather because they come from a single piece, or maybe because they were collected by a single person for a short period of time to model the composer's personality during that period. In any case, these restrictions challenge the use of existing machine learning algorithms.

We will see that current machine learning approaches range from using pre-trained models (if there are available data) to systems where the learning phase takes place on the stage (using algorithms that produce interesting and quick results for small data sets). In regard to the data collection, the approaches range from those that capture the data on-the-fly to those that use pre-existing databases. The implications of these different approaches will be discussed in turn and its possibilities presented. These

[5]Sibilia discusses the contrasting approaches that conceive technology either as a replacement or as extension of the human capacities.

[6]I chose the use of combinatorics, since, on the one hand, many of the algorithms described produce combinations of the input material, and on the other hand, because it is an iconic example of the use of formal methods in music and is therefore especially helpful to introduce the thesis perspective.

extremes contextualizing machine learning in live coding are discussed in Section 2.4.

## 1.2 Research problems

**Problem 1: A learning algorithm for creating real-time variations in live coding.** Given a set containing labeled settings of a synthesis algorithm, to create real-time variations controlling the *novelty/consistency* degree respect to the original material. For example, being able to recover the original data without modification or to explore more risky variations (the novelty degree is of course restricted within some limits).

**Problem 2: Small data, meaningful parameters and interpretability.** To design the algorithm that creates the new settings in such a way that works with small datasets and be configurable by meaningful parameters. The resulting model has to be interpretable in order to allow real-time human intervention.

**Problem 3: Data types and order independence.** To consider the restrictions imposed by the input data. Concretely, the parameters can be numerical or categorical, and it is not always clear how the parameters determine the qualities of the sound. Therefore, the model has to produce the same output regardless of the order of the features and the input data.

**Problem 4: A system for live performance.** To sketch out a system for live performance based on inductive rule learning.

### 1.2.1 Contribution to knowledge

- An unexplored rule-learning approach for automatic synthesizer programming. Automatic programmers mainly focus on subsymbolic approaches. This research explores the possibilities of symbolic rule learning for synthesizer programming.

- Exploration of inductive rule learning algorithms within live coding. Machine learning is being incorporated into the live coding practice. Nonetheless, as with automatic synthesizer programmers, most algorithms use subsymbolic approaches. Therefore, rule learning has not only not been explored but also provides a view from a symbolic perspective.

- Application of automatic synthesizer programming within live coding. Real-time synthesizer programmers have been implemented in the context of live coding but only as a side product of some other research, so there are no detailed studies around these implementations.

- Design of a system that works in an intermediate level between interactive evolution and sound matching approaches (described respectively in Sections 2.1 and 2.1.2).

- New patterns-based oversampling algorithm (Chapter 5). Oversampling algorithms typically use random processes or probability distributions to calculate synthetic instances. In this case, the patterns found in the data are the ones used to generate new synthetic instances.

- Algorithm that extends non-dimensional IF-THEN rules to n-dimensional regions (Chapter 6). The patterns found through the rule induction algorithm are used to construct fuzzy regions, which offer a complete vision of space.

- Outline of a live coding system based on inductive rule learning. Finally, the algorithms were packed in such a way that that it is possible to perform rule extraction mid-performance (described in Section 4.3.4).

### 1.2.2 Manuscript structure

This manuscript has the following structure: Chapter 2 presents the thesis topic state-of-the-art. It is divided in two main parts. The first part, Section 2.1, presents the research context on synthesizer preset generation. The second part, from Section 2.2 onwards, describes the state of the art of how machine learning is being integrated into live coding. In particular, Section 2.2 is intended to present a historical perspective from computer music to live coding. It also describes how machine learning algorithms are used by the artistic community. Sections 2.1.3 and 2.1.4 describe, respectively, related work on real time preset generation (which is the topic of the thesis) and the problems related to the impossibility of defining aesthetic concepts using mathematical formality.

Chapter 3 leads the reader through symbolic model learning, describing rule learning algorithms, decision trees, rule extraction algorithms from neural nets and neuro-fuzzy systems. At the end of Section 3.5, the rule learning algorithm that inspired the first experiments of this thesis is presented. Chapter 4 introduces the rule learning algorithm (RuLer) proposed in this thesis and presents its evaluation, as well as some possible paths for further research. Chapter 5 analyzes the usefulness of the algorithm of Chapter 4 as an oversampling algorithm and compares it with other oversampling algorithms. Chapter 6 presents a fuzzy extension of the models developed in Chapter 4 and introduces a new evaluation by using fuzzy classifiers. Finally, Chapter 7 presents a general conclusion of the manuscript and discusses some paths for further research. The document finishes with a list of the related publications, performances and recordings developed during this research.

# Chapter 2

# Sound Synthesizer Programming, Live Coding and Machine Learning

This Chapter presents an overview of three out of four conceptual axes of the thesis: sound synthesizers programming, live coding and machine learning within live coding. Throughout its description, the project perspective is also presented. At its center lies the discussion addresing machine learning algorithms being borrowed or designed for artistic practice, in particular for live coding. The missing conceptual axis, rule learning, occupies the entirety of Chapter 3.

## 2.1 Sound Synthesizer Programming

The task of finding parameter settings for a device producing aural results matching ideas in the composer's mind has been addressed in the literature as sound matching or synthesizer preset generation (Mitchell, 2012; Tatar et al., 2016; Yee-King et al., 2018; Collins, 2002b; Dahlstedt, 2009). Its difficulty relates with the huge size of devices' parameter spaces, the possible non-linear perception of the sound, and the unknown interdependencies among parameters (Dahlstedt, 2001b; Dahlstedt, 2007). For example, different settings might be perceived the same, or "similar" ones perceived as very distant sounds. A manual search is extremely time consuming, and, without extensive knowledge of the internal machinery of the synthesis algorithm, it is very difficult to form a mental image of its space.

Methodologies addressing this problem have followed two main approaches:

- **Exploring parameter spaces through interactive evolution.** This approach is more oriented to find new sounds and relies on human evaluation. The systems start from a set of sounds from which the user selects the ones that she likes. Those selected are then processed by genetic operators and the result is evaluated again. The process continues until the user is satisfied with the result. In this way, the system allows the user to navigate the space of possible sounds.

- **Sound matching.** In this approach, the musician already has a sound that she wants to "match" using a synthesizer. The algorithms used receive the target sound together with a sound synthesis algorithm and search for a configuration for the sound synthesis algorithm which causes it to emit a similar sound to the target.

I refer to these methodologies as automatic or sound matching and non-automatic or interactive evolution based, respectively.

Non-automatic methodologies use interactive evolution (Dawkins, 1986; Collins, 2002a; Collins, 2002b; Dahlstedt, 2009) or mapping strategies (Bencina, 2005; Marier, 2012). Essentially, they create a representation of the space of possible sounds that allow the user to interactively explore it. They have the advantage that the output does not have to be known in advance. These approaches are useful when the user searches for new interesting sounds or for variations of a particular sound. Also, they usually represent the space in terms of the parameter values of the synthesis algorithm, which helps the user to follow the search process. However, they require carefully listening to different options, and therefore collect only a handful of possibilities.

Automatic approaches use feature extraction techniques, for instance Mel Frequency Cepstral Coefficients (MFCC) feature vectors (Davis and Mermelstein, 1980) which are widely used for timbre instrument recognition (Bhalke et al., 2016; Chakraborty and Parekh, 2018). Then, they use optimization algorithms (e.g genetic algorithms, neural networks, hill climbers or data-driven approaches) to conduct a search approaching the target. As (Yee-King, 2011b) pointed out, literature on automated sound synthesis programming is dominated by evolutionary computation.

There are also methods that use spectral analysis (Fast Fourier Transform - FFT) on the input, and, based on the analysis, they resynthesize the output. For example, (Serra and Smith, 1990) reconstruct audio signals using particles and noise. It is also worth mentioning that, although human cognition makes it hard to create mathematical or computational representations of the human perception, there are some works on timbre representation, for instance (Esling, Bitton, et al., 2018; Esling et al., 2019; Tatar et al., 2020). These representations allow for the interpolation between sounds, creating sounds perceived between them. For example (Tatar et al., 2020) uses a Deep Learning based synthesis method that allows for interpolation or extrapolation between the timbre of multiple sounds.

Next, some representative examples of systems belonging to sound matching and interactive exploration approaches are described to provide some insight on how they work.

### 2.1.1 Automatic sound synthesizer programmers

#### Genomic

Genomic (Stoll, 2014) is a system that evolves signal processing parameters (sound treatment parameters) to produce novel results using audio samples taken from a database. Genomic uses genetic algorithms (GAs) to evolve the parameters for the sound treatment and the corpus-based synthesis[1] (Schwarz, 2007). The phenotypes are

[1]Corpus-based concatenative methods for sound synthesis use a database of sound samples from which a desired sound is built. The process is guided by a target specified either in terms of sound descriptors or by an example sound.

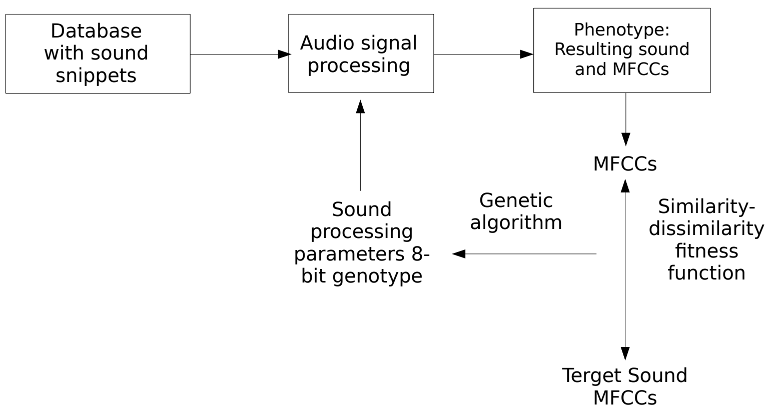
Database with sound snippets
Audio signal processing
Phenotype: Resulting sound and MFCCs
MFCCs
Sound processing parameters 8-bit genotype
Genetic algorithm
Similarity-dissimilarity fitness function
Terget Sound MFCCs

Figure 2.1: Genomic diagram. Sound snippets are transformed by signal processing. The processing parameters are represented as 8-bits genotypes and evolved by a GA. The resulting sounds, together with their MFCCs analysis constitute the phenotype. The fitness function compares the similarity/dissimilarity of the obtained sound with the MFCCs descriptors of the target.

the resulting audios, together with their MFCCs. The fitness function compares the similarity of the candidate sound MFCCs with the MFCCs of the target. Figure 2.1 shows an schematic of the system.

The system was developed within the electroacoustic music context, inspired by the transformation of sound developed in (Wishart, 1986). Genomic performs two parallel processes: One modifies one audio sample to sound more like a target sound, the other modifies a sample to sound dissimilar to the original unprocessed sound. Genomic is conceived as an exploratory tool for composition rather than for "sound matching". As the author referred: " While there might be a perfect solution for a sound transforming seamlessly from source to target, the user may also be interested in intermediate results." Then, the system tracks the transformations as they are explored.

The system evaluation is performed by analyzing the MFCCs of the evolved population and auditioning the results. During the tests, the author tries different options for the fitness function, pointing out that: "If the fitness function is realized as a similarity measure between population members and the target individual, there is a tendency to find a single solution rather quickly that dominates." Moreover, "relying on timbre alone as a metric led to some development of novelty, but more often results in sound results that are inconsistent." To improve the system, the author uses the following approach: instead of considering only the similarity with the target, consider the following three aspects as a fitness function: 1. similarity in timbre (MFCCs), 2. similarity in amplitude envelope data and 3. dissimilarity to the average of the entire active population's timbre measurements.

The system is written in Python and uses SuperCollider (McCartney, 1996) as an audio engine. The author suggests that this architecture provides opportunities for eventual use in real time.

### Sound matching using optimization algorithms

Optimization algorithms can be used to search across a solution space if the target is known. (Yee-King et al., 2018) compared sound matching results using different algorithms. Their working hypothesis was that the deep networks outperform other techniques. They used Dexed (Dexed, 2019), the digital version of the DX7 classic synthesizer. The algorithms considered were a Hill Climber, a Genetic Algorithm (GA) and three deep neural networks: a Multi Layer Preceptron (MLP), a Long Short-Term Memory Network (LSTM) and a bi-directional Long Short Term Memory with highway layers (LSTM++). To compare the algorithms, six sets of sounds with 30 members each were created. The sounds were derived from increasingly complex configurations of Dexed. These were built by freezing parameters of the six operators (oscillators) of the Dexed. Then, the non-frozen parameters were sampled from a uniform distribution in the interval [0,1]. As expected, among the techniques tested, the bidirectional, long short-term memory network with highway layers performed better than any other technique. Considering the total errors over the sis sets, the algorithms were ordered as follows: LTSM++, HC, GA, LSTM and MLP. It is interesting that the simplest programmer, the HC, achieved the second lowest error. Moreover, the authors report

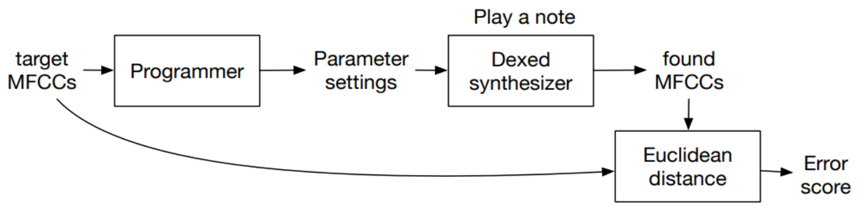

Figure 2.2: Evaluation of a programmer algorithm against a member of a set test. The programmer receives MFCCs as input and generates settings for the synthesizer as output. The resulting sound is rendered and compared to the input sound. Figure from (Yee-King et al., 2018).

that, the LSTM++ was able to match sounds in near real time, once trained. The comparisons among the targets and the candidates were performed by using MFCCs. The error used is the euclidean distance between the sounds. Figure 2.2 shows schematically the evaluation process.

### PresetGen

PresetGen, (Tatar et al., 2016) is a system for preset generation[2] that uses the OP-1 synthesizer (OP-1, 2020). The OP-1 is a semi-deterministic synthesizer with several synthesis blocks. This means that the sound generated by a given preset will be slightly different each time. It also includes effects and low frequency oscillators. For these reasons, the OP-1 is a complex synthesizer not only regarding the size of its parameter space. PresetGen uses a multi-objective Non-dominated Sorting Genetic Algorithm-II that returns a small set of presets that approximate the target best by covering the Pareto front of the multi-objective optimization. The evaluation is performed by comparing the synthesis performed by three human programmers with the synthesis of PresetGen (the comparison is made with Euclidian distance between the audio features of candidate and target sounds and by perceptual sound similarity evaluated by

[2]The authors define the preset generation problem as: "the task of finding preset(s) (i.e. set(s) of synthesizer parameters) that approximates a target sound best."

humans). The authors conclude that PresetGen is human-competitive. PresetGen represents a state of the art in sound matching systems. As the sounds are made with the OP-1 synthesizer, they can theoretically be matched perfectly if the settings can be found. Nonetheless, the problem with using PresetGen in real time is pointed out by (Yee-King et al., 2018), as it “takes 5 hours on a 50 core supercomputer to match a 2 second sound.”

### SynthBot

SynthBoy (Yee-King and Roth, 2008) is a general purpose unsupervised software for programming synthesizers (compatible with any Virtual Studio Technology, VST plug-in). It is designed to find settings for a synthesis algorithm, producing a sound as similar as possible to a given target. It works using a genetic algorithm whose fitness function measures the similarity by the sum of the squared error of the MFCCs between the target and the candidate sounds. The inverse sum squared error between the target and the candidates determines the candidate fitness. The authors suggest that MFCCs are more efficient for timbre similarity than Power Spectra (used in similar research at that time). The system is evaluated technically “to establish its ability to effectively search the space of possible parameter settings” and by musicians that compete with SynthBot to see who is the most competent synthesizer programmer. For this, a target file with random parameter settings was generated 100 times. Then, the optimization process was run with population size of 100 individuals for 100 generations. The authors compared the spectrograms of the target and the produced candidate to assess the system limitations. The experimental evaluation was performed in a two phase experiment in which expert human users competed with SynthBot. In phase one, 10 humans programmed two sounds (using synthesizers mdajx10 and mdaDx10). For each synthesizer, the targets were a real instrument and a sound made with the synthesizer in turn. The SynthBot was given the same task. In phase 2, an online evaluation was carried out. The users rated each of the sounds for similarity to their respective target. SynthBot rated the sounds using its MFCCs error metric. After the evaluation, the authors classified the system as “composer’s assistant”.

### Timbre approaches

Timbre is the set of properties that allow us to distinguish between two instruments playing the same note with the same amplitude. Some new approaches to timbre in sound synthesis (Esling, Bitton, et al., 2018), focus on models of instruments with “static” sound. Therefore, these approaches do not consider some elements of synthesizers, such as low frequency oscillators, which produce dynamic changing sounds over time (sometimes over several minutes).

In (Esling et al., 2019), a methodology is presented that relates the spaces of parameters and audio capabilities of a synthesizer in such a way that the mapping relating those spaces is invertible, encouraging high-level interactions with the synthesizer. The system allows intuitive audio-based preset exploration. The mapping is built so that *“exploring the neighborhood of a preset encoded in the audio space yields similarly sounding patches, yet with largely different parameters.”* As the mapping is invertible, the parameters of a sound found in the audio space are available to create a new preset. The system works using a modification of variational auto-encoders (VAE) (Kingma and Welling, 2013) to structure the information and create the mapping. By using VAE, parametric neural networks can be used to model the encoding and decoding distributions. Moreover, they do not need large datasets to be trained. This system works effectively as an exploratory tool in a similar sense to interactive-evolution based approaches. However, its interface is still oriented to sound matching and exploring rather than to automatically producing variations (it might yet be an interesting feature though). Furthermore, the resulting encodings are difficult to interpret from a human (especially non expert) perspective.

A deep learning based system that allows for interpolation and extrapolation between the timbre of multiple sounds is presented in (Tatar et al., 2020). Deep-learning systems are a promising path for sound synthesis applications, although their training times still do not allow for real-time feedback.

### 2.1.2 Non-automatic or interactive systems

#### MutaSynth

(Dahlstedt, 2001a; Dahlstedt, 2001b) discusses the use of generative processes, particularly interactive evolution, in composition and automatic music creation. Upon these ideas, the author presents the implementation of MutaSynth, an evolutionary interactive system for composition. MutaSynth works using interactive evolution in the following way:

1. It creates a random population of individuals.
2. The composer selects, by audition, the individuals that she "likes" the most.
3. A new population is generated based on the selected individuals with some random variations.
4. Repeat from step 2 (until some terminal criteria is met).

A finer tuning can be obtained by using logarithmic mapping on the parameter values. MutaSynth works with populations of nine genomes. The genetic operations used are: mutation, mating, insemination and morphing. Mutation (controlled by mutation probability and range) and mating (controlled by crossover probability) are the standard GA operations. Insemination is a variation of mating where the amount of genes that are inherited from each parent can be controlled. Morphing is a linear interpolation between two parent sounds. Some other controls are available. For example, MutaSynth offers the possibility of setting a "desirable gene", which will be copied without changes to the next generation. Also, possibilities such as saving genomes, changing genes manually, etc. are available through the user interface.

As a motivation, the author comments on the use of formal methods for music creation. From his perspective, they are generally used to generate content, i.e musical material that, for instance, is used to fill a section in a predefined structure. Some other reasons for using formal methods for music creation can be:

- They are tools for *stepping out* of what one has already created.
- They have the potential to find sounds or music that would not have existed otherwise.
- Formal methods can be used on different levels in the compositional work. In particular (Dahlstedt, 2001a) proposes the follow hierarchy.

    1. Sound design.
    2. Material generation
    3. Structure generation
    4. Generation of large scale form

According to the author, formal methods work best at low levels and structure generation, i.e for tasks from 1 to 3.

Similarly, in Section 4.1.2 it is discussed how "being able of stepping out or extending what someone has already created to find new sounds" is one of the system's goals. Also, ideas on sound design, material generation, and structure generation will be addressed in Section 6.2.

(Dahlstedt, 2001a) reflects on the concept of authorship in cases where algorithms are used to create music. When a piece relies heavily on an algorithm, it is not clear whether the artwork is the piece (an individual instance) or the algorithm (the system). For (Dahlstedt, 2001a), if the algorithm is not published, then the artwork is the piece. However, if the algorithm is published and used to produced several pieces, it becomes the artwork itself. In that case: "the composition is not the piece, but a parameter space of possible pieces". This focus is similar to what is discussed at the end of Section 2.2.1, where some systems can be considered the artwork. Some examples that fit into this category are presented in Section 2.4.

The parameter space of an algorithm can be huge. The non-linear response of perception to changes in the parameters and the interdependencies among them disallow the composer from knowing the result of every parameter setting. For (Dahlstedt, 2001a), interactivity is an interesting way to explore such a complex space. Given that the composer has special preferences depending on her background and the specific context, human interaction, in contrast to pure chance, uses those preferences to zoom in and explore specific “interesting” regions. However, since those preferences/preconceptions may stop the composer from exploring beyond the found regions, combining human audition and chance is a possibility. Interactive evolution is a good combination of human taste and chance.

For Dahlstedt, music history or the life-long learning processes of composers are successful examples of exploration of vastly varying spaces from which amazing pieces of music have been created.

### Interactive evolution of breakbeat sequences, motifs and synthesis parameters

In (Collins, 2002b), interactive evolution is applied to find successful parameter settings for algorithmic composition routines. Specifically, for breakbeat cutting algorithms that re-splice segments of audio drum loops. Collins also experiments with interactive evolution of motifs (short phrases). Motifs “live” in an intermediate hierarchical level between the blocks produced by the BBCutLibrary (Collins, 2002c) -that makes the breakbeat cutting- and the musical phrases.

The individuals evolved are parameter settings of non-deterministic algorithms, so, when applied, the result after running the algorithm is not always the same (as it has some random decision making). As a consequence, during the evaluation process, the composer listens to some of the possibilities of a particular setting. The author suggests that a possibility would be to carry out some statistical analysis over many runs. This is an interesting thought, as, in algorithmic composition, many algorithms do not produce the same static output each time.

As in other cases, to provide the user with more musical affordances and to have control of the evolution process, some functionalities are included to fix some parameter values while evolving the others. This helps with "isolating behaviour". Collins notes that it is difficult to assess how effectively the evolution is being controlled. Especially as he is evolving algorithms that change their output each time they run.

In (Collins, 2002a), results of exploring parameter spaces of synthesis algorithms for sound design through interactive evolution are presented. The research conceptualizes the perspective of algorithmic composition where generative algorithms can be more "messy". As in (Dahlstedt, 2001a), (Collins, 2002a) assigns a range and allowed degree of mutation to the genes. Collins points out that this has more "musical meaning" than just free random mutation. This idea of controlling the mutation range in evolutionary processes will be re-covered in Section 4.1.2, where the "recombination" of the material is controlled by the user as the possible searched patterns are defined. For Collins, "this is a specification of the bounds of the parameter space, and the way in which variation may occur to points in that space under the ... algorithm operations". To provide the user with finer control of the process, the user interface used for the experiments allows auditioning and rating eight candidate parameter sets per generation and provides random presets for the candidates as well as possibilities for changing mutation probabilities, weighted parenting, etc.

### 2.1.3 Related work: on-the-fly synthesizers preset generation

In live coding, sound synthesizers are programmed and/or modified during the performance. Its parameters are the channels for live interaction (Rohrhuber and De Campo, 2009). Possibilities for live programming sound synthesizers, using evolutionary systems, have been suggested in (Dahlstedt, 2009; Yee-King, 2011b). Dahlstedt (Dahlstedt, 2009) proposes possible options for live interaction with an evolutionary system: evolving continuous sound textures or one-shot sounds accepting unpleasant sonic surprises as part of the performance[3]. Yee-King, who has extensive work on

[3]He also suggested that it is possible to evolve sounds using headphones and presenting the results to the audience when ready. He calls these approaches "indeterminate cousins to live coding"

automatic programming methods (Yee-King and Roth, 2008; Yee-King et al., 2018; Roth and Yee-King, 2011; Yee-King, 2011a; Yee-King, 2011b), suggests possibilities that go from on-the-fly adjusting of the genetic algorithm parameters or the weighting used to change its behavior to completely re-programming it on-the-fly. He proposes "a group performance running several different algorithms simultaneously, where each is fed from a different member of the group, then the laptop performer can choose which algorithm(s) is heard based on which is currently displaying the most interesting behaviour." In (Yee-King and Peters, 2011), Yee-King uses live coding control of evolutionary systems and Markov models to track saxophonist-flutist Finn Peters (Collins, 2015) and produce new material. Finally, there are approaches that, once trained, are able to match sounds in real time, such as the bidirectional, long short-term memory network with highway layers described in (Yee-King et al., 2018). The approach presented in this thesis was inspired by these possibilities.

### 2.1.4 What we search for

An important thing to clarify is what we search for when using an algorithm to create new material. Aesthetic searches are problematic to define. This is actually one of the things that makes it difficult to create big datasets or benchmarks. For example, for (Yee-King et al., 2018) interesting behavior might be: "highly dynamic behaviour, producing output which effectively adds to the current mood or highly contrasting output which might lead the overall improvisation in a new direction." In (Dahlstedt, 2009), Dahlstedt writes: "How does one codify what is beautiful, good, or suitable (for the context)? Maybe some ugly music is wanted, to provide contrast, and at another time something beautiful (whatever that means). Sometimes we do not even know what we are looking for." What is clear from these quotes is that what we search for is context dependent. What is desired in a particular moment might not be in another. Both authors agree that it is a creative search that can de directed to some extent. Then, the algorithm accompanies the performer to navigate the space.

In live coding, generative algorithms are used to automatically generate material (Brown and Sorensen, 2009). Many authors mention that providing manageable variations is a desired feature of the algorithms used in algorithmic composition or live coding. For example, (Collins, 2001) says that "Algorithmic composition can help to automate certain processes, and to generate subtle variations on basic beats without breaking with the consistency of a style." In (Sorensen et al., 2014), the authors pointed out that, as live coding manages the musical form through formal methods, at all levels, it requires a delicate balance of coherence and novelty in the produced material.

From this perspective, the algorithms developed, although they can be used asynchronously, are conceived to be used on-the-fly, to perform an exploratory search in the sense described above. The idea is that the performer has some material previously explored and freely classified (labeled). This material is the input of the algorithm. Then, during the performance, the performer uses the algorithm to create new material out of the input data by exploring its possible recombination or extension. If new, interesting instances that are not in the input data are found during the performance, these can be added to the data set. The parameters of the algorithm allow the performer to manage the coherence and novelty of the produced material with respect to the original input data. In such a way, the algorithm allows one to conduct a directed search of the parameter space.

## 2.2 Computing and Music

Computing and music are inherent to humans. Computation precedes the computer. It is an activity for which we developed machines to help us compute (McLean, 2011). The use of formal methods to model sound phenomena dates back to ancient Egypt and Greece, where the first studies on the relationship between the length of a string and the produced tone are found. Music and mathematics relate across time and cultures (Collins, 2018). Let us select, as an example, the use of combinatorics to create many possibilities out of a small set of material, as it relates to the algorithms described later. Although this is an iconic example, similar paths can be followed with emphasis on statistical procedures, geometry, etc.

### 2.2.1 Combinatorics as a creative tool

Combinatorics in music can be traced back to the generative rules of Guido D'Arezzo, in approximately year 1000 B.C. The ideas were probably present before, but, since Guido is the oldest written reference, his Micrologus is normally considered the beginning of generative or algorithmic music. Combinatoric patterns were also present by that time in Maya (250 – 900 AD) architecture as well as in Indian music. The work of Ramon Llull (1232 - 1316) is considered a central piece in combinatoric creativity. He used combinatorics as an epistemic tool by relating different sets of rules to create new knowledge through his "Thinking Machine". It consisted of a complex mechanism of geometrical shapes and symbols that combined letters and concepts to uncover universal relationships. Being born in Mallorca, Llull was a strong connoisseur of the Arab tradition given the social and political contexts of the moment. It is not surprising then, that his Thinking Machine was influenced by the Arab astrological machine the *zairja*, which produced answers by combining sets of concepts (Eco, 2017; Magnusson, 2019). Llull's work inspired Leibnitz's De arte combinatoria (1666).

Since the 13th century, combinatoric dice games were use to generate melodies and harmonies. For that purpose, different pieces (e.g minuets) in the same key were written and then combined by randomly taking parts to create new ones. Haydn and Mozart have been credited with writing works using this technique, although this has been disputed (Nierhaus, 2009).

From 1600 to 1700, combinatorics were conceived as tools to enumerate a space. The scholar Kircher (1602-1680) developed various systems for generating and counting all combinations of a finite set (some of his work is based on Llull). His methods and diagrams are discussed in Ars Magna Sciendi, sive Combinatoria, 1669, and influenced the work of the poet Juana Inés, who also researched the connections between numbers, harmony and geometry. According to (Pareyon et al., 2017), Juana Inés's conceptions such as the use of spirals for harmonic modeling, are still powerful tools for the conceptual study of music. Her treatise, "El Caracol", supposedly containing a musical method based on her conclusion that musical harmony should be conceived as a spiral instead of a circle, was unfortunately lost. Kircher's influence in Juana's work can be found in her Romance 50 (Finley, 2014):

Pues si la Combinatoria,
en que a veces kirkerizo,
en el cálculo no engaña
y no yerra en el guarismo

Translated by Finely as: "However if the combinatorial analysis,/ in which at times I Kircherize, / does not deceive in the sum, / nor does it err in the figure".

In Mersenne's Harmonie universelle (1636), the possibilities of combinatorics are seen as means to enable the composer to explore different versions of a piece within a space of possibilities. Interestingly the focus was placed on the compositional system rather than in the individual pieces (Magnusson, 2019). This conception is closer to those of the systems that we will analyze later.

The described approaches were the basis for the combinatoric techniques used later from Bach to Cage. The creative possibilities of combinatorics have continued to be explored either from the perspective of finite group algebra, as in the case of Estrada's work (Estrada and Gil, 1984), or within the context of self similarity as is the case in (Pareyon, 2011). The continuous relationship between music and mathematics across time formed the basis of algorithmic music thinking. For a detailed review of its origins, the reader is referred to (Collins, 2018), and, for a more general overview of algorithmic composition, to (Roads, Strawn, et al., 1996; Loy, 2007; Nierhaus, 2009; Collins, 2010).

### 2.2.2 Computer Music: Between the automatic composer and the prosthesis

Computer music is the genre in which the computer plays a role in the process of making music in the composition, performance or designing of the sound (Dean, 2009). Sometimes, the term is also applied to the technical discipline that builds tools to be used in music production (for example audio processor design). Computer music has, in its history, experiments that intended to use technology as a replacement of human musicians. This has its origins in devices such as Greek water organs, eolian harps or the musical automatas developed by Vaucanson (Collins, 2018). It is interesting that the authors of the book that contains the first description of an automated instrument (Book of Ingenious devices published in 850 in Baghdad by Banú Músà Brothers: Ahmad, Muhammad and Hasan bin Musa ibn Shakir) worked in the same "House of Wisdom" as the mathematician al-Khwarizmi, from whose name the word algorithm derives (Keislar, 2009). The use of technology with the intention of replacing humans overlaps with the vision that understands technology as a prosthesis that extends human capabilities. We find these two intentions throughout the following different examples, although the later conception is more present in live coding.

The first experiments in computer music were performed as soon as the first computers appeared in the early 1950s (Doornbusch, 2017). Further experiments will eventually point out to what Ada Lovelance (1815-1852) had already visualized: the machine had applications beyond pure calculation. Lovelance wrote the first algorithm for a mechanical machine and described what could be applied to composers such as Xenakis and Nancarrow: the composition of "scientific pieces of music of any degree of complexity".

In 1963, Mathews published the paper "The digital computer as a musical instrument", in which digital sound synthesis is developed. Digital sound synthesis expanded the timbric capabilities of the computers, limited until that moment to beeps and single frequency generators. In his paper, Mathews analyzed previous works that used the computer as a musical device, among them, Hiller and Isaacson pieces from 1957 the Illiac Suite for String Quartet, which can be considered the first "proper" computer generated composition.

The Illiac was composed by writing four different programs with different intentions (Pearce et al., 2002). Their outputs were put together to form a single composition. It was a research project to explore the different possibilities of computers in making music. The first two programs were designed "to demonstrate that standard musical techniques could be handled by computer programming", the third "that computers might be used by contemporary composers to extend present compositional techniques" and the fourth "that computers might be used in highly unusual ways to produce radically different species of music".

(Keislar, 2009) highlights two important aspects of algorithmic composition present in these intentions, which have been extensively developed later: 1) Emulation of traditional styles, for which an algorithm for counterpoint generation was implemented in the Iliac suite. 2) Contemporary composition, in which the algorithms are intended to help the composer's aesthetic exploration.

In imitating musical styles, the most famous software is probably that of David Cope "Experiments in musical intelligence" (1996). For all but specialized listeners, the pieces generated by this system pass the musical Turing test (in (Ariza, 2009), and (Sturm et al., 2019) some criticism about the validity of such tests can be found). In the second category, it is worth mentioning the works of Xenakis, Pareyón, Luque, Morales, and Cage (Xenakis et al., 1987; Pareyon, 2011; Luque, 2009; Soria and Morales-Manzanares, 2013; Pritchett, 1996). The degree in which the algorithms are used in the composition vary. Sometimes complete musical sections are generated (as in the individual algorithms of the Illiac Suite) or even complete musical pieces (as in the Autocousmatic (Collins, 2012), that generates entire electroacustic works). Other times, only small calculations, for example, the generation of low level material, are performed by the algorithms. In "En Casa" (Paz, 2017b), randomly generated numbers are used to set the synthesizer's presets each time the performer changes the section. The algorithms used in computer music resemble the history of artificial intelligence, going from expert systems to subsymbolic approaches, passing by cellular automata, generative grammars (Ames, 1987; Roads, Strawn, et al., 1996; Nierhaus, 2009) to machine learning algorithms such as deep learning (Fiebrink et al., 2016).

Finally, let us discuss some useful distinctions. It is important to distinguish between machines that play music that has been already composed by a human (expressive performance, see for example (De Mantaras and Arcos, 2002)) and machines that create new music, as in the examples described above. Another useful distinction is that of sound-based versus note-based music proposed by (Landy, 2009). Note-based music uses discrete frequencies that can be represented by symbolic notations, such as the score or the MIDI protocol. Note-based music is normally characterized by well defined tonal and harmonic progressions and hierarchies, as well as rhythmic patterns (Dean and McLean, 2018). Sound-based, in contrast, relies less on discrete frequencies and rhythm. It can use either continuous or discrete transitions in the frequency domain for which the importance of the melodies or harmonic progressions is not in the foreground. In contrast, the evolution of the timbre often develops the piece, therefore giving place to deep spectral organizations ((Goldmann, 2015) :pages 22 - 47). In live coding, a note-based example could be Study in Keith (Sorensen, 2019), while a sound based example could be Chain Reaction (Olofsson, 2015).

### 2.2.3 The technological shift : On the fly-programming

Live coding (Collins et al., 2003; Collins, 2011b; Magnusson, 2015) is a performative practice and a creative technique that explores writing computer programs, in real time, in such a way that the process of writing is a part of the running program. The computer programs are algorithms, i.e. sets of instructions to perform a task, that are written and manipulated by humans in real time using interactive programming (Goldin et al., 2006). In this way, during a live coding performance the livecoder maintains a continuous interaction with the program (Dean and McLean, 2018). Also, the code is shown in the venue, so the public can follow the programming process. Live coding is mainly used to create sound, music and visuals, although it extends to other activities (see for example (Sicchio, 2014)). Live coding works in all musical genres, and, despite having preference for computational synthetic aesthetics, it allows for very diverse sounds and visual results (some of the possibilities can be found in (TOPLAP, 2004a)).

Live coding origins can be traced back to the moment when affordable computers were powerful enough to allow modifying programs as they run ((Rohrhuber and De Campo, 2009)). That is, it was a technological shift that allowed exploring the machine capacities and possibilities for creative purposes by interactively writing code. As often happens with technological shifts, these explorations are pointing towards new means of human expression, producing music and visuals that humans could not otherwise have created. Also, the emphasis on aesthetics allows creative elements to contribute to new research questions and technology development (See for example (McLean and Wiggins, 2010b)).

## 2.3 Artificial Intelligence and Music

Artificial intelligence (AI) algorithms have been used for music composition since the origins of AI. Its use resembles the development of AI methods, from expert systems to deep learning. Even before computers, mechanical automatas, combinatorial methods, etc., also resembled the development of algorithmic thinking. These processes are extensively documented in literature, see for example (Collins and d'Escriván, 2017; Herremans et al., 2017; Fernández and Vico, 2013; Miranda, 2013; Roads, 1985).

Machine learning is the branch within AI focused on having machines learn from data to perform tasks without being explicitly programmed, relying on patterns and inference. The use of machine learning for music composition has grown since the beginning of the millennium, and it is also extensively documented in literature. For example, (Sturm et al., 2016) describe the use of deep learning in music modeling and composition, and, in (Briot et al., 2017), a survey on Deep learning techniques for music generation is presented. (Fiebrink et al., 2016) describes supervised learning for composition and performance with real time human interaction, and, in (Fiebrink and Caramiaux, n.d.) the machine learning algorithm as a creative musical tool is discussed. In (Huang et al., 2019), the doodle for automatic harmonization of canons in the style of bach is presented and discussed. However, little has been written about using machine learning in live coding. Here, I present some foreseeing of potentials of the application of machine learning to music creation. Then, in Section 2.4, I discuss some live coding systems using machine learning from the perspective of those possibilities.

### 2.3.1 New instruments

An interesting potentiality of the use of machine learning in music, from the live coding perspective, is the possibility of creating new musical (digital) instruments. An example of this is the work of (Knotts, 2019), discussed in Section 2.4.1. As (Magnusson, 2019) points out, the nineteenth century saw acoustic instruments reach perfection; the twentieth century saw the development of electronic instruments (from the theremin to the modular synthesizers); and the twenty-first century is the century of digital instruments. It is true that digital instruments were around before the beginning of the twenty-first century, but the first digital instruments were intended to emulate electronic ones. Think, for example, of all the Digital Audio Workstations' synthesizers that basically emulated analog synthesis modules. The digital instruments to which (Magnusson, 2019) is referring are conceptually different from digital instruments that emulate electronic ones (e.g an instrument that changes by interacting with the performer, see (Knotts, 2019) described in Section 2.4.1, or instruments that use deep learning to produce non-existent sounds, such as the Nsynth). The Neural Synthesizer "Nsynth" (Engel and Norouzi, 2017) uses deep neural networks to fuse audio samples, e.g. combining the qualities of a bass and a clarinet. However, the sounds are not "mixed" together but rather their underlying qualities are recombined. In both examples, the instruments exploit the algorithmic nature of the computers.

The development of these algorithmic instruments started on a small experimental scale and is now being incorporated in commercial software. For example, the inclusion of MAX patches in Live (Max for Live) (Manzo and Kuhn, 2015) allowed the use of algorithmic patterns to control different parameters of the instruments available at the Digital Audio Workstation.

Digital instruments combined with machine learning techniques offer possibilities such as personalizing them through its use or, in contrast, to train them using multi-user databases to average the users' personalities.

Other possibilities of machine learning instruments have to do with the possibilities of finding hidden behaviors that could result from the interaction of the user with the algorithms, i.e surprising behaviors that no one explicitly programmed, due to for example, non-linearities or particularities in the data acquisition or training process (Fiebrink et al., 2016; Mudd et al., 2015).

### 2.3.2 Machine learning algorithms as creative musical tools

(Fiebrink et al., 2016) explored the use of machine learning algorithms in creative musical contexts. They identify a range of different relationships with the algorithms, from scenarios where the user collects proper data sets, train the models and validates the results by means of accuracy measures to cases in which the musicians break the rules and explore the system's capabilities in unexpected ways. They suggest that, in the latter scenarios, systems evaluations (validation or usefulness) are not necessarily based in accuracy metrics anymore. An interesting reflection along the same line is carried out in (Sturm et al., 2019), where it is questioned up to what point quantitative measures, like sequence likelihoods and/or qualitative listening tests (user tests), are useful for machine learning applied to music research. (Fiebrink et al., 2016) look at machine learning algorithms as human-computer interfaces that can be characterized, as in the case of other interfaces, by the way their affordances intersect with the objectives of the human users. Based on these observations, they center the discussion by analyzing the nature of the interactions between humans and machine learning algorithms from this human-centered perspective. Among the roles of machine learning they identify: Creating models that act as creative agents with human-like capacities (which has been the must spread idea of machine learning in music), generating musical material in real time (with different degrees of novelty), and augmenting human capacities. These last objectives range from playing with the model trained as an "imperfect mirror" (Pachet, 2008) to using the algorithm to "not repeat myself" during live coding performances ((Knotts, 2019) see the CYOF system in Section 2.4.1).

### 2.3.3 Machine listening in live coding

Machine listening is about designing algorithms and systems to endow machines with audio understanding. This requires: receiving the audio signals, extracting high-level representations and analyzing them. This process is similar to human audition, which is not only about perception but also about structuring and processing the information (even through conceptual listening frames).

Machine listening technologies started in the 60's. During the 80's they were included as tools in standard computer software (for audio analysis). Between the 90's and the year 2000, the increase in computer power that allowed sound synthesis in real-time for live coding allowed personal computers to use machine listening in real time too.

(Collins, 2015) explores machine listening possibilities in live coding. In particular, he explores how the acquisition and analysis of the audio input can be controlled through live coding and (what he refers as a more radical possibility) how information can control the code being written. As in the case of machine learning, this area has received less attention in live coding, perhaps because of its technical difficulties. While it is true that simple tasks such as real time monophonic pitch detection are successfully solved by existing algorithms, more complex situations, like real time polyphonic pitch tracking, are still a matter of research.

Real-time machine listening allows live coding performers to interact with the current sound, for example, detecting the pitch to respond in the same key, or detecting the rhythm (onset detection). It is also possible to transform what is being received, for example, resynthesizing the sound by using inverse Fast Fourier Transform. Combining machine learning, machine listening and music information retrieval algorithms, it is possible to analyze patterns at different hierarchical level, detect similarities comparing the input signal with a corpus, etc.

There are still a few examples of machine listening in live coding, nonetheless, and they suggest interesting research paths. For example, (Knotts, 2019) uses signal analysis to assess how likely is for a livecoder to go in one direction or another, comparing the analysis with a database. (Ocelotl, 2016) uses a system, although sometimes more similar to an autonomous agent as it is only slightly tweaked on-the-fly, to receive a cello signal and resynthesize it with some changes. The system is based on the SuperCollider Music Information Retrieval library (Collins, 2011c). (Yee-King and Peters, 2011) use live coding to control the machine listening using SuperCollider. Finally, (Collins, 2015) discusses a feedback loop where the analysis of the input can be used to synthesize the output and so on.

## 2.4 Live Coding and Machine Learning

As I write these lines, live coding systems are incorporating machine learning[4]. There are few documented systems, yet, they provide an overview of the possibilities, for example, to encourage the performer into more innovative improvisations (Knotts, 2019), to automatically create new material for specific contexts (Paz, 2019b), or to learn probabilistic patterns from character strings written on-the-fly (Reppel, n.d.). An in-

[4]Between the first and the current revision of this Chapter, I participated in the 2020 International Conference on Live Coding (ICLC) and in the 2020 Network Music Festival. I witnessed how the number of live coding projects using machine learning increased in comparison with the 2019 ICLC.

teresting approach that performs space exploration using genetic algorithms is discussed in (Dahlstedt, 2009), who interactively evolved populations of presets, auditioning the candidates on stage (without using code as the main interface). These live interactive evolution processes are described by the author as "indeterminate cousins to live coding". Finally, and probably the clearest example of machine learning being integrated into live coding, is "Sema": a playground designed for prototyping live coding mini-languages, which integrates signal synthesis, machine learning and machine listening (MIMIC, 2019b). Sema is part of the Musically Intelligent Machines Interacting Creatively project (MIMIC[5]), which is a web platform for the artistic exploration of musical machine learning and machine listening (Kiefer and Magnusson, 2019).

From a live coding perspective, a key feature characterizing these systems is the way they allow real-time interaction with the different steps of the learning process. To analyze where the real-time feedback occurs, we see if the data collection and the training processes are carried out before or during the performance.

The latest version of Sema, for example, allows the performer either to write and train algorithms in real-time including data collection or to use pre-trained models. In the latter case, it uses TensorFlow.js (Smilkov et al., 2019), whose models are trained over big corpus (this might take days or even weeks). In this case, the models are used but not trained during the performance, and the livecoder does not have access to the data collection. On the other extreme is Reppel's approach for real-time learning of variable order Markov chains (see Reppel, n.d.), in which the data collection and the training (statistical analysis of the data) is intended to be done during the performance. Between these extremes is CYOF (Knotts, 2019), in which previous and current performance data is combined. The different degrees of "liveness" offer different creative possibilities. For example, if the system is already trained, the instrument is "fixed", and the user plays with it during the performance. However, as the training can take more time, it could use bigger data sets and probably still be very accurate.

[5]MIMIC is a collaboration among Goldsmiths College, Durham University and the University of Sussex.

If the system is trained on-the-fly, the training process becomes an integral part of the performance. The resulting model may be clumsy, but sharing the training may result in a more transparent performance. An excellent example of this is Marije Baalman's gig, presented during the MIMIC Artist Summer Workshop in Brighton 2019 (MIMIC, 2019a). Obvious limitations for the systems are: the algorithmic cost, the computational capacity of the machines (which increases continuously) and the limitations in the size of the training sets inherent to the data collection in real time. Next, existing systems are presented, and a general conclusion regarding live coding and machine learning can be found in Section 2.5.

### 2.4.1 CYOF

(Knotts, 2019) describes CYOF as an algorithmic performance system aiming to "gently encourage the performer into more innovative improvisation". For that, the system analyses performer's current and previous gigs data to infer which would be the more novel code to write next and to suggest ideas in real time. The analysis is performed using music information retrieval (MIR), which includes real time audio feature data (e.g MFCCs, loudness, etc.) and text analysis tools. The degree of novelty is measured by comparing the current code with performance's own history. The performer can see the current code in white (as well as the audience, as the code is projected) alongside the most likely (in orange) and the less likely (blue) possible future code. The on-the-fly analysis uses 10 second windows and is carried out by the SuperCollider Music Information Retrieval Live (SCMIRLive) library (Collins, 2011c). The analysis of the data of the past performances is used to generate a data set of likely code combinations, audio feature combinations and performance trajectory.

The system also visualizes the audio feature data. For this, at the beginning of the performance, a random past performance is selected and its data visualized. The visualization uses blocks (resembling bricks) in which the features are represented by using the gray scale. Then, the visualization is updated, as the performance goes, using the current data. Future blocks on the gray scale acquire the color of the most likely future given the present context. A performance of Knotts using the system is available at (Knotts, Dec 2017).

CYOF is interesting because it clearly shows some machine learning possibilities within live coding.

1. A personalized instrument that changes with time as the database is updated. Notice, however, that if we use a past (unchanged) database, the system will be "frozen" in time, which would be like playing guided by our "old self".

2. The system, although is intended to encourage more novel improvisation, can be also used to be conservative if we decide to play the most similar future code each time.

Finally, notice that the model building and the data collection occur both offline and in real time.

### 2.4.2 Learning rules on-the-fly

cross-categorized-seeds (Paz, 2019b) is a live coding system that uses inductive rule learning. It is inspired by the idea that coding a piece on-the-fly requires guiding the sound by changing the parameter settings of the sound devices. The system uses the "RuLer" algorithm (Paz et al., 2019) for supervised learning. To create the dataset (the example input-output pairs), a linguistic label that describes the characteristics of the sound (e.g. calm, harsh) or a musical context (e.g. intro, break) is assigned to each combination (or setting). The settings are collected during an aural exploration of the parameter space, which can be performed offline or in real-time. The RuLer generalizes the input data with the aim of offering different degrees of variation to be used during the performance. For this, it searches for patterns in the data, based on a dissimilarity function. The patterns found are used to "guess" the new setting. The patterns are expressed as IF-THEN rules. The production of rules applies crossover mutation to the original material in a similar way to genetic algorithms. The user controls the level of generalization of the rules through the system parameters that control the allowed dissimilarity and consistency and by listening to the material on-the-fly. The RuLer algorithm forms part of this thesis and is extensively described in Chapter 4.

During the performance, the form of the piece is created by calling the combinations in the corresponding moments using the perceptual labels. The tensions and relaxations created by the recombination levels of the original material can be used to shape the inner dynamic of the sections. Hence, cross-categorized-seeds can be seen as a directed search in a vast space of possibilities, used as a compositional tool.

If the data set is collected on stage, the exploration process becomes part of the performance. In that case, the rule learning process is also performed in real time, for which the user can either use algorithm parameters that search for conservative results or embrace unpleasant surprises.

A performance using the system is available at (Paz, 2019b). An advantage of the resulting model is that the rules are human readable entities, especially if the size of the data set is small enough.

### 2.4.3 Learning probabilistic automata in Megra

Megra (Reppel, n.d.) is a mini-language to make music based on variable-order Markov chains. Among other algorithms, it includes a simplified version of the one proposed in (Ron et al., 1996). The algorithm learns probabilistic automatas with variable memory length from sequences. The Markov processes of variable memory length can be described by a subclass of probabilistic finite automata (which are the resulting models learned by the algorithm). The algorithm is efficient in terms of computing cost, and it exhibits good accuracy. Its authors proved that the KL-divergence between the target distribution and the learned can be made small in polynomial time. In the Megra implementation, the input is a string of characters (e.g xoxxo—-xxo) that is analyzed in real time (the strings are written by the performer on the stage) to learn the probability of each symbol. Then, the probability of each character is mapped to specific samples or synths. The whole process, data collection, learning and the production of the output, is performed in real-time.

### 2.4.4 Cibo

During the fourth International conference on live coding, held in Madrid on January 2019, Cibo (Stewart and Lawson, n.d.), a machine learning algorithm trained to manipulate Tidal (McLean and Wiggins, 2010a) code (i.e read, change, execute and repeat) performed for the first time in front of an audience. Cibo was trained using code from a database of Tidal live coding performances, without having access to any sound information (like signal descriptors). In this way, the authors explore how the changes in the code reflect the way a human performer changes their code from their perception of sound. The second version of the agent, Cibo v2, was presented at the fifth International Conference on Live Coding (Limerick, 2020).

### 2.4.5 Generative improvisation: exploring vast parameter spaces on-the-fly

Evolutionary algorithms can be used as tools to evolve (shaping by means of predefined operations, crossover and mutation being the must common) an initial randomly selected population, until it has the desired characteristics. The initial population is passed through a fitness function (that models the environment) that assigns a score to each individual. Then, the individuals scoring under a specific threshold are eliminated, and those that remain are modified by using the predefined operations. The new resulting population passes again through the fitness function, and the process continues until a terminal criteria is meet.

Evolutionary algorithms have been used by numerous composers (Dahlstedt, 2001b; Collins, 2002a; Dahlstedt, 2009) to explore devices with vast spaces of sonic possibilities, as in the case of modular synthesizers. As it is impossible to codify a fitness function for what the composer is looking for, given that it changes from one situation to another, the composer auditions the results and selects the individuals that will pass to the next generation (i.e, the composer's audition acts as the fitness function). This technique was named interactive evolution by Richard Dawkins (Dawkins, 1986).

To clarify the context dependency of the sound, imagine that some ugly and beautiful musics are wanted to create contrast between two parts (whatever ugly or beautiful music means). While we could try to find a representation in terms of low level audio features of the ugly and beauty categories, each representation would be different. It would vary from person to person, and what is ugly in a context may not be in another, even for the same person.

Furthermore, using composer's audition to guide the evolution brings other possibilities. The fitness function allows genetic algorithms to optimize the characteristics of the population. However, optimization is not the only possibility of evolutionary algorithms within creative applications, they can also be used as exploration tools as will be discussed next.

Evolutionary algorithms allow one to explore spaces of possible solutions, especially when the exact form of the solution is not known or when we do not know what we are looking for. In such cases, exploring the unknown space is essentially a creative act (Dahlstedt, 2009). In that scenario, the goal is not about optimization anymore but searching for novel musical material that we do not now exists until we find it. When interactive evolution is used in this way to direct generative processes, it can be considered a creative tool that operates on a meta generative level.

Interactive evolution has been generally used in the studio (offline). For example, Collins 2002 used interactive evolution for sound design for exploring parameter spaces of reverberation algorithms, wavetable synths and synthesis of percussive sounds. Here, the technique is used to find solutions to projects that would be difficult to solve by simple trial and error sampling of the parameter space. (Johnson, 2003) presents an evolutionary synthesizer that uses a graphic user interface. The artifact is intended to provide "a middle way between the complexity of programming -the synthesizer is implemented in CSound- and the simplicity of using preset sounds". In the publication, the system is applied to adjust the synthesis parameters of the granular synthesizer

Fonde d'Onde Formantique (FOF) algorithm (Vercoe, 1986). Again, the system is intended the be used offline in the context of sound design.

Nonetheless, these applications are conceived as offline exploration. The interactivity of the technique suggests real time applications. These, of course, are limited by the size of the population to be evolved and the time required to audition the examples.

Brown, who is a member of the live coding duo aa-cell together with Sorensen[6], introduced the term "generative improvisation" to describe the experience of live performing using evolutionary systems (Brown, 2002). Playing with evolutionary algorithms combines the feeling of being able to direct the search towards the desired results, but it requires the composer to be able to embrace surprises. Palle Dahlstedt, who has extensively worked with evolutionary systems for sound and music creation, describes his experience and the possibilities of using these algorithms in real time performances in the following way:

"On stage, you can either evolve continuous sound textures or one-shot sounds openly, in interaction with other musicians, accepting unpleasant sonic surprises as part of the game, or the current population can be auditioned through headphones, projecting the most fit sounds to the audience when ready. Another approach tried by the author is to evolve whole pieces in headphones on stage while other people perform, to be played as interludes when ready after a few minutes, as a kind of 'action composing'." (Dahlstedt, 2009).

It is interesting how he visualizes the different degrees in which the evolution process is shown to the audience, from showing nothing when the candidates are evolved using headphones and only shown to the audience when they are completely ready to evolving sounds openly making the evolutionary process part of the performance. In contrast he also comments on 'action composing', which would be closest to offline composition where the algorithms are compose on stage and the results shown when ready. Finally, it

[6]They wrote the paper Interacting with generative music through live coding (Brown and Sorensen, 2009), in which they characterize the algorithm's properties that make them more suitable for live coding.

is also interesting how the human-system interaction suggests to him that the technique may well be considered an "indeterminate cousins to live coding".

### 2.4.6 Sema

Sema is an ecosystem for prototyping live coding mini-languages that integrates sound synthesis, machine learning and machine listening (MIMIC, 2019b; Bernardo et al., n.d.).

Machine listening is carried out by MMLL, a Musical Machine Listening Library (Collins and Knotts, 2019) that offers different musical listening facilities, such as onset detection, major/minor chord detection and beat tracking. The library has listening capacities than operate faster than real time.

Machine learning algorithms and data collection can be either directly coded or live coded by the user, or models from the TensorFlow.js (Smilkov et al., 2019), Rapidlib.js (Zbyszynski et al., 2017) or Magenta.js (Roberts et al., 2018) libraries (Smilkov et al., 2019) can be called. Each choice has its own implications with respect to how the user is involved in the data collection and the model training.

Sema has also a real time compiler for Backus-Naur form grammars that allows one to write or modify the live coding language. If the language is consider the instrument, this is a clear example of an instrument that could change during the performance.

The system is part of MIMIC web platform designed for artistic exploration of machine learning and machine listening for music creation.

The interesting things about the system are its objectives, which show a clear intention to explore the creative possibilities of machine learning and machine listening in live coding. The system is designed to be portable (that's why it runs in the browser) since it is intended to popularize machine learning and machine listening tools for a larger audience.

### 2.4.7 Undocumented systems

As this is an emerging field, there are a lot of systems currently being documented. These incorporate machine learning for different tasks of the live coding systems. Some of these examples were collected via personal communication with their authors during 2019.

Jason Levine is a New York based musician and live coder, that performs navigating audio samples in a virtual space using the live coding language "Extempore" (Sorensen and Gardner, 2017). The samples are organized by the t-SNE algorithm and the resulting structure offers an interpretable space for the coder to navigate.

Pablo Riera is a live coder currently working at the Applied Artificial Intelligence Lab of the Buenos Aires University. He has done some similar experiments using learning algorithms to structure a database containing sounds, which are then controlled and sent to Tidal using a MIDI controller.

These types of approaches are being currently explored, but the systems are still undocumented. Certaintly, the following years will see derivations of these systems and their corresponding documentation.

## 2.5 Conclusion

Writing source code in real time to produce sound mixes both technology and artistic practice. Analysing live coding tools, instruments and technologies provides insight into how they shape the way we structure sound and musical ideas. For example, the source code becomes our notation language. It is the score of the piece, as the algorithms, functions, etc. describe sound unfolding in time. Different languages provide different conceptions of time. For example cyclical, in which a declared pattern repeats every cycle, provides a different conception than say, linear, in which events are conceived with a start, a duration and an end. A deep reflection on how technology shapes our music making can be found in (Magnusson, 2019), which looks at the technologies of material instruments, the symbolic notations of music and the signal inscriptions or recording mediums.

The digital nature of live coding brings new algorithmic possibilities to the core of the new instruments that are emerging alongside new technologies. These are artifacts that change by interacting with the performer, e.g Knotts, 2019, or that are able to perceive and adapt to their external reality by means of machine listening.

At the time of writing these lines, machine learning is being explored within live coding. While most early live coding performances used simple sound generators and processors, such as sine waves, filters, etc., today it is increasingly common to hear performances, even those starting *from scratch* (Villaseñor and Paz, 2020), using machine learning to perform specific tasks, for instance, creating clusters in a music database so that the performer can navigate an ordered space.

The use of machine learning algorithms has implications in live coding practice. For example, real time training v.s offline training implications can be viewed through the lens of the live coding ManifestoDraft (TOPLAP, 2004b), which positioned live coding within the digital arts back in 2004 and has been a reference for live coding practice. Here is a fragment:

We demand:

- Give us access to the performer's mind, to the whole human instrument.
- Obscurantism is dangerous. Show us your screens.
- Live coding is not about tools. Algorithms are thoughts. Chainsaws are tools. That's why algorithms are sometimes harder to notice than chainsaws.

We recognise continuums of interaction and profundity, but prefer:

- Insight into algorithms
- The skillful extemporisation of algorithm as an expressive/impressive display of mental dexterity

Integrating machine learning into live coding raises such questions as how can machine learning algorithms be visualized within live coding? How do training times and/or size and nature of databases influence the design and use of algorithms? To what extent does on-the-fly machine learning exist with the current technology?

It is possible to use algorithms trained over large datasets, such as in melody generation. Sema, for example, is a playground for prototyping live coding mini-languages that integrates signal synthesis, machine learning and machine listening. It allows the use of models from TensorFlow (the end-to-end open source platform for machine learning). The training time of some models can take minutes, hours or days depending on the algorithm, the dataset and the hardware. Once trained, the models can be fairly accurate, rarely surpassed by other systems. This approach reaches philosophical limits, like the one suggested by Collins, 2016 in his paper entitled: "Towards Machine Musicians Who Have Listened to More Music Than Us". Indeed, an algorithm can be trained over corpora of music that would take a human years to listen through.

Some artists have eschewed such large-scale datasets. Niklas Reppel opened his presentation at the fifth International Conference on Live Coding (Limerick 2020) with the question: Why small data? Stating the ideas behind the design of *megra*, a mini-language to make music with variable-order Markov chains (Reppel, 2020) he answered in the following way: Small data is a defined response to the current algorithms, which are mostly focussed on big data sets and specialized hardware, and therefore have long training times that won't fit into the live coding performance. *megra* is designed taking into account everything you can do with tiny datasets having real time feedback.

These approaches define extreme possibilities. Each has to embrace its necessary consequences. However, just as technologies condition our way of making music, live coding, being a well defined practice, is shaping technological developments, designing systems that learn, exploring the limits and possibilities of machine learning algorithms from a creative perspective, analysing how they change through different training datasets and writing new ones that provide desired affordances.

This is an emergent field in which I hope these lines help to conceptualize the algorithms to come.

# Chapter 3

# Rule Learning Preliminaries

This Chapter presents foundations of rule learning algorithms. It begins by reviewing the two main strategies for building symbolic models: decision trees and rule learning systems. For this, the main algorithms for the induction of decision trees are reviewed. Then, after arguing that rule models are more interpretable, it summarizes rule learning from concept learning to the covering algorithm. Then, the Chapter goes through the main algorithms for inductive rule learning with special focus on the LR-FIR and Mixed fuzzy rules algorithms, which inspired the algorithms presented in Chapters 4 and 6.

Symbolic modelling represents problems in a human-readable way. This allows manual intervention of the model or data, just as the genotype representations discussed in the previous Chapter allow manual selection of genes. Symbolic models also provide an insight into the relationships between labels and feature values.

In the context of this research, the extracted models aim to provide the following things: Interpretable tools to automate the creation of new material; a human-readable representation of the relationships between the resulting sound and the parameter settings of a synthesis algorithm; and real-time learning and intervention of the model and data.

Throughout the Chapter, the characteristics and limitations of the specific algorithms are highlighted when they lead to the development of the algorithms described in Chapters 4 and 6. As an example, consider the fact that linguistic variables (used in many fuzzy systems) assume monotony in their values. Variable *age*, for example, can take values such as young, mature or old. However, some synthesizers' variables such as *waveform*, which takes values such as Square, Sinusoidal and Sawtooth, do not satisfy this property. If output values are considered, this phenomenon is more extreme, given that the labels used to described the resulting sound can be arbitrary.

Machine learning and data miming deal with the discovery of models, patterns and other regularities in data. Machine learning approaches can be coarsely divided in two groups:

- Symbolic approaches that include inductive learning of symbolic models such as rules, decision trees and other logical representations (De Raedt, 2008; Witten and Frank, 2005).

- Statistical approaches (Sugiyama, 2015) that include statistical methods for pattern recognition such as k-nearest neighbours, bayessian classifiers, neural networks and support vector machines.

There are, of course, mixed techniques that use elements from both approaches. For example, there are random forest classifiers that statistically combine the results of logical models to make a prediction (Mease and Wyner, 2008).

The algorithms developed in this research belong to the first class, producing human-readable symbolic models mainly intended to classify and generate new data items. From this perspective, as is pointed out by (Fürnkranz et al., 2012), they should be conceived as hypothesis constructors rather than hypothesis testers[1] (i.e., The algorithms

[1]This idea differentiates the intentions of classic statistical methods and machine learning or data mining algorithms. Statistical methods assume a hypothesis, which is accepted or rejected after analysing the data. Machine learning or data mining algorithms search for patters without assuming a specific structure, and the resulting model can be conceived as a hypothesis that explains the data.

are intended to generate possible explanations of the data rather than confirming or rejecting a hypothesis). The explanations built (hypothesised) aim to extract implicit, unknown and interesting information contained in the data.

Next, the common predictive data mining symbolic algorithms (decision tree and rule set learners) are reviwed.

## 3.1 Decision Trees

Decision tree learning starts from a dataset of observations, in which each datum is composed by a set of attributes, labeled with an associated class. A decision tree can be seen as an algorithm composed by conditional statements that evaluate the attribute values and separate the data in accordance with the results. Each conditional statement or test is a node of the tree and is labeled with an input feature. For example, let us assume that we have a dataset with observations composed by different attributes and a class label that tells us if a person is a vampire or not[2]. Assuming attribute "eats garlic" and the test "'does the person eat garlic?", with possible answers Yes and No, then this node will separate the data into two subsets containing those that do eat garlic and those that do not. When successive nodes are combined, the branches represent the conjunctions of the features of the branch (e.g "eats garlic" = Yes AND "has shadow" = Yes).

Trees take its name from its structure that resembles an inverted tree. At the top is the *root* node, which is the attribute whose test produces the partition into more "homogeneous" sets among all the nodes (i.e, the partition that produces the sets containing as many elements of the same class as possible in each set). When a set resulting of a partition has observations of only one class, it is considered a leaf. Once the root node is selected, the same criteria is used to select the subsequent (internal) nodes until all the leafs of the tree have been created. In this way, decision trees perform recursive partitions of the space to divide the data. To select the attribute producing

[2]This example is taken from MIT OpenCourseWare https://youtu.be/SXBG3RGr_Rc

the mots homogeneous partition, information theory is used to measure the "degree of disorder" of each created set[3]. The selected attribute is the one producing the most "ordered" sets after the test.

Decision trees are constructed in a top-down fashion, selecting the most general tree (the root node) and refining it into a more specific tree structure.

The key process during the induction of the tree is selecting the "right generality level", as if we construct a lief node for every instance it would overfit the data. This is the role of the functions: to measure the disorder or "purity" of the partitions created by the nodes (i.e, to find the degree in which the resulting sets contains examples of a single class). The objective is to construct a complex tree, able to classify the data and general enough to be interpretable and to avoid overfitting.

Finally, it is worth mentioning that many decision tree algorithms have a post processing part to prevent overfitting due to the recursive partitioning of the data during the tree construction. The recursive partitioning produces fewer and fewer examples ending up at each node, and the consequent complex subtrees explain the data but do not generalize well. Common post processing strategies prune the branches near the leaves, replace the interior nodes with new leaves (removing subtrees rooted at that node) and assign to that leaf the label of its most frequent class.

### 3.1.1 Decision tree algorithms

Decision tree induction algorithms can be classified by their splitting criteria. For example: Impurity-based, Information gain, Gini index, Likelihood ratio, Chi-squared statistics, DKM, Normalized Impurity, Gain ratio, Distance Measure, Kolmogorov-Smirnov, etc. A general review of the different splitting criteria and algorithms can be found in (Rokach and Maimon, 2008; Priyanka and Kumar, 2020). Here, I present algorithms C4.5, CART, CHAID and QUEST, as their splitting criteria provide a gen-

[3] In a simple two class problem the first approach would be Disorder(set) = $-\frac{P}{N} log_2(P/N) - \frac{N}{T} log_2(N/T)$ where T is the total number of examples to test, N are the Negative and P the positive. Negative and Positive refer to whether or not the example has the tested characteristic.

eral perspective of the different possibilities. These algorithms use univariate splitting criteria, which splits the internal nodes of the trees according to the value of a single attribute. Then, the inducer searches for the best attribute upon which to split.

### Notation

To describe the splitting criteria I will use the selection operator of relational algebra ($\sigma$)[4]. In a batch schema the input dataset has the form $B(A \cup y)$. The input data has $n$ attributes $A = \{a_1, ..., a_i, ..., a_n\}$. Attribute $a_i$ has domain values denoted $dom(a_i) = \{v_{i,1}, v_{i,2}, ..., v_{i,j}, ..., v_{i,|dom(a_i)|}\}$, where $|dom(a_i)|$ has a finite cardinality. Similarly, the target attribute is $dom(y_i) = \{c_1, ..., c_{|dom(y)|}\}$.

Then, the set of all possible examples is $\mathbf{X} = dom(a_1)$ x $dom(a_2)$ x . . . x $dom(a_n)$, and the Labeled instances space is $U = \mathbf{X}$ x $dom(y)$. With these definitions, the training set is: $S(B) = (< x_1, y_1 >, .... < x_m, y_m >)$ such that $x_q \in \mathbf{X}$ and $y_q \in dom(y)$.

### C4.5

The C4.5 algorithm (Quinlan, 2014), a modified version of the ID3 algorithm (Quinlan, 1986), uses as splitting criterion the normalized information gain or relative entropy[5]. The attribute with the highest normalized information gain is chosen to make the decision.

The algorithm uses these base cases to avoid getting stuck:

[4]In relational algebra, the unary operator **Selection** describes the subset of a dataset, $\sigma_\varphi(S)$, where $\varphi$ is a propositional formula consisting of atoms and logical operators. For instance, selecting live coding music produced using SuperCollider in a music database can be $\sigma_{\text{live coding = True AND language SuperCollider = True}}(musicDatabase)$, which would result in a relation containing every attribute of every unique record where live coding = True and language SuperCollider = True.

[5]The relative entropy, or Kullback–Leibler divergence, is a measure of how one probability distribution differs from another "apriori probability". For discrete probability distributions $p, q$, defined on the same probability space, $X$, the Kullback–Leibler divergence from $q$ to $p$, being $p$ the apriori probability, is: $D_{KL}(p|q) = -\sum_{x \in X} p(x) \log\left(\frac{q(x)}{p(x)}\right)$

1. If all the observations in the list belong to the same class, then create a leaf node that assigns every observation to that class.
2. If none of the features provide any information gain, then create a decision node higher up the tree using the expected value of the class.
3. If an instance of a previously-unseen class is encountered, then create a decision node higher up the tree with the expected value of the class.

The general structure of the algorithm is as follows:

1. Check if any of the base cases hold.
2. For each attribute $a$, calculate the normalized information gain ratio resulting from splitting on $a$.
3. If $a^*$ is the attribute with the highest normalized information gain.
4. Create a decision node splitting on $a^*$.
5. Use recursion on the sublists obtained by selecting $a^*$ to split, and add those nodes as children of node.

The trees extracted by the C4.5 can be used for classification. As such, this algorithm is also considered a statistical classifier. To understand the calculus of the information gain, let us first analyze the impurity measure, the probability of the target value and the impurity criteria.

**Impurity measure:** Let $x$ be a random variable with $k$ discrete values with probability $P = (p_1, p_2, ..., p_k)$. An impurity measure is a function: $\phi : [0, 1]^k \to R$ such that:

- $\phi(P) \geq 0$
- $\phi(P)$ is minimum if $\exists\, i \cdot \ni \cdot p(i) = 1$

- $\phi(P)$ is maximum if $\forall\, i\ 1 \le i \le k$, $p = 1/k$
- $\phi(P)$ is symmetric with respect to components of $P$
- $\phi(P)$ is differentiable in all its range

**Probability of the target value:** Given a training set $S$, the probability of the target attribute is defined as in (3.1):

$$P_y(S) = \left( \frac{|\sigma_{y=c_1} S|}{|S|}, ..., \frac{|\sigma_{y=c_{|dom(y)|}} S|}{|S|} \right) \tag{3.1}$$

This means that the probability of a particular target value is given by the amount of examples having that value divided by the cardinality of $S$.

**Impurity criteria:** The impurity-based criteria or "quality" of a split due to the discrete attribute $a_i$ is defined as the reduction in impurity of the target given by partitioning $S$ using the values $i, j$ for $v_{i,j} \in dom(a_i)$, which is defined as in (3.2) using an impurity measure $\phi$.

$$\Delta\phi(a_i, S) = \phi(P_y(S)) - \sum_{j=1}^{|dom(a_i)|} \frac{|\sigma_{a_i=v_{i,j}} S|}{|S|} \cdot \phi(P_y(\sigma_{a_i=v_{i,j}} S)) \tag{3.2}$$

**Information gain**

Information gain measures the impurity using entropy. It is defined in (3.3):

$$\text{Information gain}(a_i, S) = Entropy(y, S) - \sum_{v_{i,j} \in dom(a_i)} \frac{|\sigma_{a_i=vi,j} S|}{|S|} \cdot Entropy(y, \sigma_{a_i=v_{i,j}} S) \tag{3.3}$$

where the entropy is defined as in Equation (3.4)

$$Entropy(y, S) = \sum_{c_j \in dom(y)} - \frac{|\sigma_{y=c_j} S|}{|S|} \cdot log_2 \frac{|\sigma_{y=c_j} S|}{|S|} \tag{3.4}$$

## CART

Classification and regression trees (CART) (Breiman et al., 1984) is an algorithm for learning binary trees. In a binary tree, each node has only two outgoing edges. To select the splits, the Twoing criterion is used, and the obtained trees are pruned by using complexity cost measures. CART is able to learn regression tress (which have the capacity to predict not classes but numbers).

Binary criteria divide the input attribute domain into two subdomains. This can be represented as $\beta(a_i, dom_1(a_i), dom_2(a_i), S)$, where, for attribute $a_i$, the sample $S$ is split into $dom_1$ and $dom_2$. The optimal division produces a value that is used to compare attributes: $\beta^*(a_i, S) = \underset{\forall\, dom_1(a_i);\, dom_2(a_i)}{\max} \beta(a_i, dom_1(a_i), dom_2(a_i), S)$

The binary Twoing criterion is defined as in Equation (3.5):

$$\text{twoing}(a_i, dom_1(a_i), dom_2(a_i), S) = 0.25 \cdot \frac{|\sigma_{a_i \in dom_1(a_i)} S|}{|S|} \cdot \frac{|\sigma_{a_i \in dom_2(a_i)} S|}{|S|} \cdot \left( \sum_{c_i \in dom(y)} \left| \frac{|\sigma_{a_i \in dom_1(a_i) AND y=c_i} S|}{|\sigma_{a_i \in dom_1(a_i)} S|} - \frac{|\sigma_{a_i \in dom_2(a_i) AND y=c_i} S|}{|\sigma_{a_i \in dom_2(a_i)} S|} \right| \right)^2 \tag{3.5}$$

## CHAID

Chi-squared-automatic-interaction-detection (CHAID) uses chi-squared statistics to identify the optimal splits. It works as follows:

1. The algorithm tries to break in every attribute and finds the attribute and the breaking point producing the biggest intergroup distance between the class (dependent variable) measured by chi-squared.

2. Step 1 is repeated for the two resulting groups. One of these groups is split in the same way, yielding 3 groups.

3. Recurse until a minimum chi-squared value is reached.

The likelihood-ratio Chi-squared is defined as: $G^2(a_i, S) = 2{\cdot}ln(2){\cdot}|S|{\cdot}\text{InformationGain}(a_i, S)$ and measures the statistical significance of the information gain criterion (Equation

(3.3)). The null hypothesis is that the input and target attributes are conditionally independent. If this hypothesis holds, the test statistic is distributed as $\chi^2$, with degrees of freedom given by $(dom(a_i) - 1) \cdot (dom(y) - 1)$.

**QUEST**

Quick, unbiased, efficient, statistical tree (QUEST) compares the association between the attribute and the target, to create each split. For that, it uses ANOVA F-Test or Levene's test, for continual attributes, and Pearson's chi-square, for nominal attributes. It operates as follows:

1. For each attribute compute an ANOVA F-statistic. To split the attribute (if it is an ordered variable), proceed as follows: Let X be the selected attribute to split node $t$. Divide the $C_t$ classes into two superclasses A and B by using the 2-means clustering algorithm with the two most extreme sample means as initial cluster centers. If the sample means are identical, let A contain the most populous class and B contain the other classes.

2. If the largest F-statistic is greater than a established threshold, the attribute with largest F-value is selected to split the node.

3. Else, compute for each attribute the Levene's test for unequal variances. Again, if the largest Levene's statistic value is greater than a established value, then the attribute with largest Levene value is used to split the node.

4. Else, split the node using the attribute with largest ANOVA F-value.

The QUEST algorithm uses 10-fold cross validation to prune the trees. For multimodal target attributes, two-means is used to create two clusters, each representing a super class. Then, the attribute with highest association is selected to split the node by using discriminant analysis to select the breaking point within the attribute.

## 3.2 A Brief Overview of Rule Learning

Rule models for classification are sets of "IF-THEN" rules. A rule is a conjunction of attribute values located in the conditional "IF" part and a class label in the consequent "THEN" part.

Propositional IF-THEN rule learners solve the problem of: given a set of training examples, to find a set of rules that can be used for prediction or classification of new instances. Algorithms such as CN2 (Clark and Boswell, 1991), RIPPER (Cohen, 1995) and PRIM (Friedman and Fisher, 1999) are representatives of this class of learners. The extracted "IF-THEN" rules are implications of the form:

$$\text{Condition} \rightarrow \text{class}.$$

### 3.2.1 Rule learning v.s Decision trees

Rule sets are normally simpler and more understandable than decision trees. The general ideas in rule-learning are similar to those used in decision trees. Nonetheless, there is a subtle difference. Decision trees create recursive partitions of the space by considering all nodes and optimizing the purity to select the one to split. This strategy is commonly termed "Divide and conquer". Rule learning can be pictured as "expanding one single successor node at a time", by learning a rule that covers some examples in the data and removing them from the training set once the rule is learned. This strategy is referred by (Fürnkranz et al., 2012) as "Separate and conquer".

(Rivest, 1987) showed that ordered rule sets with, at most, $k$ conditions per rule are more expressive than decision trees of depth $k$.

(Cendrowska, 1987) showed that the minimal decision tree for the concept X, defined as:

$$\text{IF A} = 3 \text{ AND B} = 3 \text{ THEN X}$$
$$\text{IF C} = 3 \text{ AND D} = 3 \text{ THEN X}$$

Has 10 interior nodes and 21 leafs assuming that attributes A to D have three different values.

### 3.2.2 Concept learning

Most rule learning algorithms aim to learn a target concept, $c$ (sometimes denoted by the symbol $\oplus$), from training information consisting of positive and negative examples. A **complete** hypothesis (model) is one that covers all positive examples, and the model is **consistent** if it covers no negative examples.

Single-concept learning constructs a theory for a single class, and all the instances that are not covered by the learned rules are classified as negative. For multiclass learning, single concept learning can be iteratively applied. However, searching for complete and consistent models in the presence of noisy data (which contains errors in the instance descriptors and in the class labels) can lead to overfit of the data, as the rules will explain the errors as well. Another problem is when classes are not completely disjointed. In such cases, the rule construction processes are guided by heuristics, such as the high-predictive accuracy or the coverage of the extracted rules, rather than searching for completeness and consistency. This separates "theoretical" learners from algorithms design to handle "real live" data.

### 3.2.3 Data representation

The input of a rule learner is a set of training examples. Each example contains a set of values with an associated class label. Examples are represented in the following form:

$$\nu_{1,j},\ \ldots \nu_{n-1,j},\ \ c_j$$

Element $\nu_{i,j}$ represents the value that attribute $A_i$ has for the example in the position $j$ of the set of training examples. Attributes can be of several types (e.g. discrete or continuous). Every attribute $A_i$ can have a different number of possible values. Finally,

$c_j \in \{c_1, ...c_c\}$ are the possible classes for the examples. A set of examples is organized in a table. The attributes represent the columns and the examples the rows.

### 3.2.4 Rule representation

Learned rules have the form:

$$\text{IF } f_1 \text{ AND } f_2 \ . \ . \ . \text{ AND } f_L \text{ THEN Class} = c_i$$

The conditional part contains a logical conjunction and the consequent part contains the class of the rule. L is the rule length. Each $f_k$ is a **test** that verifies if the entrance $k$ of an example being compared with the rule has a specific property or not. One possible test, for instance: is the entrance $k$ of the example a subset of $\{1, 2, 3\}$? Another possibility could be: does the entrance $k$ of the example $= 2$?

All the examples that satisfy the conditional part will be classified as $c_i$ by the rule.

### 3.2.5 Rule models

A set of logical rules can be interpreted as a classification model that provides an approach to the posterior probability $p(C_k|\mathbf{X}; M)$ where $\mathbf{X}$ is an input vector and $M$ is the model. This is the probability of the input vector $\mathbf{X}$ having class $C_k$ considering the model $M$.

In this interpretation, the conditional part of the rule is a conjunction of predicate functions. A predicate function $L_j$ is defined as $L_j : \mathbf{X} \to \{T, F\}$. In most of the cases, the tests functions operate on a single attribute. Two examples of a test could be:

$$L_{i,j}(X_i) = T \text{ if } X_i \in \text{a subset of values}$$

or

$$L_{i,j}(X_i) = F \text{ if } X_i \notin [X_{i,j_-}, X_{i,j_+}] \text{ (some interval).}$$

In these cases, the index $j$ enumerates the values or intervals for attribute $X_i$ associated with test $L_{i,j}$.

The conjunctions of conditions define the areas (for example hyperrectangular) in the feature space where the rule is true. For example, if the predicate functions consider conditions $L_{i,j}(X_i) = F$ if $X_i \notin [X_{i,j_-}, X_{i,j_+}]$, the rule:

$$L_{1j_1}(X_1) \wedge L_{2j_2}(X_2) \ldots \text{THEN Class}(\mathbf{X}) = C_k \tag{3.6}$$

will assign class $C_k$ to all instances located at the intersection of the intervals. Note that subindexes of $j$, for instance $j_1$, indicate that, for each variable $i$ in the conditional part of the rule, the interval considered can be different.

There are rule learning algorithms that can produce hyperrectangles that overlap. There are also different ways to solve the created conflicts at the time of assigning a class, such as selecting rules with the lowest false positive rate.

Algorithms such as decision trees avoid overlapping. Most decision trees use tests like $L_{i,j} < T_{i,j}$. Then, if the variables are ordered, the test defines a hyperplane perpendicular to $X_i$ and intersecting the axis at threshold $T_{i,j}$. This constitutes the recursive partitioning of decision trees. Figure 3.1 shows the decision surface of the decision trees trained on pairs of features of the Iris dataset (Fisher, 1936).

In fuzzy set theory (Zadeh, 1965), if $S_{i,j}$ is the $j$ fuzzy set on attribute $i$, then $X_i \in S_{i,j}$ to a degree determined by the membership function $L_{i,j}(X_i)$. In the rule conditions, instead of input crisp values for $X_i$, fuzzy sets are used. In such a way, fuzzy sets provide more flexible decision boundaries, removing the discontinuities of the crisp logical rules, which jump from one interval to the adjacent one. Figure 3.2 illustrates the shapes of different decision borders created by (a) clusters; (b) fuzzy rules with product of membership functions; (c) fuzzy rules with trapezoidal membership functions; (d) crisp logical rules.

### Different forms of expressing rules

The most common form for expressing rules is using conjunctions. However, there are other possibilities that are worth reviewing. For instance, M out of N rules, in which, for the rule to be considered true, only M out of N tests on the attributes need to be

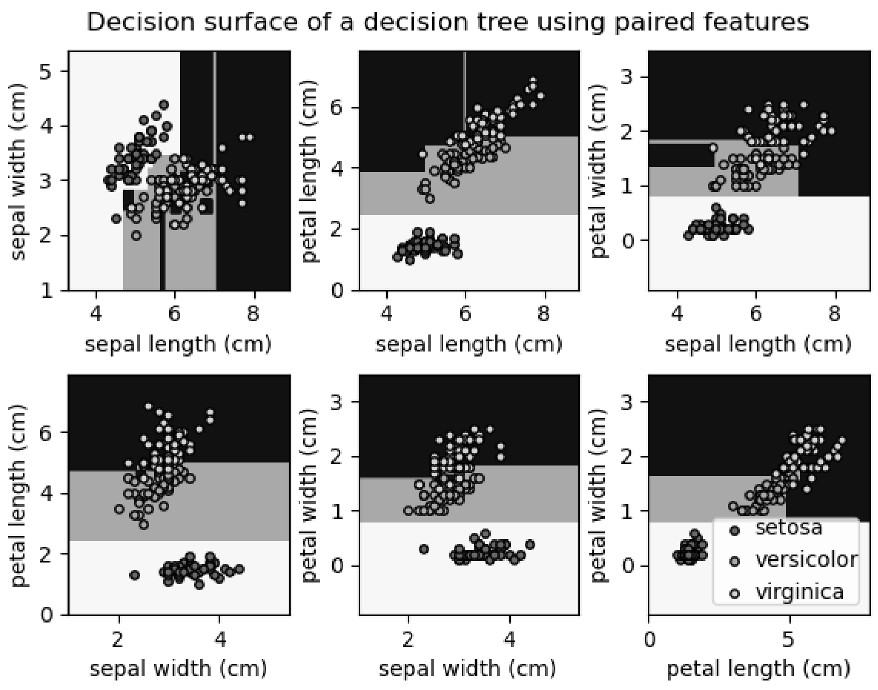

Figure 3.1: Decision surface of the decision trees trained on pairs of features of the iris dataset.

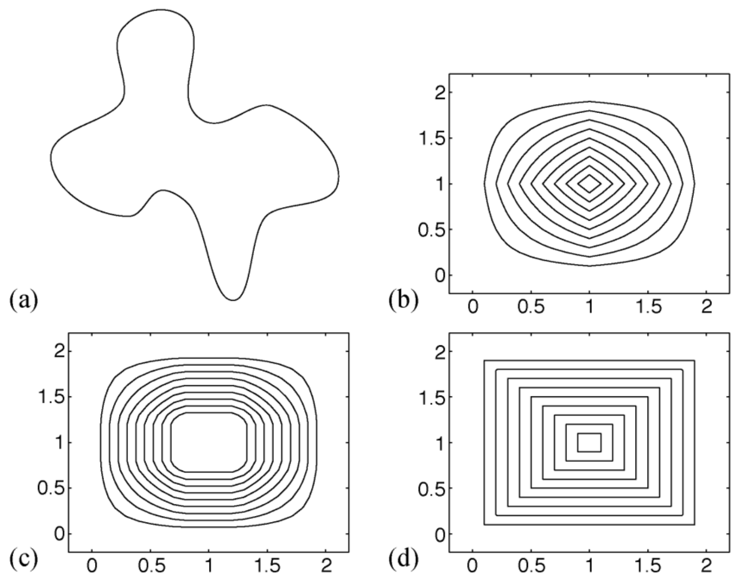

Figure 3.2: Shapes of different decision borders created by (a) clusters; (b) fuzzy rules with product of membership functions; (c) fuzzy rules with trapezoidal membership functions; (d) crisp logical rules. Figure from Duch et al., 2004.

fulfilled. These types of rules can be expressed (in a simple form) by using predicate functions that return values 0 or 1 as:

$$\text{IF} \sum_{i}^{N} L_{i,j}(X_i) \geq \text{M THEN Class}(\mathbf{X}) = C_k \tag{3.7}$$

where $i$ denotes the feature or attribute and $j = j(i)$ the corresponding test of the attribute. $M$ is a threshold value, so these rules can be represented using a threshold function $\Theta$ in the following way:

$$\Theta(\mathbf{W} \cdot \mathbf{X} - M)$$

where $X_i$ is binary, $W_i = 1\ \forall i$ and $\Theta(x) = 0$ for $x < 0$ or $\Theta(x) = 1$ for $x \geq 0$ (note that $\Theta(x)$ is a general function).

This representation connects rule systems with neural nets, as, in the way it is expressed, the threshold function is a logical neuron. Moreover, if $X_i, W_i \in \mathbb{R}$ and $\Theta$ is replaced by a sigmoid, we obtain a perceptron.

As discussed previously, propositional rules can be expressed by using predicate functions $L_j(\mathbf{X})$ that compare several attributes of $\mathbf{X}$. In that case, the expression $\|\mathbf{X} - \mathbf{R}\| < \Theta(\mathbf{R})$ measures the distance between $\mathbf{X}$ and the prototype $\mathbf{R}$.

Fuzzy logic uses operators named T-norms, which combine the degrees of membership given by membership functions $L_i(X_i), L_j(X_j)$. For instance, if $L_i(X_i) = A$, $L_j(X_j) = B$ and $A, B$ are continuous in $[0, 1]$, then $A \cdot B = min(A, B)$. In this way, in fuzzy rules, the logic operator $\wedge$ is usually replaced by the T-norm that takes the minimum of the membership values of the attributes. Then, a rule $R$ is satisfied to a degree $L_R(\mathbf{X})$ where:

$$L_R(\mathbf{X}) = \prod_{(i,j)\in R} L_{i,j}(X_i) \tag{3.8}$$

For a class $k$, calculating the sum of fulfillment, we get an approach to the class classification probability:

$$p(C_k|\mathbf{X}; M) = \frac{\sum_R(k) L_{R(k)}(\mathbf{X})}{\sum_R L_R(\mathbf{X})}$$

This process is analogous to the cases where fuzzy systems use gaussian or sigmoid functions (that create circles or ovals as decision borders). That is, all the network outputs (rules) corresponding to class $C_k$ are added and normalized by dividing the result by the sum of all the outputs (considering all classes). Also, if the structure of the output or target is known, the class can be generalized to real values ($C_k \in [0, 1]$), creating general mappings from the feature space ($\mathbf{X}$) to the class labels $C$. These mappings create more flexible decision regions, although they require predefined fuzzy sets for the input and the output.

### 3.2.6 Input and output discretization

Some rule extraction algorithms discretize the input space (in case it is continuous) into linguistic variables. There are several ways to do that, ranging from using homogeneous discretization for all the attributes to feature-dependent partitions. There are many algorithms to automatically learn the best partition (for example in (Lavangnananda and

Chattanachot, 2017) a study of discretization methods in classification is presented).

In fuzzy logic, for example, linguistic variables are used to create the partitions of the features. For this, expert knowledge is used to define the membership functions of the input and output. In this way, the exponential growth produced by considering the same partition for all the features is avoided. For example, if sixteen features are present in the data, and each of them is divided in eleven membership functions, the resulting IF-THEN rules can be hard to interpret.

If expert knowledge is available, creating a coarse division of the space can lead to simple, expressive and accurate models. Assume, for example, that $X_1$ contains frequency values (freq) of a Saw wave, and $X_2$ describes the cut frequency (cutFreq) of a Low pass filter that cuts the upper harmonics of the Saw wave.

The SuperCollider implementation of this experiment is:

```
Ndef(\x,{arg freq, cutFreq;
  var sig;
  sig = Saw.ar(freq);
  sig = LPF.ar(sig, cutFreq)
})
```

As it is a two parameters synth, a couple of knobs allow anyone to explore the space and propose linguistic descriptors to divide the features into intervals, for instance:

$$L_{1,1}(X_1) = \text{low iff } 0 < X_1 \leq 30;\ L_{1,2}(X_1) = \text{high iff } 30 < X_1$$
$$L_{1,3}(X_2) = \text{low iff } 0 < X_2 \leq 50;\ L_{2,2}(X_2) = \text{high iff } 50 < X_2$$

Using these intervals, an aural exploration can lead to formulation of some simple rules, for example:

$$\text{IF } (L_{1,1}(X_1) = \text{low}) \text{ THEN } C = \text{beats}$$
$$\text{IF } (L_{1,1}(X_1) = \text{high} \wedge L_{2,1}(X_2) = \text{low}) \text{ THEN } C = \text{soft tone}$$
$$\text{IF } (L_{1,1}(X_1) = \text{high} \wedge L_{2,2}(X_2) = \text{high}) \text{ THEN } C = \text{harsh tone}$$

These simple rules provide a compact description mapping the input and output spaces (in this case for the one who created the linguistic labels).

However, the feature space is not always explorable. Our example with two parameters is a toy example, but adding a third and a fourth . . . A good analogy might be the three-body problem of physics. Here, adding an extra parameter or making the relationship between parameters more complex can create a problem that can not be exhaustively solved anymore.

Furthermore, using membership functions to describe an attribute assumes monotony in the linguistic variables. In our example, in {low, high} it is assumed that low precedes high. As discussed in Chapter 2, although variables such as frequency can be divided with respect to our perceptual windows, when they are used to modulate other parameters the resulting behavior does not vary monotonously. Moreover, when labeling the aural results of a synthesizer we could associate as category the "part of the piece" the setting is intended to be used on (e.g intro, break, break2, main, transition, end). Although it may appear that there is an order between these parts, one does not necessarily precede the other. So once again, relying on experts to discretize the space is not an option. In Section 3.5.7, automatic discretization of the features and output based on the information provided by the data is explored.

## 3.3 Rule Learning Process

Rule learning has three conceptual processes:

- Feature construction, which prepares the object descriptors.
- Rule construction, which constructs individual rules, each covering a fraction of the examples. This is normally done by fixing a class and heuristically searching for the conjunction of features most predictive of the selected class.
- Hypothesis or model construction, in which the model consisting of a set of rules describing the data is built. In propositional rule learning, model construction

iteratively learns single rules until no new rules can be learned or some other terminal criteria is reached.

Section 3.2.6 has briefly discussed Feature construction. Sections 3.3.1 and 3.4 describe, respectively, the processes of learning a single rule and iteratively learning rules to build a model.

### 3.3.1 Learning rules as a search problem

According to Mitchell, 1982, *"learning is the ability to generalize: to take into account a large number of specific observations, then to extract and retain the important common features that characterize classes of these observation"*. For him, the generalization problem is essentially a search problem. This conception engages with the perspective of the thesis on generalization by searching within a space of possible solutions. However, in the context of this work, as discussed in Section 2.1.4, we have to keep in mind that *what we search for* (i.e what is a solution for a problem, is both context and subject dependent). With this in mind, the problem can be formalized in the following way.

To state a search problem we define:

- a search space
- a search strategy
- a quality function that determines to what degree a rule is a solution of what is being search.

The search space constitutes all possible expressions that can be formed with the hypothesis language. In this work, the search space is formed by all the possible combinations that a particular synthesis algorithm allows.

Search strategies include top-down and bottom-up. The first starts from the most general rule, making it more specific, so it covers only examples of the given class. Bottom-up strategies start from a specific rule (e.g the empty rule covering no examples or a rule covering a single instance) and generalize it until it can not be generalized without covering negative examples. Normally, top-down searches obtain more general rules. Top-down search is useful for working with noisy data, as it can be guided by heuristics that make big inductive leaps.

Bottom-up search is "better suited for cases where fewer examples are available and for interactive and incremental processing" (Fürnkranz et al., 2012). These algorithms, however, might have problems dealing with noisy data. This approach is the one used by the algorithms of this work, as the application domain allows interactive and incremental processing. Also, as the data is collected by the composers, it can be assumed to be less noisy (or noisy examples are included on purpose), or the unwanted examples can be removed or added by the composers. Fuzzy rule learning, explored in Chapter 6, is a recurrent strategy for dealing with noisy data.

Finally, a quality function that determines to what degree a rule is a solution of what is being searched has two parts: One that is developed in Sections 4.1.3 and 4.1.4 establishes that a new rule has to fulfill the conditions of not generating contradictions with the input data and to contain a specific percentage of the original data established by the user. The second part cannot be formalized, as the solutions are context and subject dependent. This means that no matter what formal requirements the rule fulfills, human perception within a context will always be the final judge.

## 3.4 Learning Rule Models

Algorithms that learn sets of rules iteratively learn one rule that covers part of the examples until a terminal criteria is met or no more examples remain.

This general idea can be traced back to the covering algorithm (Bagallo and Haussler, 1990). The strategy, termed separate-and-conquer, has two main processes: learn a rule that covers part of the examples and eliminate them from the training set (the separate part), and recursively learn the rest of the examples (the conquer part).

**Definition: COVERED** Given $r_1$ and $r_2$, $r_1$ is more general that $r_2$ (denoted $r_2 \subseteq r_1$) if and only if $r_1$ and $r_2$ have the same consequent and COVERED($r_2$,$\varepsilon$) $\subseteq$ COVERED($r_1$,$\varepsilon$), where COVERED($r$,$\varepsilon$) denotes the subset of examples $e \in \varepsilon$ covered by $r$.

Algorithms 1 and 2 show the Covering algorithm and an algorithm to learn rule sets of multi-class problems.

**Algorithm 1** Learn set of rules

```
 1: function LEARN SET OF RULES($c_i$, $P_i$, $N_i$)
Require: $c_i$: a class value
Require: $P_i$ positive examples with class $c_i$
Require: $N_i$ negative examples for $c_i$ $N_i = \epsilon \setminus P_i$
Require: $\epsilon$ is the set containing all the examples
 2:     $P_i^{cur} := P_i, N_i^{cur} := N_i\ R_i := \emptyset$
 3:     repeat
 4:     r := LearnOneRule($c_i$,$P_i^{cur}$,$N_i^{cur}$)
 5:     $R_i := R_i \cup \{r\}$
 6:     $P_i^{cur} := P_i^{cur} \setminus Covered(r, P_i^{cur})$
 7:     $N_i^{cur} := N_i^{cur} \setminus Covered(r, N_i^{cur})$
 8:     stops if terminal condition is met (e.g $P_i^{cur} = \emptyset$)
 9:     return $R_i$
10: end function
```

Algorithm 1 requires a class $c_i$, positive $P_i$ and negative $N_i$ examples for that class. Then, it iteratively learns a rule and removes, either from $P_i$ or $N_i$, the examples covered by the learned rule. The iterations finish when a terminal condition is met.

Algorithm 2 learns a rule base for each class $R_i$ and creates the rule base as $R = \cup_i R_i$ (this strategy is called one-against all, as one class is considered positive while all the others negative during the iterations of the algorithm). The algorithm presented in Chapter 4 has this general structure.

**Algorithm 2** Learn rule base

```
1: function LEARN RULE BASE($\epsilon$ $R$ )
Require: $\epsilon$ training set
Require: $R := \emptyset$
2:     for each $c_i$ do
3:         $P_i := \{$subset of examples $\in \epsilon$ with class $c_i\}$
4:         $N_i := \{$subset of examples $\in \epsilon$ with class $c_i\}$
5:         $R_i :=$ LearnSetOfRules($c_i$,$P_i$,$N_i$)
6:         $R := R \cup R_i$
7:     end for
8:     $R := R\cup$ $\{$default rule that assigns examples to the majority class$\}$.
9:     return $R$
10: end function
```

Algorithms 1 and 2, together with a LearnOneRule function, are the general structure for learning a rule model. The LearnOneRule function can start from the *most general rule* and then build refinements of it by analyzing the examples "cutting the rule" (top-bottom), or start from a single example (bottom-up) and "extend" the rule, as long no contradictions appear, until it covers the greatest amount of examples.

### 3.4.1 Evaluation of the predictive accuracy of the rules

There are many measures in the machine learning/data mining fields developed for rule evaluation, including precision (based on the confusion matrix), information gain, m-estimate, etc. For example, a coverage-consistency trade off, as a multi-optimization problem, searches for general rules to maximize the coverage but minimize the number of negative examples covered. The covered and not covered examples are expressed in the confusion matrix, shown in Table 3.1. The coverage-consistency test can be used as a criterion to select among rules. In this way, the rules extracted by a procedure (e.g

LearnOneRule) can be compared and the best one selected.

Table 3.1: Confusion matrix.

| **Examples** | **Covered** | **Not Covered** | **Total = P + N** |
|---|---|---|---|
| Positive | true positives | false negatives | P (all positives) |
| Negative | false positives | true negatives | N (all negatives) |

The classification accuracy of a rule set is defined as the percentage of the total number of correctly classified examples in all classes relative to the total number of examples.

k-fold cross-validation can be used to estimate a model's accuracy (Fürnkranz et al., 2012). In Section 6.2.1, cross-validation is used to estimate the average accuracy of the resulting models.

Once again, it is important to mention that, no matter how good the accuracy results are, unless the scoring metric is the human ear, these do not indicate with certainty how the results will be perceived. Therefore, from now on I will make the distinction between user tests, in which we are interested in providing a subjective evaluation, and accuracy results, which are intended to provide a general evaluation of the model.

## 3.5 Inductive Rule Learning Algorithms

This Section provides an overview of important covering algorithms. Throughout the section, advantages, disadvantages and ideas taken for the development of the algorithms presented in Chapters 4 and 6 are commented on, from the point of view of this research.

### 3.5.1 AQ

The Algorithm Quasi-optimal (AQ) was introduced in the late sixties (Michalski, n.d.; Cervone et al., 2010). Its name derives from its basis on an algorithm for determining quasi-optimal solutions for the general covering problem. In fact, it shares many of the functions and features with the covering algorithm described in Section 3.4.

It learns patterns form labeled data and handles multiclasses by considering, in turn, each class as positive and all other classes as negative.

The algorithm has two operation modes, theory formation, and pattern discovery, that guide the coverage-consistency of the extracted rules. For theory formation the algorithm searches for rules that cover all the positive examples and do not cover any of the negative. In its other mode, the algorithm accepts trading coverage to gain simplicity of pattern. Also, the generated rules go through an optimization process that generalizes and/or specializes the learned rules to simplify the patterns.

The AQ algorithm works with the so-called "star generation", which selects a positive example (seed) from which to generalize. This process is repeated until all the positive events are covered.

Newer versions of the algorithm use a top-down beam search to find the best rule set. For multiclass problems, first, for each class, a seed sample is selected. Then, for each example the most general rules covering it (and other examples with the same class) are built. Then, the algorithm uses an evaluation function (that may consider, for example, rule precision, number of false positives, etc) that evaluates the constructed rules, selecting the one with the highest function value. As in the covering algorithm, the examples covered by the selected rules are eliminated and the algorithm iterates this process until the final criteria are met.

### 3.5.2 CN2

The CN2 algorithm was developed by (Clark and Niblett, 1989) and afterwards extended by (Clark and Boswell, 1991). The general idea was to combine the capacities of the decision tree algorithm ID3 to avoid overfitting with those of the AQ algorithm for rule induction. To combine a decision tree with a rule learning algorithm, the authors observed that learning a rule is like learning a single branch of the tree. In order to not overfit the induced concept, it is necessary to relax the constraint of classifying the data perfectly.

The ID3 can manage noisy data because of its top-down approach. The AQ function for refining rules is also general-to-specific. It only considers specializations that exclude some covered negative examples, i.e, it only searches for refinements that are completely consistent with the training data. With this in mind, they modified the AQ to remove its dependence on specific examples and to enlarge the space of rules searched.

The search strategy in the CN2 is top-down. It starts with a rule that assigns all the training data to a class (IF True THEN Class = $C$). Then, it performs a beam search to refine the rule. For this, the algorithm can use measures such as the precision of the rule or entropy measures. To relax the search, together with the training data, a set of conditions is also used. These conditions guide the refinements, and the algorithm is able to obtain more general and interpretable rules. The CN2 has the merit of being the first rule learning algorithm to consider overfiting.

Other algorithms followed the idea of evaluating the resulting quality of refining a rule by different means. For example, Foil is a relational learning algorithm (Quinlan, 1990) that uses information gain to evaluate how much a rule improves when refining it.

### 3.5.3 RIPPER

Even though the algorithms mentioned focussed on overfitting, either their methods were inefficient (for example the stopping criteria), or the size of the theory learned grew with the size of the training set. RIPPER effectively countered overfitting. One of the key ideas of this algorithm was to include a post-processing phase for optimizing a rule considering other rules. For this, one rule, from a previous set learned during the iterations, is removed, and the algorithm tries to relearn it again in the context of the previous rules and the subsequent ones.

This algorithm (Cohen, 1995) creates logical conjunctive rules in a similar way as decision trees do, i.e splitting the data using information gain. A rule is developed until it covers examples of a single class. Then, the algorithm uses "incremental reduced error pruning". For this, the algorithm removes the last added conditions until the number of correctly covered cases minus the number of incorrectly covered cases divided by the number of covered cases is maximized. Once a rule is learned, the examples covered are removed from the training set.

By using these strategies, the learned rules are more likely to represent strong patterns rather than random occurrences.

### 3.5.4 PROGOL

PROGOL is an inductive logic programming system that mixes ideas of inductive programming and the Algorithm Quasi-optimal. Considering a maximum inference depth, it selects an example and computes a minimal generalization with respect to the available background knowledge. The resulting rule is then used for constraining the search space in a top-down search that looks for the best generalization. Therefore, the algorithm contrasts bottom information with a general-to-specific search. To find the best generalization, the algorithm uses a variant of the best-first search. This is possible given that the restriction of the space makes it manageable.

The idea of restricting the generalization using a threshold is also implemented in the algorithm presented in Chapter 4

### 3.5.5 OPUS

Optimizing pruning for unordered search (OPUS) is a learning system that made feasible the performance of an exhaustive search, trying all possible rule bodies to find the one that maximizes a condition given by some heuristic. For that, the algorithm uses an ordered search to avoid generating a rule more than once. For this, from the $l!$ possible orders of the $l$-conditions of a rule, only one can be used by the learner to build the rule. In addition, OPUS prunes parts of the space to make it even smaller. Given its capabilities to perform exhaustive search, this algorithm is especially popular for association rules.

A similar idea of performing an exhaustive search in a restricted space is also applied in the algorithm presented in Chapter 6. In that case, the search space is restricted to analysis of only the parts of the space where rules of different classes intersect each other. Then, to solve the contradictions, for a rule of length $l$, it is sufficient to try $l$-possible partitions. Finally, to decide which partition to select, a heuristic (in this case maximum space coverage) is used.

### 3.5.6 LR-FIR

The Linguistic Rules in a Fuzzy Inductive Reasoning Methodology (LR-FIR) (Castro et al., 2011) is a rule extraction algorithm able to derive linguistic rules from the models created by the Fuzzy Inductive Reasoning Methodology FIR (Escobet et al., 2008; Escobet Canal et al., 2015).

This methodology performs two steps, fuzzification and qualitative modeling, to produce the model from which the LR-FIR starts.

**Fuzzification** converts qualitative data into qualitative fuzzy data. For this, the user defines the fuzzy sets (number and shape), which are used to convert the numeric values into fuzzy descriptors. The FIR uses triplets that allow it to not lose information containing the class, membership value and side of the membership function to which the original data belongs.

Once the data is fuzzyfied, the **qualitative modeling** module looks for the model that best predicts the future behavior of the studied system. The model's structure in the FIR methodology is called a *mask*. The model looks for relationships among the variables that best determine the output of the system. The data is represented in the following way:

Table 3.2: Values of the variables $u_1$, $u_2$ and $y$ during tree $\delta t$ temporal intervals. Symbols “-” and ”+” represent mask inputs and mask outputs, respectively, and “0” unused connections. Example taken from (Escobet et al., 2008).

| **Time** | $u_1$ | $u_2$ | **y** |
|---|---|---|---|
| t - 2$\delta$t | $-1$ | 0 | $-2$ |
| t - $\delta$t | 0 | $-3$ | 0 |
| t | $-4$ | 0 | $+1$ |

A mask denotes a dynamic relationship among variables. A mask considers a certain number of rows, which represent the temporal domain that influence the output (in a similar way to Markov chains of n-order). Table 3.2 shows an example of a mask.

The FIR methodology considers the *ensemble* of all possible masks, from which the best is then chosen by exhaustive search (although genetic and tree algorithms are also available). The system searches for masks of depth two, then three and so on until the maximum depth is reached. The optimality of a mask is evaluated with respect to its ability to forecast quantified by the Shannon entropy. The behavior of the system is stored in a pattern rule-base, which contains registers of the values of the variables together with their output. The LR-FIR uses the set of patterns as input.

The LR-FIR algorithm is important as it served, among other algorithms, as inspiration for the algorithms developed in this manuscript (See for example (Mugica et al., 2015)). The main processes performed by LR-FIR are shown in Figure 3.3.

First, LR-FIR removes the few represented behaviors. These patterns are eliminated either by a threshold or by user inspection. Then, it performs basic compaction, in which, for every premise, it looks for subsets of rules that can be compacted under the following criteria: When all premises except one (called Pa) and the conclusion have the same values, and all the seen values of Pa are represented in the subset, then the

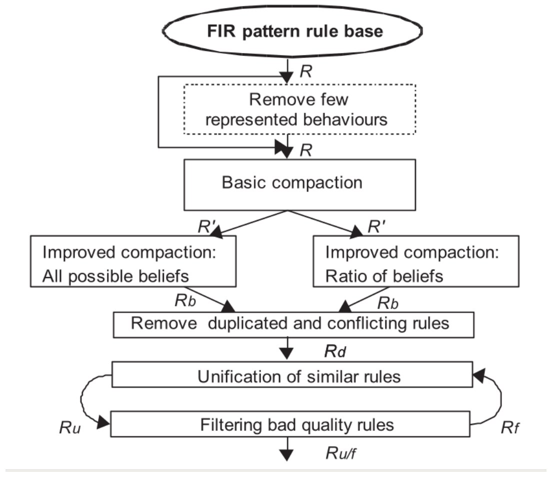
FIR pattern rule base
R
Remove few represented behaviours
R
Basic compaction
R'
R'
Improved compaction: All possible beliefs
Improved compaction: Ratio of beliefs
Rb
Rb
Remove duplicated and conflicting rules
Rd
Unification of similar rules
Filtering bad quality rules
Ru
Rf
Ru/f

Figure 3.3: Main processes of the LR-FIR algorithm. Figure from (Castro et al., 2011).

rule can be compacted. In that case, LR-FIR places a “-1” to indicate that Pa “is not important” in this case and that the set of rules can be compacted in a unique rule. An example of this process is shown in Table 3.3. The example assumes that premise Pa has only three different values in the training data.

Table 3.3: Example of the basic compaction step in LR-FIR.

| **Premises** | | | | **Consequent** |
|---|---|---|---|---|
| Rule ID | $P_a$ | $P_2$ | $P_3$ | consequent |
| 1 | 1 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 2 | 2 | 2 |
| Compacted Rule | -1 | 2 | 2 | 2 |

In the basic compaction step, premises are reviewed from left to right. If, during the basic compaction, more than a “-1” is going to be placed in a pattern, the algorithm expands the rule to all its possible valid cases in which the premises have a “-1” and checks for contradictions in the original rule base. Contradictions are considered when two rules have the same values in all their premises but different value in the consequent. If no contradictions are found, the compaction proceed.

One of the objectives of the LR-FIR is to provide a set of rules that is as compact as possible. After the basic compaction, the algorithm has two possible processes to *improve compaction*: All possible believes and ratio of believes.

In both cases, for each premise, the algorithm replaces it with a "-1", creates all the single rules that would be contained in the candidate rule (considering the "-1") and looks into the original rule base. If *all possible beliefs* is selected, it is sufficient to be free of inconsistencies (contradictions) for the candidate rule (with the -1) to be accepted. If the option *ratio of beliefs* is selected, for a rule to be accepted a *ratio* of the instances contained in a rule have to be present in the original rule base.

**Example of improving compaction**

Consider, for example, Rule ID 6 = [2, 2, 1, 2] in the Basic compaction set of Table 3.4. Further down in the Table, the results of applying *all possible beliefs* and *ratio of beliefs* to Rule ID 6 are shown. Lets focus on $P2$. The observed values for $P2$ are {1,2 3}.

Applying all possible beliefs to Rule ID 6, when substituting "-1" in $P2$, for the "-1" to be accepted, it requires that no contradictions with rules: [2, 1, 1, 2], [2, 2, 1, 2][2, 3, 1, 2] exist. As this condition is fulfilled, 6-All-Possible-Beliefs [2, -1, -1, 2] has a -1 in $P2$.

Applying ratio of beliefs to Rule ID 6, when substituting "-1" in $P2$, for the "-1" to be accepted, it requires that no contradictions with rules: [2, 1, 1, 2], [2, 2, 1, 2][2, 3, 1, 2] exist and that a ratio of them exists in the original data. As only rule [2, 2, 1, 2] is present in the original data, assuming, for example, a ratio of 2/3, the candidate rule is not accepted, and 6-Ratio-of-Beliefs = [2, 2, -1, 2]. This contrasts with $P3$. When substituting the "-1" in $P3$, either in all possible or ratio of beliefs, rules [2, 2, 1, 2] and [2, 2, 2, 2] are in the original data, so the "-1" is accepted in both cases.

Table 3.4: LR-FIR compaction examples. From top to bottom: original dataset, rule set resulting from the basic compaction, rules in the "basic compaction" that change with the improved compaction process using all possible beliefs and ratio of beliefs. Examples taken from (Castro et al., 2011).

| **Original data** | | | | |
|---|---|---|---|---|
| Rule ID | $P_1$ | $P_2$ | $P_3$ | consequent |
| 1 | 1 | 1 | 2 | 1 |
| 2 | 1 | 2 | 1 | 1 |
| 3 | 1 | 2 | 2 | 2 |
| 4 | 1 | 3 | 1 | 2 |
| 5 | 1 | 3 | 2 | 2 |
| 6 | 2 | 2 | 1 | 2 |
| 7 | 2 | 3 | 2 | 2 |
| 8 | 2 | 2 | 2 | 2 |
| **Basic compaction** | | | | |
| 1 | 1 | 1 | 2 | 1 |
| 2 | 1 | 2 | 1 | 1 |
| 3-8 | -1 | 2 | 2 | 2 |
| 4-5-7 | -1 | 3 | -1 | 2 |
| 6 | 2 | 2 | 1 | 2 |
| **Improve compaction all possible beliefs** | | | | |
| 1-All-Possible-Beliefs | -1 | 1 | -1 | 1 |
| 6-All-Possible-Beliefs | 2 | -1 | -1 | 2 |
| **Improve compaction ratio of beliefs** | | | | |
| 6-Ratio-of-Beliefs | 2 | 2 | -1 | 2 |

After the *improve compaction* step, the LR-FIR algorithm has modules to remove duplicated and conflicting rules. Conflicting rules are created in the context of the LR-FIR as a consequence of the discretization of the space. After the conflicting rules are removed, there is a module for unifying similar rules. This module follows the design criteria of the LR-FIR, intended to produce a compact representation of the data. Finally, a filtering process that eliminates "bad quality" rules is applied to the rule set.

### 3.5.7 Mixed fuzzy rules

(Berthold, 2003) presents an induction algorithm that learns fuzzy rules directly from examples, without the need of predefine fuzzy partitions for the class and attributes. This algorithm is influenced by the ideas behind fuzzy graphs. According to (Zadeh, 1999), fuzzy graphs are a way to represent "imprecisely" defined dependencies among variables. The general idea is to use fuzzy sets to represent the relationships among variables (e.g among sets of attribute values and their labels).

When working with labeled datasets, fuzzy sets allow one to extend the crisp information to regions. For example, if only one point of class $A$ is given, it represents all of the known space. A model of this single observation will only contain that point with its associated class. Let us suppose that another instance of class $B$ "arrives". The ground truth that we have now is that one point in the space has class $A$ and the other class $B$. We do not know anything about the space between (or beyond) those points. However, if we assume that the classes are continuous throughout the space (other possibilities can also be considered but this is probably the most intuitive), it is possible to use fuzzy sets to model how the transition between the classes occurs. For example, we can assume that the membership to each class decreases as we move away from those points whose class is known and vice versa. Figure 3.4 shows how the values of a hypothetical attribute $x$ relate with membership to classes $A$ and $B$.

Using fuzzy sets, it is possible to approximate functions, contours and relationships;

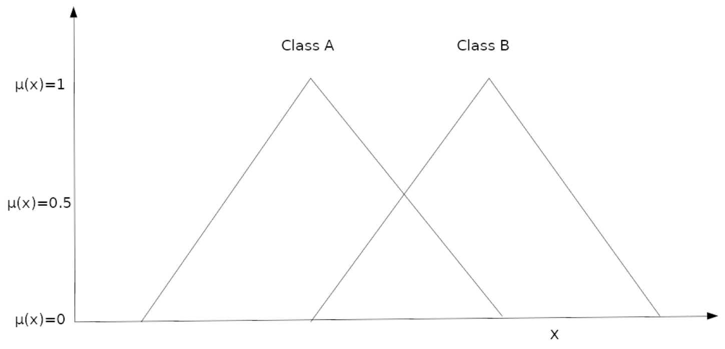

Figure 3.4: Two fuzzy sets describing the membership of an $x$ value to class $A$ and $B$.

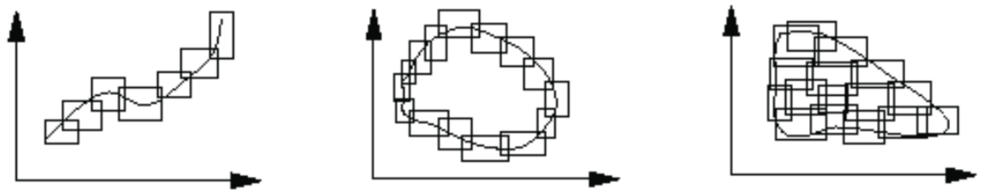

Figure 3.5: From left to right, approximate representation of a function, a contour and a relationship. Image taken form (Berthold and Huber, 1999).

this is shown with rectangular shapes in Figure 3.5. Although neural nets have proven to be very good approximators, algorithms that create fuzzy graphs from examples have lower complexity and the capacity to produce more interpretable outputs without the need of algorithms to extract fuzzy rules out of neural nets (briefly described in Section 3.6).

Many algorithms for learning fuzzy rules directly from examples have been proposed (Mordeson and Nair, 2012). They use different approaches either to simplify the input variables or to learn its structure. However, these approaches either produce complicated rule sets or need to adjust an apriori grid to the classes and attributes. To create these grids we need to know in advance into how many fuzzy sets or classes each parameter is divided. This task requires one to analyze the input data to infer and adjust the membership functions, increasing the complexity of the algorithms. Other algorithms consider equipartitions of the input space, then optimize the "right" number of partitions for each parameter.

In (Berthold, 2003), the general idea is to learn from the examples as they arrive, adjusting the existing model to the new evidence. A software implementation of the algorithm is available at (Berthold et al., 2013). The algorithm uses trapezoidal membership functions for the rules. Each membership function has a *core* and a *support*. The core of a membership function for a fuzzy set $A$ is defined as that region of universe satisfying that: $\mu_A(x) = 1$ (i.e, all elements $x \in X$ that have a complete membership to the set $A$). The support of a membership function for a fuzzy set $A$ is composed by those elements $x$ of the universe, such that $\mu_A(x) > 0$ (i.e, all elements with non-zero membership to the fuzzy set $A$).

Let us suppose that we have three attributes with respective hypothetical dimensions:

$$D_1 = [-10, 100]$$

$$D_2 = \{\mu_{low}, \mu_{high}\} \text{ with } \mu_{low}(x) = 1 \text{ for } x < 10 \;\; \mu_{high} = 1 - \mu_{low}$$

$$D_3 = \{red, black, blue\}$$

Then, a possible rule within that space could have *Supports*: $C_1^{supp} = [35, 41), C_2^{supp} = true$, and $C_3^{supp} = \{red, black\}$. And *Cores*: $C_1^{core} = [40, 41)$, $C_2^{core} = \{\mu_{low}\}$, and $C_3^{core} = \{black\}$. When the support of any attribute equals *true* (e.g $C_2^{supp} = true$), it indicates that there is no evidence restricting that dimension.

The algorithm works incrementally; every time a new datum is included, the rules are analyzed and modified accordingly. The algorithm keeps two parameters for internal use: $w$, which counts how many patterns are explained by a rule, and $\overrightarrow{\lambda}$, which remembers the training example that created the rule. The algorithm has three cases that allow modification of the rule set each time a new example $(\overrightarrow{x}, class)$ arrives.

- Covered. If the new pattern is covered by a rule with the same class and does not lie in the core, the core of the rule is extended. The extension covers the old core together with the contribution of the instance. Also, the $w$ of the rule is increased.

- Commit. If no rule of the same class covers the pattern, a new rule is created. The new rule is created as follows: $C_i^{supp} = true$, $\forall\ i$ and the core of the rule equals $\overrightarrow{x}$. The parameter $\overrightarrow{\lambda}$ equals $\overrightarrow{x}$ and $w = 1$.

- Shrink. If the new pattern touches a rule with a different class, there are two possibilities:

  1. If the new pattern lies in the support but not in the core, only the support is shrunk.
  2. If the pattern lies in the core, we have to shrink both core and support. For this, we analyze all features, and we shrink the one causing the minimum volume loss.

The algorithm runs several epochs. After presenting all the patterns, the rules are reset in the following way: for each rule, core $= \overrightarrow{\lambda}$, $w = 0$ and its supports are kept. With this configuration, a new epoch starts. The model's train finishes when, throughout the epochs, the rule base does not change further.

The “Mixed fuzzy rules” algorithm is able to manage different data types and small data as well. Its incremental nature, in the way it adds new information by adjusting the existing rule set, inspired the algorithm presented in Chapter 6. However, it has two characteristics that limit its use from the perspective of the tasks addressed in this manuscript. First, the algorithm has no parameters to configure. This offers simplicity but makes the algorithm a “one task tool”, which does not provide the user with the opportunity to try different induction levels by controlling algorithm parameters. Second, the result of the algorithm depends on the order of the attributes of the input data. Therefore, the level of generalization for a given attribute might be different according to its position. This forces the user to have apriori knowledge of the attributes functionalities, to decide in which of these to seek more generalization.

## 3.6 Rule Learning Algorithms and Neural Networks

Models built by neural networks have great performance but are hard to interpret. This has driven research in algorithms to construct symbolic rules out of trained neural nets (Hailesilassie, 2016; Murdoch and Szlam, 2017). The neural network creates a model from which the rule extraction algorithm derives a symbolic model. Another approach that connects neural nets and rule learning consists of training an inference system using a neural net, as is the case of Adaptive-Network-based Fuzzy Inference System (ANFIS, Jang, 1993). Below, the general ideas behind these algorithms are presented. Although the combined use of neural nets and rule extraction algorithms still does not allow real-time feedback, reviewing its basics provides context of their possibilities.

### 3.6.1 Global algorithms

Consider the most general case, a binary classification in which all attributes take binary values. Then, the space of possible inputs is given by the set of different combinations that we can have where the attributes and the class take values 0 or 1. In such a case, the performance of any trained network (having weights 0 or 1) would be equivalent to a set of logical rules found by searching the space of attributes and class combinations. To simplify this search, it is possible to use heuristics, such as restricting the number of considered antecedents, the number of attributes that can take value of 0, etc.

An extension of this general approach that is able to handle intervals and continuous inputs is the Validity Interval Analysis (VIA, Thrun, 1993). It extracts symbolic if-then rules by analyzing the input output behavior of feedforward neural networks. VIA searches for intervals with maximum activation range for each input, optimizing the intervals by using linear programming. Then, these intervals (which can be considered rule-like knowledge) are propagated throughout the network (either forward or backward).

VIA relies on a rule refinement algorithm that works on these created rules with the form "if the input of the neural network is in the region $R_i$, then its output is in the region $R_0$". The created regions are axis-parallel hypercubes defined by the intervals. The refinement algorithm iteratively checks rules for inconsistency until it obtains a final set. This process can be computationally expensive, so the extraction of the rules requires time.

Neural network inversion methods seek to find one or more input values that produce a desired output response given a fixed set of weights. The general ideas on inverting feedfoward networks for symbolic rule extraction are clearly presented in (Jensen et al., 1999). In (Hernández-Espinosa et al., 2003), an algorithm based on interval arithmetic that works for particular target outputs is presented. It can create rules with N-dimensional intervals in the input space.

The algorithm works on trained neural nets in the following way:

1. Select a point and calculate its output for the neural network.

2. Select an interval vector (target) that agrees with the classification class of the output of the neural network.

3. Apply the inversion algorithm and extract a rule.

4. Select a new point (not included in the rules already obtained) and calculate the neural network output for this new point.
5. Select a target that agrees with the output of the neural network.
6. Apply the inversion algorithm and extract a new rule.
7. Repeat from 4 until the input space is covered.

Generally speaking, neural network inversion algorithms can be broadly classified within three categories: 1) exhaustive search 2) single-element search; and 3) population-based evolutionary methods that operate on a population of potential solutions. They have pros and cons. For example, exhaustive search is feasible when both the input dimension and the range of the input variables are small. In single-element search, one search point explores the space defined for fixed weights as a function of the inputs. The search can be performed, for example, using gradient descent error backpropagation. Evolutionary methods seek to minimize the objective function using populations of points, resulting in numerous solutions. These approaches, depending on the search space, can also be computationally expensive.

In general, the algorithms for rule extraction differ in the type of extracted rules and the searching strategy. However, they have a common problem, their computational cost increases exponentially with the number of parameters in the neural network (weights or neurons). So, it is common to apply pruning algorithms to reduce the size of the network before the application of the rule extraction method. Pruning algorithms might create a coarse representation of the space, discard some weights, etc.

### 3.6.2 Local algorithms

Local algorithms do not analyze the whole network but parts of it, normally a single hidden neuron. The general idea is to consider the inputs of the analyzed node and to understand how the other neurons contribute to its activation. These actions are then translated into rules.

To understand the idea behind these algorithms, consider the simplest case: using step functions such that the output of each neuron is binary. Then, we aggregate the contributions reaching a neuron using a sigmoidal function. As sigmoidal functions preserve (or reverse) the order of the input-output values, and its range equals $[0, 1]$, the contributions to the neuron only depend on the sign of the weights. In this way, the contribution of the network nodes to the single neuron can be estimated. Analyzing the network for rules with these conditions is equivalent to considering the $2^n$ possible combinations of activation or non-activation of the neurons. Once again, there is a need to restrict the search. For example, the weights can limit the search tree by identifying the nodes that contribute more to the rule antecedents. Some approaches analyze the largest weights and whether they provide a sufficient explanation for the activation of the selected, hidden neuron. Variations of these algorithms can be found in the literature of local searches, see for example (Chorowski and Zurada, 2011).

Some general considerations regarding global and local algorithms are: 1) Before applying any algorithm the neural net has to be trained. 2) They simplify the input output space to offer interpretable and compact representations. As is discussed in Chapters 2 and 4, live coding has different requirements (e.g real-time feedback or being able to work directly with the synthesizer parameter values) although these design considerations are useful in some application domains. In (Bell, Dec. 9, 2014; Paz, April 3, 2020), the synthesizer parameter values are shown to the audience both as an aesthetic artifact and to communicate the decisions made during the performance.

### 3.6.3 M-of-N rules

M-of-N rules are naturally implemented by analyzing network nodes and estimating which are the more influential links for a particular neuron (discarding or simplifying the others). To do this, the network is simplified. Some algorithms, for example, group similar weights and replace them by their mean. Groups that have little influence can be eliminated. In such a way, the network is simplified and more simple rules can be extracted by the network analysis. One of the algorithms using this technique is the Knowledge-Based Neural Network algorithm (Towell et al., 1990). Another possibility is simplifying the neural net via regularization. This can be applied either in the cost function or in the sum of the absolute values of the weights. For example, if only weights smaller than a certain threshold are allowed, the hidden units become inactive or completely active. See for example (Géczy and Usui, 1999). In the MLP2NL algorithm, the weights of an MLP hidden layer are forced to be {-1,0,1}.

Incremental algorithms, such as "Rule extraction as learning" (Craven and Shavlik, 1994) and RULENEG (Andrews et al., 1995), create one cunjunctive rule for each input pattern (adding it to $R$). Then, each time a new training input is not correctly classified by an existing rule in $R$, a new rule is created and possible contradictions adjusted.

Other rule-based approaches, (e.g Tickle et al., 1994; Craven and Shavlik, 1996) use the trained network as a predictor to compare variations of the training set in order to find the minimal information necessary in a pattern for it to be distinguished by the network.

### 3.6.4 Neuro-fuzzy systems

Neuro fuzzy systems are fuzzy systems that are trained with Neural Networks Algorithms. For example, a fuzzy controller can be represented using the structure of a neural net and then trained using algorithms such as backpropagation or genetic. In the Mamdani approach, the Neuro-fuzzy system has the following components: Input layer, Fuzzification, AND operation, Fuzzy inference and Defuzzification. These layers can be seen in Figure 3.6, which describes a control system for an intelligent autonomous robot in (Pratihar and Pratihar, 2017).

Neuro-fuzzy systems were developed to overcome concerns about the computational speed and complexity required for building interpretable predictive models. The most used inference system is the Takagi-Sugeno-Kang (TSK), which is capable of mod-

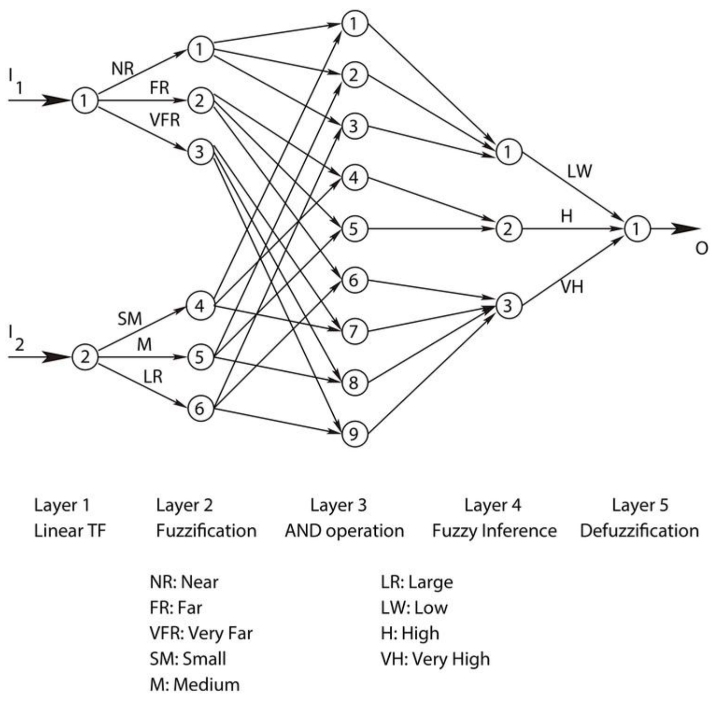

Figure 3.6: Distinct layers of a Mamdani-type neuro-fuzzy system. In the Fuzzyfication layer the linguistic categories established for each variable can be appreciated.

elling non linear dynamic systems by combining sublinear models. For instance, the Adaptive neuro fuzzy inference system (ANFIS) uses TSK. It defines parameter sets for premises and consequences and then uses the inference system to define the fuzzy if-then rules that relate the sets. A complete survey of Neuro-fuzzy systems is presented in (Shihabudheen and Pillai, 2018). The authors classify the techniques in five general categories based on their learning algorithm. These are: Gradient, Hybrid, Population-based, Extreme learning machine based and SVM based.

### Gradient descendent

Gradient descendent neuro-fuzzy systems employ either TSK or Mamdani inference models, training their parameters by means of backpropagtion (or variations of the algorithm). For example, it is possible to learn all the parameters of the membership functions, consequents and feedback structure of a recurrent neural net using ordered derivative back-propagation. Many neuro-fuzzy systems are used for control. For example, from all the systems using backpropagation in (Shihabudheen and Pillai, 2018) only (Chakraborty and Pal, 2004) is used for classification. It allows the performance of feature selection and learning a classifier. The architecture uses a four-layered feed-forward network to build a fuzzy rule-based classifier, training the network with backpropagation in three phases of learning and pruning.

Gradient descendent-based methods are generally very slow, because they normally have improper learning steps. For example, in (Chakraborty and Pal, 2004), for the first step the network, learns the important features, and then it is pruned. Also, they can easily get trapped in local minima, for which iterative learning steps are required.

### Hybrid neuro-fuzzy systems

Hybrid neuro-fuzzy systems use two or more learning techniques to find the parameters of the network, fuzzy sets, etc. Most of the systems are self-organizing neuro-fuzzy systems. These methods, although capable of easily escaping local minima, require time to be properly trained. In (Karaboga and Kaya, 2019), a survey on approaches to Adaptive network-based fuzzy inference system training is presented.

### Population based neuro-fuzzy systems

When backpropagation is used to train the network, the algorithms can fall into local minima. These algorithms are also hard to apply when the derivative is hard to obtain. In such cases, population based neuro-fuzzy system are an alternative. Among population based systems, architectures are basically all types of evolutionary algorithms (e.g genetic algorithms, differential evolution, ant colony optimization, artificial bee colony and particle swarm optimization). These algorithms are used to optimize the network and the parameters of the membership functions. An extensive survey on these systems is presented in (Shihabudheen and Pillai, 2018).

### Suport vector neuro-fuzzy systems

Support vector machines gained a great deal of attention given their powerful classification capacities. The main difference between them and other classifiers is, instead of minimizing the squared error, they search for vector points that define the decision boundary. To do that, they look for a decision border that maximizes the separation boundary, while, at the same time, minimizing the number of misclassified points (this process is controlled by the regularization parameter). SVM can be also used for regression problems by means of the $\epsilon$-insensitive loss function, which is required to solve a quadratic programming problem normally involving a great number of parameters to optimize. Different proposals to simplify this problem have been presented. In particular, in Fuzzy support vector machines, a fuzzy membership function is assigned to each input point. The fuzzy nature implies that different input points, with distinct memberships, contribute in a different way to the decision surface. As an example of support vector machines and neuro-fuzzy systems, see (Miranian and Abdollahzade, 2012). Therein, a local neuro-fuzzy approach based on the architecture of least-squares support vector machines is proposed for the analysis of chaotic time series.

### Neuro-fuzzy systems based on extreme learning machines

In extreme learning machines (ELM), the parameters of hidden networks are selected randomly while the parameters of the input output neurons are analyzed using minimum norm least-squares estimation. This choice, reduces the computational time (in comparison with feedforward networks). ELM seek to obtain the smallest training error together with the smallest norm of the output weights. If conventional networks are replaced by ELM, we obtain an ELM based neuro-fuzzy approach. In the majority of the ELM neuro-fuzzy systems, the membership function of the fuzzy system can be tuned by ELMs reducing the tuning time required to tune the membership functions in other approaches. (Shihabudheen et al., 2018) present a system based on particle swarm optimization for extreme learning neuro-fuzzy systems (tested for regression and classification), and (Pillai et al., 2014) present extreme learning ANFIS for control applications.

## 3.7 Conclusion

Symbolic rules provide high-level (human-readable) descriptions of systems behavior. Inductive algorithms are able to build general models out of label observations. In the context of this research, single parameter combinations labeled by the user provide the type of data needed for inductive learning. The most common algorithms for inductive learning of symbolic descriptors are decision trees and symbolic rule learners. However, decision trees are more complex to understand than rule learning algorithms.

In a live coding context, we seek interpretable models, built through a generalization process that allows real time feedback and is controllable by means of meaningful parameters. Therefore, allowing us to build different models mid-performance, while exploring different degrees of variation of the input data.

Many rule learning algorithms have been proposed. However, the tasks for which they are designed, to automatically build a general compact model of the system under study, make these algorithms not directly applicable to the tasks addressed here.

Many rule learning algorithms require coarse partitions of the input and output spaces to be defined by the user. This requires knowledge of the input/output variables. Other systems learn these partitions, but they have to iteratively optimize them, which is time consuming and real time feedback is lost. Most of the algorithms seek to create as few rules as possible (normally selecting the more general ones by pruning) as they are designed to process big data sets. In such cases, general patterns are preserved but variability might be lost within the process.

Algorithms for extracting symbolic rules out of trained neural nets have been proposed. Neural networks are great function approximators. However, most algorithms required the input data to be continuous given the numerical way the neural nets operate. Some algorithms only operate with binary data, while others have to discretize the inputs before processing them. Once again, the objective of the algorithms is to simplify the underlying neural model by approximating it by logical rules. For that, a great part of the algorithm focusses on finding good linguistic variables that translate the underlying network into symbolic rules.

Neuro-fuzzy systems are versatile but hard to apply in in real time. In addition, many of the systems assume that the number and shape of the membership functions is known in advance. Whenever the shape and number of the membership functions is not known in advance, these can be learned or adjusted (e.g Alcalá-Fdez et al., 2009). This, on the other hand, increases the computational cost as well as the complexity for the operation of the algorithms.

Moreover, even knowing the right number of membership functions some continuity in the parameters is assumed, i.e low, medium, high. Synthesizers' parameters do not always satisfy this property. For instance, a parameter describing the wave form can take values: Sinusoidal, Sawtooth, Triangular or Square. Furthermore, if clustering algorithms are used to infer the membership functions, such clusters can rapidly change with the introduction of new data.

Generally speaking, the training times and the software complexities of the neuro-fuzzy systems are so big that there is ongoing research looking for hardware implementations. See, for example, the work by (Bosque et al., 2014), which presents a historical review of hardware implementation and platforms for fuzzy systems, neural networks and neuro-fuzzy systems.

Considering the restrictions imposed by on-the-fly artistic programming, symbolic rule learning algorithms that do not assume previous knowledge of the variables and that allow control of the induction process inspired the algorithms of Chapters 4 and 6. These algorithms are described in (Berthold, 2003; Castro et al., 2011). Essentially, two things were sought. First, the algorithm needed to allow for control of the amount of induction to be carried out on the input data. Second, the user must not need prior knowledge of the system under study (although it is OK if she does!).

# Chapter 4

# Synthesizer Programming with Rule Learning (RuLer)

This Chapter introduces the rule learning algorithm "RuLer", proposed in this manuscript for on-the-fly synthesizer programming. The Chapter starts by discussing the use of inductive rule learning for synthesizer programming and the algorithm requirements from the live coding perspective. Then, the algorithm is presented alongside basic rule extraction examples to give the reader a taste of its possibilities. The Chapter finishes with the algorithm evaluation, including two parts: the first focuses on subjective perception and includes user tests, in which experienced livecoders evaluate the new generated instances, as well as a section describing the performances, recordings and criticisms received. The second part focuses on numerical analysis of the RuLer inductive capacities, by using it for oversampling tasks. This part is presented, given its extension, in Chapter 5.

The phrase "live coding" implies programming sound synthesis algorithms in real time. To do this, one possibility is to have an algorithm that automatically creates variations out of a few presets[1]. However, the need for real-time feedback and the small size of the

[1]A preset is a configuration of a sound synthesis algorithm together with a label describing the resulting sound selected by the user (Paz, 2019b).

data sets, which can even be collected mid-performance, act as constraints that make existing automatic synthesizer programmers and other learning algorithms unfeasible to use. Furthermore, the design of such algorithms is not oriented to create variations of a sound but to find the synthesizer parameters that match a given one.

Inductive learning of symbolic rules allows one to build general models from single labeled observations. Nonetheless, the restrictions imposed by the live coding practice suggest the need for algorithms that do not assume previous knowledge of the variables, provide real-time feedback and allow control of the induction process. This Chapter builds the proposal of a symbolic rule learning algorithm upon these restrictions.

Experienced live coding practitioners will of course notice that it is possible to modify the synthesizer on-the-fly, which is basically modifying the instrument during the performance. In this case, the synthesizer's parameters or its function may change. This can always be done and, as some authors point out, the performer must be willing to welcome surprises both good and bad. Nonetheless, for the purposes of this study, and as all synthesizer programming literature does, we will assume that the instrument does not change.

As an example of how an algorithm can be rewritten, consider the following three modifications of a SuperCollider node proxy named "\x" that contains a synthesis algorithm:

```
Ndef(\x,{
            arg freq, freq1, amp;
            var sig = SinOsc.ar([freq, freq + 1, freq1, freq1 - 2],0, amp);
})

Ndef(\x,{
            arg freq, freq1, amp, mix, room, damp;
            var sig = SinOsc.ar([freq, freq + 1, freq1, freq1 - 2],0, amp);
            sig = FreeVerb.ar(sig, mix, room, damp)
})
```

```
Ndef(\x,{
            arg freq, freq1, amp, mix, room, damp;
            var sig = Saw.ar([freq, freq + 1, freq1, freq1 - 2], amp);
            sig = FreeVerb.ar(sig, SinOsc.kr(mix), SinOsc.kr(room), SinOsc.kr(damp))
})
```

The first *Ndef* has parameters *freq, freq1*, which control two sinusoidal oscillators, and *amp*, which controls their amplitudes. In the second *Ndef*, a reverb "*FreeVerb*" with parameters *mix, room* and *damp* is added. Finally, in the third *Ndef*, parameters *mix, room* and *damp* control sinusoidal oscillators that modulate the parameters *mix, room* and *damp* of the *FreeVerb*. In addition, the sound wave is not sinusoidal but sawtooth.

By tweaking parameters, the live coder explores, categorizes and selects the appropriate combinations for different musical contexts. Thus, a piece can be seen as a succession of combinations creating its own spatial and temporal structures. From this perspective, coding a piece on-the-fly requires guiding the sound by changing the parameter settings. Such a task, as discussed in Section 2.1, imposes cognitive challenges due to the huge size of devices' parameter spaces and due to the possibility of non-linear sound variation built-in within them (Dahlstedt, 2001b). Therefore, one possibility is to have some pre-selected parameter combinations, of which the aural result is known as a starting point for the performance. However, when only a few combinations are used, the performance can quickly become repetitive and boring for the listener, and again, remembering many combinations imposes practical challenges. Generative algorithms (McLean and Dean, 2018) have been used to address this problem by automating the production of low-level material, while the coder works on guiding the high-level evolution of the piece.

This section introduces "RuLer", a rule learning algorithm that inductively generalizes labeled combinations of parameters. The labels are linguistic descriptors for the perceptual characteristics of the sound, the intended musical context for each combination, etc. The algorithm produces different degrees of variations of the input material controlled by the algorithm parameters. For that, the algorithm searches for patterns based on a dissimilarity function and constructs new rules that generalize the input data. The production of rules (given the "create rule" function used in Section 4.1.4) performs crossover mutations of the original material in a similar way as genetic algorithms do. The tensions and relaxations created by the recombination levels of the material can be used to conduct the dynamicity of the sound. From this perspective, the algorithm accompanies a directed search in a vast space of possibilities, which can be used as a compositional tool.

## 4.1 Inductive Rule Learning for Automatic Synthesizer Programming

### 4.1.1 Algorithm requirements

As discussed previously, the context for which the algorithm was developed imposes specific requirements:

- Capacity to work with different data types, e.g numerical, categorical, etc. This is because sound synthesizer parameters can be of different types. For instance, the frequency of an oscillator is numerical, while the waveform selection is categorical (e.g sinusoidal, sawtooth, triangular, or square).
- Production of good results for small datasets. If the parameter combinations are selected by a human, the bottleneck is given by its capacities to collect data. Besides, there are occasions when we want to extend a set of combinations that already exists (as is the case in the example presented in Table 4.6, which has only thirteen presets). In this case, the problem is not the inability to gather more data but that the data sets are small by nature.
- Order independence. If the output depends on the order of the input data, the level of generalization might not be the same for all the variables. For example, if

pattern-seeking comparisons are performed by pairs, and the compaction process eliminates information to simplify the rule, there might be patterns that will be lost in this process. As devices parameters modify the sound with different degrees of expressiveness[2], the user would need to have the musical knowledge (of the parameters and their interactions) to sort the data accordingly and get the “best model”.

- Control of the induction process. As the algorithm is intended to be, in the sense described in (Magnusson, 2019), an extension of our expressive capacities, its parameters have to provide the user with simple and interpretable controls of the induction process. Specifically, to define how much new material is included in the resulting rules and how dissimilar two presets can be to be used for the induction process.

- Production of an interpretable output since it is a tool to support the composition and analysis of music.

- Allowing of real-time feedback.

### 4.1.2 RuLer: a rule learning algorithm

RuLer is an inductive rule learning algorithm designed in the context of live coding for automatic synthesizers programming (Paz, 2019b). It takes as input a set of labeled presets (e.g “intro” for the introduction of a piece or “harsh” as a linguistic label). Then, from those presets, a set of IF-THEN rules generalizing them is obtained. The generalization process is based on the patterns found through the iterative comparison of the presets. To compare the presets, a dissimilarity function receives a pair of them and returns *True* whenever they are similar enough according to the specific form of the function and a given threshold. The dissimilarity threshold ($d \in \mathbf{N}$) is established by the user. The algorithm works as follows:

[2]Think, for example, of the parameters controlling the amplitude and the frequency of an oscillator.

- Each preset is considered an IF-THEN rule and represented as an array of size $N$. Its first $N-1$ entries (the rule antecedents) correspond to one parameter of the synthesis algorithm, and the last entry to the label is assigned to the combination (rule consequent). For example, a rule r = [{3}, {5}, intro] is read in the following way: "if the first parameter takes a value of 3 and the second a value of 5, then the preset label is intro". A rule $r$ = [{1,2,3}, {7}, . . . , {3}, intro] is read as: IF $r$[1] = 1 OR 2 OR 3 AND $r$[2] = 7 AND . . . AND $r$[N−1] = 3 THEN, label = intro. The rules are stored in a list, so they can be accessed by its index.

  The algorithm iterates as follows, until no new rules can be created:

  1. Take the first rule from the rule set (list).
  2. Compare the selected rule with the other rules using the dissimilarity function (Section 4.1.3). If a pattern is found, (i.e. the rules have the same class and the dissimilarity between them is less than or equal to the threshold $d$ established by the user), create a new rule using the *create_rule* function (Section 4.1.4).
  3. Eliminate the redundant rules from the current set. A rule $r_1$ is redundant with respect to a rule $r_2$ (of the same class) if $\forall\, i \in \{0, . . . \text{N}-1\}$, $r_1[i] \subset r_2[i]$. Note that other definitions of redundancy can be used.

- Append the created rules to the end of the rule set.

### 4.1.3 *Dissimilarity* function

The *dissimilarity* function receives two rules $(r_1, r_2)$ together with a threshold $d \in \mathbf{N}$ and returns *True* if the rules have the same category and *dissimilarity*$(r_1, r_2) \leq d$. It returns *False* otherwise. The parameter $d$ is an input parameter of the algorithm.

The *dissimilarity* function, currently implemented in the RuLer algorithm, counts the number of empty intersections between the sets of the corresponding entries in the rules. For example, if $r_1$ = [{1}, {3,5}, intro] and $r_2$ = [{1,3}, {7,11}, intro], *dissimilarity*($r_1$,$r_2$) = 1. If $r_1$ = [{1}, {3,5,7}, intro] and $r_2$ = [{1,3}, {7,11}, intro], *dissimilarity*($r_1$,$r_2$) = 0.

### 4.1.4 *Create_Rule* function

This functions receives pairs of rules $r_1, r_2$, satisfying that *dissimilarity*$(r_1, r_2) \leq d$. Then, it creates a new rule according to the way it is defined. The function currently used creates a new rule by taking the unions of the corresponding sets of the rules received. For example, if $r_1$ = [{1}, {3,5,7}, intro] and $r_2$ = [{1,3}, {7,11}, intro], then $r_{1,2}$ = [{1,3}, {3,5,7,11}, intro]. The candidate rule is accepted if the following conditions are met:

1. No contradictions (i.e., rules with the same parameter values but different label) are found during the generalization process.

2. From all the possible presets contained in the candidate rule, the percentage of them contained in the original data are greater than or equal to a *ratio* $\in$ [0,1]. This number is also an input parameter of the algorithm defined by the user. A *ratio* = 1 implies that 100% of the instances contained in a candidate rule have to be present in the input data for the rule to be accepted, while a *ratio* = 0.5 needs 50% of the instances, etc. For example, for the previous candidate rule $r_{1,2}$ = [ {1,3}, {3,5,7,11}, intro ] all the possible presets contained in $r_{1,2}$ are: [ {1}, {3}, intro ], [ {1}, {5}, intro ], [ {1}, {7}, intro ], [ {1}, {11}, intro ], [ {3}, {3}, intro ], [ {3}, {5}, intro ], [ {3}, {7}, intro ], [ {3}, {11}, intro ]. If a *ratio* = 1 is defined, all eight combinations should exist in the input data for the rule to be accepted. If *ratio* = 0.5, half of them should be present in the original presets.

### 4.1.5 Domain specific functions

Note that the *dissimilarity* and *create_rule* functions can be changed according to the objects being compared and the desired generalization. For example, for harmonic objects, we probably want a dissimilarity that looks at harmonic content, while for rhythms, temporal factors need to be addressed. See (Toussaint, 2004), for a comparison of rhythmic similarity measures.

The pseudocode for the rule extraction process is shown in Algorithm 3.

## 4.2 Simple Rule Extraction Examples

### 4.2.1 Example 1: setting d = 1 and ratio = 1

Given $r_1$, $r_2$, $d = 1$ and ratio = 1, the algorithm creates a new rule if:

1. The dissimilarity between $r_1$ and $r_2 \leq 1$.
2. All the instances contained in the new rule are present in the original input data.

Table 4.1 shows a rule extraction example for this setting, using a set of combinations, all categorized as *intro*, with only two parameters.

The original dataset is composed of four rules numbered $r_1$, ... $r_4$. The algorithm starts by comparing $r_1$ with $r_2$, $r_3$ and $r_4$. Then, it compares $r_2$ with $r_3$ and $r_4$, and so on. The created rules are appended (in order) at the end of the rule set. Their subindexes represent the rules from which they where created. When the last two rules of the current rule set have been compared (in this case $r_3$ and $r_4$), the redundant rules are eliminated. The rule set, located after eliminating the redundant rules of the first iteration, contains the original rules plus those that have been created during the iteration. In this case, the first created rule is $r_{1,3}$, as $r_{1,2}$ cannot be created as dissimilarity($r_1$,$r_2$)= 2.

After deleting the redundant rules, a second iteration, now using $r_{1,3}$, $r_{1,4}$, $r_{2,4}$ and $r_{3,4}$, is performed. Comparing the first two rules, $r_{1,3}$ and $r_{1,4}$, rule $r_{1,3,4}$ is created.

**Algorithm 3** Rule extraction process

```
1: function RULE_EXTRACTION_ALGORITHM(rules, d, ratio)
Require: rules, d:$\in \mathbb{N}$, ratio $\in [0,1]$
2:     newRules $\leftarrow$ [ ]
3:     for $i \leftarrow 0$ to size of rules do
4:         r1 = rules[i]
5:         for $j \leftarrow i+1$ to size of rules do
6:             r2 = rules[j]
7:             pattern = dissimilarity(r1, r2, d)
8:             if pattern then
9:                 rule = create_rule(r1, r2, ratio, rules)
10:                if rule then
11:                    newRules.append(rule)
12:                end if
13:            end if
14:        end for
15:    end for
16:    rules.append(newRules)
17:    rules = delete_redundant(rules)
18:    return rules
19: end function
```

Next, comparing rules $r_{1,3}$ with $r_{2,4}$, rule $r_{1,2,3,4}$ = [ {1,2,3}, {1,2}, intro ] is created. However, as ratio = 1, all the rules contained in the new rule must be present in the original input, which is not the case for [ {3}, {1}, intro ], so the rule is rejected. Comparing the rest of the rules, no new rules are created. By eliminating the redundant rules after the second iteration, we get the final rule set. When comparing the two final rules, no new rules are created, so the algorithm ends. Note that in this example no generalization is produced.

### 4.2.2 Example 2: using d = 2 and ratio = 3/4

Table 4.2 shows a rule extraction example using d = 2 and ratio = 3/4 on the same data set of the example presented in Table 4.1.

In the first iteration, all the created rules are accepted. In the second iteration, comparing any two rules, the rule $r$ = [ {1,2,3}, {1,2}, intro ] is created, as in the previous example. This rule contains the instances shown in Table 4.3. Instances 1 and 5, marked with an "*", are not present in the original dataset. Since $\frac{4}{6} < \frac{3}{4}$, the rule is rejected; therefore, no new rules are created, and the algorithm ends. However, note that in this case a generalization is produced, since the combinations [{1}, {1}, intro] and [{3}, {2}, intro] that appear in the final rule set do not appear in the original rule set.

### 4.2.3 RuLer characteristics

The RuLer algorithm is designed to return all existing patterns, as its main intention is to offer all possibilities for creating new instances, expressing as rules those pairs of instances satisfying $dissimilarity(r_1, r_2) \leq d$. Therefore, it is possible for a single instance, let us call it $r_2$, to be included in more than one valid rule if $r_1, r_2$ and $r_3$ are single rules that satisfy: $dissimilarity(r_1, r_2) \leq d$, $dissimilarity(r_2, r_3) \leq d$ and $dissimilarity(r_1, r_3) > d$.

To illustrate this case, consider the dataset of Table 4.4.

Table 4.1: Rule extraction example using d = 1 and ratio = 1.

| **Original Rule set** | **Parameter values and category** | | |
|---|---|---|---|
| | $p_1$ | $p_2$ | ***category*** |
| $r_1$ | {1} | {2} | intro |
| $r_2$ | {2} | {1} | intro |
| $r_3$ | {3} | {2} | intro |
| $r_4$ | {2} | {2} | intro |
| **First Iteration** | **Parameter values and category** | | |
| | $p_1$ | $p_2$ | ***category*** |
| $r_1$ | {1} | {2} | intro |
| $r_2$ | {2} | {1} | intro |
| $r_3$ | {3} | {2} | intro |
| $r_4$ | {2} | {2} | intro |
| $r_{1,3}$ | {1,3} | {2} | intro |
| $r_{1,4}$ | {1,2} | {2} | intro |
| $r_{2,4}$ | {2} | {1,2} | intro |
| $r_{3,4}$ | {3,2} | {2} | intro |
| **Delete redundant rules after first iteration** | | | |
| **Input for Second iteration** | **Parameter values and category** | | |
| | $p_1$ | $p_2$ | ***category*** |
| $r_{1,3}$ | {1,3} | {2} | intro |
| $r_{1,4}$ | {1,2} | {2} | intro |
| $r_{2,4}$ | {2} | {1,2} | intro |
| $r_{3,4}$ | {3,2} | {2} | intro |
| **Second Iteration** | **Parameter values and category** | | |
| | $p_1$ | $p_2$ | ***category*** |
| $r_{1,3}$ | {1,3} | {2} | intro |
| $r_{1,4}$ | {1,2} | {2} | intro |
| $r_{2,4}$ | {2} | {1,2} | intro |
| $r_{3,4}$ | {3,2} | {2} | intro |
| $r_{1,3,4}$ | {1,2,3} | {2} | intro |
| **Delete redundant rules after second iteration** | | | |
| **Final Rule set** | **Parameter values and category** | | |
| | $p_1$ | $p_2$ | ***category*** |
| $r_{2,4}$ | {2} | {1,2} | intro |
| $r_{1,3,4}$ | {1,2,3} | {2} | intro |

Table 4.2: Rule extraction example using d = 2 and ratio = 3/4.

| **Original** | **Parameter values and category** | | |
|---|---|---|---|
| **Rule set** | $p_1$ | $p_2$ | ***category*** |
| $r_1$ | {1} | {2} | intro |
| $r_2$ | {2} | {1} | intro |
| $r_3$ | {3} | {2} | intro |
| $r_4$ | {2} | {2} | intro |
| **First** | **Parameter values and category** | | |
| **Iteration** | $p_1$ | $p_2$ | ***category*** |
| $r_1$ | {1} | {2} | intro |
| $r_2$ | {2} | {1} | intro |
| $r_3$ | {3} | {2} | intro |
| $r_4$ | {2} | {2} | intro |
| $r_{1,2}$ | {1,2} | {1,2} | intro |
| $r_{1,3}$ | {1,3} | {2} | intro |
| $r_{1,4}$ | {1,2} | {2} | intro |
| $r_{2,3}$ | {2,3} | {1,2} | intro |
| $r_{2,4}$ | {2} | {1,2} | intro |
| $r_{3,4}$ | {3,2} | {2} | intro |
| **Delete redundant rules after first iteration** | | | |
| **Input for** | **Parameter values and category** | | |
| **Second iteration** | $p_1$ | $p_2$ | ***category*** |
| $r_{1,2}$ | {1,2} | {1,2} | intro |
| $r_{1,3}$ | {1,3} | {2} | intro |
| $r_{2,3}$ | {2,3} | {1,2} | intro |
| **Final** | **Parameter values and category** | | |
| **Rule set** | $p_1$ | $p_2$ | ***category*** |
| $r_{1,2}$ | {1,2} | {1,2} | intro |
| $r_{1,3}$ | {1,3} | {2} | intro |
| $r_{2,3}$ | {2,3} | {1,2} | intro |

Table 4.3: Rules contained in rule $r$ at the top of the Table. Rules 1 and 5 marked with an "*" are not contained in the original ruleset of Examples 1 and 2.

| **Rules contained** | **Parameter values and category** | | |
|---|---|---|---|
| **in rule $r$** | $p_1$ | $p_2$ | ***category*** |
| $r$ | {1,2,3} | {1,2} | intro |
| 1 * | {1} | {1} | intro |
| 2 | {1} | {2} | intro |
| 3 | {2} | {1} | intro |
| 4 | {2} | {2} | intro |
| 5 * | {3} | {1} | intro |
| 6 | {3} | {2} | intro |

Table 4.4: Dataset to illustrate instances that appear in more than one rule.

| **Rule** | **Parameter1** | **Parameter2** | **Class** |
|---|---|---|---|
| $r_1$ | {3} | {2} | intro |
| $r_2$ | {2} | {2} | intro |
| $r_3$ | {1} | {2} | intro |
| $r_4$ | {2} | {1} | intro |

The RuLer algorithm with parameters $d = 1$ and $ratio = 1$ produces the rules: [{2}, {1, 2}, 'intro'], [{1, 2, 3}, {2}, 'intro']. These rules are shown, with their possible extensions in a solid line and a dashed line respectively, at the left of Figure 4.1.

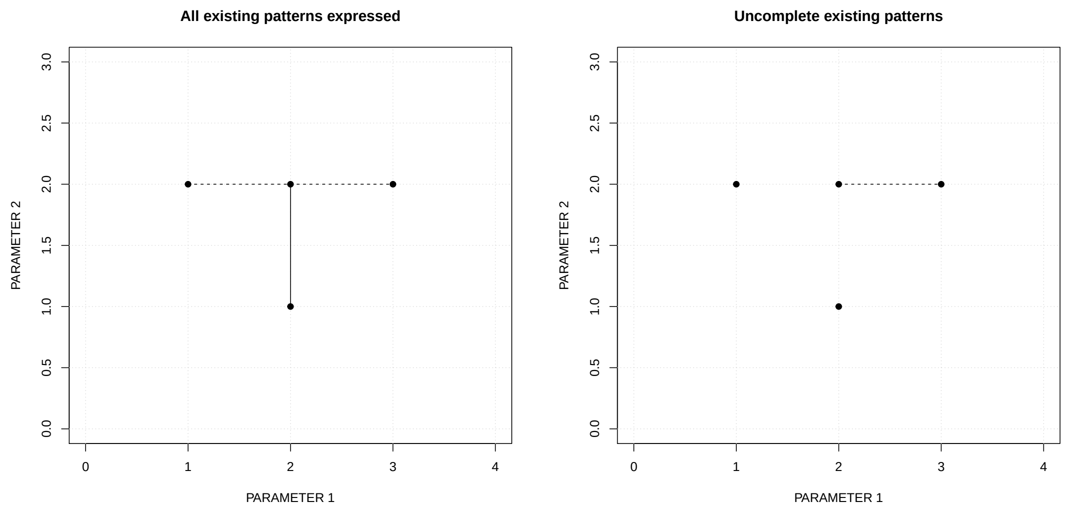

Figure 4.1: Resulting rules using data of Table 4.4 with their possible extensions in a solid line and a dashed line. Left, extracted by the RuLer algorithm with parameters $d = 1$ and $ratio = 1$. Right, extracted by using the Hamming distance $d = 1$ and, whenever a pattern is found, creating a new rule by taking the unions of the parameter values and eliminating the component rules.

Notice that the combination [{2},{2},'intro'] is present in both rules. As mentioned, if this were not the case, one of the patterns might fail to return to the user. To illustrate this, consider the same dataset and let us use the Hamming distance ($d = 1$) as the similarity function. Then, suppose that the create_rule function, whenever a pattern is found, creates a rule by taking the unions of the parameters of the respective rules and eliminating the component rules after producing the new one. With these conditions, comparing $r_1$ and $r_2$ produces the rule $r_{1,2}$ = [{2,3},{2},'intro']. This rule will not produce another rule when compared with the remaining data: $r_3$ = [{1},{2},'intro'] and $r_4$ = [{2},{1},'intro']. Therefore, the resulting rule set is: $r_{1,2}$ = [{2,3}{2},'intro'], $r_3$ = [{1},{2},'intro'] and $r_4$ = [{2},{1},'intro']. This is shown at the right of Figure 4.1. The resulting rule set does not express the existing patterns between [{2},{1},'intro'] and [{2},{2},'intro'] nor between [{1},{2},'intro'] and [{2},{2},'intro'] or [{3},{2},'intro']. To avoid this, the create_rule and the dissimilarity function were conceived to return all patterns found in the data.

Regarding how $d$ and *ratio* work, consider the simple set of individual rules presented in Table 4.5.

Table 4.5: Dataset to illustrate instances that appear in more than one rule.

| **Original presets** | | | |
|---|---|---|---|
| {1} | {4} | {6} | 'a' |
| {2} | {5} | {6} | 'a' |
| {3} | {6} | {6} | 'a' |
| **Rule set extracted with** $d = 2$ $ratio = 1/4$ | | | |
| {1, 2, 3} | {4, 5, 6} | {6} | 'a' |
| **Rule set extracted with** $d = 2$ $ratio = 1/2$ | | | |
| {1, 2} | {4, 5} | {6} | 'a' |
| {1, 3} | {4, 6} | {6} | 'a' |
| {2, 3} | {5, 6} | {6} | 'a' |

If $d = 2$ and $ratio = 1/4$, the single rule that models the dataset is in the middle section of Table 4.5. The number of allowed empty intersections among the single rules at the top of the Table is two. Then, every pair of rules can be compacted into a new rule during the process. As the ratio of single rules that have to be contained in the original data for any created rule is 1/4, the rule in the middle section can be created as it contains all the instances in the original data, which is 1/3 of the number of single instances of the rule (nine). Note that this is true if all seen values are: for the first attribute: 1, 2 and 3; for the second attribute: 4, 5 and 6; for the third attribute: 6.

If $d = 2$ and $ratio = 1/2$, the rule model extracted by the algorithm is presented at the bottom of Table 4.5. In this case, half of the instances contained in any created rule must be in the original data, and the rule at the middle of Table 4.5 cannot be created.

The parameter ratio is constant during the extraction process because it defines the level of generalization that the user of the algorithm wants to explore. The ratio allows for the extension of the knowledge base to cases that have not been previously used to build the model. If the user is more conservative, the ratio should be closer to 1. If the goal is to be more exploratory, lower ratios are needed.

The algorithm complexity serves to estimate its performance. If $m$ is the size of input data, the algorithm complexity is $O(m * (m - 1))$. This complexity considers the dissimilarity and create_rule functions described. This complexity is better than the previous versions of the algorithm presented in (Paz et al., 2016; Paz et al., 2017), given by $O(2^m - 1)$.

To create a rule, the algorithm presented in (Paz et al., 2016) compares the input data using the Hamming distance and requires that all the observed values of a parameter are present in the candidate instances to be compacted. For example, the following input data set: [1, 1, harsh], [1, 2, harsh], [1, 3, harsh], [1, 11, harsh] with $d = 1$, will produce the rule: IF the first parameter has a value of 1, THEN the class is harsh. But if the set is: [1, 1, harsh], [1, 2, harsh], [1, 3, harsh], [1, 11, **soft**], it cannot be compacted by this algorithm (as there exist the combination [1, 11] with class **soft**).

The algorithm described in (Paz et al., 2017) overcomes this limitation by requiring the user to define compaction intervals. This algorithm uses the Hamming distance and, for each interval, requires that all the observed values of a parameter are present in the candidate instances to be compacted.

For example, the input data set of [1, 1, harsh], [1, 2, harsh], [1, 3, harsh], [1, 11, soft] with the intervals [1-10], (10-20] for the second parameter and $d = 1$ would create the rules: IF the first parameter has a value of 1, THEN the class is harsh; and IF the first parameter has a value of 11, THEN the class is soft. That is, the algorithm compacts the instances only if the intervals selected by the user separate the classes *harsh* and *soft*. If, otherwise, the intervals do not separate the classes (e.g using interval [1-20] for the second parameter), the algorithm does not produce any rules.

## 4.3 Evaluation

### 4.3.1 Evaluation of automatic synthesizer programming algorithms

Automatic synthesizer programming algorithms are usually evaluated by listener surveys (in the sense described by Ariza, 2009) or by estimating the error between a target sound and the algorithm output. Some authors have performed empirical evaluations, based on the John Henry Test (Ariza, 2009), in which human sound designers compete with the programming algorithms. Examples can be found, respectively, in (Mitchell, 2010; Yee-King et al., 2018; Mitchell, 2012).

Let us consider the unsupervised software synthesis programmer "SynthBot" (Yee-King and Roth, 2008), which uses a genetic algorithm to search for a target sound. The search is guided by measuring the similarity of the current candidate and the target, using the sum of squared errors between their MFCCs. The system was evaluated "technically to establish its ability to effectively search the space of possible parameter settings". Then, musicians competed with SynthBot to see who was the most competent sound synthesizer programmer. The sounds proposed by SynthBot and the musicians were compared with the target by using sound similarity measures.

In (Yee-King et al., 2018), a hill climber, a genetic algorithm and three deep neural networks are used for sound matching. The results are evaluated by calculating the error score associated with the euclidean distance between the MFCCs of the proposed sound and the MFCCs of the target.

In PresetGen (Tatar et al., 2016), the authors evaluated the system using computational and perceptual sound similarities.

(Ariza, 2009) proposes two general categories encompassing the motivations of generative musical systems: "Creative tools" and "Compositional models". Furthermore, Ariza suggests that, besides its design motivations, as proposed by (Pearce et al., 2002), the systems have to be evaluated in the context of use-case application.

### 4.3.2 RuLer evaluation

The evaluation of the rule model includes:

1. Listening surveys, where composers used the system with the same dataset and commented on the new created instances (Section 4.3.3).

2. Performances where the project has been presented, recordings performed using the algorithms and lists where the compositions made with the algorithm (Section 4.3.4).

3. Resampling tests (minority-oversampling), comparing the results with those obtained by state-of-the-art algorithms using extrinsic-benchmarks (Chapter 5).

In this way, both subjective and "numerical" evaluations are considered. Resampling algorithms were selected as they perform a process "similar" to the objectives pursued by the RuLer algorithm. Namely, they receive labeled input data and create new data classified by the model as belonging to the categories of the data used to create them. Although **the new data being classified as expected by a classification algorithm does not imply that it will be perceived by a composer as consistent with the classes**, contrasting this evaluation with the listening surveys offers a better overview of the algorithm's possibilities and limitations. The RuLer algorithm was designed to create on-the-fly variations out of an initial set of labeled examples. This task differs from sound matching, as the expected sound does not seek to replicate the examples but to produce different degrees of variation. Therefore, error scores are not suitable to evaluate the algorithm, as spectral differences are expected. Human vs machine tests (such as the John Henry Test) are difficult to apply as there are no clear criteria to select the winner. Survey tests usually compare computer generated vs human generated outputs (Tatar et al., 2016; Yee-King and Roth, 2008), or the computer generated outputs are rated by humans.

Under these considerations, to evaluate the RuLer, we perform user experience tests as suggested by (Ariza, 2009). As mentioned, since the task performed for the algorithm is similar to re-sampling processes, where new instances consistent with a class are produced from labeled data, re-sampling tests (accuracy scores) were also performed. This evaluation needs to be framed within the limits discussed by (Holland, 2000), who points out that music composition, as other open-ended domains, are "problem seeking" rather than "problem solving". From this perspective, there are no criteria to test correct answers. Therefore, as (Ariza, 2009) concludes, when used as creative systems, generative music systems can be best evaluated by studies of user interaction and experience, as well as through analysis and presentation. Also, limitations discussed in (Sturm et al., 2019) about evaluating machine learning tools for music generation using quantitative measures apply to the accuracies obtained. The discussed system-use case is within the problem-seeking domain, and the presented evaluation intends to contribute to the analysis of its possibilities rather than providing a quantitative validation.

### 4.3.3 Listening surveys (system-use case): creating new combinations

As a system-use case, we extend a set of existent presets, shown in Table 4.6, taken from the music work *Tiempos de Aguacero*, available at (Paz, 2017a). The piece is composed of four parts: intro, main, break and end. To build the data set, each preset was labeled with the name of the part in which it is used. The objective of the experiment is to extract a set of new presets intended to create variations within the respective parts. The presets control the parameters of four synthesizers named x3, x4, x5 and x6. The architecture of the synthesizer x3 = x5 and synthesizer x4 = x6.

#### Synthesizers

Synthesizers x3 and x5 perform additive synthesis of sawtooth waves. There are four wave generators with frequencies, freq1, freq2, freq1 - 1 and freq2 - 1, all in Hz. The output is controlled by a general normalized amplitude. The parameter ranges are

Table 4.6: Presets, taken from the music work *Tiempos de Aguacero*, available at (Paz, 2017a). The piece is composed of four parts (intro, main, break and end).

| preset | x3_freq1 | x3_freq2 | x3_amp | x4_freq1 | x4_width1 | x4_freq2 | x4_width2 | x4_amp | x5_freq1 | x5_freq2 | x5_amp | x6_freq1 | x6_with1 | x6_freq2 | x6_width2 | x6_amp | part |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | {200} | {159} | {0.2 } | {161} | {0.4} | {160} | {0.5} | {0.27} | {200} | {159} | {0.01} | {150} | {0.3} | {161} | {0.5} | {0.001} | 'intro' |
| 2 | {200} | {150} | {0.23} | {150} | {0.3} | {160} | {0.5} | {0.3 } | {200} | {150} | {0.01} | {150} | {0.3} | {160} | {0.5} | {0.01} | 'intro' |
| 3 | {200} | {150} | {0.26} | {150} | {0.3} | {160} | {0.4} | {0.36} | {200} | {150} | {0.001} | {150} | {0.3} | {160} | {0.4} | {0.01} | 'intro' |
| 4 | {200} | {159} | {0.2 } | {150} | {0.3} | {161} | {0.5} | {0.3 } | {200} | {159} | {0.01} | {150} | {0.3} | {161} | {0.5} | {0.001} | 'intro' |
| 5 | {20} | {150} | {0.25} | {101} | {0.3} | {102} | {0.5} | {0.33} | {20 } | {150} | {0.01} | {101} | {0.3} | {102} | {0.5} | {0.01} | 'main' |
| 6 | {200} | {150} | {0.28} | {150} | {0.6} | {160} | {0.4} | {0.38} | {200} | {150} | {0.01} | {150} | {0.6} | {160} | {0.4} | {0.01} | 'main' |
| 7 | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.33} | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.33} | 'main' |
| 8 | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.28} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'break' |
| 9 | {440} | {880} | {0.2 } | {660} | {0.3} | {661} | {0.5} | {0.25} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'break' |
| 10 | {440} | {660} | {0.2 } | {441} | {0.3} | {661} | {0.4} | {0.22} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 11 | {220} | {440} | {0.2 } | {660} | {0.3} | {221} | {0.5} | {0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 12 | {110} | {240} | {0.2 } | {330} | {0.3} | {221} | {0.5} | {0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 13 | {440} | {660} | {0.2 } | {165} | {0.3} | {221} | {0.5} | {0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |

Table 4.7: Parameter names and ranges for synthesizers x3 and x5.

| **Parameter name** | **Range** |
|---|---|
| frequency 1 (x3_freq1, x5_freq1) | 0 - 16,000 Hz |
| frequency 2 (x3_freq2, x5_freq2) | 0 - 16,000 Hz |
| amplitude (x3_amp, x5_amp) | 0 - 1 |

Table 4.8: Parameter names and ranges for synthesizers x4 and x6.

| **Parameter name** | **Range** |
|---|---|
| frequency 1 (x4_freq1, x6_freq1) | 0 - 16,000 Hz |
| pulse ratio 1 (x4_width1, x6_width1) | 0 - 1 |
| frequency 2 (x4_freq2, x6_freq2) | 0 - 16,000 Hz |
| pulse ratio 2 (x4_width2, x6_width2) | 0 - 1 |
| amplitude (x4_amp, x6_amp) | 0 - 1 |

summarized in Table 4.7.

Synthesizers x4 and x6 perform additive synthesis using band limited pulse waves. There are two pulse generators controlled by parameters frequency 1 and 2 (in Hz) and width pulse ratios[3] 1 and 2, varying from 0 to 1 (where 0.5 makes a square wave). The two pulse generators share a common normalized amplitude. Table 4.8 shows the parameter names and their valid ranges.

### Rule extraction and user survey

For the user tests, three live coders were asked to use the system in real time. First, the presets of Table 4.6 were introduced, so the live coders could become familiar with the material. Then, each of them used the algorithm to extend the original set while going through the different parts. Table 4.9 shows different configurations of the algorithm and the percentages of new combinations evaluated as satisfactory by the composers

[3]The ratio between the pulse duration and the period.

Table 4.9: Configurations of the algorithm parameters ($d$, $ratio$) and percentage of new combinations successfully evaluated for each composer.

| **Parameters** $d$**,** $ratio$**:** | 1,1 | 3, 1/8 | 6, 1/8 | 9, 1/16 | 12, 1/16 |
|---|---|---|---|---|---|
| Composer 1 | - | 100% | 100% | 90% | 80% |
| Composer 2 | - | 100% | 100% | 90% | 85% |
| Composer 3 | - | 95 % | 90% | 80% | 70% |

for different dissimilarities, $d$'s.

To visualize the algorithm performance, rules for different combinations of $d$ and $ratio$ are shown in Table 4.10. Each set is discussed in turn in the following section.

### Extracted rules

Table 4.10 shows three sets of extracted rules for parameter combinations: $d = 6$, $ratio = 1/8$; $d = 12$, $ratio = 1/16$ and $d = 15$, $ratio = 0$. The order in which the rules appear is the order of the algorithm's output. Table 4.11 shows the number of new rules (new single instances) created for each pair $(d, ratio)$ of Table 4.10.

It is difficult to grasp, beyond the limit cases, how the output of an algorithm depends on the input data. However, this is an illustrative example, as it shows contrasting behavior between parts *intro* and *break* with parts *main* and *end*.

The number of individual instances contained in the rules (termed cardinality) describing parts 'intro' and 'break' increases gradually as $d$ increases and $ratio$ decreases. For 'intro', setting $d = 6$ and $ratio = 1/8$, 14 new combinations are created. With $d = 12$ and $ratio = 1/16$, this number reaches 44. Finally, if $d = 15$ and $ratio = 0$, 9208 possible combinations can be formed with the input data! This is due to, in the original data, the individual rules having many common values.

For the 'end' category, at the top of Table 4.10, rules 9 to 12 describe parameter combinations with $d = 6$ and $ratio = 1/6$. These rules were generated from presets 10 to 13 of Table 4.6. Calculating its cardinality, the total number of combinations is 40,

Table 4.10: Extracted rules for different configurations of the algorithm.

| preset | x3_freq1 | x3_freq2 | x3_amp | x4_freq1 | x4_width1 | x4_freq2 | x4_width2 | x4_amp | x5_freq1 | x5_freq2 | x5_amp | x6_freq1 | x6_with1 | x6_freq2 | x6_width2 | x6_amp | part |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Rules extracted with d = 6, ratio = 1/8.** | | | | | | | | | | | | | | | | | |
| 1 | {200} | {150} | {0.23} | {150} | {0.3} | {160} | {0.5} | {0.3} | {200} | {150} | {0.01} | {150} | {0.3} | {160} | {0.5} | {0.01} | 'intro' |
| 2 | {200} | {150} | {0.26} | {150} | {0.3} | {160} | {0.4} | {0.36} | {200} | {150} | {0.001} | {150} | {0.3} | {160} | {0.4} | {0.01} | 'intro' |
| 3 | {200} | {159} | {0.2} | {161, 150} | {0.4, 0.3} | {160, 161} | {0.5} | {0.27, 0.3} | {200} | {159} | {0.01} | {150} | {0.3} | {161} | {0.5} | {0.001} | 'intro' |
| 4 | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.28} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'break' |
| 5 | {440} | {880} | {0.2} | {660} | {0.3} | {661} | {0.5} | {0.25} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'break' |
| 6 | {20} | {150} | {0.25} | {101} | {0.3} | {102} | {0.5} | {0.33} | {20} | {150} | {0.01} | {101} | {0.3} | {102} | {0.5} | {0.01} | 'main' |
| 7 | {200} | {150} | {0.28} | {150} | {0.6} | {160} | {0.4} | {0.38} | {200} | {150} | {0.01} | {150} | {0.6} | {160} | {0.4} | {0.01} | 'main' |
| 8 | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.33} | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.33} | 'main' |
| 9 | {440} | {660} | {0.2} | {441, 165} | {0.3} | {221, 661} | {0.4, 0.5} | {0.22, 0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 10 | {220, 110} | {440, 240} | {0.2} | {330, 660} | {0.3} | {221} | {0.5} | {0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 11 | {440, 220} | {440, 660} | {0.2} | {660, 165} | {0.3} | {221} | {0.5} | {0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 12 | {440, 110} | {240, 660} | {0.2} | {330, 165} | {0.3} | {221} | {0.5} | {0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| **Rules extracted with d = 12, ratio = 1/16.** | | | | | | | | | | | | | | | | | |
| 1 | {20} | {150} | {0.25} | {101} | {0.3} | {102} | {0.5} | {0.33} | {20} | {150} | {0.01} | {101} | {0.3} | {102} | {0.5} | {0.01} | 'main' |
| 2 | {200} | {150} | {0.28} | {150} | {0.6} | {160} | {0.4} | {0.38} | {200} | {150} | {0.01} | {150} | {0.6} | {160} | {0.4} | {0.01} | 'main' |
| 3 | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.33} | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.33} | 'main' |
| 4 | {100} | {100} | {0.25} | {100} | {0.6} | {102} | {0.4} | {0.28} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'break' |
| 5 | {440} | {880} | {0.2} | {660} | {0.3} | {661} | {0.5} | {0.25} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'break' |
| 6 | {440} | {660} | {0.2} | {441, 165} | {0.3} | {221, 661} | {0.4, 0.5} | {0.22, 0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 7 | {440, 220, 110} | {440, 240, 660} | {0.2} | {330, 660, 165} | {0.3} | {221} | {0.5} | {0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 8 | {200} | {159} | {0.2} | {161, 150} | {0.4, 0.3} | {160, 161} | {0.5} | {0.27, 0.3} | {200} | {159} | {0.01} | {150} | {0.3} | {161} | {0.5} | {0.001} | 'intro' |
| 9 | {200} | {150} | {0.23, 0.26} | {150} | {0.3} | {160} | {0.5, 0.4} | {0.3, 0.36} | {200} | {150} | {0.01, 0.001} | {150} | {0.3} | {160} | {0.5, 0.4} | {0.01} | 'intro' |
| **Rules extracted with d = 15, ratio = 0.** | | | | | | | | | | | | | | | | | |
| 1 | {440, 220, 110} | {440, 240, 660} | {0.2} | {441, 330, 660, 165} | {0.3} | {221, 661} | {0.4, 0.5} | {0.22, 0.23} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'end' |
| 2 | {200, 100, 20} | {100, 150} | {0.25, 0.28} | {100, 101, 150} | {0.3, 0.6} | {160, 102} | {0.5, 0.4} | {0.33, 0.38} | {200, 100, 20} | {100, 150} | {0.25, 0.01} | {100, 101, 150} | {0.3, 0.6} | {160, 102} | {0.5, 0.4} | {0.33, 0.01} | 'main' |
| 3 | {200} | {150, 159} | {0.2, 0.26, 0.23} | {161, 150} | {0.4, 0.3} | {160, 161} | {0.5, 0.4} | {0.27, 0.36, 0.3} | {200} | {150, 159} | {0.01, 0.001} | {150} | {0.3} | {160, 161} | {0.5, 0.4} | {0.01, 0.001} | 'intro' |
| 4 | {440, 100} | {880, 100} | {0.25, 0.2} | {100, 660} | {0.6, 0.3} | {661, 102} | {0.4, 0.5} | {0.28, 0.25} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | {0} | 'break' |

Table 4.11: Number of new combinations created during the rule extraction process.

| **Part** | d = 6, ratio = 1/8 | d = 12, ratio = 1/16 | d = 15, ratio = 0 |
|---|---|---|---|
| intro | 14 | 44 | 9208 |
| main | 0 | 0 | 331773 |
| break | 0 | 0 | 254 |
| end | 36 | 39 | 284 |

of which 36 are unique. The set of rules extracted when $d = 12$ and $ratio = 1/16$ (rules 6 and 7 at the middle of Table 4.10) produces 39 unique combinations. Finally, when $d = 15$ and $ratio = 0$ (rule 1 at the bottom of Table 4.10), there are 284 new settings.

Analyzing the extracted rules for classes 'main' and 'break' as $d$ increases in Table 4.10, it can be seen that no new rules are created either for $d = 6$ or 12. This is because the differences between the original presets are bigger than the allowed dissimilarities. Then, when $d = 15$ and $ratio = 0$, we obtain rules 2 and 4 of Table 4.10, which give all the possible combinations between the values of the parameters.

This example shows how the input data, in combination with the algorithm parameters, can produce extreme results.

Audio examples of the 'main' part, with and without RuLer, can be found in (Paz, 2018). In (Paz, 2019a), from minute 9:00 to 19:43, the system with presets of Table 4.6 was used in real-time during the First Live Coding Music seminar held at the Institute for Pure and Applied Mathematics of Rio de Janeiro (where I was invited to give a Keynote and perform). In both examples, the algorithm adds variability to the original material. In the second, the algorithm allows one to extend the piece from its original duration, 03:42 (Paz, 2017a), to 10:34 minutes.

### 4.3.4 Live Performances and Recordings

A series of live coding performances and recordings have accompanied the design and testing of the algorithm. These have been developed in different contexts and venues including universities, artistic research centers, theatres, online streaming, smoky bars, etc.

They allow for the evaluation of:

1. The algorithm's affordances and capacity to produce "interesting variations" over the input data during the performance.

2. How the community receives the music generated using the algorithms.

A more extensive list of performances, recordings, mentions and lists where music composed with the algorithms or ideas developed in this thesis can be found in Section Related Publications.

The first ideas of the project, explored in *Multiparametric representation space as a perceptual exploration interface* (Paz, 2016), were presented at the end of May 2016 with a performance at the Artistic Research Center of Barcelona **Hangar**.

The EP studio album **Visions of Space** (Paz, Iván, 2017a), featured by the Berliner record label *Bohemian drips*, applied IF-THEN rules to generate sections of tracks 4 (*Visions of Space*) and 5 (*En Casa*). It was recorded at *Zuhause* and released in a concert at the legendary Schrippe Hawaii (Neukölln, Berlin, Paz, Iván, 2017b).

During the third International Conference on Live Coding, held at the *Centro Mexicano de la Música y las Artes Sonoras* (Morelia, Mexico 2017), the performance "Live Coding through Perceptual Spots on the Parametric Space" was included in the opening concert. For the performance, I used IF-THEN rules to generate in real-time the material of different sections of the piece.

In 2018, **Bandcamp Daily** featured the album **Visions of the Space** together with nine other albums released in 2017, under the list *Meet the Artists Using Coding, AI, and Machine Language to Make Music* (Chandler, 2018).

The live performance presented during the **live coding => music;** seminar (Paz, 2019c), held at the Instituto Nacional de Matemática Pura e Aplicada (National Institute for Pure and Applied Mathematics) of Rio de Janeiro, is presented in (Paz, 2019a).

A multichannel (34 speakers) version of (Paz, 2017b) was presented at the Electroacustic Musical Festival Zeppelin of Barcelona during October (Zeppelin, 2019).

Visions of Space tracks 2 (*Tiempos de Aguacero*) and 5 (*En Casa*) were included in the track-list of the a-Musik Radio (Köln) and DubLab Barcelona broadcasts (Brauneis, 2019; Cassamajó, 2019).

The online performance presented during the **EulerRoom Equinox 2020**, which featured 72 hours of live coding performances around the world (20–22 March), is available in (Paz, 2020).

A performance using rules extracted with the RuLer algorithm was presented as part of *8:08 La Hora del Live Coder* a set of performances in which, during the Barcelona lockdown of March 2020, every night one member of the TOPLAP-Barcelona collective performed online live at 8:08pm (Paz, April 3, 2020). A similar approach was used, using Sema (described in Section 2.4.6), during the Network Music Festival in July 2020 (Paz, 16th July, 2020).

Although a subjective appreciation, the algorithm has shown effective capacities to produce new interesting material on-the-fly. The current version allows for the preloading of data before the performance and/or the saving of new instances as they are found. If all the instances are captured in real time, the space exploration process becomes part of the performance. The current implementation does not overwrite the input data with the extracted model, so the performer can extract different sets using different combinations of $d$ and $ratio$ while conducting the piece.

## 4.4 Conclusion

Symbolic rule learning offers an interesting approach for real-time sound synthesizer programming, specifically, the task of producing different degrees of variation from an input material. This idea is built on the observation that, as in other improvisation practices, live coders either have some preselected material or select combinations mid-performance. These are the “sweet spots” of the synthesis algorithm from which the performance unfolds. Each selection is a combination of parameter values in the synthesis parameter space. Finally, to create labeled data set, in order to use inductive learning, a linguistic category or class is assigned to each combination.

The presented algorithm searches for patterns among the examples of the same class using a dissimilarity function and obtains new unheard combinations based on the patterns found. In the current implementation, the new examples are created by recombining the material of those combinations that exhibit a pattern. The algorithm parameters control the degree of dissimilarity among the combinations with the same class that can be use for the induction process and the amount of new settings that can be produced. The proposed algorithm creates human-readable models which are independent of the order of the input data. It also allows working with different data types (numeric, categorical, etc.) and small data sets.

The algorithm can be included within bigger systems allowing, for example, addition or elimination of new instances on-the fly.

The parameters of the algorithm allow one to control the induction process and intuit, from a musical perspective, the novelty or consistency with the input data of the new generated combinations. The evaluation presented in this chapter is intended to provide a perceptual evaluation.

The user surveys point out that the algorithm can be effectively used in real time to extend an input data set. As it operates right now, once the input data set is given, the possible rules for the ($d$, $ratio$) pairs are determined. The user only explores the system in real time, playing with the material as can be seen in (Paz, 2019a). Note, however, that the possibilities depend on the perceptual topology of the synthesis algorithm and the context (i.e FM synthesis is not additive). These limitations were already pointed out by (Yee-King, 2011b; Dahlstedt, 2009).

The user-surveys should be taken carefully. They intend to provide an overview of the system capacities in real world examples, rather than to provide a conclusive evaluation of the system. Remember that music's creative pursuit is more a problem-seeking than problem-solving activity (Holland, 2000).

The venues to which the music composed has been invited, including four physical geographies, suggest a good reception of the results by the community. In his review, (Chandler, 2018) writes: “The album’s droning yet often harsh electronic soundscapes were put together using musical algorithms whose parameters Paz varies sequentially through time, in much the same way that the parameters controlling an artificial intelligence are altered by the process of learning. Yes, this is all too abstract to express sufficiently in a single paragraph, but the unnerving, sinister power of the dystopian title track alone”(the track *Tiempos de Aguacero*)“is enough to prove it’s an effective method.”

# Chapter 5

# Oversampling Tests

This chapter evaluates the extracted rule model by comparing its capacity to generate synthetic instances with that of other oversampling algorithms. Thus, continuing the evaluation presented in Section 4.3 and putting both evaluations together, we have subjective and numerical assessments. By its nature, the RuLer algorithm can deal with the problems derived from having unbalanced data in classification and regression problems, generating new values just as the oversampling algorithms that can be found in the literature do. That is, taking a set of labeled data, analyzing the instances of a class and creating new ones that could be cataloged as (variations) belonging to such class.

## 5.1 Oversampling algorithms

The RuLer algorithm identifies patterns based on its dissimilarity function. When two examples exhibit a pattern and the restriction imposed by the *ratio* is met, it creates a rule. The rule contains the examples and can also recombine or extend its material. In the latter case, the resulting rule contains new inducted combinations of values, intended to be consistent (up to certain degree) with the category of the rule. This process is similar to that performed by oversampling algorithms such as

Smote: Synthetic Minority Over-sampling Technique (Chawla et al., 2002) or Adasyn: addaptive synthetic sampling (He et al., 2008), which produce new instances in a data set expected to be consistent with a specific class.

Nonetheless, there are other oversampling algorithms (Amin et al., 2016). Here we only deal with the referred two as they are the most used as well as the basis of other modern algorithms (either to improve them or to compensate for their flaws, see for example, Zheng et al., 2016; Basgall et al., 2018; Wei et al., 2020).

### 5.1.1 Smote

Smote is an oversampling algorithm in which, unlike simple approaches that randomly select and duplicate examples from the minority class (with or without replacement), "synthetic" examples are produced by applying certain operations to real data[1]. To oversample a dataset, the Smote algorithm takes an example and considers its k nearest neighbors in the feature space. Then, to create the synthetic points, it takes the vector between one of the neighbors and the current point and multiplies it by a random number between 0 and 1. Generating synthetic points in this way forces the decision region of the minority class to become more general.

Finally, depending on the amount of over-sampling required, neighbors from the k nearest neighbors are randomly chosen. For example, if 200% of oversampling is required, and k=5, only two neighbors from the five nearest neighbors are chosen, and one sample is generated in each direction.

### 5.1.2 Adasyn

Adasyn is a modified version of Smote that uses a density distribution for different minority class examples, according to their level of difficulty in learning, and generates more synthetic data for the minority class examples that are harder to learn. This is

[1]The idea was inspired by oversampling techniques for image recognition, in which operations such as rotation, reflection, etc, are used to extend the dataset.

done in the following way: For each example $\boldsymbol{x}_i$, the number of examples that need to be generated is defined as:

$$g_i = \hat{r}_i \text{ x } G \tag{5.1}$$

Where $G$ is the number of synthetic examples that need to be generated according to the desired balance degree[2], and $\hat{r}_i$ is a density distribution built in the following way:

$$\hat{r_i} = r_i / \sum_{i=1}^{m_s} r_i \tag{5.2}$$

Where $m_s$ is the number of samples of the minority class, and $r_i = \Delta_i / k$, $k$ being the selected number of nearest neighbors, and $\Delta_i$ is the number of minority class examples in the $k$ nearest neighbors of $\boldsymbol{x}_i$.

Once $g_i$ is calculated, then, for $i = 1$ to $g_i$, the algorithm creates a synthetic sample in the following way:

$$\boldsymbol{s}_i = \boldsymbol{x}_i + (\boldsymbol{x}_{zi} - \boldsymbol{x}_i) * \lambda \tag{5.3}$$

Where $\boldsymbol{x}_{zi}$ is a randomly chosen point from the minority class out of the $k$ nearest neighbors of $\boldsymbol{x}_i$; $(\boldsymbol{x}_{zi} - \boldsymbol{x}_i)$ is the vector difference; and $\lambda \in [0, 1]$ is a random number.

## 5.2 Datasets and experiments

Oversampling tests were performed using the benchmarks available at the KEEL-dataset repository for imbalance datasets (Keel, 2010). The selected datasets were: glass-0-1-2-3_vs_4-5-6, page-blocks0, ecoli3, yeast3, segment0, ecoli2, yeast-2_vs_4, glass0, glass1, vehicle3, and new-thyroid1. The characteristics of each dataset (number of attributes, instances, and imbalance ratio) are shown in Table 5.1. They have two classes

[2]If $m_s, m_l$ are the minority and majority class examples, respectively. Then, $G = (m_s - m_l) * \beta$, where $\beta \in [0, 1]$ determines the balance degree. $\beta = 1$ is a full balance.

positive and negative, positive being the minority. The number of instances they have ranges between 214 and 5472, and the imbalance ratio is between 1.82 and 9.08.

For the tests, the Smote, Adasyn and RuLer algorithms were used to oversampling the minority class, introducing for each set the number of synthetic instances needed to balanced the classes (50% positive - 50% negative). Then, classification algorithms were run over each dataset. A baseline dataset (original data without oversampling) was also used for comparison. The classification algorithms used were K-Nearest Neighbours, Randomforest and a Support Vector classifier with a linear kernel.

The K-Nearest Neighbours does not require a training period (these types of algorithms are known as instance based learners). It stores the training data and learns from it (analyzing the data) as it performs real-time predictions. While this has some disadvantages (it is sensitive to outliers, for example), it also makes the algorithm much faster than those that require training, such as SVM. By assigning the classes only by looking at the neighbors, new data can be added with little impact to its accuracy. These characteristics make KNN very easy to implement and to interpret (only two parameters are required: the value of K and the distance function).

The Random forest classifier was selected, as decision trees are the natural alternative to rule-based classifiers as discussed in Section 3.1.

Support Vector Classifiers (SVCs) create high accuracy models and have great generalization capacities, as they allow regularization that prevents over-fitting. They can also handle non-linear data using the kernel trick. Moreover, these algorithms build stable hyperplane models that are not affected by small changes in the data. However, SVCs require long training time and produce models that are difficult to interpret by humans. The support vectors are stored in the memory (and its number grows quickly with the training dataset size), for which SVCs have high algorithmic complexity and memory requirements. They can also be complex to operate, as selecting an appropriate kernel function (for non-linear data) is not a simple task.

## 5.3 Results

The results of the average precision scores[3] for the KNN, Randomforest and SVC algorithms applied to the datasets are shown in Table 5.1. The scores were calculated using 70% - 30% of the data for training and testing, respectively.

Table 5.1: Average precision scores for the different datasets. The algorithms used for oversampling the data were Smote, Adasyn and RuLer. The classifiers used were KNN, Randomforest and Support Vector Classifier SCVs with linear kernel. IR stands for imbalanced ratio.

| **Dataset: glass-0-1-2-3_vs_4-5-6 attributes:9 instances:214 IR:3.2** | | | |
|---|---|---|---|
| Algorithm | KNN k = 2 | randomforest trees = 10 | SVC linear kernel |
| Baseline | 0.809 | 0.931 | 0.949 |
| Smote | 0.796 | 0.902 | 0.966 |
| Adasyn | 0.796 | 0.878 | 0.957 |
| RuLer | 0.842 | 0.947 | 0.944 |
| **Dataset: page-blocks0 attributes:10 instances:5472 IR:8.79** | | | |
| Algorithm | KNN k = 3 | randomforest trees = 10 | SVC linear kernel |
| Baseline | 0.744 | 0.908 | 0.727 |
| Smote | 0.711 | 0.900 | 0.687 |
| Adasyn | 0.685 | 0.886 | 0.657 |
| RuLer | 0.715 | 0.983 | 0.735 |
| **Dataset: ecoli3 attributes:7 instances:336 IR:8.6** | | | |
| Algorithm | KNN k = 4 | randomforest trees = 10 | SVC linear kernel |
| Baseline | 0.633 | 0.685 | 0.732 |
| Smote | 0.615 | 0.695 | 0.608 |
| Adasyn | 0.534 | 0.680 | 0.600 |

[3]The Average Precision Score (AP) summarizes the precision-recall curve as follows: $AP = \sum_n (R_n - R_{n-1}) * P_n$, where $P_n$ and $R_n$ are the precision and recall at the $n^{th}$ step.

| | | | |
|---|---|---|---|
| RuLer | 0.774 | 0.951 | 0.748 |

**Dataset: yeast3 attributes:8 instances:1484 IR:8.1**

| Algorithm | KNN k = 4 | randomforest trees = 10 | SVC linear kernel |
|---|---|---|---|
| Baseline | 0.709 | 0.751 | 0.816 |
| Smote | 0.606 | 0.719 | 0.802 |
| Adasyn | 0.561 | 0.703 | 0.793 |
| RuLer | 0.719 | 0.917 | 0.803 |

**Dataset: segment0 attributes:19 instances:2308 IR:6.02**

| Algorithm | KNN k = 4 | randomforest trees = 10 | SVC linear kernel |
|---|---|---|---|
| Baseline | 0.986 | 0.999 | 0.997 |
| Smote | 0.950 | 0.998 | 0.999 |
| Adasyn | 0.924 | 0.999 | 0.998 |
| RuLer | 1.0 | 0.999 | 0.999 |

**Dataset: ecoli2 attributes:7 instances:336 IR:5.46**

| Algorithm | KNN k = 4 | randomforest trees = 10 | SVC linear kernel |
|---|---|---|---|
| Baseline | 0.926 | 0.872 | 0.864 |
| Smote | 0.881 | 0.881 | 0.875 |
| Adasyn | 0.793 | 0.913 | 0.913 |
| RuLer | 0.906 | 0.963 | 0.868 |

**Dataset: yeast-2_vs_4 attributes:8 instances:514 IR:9.08**

| Algorithm | KNN k = 4 | randomforest trees = 10 | SVC linear kernel |
|---|---|---|---|
| Baseline | 0.697 | 0.782 | 0.796 |
| Smote | 0.677 | 0.785 | 0.813 |
| Adasyn | 0.698 | 0.813 | 0.802 |
| RuLer | 0.847 | 0.975 | 0.834 |

**Dataset: glass0 attributes:9 instances:214 IR:2.06**

| Algorithm | KNN k = 3 | randomforest trees = 10 | SVC linear kernel |
|---|---|---|---|
| Baseline | 0.768 | 0.841 | 0.596 |

|  |  |  |  |
|---|---|---|---|
| Smote | 0.802 | 0.864 | 0.730 |
| Adasyn | 0.788 | 0.850 | 0.681 |
| RuLer | 0.831 | 0.934 | 0.745 |
| **Dataset: glass1 attributes:9 instances:214 IR:1.82** |  |  |  |
| Algorithm | KNN k = 4 | randomforest trees = 10 | SVC linear kernel |
| Baseline | 0.827 | 0.824 | 0.443 |
| Smote | 0.853 | 0.852 | 0.495 |
| Adasyn | 0.805 | 0.846 | 0.442 |
| RuLer | 0.875 | 0.938 | 0.604 |
| **Dataset: vehicle3 attributes:18 instances:846 IR:2.99** |  |  |  |
| Algorithm | KNN k = 2 | randomforest trees = 10 | SVC linear kernel |
| Baseline | 0.415 | 0.655 | 0.689 |
| Smote | 0.401 | 0.639 | 0.680 |
| Adasyn | 0.372 | 0.612 | 0.681 |
| RuLer | 0.603 | 0.942 | 0.720 |
| **Dataset: new-thyroid1 attributes:5 instances:215 IR:5.14** |  |  |  |
| Algorithm | KNN k = 2 | randomforest trees = 10 | SVC linear kernel |
| Baseline | 1.0 | 0.994 | 1.0 |
| Smote | 1.0 | 0.997 | 1.0 |
| Adasyn | 1.0 | 1.0 | 1.0 |
| RuLer | 1.0 | 1.0 | 1.0 |

Table 5.1 shows that the average precision scores obtained by the classification algorithms using data oversampled with the RuLer are comparable and in most cases even better than those oversampled with Smote and Adasyn. A visual representation of Table 5.1 is shown in Figure 5.1.

To assess whether there are statistically significant differences between the distributions of the paired groups, a Friedman test was used. As we have more than two groups, the Friedman test identifies if there is a significant difference between groups,

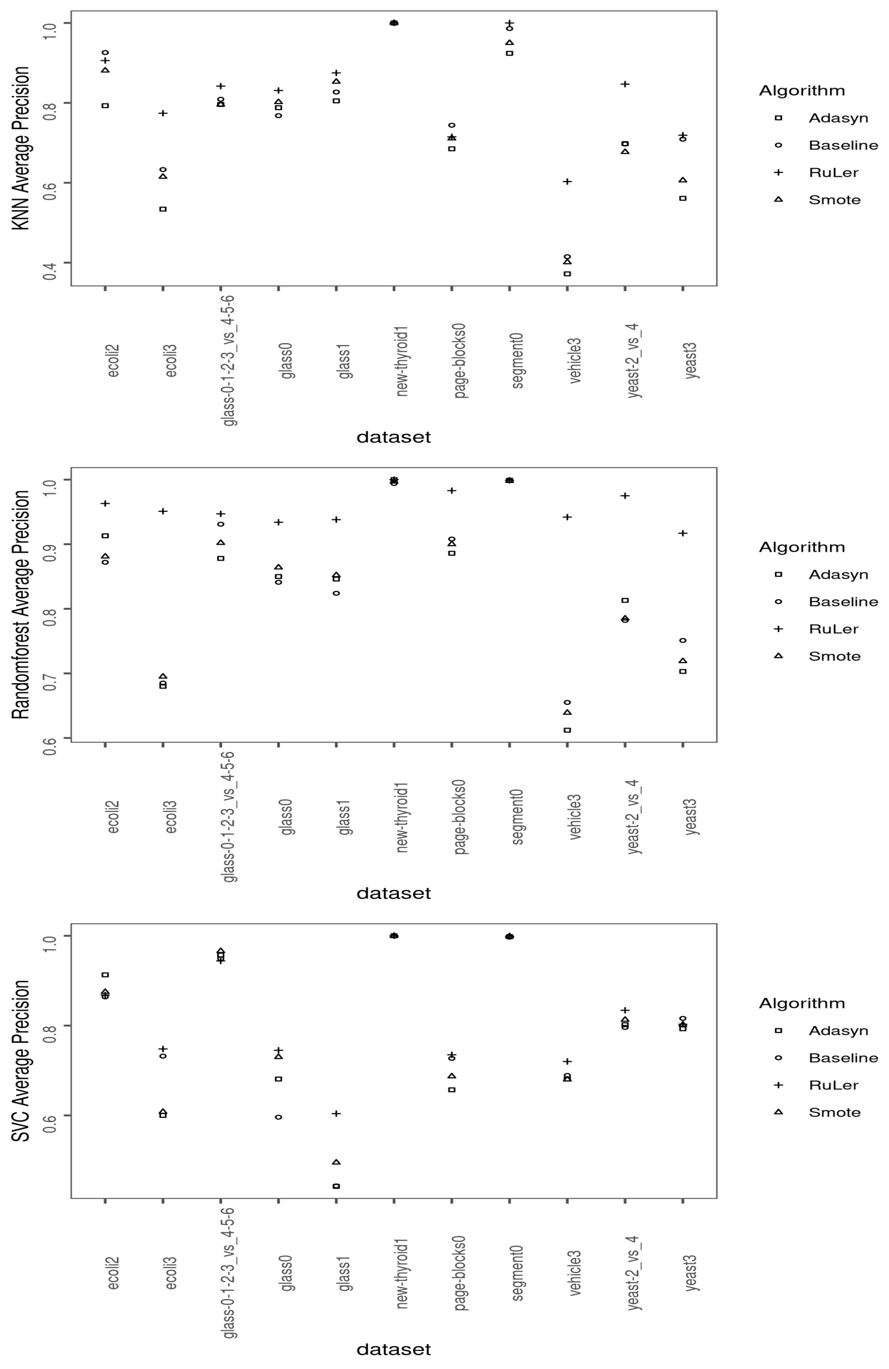

Figure 5.1: Average precision scores by classifier (KNN, Randomforest and SVC) for Baseline, Smote, Adasyn and RuLer algorithms, for each dataset on Table 5.1.

but we do not know which pairs are different. Therefore, pairwise comparisons were performed using Wilcox and Sign tests (comparing their two results). The Friedman tests and the pairwise comparisons using Wilcox and Sign tests are shown for each classifier (Table 5.2). For the KNN classifier there are significant differences between the RuLer algorithm and both Adasyn and Smote. In the case of the RandomForest, there are significant differences among the RuLer algorithm with Smotote, Adasyn and the Baseline. In the case of the SVC, there are not significant differences among the groups.

Table 5.2: Friedman test and pairwise comparisons using Wilcox and Sign tests for classifiers KNN, Randomforest and SVC. In the Friedman test, n is the sample size, df are the degrees of freedom and p is the significance. For the Wilcox and Sign tests, n1 and n2 are the sample sizes of the respective groups, df are the degrees of freedom, p is the significance and p.adjust are the adjusted P-values for Multiple Comparisons. Finally, psdj.signif returns ns if there are no significant differences among groups, or it returns either * or ** if p > 0.05 or p > 0.01, respectively, as the convention stands.

| **KNN Friedman test** | | | | |
|---|---|---|---|---|
| n | statistic | df | p | method |
| 11 | 20.0 | 3 | 0.000167 | Friedman test |

| **Pairwise comparisons Wilcox test** | | | | | | | |
|---|---|---|---|---|---|---|---|
| group1 | group2 | n1 | n2 | statistic | p | p.adj | p.adj.signif |
| Adasyn | Baseline | 11 | 11 | 4 | 0.019 | 0.115 | ns |
| Adasyn | RuLer | 11 | 11 | 0 | 0.006 | 0.036 | * |
| Adasyn | Smote | 11 | 11 | 2 | 0.018 | 0.106 | ns |
| Baseline | RuLer | 11 | 11 | 7 | 0.042 | 0.249 | ns |
| Baseline | Smote | 11 | 11 | 43 | 0.126 | 0.756 | ns |
| RuLer | Smote | 11 | 11 | 55 | 0.006 | 0.036 | * |

**Pairwise comparisons using Sign test**

| group1 | group2 | n1 | n2 | statistic | df | p | p.adj | p.adj.signif |
|---|---|---|---|---|---|---|---|---|
| Adasyn | Baseline | 11 | 11 | 2 | 10 | 0.109 | 0.654 | ns |
| Adasyn | RuLer | 11 | 11 | 0 | 10 | 0.002 | 0.012 | * |
| Adasyn | Smote | 11 | 11 | 1 | 9 | 0.039 | 0.235 | ns |
| Baseline | RuLer | 11 | 11 | 2 | 10 | 0.109 | 0.654 | ns |
| Baseline | Smote | 11 | 11 | 8 | 10 | 0.109 | 0.654 | ns |
| RuLer | Smote | 11 | 11 | 10 | 10 | 0.002 | 0.012 | * |

**Randomforest Friedman test**

| n | statistic | df | p | method |
|---|---|---|---|---|
| 11 | 17.3 | 3 | 0.000601 | Friedman test |

**Pairwise comparisons Wilcox text**

| group1 | group2 | n1 | n2 | statistic | p | p.adj | p.adj.signif |
|---|---|---|---|---|---|---|---|
| Adasyn | Baseline | 11 | 11 | 22.5 | 0.646 | 1 | ns |
| Adasyn | RuLer | 11 | 11 | 0 | 0.009 | 0.055 | ns |
| Adasyn | Smote | 11 | 11 | 24 | 0.45 | 1 | ns |
| Baseline | RuLer | 11 | 11 | 0 | 0.006 | 0.036 | * |
| Baseline | Smote | 11 | 11 | 33 | 1 | 1 | ns |
| RuLer | Smote | 11 | 11 | 66 | 0.000977 | 0.006 | ** |

**Pairwise comparisons using Sign test**

| group1 | group2 | n1 | n2 | statistic | df | p | p.adj | p.adj.signif |
|---|---|---|---|---|---|---|---|---|
| Adasyn | Baseline | 11 | 11 | 5 | 10 | 1.00e+0 | 1 | ns |
| Adasyn | RuLer | 11 | 11 | 0 | 9 | 4.00e-3 | 0.023 | * |
| Adasyn | Smote | 11 | 11 | 4 | 11 | 5.49e-1 | 1 | ns |
| Baseline | RuLer | 11 | 11 | 0 | 10 | 2.00e-3 | 0.012 | * |
| Baseline | Smote | 11 | 11 | 5 | 11 | 1.00e+0 | 1 | ns |
| RuLer | Smote | 11 | 11 | 11 | 11 | 9.77e-4 | 0.006 | ** |

| **SVC Friedman test** | | | | |
|---|---|---|---|---|
| n | statistic | df | p | method |
| 11 | 7.67 | 3 | 0.0534 | Friedman test |

| **Pairwise comparisons using Wilcox test** | | | | | | | |
|---|---|---|---|---|---|---|---|
| group1 | group2 | n1 | n2 | statistic | p | p.adj | p.adj.signif |
| Adasyn | Baseline | 11 | 11 | 25.5 | 0.878 | 1 | ns |
| Adasyn | RuLer | 11 | 11 | 9 | 0.066 | 0.399 | ns |
| Adasyn | Smote | 11 | 11 | 9.5 | 0.074 | 0.445 | ns |
| Baseline | RuLer | 11 | 11 | 8 | 0.053 | 0.317 | ns |
| Baseline | Smote | 11 | 11 | 22 | 0.61 | 1 | ns |
| RuLer | Smote | 11 | 11 | 38 | 0.076 | 0.454 | ns |

| **Pairwise comparisons using Sign test** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| group1 | group2 | n1 | n2 | statistic | df | p | p.adj | p.adj.signif |
| Adasyn | Baseline | 11 | 11 | 5 | 10 | 1 | 1 | ns |
| Adasyn | RuLer | 11 | 11 | 2 | 10 | 0.109 | 0.654 | ns |
| Adasyn | Smote | 11 | 11 | 2 | 10 | 0.109 | 0.654 | ns |
| Baseline | RuLer | 11 | 11 | 2 | 10 | 0.109 | 0.654 | ns |
| Baseline | Smote | 11 | 11 | 4 | 10 | 0.754 | 1 | ns |
| RuLer | Smote | 11 | 11 | 7 | 9 | 0.18 | 1 | ns |

Figure 5.2 shows the average precision-score box-plots for classification algorithms (KNN, Randomforest and SVC) using Baseline, Smote, Adasyn and RuLer algorithms, considering the eleven datasets. The Friedman tests are shown for each classifier at the top of its corresponding graph. In Figure 5.2, the RuLer algorithm is the one with both the highest median for KNN and RandomForest classifiers and the most compact quartiles for all classification algorithms. The pairwise comparisons using Sign test with *p.adjust Bonferroni* appear marked with "*" or "**" when there are significant differences among groups, as in Table 5.2. For the Wilcox and Sign tests n1 and n2

are the sample sizes of the respective groups, df are the degrees of freedom, p is the significance, p.adjust is the adjust P-values for Multiple Comparisons and psdj.signif returns ns if there are no significant differences among groups or either * or **, if $p > 0.05$ or $p > 0.01$, respectively, as the convention stands.

The Friedman test yields statistically significant differences between the distributions for KNN and Randomforest algorithms. The summary statistics for each classifier is available in Table 5.3.

To understand why there are no significant differences in the case of the SVC, the means and standard deviations for its average precision are analyzed (Table 5.3). The difference between the lower (Adasyn, 0.775) and the higher (RuLer, 0.818) means is only 0.042. Also, the average precision has low *sd* (the max being 0.181 for Adasyn). This explains why SVC algorithms do not have significant differences among the over-sampling algorithms.

Table 5.3: Summary statistics for the average precision score of classification algorithms applied to Baseline data and data oversampled by Smote, Adasyn and RuLer. "n" is the sample size, "iqr" is the Interquartile Range, "sd" the standard deviation, "se" stands for the standard error and "ci" for the confidence interval.

| **KNN Average precision** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Algorithm | n | min | max | median | iqr | mean | sd | se | ci |
| Adasyn | 11 | 0.372 | 1 | 0.788 | 0.178 | 0.723 | 0.18 | 0.054 | 0.121 |
| Baseline | 11 | 0.415 | 1 | 0.768 | 0.174 | 0.774 | 0.168 | 0.051 | 0.113 |
| RuLer | 11 | 0.603 | 1 | 0.842 | 0.144 | 0.828 | 0.121 | 0.036 | 0.081 |
| Smote | 11 | 0.401 | 1 | 0.796 | 0.221 | 0.754 | 0.174 | 0.052 | 0.117 |
| **Randomforest Average precision** | | | | | | | | | |
| Algorithm | n | min | max | median | iqr | mean | sd | se | ci |
| Adasyn | 11 | 0.612 | 1 | 0.85 | 0.141 | 0.835 | 0.125 | 0.038 | 0.084 |
| Baseline | 11 | 0.655 | 0.999 | 0.841 | 0.153 | 0.84 | 0.115 | 0.035 | 0.077 |
| RuLer | 11 | 0.917 | 1 | 0.951 | 0.039 | 0.959 | 0.027 | 0.008 | 0.018 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Smote | 11 | 0.639 | 0.998 | 0.864 | 0.149 | 0.839 | 0.118 | 0.035 | 0.079 |
| **SVC Average precision** | | | | | | | | | |
| Algorithm | n | min | max | median | iqr | mean | sd | se | ci |
| Adasyn | 11 | 0.442 | 1 | 0.793 | 0.266 | 0.775 | 0.181 | 0.054 | 0.121 |
| Baseline | 11 | 0.443 | 1 | 0.796 | 0.198 | 0.783 | 0.171 | 0.052 | 0.115 |
| RuLer | 11 | 0.604 | 1 | 0.803 | 0.166 | 0.818 | 0.125 | 0.038 | 0.084 |
| Smote | 11 | 0.495 | 1 | 0.802 | 0.237 | 0.787 | 0.165 | 0.05 | 0.111 |

The pairwise comparisons using Wilcox and Sign tests, for the results of the KNN classifier, show significant differences between groups Adasyn - RuLer and RuLer - Smote. However, they do no yield significant differences between groups Baseline and RuLer. Looking at Table 5.3, although the RuLer algorithm (mean 0.828) outperforms the Baseline (mean 0.774), the difference is not enough to yield a consistent difference.

For the Randomforest classifier, the pairwise comparisons for the Wilcox and Sign test yield, respectively, significant differences between RuLer - Baseline and RuLer - Smote, as well as between RuLer - Adasyn and Baseline - Smote. This would be expected from the summary of Table 5.3, where the differences with Baseline, Smote and Adasyn are equal to or greater than 0.119.

These results show that the RuLer algorithm behaves either similar to Smote and Adasyn algorithms or better.

Moreover, considering all the classifiers, the mean accuracy of the RuLer surpasses the other classifiers.

### Precision and new instances

While oversampling applications are mainly focused on improving the precision of the classifiers trained with the resampled data, this research has special interest in newly generated instances. In other words, it is important to have instances that, when used to train the models, not only improve their prediction capacities but also produce a number of new instances that add variability or possibilities to the existing material.

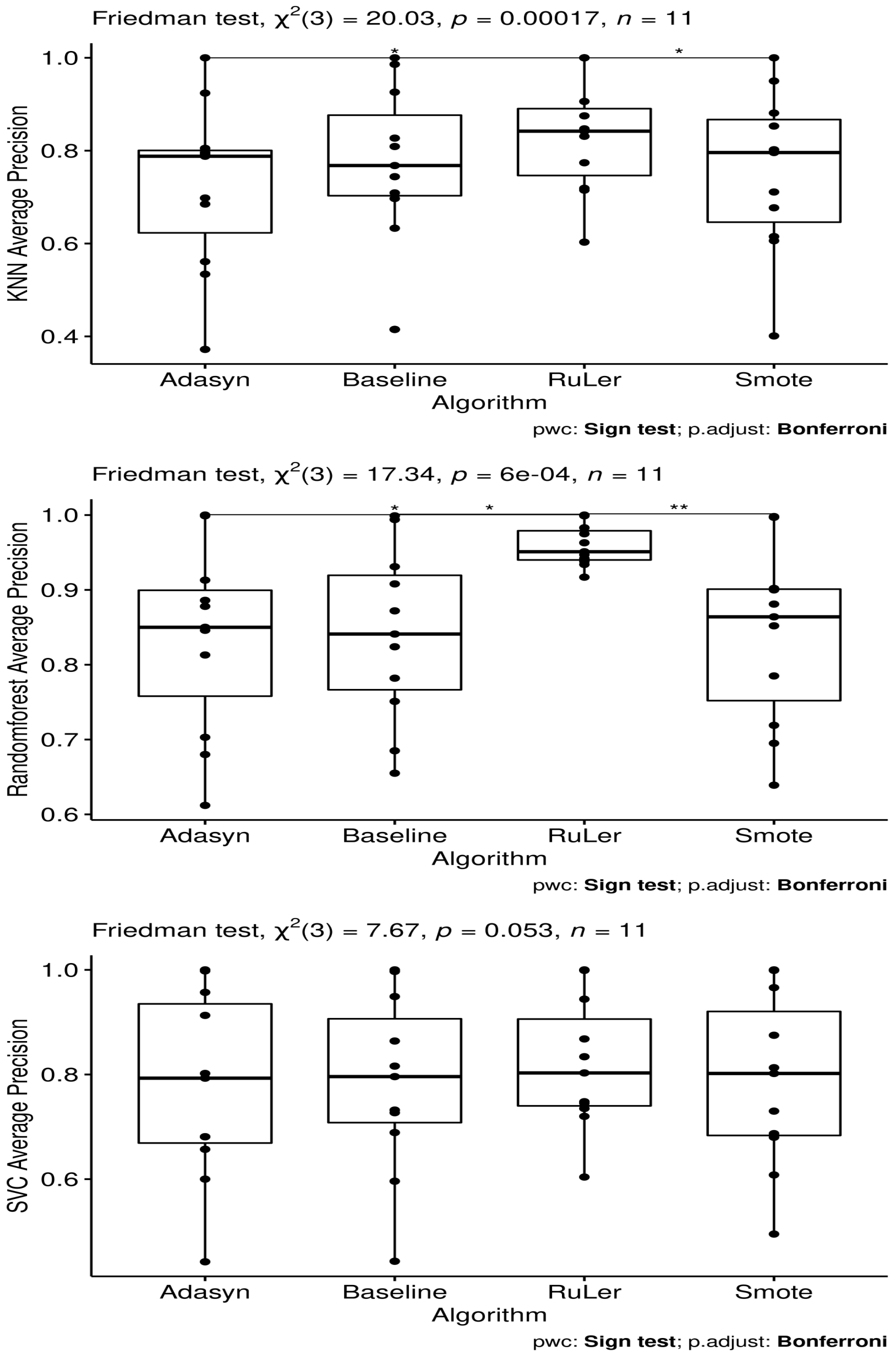

Figure 5.2: Boxplots of the average precision scores of the classifier (KNN, Randomforest and SVC) for Baseline, Smote, Adasyn and RuLer algorithms, considering all the dataset on Table 5.1.

Therefore, although the precision does not improve much in some particular cases, or it might even be slightly less, the new instances obtained fulfill the objective sought.

I cite a couple of these examples. The page-blocks0 dataset has 5472 original instances from which 4914 are negative. The RuLer algorithm oversampled 6293 positive instances, from which some were randomly selected to balance the dataset. The precision score obtained using the SVC improved from 0.72 in the Baseline case to 0.73 using the RuLer. However, the number of instances available after the oversampling would provide the performer with around 3000 new instances of a particular class to explore. Another example is the data set segment0 which originally had 230 positive instances and 1384 negative ones. The synthetic data produced by the RuLer algorithm achieved 2241 positive instances.

## 5.4 New created instances

The best results are reached when the random forest classifier is applied to the data oversampled with the RuLer algorithm. Further work derived from this research will be focused on analyzing this result. A possible hypothesis is that the patterns identified by the ruler with the dissimilarity function used, together with the new instances created by the create rule function, reinforce the way in which decision trees cut the space, following hyperplanes in the feature domain.

Figure 5.3 shows the original points for a tiny artificial, two-class example (positive and negative), together with the synthetic points extracted by Smote, Adasyn and RuLer algorithms. The RuLer algorithm creates new points by combining the values of the existing ones. This is because the way the *create_rule* function works. The new points share at least one value with an original one.

Ruler appears to be the boldest algorithm with respect to space exploration (i.e, the points created by the RuLer algorithm are the most distant ones from the original data). To verify this, one analyzes the distributions of the euclidian distances connecting the points created by Smote, Adasyn and RuLer, with respect to their closest point in the

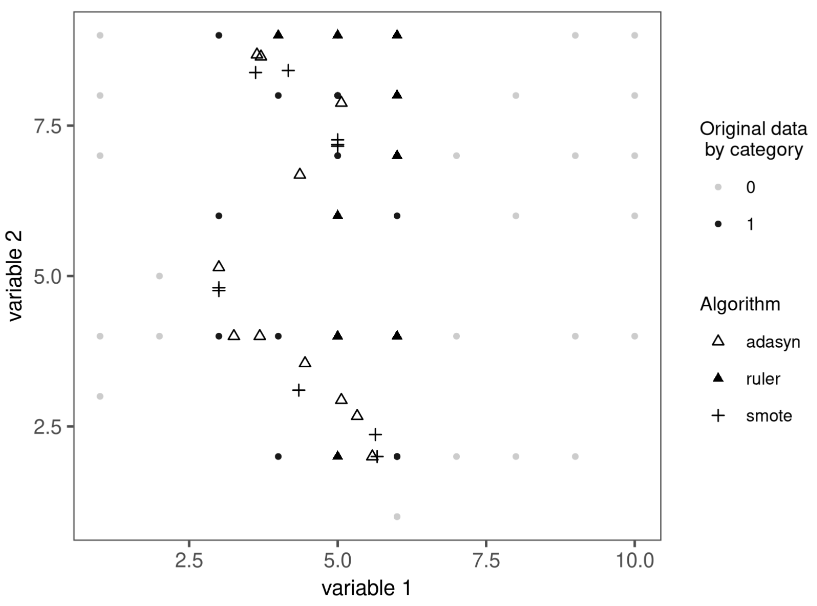

Figure 5.3: Points created by Smote, Adasyn and RuLer (dissimilarity metric = number of empty intersections among sets containing parameter values for each variable).

original data. This is done for *the sample dataset* composed of datasets: glass-0-1-2-3-vs-4-5-6, segment0, ecoli2 and yast-2-vs-4. In other words, for each point created by an oversampling algorithm, its distance to the closest point in the original data was calculated.

Figure 5.4 shows the histograms of the euclidian distances among the points generated by Adasyn, RuLer and Smote and its closest point in the original data for *the sample dataset*. Some of the points created by the RuLer are farther from the original data than points created by Smote or Adasyn. For example, in the first graph of Figure 5.4, showing the histogram of distances for glass-0-1-2-3-vs-4-5-6 dataset, points are seen beyond 4. Note that the euclidian distances respect the original values of the data.

To analyse the results shown in Figure 5.4, the pairwise comparisons among the frequency distributions of such distances (for *the sample dataset*) were compared using a one way anova test. The results are shown in Table 5.4.

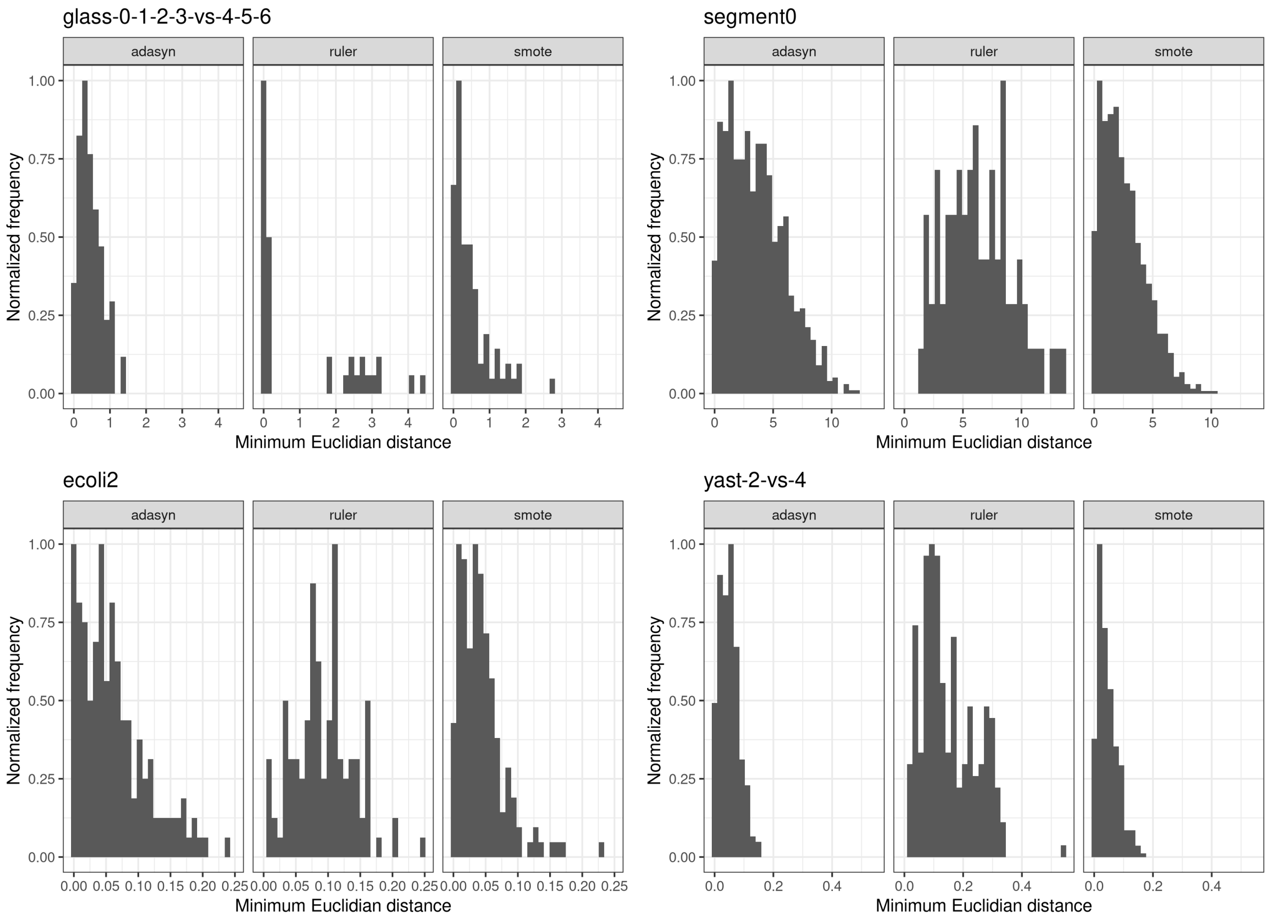

Figure 5.4: From top left to bottom right, histograms counting the distances among the points generated by adasyn, ruler and smote and the closest point in the original data for *the sample dataset*.

Table 5.4: One way anova test comparing the distributions of the distances to the closest point in the original data for the oversampled points created by algorithms Smote, Adasyn and RuLer. diff: difference between means of the two groups. lwr, upr: lower and the upper end point of the confidence interval at 95%. p adj: p-value after adjustment for the multiple comparisons.

| **dataset:glass-0-1-2-3_vs_4-5-6 attributes:9 instances:214 IR:3.2** | | | | |
|---|---|---|---|---|
| algorithm | diff | lwr | upr | p adj |
| ruler-adasyn | 0.575498836 | 0.2451193 | 0.9058783 | 0.0001623 |
| smote-adasyn | -0.007475601 | -0.3378551 | 0.3229039 | 0.9984308 |
| smote-ruler | -0.582974437 | -0.9133539 | -0.2525949 | 0.0001309 |
| **dataset:segment0 attributes:19 instances:2308 IR:6.02** | | | | |
| algorithm | diff | lwr | upr | p adj |
| ruler-adasyn | 2.9185273 | 2.017582 | 3.819472 | 0.0000000 |
| smote-adasyn | -0.7278831 | -1.628828 | 0.173062 | 0.1393161 |
| smote-ruler | -3.6464104 | -4.547356 | -2.745465 | 0.0000000 |
| **dataset:ecoli2 attributes:7 instances:336 IR:5.46** | | | | |
| algorithm | diff | lwr | upr | p adj |
| ruler-adasyn | 0.03319504 | 0.01986541 | 0.046524677 | 0.0000000 |
| smote-adasyn | -0.01854278 | -0.03187242 | -0.005213148 | 0.0033288 |
| smote-ruler | -0.05173782 | -0.06506746 | -0.038408189 | 0.0000000 |
| **dataset:yeast-2_vs_4 attributes:8 instances:514 IR:9.08** | | | | |
| algorithm | diff | lwr | upr | p adj |
| ruler-adasyn | 0.101482430 | 0.08879263 | 0.114172231 | 0.0000000 |
| smote-adasyn | -0.005198103 | -0.01788790 | 0.007491698 | 0.6011233 |
| smote-ruler | -0.106680534 | -0.11937033 | -0.093990732 | 0.0000000 |

In Table 5.4, the distributions are significantly different among RuLer-Smote and RuLer-Adasyn, as the p adj values of Table 5.4 show. On the contrary, the results

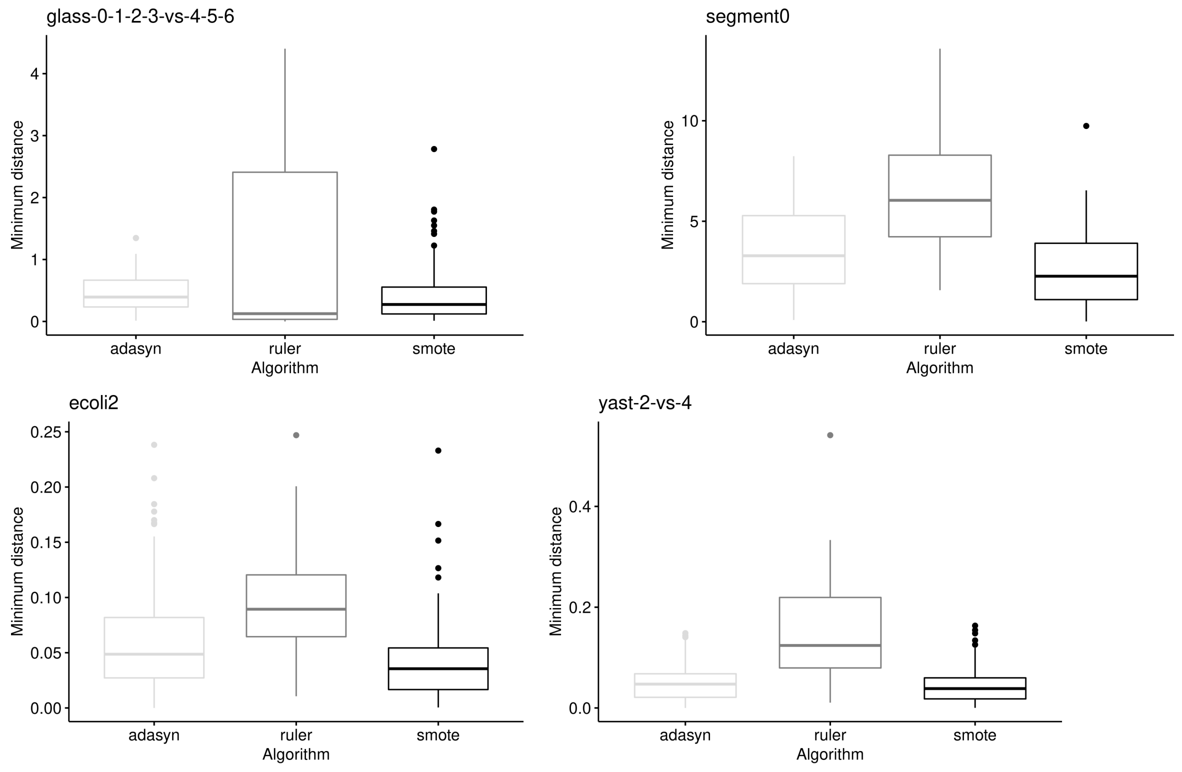

Figure 5.5: From top left to bottom right, boxplots of the points gerated by adasyn, ruler and smote for datasets glass-0-1-2-3-vs-4-5-6, segment0, ecoli2 and yast-2-vs-4 (*the sample dataset*).

comparing Smote-Adasyn show, as might be expected given that Adasyn is a modified version of Smote, that the resulting differences in the created data are not statistically significant.

The box-plots visualizing the distributions of the minimum distances from the created points to its closest one in the original data, presented in Figure 5.4 for each algorithm, are shown in Figure 5.5. The standard deviation of the data oversampled with the RuLer algorithm is greater than the standard deviations of the datasets oversampled using Smote or Adasyn. These results suggest that the RuLer algorithm explores different points from those sampled by Smote and Adasyn, yet the accuracy obtained by the classifiers trained by using the synthetic instances is comparable. An interesting possibility for further exploration would be to combine the algorithms in order to obtain a variety of points.

## 5.5 Conclusion

The evaluation of the algorithm presented in this Chapter is intended to complement the perceptual evaluation carried out in Chapter 4. Perceptual evaluation provides human feedback on the capacities of the newly generated instances being used within live performance, as well as on the affordances of the algorithm. However, the tests presented in this chapter provide a numerical evaluation on the algorithm's capacities for producing new instances that "behave", in principle, as the instances of a selected class. Strictly speaking, it is to what extent, when used for training, the classification algorithms improve their precision. In addition, we can look at the characteristics of the newly produced instances (e.g its capacity to explore the space).

The re-sampling tests yield remarkable state-of-the-art results. With instances appearing as significantly different from those produced by the Smote and Adasyn algorithms, which are the most widely used oversampling algorithms. In addition, the number of produced instances allowed balancing of the different datasets. Therefore, even in those cases in which the improvement in precision is not high, from the research perspective, the number of produced instances provide the live coder with new material to explore. Furthermore, this material explores wider regions of the space compared to Smote and Adasyn algorithms.

Oversampling algorithms perform the task most similar to the RuLer algorithm (i.e. extending a labeled data set searching for new data consistent with the classes). Strictly speaking, RuLer is intended to produce variations of the material with different degrees of "similarity" to the original data. But, up to certain limits, the idea of creating new data by analyzing the properties of the existent is shared.

Although these results do not intend to be conclusive (see the critic for the evaluation of musical systems presented in the text: Machine Learning Research that Matters for Music Creation: A Case Study (Sturm et al., 2019)), they provide an estimation of the algorithm capacities.

# Chapter 6

# FuzzyRuLer

This Chapter introduces FuzzyRuLer, an algorithm that extends the IF-THEN rules extracted by the RuLer algorithm, described in Chapter 4, to fuzzy rules that cover the whole feature space. The algorithm builds the cores of the fuzzy membership functions by avoiding contradictions during the induction process. The fuzzy model is evaluated by using cross validation, for which data collected during user tests together with extrinsic benchmarks (i.e datasets not belonging to the sound domain) were used. This numerical evaluation produces the best results for the data collected by users, and, for the external datasets, though it does not imply anything about the semantic perception of the instances, it does yield state-of-the-art results.

RuLer is an algorithm, designed for live coding performance, that receives a set of labeled presets and creates real time variations out of them (Paz, 2019b). It also allows for the addition of new input presets in real time and starts working with only two presets. The algorithm searches for regularities in the input data from which it induces a set of IF-THEN rules that generalize it. However, these rules only describe points that do not cover the whole feature space, providing little insight into how the preset labels are distributed. This Chapter introduces FuzzyRuLer (Paz et al., 2020), an algorithm able to extend IF-THEN rules to hyperrectangles, which in turn are used as the cores of membership functions to create a map of the input feature space. For such a pursuit, the algorithm generalizes the logical rules, solving the contradictions by following a maximum volume heuristic. The user controls the induction process through the parameters of the RuLer algorithm, designed to provide the affordances to control the balance between novelty and consistency in respect to the input data. The algorithm was evaluated both in live performances and by means of a classifier using cross-validation. In the latter case, as there are no datasets, we used a dataset collected during user tests and extrinsic standard benchmarks. The latter, although they do not provide musical information, do provide general validation of the algorithm.

Even though this is a purely aesthetic pursuit that seeks to create aesthetically engaging artifacts, it is interesting and relevant that the accuracy of the models reaches state-of-the-art results. This, together with the positive criticism that the performances and recordings received (see Section 4.3.4), suggests that rule learning is a promising approach, able to build models from few observations of complex systems.

## 6.1 FuzzyRuLer Algorithm

The FuzzyRuLer algorithm constructs a fuzzy rule set of trapezoidal membership functions out of logical IF-THEN rules. For that, it builds hyperrectangles (Section 6.1.1), which are the cores of the trapezoidal membership functions and, in turn, are used to fit the supports (Section 6.1.2).

### 6.1.1 Building cores

To build the cores, the algorithm extends the **sets** contained at the entries of the logical IF-THEN rules to **intervals** between their respective minimum and maximum values. For example, $r_1$ = [{1,4}, {3,5}, intro] is extended to $r_1$ = [ [1,4], [3,5], intro], including all the values in between 1 and 4 as well as between 3 and 5. Then, instead of four values, we have a region to choose from! Next, the contradictions that might appear between the created intervals are resolved. A contradiction appears when two rules with different labels or classes intersect each other. Two rules $r_1$ and $r_2$ intersect if, for all $i$ (i.e., parameter placed at position $i$ in the antecedent of the rule), there exists $x$ in $r_1[i]$ such that $y_1 \leq x \leq y_2$ with $y_1$, $y_2 \in r_2[i]$. If two rules with different classes intersect, it is enough to "break" one parameter to resolve the contradiction. For example, the contradiction between the rules $r_1$ and $r_2$ (at the top of Table 6.1 and depicted in Figure 6.1) can be resolved either as shown on the left or on the right of Figure 6.2.

Table 6.1: The contradiction between $r_1$ and $r_2$ can be resolved by "breaking" one parameter.

| **Rule** | **Parameter1** | **Parameter2** | **Class** |
|---|---|---|---|
| $r_1$ | [1,5] | [2,4] | calm |
| $r_2$ | [2,3] | [1,5] | harsh |
| | **First, partition** | | |
| $r_{1a}$ | [1] | [2,4] | calm |
| $r_2$ | [2,3] | [1,5] | harsh |
| $r_{1b}$ | [5] | [2,4] | calm |
| | **Second partition** | | |
| $r_1$ | [1,5] | [2,4] | calm |
| $r_{2a}$ | [2,3] | [1] | harsh |
| $r_{2b}$ | [2,3] | [5] | calm |

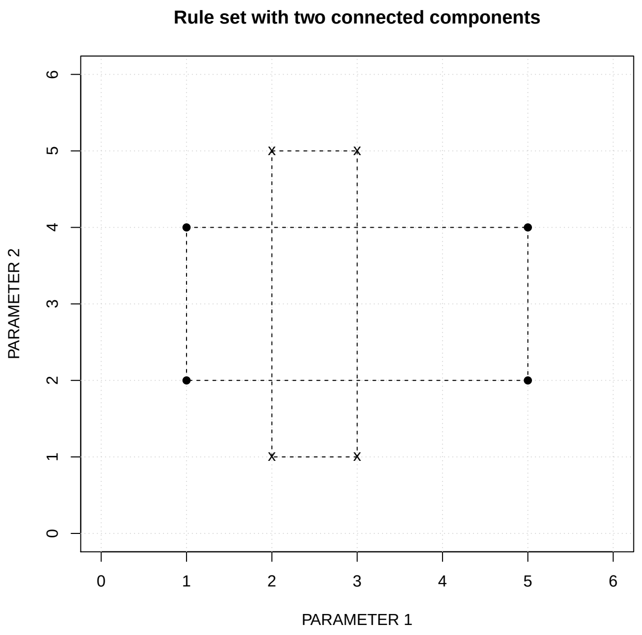

Figure 6.1: Rule [[2,3], [1,5], harsh] intersects rule [[1,5], [2,4], calm]. Harsh is represented by an "x" and Calm by a "." in the plot.

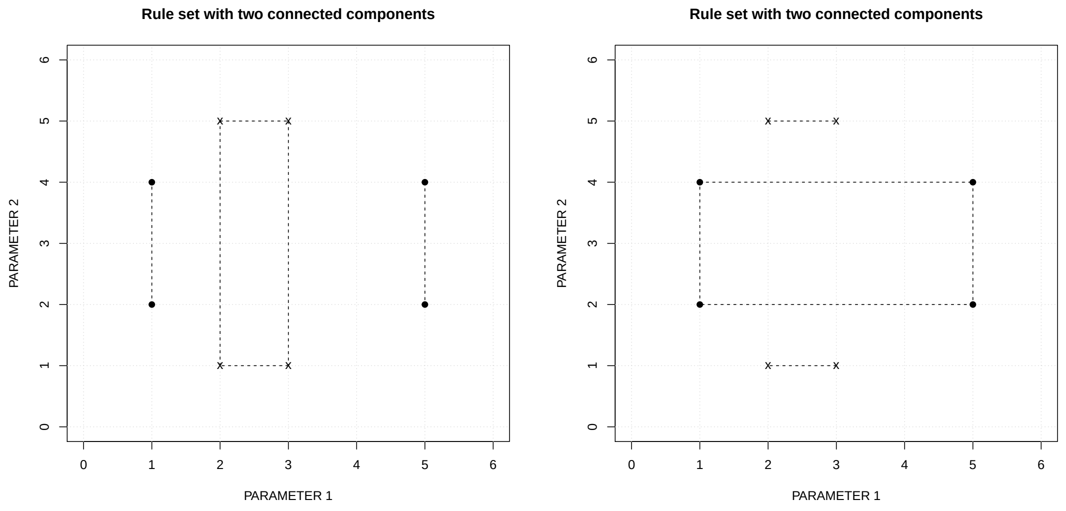

Figure 6.2: Two possible ways of resolving the contradiction that appears in Figure 6.1.

To select the partition, the Measure of each set of rules is calculated, and the one with maximum value is selected. The set with maximum Measure value is selected as it is the one that covers a wider region of the feature space. While the inductive process of the RuLer algorithm is intended to create new points, the generalization process of the FuzzyRuLer covers the entire observed space. Therefore, maximum coverage is the goal. The Measure of a single rule has components: *Extension* ($E$ ) and *dimension*, defined in Equation (6.1):

$$\begin{gathered} E = \sum_{i=0}^{N-1} E_i, \text{ where } E_i = |max_i - min_i| \\ dimension = \text{Number of } E_i \text{ such that } E_i \neq 0. \end{gathered} \tag{6.1}$$

In Equation (6.1), for each parameter $i$ in the rules, $E_i$ is the absolute value between its maximum and minimum values. For example, if $r[i] = \{11,13,15\}$, then $E_i = 4$, which is $|15 - 11|$. If $r[i] = \{3\}$, then $E_i = 0$.

The *Measure* of a set of rules collects the individual measures of the rules, adding those who have the same dimension. It is expressed as an array containing the extension for each dimension. When two measures are compared, the greatest dimension wins. For example, ($Extension = 1$, $dimension = 2$) $>$ ($Extension = 4$, $dimension = 1$). In the same way, ($Extension = 1$, $dimension = 3$) $>$ ($Extension = 100$, $dimension = 2$; $Extension = 100$, $dimension = 1$). Table 6.2 presents an example.

### 6.1.2 Fuzzy rule supports

Once the cores are known, there are many possibilities for building the supports of trapezoidal membership functions. Here, as the algorithm is designed for real performance, we construct the supports using the minimum and maximum values observed for each variable. In this way, the slopes of each trapezoidal membership function are defined automatically by how close the core is to the respective minimums and maximums. Thus, each rule covers the whole observed space and the supports are defined automatically by the cores, avoiding costly procedures that iteratively adjust the supports while the information is processed. This is done in the following way: For each parameter, the minimum and maximum values observed are calculated. If the parameter values are normalized, these values are 0 and 1. Then, the algorithm connects the extremes of each core with the respective minimum and maximum values of each parameter. See Figure 6.3 for an example.

Table 6.2: Example of *extension (E)* and *dimension (dim)* for a set of rules. Note that rules with different categories contribute to the global Measure.

| **Rules and** | **Parameter Values and Category** | | |
|---|---|---|---|
| **Measures** | ***Parameter 1*** | ***Parameter 2*** | ***Category*** |
| rule $r_1a$ | [1] | [2,4] | calm |
| Measure $r_1a$ | $E_1 = 0$ | $E_2 = 2$ | $E = 2$, $dim = 1$ |
| rule $r_2$ | [2,3] | [1,5] | harsh |
| Measure $r_2$ | $E_1 = 1$ | $E_2 = 4$ | $E = 5$, $dim = 2$ |
| rule $r_1b$ | [5] | [2,4] | calm |
| Measure $r_1b$ | $E_1 = 0$ | $E_2 = 2$ | $E = 2$, $dim = 1$ |
| **Measure: $E$=5, $dim$=2; $E$=4, $dim$=1** | | | |
| rule $r_1$ | [1,5] | [2,4] | calm |
| Measure $r_1$ | $E_1 = 4$ | $E_2 = 2$ | $E = 6$, $dim = 2$ |
| rule $r_2a$ | [2,3] | [1] | harsh |
| Measure $r_2a$ | $E_1 = 1$ | $E_2 = 0$ | $E = 1$, $dim = 1$ |
| rule $r_2b$ | [2,3] | [5] | harsh |
| Measure $r_2b$ | $E_1 = 1$ | $E_2 = 0$ | $E = 1$, $dim = 1$ |
| **Measure: $E = 6$, $dim = 2$; $E$=2, $dim = 1$** | | | |

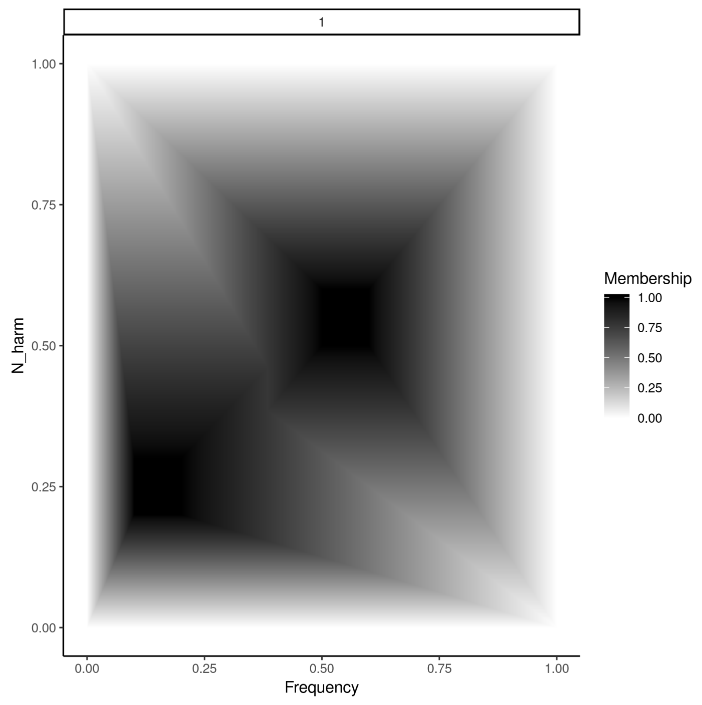

Figure 6.3: Two fuzzy rules (scaled into [0,1]) of a hypothetical Category 1 (shown at the top of the graph). The $x$-axis represents the frequency of an oscillator and the $y$-axis the number of upper harmonics added to it. The membership of a point (Frequency, N_harm) to Category 1 is indicated by the membership scale at the right of the graph.

In Figure 6.4, the fuzzy-rule model composed by rules: [[0.1, 0.2], [0.2, 0.3], 1], [[0.5, 0.6], [0.5, 0.6], 1], [[0.7, 0.9], [0.4], 2] is shown. Note that these rules describe the cores of the membership functions from which the supports are build. In the Figure, each point in the space is assigned to its maximum membership. The points with class "1" are shown at the top of the Figure; the cores of the rules appear with membership 1. The $x$-axis represents the frequency of an oscillator and the $y$-axis the number of upper harmonics added to it. The intervals are normalized, as in the previous example, to [0,1]. A similar example is shown in Figure 6.5.

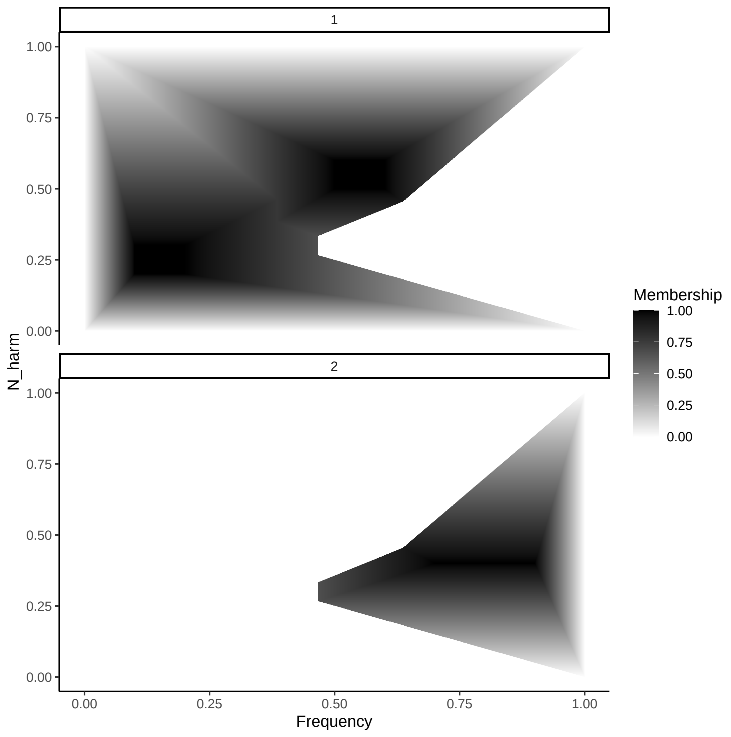

Figure 6.4: The space is classified by the fuzzy-rule model composed by rules: [[0.1, 0.2], [0.2, 0.3], 1], [[0.5, 0.6], [0.5, 0.6], 1], [[0.7, 0.9], [0.4], 2]. Note that the rules describe the cores of the membership functions. Then, the cores connect with the extremes of the intervals. The degree of membership to each class is shown by the scale at the right of the Figure. The rules with class “1” are shown at the top of the figure. The $x$-axis represents the frequency of an oscillator and the $y$-axis the number of upper harmonics added to it. The intervals are normalized to [0,1].

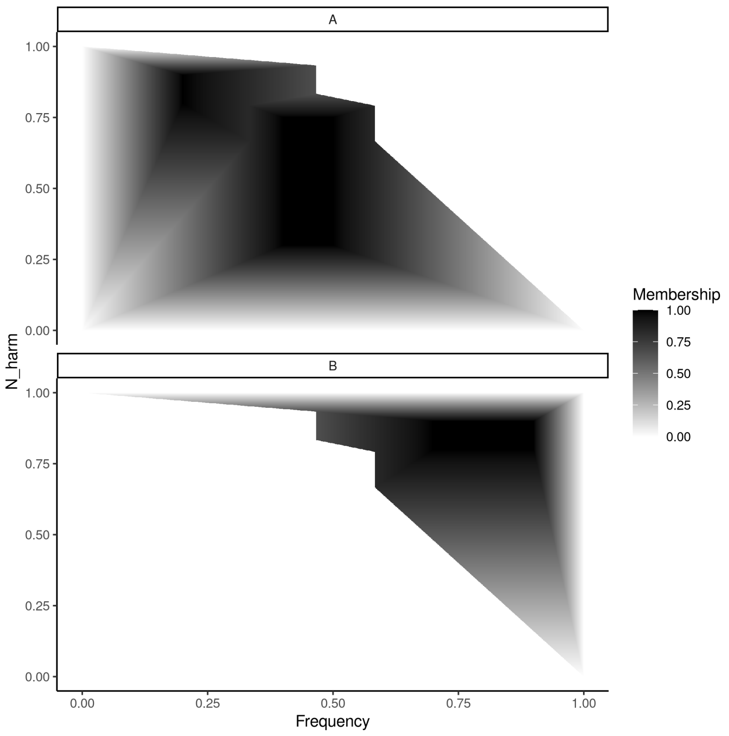

Figure 6.5: Membership values of the space (normalized to [0,1]) classified with a fuzzy-rule model composed by rules: [[0.2], [0.8, 0.9], A], [[0.4,0.5], [0.3, 0.75], A], [[0.7,0.9], [0.8,0.9], B]. The $x$-axis represents the frequency of an oscillator and the $y$-axis the number of upper harmonics added to it.

The process to build the supports out of the cores can be formally described as follows: Each rule $r \in R$ has the form $r = r_1, r_2, ..., r_{n-1}, c_i$ where each $r_i$ is an interval and $c_i \in C$ is one of the possible categories. For example, $R = \{r_1, r_2\}$, where $r_1 =$ [ [1,3], [1], A ] and $r_2 =$ [ [5,7], [3,5], A ].

Let $r^*$ be the set defined in Equation 6.2:

$$r^* = \{r_1^*, ..., r_{n-1}^* | \, \forall \, r_i \text{ in } r \in R, r_i \subset r_i^*\} \tag{6.2}$$

In our example $r^* = \{r_1^*, r_2^*\} = \{\{1, 3, 5, 7\}, \{1, 3, 5\}\}$. $r_i^*$ is a set containing all the values for each of the $i$ parameters of the current set of rules. Then, Algorithm 4 describes how to build the membership functions that create the fuzzy rules.

**Algorithm 4** Fuzzy rules with trapezoidal membership functions

```
1: function FUZZY-RULES-WITH-TRAPEZOIDAL-MEMBERSHIP-FUNCTIONS(R)
Require: set of rules
2:     create $r^*$ as in Equation (6.2)
3:     Fuzzy-Rules = list
4:     for $j \leftarrow 0$ to size of R do
5:         r = R[j]
6:         fuzzy-r = list
7:         for $i \leftarrow 1$ to $n - 1$ do
8:             a = min($r_i^*$)
9:             b = min($r_i$)
10:            c = max($r_i$)
11:            d = max($r_i^*$) (As the rules are normalized a = 0 and d = 1).
12:            append list [a,b,c,d] to fuzzy-rule
13:        end for
14:        append r[n] to fuzzy-r and save fuzzy-r in Fuzzy-Rules
15:    end for
16:    Return Fuzzy-Rules
17: end function
```

### 6.1.3 Fuzzy classifier

To classify a new preset $P = (v_1, \ . \ . \ . \ ,v_{N-1})$, proceed as follows: For each rule $r_k$, calculate the membership of each feature value i.e., $\mu_{k,i}(v_i)$. Then, calculate its firing strength $\tau_k(P)$, which measures the degree to which the rule matches the input parameters. It is defined as the minimum of all the membership values obtained for the parameters (see Equation (6.3)).

$$\tau_k(P) = min\,\{\,\mu_{k,i}(v_i)\,\} \tag{6.3}$$

Once the firing strength has been calculated for all rules, the assigned class will be equal to the class of the rule with maximum firing strength, as in Equation (6.4):

$$Class(P) = \text{Class of } R_c \text{ where C} = \underset{k}{arg\,max}\,\{\tau_k(P)\} \tag{6.4}$$

An example of the classification process for a hypothetical system with two rules each with two parameters is shown in Figure 6.6.

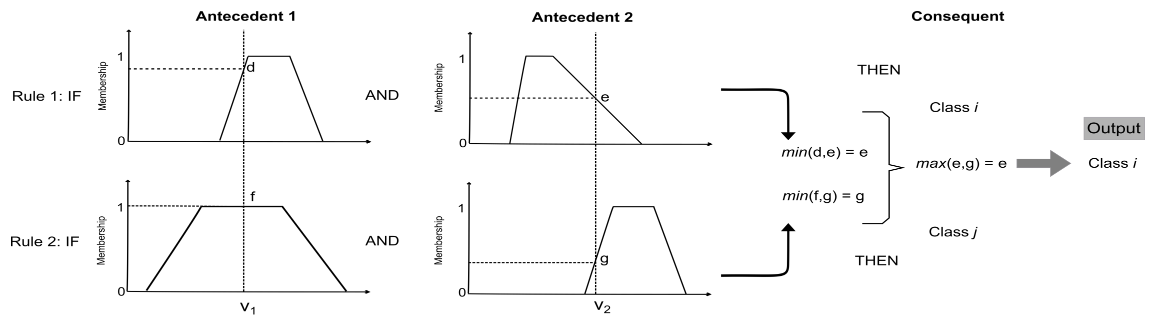

Figure 6.6: Example of classification process for a system with two rules and two parameters. The new combination $P = (v_1, v_2)$. For the first rule $\mu(v_1) = d$ and $\mu(v_2) = e$. The minimum of these values is $e$. For the second rule $\mu(v_1) = f$, $\mu(v_2) = g$ and $min(f, g) = g$. Finally, $max(e, g) = e$ and therefore the class assigned to the instance is $Class\ \ i$.

## 6.2 Evaluation

Evaluation of automatic synthesizer programmers has followed two main approaches: user tests, in which expert musicians are interviewed after using the algorithm; And, similarity measures, in sound matching tasks, where candidate sounds are compared with the target are used.

In our case, the evaluation includes:

1. The analysis of how the model generalizes a user test dataset. This evaluation is reinforced by other extrinsic benchmarks (Section 6.2.1).

2. Analysis of the extracted rules (Section 6.2.2).

3. The evaluation of the performances where the project has been presented and the lists where the compositions made with the algorithms have been included (Section 4.3.4).

As one of the objectives of the FuzzyRuLer algorithm is to provide new presets classified with the same labels of the input data, the generalization using the user-labeled data are evaluated by cross-validation. The classifier used for that purpose was presented in Section 6.1.3 . When new instances are classified, the classifier assigns to them the label that it will assign to the same combinations if the rule model is used to produce new presets. In addition, cross-validation allows for the assessment of the performance of the algorithm using benchmarks in a task for which datasets might not exist, as it is the case of live coding.

### 6.2.1 Cross-validation

To test how the algorithm models the feature space of a synthesis algorithm, we used the data set described in (Paz et al., 2017). This dataset was generated by user tests, in which different configurations of a Band Limited Impulse Oscillator (Blip, 2019) were programmed by users and tagged either as *rhythmic*, *rough* or *pure tone*. For this, the users tweaked the device parameters of the synthesis algorithm: fundamental frequency and number of upper harmonics (which are added to the fundamental frequency). Then, the parameter combinations that produced any of the searched categories were saved together with the corresponding label. The data set is shown in Figure 6.7.

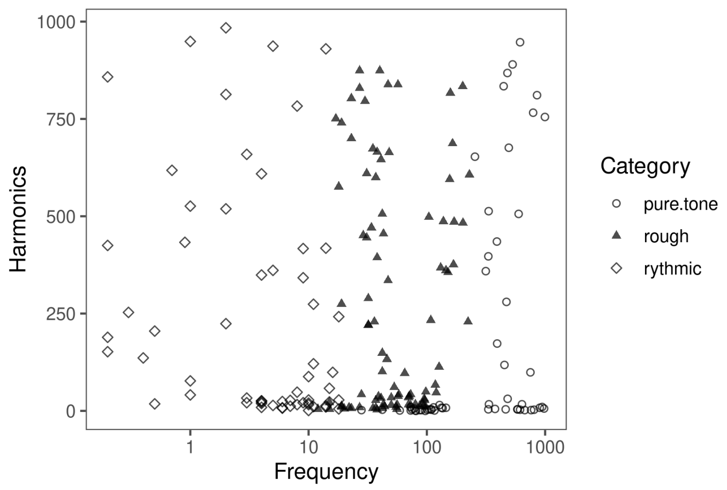

Figure 6.7: Band Limited Impulse Oscillator (Blip) data set. The $x$-axis shows the log of the fundamental frequency of the impulse generator. The $y$-axis shows the number of upper harmonics that are added to the fundamental frequency. The categories associated with the combinations (*rhythmic*, *rough* or *tone*) are shown at the right side of the graph.

In addition, four datasets from the UCI repository (Dua and Graff, 2017) were selected. As they belong to diverse domains and have different unbalanced degrees, they provide a general idea of how the algorithm behaves.

The results of the fuzzy classifier of Section 6.1.3 were compared with K-Nearest Neighbours, Support Vector Machine (with kernels linear, polynomial degree 2 and rbf) and Random forest classifiers.

The K-Nearest Neighbours classifier is described in Section 5.2. The Support Vector Machine (SVM) is an algorithm with good generalization capabilities and nonlinear data handling using the kernel trick. In addition, small changes in the data do not affect its hyperplane. However, choosing an appropriate Kernel function is difficult and the algorithmic complexity and memory requirements are very high. As a consequence, it has long training times. In addition, the resulting model is difficult to interpret.

The Random Forest (RF) is based on the bagging algorithm and uses an Ensemble Learning technique. It creates many trees and combines their outputs. In this way, it reduces both the overfitting problem of decision trees and the variance, improving the accuracy. It handles nonlinear parameters efficiently. However, as it creates lots of trees, it requires computational power and resources. Using the RF to compare is interesting because these algorithms are normally considered the alternative to rule learning. However, while an RF algorithm might indeed perform as easily and quickly as the FuzzyRuler, its only parameter, the number of trees, is not as expressive and interpretable for the user as parameters $d$ and $ratio$ for controlling the induction process.

Together, these algorithms provide a spectrum to compare the classifier against. For each dataset, the model parameters producing the highest 10-fold (70% training and 30% test) cross-validation accuracy were selected. For the SVM, tested parameter values for C and gamma were respectively [0.01, 0.1, 1, 10, 100, 1000] and [1, 0.1, 0.01, 0.001, 0.00001, 0.000001, 10]. For KNN, the tested N values were [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and for the Random forest [1, 10, 100, 500,1000] trees were considered. In the case of the FuzzyRuLer, $d$ was explored from 1 to half the number of features in the dataset and $ratio$ with [0.9, 0.8, 0.7, 0.6, 0.5] values. Table 6.3 presents for each model the parameter selected and the accuracy obtained.

Table 6.3: Data sets Wine, Wine-quality-red, Glass and Ionosphere, selected from the UCI repository (Dua and Graff, 2017). The Blip data set was obtained from (Paz et al., 2017). The accuracy was calculated using 10-fold cross validation.

| **Data** | **Algorithm** | **Parameters** | **Mean Accuracy 10-fcv** |
|---|---|---|---|
| Wine | SVM linear kernel | best C = 0.1 | 0.9717 |
| | KNN | neighbors = 1 | 0.7514 |
| | RANDOM FOREST | trees = 100 | 0.9830 |
| | FuzzyRuLer | d = 9; ratio = 0.7 | 0.9554 |
| | SVM poly 2 | C = 0.01; gamma = 1 | 0.9717 |
| | SVM rbf | C = 1000; gamma = 1 * $10^{-5}$ | 0.9378 |
| Wine-quality-red | SVM linear kernel | C = 100 | 0.6 |
| | KNN | neighbors = 9 | 0.5475 |
| | RANDOM FOREST | trees = 10 | 0.59 |
| | FuzzyRuLer | d = 1; ratio = 0.5 | 0.6204 |
| | SVM poly 2 | C = 0.01; gamma = 0.001 | 0.64 |
| | SVM rbf | C = 1; gamma = 0.1 | 0.66 |
| Glass | SVM linear kernel | C = 1000 | 0.6384 |
| | KNN | neighbors = 6 | 0.6760 |
| | RANDOM FOREST | trees = 1000 | 0.6572 |
| | FuzzyRuLer | d = 6; ratio = 0.8 | 0.6636 |
| | SVM poly 2 | C = 0.1; gamma = 1 | 0.6666 |
| | SVM rbf | C = 10; gamma = 0.1 | 0.6854 |
| Ionosphere | SVM linear kernel | C = 10 | 0.8857 |
| | KNN | neighbors = 1 | 0.86 |
| | RANDOM FOREST | trees = 1000 | 0.9342 |
| | FuzzyRuLer | d = 6; ratio = 0.5 | 0.9033 |
| | SVM poly 2 | C = 0.1; gamma = 1 | 0.92 |
| | SVM rbf | C = 10; gamma = 0.1 | 0.9485 |
| Blip | SVM linear kernel | C = 1 | 0.8097 |
| | KNN | neighbors = 4 | 0.8195 |
| | RANDOM FOREST | trees = 500 | 0.8585 |
| | FuzzyRuLer | d = 2; ratio = 0.8 | 0.8690 |
| | SVM poly 2 | C = 0.1; gamma = 0.1 | 0.89 |
| | SVM rbf | C = 1; gamma = 0.1 | 0.775 |

Table 6.3 shows the cross-validation mean accuracy results obtained for each classifier and dataset. Table 6.4 presents the general mean and standard deviation for each classifier. These results show that the FuzzyRuLer yields similar results to those achieved by state-of-the-art classification algorithms. There exists abundant literature applying different machine learning algorithms to the UCI datasets; see, for instance, (Khan et al., 2018). However, the algorithms are used for a variety of purposes and under different conditions. For example, their evaluations use different partition schemes or sometimes are performed using techniques that trade execution time to gain accuracy (e.g., leave-one-out). Here, some references intended to frame the obtained results are presented. However, the reader has to keep in mind that these experiments are not completely comparable.

For the Wine dataset, according to (Dua and Graff, 2017), the classes are separable, though only Regularized Discriminant Analysis (RDA) has achieved 100% correct classification. The reported results are RDA : 10 0%, quadratic discriminant analysis (QDA) 99.4%, linear discriminant analysis (LDA) 98.9%, 1NN classifier 96.1% (z-transformed data). In all cases, the results have been obtained using the leave-one-out technique.

In (Cortez et al., 2009), using the Wine-quality-red dataset with a tolerance of 0.5 between the predicted and the actual class, the SVM best accuracies for this dataset were around 57.7% to 67.5%.

For the Glass dataset, (Khan et al., 2018) report the following accuracy results: KNN 0.6744, SVM 0.7442 and Large Margin Nearest Neighbors (LMNN) 0.9956.

Finally, for the Ionosphere dataset, in (Ding et al., 2015), Deep Extreme Learning Machines (DELM) were used for classification. According to the report, the multilayer extreme learning machine reaches an average test accuracy of $0.9447 \pm 0.0216$, while the DELM reaches an average test accuracy of $0.9474 \pm 0.0292$. In (Khan et al., 2018), they report the following results KNN 0.8, SVM 0.8286, LMNN 0.9971.

Table 6.4: Mean and standard deviation achieved for each classifier considering all the datasets.

| **Classifier** | **mean** | **sd** |
|---|---|---|
| FuzzyRuLer | 0.802 | 0.150 |
| KNN | 0.731 | 0.124 |
| Random-forest | 0.805 | 0.173 |
| SVM-linear-kernel | 0.781 | 0.159 |
| SVM-poly-2 | 0.818 | 0.153 |
| SVM-rbf | 0.801 | 0.136 |

To compare if mean accuracies are significantly different between algorithms, we performed a statistical test. As the predictor variables are categorical and their outcomes are quantitative, we performed a comparison of means test. As there are more than two groups being compared, but there is only one outcome variable, the statistical test is the one-way-ANOVA.

Table 6.5 shows that the *p*-value of the one-way analysis of variance is greater than the significance level 0.05, from which we conclude that there are not significant differences between the groups. The Tukey multiple comparisons of means yields 95% family-wise confidence level. Together, these results suggest that the fuzzy model could be used to generate new instances.

Table 6.5: One-way analysis of variance of the means shown in Table 6.4.

|  | **Df** | **Sum Mean** | **Sq** | **Fvalue** | **Pr (>F)** |
|---|---|---|---|---|---|
| Classifier | 5 | 0.0242 | 0.004832 | 0.214 | 0.953 |
| Residuals | 24 | 0.5408 | 0.022532 |  |  |

### 6.2.2 Extracted rules

Figure 6.8 shows the fuzzy rules obtained for the three categories of the "Blip" data set (shown in Figure 6.7) by using the FuzzyRuLer algorithm.

Although the Blip is a simple data set, it provides insight into the algorithm capacities for identifying the underlying structures that codify the categories. In Figure 6.8, it can be seen that the ranges in the frequency that separate the categories are consistent with the perception thresholds described in (Roads, 2001). These are: from 0 Hz to approximately 20 Hz the category is *rhythmic* no matter the number of harmonics added. From 20 Hz depending on the number of harmonics added, the sensation is *rough* until approximately 250 Hz. If the frequency is greater than 20 Hz and there are no harmonics added, or if the frequency is greater than approximately 250 Hz, the sensation is *pure tone*.

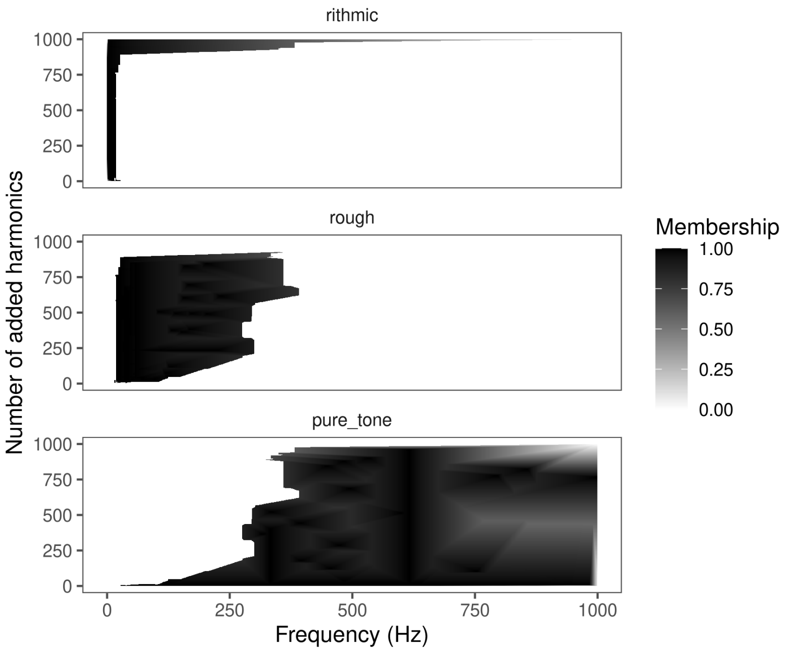

Figure 6.8: Extracted fuzzy rules for the three categories of the blip data set. The degree of membership to the class is shown at the right side of the image.

## 6.3 Conclusions

Real-time synthesizer programming in live coding imposes challenges to the intended use of learning algorithms, which provide numerous well-chosen examples, and has processes for data cleaning, learning and testing before selecting the final model.

Here, on the contrary, the examples are collected in real time, sometimes including musician mistakes that have to be managed as *glitches* and integrated into the performance. In cases when the data are pre-selected, the size of the datasets may be small. In other words, in this artistic practice, although it is also possible to include already trained models, the artists focus on having real-time feedback, creating the dataset mid-performance. Then, real-time algorithms that operate with small noisy data are also needed.

Inductive rule learning has offered interesting results within this context. However, the number of inducted instances is reduced, and the resulting IF-THEN rules provide a poor visualization of the space. The fuzzy rule learning algorithm presented in this Chapter is able to build fuzzy rule models of the feature space out of a set of IF-THEN rules. The resulting set provides an image of the class distribution in the feature space that helps musicians to have a quick insight into the inner workings of the synthesis algorithm. As the new examples only modify the rules that they "touch", the general model can manage outliers, integrating them into the model. The model has been evaluated during live performances and recordings that have been well-received by the community. The performances and reviews are available as part of the references (See references 1, 2 and 3 in Related publications/Press, critical reviews and performances). Finally, the model was also evaluated using cross-validation, comparing its results with those obtained by KNN, SVM (linear, polynomial degree 2 and rbf) and Random Forest classifiers. The one-way analysis of variance shows that there exist no significant differences among the algorithms. These results together suggest that the algorithm is a promising approach to be used in contexts, such as live coding, where the focus is not necessarily placed in model accuracy but, for example, in having real-time feedback of the algorithmic process.

# Chapter 7

# Conclusion and Further Research

This manuscript explores algorithmic processes for automatic programming of sound synthesis algorithms in the context of the performative artistic practice known as live coding (Collins et al., 2003; Magnusson, 2015). Specifically, it explores symbolic rule learning, given that its output is human-readable, and inductive methodologies allow one to create material as generative algorithms do.

Live coding implies conducting sound by real-time intervention of synthesis algorithm parameters. Coding a piece on-the-fly requires one to bridge the cognitive gap associated with devices' huge parameter spaces and the possible nonlinear sound variations built-in within them.

One possible approach is to have some pre-selected parameter combinations, of which the aural result is known, as a starting point for the performance. Another possibility is to search for combinations mid-performance embracing the risk, either way, the performance unfolds from these settings, creating variations that develop the piece. Automatic production of variations is a task that has occupied generative algorithms for a long time.

The algorithm proposed, named RuLer, operates on a set of labeled parameter combinations, from which it can produce new material with different degrees of variation by analyzing the existent regularities in the input data. The degree of variation is controlled by the performer who can then navigate a spectrum from recovering the original input material to producing strong recombinations.

This task is different from those addressed either by preset generation approaches centered in sound matching or by interactive evolution systems used for space exploration. Yet the conceptualization of the system shares similarities with both of them. For example, its exploratory nature resembles interactive evolution systems, and having initial information, from which to conduct a search, resembles automatic preset programmers.

During the performance, the live coder runs the algorithm iteratively and listens to/reviews the results, adjusting its parameters while unfolding the piece. The different degrees of variation provide the material for its development. The data collection process can be incremental, so the system can start with only a few examples, and new data can be collected during the performance or even over several performances. This can be done by allowing the user to save/delete settings on-the-fly and updating a database.

The algorithm design responds to the specific restrictions imposed by the application domain, namely: small data sets and/or data being collected mid-performance, needing real-time feedback and interpretable models.

RuLer performs inductive rule learning with a bottom-up strategy in a separate-and-conquer fashion. Iteratively, each example in the input data is selected and compared with the other instances, searching for patterns. These are defined based on a dissimilarity function that analyses the regularities in the conjunctions of rules with the same label. The live coder selects the allowed degree of dissimilarity among two rules necessary for them to exhibit a pattern. When patterns are found generalization is performed, the system avoids contradictions, and the “inductive leap” is controlled by the user. The user then defines an allowed “ratio” of the original instances that need to be contained in a candidate rule for it to be accepted. This limits the level of generalization and, therefore, the variations allowed.

The RuLer algorithm outputs an IF-THEN rule model. It produces outputs independent of the order of the input data, accepts multiple data types and returns all the regularities found expressed as rules. This last characteristic was selected because the intention of the model is to offer the performer as many variations as possible. Thus, no high-level routine to reduce the number of rules (e.g by pruning them) was implemented. However, in further implementations, for example, if the system keeps learning over various performances, that functionality might be useful.

The evaluation of the extracted rule model was performed by user surveys and by comparing the algorithm with state-of-the-art oversampling algorithms (Smote and Adasyn). In this way, subjective and numerical evaluations were considered, providing a general view of the algorithm's capacities and limitations.

Oversampling algorithms were selected as they perform a process similar to the objectives pursued by the RuLer. That is, oversampling algorithms receive labeled data and create new instances consistent (intended to be classified) with the data labels of the input.

Although the data being correctly classified (or the classifier accuracy improving by using synthetic data) does not imply that the new instances will be perceived by a user as consistent with a class or even as interesting variations, contrasting this evaluation with the user surveys offers a better overview of the algorithm.

The oversampling tests yield remarkable state-of-the-art results. The classifiers trained with data created by the RuLer, surpassed those trained with data oversampled using Smote and Adasyn. The comparisons were performed by using the average precision scores for three classifiers: KNN, SVC and Random Forest. The extrinsic, selected data sets have different numbers of instances (214 to 5472), attributes (5 to 19), imbalance degrees (1.82 to 9.08) and have binary labels. The best results, using the RuLer algorithm, were obtained by the Random Forest classifier, which reached the highest score for every data set. A possible explanation is that decision trees divide the space using planes, in a similar way in which RuLer builds new examples.

To analyze the results statistically, pairwise comparisons using Wilcox and Sign tests were performed. These tests considered the three classifiers for the eleven data sets used. Their results showed significant differences between groups Adasyn - RuLer and RuLer - Smote. Moreover, the synthetic instances produced by the RuLer explored more distant places (measured in Euclidean distance) than those produced by the other algorithms. To test this hypothesis, a one-way-ANOVA test among groups was performed. The results showed that instances created by the RuLer algorithm are indeed significantly different from those produced by Smote and Adasyn. In addition, the number of produced instances allowed one to *balance* the different data sets even in cases with a high imbalance ratio. Therefore, even in those cases in which the improvement of the classifier when trained with balanced data was not considerable the number of produced instances provide the live coder with new material to explore. Furthermore, as shown by the one-way-ANOVA, this material explores wider regions of the space than Smote and Adasyn algorithms do.

Accompanying oversampling tests, listening surveys with live coders were carried out. These used a specific data set taken from a musical piece, from which variations were produced using the RuLer algorithm. The perceived changes in the new data depend on the topology of the chosen synthesizer and, therefore, might vary from one algorithm to another. Despite that, the results of the user test show that, as the level of generalization allowed increases, the number of new combinations successfully evaluated decreases, though not more than 25%. Furthermore, it allowed recombination in 12 out of 16 attributes! This was to be expected; the greater the level of generalization, the more instances can be perceived as “randomly”generated. However, the fact that this decrease is not so drastic suggests that the generated instances maintain part of their original quality. Again, this depends on the topology of the synthesis algorithm, but, as the generalization level can be controlled by the live coder, if the topology is non-linear, then low generalization levels can be used if no harsh variations are sought.

Both evaluations (surveys performed on a set of given presets and oversampling algorithms using imbalanced data benchmarks) point in the same direction. Namely, the instances created by the inductive algorithm in both experiments are, respectively, perceived as suitable variations and classified within the data labels in most cases.

IF-THEN rules only describe points that do not cover the whole feature space, providing little insight into how the preset labels are distributed. Thus, FuzzyRuLer is presented. It is an algorithm able to extend IF-THEN rules to hyper-rectangles, which in turn are used as the cores of membership functions to create a map of the input feature space. For such a pursuit, the algorithm generalizes the logical rules solving the contradictions by following a maximum volume heuristic. Such a heuristic calculates the maximum hyper-rectangles that produce no contradictions. To reduce the search space: First, the sets of rules that would intersect during the generalization process are calculated. Then, for each set, all the possible partitions are built and their measures compared. By identifying the sets of rules that can produce contradictions and treating them separately, the algorithm actually calculates all the possible extensions and returns the optimal solution. The resulting rule set constitutes the cores of the fuzzy rules. Their supports are built such that the resulting rules cover the entire space.

The evaluation of the fuzzy rule model was carried out by means of cross-validation, comparing the results with state-of-the-art classifiers (SVM, Random Forest and KNN). For that propose, a simple fuzzy classifier was built on top of the extracted fuzzy-model. For the tests, extrinsic benchmarks were used together with a data set collected by users using a simple sound synthesis algorithm. The cross-validation tests yield results similar to those reached by state-of-the-art classifiers. Although, in these tests, the FuzzyRuLer did not produce the highest scores for all datasets, it did for the data set collected during user tests. That is, it performed better than the other classifiers with data belonging to its intended application domain.

Again, we used a one-way-ANOVA to test whether significant differences were found between the different algorithms considering all data sets. On this occasion, the comparisons suggest that no significant differences among the results of the classifiers exist. That is, the FuzzyRuler does not consistently appear lower or higher than the other algorithms. These are promising results, suggesting that the created n-dimensional regions based on the points contained in the IF-THEN rules do contain instances belonging to the labels of the rules. Moreover, the dataset collected via user tests suggest that, in that specific case and for the instances in the test-set, the regions described by the rules effectively produce instances within the corresponding perceptual labels. Furthermore, the regions in space covered by the cores of the rules provide the live coder with more points from which to choose new possible instances.

Obtaining the best results using these algorithms requires listening to the new instances produced and adjusting the desired level of generalization according to the data set. This is possible thanks to the algorithm parameters' that allow this operation to be carried out. If the perception varies strongly as a function of the space, the live coder can choose a minimum induction or no induction at all.

Some interesting future research would be to explore how the results vary for different sound synthesis algorithms or when using a greater variety of dissimilarity and creating_rule functions. It is quite possible that many of the algorithm's limitations, given that the dissimilarity function and the rule creation function are domain agnostic, could be overcome using functions that provide more musical information. However, from the point of view of the live coder, who observes the parameters while programming, it is possible that some readability is lost if, instead of frequencies, amplitudes, and cutoffs, one manipulates harmonic contents, MFCCs, etc.

Finally, a series of live coding performances and recordings have accompanied the design and testing of the algorithm. These have been developed in different contexts and venues including universities, artistic research centers, theatres, online streaming, smoky bars, etc. References regarding performances are available in Chapter 4, Section 4.3.4. Some of the *lists*, programs and places where the works are featured are also included there.

The community and public feedback have been extremely enriching, sparking conversations that range from the possibility of including various similarity measures (some of them less human-readable in the live coding context such as spectral), the limits of the algorithm or even symbolic learning vs subsymbolic approaches, to the consequences of real-time machine learning. All of these might draw no conclusive results, but this only reinforces the idea that this is an open field of research.

# Chapter 8

# Further Discussion

This Chapter extends the discussion presented in Chapter 7 with consideration of the comments received from the anonymous reviewers. Primarily, this chapter serves as an extension of the discussion, and further focuses the artistic possibilities of the algorithms presented and its designed criteria. Ideas including systems evaluation and references to recent performances using the RuLer and FuzzyRuLer algorithms are also included.

The Chapter is divided in two Section the first one discusses the design ideas, technical considerations and evaluation of the RuLer and FuzzyRuLer algorithms, the second discusses its artistic possibilities and contextualizes the research within the current machine learning applications within live coding.

## 8.1 Algorithm design considerations and evaluation

In (Knotts and Paz, 2021) a general discussion is presented regarding the implications resulting from using machine learning within live coding. At the outermost level of the discussion, we can observe that one facet of machine learning encompasses computer algorithms, which are able to learn through experience and by using data, with the primary aim of optimizing automation processes.

Live coding, is a creativity technique and a performative practice, centering on the expressive possibilities of algorithms. Therefore, with the integration of machine learning (ML) and live coding, some components are to be considered, such as: which elements to optimize; which processes to automate; what is the role of the performer; and how to present ML algorithms to an audience. These factors are regarded (whether implicitly or explicitly as discussed below) in accordance with the intended uses of the machine learning algorithms within live coding, which then actualize distinct aesthetic decisions, relationships with the algorithms and technical consequences. Therefore, most often the evaluation of the systems cannot provide information beyond the user tests. As an example, consider the report of the users experience of the Music Information Retrieval Live Coding Agent MIRLCa (Xambó, 2021), which is based on interviewing live coders after using the agent.

In the cases of the RuLer and FuzzyRuLer algorithms, the evaluation also relies on the subjective evaluation of the users within the user tests. It is directly associated with the musical experience (and expertise) of the individuals and can only be considered within that specific context. The Sawtooth and Pulse Waveforms were purposefully selected for the user tests, due to their clearly separated perceptual regions and few parameters. Having clearly separated perceptual regions and few parameters makes these architectures well-suited for a case study with "highly subjective results". The same ideas apply to the Blip and its well-studied perceptual regions. The questionnaire presented during the evaluations is available at Paz, 2017c.

### 8.1.1 Designing algorithms for real-time machine learning

When designing learning algorithms within live coding, an important element to consider is the technical restrictions of using machine learning within real-time. Real-time machine learning has technical implications relative to the size of the datasets given the computational complexity of the algorithms. For example, the training complexity of nonlinear Support Vector Machines, although depending on several factors, generally lies between $O(n^2)$ and $O(n^3)$, with $n$ being the amount of training instances (Abdiansah and Wardoyo, 2015). Also, some algorithms have many parameters which make it challenging to configure in real-time (even with default settings). For example, the Multi Layer Perceptron Classifier in Python (for a live coding system using sklearn library see Diapoulis, 2017) has 23 parameters. While it is not necessary to change all the parameters each time, without a proper mapping or expertise, the relationship between the parameter values and the aesthetic result is not immediately clear. In contrast, let us consider the Multi Layer Perceptron Classifier of Flucoma. The Flucoma (Tremblay et al., 2019) project, aims to provide "potent algorithms with a suitable level of modularity within the main coding environments used by the creative researchers". Flucoma's implementation of the FluidMLPClassifier has just 8 parameters to configure. Scaling back the parameter space allows the classifier more flexibility to run at performance time and reduces the complexity of mapping. Although Flucoma is oriented to audio analysis, its algorithm is designed with a workflow closer to creative tasks such as live coding performance. However, there still exists a time constraint of needing to find a proper mapping of the parameters providing the best affordances for creative use, which makes these algorithms difficult to use within a real-time context [1]. From this perspective, the proposed algorithms consider, first, that real-time rule learning has not been explored within live coding and further, second, that its symbolic nature helps to produce interpretable models. Moreover, the algorithms intentionally have only two parameters with which to facilitate its use during a performance. Finally, these parameters are intended to provide expressiveness by allowing the user to recover the exact data contained in the dataset v.s. something that is completely *wild* by changing a couple of parameters.

### 8.1.2 Oversampling instances for audio engines

Among the tasks in a live coding performance, one is to create new *instances* with specific characteristics, sometimes taking care of the consistency-novelty trade-off. This is similar to the definition of an oversampling problem that looks for data to adjust the

[1] Text included in Knotts and Paz, 2021.

class distribution of a data set.

To compare the RuLer with oversampling algorithms Smote and Adasyn were selected as they represent basic ideas from which further modifications/extensions have been proposed (see for example Tang et al., 2008; Wang et al., 2017). To oversample a dataset, the Smote algorithm takes an example and considers its $k$ nearest neighbors in the feature space. Then, to create the synthetic points, it takes the vector between one of the neighbors and the current point and multiplies it by a random number between 0 and 1. Adasyn is a modified version of Smote that uses a weighted distribution for different minority class examples according to their level of difficulty in learning, and generates more synthetic data for minority class examples, which are more challenging to learn. Modern algorithms follow the same principles. Some of my current research is comparing the RuLer results with modern Smote-derived algorithms, which has yielded similar results as those reported in Chapter Oversampling Tests. For example, the following 8.1 shows the 5-folds cross-validation over the glass-0-1-2-3_vs_4-5-6 dataset mentioned in Chapter 5, using oversampling algorithms RandomOverSampler, Smote, KMeansSMOTE, SVMSmote and BorderlineSMOTE with the classification algorithms KNN k = 2, 3 and 4, Randomforest and SVC. Nonetheless, this further research also takes into account that when considering creative pursuits accuracy does not necessarily imply better engagement, interaction or perceptual results.

Table 8.1: 5-folds cross validation with classification algorithms KNN k = 2, 3 and 4, Randomforest and SVC applied to glass-0-1-2-3_vs_4-5-6 data oversampled using algorithms Adasyn, RandomOverSampler, Smote, KMeansSMOTE, SVMSmote, RuLer and BorderlineSMOTE.

| **Oversampling Algorithm** | **Classification Algorithm** | **APS.mean** | **APS.sd** |
|---|---|---|---|
| Adasyn | knn_k=2 | 0.85 | 0.074 |
| Adasyn | knn_k=3 | 0.89 | 0.058 |
| Adasyn | knn_k=4 | 0.88 | 0.057 |
| Adasyn | rforest | 0.93 | 0.045 |

| | | | |
|---|---|---|---|
| Adasyn | svc | 0.92 | 0.020 |
| Baseline | knn_k=2 | 0.84 | 0.062 |
| Baseline | knn_k=3 | 0.87 | 0.073 |
| Baseline | knn_k=4 | 0.90 | 0.046 |
| Baseline | rforest | 0.91 | 0.035 |
| Baseline | svc | 0.92 | 0.019 |
| BorderlineSMOTE | knn_k=2 | 0.80 | 0.110 |
| BorderlineSMOTE | knn_k=3 | 0.84 | 0.098 |
| BorderlineSMOTE | knn_k=4 | 0.84 | 0.097 |
| BorderlineSMOTE | rforest | 0.92 | 0.079 |
| BorderlineSMOTE | svc | 0.92 | 0.023 |
| KMeansSMOTE | knn_k=2 | 0.81 | 0.093 |
| KMeansSMOTE | knn_k=3 | 0.85 | 0.059 |
| KMeansSMOTE | knn_k=4 | 0.86 | 0.046 |
| KMeansSMOTE | rforest | 0.91 | 0.073 |
| KMeansSMOTE | svc | 0.88 | 0.094 |
| RandomOverSampler | knn_k=2 | 0.82 | 0.084 |
| RandomOverSampler | knn_k=3 | 0.85 | 0.077 |
| RandomOverSampler | knn_k=4 | 0.86 | 0.070 |
| RandomOverSampler | rforest | 0.91 | 0.082 |
| RandomOverSampler | svc | 0.93 | 0.023 |
| Ruler | knn_k=2 | 0.99 | 0.004 |
| Ruler | knn_k=3 | 0.99 | 0.013 |
| Ruler | knn_k=4 | 0.98 | 0.014 |
| Ruler | rforest | 1 | 0 |
| Ruler | svc | 0.93 | 0.021 |
| Smote | knn_k=2 | 0.85 | 0.066 |
| Smote | knn_k=3 | 0.86 | 0.083 |

| | | | |
|---|---|---|---|
| Smote | knn_k=4 | 0.87 | 0.063 |
| Smote | rforest | 0.87 | 0.049 |
| Smote | svc | 0.93 | 0.023 |
| SVMSmote | knn_k=2 | 0.82 | 0.123 |
| SVMSmote | knn_k=3 | 0.88 | 0.065 |
| SVMSmote | knn_k=4 | 0.87 | 0.076 |
| SVMSmote | rforest | 0.94 | 0.035 |
| SVMSmote | svc | 0.91 | 0.038 |

There are two primary reasons to further extend the RuLer algorithm by using fuzzy sets. First, the RuLer algorithm creates only some points which are derived from the combinations of the existing data under the restrictions imposed by the algorithm parameter values, but it cannot infer beyond that. Using fuzzy rules allows the algorithm to produce unseen values (parameter values that are not present in the dataset) thus using the rules as information to infer new instances.

Furthermore, in music and sound, the perception of the material is strongly dependent on the context. Moreover, the perception can hardly be categorized as a two-class problem where something clearly belongs to one class or another. This can be understood from Figure 6.7 where the perceptual regions of the different persons overlap. These considerations lead to the FuzzyRuler algorithm presented in Chapter 6.

## 8.2 Artistic practice

Sound synthesis is the process of using electronic hardware or software to produce sound from scratch. There are several methods of sound synthesis, such as subtractive synthesis, additive synthesis, frequency modulation synthesis, phase distortion synthesis, wavetable synthesis, sample-based synthesis, vector synthesis, granular synthesis and physical modelling synthesis. These types of synthesis vary in the manner in which sounds are sourced, generated, and modulated. Live coding provides explicit constraints that push creativity in unexpected directions. This impacts all sorts of algorithms used, including sound synthesis. For example, Roma, 2016, describes a constrained environment aimed at exploring the creation and modification of sound synthesis and processing networks in real-time. These ideas can be traced back to those expressed in the timbre space conception of (Wessel, 1979) and (McAdams, 2013). The algorithms of this work were conceived to operate over Audio Engines (mainly sound synthesizers). Next, I discuss some general ideas about the different relationships that live coders have with the algorithms to contextualize the possibilities of the algorithms presented in this work.

### 8.2.1 Training v.s surprise trade-off

Anna Xambó's MIRCLa (Music Information Retrieval Live Coding Agent (Xambó, 2021), aims to explore big collections of sounds within live coding performance using machine learning, the Freesound.org database and music information retrieval algorithms. MIRLCa allows training a perceptron to recognize similar sounds from a set of given examples selected (using SuperCollider) during the training process. Once trained, MIRLCa retrieves similar sounds during the performance which are then processed or sequenced on-the-fly. Although the performers are able to control the level of training of the agent, which gives creative scope for the performers to explore aspects of surprise and risk, the agents can produce unpleasant surprises that the performer has to embrace and control on-the-fly as part of the performance. Therefore, there is not a 'perfect' training, and thus there is a trade-off between unexpected surprises and having 'predictable' models.

In contrast, it is possible to integrate algorithms pre-trained over large datasets into live coding systems. The trained algorithms are many times used for low level tasks such as in melody generation. SeMA (Bernardo et al., 2020), for example, is a playground for prototyping live coding mini-languages that integrates signal synthesis and ML. It facilitates the use of models from TensorFlow (Abadi et al., 2016), with live coded mapping of inputs and outputs. The training time of some models can take minutes, hours or days depending on the algorithm, the dataset, and the hardware. Once trained, the models can be fairly accurate, and are rarely surpassed by other systems. This approach reaches philosophical limits like the one suggested by Collins, 2016. An algorithm can be trained over a corpora of music that would take a human years to listen through.

### 8.2.2 How do we relate to models?

As mentioned at the beginning of this Chapter, how machine learning is used within live coding implies specific relationships with the algorithms, as well as aesthetic and technical decisions.

When it comes to using machine learning within live coding, machine learning models are used from different conceptual approaches. While most early live coding performances focused on low level tasks such as simple sound generators, processors, and pattern writing, today it is increasingly common to hear performances (see Villaseñor and Paz, 2020) using ML to perform specific tasks. Here I mention some iconic approaches where live coding and ML find the most confluence. These are identified in Knotts and Paz, 2021 by the artistic exploration they allow, however, this is by no means a comprehensive list of the current possibilities. Nonetheless, the categories discussed in Knotts and Paz, 2021 help to contextualize the algorithms presented in this work. Some of these systems are described in Section 2.4. The identified approaches are models used as: collaborators, autonomous systems, performative training, and Pre-labelled Data for Audio Engines.

#### Collaborators

When machine learning algorithms are used as collaborators, they automate some aspects (normally the repetitive ones) of live-coding, allowing the performer to engage with other elements such as the high-level conduction of the sound, without being overly concerned with minor details. In Conductive Bell, 2013, the performer acts as a conductor of the high-level decisions, such as rhythmic density, by turning automatic players on and off manually. In MIRLCa the live coder trains a virtual agent that can manage tasks, such as retrieving similar sounds, during the performance. MIRLCa researches agency and negotiability during the performance between the live coder, the agent and the audience. In Megra,the learned Markov processes act as autonomous collaborators which are controlled by the live coder during the performance.

### Autonomous live coding systems

Autonomous systems use machine learning algorithms to create models able to play autonomously, either in collaboration with human live coders or alone in front of an audience. Cibo (Stewart et al., 2020) is an autonomous performer that uses neural networks trained on sequential code blocks captured from TidalCycles humans performers. One trained, Cibo is able to perform a live coding set autonomously. Cacharpo (Navarro and Ogborn, 2017) is a neural net trained for automatic music making in the Cumbia Sondiera style. The autonomous performer uses Music Information Retrieval algorithms to listen to the human performer and generates code to produce complementary patterns and instruments.

### Training process as a performative practice

Live coding is about interacting with algorithms in real-time as a performative practice. Thus, despite the technical challenges, it offers opportunities to explore ML from a performative perspective. This approach is necessarily more focused on exploring the algorithm parameters and changing the mappings of the model, rather than in the optimization processes as occurs within the training processes executed offline. The latter approach is used in most of the applications of ML within musical contexts. Many live coding performances using ML use pre-trained models, however, there are approaches which explore the whole learning process (data collections, training and testing) as a performance, such as in Baalman's GeCoLa Baalman, 2020. GeCoLa presents a performative human-algorithm interaction. The algorithm is a neural net, trained to recognize physical movements and respond with specific sounds. The piece highlights the laborious, repetitive and error prone process of training the algorithms. The performance explores the creative possibilities that appear within the training process by exposing human workloads and real-time decision making.

### Pre-labelled Data for Audio Engines

For models that use this approach, the user collects data (typically a small dataset) and classifies it for a specific audio engine (the classes represent high-level descriptors of the data). The categories of the data are intended to allow the performer to conduct the sound through these high-level categories. Once trained, the algorithms are able to create real-time variations (for example tensions) with different degrees of similarity with respect to the original dataset.

Alo Allik's (Allik, n.d.) Evolver is a system based on evolutive algorithms, that allows the live recombination of preselected material. The system uses the Gene Expression Synthesis (Allik, 2014) to allow the performer to make informed selections of evolving outputs of sound synthesizers. According to Alo, one of the performance strategies involves selecting 2 chromosomes from the database to serve as source material for on-stage experiments.

Preselecting material balances the risk of exploring from scratch audio engines on stage, allowing the performer to conduct the sound by using high-level information. The RuLer and FuzzyRuLer algorithms use pre-labelled data for audio engines (specifically sound synthesizers), allowing to play with the expressive possibilities of exploring recombinations starting from a preselected dataset.

The sonic possibilities of the algorithm can be auditioned in the following references: ProxySpace (ProxySpace, 2021), a live coding session co-organized by MUTEK ES and the on-the-fly European project. TROBADA 3/4 (TOPLAP, 2020), organized by the TOPLAP community of Barcelona at the artistic research center Hangar. ALGOMA, a concert organized within the Gaudeamus Festival in Utrecht in collaboration with the live coding collective of the Netherlands **nl_cl** and Creative Coding Utrecht (Gaudeamus, 2021). The ideas of exploring parameter spaces used within the RuLer and FuzzyRuLer algorithms were applied within the design of the system for timbre recognition and real-time interaction used during the opening concert of the AI-Music festival held at the Auditori of Barcelona on October 2021 (AI-Music-Festival, 2021).

Additionally, the RuLer algorithm (and preliminary versions) have been integrated into several workshops held at the Artistic research center Hangar in Barcelona as well as the National Center of the Arts in Mexico city (Paz, 2015), where the participants experimented with the algorithms, collected small datasets auditioning the synthesizers and were are able to produce small performances, typically in the nine minutes from scratch style.

# Related Publications

**Journal**

1. Paz, I., Nebot, À., Mugica, F., & Romero, E. (2020). On-The-Fly Syntheziser Programming with Fuzzy Rule Learning. Entropy, 22(9), 969. https://doi.org/10.3390/e22090969, JCR Q2

2. Paz, I., Nebot, À., Mugica, F., & Romero, E. (2018). Modeling perceptual categories of parametric musical systems. Pattern Recognition Letters, 105, 217-225. https://doi.org/10.1016/j.patrec.2017.07.005, JCR Q2

**Conference**

1. Villaseñor, H,. Paz, I. (2020) "Live coding from scratch: the cases of practice in Mexico City and Barcelona" Proceedings of the 2020 International Conference on Live Coding (ICLC2020), University of Limerick, Limerick, Ireland, pp. 59-68. DOI:10.5281/zenodo.3939206 ISBN: 978-1-911620-23-5.

2. Paz, I., Nebot, À., Romero, E., & Mugica, F. (2019, June). Charting Perceptual Spaces with Fuzzy Rules, Proceedings of the 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), New Orleans, USA, 2019, pp. 1578-1583, DOI: 10.1109/FUZZ-IEEE.2019.8859008. ISBN: 978-1-5386-1728-1.

3. Paz, I., Roig, S. (2019). Tweaking Parameters, Charting Perceptual Spaces. Proceedings of the Fourth International Conference on Live Coding, Medialab Prado, Madrid, Spain, pp. 353. DOI: 10.5281/zenodo.3946281. ISBN: 978-84-18299-08-7.

4. Paz, I. (2019) cross-categorized-seeds. Proceedings of the Live Coding Music Seminar. Luiz Velho, Vitor Rolla, Eds.- 1. ed. Rio de Janeiro: IMPA, 38 p. ISBN 978-85-244-0466-5.

5. Paz, I., Nebot, À., Romero, E., Mugica, F., & Vellido, A. (2016, July). A methodological approach for algorithmic composition systems' parameter spaces aesthetic exploration. 2016 Proceedings of the IEEE World Congress on Evolutionary Computation, Vancouver B.C. pp. 1317-1323. DOI: 10.1109/CEC.2016.7743940. ISBN: 978-1-5090-0623-6.

6. Paz, I. (2016) MuSE: a geometric representation for multi-parameter spaces exploration. Interface Politics 1st International conference. pp. 201. ISBN: 978-84-617-5132-7.

**Book chapter**

1. Paz, I., Nebot, À., Mugica, F., & Romero, E. (2017). A fuzzy rule model for high level musical features on automated composition systems. In The Musical-Mathematical Mind (pp. 243-251). Springer, Cham.

2. , Paz, I., Nebot, À., Romero, E., & Mugica, F. (2016, September). A rule-extraction algorithm for describing perceptual parametric subspaces in algorithmic composition systems. Artificial Intelligence Research and Development: Frontiers in Artificial Intelligence and Applications (19th International Conference of the Catalan Association for Artificial Intelligence). Àngela Nebot, Xavier Binefa, Ramon López de Mántaras Eds. Volume: 288 pp: 213-220. DOI:/10.3233/978-1-61499-696-5-213. ISBN:/978-1-61499-695-8.

3. Paz, I., Mugica, F., Nebot., A., & Romero, E. (2016). Classifying and generalizing successful parameter combinations for sound design. Artificial Intelligence Research and Development - Current Challenges, New Trends and Applications, (CCIA) 2018, 21st International Conference of the Catalan Association

for Artificial Intelligence, Alt Empordà, Catalonia, Spain, 8-10th October 2018, Zoe Falomir, Karina Gibert, Enric Plaza Eds. Volume 308 pp. 256-265. DOI: 10.3233/978-1-61499-918-8-256. ISBN: 978-1-61499-917-1.

**Other (Digital/Visual Media)**

1. Album Visions of Space. Label Bohemian drips. Berlin Germany. Release: May 26th 2017. url = http://bohemiandrips.de/product/bd007-ivan-paz-visions-of-space/
   Visions of Space Release. Schrippe Hawaii, Neuköln, Berlin.
   url = https://youtu.be/sGf2nBJJx9g

**Press, critical reviews and performances**

1. TROBADA 3/4 Iván Paz + QBRNTSS. Hangar Artistic research center, Barcelona 2020.
   url = https://youtu.be/27ASshnNHOA?t=1764
2. a-Musik Radio w. Wolfgang Brauneis. 2019. Minute 59:00
   url = https://dublab.de/broadcast/a-musik-radio-wolfgang-brauneis-april-2019/
3. DubLab Ramón Cassamajó. 2019. Minute:48:48
   url = http://dublab.es/pargueland/programa-21
4. Zeppelin HyperExperimental Electroacoustic and Sound Art Festival. Centre de Cultura Contemporània de Barcelona. 2019.
   url = http://sonoscop.net/zeppelin2019/
   url = https://www.cccb.org/en/activities/file/zeppelin-2019-hyperexperimental/232026
5. Live Coding Music - Cross-categorized-seeds - Iván Paz 2019. Instituto de Matemática Pura e Aplicada. Brasil. Live Coding Music Seminar.
   url = https://youtu.be/zjTL0DOCNBo?list=PLo4jXE-LdDTQ44x3PYBvVJuzG2EJX_z8o

6. Bandcamp Daily. Simon Chandler. 2018. “Meet the Artists Using Coding, AI, and Machine Language to Make Music”.
   url = https://daily.bandcamp.com/2018/01/25/music-ai-coding-algorithms/

7. Opening concert IV International Conference on Live Coding. 2017. Centro Mexicano para la Música y las Artes Sonoras (CMMAS). Iván Paz & Ian Medina: Live Coding through Perceptual Spots on the Parametric Space.
   url = https://iclc.livecodenetwork.org/2017/en/schedule.html

**Journals in revision**

1. Paz, I. & Nebot, À. A rule-learning approach for minority oversampling. In process of submission to the Journal of Applied Soft Computing. ISSN: 1568-4946.

# Bibliography

Abadi, Martín et al. (2016). "Tensorflow: A system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283.

Abdiansah, Abdiansah and Retantyo Wardoyo (2015). "Time complexity analysis of support vector machines (SVM) in LibSVM". In: *International journal computer and application* 128.3, pp. 28–34.

AI-Music-Festival (2021). *AI-Music Festival.* `https://youtu.be/dHh4vP5T6VM`. Accessed: 2021-11-25.

Alcalá-Fdez, Jesús et al. (2009). "Learning the membership function contexts for mining fuzzy association rules by using genetic algorithms". In: *Fuzzy Sets and Systems* 160.7, pp. 905–921.

Allik, Alo (n.d.). *evolver: an audiovisual live coding performance.*

— (2014). "Gene expression synthesis". In: *ICMC.*

Ames, Charles (1987). "Automated composition in retrospect: 1956-1986". In: *Leonardo*, pp. 169–185.

Amin, Adnan et al. (2016). "Comparing oversampling techniques to handle the class imbalance problem: A customer churn prediction case study". In: *IEEE Access* 4, pp. 7940–7957.

Andrews, Robert, Joachim Diederich, and Alan B Tickle (1995). "Survey and critique of techniques for extracting rules from trained artificial neural networks". In: *Knowledge-based systems* 8.6, pp. 373–389.

Ariza, Christopher (2009). “The interrogator as critic: The turing test and the evaluation of generative music systems”. In: *Computer Music Journal* 33.2, pp. 48–70.

Baalman, Marije (2020). *the machine is learning, International Conference on Live Interfaces.*

Bagallo, Giulia and David Haussler (1990). “Boolean feature discovery in empirical learning”. In: *Machine learning* 5.1, pp. 71–99.

Basgall, María José et al. (2018). “SMOTE-BD: An exact and scalable oversampling method for imbalanced classification in big data”. In: *VI Jornadas de Cloud Computing & Big Data (JCC&BD)(La Plata, 2018).*

Bell, Renick (2013). “An approach to live algorithmic composition using conductive”. In: *Proceedings of LAC*. Vol. 2013.

— (Dec. 9, 2014). *Live coding screencast test in an experimental drum and bass algorave style.* URL: `https://youtu.be/d-_J2zLtbtg`.

Bencina, Ross (2005). “The metasurface: applying natural neighbour interpolation to two-to-many mapping”. In: *Proceedings of the 2005 conference on New interfaces for musical expression.* National University of Singapore, pp. 101–104.

Bernardo, Francisco, Chris Kiefer, and Thor Magnusson (n.d.). “Designing for a pluralist and user-friendly live code language ecosystem with Sema”. In: *International Conference on Live Coding.*

— (2020). “A Signal Engine for a Live Coding Language Ecosystem”. In: *Journal of the Audio Engineering Society* 68.10, pp. 756–766.

Berthold, Michael R (2003). “Mixed fuzzy rule formation”. In: *International journal of approximate reasoning* 32.2-3, pp. 67–84.

Berthold, Michael R and Klaus-Peter Huber (1999). “Constructing fuzzy graphs from examples”. In: *Intelligent Data Analysis* 3.1, pp. 37–53.

Berthold, Michael R, Bernd Wiswedel, and Thomas R Gabriel (2013). “Fuzzy logic in knime–modules for approximate reasoning–”. In: *International Journal of Computational Intelligence Systems* 6.sup1, pp. 34–45.

Bhalke, DG, CB Rama Rao, and Dattatraya S Bormane (2016). "Automatic musical instrument classification using fractional fourier transform based-MFCC features and counter propagation neural network". In: *Journal of Intelligent Information Systems* 46.3, pp. 425–446.

Blip (2019). *Blip*. `http://doc.sccode.org/Classes/Blip.html`. Accessed: 2019-06-23.

Bosque, Guillermo, Inés del Campo, and Javier Echanobe (2014). "Fuzzy systems, neural networks and neuro-fuzzy systems: A vision on their hardware implementation and platforms over two decades". In: *Engineering Applications of Artificial Intelligence* 32, pp. 283–331.

Brauneis, Wolfgang (2019). *a-Musik Radio w.* URL: `url=https://dublab.de/broadcast/a-musik-radio-wolfgang-brauneis-april-2019/`.

Breiman, Leo et al. (1984). *Classification and regression trees.* CRC press.

Briot, Jean-Pierre, Gaëtan Hadjeres, and François Pachet (2017). "Deep learning techniques for music generation-a survey". In: *arXiv preprint arXiv:1709.01620*.

Brown, Andrew R (2002). "Opportunities for evolutionary music composition". In.

Brown, Andrew R and Andrew Sorensen (2009). "Interacting with Generative Music through Live Coding". In: *Contemp. Music Rev.* 28.1, pp. 17–29.

Brown, Andrew R and Andrew C Sorensen (2007). "aa-cell in practice: An approach to musical live coding". In: *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 292–299.

Cassamajó, Ramón (2019). *DubLab Barcelona.* URL: `http://dublab.es/pargueland/programa-21`.

Castro, Félix, Àngela Nebot, and Francisco Mugica (2011). "On the extraction of decision support rules from fuzzy predictive models". In: *Appl. Soft Comput. J.* 11.4, pp. 3463–3475. ISSN: 15684946.

Cendrowska, Jadzia (1987). "PRISM: An algorithm for inducing modular rules". In: *International Journal of Man-Machine Studies* 27.4, pp. 349–370.

Cervone, Guido, Pasquale Franzese, and Allen PK Keesee (2010). "Algorithm quasi-optimal (AQ) learning". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.2, pp. 218–236.

Chakraborty, Debrup and Nikhil R Pal (2004). "A neuro-fuzzy scheme for simultaneous feature selection and fuzzy rule-based classification". In: *IEEE Transactions on Neural Networks* 15.1, pp. 110–123.

Chakraborty, Shruti Sarika and Ranjan Parekh (2018). "Improved Musical Instrument Classification Using Cepstral Coefficients and Neural Networks". In: *Methodologies and Application Issues of Contemporary Computing Framework*. Springer, pp. 123–138.

Chandler, Simon (2018). *Meet the Artists Using Coding, AI, and Machine Language to Make Music*. URL: `https://daily.bandcamp.com/2018/01/25/music-ai-coding-algorithms/`.

Chawla, Nitesh V et al. (2002). "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16, pp. 321–357.

Chorowski, Jan and Jacek M Zurada (2011). "Extracting rules from neural networks as decision diagrams". In: *IEEE Transactions on Neural Networks* 22.12, pp. 2435–2446.

Clark, Peter and Robin Boswell (1991). "Rule induction with CN2: Some recent improvements". In: *European Working Session on Learning*. Springer, pp. 151–163.

Clark, Peter and Tim Niblett (1989). "The CN2 induction algorithm". In: *Machine learning* 3.4, pp. 261–283.

Cohen, William W (1995). "Fast effective rule induction". In: *Machine learning proceedings 1995*. Elsevier, pp. 115–123.

Collins, Nick (2001). "Algorithmic composition methods for breakbeat science". In: *Proceedings of Music Without Walls*, pp. 21–23.

— (2002a). "Experiments with a new customisable interactive evolution framework". In: *Organised Sound* 7.3, p. 267.

— (2002b). “Interactive evolution of breakbeat cut sequences”. In: *Proc. Cybersonics.* Ed. by Institute of Contemporary Arts.

— (2002c). “The BBCut Library.” In: *ICMC.* Citeseer.

— (2003). “Generative music and laptop performance”. In: *Contemporary Music Review* 22.4, pp. 67–79.

— (2010). *Introduction to computer music.* John Wiley & Sons.

— (2011a). “Implementing stochastic synthesis for SuperCollider and iPhone”. In: *Proc. Xenakis International Symposium.*

— (2011b). “Live coding of consequence”. In: *Leonardo* 44.3, pp. 207–211.

Collins, Nick (2011c). “SCMIR: A SuperCollider music information retrieval library”. In: *ICMC.*

— (2012). “Automatic composition of electroacoustic art music utilizing machine listening”. In: *Computer Music Journal* 36.3, pp. 8–23.

— (2015). “Live Coding and Machine Listening”. In: *Proceedings of the International Conference on Live Coding*, pp. 4–11.

— (2016). “Towards machine musicians who have listened to more music than us: Audio database-led algorithmic criticism for automatic composition and live concert systems”. In: *Computers in Entertainment (CIE)* 14.3, pp. 1–14.

— (2018). “Origins of Algorithmic Thinking in Music”. In: *The Oxford Handbook of Algorithmic Music*, p. 67.

Collins, Nick and Julio d’Escriván (2017). *The Cambridge companion to electronic music.* Cambridge University Press.

Collins, Nick and Shelly Knotts (2019). “A Javascript Musical Machine Listening Library”. In: Michigan Publishing.

Collins, Nick et al. (2003). “Live coding in laptop performance”. In: *Organised sound* 8.3, pp. 321–330.

Cortez, Paulo et al. (2009). “Modeling wine preferences by data mining from physicochemical properties”. In: *Decision Support Systems* 47.4, pp. 547–553.

Craven, Mark and Jude W Shavlik (1996). “Extracting tree-structured representations of trained networks”. In: *Advances in neural information processing systems*, pp. 24–30.

Craven, Mark W and Jude W Shavlik (1994). “Using sampling and queries to extract rules from trained neural networks”. In: *Machine learning proceedings 1994*. Elsevier, pp. 37–45.

Dahlstedt, Palle (2001a). “A MutaSynth in parameter space: interactive composition through evolution”. In: *Organised Sound* 6.2, pp. 121–124.

— (2001b). “Creating and Exploring Huge Parameter Spaces: Interactive Evolution as a Tool for Sound Generation.” In: *ICMC*.

Dahlstedt, Palle (2007). “Evolutionary Computer Music”. In: ed. by Biles J.A. (eds) Miranda E.R. Springer, London. Chap. Evolution in Creative Sound Design, pp. 79–99.

— (2009). “Thoughts on creative evolution: A meta-generative approach to composition”. In: *Contemporary Music Review* 28.1, pp. 43–55.

Davis, Steven and Paul Mermelstein (1980). “Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences”. In: *IEEE transactions on acoustics, speech, and signalprocessing* 28.4, pp. 357–366.

Dawkins, Richard (1986). *The blind watchmaker: Why the evidence of evolution reveals a universe without design*. WW Norton & Company.

De Mantaras, Ramon Lopez and Josep Lluis Arcos (2002). “AI and music: From composition to expressive performance”. In: *AI magazine* 23.3, pp. 43–43.

De Raedt, Luc (2008). *Logical and relational learning*. Springer Science & Business Media.

Dean, R T (2009). *The Oxford Handbook of Computer Music*. Oxford Handbooks. Oxford University Press. ISBN: 9780199887132.

Dean, Roger T and Alex McLean (2018). *The Oxford Handbook of Algorithmic Music*. Oxford University Press.

Dexed (2019). *Dexter*. `https://github.com/asb2m10/dexed`. Accessed: 02-02-2020.

Diapoulis, Georgios (2017). *Live coding using SC3 and scikit-learn.* URL: `http://gewhere.github.io/blog/2017/10/13/live-coding-using-sc3-and-scikit-learn/`.

Ding, Shifei et al. (2015). "Deep extreme learning machine and its application in EEG classification". In: *Mathematical Problems in Engineering* 2015.

Doornbusch, Paul (2017). "Early Computer Music Experiments in Australia and England". In: *Organised Sound* 22.2, pp. 297–307.

Dua, Dheeru and Casey Graff (2017). *UCI Machine Learning Repository.* URL: `http://archive.ics.uci.edu/ml`.

Duch, Wlodzislaw, Rudy Setiono, and Jacek M Zurada (2004). "Computational intelligence methods for rule-based data understanding". In: *Proceedings of the IEEE* 92.5, pp. 771–805.

Eco, Umberto (2017). "The Ars Magna by Ramon Llull". In: *Contributions to science*, pp. 47–50.

Engel J., Resnick C. Roberts A. Dieleman S. Eck D. Simonyan K. and M Norouzi (2017). *Neural audio synthesis of musical notes with wavenet autoencoders.* URL: `https://arxiv.org/pdf/1704.01279.pdf`.

Escobet, Antoni, Àngela Nebot, and François E Cellier (2008). "Visual-FIR: A tool for model identification and prediction of dynamical complex systems". In: *Simulation Modelling Practice and Theory* 16.1, pp. 76–92.

Escobet Canal, Antoni et al. (2015). "Visual-FIR". In.

Esling, Philippe, Adrien Bitton, et al. (2018). "Generative timbre spaces: regularizing variational auto-encoders with perceptual metrics". In: *arXiv preprint arXiv:1805.08501.*

Esling, Philippe et al. (2019). "Universal audio synthesizer control with normalizing flows". In: *arXiv preprint arXiv:1907.00971.*

Estrada, Julio and Jorge Gil (1984). *Música y teoría de grupos finitos (3 variables booleanas).* 519.4 EST.

Fernández, Jose D and Francisco Vico (2013). “AI methods in algorithmic composition: A comprehensive survey”. In: *Journal of Artificial Intelligence Research* 48, pp. 513–582.

Fiebrink, Rebecca and Baptiste Caramiaux (n.d.). “The machine learning algorithm as creative musical tool, November 2016”. In: *arXiv preprint arXiv:1611.00379* ().

Fiebrink, Rebecca et al. (2016). *The machine learning algorithm as creative musical tool*. Oxford University Press.

Finley, Sarah E (2014). “Acoustic Epistemologies and Aurality in Sor Juana Inés de la Cruz”. In.

Fisher, Ronald A (1936). “The use of multiple measurements in taxonomic problems”. In: *Annals of eugenics* 7.2, pp. 179–188.

Friedman, Jerome H and Nicholas I Fisher (1999). “Bump hunting in high-dimensional data”. In: *Statistics and Computing* 9.2, pp. 123–143.

Fürnkranz, Johannes, Dragan Gamberger, and Nada Lavrač (2012). *Foundations of rule learning*. Springer Science & Business Media.

Gaudeamus, Festival (2021). *Gaudeamus Festival Utrecht 2021 organized by on-the-fly Project in Collaboration with the Live coding community of the Netherlands.* `https://youtu.be/KHpEvDZoqZc?list=PLMBIpibV-wQLKi8gwYosuz1K5lWEPbzoa`. Accessed: 2021-11-25.

Géczy, Peter and Shiro Usui (1999). “Rule Extraction from Trained Artifical Neural Networks”. In: *Behaviormetrika* 26.1, pp. 89–106.

Goldin, Dina, Scott A Smolka, and Peter Wegner (2006). *Interactive computation*. Springer.

Goldmann, S (2015). *Presets - Digital shortcuts to sound*. The Bookworm, an imprint of The Tapeworm.

Hailesilassie, Tameru (2016). “Rule extraction algorithm for deep neural networks: A review”. In: *arXiv preprint arXiv:1610.05267*.

He, Haibo et al. (2008). “ADASYN: Adaptive synthetic sampling approach for imbalanced learning”. In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. IEEE, pp. 1322–1328.

Hernández-Espinosa, Carlos, Mercedes Fernández-Redondo, and Mamen Ortiz-Gómez (2003). “Inversion of a neural network via interval arithmetic for rule extraction”. In: *Artificial Neural Networks and Neural Information Processing—ICANN/ICONIP 2003*. Springer, pp. 670–677.

Herremans, Dorien, Ching-Hua Chuan, and Elaine Chew (2017). “A Functional Taxonomy of Music Generation Systems”. In: *ACM Computing Surveys (CSUR)* 50.5, p. 69.

Holland, Simon (2000). *Artificial Intelligence in Music Education: A Critical Review.*

Huang, Cheng-Zhi Anna et al. (2019). “The Bach Doodle: Approachable music composition with machine learning at scale”. In: *arXiv preprint arXiv:1907.06637.*

Jang, J-SR (1993). “ANFIS: adaptive-network-based fuzzy inference system”. In: *IEEE transactions on systems, man, and cybernetics* 23.3, pp. 665–685.

Jensen, Craig A et al. (1999). “Inversion of feedforward neural networks: algorithms and applications”. In: *Proceedings of the IEEE* 87.9, pp. 1536–1549.

Johnson, Colin G (2003). “Exploring sound-space with interactive genetic algorithms”. In: *Leonardo* 36.1, pp. 51–54.

Karaboga, Dervis and Ebubekir Kaya (2019). “Adaptive network based fuzzy inference system (ANFIS) training approaches: a comprehensive survey”. In: *Artificial Intelligence Review* 52.4, pp. 2263–2293.

Keel (2010). *Knowledge extraction based on evolutionary learning.* URL: `https://sci2s.ugr.es/keel/imbalanced.php?order=irR#sub60`.

Keislar, Douglas (2009). *A historical view of computer music technology.* na.

Khan, Mohammad Mahmudur Rahman et al. (2018). “Study and observation of the variation of accuracies of KNN, SVM, LMNN, ENN algorithms on eleven different datasets from UCI machine learning repository”. In: *2018 4th International Con-*

*ference on Electrical Engineering and Information & Communication Technology (iCEEiCT)*. IEEE, pp. 124–129.

Kiefer, Chris and Thor Magnusson (2019). "Live coding machine learning and machine listening: a survey on the design of languages and environments for live coding". In: *Proceedings of the International Conference on Live Coding, Media Lab, Madrid, Espagne*.

Kingma, Diederik P and Max Welling (2013). "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114*.

Knotts, Shelly (2019). "Interactively evolving compositional sound synthesis networks". In: *Proceedings of the Live Coding Music Seminar*. IMPA, pp. 29–31.

Knotts, Shelly (Dec 2017). *CYOF*. URL: `https://vimeo.com/264561088`.

Knotts, Shelly and Iván Paz (2021). *Live Coding and Machine Learning is Dangerous: Show us your Algorithms*. In revision 2021 International Conference on Live Coding.

Landy, Leigh (2009). *Sound-based music 4 all*. Oxford University Press.

Lavangnananda, K and S Chattanachot (2017). "Study of discretization methods in classification". In: *2017 9th International Conference on Knowledge and Smart Technology (KST)*. IEEE, pp. 50–55.

Loy Gareth, Vol (2007). *2: The Mathematical Foundations of Music*.

Luque, Sergio (2009). "The stochastic synthesis of iannis xenakis". In: *Leonardo Music Journal*, pp. 77–84.

Macret, Matthieu and Philippe Pasquier (2014). "Automatic design of sound synthesizers as pure data patches using coevolutionary mixed-typed cartesian genetic programming". In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 309–316.

Magnusson, Thor (2011). "Algorithms as scores: Coding live music". In: *Leonardo Music Journal*, pp. 19–23.

— (2015). "Herding cats: Observing live coding in the wild". In: *Comput. Music J.* 38.1, pp. 8–16. ISSN: 1531-5169.

— (2019). *Sonic writing: technologies of material, symbolic, and signal inscriptions.* Bloomsbury Academic.

Manzo, VJ and Will Kuhn (2015). *Interactive composition: Strategies using Ableton live and max for live.* Oxford University Press, USA.

Marier, Martin (2012). "Designing Mappings for Musical Interfaces Using Preset Interpolation." In: *NIME*.

McAdams, Stephen (2013). "Musical timbre perception". In: *The psychology of music*, pp. 35–67.

McCartney, J. (1996). "SuperCollider: a New Real Time Synthesis Language". In: *Proc. Int. Comput. Music Conf.* International Computer Music Association, pp. 257–258.

McLean, Alex and Roger T Dean (2018). *The Oxford handbook of algorithmic music.* Oxford University Press.

McLean, Alex and Geraint Wiggins (2010a). "Tidal–pattern language for the live coding of music". In: *Proceedings of the 7th sound and music computing conference.*

McLean, Alex and Geraint A Wiggins (2010b). "Live Coding Towards Computational Creativity." In: *ICCC*, pp. 175–179.

McLean, Christopher Alex (2011). "Artist-programmers and programming languages for the arts". PhD thesis. Goldsmiths, University of London.

Mease, David and Abraham Wyner (2008). "Evidence contrary to the statistical view of boosting". In: *Journal of Machine Learning Research* 9.Feb, pp. 131–156.

Michalski, RS (n.d.). "On the quasi-optimal solution of the general covering problem". In: *Proceedings of the V International Symposium on Information Processing (FCIP 69)*, pp. 125–128.

MIMIC (2019a). *MIMIC Artist Summer Workshop in Brighton.* URL: `http://www.emutelab.org/blog/summerworkshop`.

— (2019b). *Sema.* URL: `https://github.com/mimic-sussex/sema`.

Miranda, Eduardo Reck (2013). *Readings in music and artificial intelligence.* Routledge.

Miranian, Arash and Majid Abdollahzade (2012). "Developing a local least-squares support vector machines-based neuro-fuzzy model for nonlinear and chaotic time series prediction". In: *IEEE Transactions on Neural Networks and Learning Systems* 24.2, pp. 207–218.

Mitchell, Thomas (2012). "Automated evolutionary synthesis matching". In: *Soft Computing* 16.12, pp. 2057–2070.

Mitchell, Thomas J (2010). "An exploration of evolutionary computation applied to frequency modulation audio synthesis parameter optimisation". PhD thesis. University of the West of England.

Mitchell, Tom M (1982). "Generalization as search". In: *Artificial intelligence* 18.2, pp. 203–226.

Mordeson, John N and Premchand S Nair (2012). *Fuzzy graphs and fuzzy hypergraphs*. Vol. 46. Physica.

Mudd, Tom et al. (2015). "Investigating the effects of introducing nonlinear dynamical processes into digital musical interfaces". In.

Mugica, Francisco et al. (2015). "A Fuzzy Inductive approach for rule-based modelling of high level structures in algorithmic composition systems". In: *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE, pp. 1–8.

Murdoch, W James and Arthur Szlam (2017). "Automatic rule extraction from long short term memory networks". In: *arXiv preprint arXiv:1702.02540*.

Navarro, Luis and David Ogborn (2017). "Cacharpo: Co-performing Cumbia Sonidera with Deep Abstractions". In: *Proceedings of the International Conference on Live Coding*.

Nierhaus, G (2009). *Algorithmic composition: paradigms of automated music generation*. Springer Science & Business Media.

Ocelotl, Emilio (2016). *Altamisa*. URL: `http://andamio.in/prod/altamisa`.

Olofsson, Fredrik (2015). *Chain Reaction*. URL: `https://youtu.be/qbyLWpXKog8`.

OP-1 (2020). *OP-1 Synthesizer*. URL: `https://en.wikipedia.org/wiki/Teenage_Engineering_OP-1`.

Pachet, François (2008). "The future of content is in ourselves." In: *Computers in Entertainment* 6.3, pp. 31–1.

Pareyon, Gabriel (2011). *On Musical Self-Similarity: Intersemiosis as Synecdoche and Analogy*. Gabriel Pareyon.

Pareyon, Gabriel et al. (2017). *The Musical-Mathematical Mind: Patterns and Transformations*. Springer.

Paz, I (2017a). *Tiempos de Aguacero*. URL: `https://bohemiandrips.bandcamp.com/track/a2-tiempos-de-aguacero`.

— (2018). *Examples using RuLer*. URL: `https://soundcloud.com/automated-composition/sets/experiments-with-a-rule-learning-algorithm`.

— (2019a). *cross-categorized-seeds Live coding music seminar. Institute for pure and applied mathematics. Rio de Janeiro, Brasil.* URL: `https://youtu.be/zjTLODOCNBo`.

Paz, I et al. (2017). "Modeling perceptual categories of parametric musical systems". In: *Pattern Recognition Letters*.

— (2019). "Generalyzing successful parameter combinations". In revision.

Paz, Iván (2017b). *En Casa*. URL: `https://bohemiandrips.bandcamp.com/track/b2-en-casa`.

Paz, Ivan (2019b). "cross-categoryzed-seeds". In: *Proceedings of the Live Coding Music Seminar*. IMPA, pp. 12–15.

Paz, Iv'an (2019c). *Live Code Music Seminal*. URL: `http://w3.impa.br/~vitorgr/livecode2019/conference.html`.

— (2020). *EulerRoom Equinox*. URL: `https://youtu.be/xhvYl4__u8I?t=8966`.

Paz, Iván et al. (2016). "A methodological approach for algorithmic composition systems' parameter spaces aesthetic exploration". In: *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, pp. 1317–1323.

Paz, Iván et al. (2020). "On-The-Fly Syntheziser Programming with Fuzzy Rule Learning". In: *Entropy* 22.9, p. 969.

Paz, Iván (16th July, 2020). *Paramix Network Music Festival*. URL: `https://networkmusicfestival.org/programme/performances/ivan-paz-paramix/`.

— (2015). *Construcción de algoritmos para sistemas de composición e improvisación. Centro Nacional de las Artes Ciudad de México Taller de Audio.* http://cmm.cenart.gob.mx/cursos/anteriores.html. Accessed: 2021-11-25.

— (2016). *Multiparametric representation space as a perceptual exploration interface.* URL: https://www.gridspinoza.net/en/projects/multiparametric-representation-space-perceptual-exploration-interface.

— (2017c). *Parametric Perceptual Exploration.* https://github.com/ivan-paz/parametric-perceptual-exploration.

Paz, Iván (April 3, 2020). *8:08pm La hora del LIVECODER.* URL: https://youtu.be/wPVwCAXsnvU?list=PLDvUWb-gizILT4qFLpRpd-kRgfoZUfzZt.

Paz, Iván (2017a). *Visions of Space.* https://bohemiandrips.bandcamp.com/album/visions-of-space. Accessed: 2020-06-26.

— (2017b). *Visions of Space release.* https://youtu.be/sGf2nBJJx9g. Accessed: 2020-11-5.

Pearce, M, D Meredith, and G Wiggins (2002). "Motivations and Methodologies for Automation of the Compositional Process". In: *Music. Sci.* 6, pp. 119–147.

Pillai, GN, Jagtap Pushpak, and M Germin Nisha (2014). "Extreme learning ANFIS for control applications". In: *2014 IEEE Symposium on Computational Intelligence in Control and Automation (CICA).* IEEE, pp. 1–8.

Pratihar, Dilip Kumar and Bitan Pratihar (2017). "A review on applications of soft computing in design and development of intelligent autonomous robots". In: *International Journal of Hybrid Intelligent Systems* 14.1-2, pp. 49–65.

Pritchett, James (1996). *The Music of John Cage.* Vol. 5. Cambridge University Press.

Priyanka and Dharmender Kumar (2020). "Decision tree classifier: a detailed survey". In: *International Journal of Information and Decision Sciences* 12.3, pp. 246–269.

ProxySpace (2021). *Iván Paz & Julia Múgica - Live at "Proxyspace" organized by Mutek and on-the-fly.* https://youtu.be/FXKoRDZu8pY?list=PLMBIpibV-wQLKi8gwYosuz1K5lWEPbzoa. Accessed: 2021-11-25.

Quinlan, J. Ross (1986). “Induction of decision trees”. In: *Machine learning* 1.1, pp. 81–106.

— (1990). “Learning logical definitions from relations”. In: *Machine learning* 5.3, pp. 239–266.

Quinlan, J Ross (2014). *C4.5: programs for machine learning*. Elsevier.

Reppel, Niklas (n.d.). *Megra*.

— (2020). *Megra International Conference on Live Coding*. URL: `https://youtu.be/6dhvNrwQTRU?t=3839`.

Rivest, Ronald L (1987). “Learning decision lists”. In: *Machine learning* 2.3, pp. 229–246.

Roads, C (2001). “Sound Composition with Pulsars”. In: *J. Audio Eng. Soc* 49, pp. 134–147.

Roads, Curtis (1985). “Research in music and artificial intelligence”. In: *ACM Computing Surveys (CSUR)* 17.2, pp. 163–190.

Roads, Curtis, John Strawn, et al. (1996). *The computer music tutorial*. MIT press.

Roberts, Adam, Curtis Hawthorne, and Ian Simon (2018). “Magenta. js: A javascript api for augmenting creativity with deep learning”. In.

Rohrhuber, Julian, Alberto de Campo, and Renate Wieser (2005). “Algorithms today notes on language design for just in time programming”. In: *International Computer Music Conference*, p. 291.

Rohrhuber, Julian and Alberto De Campo (2009). “Improvising Formalisation: Conversational Programming and Live Coding”. In: *New Computational Paradigms for Computer Music. Delatour France/Ircam-Centre Pompidou.*

Rokach, Lior and Oded Z Maimon (2008). *Data mining with decision trees: theory and applications*. Vol. 69. World scientific.

Roma, Gerard (2016). ““Colliding: a supercollider environment for synthesis-oriented live coding”. In: *Proceedings of the 2016 International Conference on Live Interfaces.*

Ron, Dana, Yoram Singer, and Naftali Tishby (1996). “The power of amnesia: Learning probabilistic automata with variable memory length”. In: *Machine learning* 25.2-3, pp. 117–149.

Roth, Martin and Matthew Yee-King (2011). “A comparison of parametric optimization techniques for musical instrument tone matching”. In: *Audio Engineering Society Convention 130*. Audio Engineering Society.

Schroeder, Manfred R (1962). “Natural sounding artificial reverberation”. In: *Journal of the Audio Engineering Society* 10.3, pp. 219–223.

Schroeder, Manfred R (1970). “Digital simulation of sound transmission in reverberant spaces”. In: *The Journal of the acoustical society of america* 47.2A, pp. 424–431.

Schwarz, Diemo (2007). “Corpus-based concatenative synthesis”. In: *IEEE signal processing magazine* 24.2, pp. 92–104.

Serra, Xavier and Julius Smith (1990). “Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition”. In: *Computer Music Journal* 14.4, pp. 12–24.

Shihabudheen, KV, M Mahesh, and GN Pillai (2018). “Particle swarm optimization based extreme learning neuro-fuzzy system for regression and classification”. In: *Expert Systems with Applications* 92, pp. 474–484.

Shihabudheen, KV and Gopinatha N Pillai (2018). “Recent advances in neuro-fuzzy system: A survey”. In: *Knowledge-Based Systems* 152, pp. 136–162.

Sibilia, Paula (2012). *El hombre postorgánico: cuerpo, subjetividad y tecnologías digitales*. Fondo de cultura económica.

Sicchio, Kate (2014). “Hacking choreography: Dance and live coding”. In: *Computer Music Journal* 38.1, pp. 31–39.

Smilkov, Daniel et al. (2019). “Tensorflow. js: Machine learning for the web and beyond”. In: *arXiv preprint arXiv:1901.05350*.

Sorensen, Andrew (2019). *Study in Keith*. URL: `https://en.wikipedia.org/wiki/File:Study_in_keith.ogv`.

Sorensen, Andrew and Henry Gardner (2017). “Systems level liveness with extempore”. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 214–228.

Sorensen, Andrew, Ben Swift, and Alistair Riddell (2014). “The many meanings of live coding”. In: *Computer Music Journal* 38.1, pp. 65–76.

Soria, Edmar and Roberto Morales-Manzanares (2013). “Multidimensional sound spatialization by means of chaotic dynamical systems.” In: *NIME*. Vol. 13, pp. 79–83.

Stewart, Jeremy and Shawn Lawson (n.d.). “Cibo: An Autonomous TidalCyles Performer”. In: ().

Stewart, Jeremy et al. (2020). “Cibo v2: Realtime Livecoding AI Agent”. In: *Proceedings of the 2020 International Conference on Live Coding (ICLC2020)*, pp. 20–31.

Stoll, Thomas M (2014). “Genomic: evolving sound treatments using genetic algorithms”. In: *International Conference on Evolutionary and Biologically Inspired Music and Art*. Springer, pp. 107–118.

Strange, Allen and Gordon Mumma (1972). *Electronic music: systems, techniques, and controls*. WC Brown Company.

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

Sturm, Bob L et al. (2019). “Machine learning research that matters for music creation: A case study”. In: *Journal of New Music Research* 48.1, pp. 36–55.

Sugiyama, Masashi (2015). *Introduction to statistical machine learning*. Morgan Kaufmann.

Tang, Yuchun et al. (2008). “SVMs modeling for highly imbalanced classification”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39.1, pp. 281–288.

Tatar, Kıvanç, Daniel Bisig, and Philippe Pasquier (2020). “Introducing Latent Timbre Synthesis”. In: *arXiv preprint arXiv:2006.00408*.

Tatar, Kıvanç, Matthieu Macret, and Philippe Pasquier (2016). "Automatic synthesizer preset generation with PresetGen". In: *Journal of New Music Research* 45.2, pp. 124–144.

Thrun, Sebastian (1993). *Extracting provably correct rules from artificial neural networks.* Citeseer.

Tickle, Alan B, Marian Orlowski, and Joachim Diederich (1994). "DEDEC: decision detection by rule extraction from neural networks". In: *QUT NRC*.

TOPLAP (2004a). *Temporal Organization for the Promotion of Art Programming.* URL: `https://toplap.org`.

— (2004b). *TOPLAP Manifesto Draft.* URL: `https://toplap.org/wiki/ManifestoDraft`.

TOPLAP, Barcelona (2020). *TROBADA 3/4 Iván Paz + QBRNTSS.* `https://www.youtube.com/watch?v=27ASshnNHOA\&t=2024s`. Accessed: 2021-11-25.

Toussaint, Godfried T (2004). "A Comparison of Rhythmic Similarity Measures." In: *ISMIR*.

Towell, Geoffrey G, Jude W Shavlik, and Michiel O Noordewier (1990). "Refinement of approximate domain theories by knowledge-based neural networks". In: *Proceedings of the eighth National conference on Artificial intelligence.* Vol. 861866. Boston, MA.

Tremblay, Pierre Alexandre et al. (2019). "From collections to corpora: Exploring sounds through fluid decomposition". In: *International Computer Music Conference and New York City Electroacoustic Music Festival.* International Computer Music Association, pp. 223–228.

Vercoe, Barry (1986). *Csound: A manual for the audio processing system and supporting programs with tutorials.* Massachusetts Institute of Technology.

Villaseñor, Hernani and Iván Paz (2020). ""Live coding from scratch: the cases of practice in Mexico City and Barcelona"". In: *V International Conference on Live Coding.*

Wang, Qi et al. (2017). "A novel ensemble method for imbalanced data learning: bagging of extrapolation-SMOTE SVM". In: *Computational intelligence and neuroscience* 2017.

Wei, Jianan et al. (2020). "NI-MWMOTE: An Improving Noise-immunity Majority Weighted Minority Oversampling Technique for Imbalanced Classification Problems". In: *Expert Systems with Applications*, p. 113504.

Wessel, David L (1979). "Timbre space as a musical control structure". In: *Computer music journal*, pp. 45–52.

Wishart, Trevor (1986). "Sound symbols and landscapes". In: *The language of electroacoustic music*. Springer, pp. 41–60.

Witten, Ian H and Eibe Frank (2005). "Data Mining: Practical machine learning tools and techniques 2nd edition". In: *Morgan Kaufmann, San Francisco*.

Xambó, Anna (2021). *Music Information Retrieval Live Coding Agent*. URL: `https://mirlca.dmu.ac.uk/`.

Xenakis, Iannis, Roberta Brown, and John Rahn (1987). "Xenakis on Xenakis". In: *Perspectives of New Music*, pp. 16–63.

Yee-King and Peters (2011). *Livecoding SuperCollider and alto flute 22nd March 2011*. URL: `https://youtu.be/H5IceNlC69w`.

Yee-King, Matthew and Martin Roth (2008). "Synthbot: an Unsupervised Software synthesizer Programmer." In: *ICMC*.

Yee-King, Matthew John (2011a). "An autonomous timbre matching improviser". In: *ICMC*.

— (2011b). "Automatic sound synthesizer programming: techniques and applications". PhD thesis. University of Sussex.

Yee-King, Matthew John, Leon Fedden, and Mark d'Inverno (2018). "Automatic Programming of VST Sound Synthesizers Using Deep Networks and Other Techniques". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2.2, pp. 150–159.

Zadeh, Lotfi A (1965). "Fuzzy sets". In: *Information and control* 8.3, pp. 338–353.

— (1999). "Fuzzy logic and the calculi of fuzzy rules, fuzzy graphs, and fuzzy probabilities". In: *Computers & Mathematics with Applications* 37.11-12, p. 35.

Zbyszynski, Michael, Mick Grierson, Matthew Yee-King, et al. (2017). "Rapid prototyping of new instruments with codecircle". In.

Zeppelin, Festival (2019). *on-the-fly Live Coding Concert; Festival Zeppelin 2019 HIPER-EXPERIMENTAL*. `https://youtu.be/sGf2nBJJx9g`. Accessed: 2020-11-5.

Zheng, Zhuoyuan, Yunpeng Cai, and Ye Li (2016). "Oversampling method for imbalanced classification". In: *Computing and Informatics* 34.5, pp. 1017–1037.