Algorithmic Composition of Classical Music through Data Mining

An All College Thesis

College of Saint Benedict and Saint John's University

by

Tom Donald Richmond

April, 2018

Algorithmic Composition of Classical Music through Data Mining

By Tom Donald Richmond

Approved By:

_____

Dr. Imad Rahal, Professor of Computer Science

_____

Dr. Mike Heroux, Scientist-in-Residence of Computer Science

_____

Dr. Jeremy Iverson, Assistant Professor of Computer Science

_____

Dr. Imad Rahal, Chair, Computer Science Department

_____

Molly Ewing, Director, All-College Thesis Program

_____

Jim Parsons, Director, All-College Thesis Program

Abstract

The desire to teach a computer how to algorithmically compose music has been a topic in the world of computer science since the 1950's, with roots of computer-less algorithmic composition dating back to Mozart himself. One limitation of algorithmically composing music has been the difficulty of eliminating the human intervention required to achieve a musically homogenous composition. We attempt to remedy this issue by teaching a computer how the rules of composition differ between the six distinct eras of classical music by having it examine a dataset of musical scores, rather than explicitly telling the computer the formal rules of composition. To pursue this an automated composition process, we examined the intersectionality of algorithmic composition with the machine learning concept of classification. Using a Naïve Bayes classifier system, the computer classifies pieces of classical music into their respective era based upon a number of attributes. It attempts to recreate each of the six classical styles using a technique inspired by cellular automata. The success of this process is twofold determined by feeding composition samples into a number of classifiers, as well as analysis by studied musicians. We concluded that there is potential for further hybridization of classification and composition techniques.

Table of Contents

List of Figures

1. Introduction

Of all major art forms, music has historically relied most upon scientific and mathematical devices in its creation. While many other forms of art are lauded for breaking the rules, and these avant-garde approaches often find themselves at the forefront of popularity, the most praised and well-respected pieces of music always seem to find themselves firmly grounded in the formal rules of composition that have been widely accepted for centuries.

The reason behind this can be easily attributed to the notion that music is well founded in the world of mathematics, and the rules of music theory are indeed built upon it. Both the relations between pitches and durations are best defined by numbers and ratios. In fact, because of its reliance on precise measurement, music was considered until fairly recently its own branch of science [1]. This fact makes it tempting to both analyze and create music through a scientific approach, and it is indeed a venture that has been attempted many times over the course of human history, making great strides since the beginning of the digital age.

1.1 Early Exploration

The intersection of mathematics and music predates the computing age quite considerably. The topic of algorithmically composing music saw its initial explorations as early as 500 B.C. in the times of Pythagoras [2], when he developed the concept of "music of the spheres," in which he drew some of the first significant connections between the world of music and mathematics. Of course, Pythagoras could not have known what he was pioneering would one day spawn the algorithmic composition of music, as the term 'algorithm' wasn't even invented until 1120 [3]. From this point on, the world of music was situated comfortably in the middle of the mathematical spectrum, and a millennium later, Flavius Cassiodorus (ca. 485-575) described mathematics as a union of the four disciplines: arithmetic, music, geometry and astronomy [4].

Once the medieval period came around, composers began to formulate rules by which pitch relations and combinations were governed, laying the groundwork for music theory as a practice that would be followed and expanded upon for centuries [5]. It was in the 1700's with a game called Musikalische Würfelspiel [6], which translates from German to 'musical dice game,' that the rules were put to use in an algorithmic fashion. The game's most popular iteration, allegedly devised by Mozart himself, saw the user roll a pair of dice, and their composition would proceed based on the outcome being mapped to a ruleset Mozart outlined. These early experiments laid the ground work for algorithmic music to come.

1.2 The Data-Driven Intelligence Age

With the framework of algorithmic music already set centuries before, it was only natural that the concepts were brought into the world of computing as early as the 1950's, at the genesis of the information age. The most famous example from this time is Hiller and Isaacson's *Illiac Suite* [7], which used rule systems and Markov chains, a stochastic predictive system with no memory, to predict the next successive note based solely on the current note. As the work was expanded upon by colleagues and interested parties, the chains were designed to implement an *n*th-order technique, which allows the process to consider the last *n* notes, rather than only the most recent [6]. This initial work with Markov chains became the springboard of computerized algorithmic compositions.

Since this advent, the topic's exploration has increased drastically, and has branched into many different realms, with new techniques and structures being used as the basic building block of the composition process. In his book "Algorithmic Composition: Paradigms of Automated Music Generation," Gerhard Nierhaus split the topic into several distinct categories, including generative grammars, transition networks, genetic algorithms, cellular automata, artificial neural networks (ANNs) and artificial intelligence [3]. As these fields grow further apart, greater strides and achievements are being made within each.

The intersection of music and computing becomes even more pronounced when you approach the topic of data mining. Many have explored the potential of classifying music of all varieties, and results have been quite successful. Researchers Lebar, Chang & Yu [8] used classifiers to distinguish between the works of various classical composers using stylistic features as attributes. Basili, Serafini and Stellato [9] tackled the topic of popular music when they classified a dataset of music into six distinct genres based on features such as intervals, instruments used and meter changes. The basic structure of this study has been conducted by many, receiving respectable results overall.

1.3 Study Overview

While it is clear that the topic of music's intersection with computer science has been explored in many facets, there is still a gap when it comes to what a computer is capable of producing, and some of the most recent studies in the field of algorithmic composition still find themselves labeled as composition inspiration software [6]. The idea of hybridizing multiple of the above concepts has therefore become attractive, in an effort to achieve the best generative characteristics from multiple approaches. For this reason, we find it worthwhile to explore new avenues, and see what kind of new directions we can bring to the topic of algorithmic composition.

It became evident during the course of our research that one such hybridization comes from the potential of using the field of data mining to inform the decisions made during certain algorithmic composition techniques. Intersecting these two concepts has the potential of creating a smarter generative process, capable of replicating nuanced differences between several different categories of music, adapting to new forms of music being introduced, and minimizing the amount of human intervention required for some techniques. One such intersection that we saw potential in was using classification intelligence to inform a cellular automata composition system. It is under the guide of this general framework that we began our work.

2. Data

With any venture into the world of data mining, the first and most important task you must address is the data that you wish to use within the experiment. The topic of music presents a particular challenge in this respect, as the data at hand is not nearly as friendly for computer use as something purely numeric such as stock numbers or attendance projections may be. For this reason, a substantial amount of time needed to be dedicated to understanding the data of music, discovering what kind of characteristics are desirable to use from the data, and what kind of computer-friendly representations we have as options moving forward.

2.1 Musical Representation

In order to properly understand the data, it is important to first have a firm background in the formalities of music. For the sake of this experiment, we will be narrowing the scope of our focus entirely upon classical music, which we define as traditional Western music ranging from the Medieval era to the Modern era (not to be mistaken with the Classical era, which is distinction within the realm of classical music). The main reason for this decision is classical music's written consistency across history [5]. Music has evolved and expanded greatly since the days of Mozart and Bach and as a result, much of what is being created today in popular music has abandoned the concept of formally creating a written representation of the music. Recent years have seen the greatest decline in non-educational production of sheet music [10]. Luckily, classical music, by virtue of its creation for performances by individuals other than those composing, as well as its educational value, has a rich history of written representation. It is still most widely recorded in this manner today, and thus provides us with a much more stable and wide backlog for analyzation.

This backlog of written classical musical literature is comprised almost entirely within the medium of musical scores, or sheet music. Sheet music is a visual representation of music made up of symbols and words which convey all the information a performer must know to play the piece. Among other information, these symbols are capable of portraying which notes must be played at what time, the volume at which they are to be

played, and in what rhythm. This manor of recording music started as early as the ancient Greek and Middle Eastern civilizations where they began using basic music symbols as written reminders. It wasn't until the 9th century that Christian Monks began recording music on sheets. From this point on, the practice exploded in popularity, and has maintained the same basic structure [10].

2.2 Digital Formats

For hundreds of years, Western music has been represented by means of these musical scores. This has been relatively unchanged because it is an ideal notation for a musician to read and perform [10]. With the advent of the digital age, the necessity for a new representation of written music was realized. This was due to the complex nature of musical scores. It is quite difficult to teach a computer to parse through the various symbols and notations of music, making the task of retrieving the data necessary for processing challenging. As a result, the computer science community was met with the challenge of creating a new representation of music that could be more easily processed for the studies to come. Though many were proposed, two have risen above the others in the world of research, MIDI and **kern musical files. Both have their own unique advantages and disadvantages.

2.2.1 MIDI

First seeing its start in 1981 [11], the Musical Instrument Digital Interface (MIDI) format is one of the most widely used digital musical formats that exist. By virtue of its creation for use with electronic synthesizers, MIDI files contain representations of the musical score that are often recorded via humans playing the score with a synthesizer, though you can also find hand compiled MIDI representations.

Over time, this format has been adapted for use in scholarly research, with many toolkits being developed, such as jSymbolic [9], to extract data from the MIDI files. Because of its widespread use for a variety of functions, the backlog of MIDI scores to be

used for potential research is vast, but also unreliable. This is due to the fact that anyone with an electronic keyboard can plug it into a computer and create these files, regardless of their accuracy level. Despite this, we found throughout our survey of previous studies that MIDI is the most widely used file type in academic research concerning computer music.

2.2.2 **kern

While the MIDI format was created for a wide variety of computer music purposes, a format known as **kern was created with a much narrower intention. **kern files are musical representation files which fit within a broader syntax known as 'Humdrum.' Described by its creator David Huron as a "general-purpose software system intended to assist musical research" [12], the software was quite literally designed for use in projects like this. Researchers Lebar, Chang & Yu [8] used this format in similar research when attempting to classify musical scores by composer.

The Humdrum software can be split up into two separate entities: The Humdrum Syntax and the Humdrum Toolkit [12]. Humdrum Syntax is a grammar by which any file that falls under its guise must adhere to. **kern is a single file type under this syntax, and indeed the most widely used of them, designed to represent the core information for common Western Music. The format is capable of representing nearly every nuance found within a musical score, down to the direction the stem of a note is facing on the page. The other half of the equation, the Humdrum Toolkit, is described by Huron as a toolbox of 'utilities,' with over 70 inter-related software tools, which can be used to manipulate any data that conforms to the Humdrum syntax [12]. These tools, combined with the vast number of features that can be represented using the Humdrum Syntax, make it an attractive option in the realm of data mining.

While this format offers many advantages, there are certainly drawbacks to it as well. Because of its rather limited usage (being designed specifically for research purposes), the amount of data available in this file type is sparse. There have been a number of people who have contributed a substantial number of scores encoded in **kern format,

however the encoding process, which must be done entirely by hand, is a tedious one (though perhaps lends itself to a greater attention to detail), and there will never be a rich well of files to choose from.

Despite this deficiency, we found the format of **kern to be most compatible with the task at hand. The Humdrum toolkit offers us an effective way to extract any and all information about the score we may find useful, and the textual representation is also much friendlier to interpret on a visual level. With this decision, we began our work in data mining.

3. Data Mining

Data mining has exploded in recent years as an emerging concept in the area of computational intelligence. The applications of this new and intellectually stimulating field are plentiful, diverse, and exciting for those focusing on the topic. The phrase 'data mining' itself defines a rather broad idea, simply described as "the process of discovering useful information in large data repositories" [13]. In the pursuit of achieving this goal, data mining has been approached using several other distinct methodologies, such as classification, clustering and association, among others [13].

While each of these data mining methods have merit, and some may indeed be useful in future works while attempting to improve the algorithmic music composition challenge, this study has chosen to focus its attention on the topic of classification. Classification is defined as "the task of assigning objects to one of several pre-defined categories" [13]. This objective is achieved through the use a learning scheme that generates a set of rules for classifying instances into these pre-defined classes. The trained classifier is then able to predict the classes or categories based on the generated rules [14]. The predictive power of this form of data mining is one of the driving forces behind our decision to focus on classification, as a predictive rule-based system provides us a nice backbone upon which to build a music generator.

3.1 Data Extraction

In order to get the most out of the data mining process, there is a large amount of preparatory work that must be done to ensure that the information received as consequence of our work is valuable and significant. Our results are only as valuable as the system from which they were derived, so it is important to ensure we make the correct decisions leading up to the actual data mining taking place. Some of these decisions include dictating which pre-defined classes to supply our classifier, which features we would like our classifier to look at in making its categorizations, and the pre-processing and data extraction required to make the data accessible for the actual data mining process.

3.1.1 Classes

The first thing we needed to do when prepping our data for processing was select the pre-defined classes by which to separate the data, as the classification methodology necessitates. In musical classification, there have been studies that have done this in several manors, whether it be by composer, genre, or even decade. For the sake of our study, we found it most appropriate to create the classes based upon musical era within the classical spectrum.

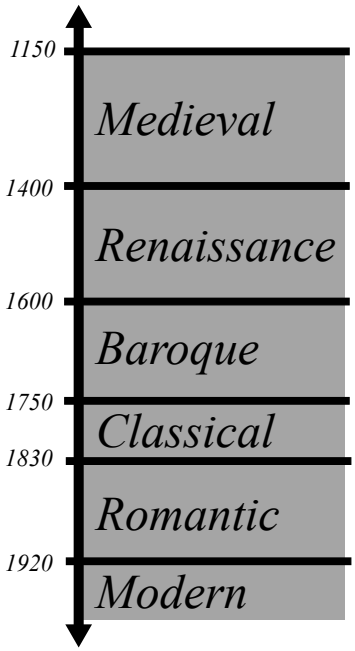

Figure 1 – A timeline displaying the order and generally agreed upon dates of the various eras of classical music

There have been several eras by which the style of a classical piece can be defined, roughly outlined in figure 1. The years in which these eras transitioned between one another have been debated by experts [5], however it is generally accepted that there are six distinct eras, ranging from the beginnings of formally composed music in the medieval era to the wildly innovative and often atonal modern era of classical music. Moreover, students and scholars of music are able to use their training in aural skills, such as identifying the interval between any two successive notes, among other musical features, to identify which of these eras a piece of classical music belongs to. This suggests that there are quantifiable differences in their structure that make it so and provides us great reason to believe a computer will be able to identify these differences as well.

3.1.2 Attributes

Our next step was to decide which attributes we would be basing our classification upon. In data mining techniques utilizing classification, these attributes – or features – are the sole factors analyzed in an attempt to generate rules for separating the data into the pre-defined classes it has been given [13]. It is therefore important to choose features that are both indicative of the stylistic-era under which the piece was composed, as well as replicable for the future generative process. The features decided upon after consideration of a number of factors, presented in figure 2, are based upon the notion of a musical interval. The task of choosing these attributes came with two major challenges; one musical and one computational.

	Attribute	Description
X_1	freqUni	Ratio at which unison intervals occur (unison/total)
X_2	freqStep	Ratio at which stepwise intervals occur (step/total)
X_3	freqThird	Ratio at which third intervals occur (third/total)
X_4	freqFourth	Ratio at which fourth intervals occur (fourth/total)
X_5	freqFifth	Ratio at which fifth intervals occur (fifth/total)
X_6	freqSixth	Ratio at which sixth intervals occur (sixth/total)
X_7	freqSeventh	Ratio at which seventh intervals occur (seventh/total)
X_8	freqOct	Ratio at which octave intervals occur (octave/total)

Figure 2 - List of attributes used in classification

By merit of the musical data we are using, there were countless numbers of attributes through which we had to sift in order to choose our features. As discussed in section 2.1, a piece of sheet music contains a vast amount of information, and our selected **kern format does little to narrow down that scope, as it does such an excellent job of preserving all the information recorded in a traditional score. Our chosen attributes must be indicative of the era the piece represents, so as to allow the classifier to accurately and practically determine which era the piece came from.

From a computational standpoint, we wanted to consider features that would lend themselves to both the classification process, as well as the generation process in the next step of our research. Classification mandates that each feature within its system be flat rather than structural – meaning that the value can be defined by either a numeric or discrete value [14]. Because of music's reliance on mathematics, this factor is not terribly delimiting, but it does help suggest which features may lend themselves best to the process: those which are finite and numerically categorized. It behooved us to focus on features which we could see as easily replicable in a future generative process, meaning features like dynamics would do little good on their own.

After consideration of these factors, the decision was made to focus upon the frequency with which certain musical intervals occur within the pieces of music. Before we delve into why exactly we made this decision, it is important to understand what an interval is.

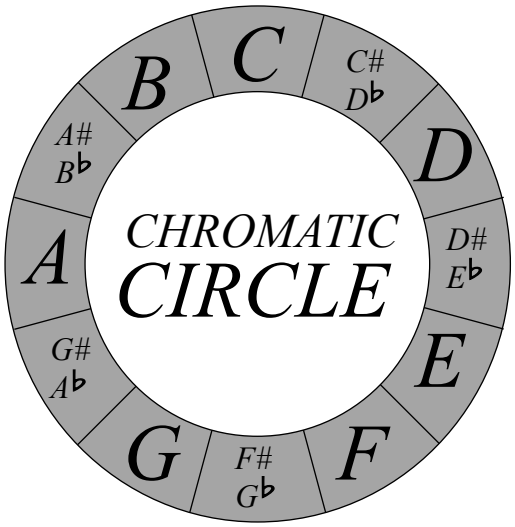

Figure 3 – A visual representation of the Chromatic Circle, the backbone on which Western music has been created.

The concept of a musical interval is built upon the very foundation of Western music: the chromatic circle (Figure 3), a cyclical scale of equal temperament made up of 12 total pitches [15]. A piece of music is comprised of a finite number of these 12 pitches

in linear progression. A musical interval is the distance between any two successive pitches within the piece, typically ranging from unison to octave (Figure 4). The most basic of these intervals is defined as an octave, which corresponds to a 2:1 ratio. For instance, we perceive a pitch at 110 Hz to be an octave below a 220 Hz, both of which represent the note 'A' [15]. Human beings perceive these ratios to be the same pitch, only at a higher or lower frequency, allowing for the cyclical nature of the scale. We can therefore identify the interval between any two successive notes based upon this scale. While it is not unheard of to have music that utilizes other pitches not represented on the chromatic scale (this is a practice that is observed in many traditional forms of music in the eastern hemisphere), this scale is truly the backbone of Western music.

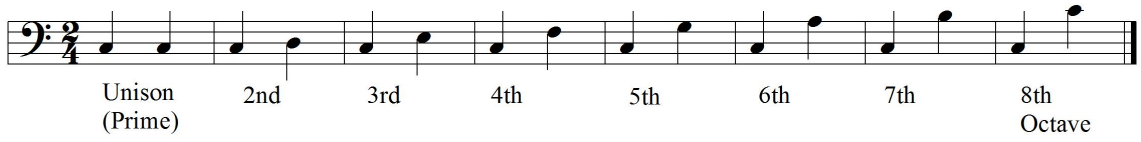

Figure 4 - Visual representation of musical intervals

The first reason for this selection comes from the realm of aural skills, in which it is common to use musical intervals as a way to identify differences between eras [16]. Though there are a number of features which are often cited when it comes to aurally distinguishing between eras, intervals are almost always presented as evidence in such efforts, and their status as a cornerstone of music theory make them an obvious answer to our query. Secondly, we found that the basis of intervals is an excellent building block upon which to build a generative system, which will be touched upon in greater detail later in our discussion.

3.1.3 Pre-Processing

Once all of these important determinations had been made, it was time to clean the data, and extract the features that had been decided upon. The first step was to collect the data to be used. Though the wealth of **kern scores are not as vast as desired, we were able to accumulate 262 unique pieces of classical music from a variety of eras (Figure 5) through two Humdrum databases. It is worth noting that the distribution of data entries between

these eras were not even across all classes, as there are far less pieces of pre-baroque music that have been encoded using **kern format than that of eras such as the classical or romantic era, which feature much more notable composers and pieces which have endured the test of time.

Class	Number of Data Entries
Medieval	10
Renaissance	26
Baroque	77
Classical	50
Romantic	70
Modern	29
Total	**262**

Figure 5 – Distribution of data between class types

The next step was to extract the features that we desired to use in the classification process. This was perhaps the most tedious task, though we were able to do so in a Linux command line window with a combination of both the Humdrum toolkit, designed for the **kern file format (and other formats following the Humdrum Syntax), as well as pattern matching using egrep. In the end, we stored the number of times each individual interval appeared and set it as a ratio against the total number of musical intervals encountered.

We appended these ratios (Figure 2), along with the era with which the piece is categorized (Figure 1), to the end of an .arff (Attribute-Related File Format) file with appropriate headings. Doing this in a loop, we were able to create one file with all 262 musical scores represented. It is with this document that we begin our classification.

3.2 Classification

Classification is an umbrella term to define the task of separating data into distinct categories, and as such there are a large variety of methods that can be implemented in order to achieve the same goal. It became obvious that we would need to test our dataset with a variety of these classification methods in order to receive the best results possible,

and we began work on feeding the data we compiled into five different classification approaches of varying sophistication levels.

The two high-level algorithms we utilized in our tests were Multilayer Perceptron (MLP) and Logistic Regression. Based upon an artificial neural network, MLPs use layers of input nodes, output nodes, and two or more layers of hidden nodes to find the most likely path from our input data (comprised of the aforementioned musical interval attributes) to an output identifying whether the data falls within a given class (musical era) or not [13] (Tan). Logistic Regression on the other hand implements a statistical model built upon the probability that a certain piece of data falls within a given class or not. While both of these methods are dichotomous (only have one of two outcomes), they can be used to classify sets with more than two classes when given the dichotomous options of "within the given class" or "not within the given class".

While Naïve Bayes falls into the category of a lower level classifier, it perhaps deserves a little more recognition than the title suggests. While it does not use sophisticated algorithms like the above outlined MLP and Logistic Regression models, it is a very well-respected model in the data mining community, and it indeed performs just as well or better than sophisticated models in some instances. The premise of this model is simple, based upon Bayes theorem, which provides a way of calculating the posterior probability of an attribute fitting a defined class [17]. The success of this algorithm lies in the fact that each given attribute is considered independent of one another. As a result, the most probable class is calculated based upon each attribute identified separately, and these probabilities are then multiplied against each other to determine the probability that the piece of data, in this case a musical piece, falls into a given class.

The other lower level of classifiers used in our study fall into the category of rule-based and decision tree induction predictors. We selected one of each such classifiers, JRip (Rule-Based) and J48 (Decision Tree Induction). JRip uses simple if…then rule structures to split the data into the given classes [13]. J48 uses a similar system within a decision tree structure, where there is a leaf node associated with each of the pre-determined classes,

and classification rules are derived and placed within the ascending nodes as the data is analyzed [17].

3.3 Results

	Medieval	Renaissance	Baroque	Classical	Romantic	Modern	**Average**
MLP	0.964	0.958	0.854	0.988	0.836	0.996	**0.933**
LR	0.981	0.951	0.808	0.921	0.885	0.927	**0.885**
Naïve	0.938	0.931	0.73	0.889	0.853	0.871	**0.838**
JRip	0.705	0.841	0.73	0.874	0.704	0.836	**0.773**
J48	0.798	0.777	0.681	0.804	0.741	0.753	**0.753**

Figure 6: Results of classifier based on AUC of ROC graph.

The charts outlined in Figure 6 show a complete picture of the results received from each of the five aforementioned methods of classification. Using a method of testing known as ten-fold cross validation, the set of data is partitioned into ten equal segments. During each iteration of testing, $9/10^{ths}$ of the data gets assigned to act as a training set, used to educate the classifier and build its predictive ability. The other $1/10^{th}$ of the data is designated to be the test set, used to analyze how well the classifier is able to predict the class the data belongs to. This process is reiterated ten times, until all the data has been used as part of a test set.

In analyzing the results, we chose to focus on the value of the AUC (area under the curve) of a Receiver Operating Characteristic graph as an indication of the success of our classifiers. The reason for this decision is due to the inconsistent number of data pieces between each class represented (Figure 5). The Receiver Operating Characteristic (ROC) Curve maps the True Positive Rate (true positives / all positives) against the False Positive Rate (false positives / all positives). This produces a curve that will represent how often a piece is mistakenly identified as other than its proper class, rather than produce a true precision rate, which may be skewed as a result of the uneven distribution of data. A perfectly classified set of data would have an AUC of 1.

As seen in the charts, our five classifier models performed at varying levels of accuracy. The highest-level algorithm used, the Multilayer Perceptron model, produced AUC rates of .933, while our rule-based and decision tree classifiers lagged behind with AUC rates of .773 and .753 respectively. Based on the complexity of each algorithm, it didn't come as a surprise that the results fell the way they did. Higher level algorithms such as MLPs or Logistic Regression have a natural head start on decision tree or rule-based algorithms. Perhaps the biggest outlier in the classifiers presented is the Naïve Bayes model, with an excellent AUC rate of .838, despite the algorithm being quite simple and intuitive.

4. Generation

After analyzing the results of the classifiers, the first step in creating our algorithmic composition software was to choose one classifier to use going forward in the hybridization process. On top of providing class predictions, each classifier supplied an additional output, intended to inform the reader on how it's decision rules were devised. These outputs are important, as they are the building block upon which we intend to build our music generator. Of the five classifiers, the first two eliminated were the rule-based and decision tree models, JRip and J48. While the classifiers provided positive features, such as easy to understand outputs that outlined the rules used explicitly, it was clear that these approaches were simply not of the same accuracy as their higher-level counterparts.

Of our three remaining classifiers, we chose next to eliminate the higher-level classifiers, Multilayer Perceptron and Logistic Regression. Despite these algorithms statistically doing a better job of classifying the musical scores, MLPs and Logistic Regression are very complex algorithms, and as a result, the output does not give a digestible answer as to why the classes were separated the way they were. For this reason, it was difficult to conceive of a way to use these classifiers to inform the generative process of any algorithmic composition software.

We decided to proceed using the Naïve Bayes approach because it supplied us with a nice middle ground between the previously mentioned choices. It provides an easy, statistical output for us to easily adapt to the generative process. On top of this, the Bayes model yielded a much more respectable AUC value (.838) than the lower-level algorithms of J48 (.753) and JRip (.773).

4.1 Method

In perhaps our most contributory work, we move to the generation process of the experiment. The task laid ahead of us was to find a way to utilize the intelligence gained from our Naïve Bayes classifier to inspire the algorithmic composition of music. After consideration of the classifier results and output, we decided to turn our attention to an avenue of algorithmic composition that has been less explored than some others such as artificial neural networks and formal grammars: Cellular automata.

4.1.1 Cellular Automata

The concept of cellular automata (Singular: Automaton) was first proposed by John von Neumann in the 1950's and reached a peak in popularity during the 70's due to John Conway's now famous "Game of Life" 3-D cellular automata model [17]. Based upon the biological cellular replication process, a cellular automata model is represented by a grid of cells, each of which is represented as one of a finite number of states (i.e. "ON" or "OFF"). This grid can be of any finite number of dimensions. The grid progresses in temporally-linear fashion, with each cell shifting states at any given step in time. This shift of the cell states is based upon two factors: the states of the surrounding cells in a pre-determined area defined as it's neighborhood, and a set of transitionary rules which dictate the outcome based on that neighborhood [17]. One of the most famous example of a cellular automata, the Wolfram Elementary Algorithms (Figure 7), adds a new line of cells below the previous generated line with each sequential step in time, with the states of these new cells based upon a neighborhood of the three cells directly above it, and a selected transitionary rule set [18]. With 256 possible rule sets, there are countless possibilities of

how the algorithm can compose the sequence of cells, and many produce interesting patterns, such as fractals.

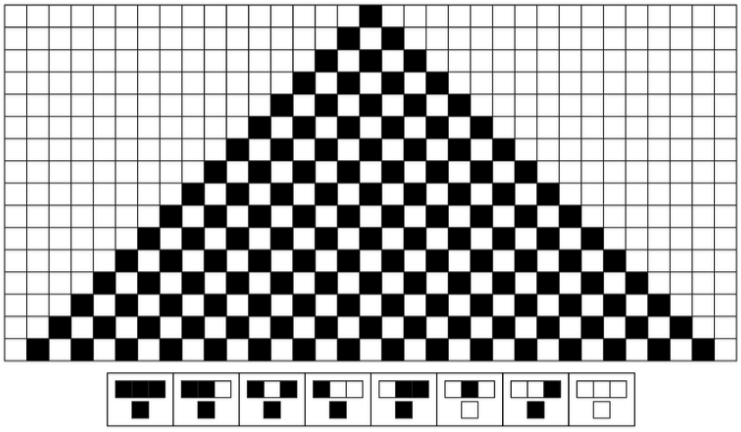

Figure 7 - Rule 250 in the Wolfram Elementary Algorithm Suite

Rule model's such as Wolfram's provide a unique avenue of exploration for musical composition. The patterns found within these automata rules provide a built-in approach to chaotic music composition. However, those preliminary cellular automata models were only able to create music in an "uncontrolled" way and resulted in music that was not necessarily homogenous with any preconceived style [6]. The next natural step was to create transitionary rules that were informed by music theory, so as to control the music being generated.

4.2.2 Adapted Musical Model

In an attempt to explore this avenue of musically informed cellular automata rule generation, we devised of a system inspired by the aforementioned Wolfram Algorithm. In a four-wide grid, each cell has one of two states: "On" and "Off." These states allow for us to interpret a four-cell phrase as a binary sequence. We chose to map these binary sequences (16 sequences for a four-byte binary number) to the 12 notes of the chromatic circle (Figure 8). While this rudimentary system does not take into account rhythm, a rest character was also encoded for potential future works.

Binary Represenation of Notes	
0000	START
0001	C
0010	C#/D♭
0011	D
0100	D#/E♭
0101	E
0110	F
0111	F#/G♭
1000	G
1001	G#/A♭
1010	A
1011	A#/B♭
1100	D
1101	C
1110	Rest
1111	TERMINATE

Figure 8 – A table mapping the values of a four-bit binary sequence to the values within the chromatic circle for use in conjunction with cellular automata musical composition

After the groundwork of our cellular automata model was fully laid out, it was time to create transitionary rules inspired by the intelligence gained through our classification process. Before each shift in states, a random float value between 0 and 1 was generated. Using the output from our Naïve Bayes classifier, which gave us statistical probabilities of each musical interval occurring at a given shift in time for a given era, this random float was mapped to one of the eight interval possibilities. The states of each cell in the four-byte sequence would then transition from the previous states to a new sequence of differing states based on this mapping. The distance between the old sequence and new would therefore be equivalent to the musical distance of the determined interval. We are essentially generating the interval between the notes, rather than the note itself. Along with creating more aurally pleasing musical phrases, this helps ease the challenges of representing key signatures within pieces of music.

To further demonstrate the potentials of this system, the software is able to switch between eras at the will of the user. Based upon the values output by the Naïve Bayes classifier, the system will replace the statistical values for the generated rule to each respective era of classical music at the click of a button, so as to encourage the system to follow the tendencies of the desired era. This feature helps the software stand out and puts to use the predictive power of our classification approach to rule generation.

The last feature we implemented was a range-check system. In preliminary testing, we found that allowing the note to change in ascending or descending fashion on a 50-50 basis, while relatively common sight within the world of music, was not controlled enough for our experiment, as the true randomness allowed for many algorithmic compositions to get out of hand in terms of range. We therefore found the average distance between the highest note and lowest note within an era of music and dictated that the composition software stays within that range when composing. This allows music that has traditionally had more range to flourish in this sense, while static pieces from earlier eras stick within a more contained range of notes.

4.2 Results

The result of our efforts is a composition software that is able to imitate any one of six distinct eras of classical music. The system linearly produces a sequence of successive notes based upon the intervals between the previous note and the newly generated note. The pitches are outputted as they are generated using a Java MIDI import at a constant rate that can be changed in the code (currently set to one note every 750 milliseconds).

With the system functioning in the desired fashion, our next step was to analyze just how well our composition software was able to imitate the various classical eras. We chose to implement two different methods of analyzation, to see how well the system was able to reproduce the various eras in both a mathematical and an aural fashion.

4.2.1 Indirect Analyzation

In our first of two efforts to analyze the results of our compositions, we used an indirect approach closely tied to the ways in which we created the software – classification. While we previously described a 'ten-fold cross verification' approach during our initial classification process, we decided upon using a 'test set' approach for the following exercise. In this approach, we feed the classifier a set of data points known as a training set to develop its knowledge on what distinguishes the different classes, and then feed it a set of data points known as a test set to see how accurately it is able to classify those pieces within the given classes.

To do this, we generated sixty pieces of algorithmically composed music – ten within each era and each piece with a length of 100 notes. We extracted from these compositions the same features we outlined in section 2.1.2, and translated the results into an .arff file mirroring the structure of our previously used .arff file. We then used this file as our test set and provided the file from our initial classification exercise as a training set. We ran these classification techniques on four of the five classifiers used in our original exercise, excluding the Naïve Bayes classifier we used to inform the composition software, as it would provide an unnaturally insightful look into the data, resulting in skewed results. The classifiers' results are displayed in the chart below (Figure 9).

	Medieval	Renaissance	Baroque	Classical	Romantic	Modern	**Average**
MLP	0.942	0.9	0.858	0.918	0.754	0.986	**0.893**
LR	0.978	0.938	0.824	0.946	0.836	0.998	**0.92**
JRip	0.852	0.753	0.662	0.816	0.582	0.786	**0.742**
J48	0.812	0.757	0.757	0.8	0.678	0.826	**0.772**

Figure 9: Our compositions' classification results based on AUC of ROC graph.

The classifiers performed quite well in determining the era which our composition software was attempting to replicate. In fact, the classifiers success rates were nearly identical to the success rates they experienced with traditionally composed pieces of music, with their short comings being seen in the same categories. The only classifier that saw significant changes in performance was that of the logistic regression approach, which saw

the average ROC percentage jump from .885 to .92. These results alone are highly encouraging.

4.2.2 Direct Analyzation

To double down on our analysis, we decided to take a direct approach to the matter as well and consulted a number of experts in music. In total, five scholars of music took part in a survey to determine how well they could distinguish the success of our classifier. The exercise was simple: We generated three 15 second clips of music from each era and presented them together in a random order to the experts. We asked at the conclusion of each triplet for the experts to indicate which era they believed the composition software was meant to represent, and their confidence on a scale from 1-5. We also gave the experts an opportunity to explain how they arrived at that answer, and why they gave the confidence level they did.

The results of our direct method of analysis were not as encouraging as the indirect method. Of our experts, only one was able to predict 50% of the eras correctly, and one failed to predict a single era. The confidence levels hovered between one and three for most answers, with a distinct increase in both confidence and accuracy with the modern era, of which four of our five experts correctly predicted.

5. Discussion

It is clear that the results of our direct method of analysis tell a very different story than the indirect method. While our classifiers were able to tell which era of music was being replicated with our composition software to a high level of accuracy, experts in music had a much harder time doing so, with a success rate of below 30% when presented the option of all six eras.

Because of the nature of the process, it comes as no surprise that our direct and indirect methods of analysis yielded such different results. This is likely because of the

limited scope with which we approached the problem, deciding to focus on a very select number of features, even though the differences in musical styles between the eras is defined by many more features, such as rhythm and harmony (A distinction many of our experts pointed out during their survey), as well as the types of instruments being used in the pieces, which is ignored by using a MIDI output. Despite this, it is certainly promising that the features we did choose to use in the experiment yielded such high results in our indirect method of analysis. This suggests that, even if the music is not very aurally identifiable yet, trained AI has the ability to distinguish the differences. This result suggests that the project has potential moving forward, and direct results may be achieved by hybridizing this method with others designed to take rhythm and harmony into account.

5.1 Applications

For now, it seems the application of this software lays firmly in the category of 'composition inspiration software' that encompasses so much of the work that has been done in the field, though it certainly shows signs that it has the potential to be more. The success of our classifiers in determining which era the piece was meant to replicate indicates that there is a lot of potential in the system, when put to use in the correct fashion. The cellular automata system also lends itself to be used with different classifiers, or perhaps even different types of music, as it has been designed to be adapted to any kind of transitionary rule set.

5.2 Future Works

At the end of the study, our thoughts on moving forward are much the same as they were when we began. The prospect of hybridizing the various methods of algorithmic music composition with data mining is a vast well of potential which this study has only begun to scratch the surface of. Based on the experts' opinions that our focus on the feature of musical intervals was not enough to encompass all the characteristics of a classical musical era implies that more hybridization must be done with this system to make it more aurally accurate.

There are a number of avenues that could be explored in the pursuit of improving the system in such a manor. This could include varying the instrumentation based on which era it derives from, factoring into the composition rhythm and dynamics, and creating a two-line system that generates harmonious interval sequences. Another feature that could yield positive results would be to adapt the system to employ an n^{th}-order technique. This would allow the music to flow with more natural phrasing and would allow the intervals to take into account where it appears in the musical phrase. Lastly, improvements could be made to the range-check system implemented in this study, which would go hand-in-hand with the phrasing achieved in the n^{th}-order additions.

```java
/*
 * Algorithmic Music Composition Software
 * @author Tom Donald Richmond
 * @version 2.0
 * @since 02/12/17
 */

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ConcurrentModificationException;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.Timer;
import javax.swing.JOptionPane;

import javax.sound.midi.*;

public class CellularAutomataMusic  extends JFrame{

    private static final Color white = Color.WHITE, black = Color.BLACK;

    private Board board;
    private JButton start_pause, medieval, renaissance, baroque, classical, romantic, modern;
    // variables to track total number of interval occurrences
    int t;
    // variables to track the occurrences of each interval for testing
    int[] totals = new int[8];
    // variable to hold string value representing era
    String era;
    // Boolean variable representing
    Boolean analysis = false;

    /*
    * Creates blank board to feature automata, with start button to
    * commence composition, as well as buttons to select epoch
```

```
42      * */
43      public CellularAutomataMusic(){
44
45          board = new Board();
46          board.setBackground(white);
47
48          /*
49          * Create buttons for start/stop
50          * */
51          start_pause = new JButton("Compose");
52          start_pause.addActionListener(board);
53
54          /*
55           * Create buttons for epoch selection
56           * */
57          medieval = new JButton("Medieval");
58          medieval.addActionListener(board);
59          renaissance = new JButton("Renaissance");
60          renaissance.addActionListener(board);
61          baroque = new JButton("Baroque");
62          baroque.addActionListener(board);
63          classical = new JButton("Classical");
64          classical.addActionListener(board);
65          romantic = new JButton("Romantic");
66          romantic.addActionListener(board);
67          modern = new JButton("Modern");
68          modern.addActionListener(board);
69
70          /*
71           * Subpanel for epoch selection
72           * */
73          JPanel subPanel = new JPanel();
74          subPanel.setLayout(new java.awt.GridLayout(6, 1));
75          subPanel.add(medieval);
76          subPanel.add(renaissance);
77          subPanel.add(baroque);
78          subPanel.add(classical);
79          subPanel.add(romantic);
80          subPanel.add(modern);
81
82          /*
83           * Add buttons to layout
```

```
84          * */
85         this.add(board, BorderLayout.CENTER);
86         this.add(start_pause, BorderLayout.SOUTH);
87         this.add(subPanel, BorderLayout.WEST);
88         //this.setLocationRelativeTo(null);
89 
90         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
91         this.pack();
92         this.setVisible(true);
93 
94     }
95 
96     public static void main(String args[]){
97         new CellularAutomataMusic();
98     }
99 
100    /*
101     * Board object featuring 4x15 Automata model, black and white values
102     * */
103    private class Board extends JPanel implements ActionListener{
104 
105        // Variables for board dimensions
106        private final Dimension DEFAULT_SIZE = new Dimension(15, 4);
107        private final int DEFAULT_CELL = 40, DEFAULT_INTERVAL = 100,
   DEFAULT_RATIO = 50;
108        private Dimension board_size;
109        private int cell_size, interval, fill_ratio;
110 
111        //boolean whether the composer is active
112        private boolean run;
113        // Timer for playing notes evenly
114        private Timer timer;
115        // variables to ensure the composer runs linearly
116        public int myOctave = 5, currentDiff = 0, range;
117        // variable to store the probability of each interval
118        double uni, step, third, fourth, fifth, sixth, seventh, octave;
119        // boolean to see if an epoch has been selected
120        boolean selected = false;
121        //grid to display automata-model
122        private Color[][] grid;
123 
124 
```

```
125         /*
126          * Default constructor for Board object
127          */
128         public Board(){
129             board_size = DEFAULT_SIZE;
130             cell_size = DEFAULT_CELL;
131             interval = DEFAULT_INTERVAL;
132             fill_ratio = DEFAULT_RATIO;
133             run = false;
134 
135 
136             grid = new Color[board_size.height + 1][board_size.width + 1];
137             for (int h = 0; h < board_size.height; h++)
138                 for (int w = 0; w < board_size.width; w++){
139                     //int r = (int)(Math.random() * 100);
140                     //if (r >= fill_ratio)
141                     //grid[h][w] = black;
142                     //else grid[h][w] = white;
143                     grid[h][w] = white;
144                 }
145             timer = new Timer(interval, this);
146         }
147 
148         @Override
149         public Dimension getPreferredSize(){
150             return new Dimension(board_size.height * cell_size,
    board_size.width * cell_size);
151         }
152 
153         @Override
154         public void paintComponent(Graphics g){
155             super.paintComponent(g);
156             for (int h = 0; h < board_size.height; h++){
157                 for (int w = 0; w < board_size.width; w++){
158                     try{
159                         if (grid[h][w] == black)
160                             g.setColor(black);
161                         else if (grid[h][w] == white)
162                             g.setColor(white);
163                         g.fillRect(h * cell_size, w * cell_size,
    cell_size, cell_size);
164                     }
```

```
165                     catch (ConcurrentModificationException cme){}
166                 }
167             }
168         }
169 
170         /*
171          * Method to re-adjust the probability values when new epoch is
    selected
172          * @param String representing epoch
173          */
174         public void changeEpoch(String epoch) {
175             if(epoch=="medieval") {
176                 playNote(60);
177                 uni = 0.1484;
178                 step = 0.4998;
179                 third = 0.1178;
180                 fourth = 0.0371;
181                 fifth = 0.0234;
182                 sixth = 0.004;
183                 seventh = 0.0014;
184                 octave = 0.0057;
185                 range = 14;
186                 era = "Medieval";
187             }
188             else if(epoch=="renaissance") {
189                 playNote(62);
190                 uni = 0.2571;
191                 step = 0.4305;
192                 third = 0.1061;
193                 fourth = 0.0728;
194                 fifth = 0.048;
195                 sixth = 0.0048;
196                 seventh = 0.0006;
197                 octave = 0.0094;
198                 range = 22;
199                 era = "Renaissance";
200             }
201             else if(epoch=="baroque") {
202                 playNote(64);
203                 uni = 0.2623;
204                 step = 0.3558;
205                 third = 0.1114;
```

```
206                 fourth = 0.0728;
207                 fifth = 0.0442;
208                 sixth = 0.0292;
209                 seventh = 0.0108;
210                 octave = 0.0379;
211                 range = 23;
212                 era = "Baroque";
213             }
214             else if(epoch=="classical") {
215                 playNote(66);
216                 uni = 0.148;
217                 step = 0.3964;
218                 third = 0.1713;
219                 fourth = 0.0818;
220                 fifth = 0.0574;
221                 sixth = 0.0435;
222                 seventh = 0.0195;
223                 octave = 0.0353;
224                 range = 25;
225                 era = "Classical";
226             }
227             else if(epoch=="romantic") {
228                 playNote(68);
229                 uni = 0.207;
230                 step = 0.2791;
231                 third = 0.1112;
232                 fourth = 0.0649;
233                 fifth = 0.0416;
234                 sixth = 0.0282;
235                 seventh = 0.0123;
236                 octave = 0.0217;
237                 range = 30;
238                 era = "Romantic";
239             }
240             else if(epoch=="modern") {
241                 playNote(70);
242                 uni = 0.3086;
243                 step = 0.2153;
244                 third = 0.1011;
245                 fourth = 0.1053;
246                 fifth = 0.0723;
247                 sixth = 0.0591;
```

```
248                 seventh = 0.0364;
249                 octave = 0.0571;
250                 range = 37;
251                 era = "Modern";
252             }
253             else {
254                 System.out.println("Woah, how'd you manage that bud?");
255             }
256         }
257 
258         /*
259          * Method designed to generate a new musical note value based on given previous note value
260          * @param int prevVal
261          * @returns int newVal
262          * */
263         public int ruleGenerator(int prevVal){
264             if (prevVal == 0){
265                 return 1;
266             }
267 
268             /* Sets ascLim and descLim to half of the average range of the
269              * given epoch. DescLim gets the ceiling arbitrarily*/
270             int ascLim = range/2;
271             int descLim= (range/2) + (range%2);
272 
273             double running = 0.0;
274             double value = Math.random();
275 
276             int newVal;
277             int diff = 0;
278             int direction = (int)(Math.random()*2);
279 
280             /* determines before each note whether it was generated to be ascending
281              * or descending. This process is regulated with ascLim and descLim */
282             boolean ascending = false;
283             if(direction == 1)
284                 ascending = true;
285 
286             /* Resets the valFound var to false for next note generation
```

```
   */
287             boolean valFound = false;
288
289             /* checks which range the generated number falls in and
    produces a
290              * note based on this value. Once note is found, valFound is
    set to
291              * true, and no other if statements are reached. It will
    access each
292              * if statement until the correct is found, increasing running
    total
293              * as it goes. */
294             if (value <= uni){
295                 totals[0]+=1;
296                 t+=1;
297                 diff = 0;
298                 valFound = true;
299                 System.out.println("Unison");
300             }
301             running += uni;
302             if ((value <= step + running) && valFound == false){
303                 totals[1]+=1;
304                 t+=1;
305                 diff =  1;
306                 valFound = true;
307                 System.out.println("Step");
308             }
309             running += step;
310             if (value <= third + running && valFound == false){
311                 totals[2]+=1;
312                 t+=1;
313                 diff =  2;
314                 valFound = true;
315                 System.out.println("Third");
316             }
317             running += third;
318             if (value <= fourth + running && valFound == false){
319                 totals[3]+=1;
320                 t+=1;
321                 diff =  3;
322                 valFound = true;
323                 System.out.println("Forth");
```

```
324             }
325             running += fourth;
326             if (value <= fifth + running && valFound == false){
327                 totals[4]+=1;
328                 t+=1;
329                 diff =  4;
330                 valFound = true;
331                 System.out.println("Fifth");
332             }
333             running += fifth;
334             if (value <= sixth + running && valFound == false){
335                 totals[5]+=1;
336                 t+=1;
337                 diff =  5;
338                 valFound = true;
339                 System.out.println("Sixth");
340             }
341             running += sixth;
342             if (value <= seventh + running && valFound == false){
343                 totals[6]+=1;
344                 t+=1;
345                 diff =  6;
346                 valFound = true;
347                 System.out.println("Seventh");
348             }
349             running += seventh;
350             if (value <= octave + running && valFound == false){
351                 totals[7]+=1;
352                 t+=1;
353                 diff =  7;
354                 valFound = true;
355                 System.out.println("Octave");
356             }
357 
358             //System.out.println((currentDiff+diff) +": total diff");
359             if (ascending && currentDiff + diff >= ascLim) {
360                 System.out.println("Switched, too high");
361                 ascending = false;
362             }
363             if (!ascending && -1*(currentDiff - diff) >= descLim) {
364                 System.out.println("Switched, too low");
365                 ascending = true;
```

```
366             }
367             System.out.println("Ascending = "+ascending);
368             if(ascending){
369                 currentDiff += diff;
370                 System.out.println(currentDiff);
371                 newVal = prevVal;
372                 for (int i = 0; i < diff; i++){
373                     if (newVal == 5 || newVal == 12)
374                         newVal += 1;
375                     else
376                         newVal += 2;
377                     if (newVal > 12) {
378                         myOctave++;
379                         newVal -= 12;
380                     }
381                 }
382             }
383             else{
384                 currentDiff -= diff;
385                 System.out.println(currentDiff);
386                 newVal = prevVal;
387                 for (int i = 0; i < diff; i++){
388                     if (newVal == 6 || newVal == 13 || newVal == 1)
389                         newVal -= 1;
390                     else
391                         newVal -= 2;
392                     if (newVal < 1) {
393                         newVal += 12;
394                         myOctave--;
395                     }
396                 }
397             }
398             System.out.println(newVal + " " + ascending);
399             int noteVal = toNote(newVal, ascending);
400 
401             //System.out.println(prevVal);
402             //newVal = 1+((int)(Math.random()*12));
403             return noteVal;
404         }
405 
406         /*
407          * Method designed to generate a new musical note value based on
```

```
   given previous note value
408          * @param int prevVal
409          * @returns int newVal
410          * */
411         public void ruleGeneratorAnalysis(){
412 
413             double running = 0.0;
414             double value = Math.random();
415 
416             /* Resets the valFound var to false for next note generation
   */
417             boolean valFound = false;
418 
419             /* checks which range the generated number falls in and
   produces a
420              * note based on this value. Once note is found, valFound is
   set to
421              * true, and no other if statements are reached. It will
   access each
422              * if statement until the correct is found, increasing running
   total
423              * as it goes. */
424             if (value <= uni){
425                 totals[0]+=1;
426                 t+=1;
427                 valFound = true;
428             }
429             running += uni;
430             if ((value <= step + running) && valFound == false){
431                 totals[1]+=1;
432                 t+=1;
433                 valFound = true;
434             }
435             running += step;
436             if (value <= third + running && valFound == false){
437                 totals[2]+=1;
438                 t+=1;
439                 valFound = true;
440             }
441             running += third;
442             if (value <= fourth + running && valFound == false){
443                 totals[3]+=1;
```

```
444                 t+=1;
445                 valFound = true;
446             }
447             running += fourth;
448             if (value <= fifth + running && valFound == false){
449                 totals[4]+=1;
450                 t+=1;
451                 valFound = true;
452             }
453             running += fifth;
454             if (value <= sixth + running && valFound == false){
455                 totals[5]+=1;
456                 t+=1;
457                 valFound = true;
458             }
459             running += sixth;
460             if (value <= seventh + running && valFound == false){
461                 totals[6]+=1;
462                 t+=1;
463                 valFound = true;
464             }
465             running += seventh;
466             if (value <= octave + running && valFound == false){
467                 totals[7]+=1;
468                 t+=1;
469                 valFound = true;
470             }
471 
472             /* When the composer has generated 100 notes,
473              * it automatically calculates the results and prints
474              * for analysis process */
475             if(t==100) {
476                 System.out.println(kernResults());
477                 //JOptionPane.showMessageDialog(null,kernResults());
478                 clearStats();
479             }
480         }
481 
482         /*
483          * Method that takes note value representation from binary as integer, prints corresponding
484          * value and plays note using MIDI output
```

```
485          * @param int val - Value of note (1-13) generated by the rule system
486          * @returns String letter value equivelant to corresponding int value
487          * */
488         public int toNote(int val, Boolean asc){
489             int noteVal;
490             int C = myOctave * 12;
491 
492             if(val == 1 || val == 13){
493                 noteVal = C+0;
494                 System.out.println("C");
495             }
496             else if(val == 2){
497                 noteVal = C+1;
498                 System.out.println("C#/D-");
499             }
500             else if(val == 3){
501                 noteVal = C+2;
502                 System.out.println("D");
503             }
504             else if(val == 4){
505                 noteVal = C+3;
506                 System.out.println("D#/E-");
507             }
508             else if(val == 5){
509                 noteVal = C+4;
510                 System.out.println("E");
511             }
512             else if(val == 6){
513                 noteVal = C+5;
514                 System.out.println("F");
515             }
516             else if(val == 7){
517                 noteVal = C+6;
518                 System.out.println("F#/G-");
519             }
520             else if(val == 8){
521                 noteVal = C+7;
522                 System.out.println("G");
523             }
524             else if(val == 9){
```

```
525                 noteVal = C+8;
526                 System.out.println("G#/A-");
527             }
528             else if(val == 10){
529                 noteVal = C+9;
530                 System.out.println("A");
531             }
532             else if(val == 11){
533                 noteVal = C+10;
534                 System.out.println("A#/B-");
535             }
536             else if(val == 12){
537                 noteVal = C+11;
538                 System.out.println("B");
539             }
540             else {
541                 return 0;
542             }
543             //System.out.println(noteVal);
544             playNote(noteVal);
545             return val;
546         }
547 
548         /*
549          * (non-Javadoc)
550          * Action Listener for all buttons, compose, terminate, medieval,
551          * renaissance, baroque, classical, romantic and modern.
552          * @see
    java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
553          */
554         public void actionPerformed(ActionEvent e) {
555 
556             //reads binary value of last sequence
557             int a = 0, b = 0, c = 0, d = 0, val = 0;
558 
559             //counts binary from board for conversion to decimal
560             if (grid[0][board_size.width-1]  == black)
561                 a = 1;
562             if (grid[1][board_size.width-1]  == black)
563                 b = 1;
564             if (grid[2][board_size.width-1]  == black)
565                 c = 1;
```

```
566             if (grid[3][board_size.width-1]  == black)
567                 d = 1;
568 
569             //converts binary sequence into decimal with variable val
570             if(a==1)
571                 val+=8;
572             if(b==1)
573                 val+=4;
574             if(c==1)
575                 val+=2;
576             if(d==1)
577                 val+=1;
578 
579             //shifts bottom n-1 sequences up to make room for next
    sequence
580             for (int h = 0; h < board_size.height; h++){
581                 for (int w = 0; w < board_size.width-1; w++){
582                     grid[h][w] = grid[h][w+1];
583                 }
584             }
585 
586             //repaints the bottom line sequence based on rule
587             if (e.getSource().equals(timer) && analysis == false){
588                 int newNote = ruleGenerator(val);
589 
590                 if (newNote >= 8){
591                     grid[0][board_size.width-1] = black;
592                     newNote = newNote-8;
593                 }
594                 else
595                     grid[0][board_size.width-1] = white;
596                 if (newNote >= 4){
597                     grid[1][board_size.width-1] = black;
598                     newNote = newNote-4;
599                 }
600                 else
601                     grid[1][board_size.width-1] = white;
602                 if (newNote >= 2){
603                     grid[2][board_size.width-1] = black;
604                     newNote = newNote-2;
605                 }
606                 else
```

```
607                     grid[2][board_size.width-1] = white;
608                 if (newNote >= 1){
609                     grid[3][board_size.width-1] = black;
610                     newNote = newNote-1;
611                 }
612                 else
613                     grid[3][board_size.width-1] = white;
614                 repaint();
615                 Color[][] newGrid = new Color[board_size.height]
    [board_size.width];
616             }
617 
618             //repaints the bottom line sequence based on rule
619             if (e.getSource().equals(timer) && analysis == true){
620                 ruleGeneratorAnalysis();
621             }
622 
623             //Start-Pause button processing
624             else if(e.getSource().equals(start_pause)){
625                 if(run){
626                     timer.stop();
627                     //JOptionPane.showMessageDialog(null,printResults());
628                     JOptionPane.showMessageDialog(null,printResults());
629                     start_pause.setText("Compose");
630                 }
631                 else {
632                     if (selected) {
633                         timer.restart();
634                         start_pause.setText("Terminate");
635                 }
636                 else {
637                         JOptionPane.showMessageDialog(null, "Must first
    select an epoch from which to compose");
638                         run = !run;
639                 }
640             }
641             run = !run;
642             }
643 
644             //Medieval button processing
645             else if(e.getSource().equals(medieval)){
646                 medieval.setEnabled(false);
```

```
647                 renaissance.setEnabled(true);
648                 baroque.setEnabled(true);
649                 classical.setEnabled(true);
650                 romantic.setEnabled(true);
651                 modern.setEnabled(true);
652                 changeEpoch("medieval");
653                 selected = true;
654             }
655             //Renaissance button processing
656             else if(e.getSource().equals(renaissance)){
657                 medieval.setEnabled(true);
658                 renaissance.setEnabled(false);
659                 baroque.setEnabled(true);
660                 classical.setEnabled(true);
661                 romantic.setEnabled(true);
662                 modern.setEnabled(true);
663                 changeEpoch("renaissance");
664                 selected = true;
665             }
666             //Baroque button processing
667             else if(e.getSource().equals(baroque)){
668                 medieval.setEnabled(true);
669                 renaissance.setEnabled(true);
670                 baroque.setEnabled(false);
671                 classical.setEnabled(true);
672                 romantic.setEnabled(true);
673                 modern.setEnabled(true);
674                 changeEpoch("baroque");
675                 selected = true;
676             }
677             //Classical button processing
678             else if(e.getSource().equals(classical)){
679                 medieval.setEnabled(true);
680                 renaissance.setEnabled(true);
681                 baroque.setEnabled(true);
682                 classical.setEnabled(false);
683                 romantic.setEnabled(true);
684                 modern.setEnabled(true);
685                 changeEpoch("classical");
686                 selected = true;
687             }
688             //Romantic button processing
```

```
689             else if(e.getSource().equals(romantic)){
690                 medieval.setEnabled(true);
691                 renaissance.setEnabled(true);
692                 baroque.setEnabled(true);
693                 classical.setEnabled(true);
694                 romantic.setEnabled(false);
695                 modern.setEnabled(true);
696                 changeEpoch("romantic");
697                 selected = true;
698             }
699             //Modern button processing
700             else if(e.getSource().equals(modern)){
701                 medieval.setEnabled(true);
702                 renaissance.setEnabled(true);
703                 baroque.setEnabled(true);
704                 classical.setEnabled(true);
705                 romantic.setEnabled(true);
706                 modern.setEnabled(false);
707                 changeEpoch("modern");
708                 selected = true;
709             }
710         }
711     }
712 
713     /*
714      * Method to play note value using MIDI synthesizer based upon input
    note
715      * @param int representing the MIDI value of desired note.
716      */
717     public void playNote(int i) {
718         try{
719         /* Create a new Synthesizer and open it.
720          */
721             Synthesizer midiSynth = MidiSystem.getSynthesizer();
722             midiSynth.open();
723 
724             //get and load default instrument and channel lists
725             Instrument[] instr =
    midiSynth.getDefaultSoundbank().getInstruments();
726             MidiChannel[] mChannels = midiSynth.getChannels();
727 
728             midiSynth.loadInstrument(instr[0]);//load an instrument
```

```
729             mChannels[0].noteOff(i);//turn off the previous note
730             mChannels[0].noteOn(i, 120);//On channel 0, play note number i
    with velocity 120
731             try {
732                 //Following line controls duration of notes played. 1000
    used for samples of 30 seconds. 750 used for samples of 15 seconds
733                 Thread.sleep(750); // wait time in milliseconds to control
    duration
734             }
735             catch( InterruptedException e ) {}
736         }
737         catch (MidiUnavailableException e) {}
738     }
739 
740     /*
741      * method that returns string that prints composition statistics for
    visual analysis
742      * @returns String statistics
743      */
744     public String printResults() {
745         return "Total length of composition: "+t+"\n"
746                 +"\tStatistics:\n"
747                 +"\nUnison:\t "+((double)totals[0]/t)
748                 +"\nStep:\t "+((double)totals[1]/t)
749                 +"\nThird:\t "+((double)totals[2]/t)
750                 +"\nForth:\t "+((double)totals[3]/t)
751                 +"\nFifth:\t "+((double)totals[4]/t)
752                 +"\nSixth:\t "+((double)totals[5]/t)
753                 +"\nSeventh:\t "+((double)totals[6]/t)
754                 +"\nOctave:\t "+((double)totals[7]/t);
755     }
756 
757     /*
758      * method that returns string that prints composition statistics for
    analysis
759      * @returns String statistics
760      */
761     public String kernResults() {
762         //variable to store percentage of most common interval
763         int max = 0;
764 
765         // computes the most common interval
```

```
766         for(int i = 0; i<8;i++) {
767             if(totals[i] > max){
768                 max = totals[i];
769             }
770         }
771 
772         //returns expected String output based on totals array and above
    computation
773         return ""+((double)totals[0]/t)
774                 +","+((double)totals[1]/t)
775                 +","+((double)totals[2]/t)
776                 +","+((double)totals[3]/t)
777                 +","+((double)totals[4]/t)
778                 +","+((double)totals[5]/t)
779                 +","+((double)totals[6]/t)
780                 +","+((double)totals[7]/t)
781                 +","+((double)max/t)
782                 +","+era;
783     }
784 
785     /*
786      * Method to clear the statistics after terminations for next
    composition
787      */
788     public void clearStats() {
789         //loops through all saved data and resets to 0 for future
    processing
790         for (int i = 0; i < 8; i++) {
791             totals[i] = 0;
792         }
793         t = 0;
794     }
795 }
```

References

[1] P.P. Wiener, *Dictionary of the History of Ideas. Studies of Selected Pivotal Ideas*. III, Chales Scribner's, 1973.

[2] A. Boethius, “Fundamentals of Music,” in Strunk’s Source Readings in Music History, ed. O. Strun, 1998.

[3] G. Niederhaus, *Algorithmic Composition: Paradigms of Automated Music Generation*. Vienna, Austria: Springer-Verlag, 2009.

[4] G. Diaz-Jerez, *Algorithmic Music: Using Mathematical Models in Music Composition*. The Manhattan School of Music, 2000.

[5] V. Duckles, et al, Musicology. *Grove Music Online*, 2001.

[6] J.D. Fernandez and F. Vico, "AI Methods in Algorithmic Composition: A Comprehensive Survey," *Journal of Artificial Intelligence Research.*, vol. 48, pp. 513-582, 2013.

[7] L.A. Hiller and L.M. Isaacson, “Musical composition with a High-Speed digital computer”. *Journal of the Audio Engineering Society*, 6 (3), pp. 154–160, 1958.

[8] J. Lebar, et al., ‘Classifying Musical Scores by Composer’, Stanford University, 2008.

[9] R. Basili, et al., ‘Classification of Musical Genre: A Machine Learning Approach’, University of Rome Tor Vergata, 2004.

[10] N. Tawa, Sheet Music. *Grove Music Online*, 2014.

[11] C. Anderton, ‘Craig Anderton’s Brief History of MIDI’, 2014. [Online]. Available: https://www.midi.org/articles/a-brief-history-of-midi. [Accessed: 01- Mar- 2018].

[12] D. Huron, “The Humdrum User Guide”, 1999.

[13] P. Tan, et al. *An Introduction to Data Mining.* Pearson Nueva Delhi (India). 2016.

[14] S.C. Suh, *Practical Applications of Data Mining.* Texas A&M University. Jones & Bartlett Learning, 2012.

[15] R. Hall, ‘Intervals and Pitches’ in *Sounding Number: Music and Mathematics from Ancient to Modern Times,* 2017.

[16] J. James, “Identifying and presenting eras of classical music”, from *Music Teacher,* 2017.

[17] T.M. Li, “Cellular Automata”, New York: Nova Science Publishers, Inc., 2011.

[18] S. Wolfram, “A New Kind of Science”, Champaign: Wolfram Media, Inc., 2002.