Music Generation and Evaluation Using R

By
Andrea Lanz

A THESIS

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Mathematics
(Honors Scholar)

Presented March 5, 2020
Commencement June 2020

AN ABSTRACT OF THE THESIS OF

Andrea Lanz for the degree of Honors Baccalaureate of Science in Mathematics presented on March 5, 2020. Title: Music Generation and Evaluation Using R

Abstract approved:

Charlotte Wickham

The main goal of this project is to implement an algorithmic music composition procedure and present all procedures and results in a reproducible fashion by creating a web application produced by R. To approach the implementation of the algorithmic composition of music, we start by exploring various models, but ultimately we sample musical notes using a stochastic approach, specifically by employing a Hidden Markov Model using methods found in [15]. The process of music generation and evaluation will be documented and made reproducible by the development of a complementary web application to allow for easy visualization, verification, and widespread use of the developed methods.

Key Words: Hidden Markov Model, algorithmic composition, R Shiny

Corresponding e-mail address: lanza@oregonstate.edu

Music Generation and Evaluation Using R

By
Andrea Lanz

A THESIS

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Mathematics
(Honors Scholar)

Presented March 5, 2020
Commencement June 2020

Honors Baccalaureate of Science in Mathematics project of Andrea Lanz presented on March 5, 2020

APPROVED:

Charlotte Wickham, Mentor, representing Statistics

Vrushali Bokil, Committee Member, representing Mathematics

Sarah Emerson, Committee Member, representing Statistics

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University Honors College. My signature below authorizes release of my project to any reader upon request.

Andrea Lanz, Author

Contents

1 Introduction

To approach the generation of a novel piece of music, whether the music created inherits from certain composer, genre, or style, will depend on the method chosen. There are many models for generating music, beginning with a model as simple as using a roll of a die [4] to determine the next musical note, moving on to simpler monophonic (single-noted) Markov models up to more advanced Hidden Markov models, and even methods that implement machine learning. Algorithmic music composition is a large field of study, with its history beginning even before the existence of modern computers. This field of study has been highly explored, with a variety of styles of music being generated. This paper will explore the topic of algorithmic composition in four sections. We begin by providing a brief summary of the field of algorithmic composition as well as this project's selected method for generating music, chosen data set, and the reasons behind these decisions in section 2. Using methods from [15], we next describe our chosen approach to generate music in section 3. Employing this approach, the music generation process is streamlined into development and deployment of a web application in section 4. Finally the challenges of our project and future work are summarized in 5.

2 Background

When exploring the topic of music generation and implementing a model, different aspects of this implementation must be considered, such as the specific model that will be used, a measure or technique used to evaluate the resulting music, and musical data to build the model. This section will explore these topics for various composition models available.

2.1 Models

As there are a variety of unique methods for generating music, this paper will describe different applications of the Markov model as well as providing a description for the machine learning method. Different models may be used to achieve different goals, so one must consider the benefits and drawbacks of a model before deciding on which to implement.

2.1.1 Markov Models

The most basic form of a Markov Model used to generate music is monophonic, meaning notes are sampled and played one at a time. This model includes generating a transition matrix from the data set to create a new piece similar to the existing pieces. Note that this method does not allow for chords or for generating more complex aspects of music.

One Markov model of interest is the Hidden Markov Model (discussed in more

detail in section 2.5), which in [4] is described as a process using two different HMM's, one in which the observed state are the chords with hidden harmonic symbols being determined, and another where the observed harmonies are the observed state and the chords are the predicted hidden states. HMM's are also implemented in [2], where the visible state is the melody notes, and the hidden state is the chords, similar to [4] approach. In [2], for a more simple HMM, is it noted that this method of generation does not take into account the flow of music and may also suffer from sparse data, where significant characteristics of the data set are not seen in the generated results. As mentioned in [5], an additional decoding step can be applied, using the Viterbi Algorithm, for finding the most likely hidden state sequence.

2.1.2 Machine Learning

One method of machine learning applied to music generation works by building a model that is learned from the data, which is raw audio (as opposed to MIDI which is a symbolic representation) [7]. Regarding the specifics of generating a piece, waveforms are generated directly, which rely on an autoregressive models. Methods of machine learning can even go as far as to implement autoencoders, which is a type of neural network in machine learning. Specific to the findings in [7], one aspect of this model is the need for a balance between modeling local and large-scale structure, which can be applied to most music generation methods in general.

2.2 Evaluation

[8] discusses how evaluating generated music depends on the goal. Evaluating a song using human listeners might measure the quality of a song, while theoretical rules in music or machine learning techniques address whether a generated song is of the same class as another.

With the goal of producing music similar to a chosen class of music, one difficulty in evaluating a piece is the existence of multiple musical attributes such as pitch, duration, and onset time and different levels such as rhythmic, melodic, and harmonic [6]. Note that one method of addressing this complexity is to apply a multiple viewpoints model, where sequences of events (which include characteristics such as pitch, duration, and onset time) is transformed into a more abstract sequence after applying viewpoints [6].

Another method of evaluation, described in [2], includes comparing the studied model to a simpler, already-existing model by using measures of harmonization and fit to the training data. Note that this method addresses the goal of generating music more similar to a chosen class, rather than measuring musical quality.

2.3 Training Data

Most methods for generating music require an existing sample of music used as "training" data. For Markov models specifically, this data is required to construct transition matrices.

The type of data set to use depends on the goals of the project. If the goal is to generate music from a specific genre or composer, a data set of music fitting this style of music is required. In other studies, a single instrument is focused on. Note that when selecting a data set, raw data is much harder to use, since it is difficult to capture the high level structure of music [7]. More simple samples of music may also be used, as opposed to raw data, as illustrated by [3], where 37 common meter hymn tunes were used in a simple Markov model.

2.4 Generating Music Using a Simple Markov Model

One of the most basic methods of music composition includes randomly sampling from a selection of notes. This method does not generate very impressive results, as expected; however, this method can be developed further by modeling the sequence of notes as a simple Markov Model. In a previous project, I explored this idea further: "Simulating Music Using R", which can be found at `https://github.com/ST541-Fall2018/andrealanz-project-musicsim`. This project included defining a random sampling method for music generation that was then developed into a simple Markov Model.

2.4.1 Methods and Results

In this previous project, the general process used for generating a song included implementing functions that generate a sample of notes, then implementing another

function that outputs a playable file. This project employed three basic music generation models: a simple random sample of quarter notes, a simple random sample of notes with random note length, and finally a simple Markov Chain model based off of the C major scale. In this third method, Markov chains were used to sample notes by incorporating an 88 x 88 transition matrix (since pianos have 88 keys). Note that this method was just focused on the C major scale to provide consonance in the generated music.

The project used the LilyPond file format, discussed in more detail later, which is a compiled text program that converts text to an audio file as well as into sheet music. The compatibility of R and the LilyPond file format was made possible by the 'tuneR' package, allowing notes to be sampled and written to LilyPond using R.

The project produced a way to sample notes in the form of wave objects and generate playable files. A form of rhythm was also implemented using random sampling, and consonance was achieved by implementing relationships between notes using the concept of scale. In terms of the songs produced, the first method of random sampling of notes was not successful, as there was no way to ensure consonance. Combining Markov chains and sampling for variation in speed allowed for more pleasant song results.

2.5 The Hidden Markov Model

As a continuation of my last project, which utilized the simple Markov Model, the natural progression of this project includes moving towards a more sophisticated model: the Hidden Markov Model. This section begins by defining a basic Hidden Markov Model and describes some examples of its application in music. As defined in [10], let $\{X_n, n = 1, 2, \ldots\}$ be a Markov chain with transition probabilities $P_{i,j}$ and initial state probabilities

$$p_i = P\{X_1 = i\}, i \geq 0$$

Suppose that there is a finite set $\mathcal{P}$ of signals, and that a signal from $\mathcal{P}$ is emitted each time the Markov chain enters a state. Further, suppose that when the Markov chain enters state j then, independently of previous Markov chain states and signals, the signal emitted is s with probability $p(s|j)$, where

$$\sum_{s \in \mathcal{P}} p(s|j) = 1$$

That is, if S_n represents the nth signal emitted, then

$$P\{S_1 = s|X_1 = j\} = p(s|j),$$

$$P\{S_n = s|X_1, S_1, \ldots, X_{n-1}, S_{n-1}, X_n = j\} = p(s|j)$$

A model of the preceding type in which the sequence of signals $S_1, S_2, \ldots$ is observed, while the sequence of underlying Markov chain states $X_1, X_2, ...$ is unobserved, is called a hidden Markov chain.

In [9], the author describes that given an observation sequence, $O = (S_1, S_2, \ldots)$, there are three basic problems:

1. How can the probability of this sequence be determined?

2. How can the optimal state sequence be determined?

3. How can the model $\lambda = (A, B, \pi)$ maximize $P(O|\lambda)$, where A, B, and π are the state distribution, observation distribution, and initial distribution respectively?

The "Forward-Backward" procedure is used to answer problem 1, since calculating the probability of all possible state sequences given an observed sequence is unfeasible (note that this procedure is also described in [10]). Problem 2 can be solved with the Viterbi algorithm described in [10]. The third problem is where "training" the model occurs, where a training sequence is an observed sequence that is used to adjust the model λ and its parameters. The paper notes the Baum-Welch method (or the expectation-modification method) to locally maximize $P(O|\lambda)$ or use gradient techniques. The paper describes the Baum-Welch method, which is a complex optimization problem.

2.5.1 Applying the HMM to Music

Now that we have introduced Hidden Markov Chains, we can now discuss how they have been applied to algorithmic music composition. In this section, two applications of the Hidden Markov Model (HMM) will be discussed.

In [2], the author uses the HMM to compose chorale harmonizations, which "sound pleasant when played simultaneously with the original melody". The author uses melody notes as visible states and chords as the hidden states. They note that chorales have underlying harmonic structure, where notes can be selected that are compatible with this harmonic state. Separate models were used for different minor and major keys. Their model predicts three notes at each time step, one for each musical line. They represent the hidden state as a list of pitch intervals. This model is used to produce harmonizations, not to generate an original piece, so this model was not adopted for this project.

In [12], the authors define a model (SuperWillow), similar to the HMM, that is a modification of Willow, which is a "rule-based, music-composition system that works on the premise that the composition process can be modeled by tree operations on ranked trees". Their model includes a filter for the music data (e.g. they remove pieces with a change in time signature) and changing the music to the key of C. Note that the SuperWillow program takes chord and duration data from the music sample, MC and HMMs are applied, then the analyzed data is grouped by style. The composition process of this paper is separated into steps such as chord progression, melodic curve,

cadence, etc. Markov chains are used in chord duration, chord progression, and rhythm progression. They used time-independent transition probabilities estimated using empirical counts and calculating the likelihood estimate.

This model uses prediction suffix automatons (PSAs), which are equivalent to mixed-order Markov Chains (MCs). The LearnPSA algorithm is used to train the PSA. The data extracted from the music files include: tempo, scale, time signature, then MCs are applied to chord duration, chord progression, and rhythm progression, where the chords themselves are represented by roman numerals. The durations are found by grouping chords next to each other and adding their durations. The training data is found by extracting frequency counts from the data. A HMM is applied to the melodic arc, where the "hidden alphabet" includes the index of the melody note and how it's related to the previous note, indicated by +, -, or =. The melody note is then matched to an accompaniment chord in the observed sequence. Note that the melody note is the right hand note, and the accompaniment chord is played by the left hand.

In summary, this process works by getting a style from the user, then the tempo, time signature, and scale are chosen based on the user's selection. Chord duration and rhythm use MC's to generate a sequence of notes that sum to a specified duration. Carmel, a transducer package, is used to produce the rhythm and chord progression. This method of composition is out of scope for this project, due to its complexity.

3 Methods

This primary goal of this project is to replicate the approach of Yanchenko & Mukherjee [15], so their approach is described in some detail. Finally, the details specific to the implementation of my work, the data used, and types of file used to represent music, are discussed.

3.1 Classical Music Composition Using State Space Models

This results of this project depend on the model defined in an article titled "Classical Music Composition Using State Space Models" authored by Anna K. Yanchenko and Sayan Mukherjee [15]. The purpose of their project is summarized by exploring state space models, such as the Hidden Markov Model and variations of this model, in the algorithmic composition of classical piano pieces from the Romantic Era. They argued for narrowing their sample down to the Romantic Era (a period spanning most of the 19th century) since this era's style makes calculating and comparing metrics to evaluate the generated pieces much simpler; according to [14] (noted in [15]) the music in this era "tended to value emotion, novelty, technical skill, and the sharing of ideas between arts and other disciplines". Their project concluded that the state space models were "fairly successful in generating new pieces that have largely consonant harmonies" [15], where their methods included training an HMM (and variations on this model) using classical pieces with simple harmonic structure from the Romantic Era. Note that their biggest issue with this model is the lack of melodic progression,

meaning a lack in global structure of a generated piece.

Historically, HMM's have been used for mostly classification instead of music composition, so their project served to further develop the model for this purpose. As discussed in section 2.1.2, there are alternative models to the HMM for algorithmic music composition, specifically recurrent neural networks (RNNs). The HMM was chosen due to its ease of implementation and capability to model complex pieces. The works that investigate these models are discussed in more detail in section 2.1.2 as well as in [15].

3.1.1 The Model

There are 15 models defined in [15] increasing in complexity, but this project focused on the first-order HMM. To define the model, we first define a composition as it is in [15], which is a sequence of pitches associated with time steps $X_1, \ldots, X_T$ (where the time step is typically a sixteenth note or eighth note). Each element of the sequence is a pitch represented as an integer ranging from 0-127. The elements of this sequence are the observed states of the HMM. As defined in [15], the likelihood for a first-order HMM is defined as:

$$p(x_{1:T}|z_{1:T}, \theta, \theta', \pi) = \pi(z_1)p_\theta(x_1|z_1)\prod_{t=2}^{T} p_{\theta'}(z_t|z_{t-1})p_\theta(x_t|z_t) \tag{1}$$

where $z_{1:T}$ are the hidden states, the parameters θ and θ' are the emission probabilities ($p_\theta(x|z)$), and the transition probabilities ($p_{\theta'}(z_t|z_{t-1})$) respectively, and π is the initial

distribution.

As used in [15], this project will be using a first-order HMM with 25 hidden states. Each model in [15] is trained using a sample of ten piano pieces from the Romantic era. To train the model, the Baum-Welch algorithm, discussed further in Section 3.2, is used to estimate the HMM parameters. To do this, each sample has its notes assigned a time-stamp, and to simplify, the model assumes the observed notes are equally spaced in time (e.g. the notes are treated as a univariate time series with notes of chords considered sequentially).

3.1.2 Quality Metrics

Harmony and Melody A large portion of [15] includes evaluating the quality of generated pieces. The authors use the concepts of harmony and melody to define informative metrics of musical quality. As defined in their work, harmony is "the arrangement of chords and pitches and the relation between chords and pitches", where melody is defined as "a sequence of tones that are perceived as a single entity and is often considered a combination of pitch and rhythm". Another integral concept in defining useful quality metrics is the musical interval, which is the difference in pitch between two notes. There are two types of intervals: harmonic and melodic. The harmonic interval is defined as the interval of two successive notes, whereas the melodic interval is defined as the interval of two simultaneous notes. Note that either can either be consonant or dissonant. Dissonant intervals are characterized

by sounding incomplete and needing a resolution of a consonant interval. Dissonant intervals have the effect of building tension, which is often resolved in Romantic Era pieces [15].

Metrics and Results There are three different categories of metrics used to evaluate the generated pieces in [15]: originality, musicality, and temporal structure. The originality metrics test for how predictable the generated pieces are; the musicality metrics measure the amount of dissonance using the generated harmonic and melodic intervals as well as note distribution; and the temporal structure metrics measure the correlation of the notes.

To use the calculated metrics for quality of a generated piece, the root mean squared error was calculated for most of the metrics on a sample of 1000 generated pieces. This process resulted in generally favorable results for each training piece tested. The authors used these metrics in conjunction with human listening studies to determine that the pieces generated were more successful with harmony than melody, and that most pieces lacked a global melodic structure.

3.1.3 Relevance to this Project

This project will implement the methods as described in section 3.1.1, the first-order HMM using 25 hidden states. The same training data will be used as in [15] in addition to Romantic Era piano samples from a different source (discussed in more detail in section 3.3). Although [15] provides a method for measuring the quality of

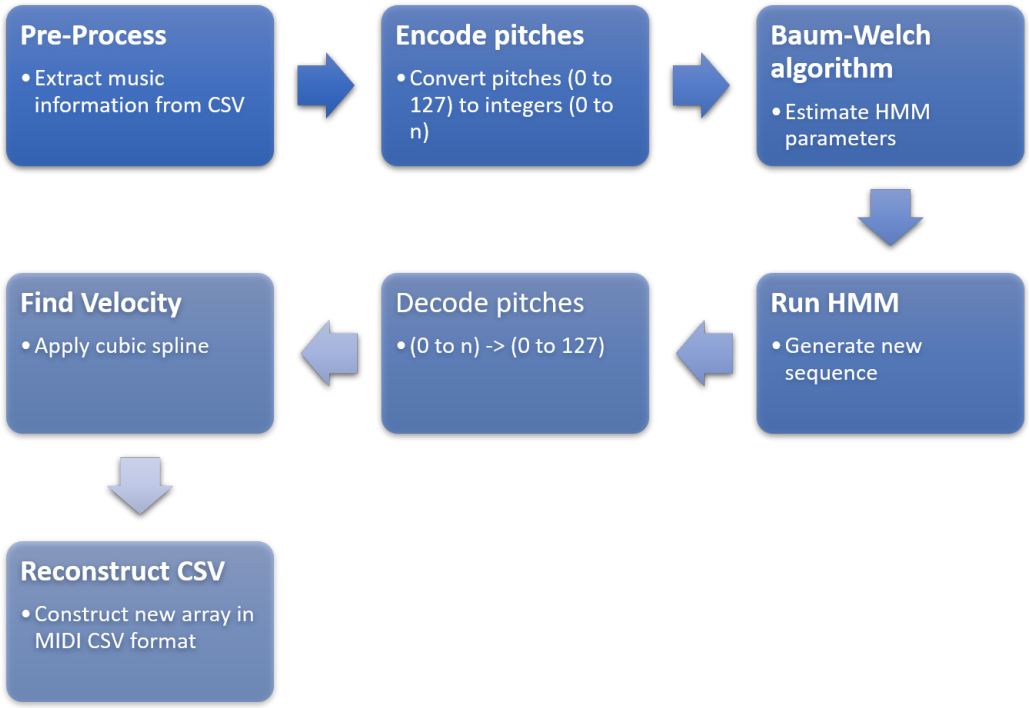

Figure 1: The algorithm steps summarized.

results, this was not applied in this project.

3.2 The Algorithm

Combining the concepts previously discussed and using the implementation by [15] (where the project code can be found at `https://github.com/aky4wn/Classical-Music-Composition-Using-State-Space-Models`), the process of implementing this model is described below, where the input to this algorithm is a music CSV generated from an outside program. This CSV contains musical data about a given piece, which is discussed further in section 3.4. Refer to figure 1 for a summary of the algorithm. The steps of the algorithm behind this model are detailed below:

HMM_Compose:

1. **Pre-process:** This step includes translating the data stored in a music CSV into a form for easy use in the model. Specifically, this step extracts the CSV data into an array, where the notes data can be easily accessed. Note this also makes music information such as key, measures, etc. more accessible.
2. **Encode pitches:** In a music CSV, the notes are represented as pitches ranging from 0-127. This step encodes the pitch values as integers ranging from 0 to the number of unique pitches. This step makes the modeling process easier when the encoded pitches are used as the observed states in the HMM.
3. **Baum-Welch algorithm:** This well-known algorithm is applied to the encoded frequencies to determine the HMM parameters, which is an application of the Expectation-maximization algorithm, which finds the maximum likelihood estimate of the HMM parameters. This algorithm is used to estimate parameters θ, θ', π found in equation 1, which represent the emission, transition, and inital distributions respectively. A more detailed explanation and derivation of this algorithm can be found at [11].
4. **Use estimated parameters to run HMM:** The parameters estimated by the previous step are now used to generate a sequence of new notes (i.e. encoded pitches) using equation 1.
5. **Decode pitches:** The pitches now decoded in the same fashion as they were encoded: going from an integer ranging from 0 to the number of unique pitches

to an integer from 0-127, representing a specific pitch as defined by MIDI.

6. **Find velocity:** Velocity is not modeled in this implementation, so as a substitute, a cubic spline is used on the original velocity data to produce a set of new velocities. The reasoning behind this is not clear, and there is no justification in [15]. Velocity is simply the volume of given note, so fluctuations in velocity will not change a song significantly, so at first glance, this strategy seems acceptable.
7. **Format array of notes to convert to CSV:** Finally, using the new notes and velocities, a new array is constructed, which conforms to the MIDI CSV guidelines. Thus, a CSV representing a newly generated song based on the training sample is created, and can be now converted to MIDI.

3.3 Data

All of the training samples used in [15] were used as model data. Note that these samples are MIDI files found at `https://mfiles.co.uk` and `https://midiworld.com`. In addition to the resources from [15], an additional set of data was used from a midi repository at `http://piano-midi.de` with classical piano pieces from the Romantic Era. These pieces were also used to test the algorithm (as defined in section 3.2) with mixed results (e.g. various errors) that are discussed further in Section 5.1.

3.4 File Types

As mentioned in Section 3.2, the input to the HMM algorithm is a CSV file, but the source music (Section 3.3) is in MIDI format. This section will discuss different ways music can be represented digitally, including these formats.

There are two general approaches to representing music digitally. One approach is an audio encoding format, that is, an actual recording of sound, while the other represents the underlying musical information, such as the notes to be played, the tempo, etc. Both types of musical representation are useful in the process of generating new music. The methods used in this project to generate new music rely on definitive data about a musical piece, where a file format containing musical information might be more useful. However, for the purpose of making the results of this project more accessible, a file format that contains actual audio may be more useful.

Of the file formats with actual audio recording, two file formats specifically are most relevant to this project: MP3 and WAV. The difference between these two formats is whether the file is compressed or not (MP3 is compressed, while WAV is not). Regarding the file formats containing actual note information, two types are most relevant to this project: LilyPond and MIDI (Musical Instrument Digital Interface). Most importantly, MIDI is a file type that standardizes music information by providing a communication protocol between musical devices. This file type contains information about the music (not a recording of the music itself).

Lilypond (.ly) is a compiled text program that converts text to MIDI as well as

into sheet music. The project discussed in section 2.4 utilizes the LilyPond notation as a means to generate new music using a simple Markov Model. This project began using LilyPond in an attempt to use the HMM to build a LilyPond piece from scratch. The project began by using an original .ly, which was converted to MIDI in order to apply the model, where the transformed data was converted back to .ly. Note that converting MIDI to .ly is not recommended for human-generated MIDI files, since this conversion process has accuracy issues due to the complexity of the LilyPond notation. The methods attempted did not work for polyphonic pieces, even though there was partial success in converting monophonic pieces to MIDI. LilyPond's notation is very specific and difficult to generate, which is why MIDI is a more appropriate file format for this project.

3.4.1 MP3 and WAV

As mentioned, while note information is useful in a file format when generating new music, audio recordings make playing generated results much simpler. Specifically, the program used in this project to generate music will produce results in the MIDI format, but MIDI players do not exist on many systems. To play the results in a web browser, it is much easier to use MP3 or WAV file types. Converting from MIDI to MP3 or WAV is not trivial, and the conversion process will be discussed in more detail in section 4.2.

3.4.2 MIDI to CSV

One benefit to the structure of a MIDI file is that it allows for reliable conversion to the CSV (comma-separated values) format — a plain text format, easily imported and manipulated in many programming languages. This allows for the note information that a MIDI stores to be easily saved in a tabular format. Note that this also allows for easy conversion in the reverse direction, from CSV back to MIDI (given the correct structure of CSV), which makes this format useful when editing or generating musical information. This conversion is supported by a program found at (`https://www.fourmilab.ch/webtools/midicsv/`). For these reasons, the MIDI format makes for an appropriate file format for reading in music, applying a model, and generating new music.

Required fields Once the source music for the model is converted from MIDI to CSV, the structure of the CSV must be maintained in order to successfully convert back to MIDI, allowing the results to be playable. The basic structure of a CSV storing MIDI data is described below. Note that this basic structure is defined in more detail at [13].

Each record (line) in the CSV must have values for three fields:

- **Track:** A number indicating the current track of the record. Typically track 0 holds header information.
- **Time:** The time that the record occurs in units of MIDI clocks. Time 0 is used

for header or meta-events (events with no note data).

- **Type:** The type of record (e.g. 'Header', 'Start_track', etc.).

Other fields may be required depending on the value in the "Type" field.

Basic File Structure Going into more detail, we next describe a simplified version of the structure of records in a CSV generated from a MIDI.

Track	Time	Type			
0	0	Header	[format]	[number of tracks]	[division]
[track]	0	Start_track			
[track]	[end time]	End_track			
0	0	End_of_file			

The first line in the CSV file always has a type of "Header", and the last line type "End_of_file". The "Header" row contains information relevant to the entire piece of music:

- **[format]:** contains the type of MIDI (either 0, 1, or 2). Midi file type 0 contains a single track, type 1 contains multiple tracks played simultaneously, and type 2 contains multiple independent tracks.
- **[number of tracks]:** indicates the number of tracks in the file.
- **[division]:** indicates the number of clock pulses per quarter note.

Additional file level information may also be stored in Meta-Events.

Lines with type “Start_track” and “End_track” denote the start and end of tracks, with the lines between them describing the events that occur on the track.

Common Channel Events Actual note information is encoded in channel events — lines between those with types “Start_track” and “End_track”. The most common channel events are those with types:

- **Note_on_c:** Plays a note with the given specifications.
- **Note_off_c:** Stop playing the note with the given specifications.

Additional fields for these events include:

- **[note]:** assuming the notes are based on a 88 note piano, this is a numeric value ranging from 21-108 where middle C is represented as 60. This scale is illustrated in the Figure 2 below.
- **[velocity]:** a numeric value ranging from 0-127 indicating the volume of the note.

Common Meta-Events Meta events describe other file level information. Some common types are:

- **Title_t:** Indicates the title of the given track.
- **Time_signature:** Indicates the time signature of the file, a measure of rhythm, given in the number of 32nd notes per quarter note (24 MIDI clocks).
- **Key_signature:** Indicates the key signature of the file, giving information

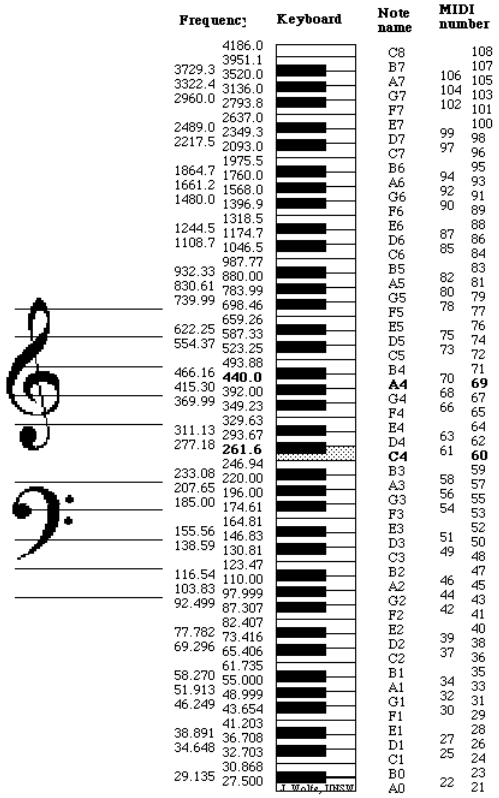

Figure 2: MIDI note names [1]

0	0	Header	1	2	256
1	0	Start_track			
1	0	Time_signature	4	2	24
1	0	Key_signature	2	major	
1	0	Tempo	483869		
1	0	Title_t	"Ode to Joy"		
1	0	Text_t	"Ludwig van Beethoven"		
1	1	End_track			
2	0	Start_track			
2	0	Note_on_c	0	50	63
2	256	Note_off_c	0	50	0
2	257	End_track			
0	0	End_of_file			

Figure 3: Example CSV representation of a MIDI file

about the sharp and flat notes in a piece. Note that the value for the key ranges from -7 to 7 where the key of C is represented by 0 and negative numbers indicate the number of flats below C or positive for the number of sharps above C.

- **Tempo:** The tempo of the song in number of microseconds per quarter note, in the range of 1-16777215.

Consider Figure 3 for a simple example of a CSV that could be generated from a MIDI. This example is a simplified version of the CSV generated from the "Ode to Joy" MIDI file found at `https://www.mfiles.co.uk/classical-midi.htm`. This example represents a MIDI file that plays a single note.

In relation to applying the HMM using this MIDI-CSV representation, the model will only be applied to the note events (i.e. the channel events), where the meta-events and header information will remain the same between the training sample and

the generated sample.

4 Results

Given we have a working program sourced from [15], the main deliverable of this project is a web application providing a user interface for easy conversion of simple MIDI piano pieces. This app also provides accessibility to the model (or algorithm) defined in this project, making algorithmic composition straightforward. This app can be found at `http://ec2-54-201-39-45.us-west-2.compute.amazonaws.com/hmm_player`. The app code can be found at `https://github.com/andrealanz/hmm_player`. As an overview, the app accepts an upload of a MIDI file and runs the HMM_Compose backend script after conversion to CSV. The resulting CSV is then converted to MIDI, then finally to WAV to be played in the browser. Depending on the complexity of the song, this process may have a considerable wait time. This section will cover the details of building this app using R, how multiple file types and programming languages were handled, and finally the different deployment options.

4.1 R Shiny

The language of choice for this app is R, which is a programming language typically used in statistical computation and analytics. Specifically, R Shiny is a package that makes interactive web application development and deployment simple, with no need for knowledge of any other web development languages (e.g. html, javascript). The

architecture of a Shiny app includes a file for the server of the app, which controls the backend functions of the app, and a file that defines the user interface of the app. The functions that handle the HMM_Compose algorithm as well as the conversion steps are in the server file of the project. In addition to these two files, a global declarations file is also included to load external functions and packages into the app. More detail on what goes into a Shiny app can be found at `https://shiny.rstudio.com/tutorial/`.

4.2 Handling File Conversion

The nature of this project requires multiple file types and conversions. All music samples are in MIDI form when uploaded to the app, but the generation algorithm accepts CSV as input. As discussed in section 3.4, the MIDI format contains note information rather than audio recording, making a CSV conversion possible. An open source project found at `https://www.fourmilab.ch/webtools/midicsv/` provides programs, written in C, to convert between MIDI and CSV. Note that this project was also utilized by [15]. To use this program in our app, the source code was compiled to produce an appropriate executable. R functions were then created to run these executables by calling a system command and passing in a file path for the MIDI/CSV to be converted.

The purpose of the web application is to provide a user interface to the HMM_Compose algorithm. This includes making results of the model more accessible, specifically,

having the functionality of playing audio within the app. As discussed before, MIDI does not contain any audio recording, so in order to play this type of file, the note information needs to be synthesized. This describes a conversion from MIDI to a format that stores audio recording (WAV in this case). Since conversion from MIDI to WAV is not trivial, an external program must be used. The program of choice is FluidSynth (`http://www.fluidsynth.org/`), which is an open source MIDI synthesizer which requires a SoundFont file for conversion. A SoundFont file stores recordings of instruments that map to notes. The SoundFont used in the web app was found at `https://freepats.zenvoid.org/SoundSets/general-midi.html`. In order to implement this in the app, an R function was created to make a system call to the FluidSynth executable, passing in file path information as well as the specified SoundFont.

Thus, we now have methods for converting MIDI to CSV, CSV to MIDI, and MIDI to WAV to be played in-app. Consider Figure 4 for an illustration of the app's workflow.

4.3 R Compatibility

R is highly compatible with multiple programming languages. Noteworthy packages include 'reticulate', for Python and 'Rcpp' for C++ for the purpose of integrating different languages in R. See `https://cran.r-project.org/web/packages/reticulate/index.html` and `https://cran.r-project.org/web/packages/Rcpp/`

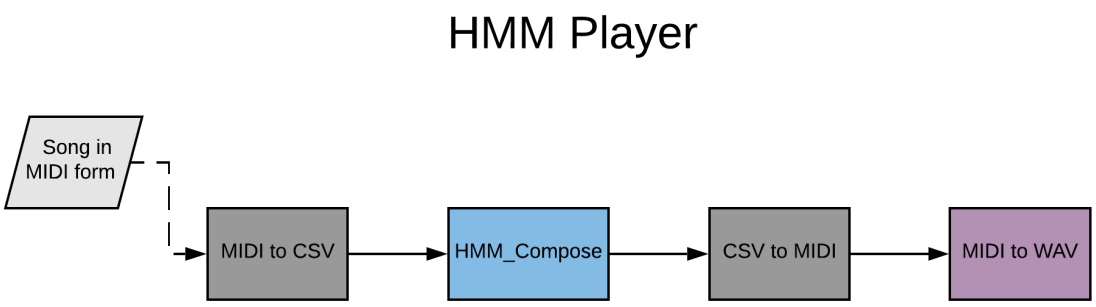

Figure 4: The web app workflow, where gray indicates the CSV/MIDI conversion in C, blue indicates the model Python script, and purple indicates the FluidSynth conversion.

`index.html` for more information about these packages and their uses.

The program provided by [15] is written in Python, so an interface between Python and R is required for this app. As mentioned, 'reticulate' provides an interface for running Python in R. Leveraging this feature of R, the source code from [15] can be run, where the HMM_Compose function can be called within the R environment, and more specifically in the Shiny app's server function.

It is worth noting that the source code for converting between MIDI and CSV is written in C, and R provides the functionality for running this code; however, this was not achieved in this project. This is a capability of R and can be implemented in future projects.

4.4 Deployment Options

There are multiple options for the deployment of a Shiny App. The simplest method for deployment is in the RStudio hosted Shinyapps.io in the cloud. It is straight-

forward and quick to get a Shiny app up and running using this option; however, due to the fact that our app depends on Python3, a versioning issue occurs within Shinyapps.io, so another option was pursued.

An alternative deployment plan is to use Shiny Server, also provided by RStudio, which allows for "on-premise" deployment of an app. It is not ideal to run this app on a local machine, so AWS Elastic Compute Cloud (EC2) service was leveraged. A t2.micro instance, the smallest instance available, was spun up. Using SSH, this instance can be connected to. The following steps were ran in order to successfully deploy the Shiny app:

1. Install R, Shiny Server, Python3, and FluidSynth
2. Install required R and Python3 packages
3. Copy over app files into server directory
4. Configure the app

See `https://tm3.ghost.io/2017/12/31/deploying-an-r-shiny-app-to-aws/` for more detailed steps for deploying a Shiny app in an AWS EC2 instance.

5 Conclusion

Using a first-order HMM and an algorithm defined by [15], we were able to create a web application built in R, where the backend file conversions and model were handled by R. This app streamlines the process of composing a new musical piece using the first-order methods described in [15]. As discovered by [15], the models

developed were shown to produce pieces that sounded human-generated, although maybe not from the Romantic Era; however, these pieces did not display the global harmonic structure of a human-composed piece. In addition, the resources provided by [15] were not incredibly reproducible. Also, the application's functionality is not completely extensive. To conclude this project, the aspect of reproducibility of the resources by [15] as well as future work will be discussed.

5.1 Reproducibility

As this project utilizes the results of [15], this project is dependent on their provided resources, which included the project code. Although their project provided substantial documentation, running their program successfully proved to be difficult: the reproducibility of their project could be improved. The obstacles encountered in attempting to reproduce their project are summarized here.

The first obstacle faced when attempting to reproduce the results of [15] is the various runtime errors. Their program relied on the same open source MIDI/CSV conversion tool as this project; however, the CSV produced by their algorithm resulted in an error in the CSV field where either "major" or "minor" is specified. Further debugging revealed a simple bug in the array that is converted to CSV within the Python script. The next error encountered was related to events being out of order within the outputted CSV. This revealed an error in the construction of the MIDI array within the script: the time field of the CSV must be in sequential order. After

further debugging, the issue was occurring due to non-note events in the CSV array (e.g. 'Control-c' or 'Pitch-bend-c'). The model is not applied to these events, so they remained in the CSV array that is output. Removing these notes does not seem to make a difference in the generated song, but remediates the runtime error. Another error encountered is how each new note's velocity is determined. As discussed in section 3.2, a cubic spline is used as a model for the generated velocities. Velocity must range from 0-127, but an error occurs due to spline overestimation for edge values: some velocities generated are out of range. To address this issue, a hard cap was set on the velocity as a quick fix (this issue may be worth revisiting in future work). The final error encountered involves an error when there are multiple tracks in a CSV that contain note information (typically more than three). This error was fixed by accepting all tracks that contain note information (e.g. if right and left hand are separated into two tracks), essentially applying the model to all tracks at once. This may be worth investigating in future work, especially for higher order models.

An additional obstacle faced is the difficulty in finding the source of the training samples. There are multiple links included in [15] that contain sample music with the same titles as the training samples; however, when run in the algorithm after conversion to CSV, the errors mentioned above occur, whereas the training CSVs provided by [15] run without issue. This indicates a discrepancy in the source provided and the actual training data, illustrating another issue with reproducibility.

In summary, there are aspects of the resources provided by [15] that make repro-

ducing their results difficult. More thorough documentation on expected input and output of the program and where the training data is found (and possibly how it was transformed) would be helpful. Additional testing of the program for debugging purposes is also recommended.

5.2 Further Study

The first area of potential study is in the functionality of our web application. One aspect of the app that is limited is the models supported. This project focused on the first-order HMM, but [15] investigated music composition for 15 models, all variations of the HMM. The more complex models investigated (e.g. the layered HMM) produced more successful results. Thus, the web app could be expanded in the future to support the more successful versions of the HMM. Note that [15] found limited global melodic structure in general and suggested that an alternative algorithmic composition model would be more ideal for this, specifically hierarchical models, natural language models, or recurrent neural networks. Thus, the app could be further developed to support completely different models for music generation in future. Another aspect of the app that could be further developed is its performance. The nature of the algorithm described in this project leads to long running times, leading to lag in our application. Either this load time could be reduced, by producing a faster algorithm, or finding a way to communicate to the user the expected load time. In addition to the load time, the app can be further developed to display more

information about the results generated (e.g. display sheet music or song metrics).

The HMM algorithm, as well as the metric analysis, were provided by [15] as Python code. Another area of the project that could lead to future work is utilizing R for these steps. R has the capability of running these types of models as well as providing useful data analysis tools to measure the quality metrics discussed in [15]. Instead of using R as the middleman for the web app, some of the algorithm's processes can be implemented directly in R. Even further, R Shiny can be used to display this quality analysis in-app, since Shiny makes it easy to embed R generated figures.

Overall, this project's work provides a way to easily compose music using the HMM with an easy-to-use web application. The music produced comes close to human-composed music, and our app has the required functionality, but our results can be improved with more app development, alternative models, and more integration with R.

References

[1] Note names, midi numbers and frequencies. https://newt.phys.unsw.edu.au/jw/notes.html.

[2] Allan, M., and Williams, C. Harmonising chorales by probabilistic inference. In *Advances in Neural Information Processing Systems 17*, L. K. Saul, Y. Weiss, and L. Bottou, Eds. MIT Press, 2005, pp. 25–32.

[3] Brooks, F. P., Hopkins, A. L., Neumann, P. G., and Wright, W. V. An experiment in musical composition. *IRE Transactions on Electronic Computers EC-6*, 3 (Sep. 1957), 175–182.

[4] Collins, T., Laney, R., Willis, A., and Garthwaite, P. H. Developing and evaluating computational models of musical style. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 30*, 1 (2016), 16–43.

[5] Conklin, D. Music generation from statistical models. In *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences* (2003), pp. 30–35.

[6] Conklin, D. Multiple viewpoint systems for music classification. *Journal of New Music Research 42*, 1 (2013), 19–26.

[7] Dieleman, S., van den Oord, A., and Simonyan, K. The challenge of realistic music generation: modelling raw audio at scale, 2018.

[8] Herremans, D., Chuan, C.-H., and Chew, E. A functional taxonomy of music generation systems. *ACM Comput. Surv. 50*, 5 (Sept. 2017), 69:1–69:30.

[9] Rabiner, L. R. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE 77*, 2 (Feb 1989), 257–286.

[10] Ross, S. *Introduction to Probability Models*. Elsevier Science, 2014.

[11] Tu, S. Derivation of baum-welch algorithm for hidden markov models. http://people.eecs.berkeley.edu/ stephentu/writeups/hmm-baum-welch-derivation.pdf.

[12] Van Der Merwe, A., and Schulze, W. Music generation with markov models. *IEEE MultiMedia 18*, 3 (March 2011), 78–85.

[13] Walker, J. Ubuntu manpage: midicsv. http://manpages.ubuntu.com/manpages/bionic/man5/midicsv.5.html.

[14] Warrack, J. The new oxford compantion to music, 1983.

[15] Yanchenko, A. K., and Mukherjee, S. Classical music composition using state space models, 2017.