GENERATING MUSICAL ACCOMPANIMENT USING FINITE STATE TRANSDUCERS

Jonathan P. Forsyth, Juan P. Bello

Music and Audio Research Lab (MARL)
New York University
New York, NY USA
{jpf211,jpbello}@nyu.edu

ABSTRACT

The finite state transducer (FST), a type of finite state machine that maps an input string to an output string, is a common tool in the fields of natural language processing and speech recognition. FSTs have also been applied to music-related tasks such as audio fingerprinting and the generation of musical accompaniment. In this paper, we describe a system that uses an FST to generate harmonic accompaniment to a melody. We provide details of the methods employed to quantize a music signal, the topology of the transducer, and discuss our approach to evaluating the system. We argue for an evaluation metric that takes into account the quality of the generated accompaniment, rather than one that returns a binary value indicating the correctness or incorrectness of the accompaniment.

1. INTRODUCTION

Harmonic accompaniment of melody is a common feature of many musical styles. The automatic generation of harmonizations of melodies has a number of applications, such as in interactive and algorithmic composition systems, real-time interactive music systems, as well as tools for education and entertainment aimed at novice musicians. There are several popular commercial products, such as Band-in-a-Box,[1] that offer this functionality. There has been a good deal of research into this task, and many different techniques have been employed, including the use of genetic algorithms, music-theoretic concepts, and neural networks. Amongst those, finite-state methods, in particular Hidden Markov Models (HMMs), have been applied to accompaniment generation.

Another type of finite state machine, the *finite state transducers* (FST) has been used primarily in the fields of natural language processing, machine translation, and speech recognition. FSTs encode a mapping from an input string to an output string, and can provide a convenient way to reduce a complex problem to a series of more manageable sub-problems. Because we can represent musical phenomena as sequences of symbols, FSTs have applications in various music-related tasks, such as an audio fingerprinting system [1], as well as a musical accompaniment generation system [2]. In addition, there are a number of available FST software libraries, such as [3], that provide highly efficient implementations of FST algorithms capable handling large data sets. Some of these libraries implement the training of probabilistic FSTs using unsupervised learning techniques.

While FSTs naturally lend themselves to the task of automated accompaniment, there are numerous steps in the process of quantizing an audio or MIDI signal to a finite set of symbols so that it can be used in an FST. Each of these steps, as well as the choice of FST topology, requires making a number of design decisions that can affect the overall quality of the system. Although it would be desirable to be able to quantify the performance of an accompaniment generation system, common evaluation metrics such as accuracy may not be appropriate. In this paper, we describe our work on a MIDI-based accompaniment generation system, and a provide a quantitative evaluation of the system's performance across a range of design parameters. Although the system described in this paper is MIDI-based, we plan to develop it into an audio-based system. However, we are currently focused on some of the fundamental design decisions, and issues related to the use of audio are outside the scope of this paper.

The remainder of this paper is structured as follows. In section 2, we review some related systems for automated accompaniment generation. In section 3, we describe finite state transducers in more detail, and provide the details of our system. Section 4 describes our evaluation methods and results, and in section 5, we discuss our conclusions and directions for future work.

2. RELATED WORK

Many music generation systems have employed various types of finite state machines, such as the factor oracle and audio oracle [4, 5], variable-length Markov chains [6], and prediction suffix graphs [7]. Finite state machines have also been used to generate automated harmonic accompaniment for a melody. A number of these systems use Hidden Markov Models (HMMs). For example, Allan and Williams [8] train a first-order HMM to generate chorale harmonizations by learning from a corpus of training data consisting of pieces by Johan Sebastian Bach. In their system, melody notes are treated as observations and chords as hidden states. Once trained, the HMM generates an accompaniment for a given melody by finding the most likely chord sequence. An additional HMM models ornamentation.

Buys and van der Merwe also implemented a system for harmonizing melodies in the style of Bach chorales [2]. In this system, probabilistic finite state transducers and automata are used model various aspects of harmonization. For example, a single-state FST is used in conjunction with a first-order Markov chain. The transducer maps chords to melody notes, and the Markov chain models chord progressions.

While the systems described above attempt to emulate Bach's specific style of harmonizing melodies, MySong [9] is a system, designed for novice and non-musicians, that generates harmonic accompaniment for vocal melodies. MySong also makes use of HMMs with melody notes as observations and chords as hidden states. The HMMs are trained on a large corpus of popular, rock,

[1] http://www.pgmusic.com/

jazz, R&B, and country songs.

Paiement, Eck, and Bengio [10] describe an approach based on a probabilistic graphical model designed to incorporate musical information at different levels in the temporal hierarchy. The authors describe models to predict root note progression given a melodic sequence, and to generate harmonic progressions, given the root and melody notes. The system developed by Raczyński et al. [11] uses a model that interpolates between a number of different sub-models that incorporate information about various aspects of the music, such as the current tonality and melody.

3. APPROACH

Our system makes use of probabilistic finite state transducers in a similar fashion to the one described in [2]. However, because we intend to expand our system for use with audio signals, we take a different approach to the quantization of the music signals. Quantizing a signal as complex as music into a finite set of symbols, a step necessary when employing finite state machines, poses some interesting challenges. We will discuss these issues in 3.2. However, we first provide some background on finite state automata and transducers.

3.1. Finite State Automata and Transducers

Finite state machines are used in a variety of fields including natural language processing, bioinformatics, and computer vision [12]. *Finite state automata* (FSAs) and *finite state transducers* (FSTs) are two closely related types of finite state machine. They are typically represented as directed graphs, which consist of nodes, referred to as states, and edges. Each edge is labelled with one or two symbols (in the case of FSAs and FSTs, respectively), and, optionally, a weight.

More specifically, an FSA $\mathcal{F}$ is defined by the 5-tuple $\langle \Sigma, Q, I, F, \delta \rangle$, where Σ is a finite alphabet, Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\delta(q, a)$ is a state-transition function. The state-transition function defines a mapping between the state $q \in Q$ and $q\prime \in Q$ given an input symbol $a \in \Sigma \cup \{\epsilon\}$, where ϵ represents a null input. An edge connecting states q and $q\prime$ labeled with ϵ allows a transition between q and $q\prime$ without consuming any input.

If we have an input string $A = a_o a_1 \dots a_n$, where $a_i \in \Sigma \cup \{\epsilon\}$, we can parse A by presenting one symbol at a time to $\mathcal{F}$, thus proceeding from one of the initial states through the automaton, governed by the state-transition function $\delta(q, a)$. If the state we are in when we reach the final symbol in our string is one of the final states, then the string is said to be *accepted* by the automaton; otherwise, the string is *rejected*. For example, if we for the moment ignore the output labels and weights in figure 1, we can view it as an FSA. The string "bca" is accepted by this FSA, while the string "cc" is rejected. For this reason, FSAs are sometimes referred to as *finite state acceptors*. FSAs can also be viewed as generative; in this case, the FSA can generate the strings $\{aa, aba, bba, bca, bcba\}$.

In a weighted FSA, weights are associated with each transition defined by δ. In addition, weights may be associated with the initial and final states. Therefore, in addition to indicating whether or not input string A has been accepted or rejected by a weighted FSA $\mathcal{F}$, parsing A will produce a set of weights accumulated along the path followed through $\mathcal{F}$. These weights are often used to represent probabilities associated with each state in the automata. We can define the weight on the transition from state q to state $q\prime$ as the probability of transitioning from q to $q\prime$; in this case, the probabilities on all the transitions from q must sum to one. For example, if the weights in the FSA in figure 1 represent probabilities, then it will generate the string "bca" with probability $0.3 \times 0.4 \times 0.8 = 0.096$.

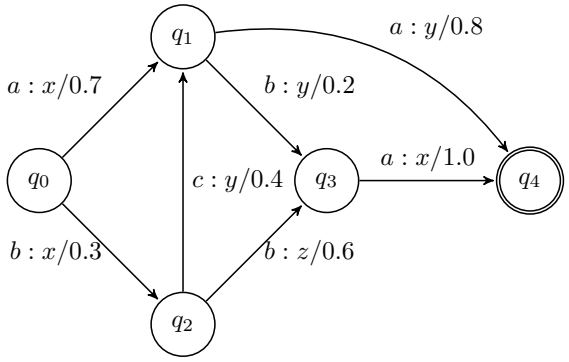

Figure 1: *Probabilistic FST with input alphabet $\Sigma = \{a, b, c\}$ and output alphabet $\Delta = \{x, y, z\}$. The initial state is q_0 and the final state is q_4. Edges are labeled as input label:output label/weight.*

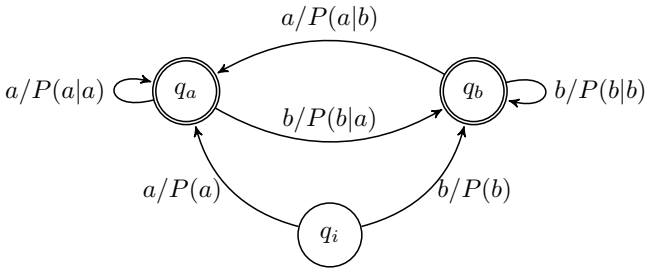

Figure 2: *Probabilistic FSA representing the bigram for the alphabet $\Sigma = \{a, b\}$. The initial state is q_i and the final states are q_a and q_b.*

Finite state transducers extend FSAs through the inclusion of an output alphabet Δ to the set of parameters, and are thus defined by the 6-tuple $\langle \Sigma, \Delta, Q, I, F, \delta \rangle$. The transition function δ still maps between two states given an input symbol, but additionally defines an output symbol $b \in \Delta \cup \{\epsilon\}$. Like an FSA, we can parse a string of symbols from the input alphabet Σ with an FST $\mathcal{T}$. As above, the input string A will be either accepted or rejected by $\mathcal{T}$. In addition, $\mathcal{T}$ will produce a series of output symbols $B = b_o b_1 \dots b_n, b_i \in \Delta \cup \{\epsilon\}$, with each input symbol a_i producing an output symbol b_i. In this way, an FST maps a string in the input alphabet to a string in the output alphabet. For example, we again consider the FST shown in figure 1. This FST accepts the strings $\{aa, aba, bba, bca, bcba\}$, as described above, and thereby produces the corresponding strings $\{xy, xzx, xyy, xyyx\}$. Note that an FSA can be represented as an FST in which the input and output labels are identical.

As in the case of weighted FSAs, weighted FSTs are transducers with weights, often probabilities, associated with each transition. These probabilities can represent the conditional probability of an output string given an input string, i.e., $P(B|A)$. For ex-

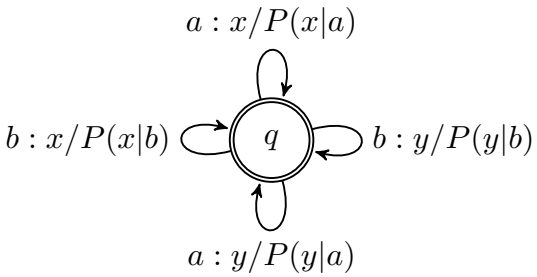

Figure 3: *Single-state probabilistic FST representing a mapping between input alphabet $\Sigma = \{a, b\}$ and output alphabet $\Delta = \{x, y\}$; q is the initial and final state.*

ample, if the FST in figure 1 encodes this conditional probability, then $P(B = xyy|A = bca) = 0.3 \times 0.4 \times 0.8 = 0.096$.

There are a number of unary and binary operations one can apply to FSTs [13]. One important operation, *composition*, allows us to combine two transducers, $\mathcal{T}_1$ and $\mathcal{T}_2$, into a larger transducer $\mathcal{T}$. This operation is notated as $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2$, and requires that the ouput alphabet of $\mathcal{T}_1$ is the same as the input alphabet of $\mathcal{T}_2$. In this way, we can break a complex problem into simpler parts. In the context of speech recognition, for example, one approach is to construct separate probabilistic FSTs to model grammar, pronunciation, and phoneme to phone mapping, which can then be composed into one larger FST [14]. There are also algorithms to reduce the size and improve the search efficiency of weighted transducers, making the use of these larger FSTs more practical [14].

With composition, we can use a probabilistic FST, $\mathcal{T}$, to estimate a string of output symbols B, given a string of input symbols A. In this case, we first create a simple transducer $\mathcal{A}$ whose edges are labelled with the symbols of A. We then use composition to form $\mathcal{T}_\mathcal{A} = \mathcal{A} \circ \mathcal{T}$. $\mathcal{T}_\mathcal{A}$ will contain all the possible paths the input string A can take through $\mathcal{T}$, and a shortest path algorithm such as the Viterbi algorithm or beam search can be used to find the most likely path through $\mathcal{T}_\mathcal{A}$, yielding the output sequence B.

In addition, composition of weighted transducers also allows us to represent a Hidden Markov model using an FSA and an FST [15]. A Markov chain, $\mathcal{M}$, can be represented by a probabilistic FSA; an example of such an automaton is shown in figure 2, which represents a bigram for the alphabet $\Sigma = \{a, b\}$. We can compose this with a single-state FST, $\mathcal{T}$, such as the one shown in figure 3. This FST maps the input alphabet $\Sigma = \{a, b\}$ with the output alphabet $\Delta = \{x, y\}$. If we have a sequence of input symbols, $\{\sigma_0\sigma_1 \ldots\}$, then $\mathcal{M}$ models $P(\sigma_t|\sigma_{t-1})$. Similarly, $\mathcal{T}$ models $P(\delta_t|\sigma_t)$, where δ_t is an output symbol. If we form $\mathcal{M} \circ \mathcal{T}$, the resulting FST, $\mathcal{H}$, will model $P(\sigma_t|\sigma_{t-1}) \cdot P(\delta_t|\sigma_t)$. If we consider $\mathcal{M}$ as modeling the transition probabilities, and $\mathcal{T}$ as modeling the emission probabilities, then we can view $\mathcal{H}$ as an HMM. In our system, we use such a composition of a Markov chain, represented as an FSA, and an FST (see section 3.3).

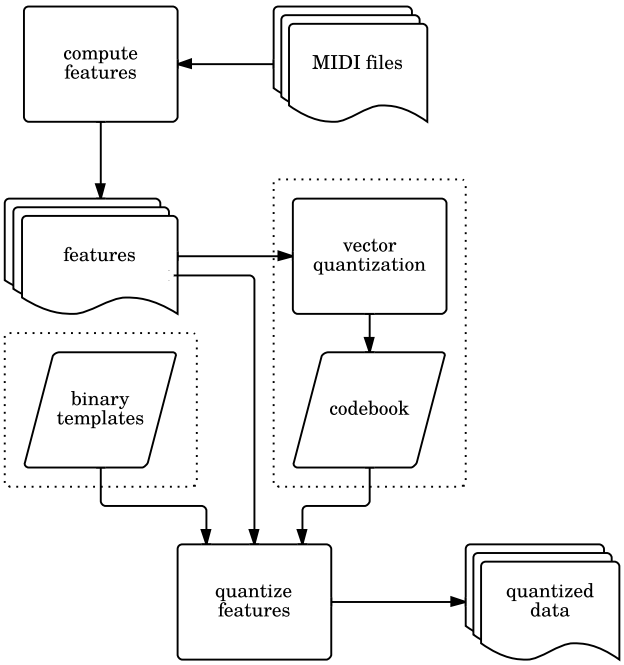

Figure 4: *Overview of the process to quantize MIDI data into sequences of symbols.*

3.2. Quantization

As in [8] and [2], we selected a set of four-voice chorales by J. S. Bach in MIDI format as a data set.[2] These chorales are are well-suited to serve as our data set for a number of reasons. In particular, the pieces are relatively similar, and are harmonically, melodically, and rhythmically simple, with a clear harmonic structure. In addition, the number of chorales in this collection is fairly large (almost 400 pieces). We use the music21 Python toolkit [16] to parse the MIDI files.

Because FSAs and FSTs require finite alphabets, our first task is to find a discrete representation for the MIDI data, both single notes (which will ultimately serve as the input to the accompaniment system) and simultaneities of notes (i.e., more than one note sounding during a particular time interval). An overview of the quantization process is shown in figure 4. In brief, a set of features is computed from the MIDI data. A vector quantization algorithm computes a codebook from the feature set; this codebook is used to quantize the features from continuous-valued vectors to a finite set of symbols. Alternatively, the quantization step can use a set of "hand-crafted" binary templates instead of a codebook.

For the monophonic melodies, we use pitch class representation, in which all notes are folded to a single octave and represented by a single value between 0 and 11, corresponding to the notes C through B. For each of the individual voices of a chorale, pitch class is a sufficient representation. However, if we wish to represent simultaneities of pitches , we can use a 12-dimensional

[2]Downloaded from `http://www.jsbchorales.net`

chroma vector x_i, where $i \in [0, 11]$ denotes a pitch class.

Although pitch class and chroma discard useful musical information, namely pitch height, there are a number of important practical advantages to these representations. Firstly, representing each note as a MIDI pitch value can create an intractably large space, while using an octave invariant representation greatly reduces the feature space and increases the number of similar samples to train the finite state machines. In addition, octave invariant representations allow us to easily transpose chroma vectors simply by performing a circular rotation. This property of chroma vectors allows us to easily key-normalize each piece in our data set (i.e., transpose all pieces to the same key).

We can also analyze the music signal at any number of levels of temporal resolution. Because the music in the training set consists largely of quarter note durations, we choose to use this as the maximum temporal quantization level. We also experimented with higher temporal resolutions, in particular eighth note and onset levels, but found that performance deteriorated at these resolutions.

At each quarter note, we represent the note in the melodic sequence by its pitch class, already a discrete value (if more than one melody note occurs with a quarter note duration, we select the one closest to the downbeat). We represent the harmonic progression as sequence of chroma vectors (referred to as a chromagram). Because the MIDI files in the data set did not contain any useful MIDI velocity information, we compute the magnitude of each dimension of a given chroma vector, denoted m_i, as:

$$m_i = \frac{dur_i}{dur_q}, i \in [0..11] \quad (1)$$

where dur_i is the duration of the note with pitch class i, and dur_q is the duration of the quantization level (in this case, a quarter note). Thus, the chromagram is normalized such that $0 \leq m_i \leq 1, \forall i \in [0..11]$.

In addition, we employ an optional key-normalization step to transpose all the pieces to the key of C major.[3] For each piece, we use the music21 implementation of the Krumhansl-Schmuckler key-finding algorithm to find the key, and perform the transposition by performing a circular rotation on the piece's chromagrams.

Because a finite state transducer maps an input string to an output string, we need to produce pairs of sequences from our training data. Each of the pieces in our data set has four voices. Because we are concerned with modeling the notion of accompaniment in general, we use each of the individual voices as the source of the input sequence, while the remaining three voices are combined to form the accompaniment. Each piece thus provides four training pairs, each of which consists of a monophonic melody and a polyphonic accompaniment.

We first compute the beat-synchronous chromagrams for each of the voices. Quantizing individual voices is trivial: the chromagram can be converted directly into a sequence of pitch class symbols. These pitch class symbols are denoted pc_i, where i indicates the pitch class. The chromagrams for the remaining three voices are summed into a single chromagram, which is then renormalized so that the maximum value in each chroma vector is 1. The next step is to quantize these chroma vectors. The two methods we employ are discussed below.

[3] minor key pieces are transposed to the key of A minor, the relative minor of C major

3.2.1. Binary templates

A binary template [17] is essentially a "hand-crafted" chroma vector that represents a particular chord type we wish to represent. A 1 is used to indicate the presence of a particular pitch class, and a 0 indicates its absence. For example, we represent a C major triad, consisting of the notes C, E, and G, with the chroma vector $v = [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]$. Note that index 0 corresponds to the pitch class C, index 1 corresponds to the pitch classes C♯/D♭, and so forth. One can use any number of possible combinations of binary templates to represent harmony; one common set, used in the chord recognition task, consists of the 24 major and minor triads [17].

3.2.2. Vector quantization

Vector quantization is a form of lossy data compression in which a set of points, referred to as a codebook, is used to represent a much larger set of data points [18]. The codebook vectors can be computed in any number of ways, including via the K-means algorithm.

In our system, we use the online vector quantization algorithm described in [19]. In this algorithm, K random points from the data set are selected as the initial codebook vectors. The algorithm then iterates through all the data points; for each point, the closest codebook vector, v, is found. The position of v is then adjusted so that it lies closer to the data point by an amount governed by a learning rate. After this process has been completed for each point in the data set, it repeats for a fixed number of iterations. The user must specify the number of iterations, as well as the learning rate and the initial size of the codebook, K.

As described in [19], we perform a circular rotation on each chroma vector in the data set so the maximum magnitude chroma bin is in the 0^{th} bin. This rotation attempts to transpose all the chroma vectors in the data set to the key of C, and thus the computed codebook will also represent chroma vectors in C. Clearly, this codebook alone would be insufficient to represent most harmonies, so, after the initial codebook has been computed, we circularly rotate each of the codebook vectors to each of the 11 other possible bins, in effect transposing the codebook to the 11 other key centers. This method has the added benefits of greatly reducing computation time and ensuring that our codebook can represent harmonies in all keys equally well.

3.2.3. Quantizing the chromagrams

At this stage, we quantize the chromagrams computed in the previous steps. We quantize the monophonic chromagrams by directly converting them to a sequence of pitch class symbols. In order to quantize polyphonic chromagrams, we use either a codebook computed as described above, or a set of binary templates.[4] In order to quantize a chroma vector v, we use Euclidean distance to measure the distance between v and each of the K codebook vectors; the closest such vector, v_{min}, is then used to represent v. We can then represent a chroma vector as a symbol k_i, where $i \in [0..K-1]$ indicates the index of v_{min}.

At this point, our training data, originally consisting of chromagrams, is now represented by pairs of input and output symbol sequences, S_{in} and S_{out}, respectively. For a given pair, S_{in} is a

[4] For convenience, we will refer to a set of binary templates as a codebook

sequence of pitch class symbols, i.e., $S_{in} = \{s_{in}^0, s_{in}^1, \ldots\}, s_{in}^i \in \{pc_0, pc_1, \ldots pc_{11}\}$, while S_{out} is a sequence of symbols corresponding to the codebook vectors in a size-K codebook, i.e. $S_{out} = \{s_{out}^0, s_{out}^1, \ldots\}, s_{out}^i \in \{k_0, k_1, \ldots k_{K-1}\}$.

3.3. Transducer model

There are a wide variety of different transducer topologies [20, 14], such as single-state transducers, transducers with states corresponding to each symbol in the input and output languages, and fully-connected networks of transducers. The model used in our system follows the design described in [2]. Specifically, our model consists of the composition of an FSA and an FST, as described in section 3.1. The FSA is constructed as an *n*-gram that models chord sequences, i.e. $P(c_t|c_{t-1}, c_{t-2}, \ldots c_{t-n})$, where c_t is a chord at time t and n is the *n*-gram order. The FST consists of a single state, and models the mapping between chords and melody notes, i.e. $P(m_t|c_t)$, where m_t is a melody note at time t.

To generate the *n*-grams, we used the OpenGRM NGram library [21]. We then converted the resulting *n*-grams to the transducer topology definition format used by the Carmel-finite state toolkit.[5] We computed $P(m_t|c_t)$ directly from the training data. Once the models had been trained, the Carmel toolkit was used to generate the output sequences.

4. RESULTS

We ran a number of tests, varying the order of the *n*-grams, the type of codebook used (i.e., either binary template or computed via online vector quantization), and the codebook size. For each test, we divided the data set into three roughly equal folds; one fold served as a test set while the remaining two were used to learn the *n*-gram and FST parameters. We computed our evaluation metrics (see section 4.1) for each test fold. Each fold was used as a test set in turn, and the metrics were averaged over all three tests.

4.1. Evaluation metrics

Following [22], we computed the percentage of correctly generated accompaniment symbols using the input/output sequence pairs in the each fold of the test set. Each of these pairs consists of an input sequence S_{in} and an output sequence S_{out}. S_{in} was presented as an input to the FST, which generated a prediction sequence $\hat{S}_{out} = \{\hat{s}_{out}^0, \hat{s}_{out}^1, \ldots\}$. Each symbol in the prediction sequence $\hat{s}_{out}^i$ was compared to the corresponding symbol in the output sequence, s_{out}^i. If $\hat{s}_{out}^i = s_{out}^i$, a count of the global number of correct predictions, $N_{correct}$, was incremented. The total number of predicted symbols generated by the FST, N_{total}, was also counted. This procedure was repeated for all sequence pairs in the fold, and the average accuracy for that fold was then computed as:

$$Accuracy = \frac{N_{correct}}{N_{total}} \times 100\% \quad (2)$$

The accuracy simply represents the average percentage of the time the system predicted the correct symbol.

In addition, we computed the per-symbol distortion to measure the overall quality of the predicted harmonic sequence, similar to the distortion measurement described in [19]. The intuition here is that an incorrectly predicted symbol may or may not be a musically reasonable choice. In other words, suppose that for a given input symbol, the predicted symbol corresponds to a C major triad, while the ground truth symbol corresponds to a Cmaj7 chord. Since the two symbols are not equal, this prediction would be counted as incorrect. However, a C major triad is a much better prediction for Cmaj7 than a C♯ major triad would be, yet this is not reflected by the accuracy.

In order to account for this problem, Chuan and Chew [22] use a map representing related chords in order to account for this situation. Our approach, however, is to compute the Euclidean distance between the codebook vector corresponding to the predicted symbol, and the actual chroma vector computed directly from the MIDI file. This distance is computed for each predicted symbol, and the average is computed in a manner similar to the accuracy computation. Finally, the distortion value is normalized to the range $[0, 1]$.

4.2. Results

We ran the tests described above using codebooks consisting of the 24 major and minor triads (denoted BT_{24}); the 36 major, minor, and diminished triads (BT_{36}); a set of 276 binary templates built from 23 common chords transposed to all 12 keys (BT_{276}); and codebooks computed using vector quantization with $K = 24$ (K_{24}), $K = 36$ (K_{36}), and $K = 276$ (K_{276}). For each codebook, the *n*-gram order was tested with $n = \{1, 2, 3, 4\}$, where possible (in the case of the codebooks BT_{276} and K_{276}, memory issues prevented us from completing tests for $n = 4$). The average accuracy ("acc") and average per-symbol distortion ("dist") computed for each of these tests are shown in the tables 1 and 2.

Although none of the accuracy scores shown in the tables are particularly high, they are similar to the accuracy values reported in [22] and [11]. In general, the accuracy scores when using key-normalization (table 2) are substantially higher than the accuracy scores when the data is not key-normalized (table 1). For example, if we consider theBT_{24} codebook and an *n*-gram order of 3, the application of key normalization improves the accuracy from 34.7 to 41.5. Similarly, the distortion values are generally lower in the key-normalized case.

Key normalization also seems to have had a particularly strong effect on the performance of the K_{24}, K_{36}, and K_{276} codebooks. These codebooks were created by running the vector quantization algorithm described in section 3.2.2. Before the vector quantization step, we rotate each chroma vector in the data set so the maximum magnitude chroma bin is in the 0^{th} bin under the assumption that the chroma vectors would correspond to root position chords (i.e., chords in which the lowest note is the root). However, because we set the magnitude of the chroma vectors based on note duration, this assumption is not guaranteed to be correct. As a result, this technique may introduce some redundancy when the initial codebook vectors are transposed, which may in turn contribute to the lower accuracy scores. The key normalization process seems to have ameliorated the problem to a certain extent, perhaps because it performs a rotation consistent across all pieces, thus reducing some of this redundancy.

While the two smaller binary template codebooks achieved the best accuracy, their per-symbol distortion values were higher than that of the other codebooks. In fact, a smaller codebook size will in general yield better accuracy results, as any random guess is more likely to be correct than with a larger codebook. The higher distortion values also make sense if we consider that 24 major and

[5] `http://www.isi.edu/licensed-sw/carmel`

Table 1: *Results for different codebook sizes and n-gram orders, without key normalization.*

	BT_{24}		BT_{36}		BT_{276}		K_{24}		K_{36}		K_{276}	
order	acc	dist	acc	dist	acc	dist	acc	dist	acc	dist	acc	dist
1	31.8	0.420	31.0	0.419	26.4	0.402	28.9	0.267	29.7	0.300	17.6	0.388
2	34.5	0.404	33.3	0.406	27.5	0.386	34.7	0.241	34.2	0.271	18.7	0.356
3	34.7	0.402	33.4	0.404	24.0	0.404	35.0	0.240	34.4	0.268	15.4	0.404
4	31.3	0.415	27.3	0.426			33.2	0.240	34.2	0.273		

Table 2: *Results for different codebook sizes and n-gram orders, with key normalization.*

	BT_{24}		BT_{36}		BT_{276}		K_{24}		K_{36}		K_{276}	
order	acc	dist	acc	dist	acc	dist	acc	dist	acc	dist	acc	dist
1	37.6	0.351	36.9	0.354	23.7	0.415	36.2	0.234	36.2	0.264	26.2	0.329
2	39.7	0.342	39.3	0.342	24.9	0.405	38.1	0.226	38.0	0.251	28.0	0.321
3	41.5	0.333	40.3	0.338	24.9	0.405	38.0	0.227	38.3	0.249	28.2	0.319
4	42.0	0.330	40.9	0.334			38.2	0.226	38.2	0.249		

minor triads, or even 36 major, minor, and diminished triads, may not adequately represent the harmonic material in the data set. This reasoning implies that larger codebooks are better able to represent the harmonic material, yielding better distortion scores, but lower accuracy scores. This is, for the most part, born out by the data.

Similarly, if we compare the results for BT_{276} and K_{276} in the key normalized case, we see that the latter codebook has both better accuracy and distortion scores. This implies that the chords selected for the BT_{276} codebook were not particularly representative of the harmonies present in the training data. On the other hand, because the K_{276} codebook is learned directly from the data, it seems to better represent the harmonies in the data. This is true when comparing binary template and vector quantization codebooks of the same size.

A direct comparison between our system and similar systems, in particular the ones described by Buys and van der Merwe [2] and Allan and Williams [8], was not feasible for a number of reasons. First, our system is designed to model a general notion of harmonic accompaniment for a melody. On the other hand, the aforementioned systems are designed to emulate Bach's compositional style, and are therefore more complex than ours. Thus, a direct comparison does not seem appropriate. In addition, the authors in [2] and [8] evaluate their systems using entropy as a metric, and it is not suitable to directly compare entropy values between such different models [15]. Because these systems were not publicly available, we were unable to evaluate them using the average accuracy and distortion metrics.

5. CONCLUSIONS AND FUTURE WORK

This paper describes the design of a basic system to automatically generate harmonic accompaniment given a melody. The system, trained on a corpus of MIDI data, has a number of design options, specifically, the choice of *n*-gram order, the type and size of the codebook used to quantize the MIDI data, and the application of key normalization. The system was then evaluated with various configurations of these components and parameters. Two evaluation metrics, accuracy and per-symbol distortion, were computed for each of the tests. It was argued that the distortion measure was a more appropriate metric for quantifying the performance than accuracy, as the former attempts to take into account the perceptual result of a predicted harmonic sequence.

Future work includes expanding the data sets used to train the system. These data sets would consist of music from a variety of genres, and would be more harmonically, melodically, and rhythmically complex music than the data set used for this work. In addition, we plan to expand our system to use audio signals as input and output.

As we pursue these plans, we will also improve our evaluation metrics. Accuracy and distortion have their limitations, and we are thus exploring different metrics, such as those based on information theory. More study is needed to determine how well these metrics correlate with human perception of the quality of a generated accompaniment. User studies would be valuable in assessing the quality of our system and our quantitative metrics. Further study is also required to determine optimal quantization strategies and transducer topology design.

6. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation (grant IIS-0844654).

7. REFERENCES

[1] Eugene Weinstein and Pedro Moreno, "Music identification with weighted finite-state transducers," in *Proceedings of the IEEE International Conference on Acoustics, Speech and*

Signal Processing (ICASSP 2007). IEEE, 2007, vol. 2, pp. II–689.

[2] Jan Buys and Brink van der Merwe, "Chorale harmonisation with weighted finite-state transducers," in *PRASA 2012: Proceedings of the 23rd Annual Symposium of the Pattern Recognition Association of South Africa*, Pretoria, South Africa, November 29-30 2012.

[3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri, "OpenFst: A general and efficient weighted finite-state transducer library," in *Implementation and Application of Automata*, pp. 11–23. Springer, 2007.

[4] Gérard Assayag and Shlomo Dubnov, "Using factor oracles for machine improvisation," *Soft Computing*, vol. 8, no. 9, pp. 604–610, 2004.

[5] Shlomo Dubnov, Gerard Assayag, and Arshia Cont, "Audio oracle: A new algorithm for fast learning of audio structures," in *Proceedings of the International Computer Music Conference (ICMC)*, 2007, pp. 224–228.

[6] François Pachet, "The continuator: Musical interaction with style," *Journal of New Music Research*, vol. 32, no. 3, pp. 333–341, 2003.

[7] José Luis Triviño-Rodriguez and Rafael Morales-Bueno, "Using multiattribute prediction suffix graphs to predict and generate music," *Computer Music Journal*, vol. 25, no. 3, pp. 62–79, 2001.

[8] Moray Allan and Christopher K. I. Williams, "Harmonising chorales by probabilistic inference," *Advances in Neural Information Processing Systems*, vol. 17, pp. 25–32, 2004.

[9] Ian Simon, Dan Morris, and Sumit Basu, "MySong: Automatic accompaniment generation for vocal melodies," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 725–734.

[10] Jean-François Paiement, Douglas Eck, and Samy Bengio, "Probabilistic melodic harmonization," in *Advances in Artificial Intelligence*, pp. 218–229. Springer, 2006.

[11] Stanislaw Raczyński, Satoru Fukayama, Emmanuel Vincent, et al., "Melody harmonisation with interpolated probabilistic models," *Research Report RR-8810, INRIA*, 2012.

[12] Colin De la Higuera, *Grammatical inference: Learning automata and grammars*, Cambridge University Press, 2010.

[13] Mehryar Mohri, "Weighted automata algorithms," in *Handbook of Weighted Automata*, pp. 213–254. Springer, 2009.

[14] Mehryar Mohri, Fernando Pereira, and Michael Riley, "Speech recognition with weighted finite-state transducers," *Handbook on Speech Processing and Speech Communication*, 2008.

[15] Daniel Jurafsky and James H. Martin, *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, Pearson Prentice Hall Englewood Cliffs, NJ, 2000.

[16] Michael Scott Cuthbert and Christopher Ariza, "music21: A toolkit for computer-aided musicology and symbolic music data," in *Proceedings of the International Symposium on Music Information Retrieval*, 2010, vol. 11, pp. 637–42.

[17] Taemin Cho, Ron J. Weiss, and Juan Pablo Bello, "Exploring common variations in state of the art chord recognition systems," in *Proceedings of the Sound and Music Computing Conference (SMC)*, 2010, pp. 1–8.

[18] Christopher M. Bishop, *Pattern recognition and machine learning*, Springer, 2006.

[19] Thierry Bertin-Mahieux, Ron J. Weiss, and Daniel P. W. Ellis, "Clustering beat-chroma patterns in a large music database," in *ISMIR 2010: Proceedings of the 11th International Society for Music Information Retrieval Conference*, Utrecht, Netherlands, August 9-13 2010, pp. 111–116.

[20] Charles Schafer, "Novel probabilistic finite-state transducers for cognate and transliteration modeling," in *7th Biennial Conference of the Association for Machine Translation in the Americas (AMTA)*, 2006.

[21] Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai, "The OpenGRM open-source finite-state grammar software libraries," in *Proceedings of the ACL 2012 System Demonstrations*. Association for Computational Linguistics, 2012, pp. 61–66.

[22] Ching-Hua Chuan and Elaine Chew, "Generating and evaluating musical harmonizations that emulate style," *Computer Music Journal*, vol. 35, no. 4, pp. 64–82, 2011.