Implementing A Music Improviser Using N-Gram Models

6.863J/9.611J Natural Language Processing - Final Project

Kristen Felch, Yale Song

{kfelch|yalesong}@mit.edu

Massachusetts Institute of Technology

I. Vision

Music improvisation is a vital element in learning music composition skills. It helps a practitioner to learn scales, chord progressions, and playing in a certain musical styles--e.g. blues, jazz, rock, folk. Once a player familiarizes oneself with the basic knowledge of music improvisation, the player often try to improvise with other musicians in order to share ones ideas and get some feedback. However, it is not always easy to find other musicians to play with and get together at a certain place. Therefore, it would be nice if we have a computer that acts as a musical companion who knows how to improvise.

In this paper, we show a statistical model that automatically generates musical phrases to imitate music improvisation. The model extracts three types of information from parsed MIDI data--scales, chords, and rhythmic patterns-- and builds several bi-gram models based on the three features. Once the model is trained, it gets a MIDI file as an input, extract the three features from the song, improvise based on the extracted features with the bi-gram models, replace a melody track with an improvised track, and outputs a modified MIDI file. We show how we implemented each step in greater detail. Also, a link to some of the example songs is complemented along with this paper so that a reader can listen to the results.

II. Steps

II.1. Classification

II.1.A. Data

We chose to focus on the jazz genre for this project. While our methods could be used on other genres, this limited the variation that comes with dealing with different styles of music. Our data consists of 183 midi files that contain well-know jazz recordings.

In order to deal with differences in key signature between the pieces, we normalized all of the notes in the piece such that the root of the I chord is note number 0. The remaining notes are numbered 1-11 starting at this root. Our parsed MIDI files are stored using note names and key signatures, but starting with chord and scale extraction we use note numbers so that information learned from n-gram models is applicable across key signatures.

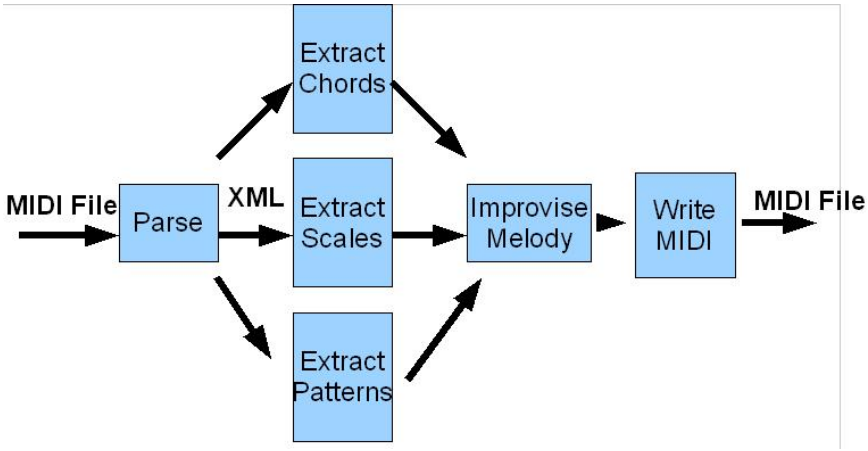

Figure 1. High-level view of music improviser steps

II.1.B. Parsing Midi Files

The International MIDI Association defines the standard MIDI file format [3]. Three types of MIDI format are defined: type 0 file contains a single multi-channel track, type 1 file contains one or more simultaneous tracks (or MIDI outputs) of a sequence, and type 2 file contains one or more sequentially independent single-track patterns. Since comprehensive MIDI parsing was not our primary concern in this project, we picked a type 1 format which is relatively straightforward to parse each individual track.

We assume that each song has one melody track and one background-chord track. The melody track is used for extracting scales and rhythmic patterns of a song, and the background-chord track is used for extracting chords of a song. Since a MIDI file does not indicate such track information, we hand-labeled the two track numbers of each MIDI file. Each of our team member used one of two MIDI software (Roland Cakewalk9 or Steinberg Cubase SX3) in order to determine the two tracks by visualizing each track information and selectively listening to an individual track. One that contains main melody of a song was selected as a melody track, and one that contains either base line or chord voicing was selected as a background-chord track.

A type 1 format MIDI file contains one block of header data and several blocks of track data. A header block contains general information of a MIDI file (number of tracks, time signature, etc), and each track block contains a list of MIDI events. Each MIDI event consists of a delta-time event (when an event should be played relative to the track's last event), event type, and some event type specific data. Note events (NOTE_ON or NOTE_OFF) have time, pitch, and velocity information. In our parsed output files, the note events for both melody and background-chord tracks are recorded.

There are many types of time signatures, but the most widely used time signature is 4/4 (common time). For simplicity, we assume all the songs have the same time signature. This allows us to divide each measure into 16 bins and record note events in a discretized manner.

With the two assumptions (each song has a melody and a background-chord track, and all the songs have the same time signature), we parsed our dataset using an open source MIDI parser [2]. Here is an example of a note event:

('note', (83, 5, 3), (52, 'E', 2, 88))

This event indicates that this is a note event that is occurred at third sixteenth of fifth measure (hence 83rd sixteenth), 'E' note with a second octave (hence pitch=52), and the velocity is 88.

II.1.C. Extracting Chords

The MIDI parser outputs two files- one with information from the melody track and the other with information from the background track. Each file contains a list of notes in the track, including information about the note's timing, volume, and pitch. Also included is information about key signatures as they change throughout the track. The background file will be used to determine the chord structure associated with the piece.
Our approach to finding the underlying chord structure for the piece is to identify the chord in the background track for each measure. This is done by looking at the notes that occur in the measure as well as an n-gram model that represents likely transitions between chords. We first classified the measure chords of each piece using only the notes in the measure, scoring each piece based on how closely the identified chords matched expected notes in a chord. Then, we used only the high-scoring pieces to create a bigram model of transition probabilities between chords. Finally, we used a combination of these bigram probabilities and notes in each measure to improve the identification of chord structure.

The first round of chord identification uses only the notes in the background track to find the closest matching chord. In order to find the chord for a given measure, we count the number of times each note occurs in the measure. These notes are numbered 0-11 as described above (see Data), and the count for each note is kept in a 1D array. We then compare this array to pattern arrays that contain the notes present in known chords. This method was inspired by the work of Bryan Pardo, more information in the works cited page. A pattern array contains only 1s and 0s, while the measure array may contain higher numbers if more than one instance of a note is present. The measure in question is compared with major, minor, dominant 7, major 7, and minor 7 chords for each of the 12 root notes. A matching score is calculated as follows.

1. For each note present in the measure that is 1 in the pattern, 1 is added to the score.
2. For each note present in the measure and 0 in the pattern, 1 is subtracted from score.
3. For each note not in the measure and 1 in the pattern, 1 is subtracted from the score.

Example: [3 0 0 0 3 0 0 2 0 0 0 0]

Looking at the above example, we can see that there are 3 instances of the 0th note for whichever key signature the piece is in. There are three instances of the 4th note, and 2 instances of the 7th. The pattern that this measure most closely matches is [1 0 0 0 1 0 0 1 0 0 0 0], which represents a major I chord. The score would be 8, since every note in the measure is 1 in the pattern.

Since some measures will contain more than one underlying chord, we allowed for the possibility that two chords could be present. To do this, we classified the whole song using one chord per measure and again using two chords per measure. To decide if each measure should contain one chord or two, we looked at whether the combined score of half-measures was better than the score of the entire measure using only one score. We therefore ended up with the entire piece composed of some measures that contained one chord and others that contained two.

After a score is calculated for each pattern, the chord with the highest score is assigned to that measure. The score of a piece is the average of the scores of each of the measures. We then filtered out only the pieces with average scores of 6 or higher to use to create our bigram model. In order to create the model, we simply counted the number of times that each chord follows each other chord. Since our dataset is not all that large, we normalized with respect to key signature and used note number rather than name in the transition probabilities. For example, a D->D transition in the key of D would be the same as a C->C transition in the key of C.

Once this bigram model was complete, we reclassified all of the pieces using our new knowledge. Instead of relying only on pattern-matching scores, we also looked at transition counts to determine which chord to assign to each measure. We did this by looking at the top 3 scoring chords by pattern matching. We then tried several different weightings for pattern match score and transition counts between the previous and current measure to choose between the three chords. In most cases, the closest matching score was also the most likely chord to follow the previous chord. However, there are some cases in which a strange chord (out of key signature for example) was the closest match, but the actual chord had a much higher transition probability and could be correctly classified by using the bigram model.

Deciding on a weighting between chord pattern matching and bigram probabilities was difficult. We decided that in cases where the chord matching scored very high, this should take precedence over any transition probabilities. Therefore, our model can accept as a parameter a number representing the number of standard deviations above the mean to be used as a threshold. The threshold for the highest score in order for transition probabilities to NOT be taken into account is calculated as follows.

1. Create a normalized set of all of the highest scores for chords in each measure
2. The mean and standard deviation are calculated for this set
3. The input parameter is multiplied by the standard deviation and added to the mean

After listening to the resulting chord structures for several pieces, we decided that 0 was the best threshold to use. This means that any measure whose highest scoring chord is above the average score for the piece will not use our bigram model. Any measure whose score falls below this average will use a combination of bigrams and pattern matching to choose the best chord for the measure. The formula that we used is pattern score + .1 * transition count, since the transition counts were about ten times higher than the pattern scores.

Once each measure has a chord associated with it, these chords are output to be used in our improvisation tool.

II.1.D. Extracting Scales

The method used for extracting scales was very similar to the method used for extracting chords. Instead of using the background track, we used the melody track to extract scales used in each measure. Just as in chord extraction, we collected an array of all of the notes contained in each measure of the track. Then, we compared these arrays to pattern arrays of major, minor, blues, and pentatonic scales. The scoring of different scales was done in the same way as chord extraction, except that no bigrams were used in the classification stage. The scale with the highest score was chosen, if that score was above a certain threshold. If the score was below the threshold, we used the chord found in the background track as the measure's scale. Our reasoning behind this is that if the melody line is not playing enough to determine the scale, we should use information from the background track to determine which scale would fit well over the measure.

Scale extraction results in a file that lists one or two scales for each measure of the piece. These scales can be one of the four types of scales listed above, or one of the various chord types addressed above if the scale scores were too low.

II.1.E. Extracting Rhythmic Patterns

When improvise, with the underlying chord progression, a player usually decide what scale to play with and what rhythmic pattern to use. There exists a limited number of scales, but the number of rhythmic patterns is almost infinite. This means that if we use a statistical model to automatically generate a rhythmic pattern, we will have the sparse data problem. In order to mimic the playing style of a song at best, we decided to extract rhythmic patterns of a song and then later pick a random pattern of a song to improvise. We used the melody track to extract rhythmic patterns because we wanted to mimic the rhythmic pattern of the melody line.

Extracting rhythmic patterns is rather different from the methods for extracting chords and scales. While scales and chords are decided based on the pitch information, rhythmic patterns are decided based on the duration and the volume information. Also, while scales and chords are extracted from each measure, patterns are extracted from more than one measures because a rhythmic pattern usually spans over more than one measure. The current implementation uses two measures to make a pattern, but our method can make a pattern with any arbitrary number of measures.

Here is an example pattern of two measures:

[[0, 2, 0], [1, 1, 87], [1, 2, 85], [0, 1, 0], [1, 5, 89], [0, 1, 0], [1, 1, 74], [1, 1, 49], [1, 1, 47], [1, 1, 87], [0, 1, 0], [1, 1, 83], [1, 1, 76], [1, 1, 71], [1, 1, 49], [1, 1, 62], [1, 1, 86], [1, 2, 109], [1, 1, 81], [1, 1, 70], [1, 1, 82], [0, 1, 0], [1, 1, 74], [1, 2, 66]]

Each element has three information: note on/off(1:on, 0:off), duration(in sixteenth scale), and volume(0~127). Note that there are 16 sixteenths in a measure. Since the pattern spans over two measures, if we sum up each duration value of each element in the pattern, it is equal to 32.

II.1.F. Building Bigram Model

The process of building our bigram model for chord transition probabilities was described briefly under Chord Extraction. We used two bigram models in this project- the first for determining the underlying chord structure and the second for use in our improvising tool. The model used for chord extraction is much simpler, since it is built by counting the transitions between chords in all files that achieved a particular score.

The bigram model that was used for our improvisation tool has the job of providing transition probabilities between notes given a particular scale for a measure. For example, if a measure is tagged with the C Blues scale, what is the probability that we will play a D if the previous note played was an F? In order to create this model, we used the original parsed XML files to obtain the notes in each measure as well as the scale listings that were generated by the scale extraction stage.

To create the model, we looked at the set of notes in each measure of all of our songs. For each transition between two notes, we record a count of how many times this transition occurs given a particular underlying scale. Therefore, our bigram model is more like a conditional bigram model conditioned on the underlying scale of the measure in which the transition occurs. We also counted the number of times each note is the starting note of a measure given a particular underlying scale.

The improvisation tool will be able to use this model to choose the notes to play in each measure. Given the scale that the measure needs to use, a random note can be chosen with higher weighting given to notes that are more likely to either start the measure or follow the previous note.

II.2. Generation

II.2.A. Improvising Melody

The melody improviser uses pattern information and scale information to generate a sequence of notes. There are two main steps in the improvisation. First, the improviser groups several chords together so that the duration of each group is equal to the duration of a pattern. Next, looping through each group from the previous step, the improviser randomly selects a pattern of a song, determines how many notes were played in the pattern, and generates a sequence of notes according to the scales in the group.

In order to choose the notes that should be used for the melody, we used the bigram model created in the previous stage. If we need the first note of a measure, then we look at the starting probabilities in our model for the scale that we are in. In order to choose a starting note, an array was created with the number of instances of each note equal to the number of times this note occurred as the starting note in a measure of the given scale. When a random index of this array was chosen, it was more likely to choose a starting note that is commonly a starting note in our training data. The same procedure was used to find the next note to play given a previous note, and therefore our improvisation tool favored note combinations that were found in the training data.

II.2.B. Writing MIDI Files

As a final step, we generate a MIDI file. First, we encode a sequence of notes from the previous stage to a sequence of MIDI events. Next, we overwrite the sequence of MIDI events to the original melody track. This is done by going backward as we have done in the parsing step.

III. Results

III.1. Improvising Tool

Our tool is composed of several modular parts, which can be run as python scripts separately. The first of these parses all MIDI files in a given folder and stores output in a 'parsed' folder as text files. Then we have several scripts that take these parsed files and create text files listing the patterns, chords, and scales for each measure. The scripts that create our bigram models use information from the chord and scale files. We have a script that takes in the patterns, chords, and scales and creates a new track. Finally, another script combines this improvised track with the original background to create a new MIDI file.

III.2. Music Samples

We have provided several music samples generated using our tool. The background tracks are maintained as from the original piece, but the melody track has been replaced with our new improvised track. This track is played by the piano. Some of the result songs can be found at

http://people.csail.mit.edu/yalesong/6.863-Music.Improviser

IV. Future Work

IV.1. Extending Rhythmic Pattern Selection

At this stage of development, we selected patterns from each piece and used these same patterns in the generated improvised track. These patterns were used in random order, but were extremely limited in variety especially for the shorter pieces. One extension to our work would be to use rhythmic patterns from different pieces in the piece we are working on. We would have to develop a method of sorting the pieces based on factors such as style and tempo, so that rhythmic patterns would 'fit' into the mood of the piece.

IV.2. Adding Dependency/Causality

When choosing the notes and the patterns for each measure, we did not consider previous notes or patterns. This is a large simplification, since music often builds on itself from the beginning to the end.

Certain phrases are repeated or altered slightly to form a refrain, rather than each measure being independent of all other measures. Adding this dependency to our framework would be difficult, but our program is modular enough that the change could be made. Instead of randomly choosing a pattern, we could choose a pattern that was already played a few measures early. When adding pitches to the patterns, we could have a higher probability of choosing the same notes that were already used in the pattern. This way, there would be some exact repetition. This repetition is often what listeners remember of the song.

IV.3. Improving Scale/Chord Classification

Our chord and scale classification methods are far from perfect. For example, in order to choose the threshold for use of the bigram model we listened to the resulting chords chosen by models with different thresholds. We chose to use a threshold of 0, because the chords chosen sounded the most like the original piece. Zero in this case means that about half of the chords were chosen because the measure notes matched a chord template very closely and the other half were chosen based on a combination of this template-matching and the bigram model. There is of course a much more scientific way to choose the threshold if time allows. All of the songs would have to be hand-parsed and then a program would try lots of thresholds and choose the one that has the highest percent accuracy.

For choosing scales, our model would improve if we included more of the many types of scales that exist. We choose to limit our scales to five types, because the others are more obscure and often caused dissonance if not used correctly. For example, improvising using a half-diminished chord is dangerous unless the correct notes of the chord are emphasized. Major, minor, blues, and pentatonic are much more likely to sound good with the background chords given our simple chord parser and semi-random choice of notes.

IV.4. Increasing Database

Like any classification problem, increasing the size of our dataset would increase the accuracy of our results. Our bigram model in particular would benefit from more data, because we would have a more accurate model of transition probabilities between notes given particular key signatures. At this point, some transitions are not even represented in the model.

IV.5. Extending to Other Genres

We chose to work in the jazz genre for this project, because improvisation is most commonly used in jazz pieces. However, the work could certainly be extended to other genres. New bigram models would have to be built for each genre, since what differentiates types of music is the patterns used and note transitions.

V. Contributions

1. Developed and implemented method for parsing and writing MIDI files
2. Implemented scale and chord Classification
3. Created Improvisation Tool

References

[1] Pardo, Bryan. The Chordal Analysis of Tonal Music. University of Michigan: 28 March, 2001.
[2] Ware, Will. MIDI Parser (midi.py), http://wiki.python.org/moin/PythonInMusic
[3] The International MIDI Association, Standard MIDI-File Format Spec. 1.1,
http://www.ta7.de/txt/musik/musi0006.htm