Song Mashups and Mixes with DJ MARKOV

Alex Gao, Andrew Kim, Brian Yang

November 21, 2018

1 Introduction

We had two goals:

1. Generate original songs based off source material
2. Combine those original songs into a smooth and original mash-up/remix, simulating a DJ at a party

To accomplish the first goal, we parsed musical notes from Midi files to generate DTMCs for single songs. This gave DJ MARKOV a repository of original compositions to mix together for the second goal. This was a fairly basic step; however, while working on the second goal, we discovered some interesting innovations to the basic DTMC approach which we will discuss later. To accomplish the second goal, which was far more intensive, we needed to do two things.

First, we needed to make sure that DJ MARKOV's mixes were smooth and connected rather than arbitrary concatenations of songs. To accomplish this, we modified the state space of the basic DTMC to involve n-grams as states (simulating n^{th} order DTMCs). This rich featurization allowed us to capture local dependencies beyond just the previous note. This captured the concept of a 'musical phrase' rather than a single note, which let us mix songs such that a musical phrase from one song led smoothly to a similar phrase from another. The precise value of the parameter 'n' also allowed DJ MARKOV to improvise more or less in its compositions; we discuss this in a later section.

Second, we wanted DJ MARKOV to mix together similar-sounding compositions more frequently. To accomplish this, we created a CTMC based on song similarity to mediate the transitions between different songs in the mix. This was made possible through the n-gram featurization; without it, we would not have had a musically valid way of representing song similarity (other ideas, such as stationary distribution analysis, were explored but didn't work).

We implemented the n^{th} order DTMC and CTMC models from scratch, using only numpy as an external dependency, to solidify our understanding of the material. With this framework and some other clever music representations, DJ MARKOV was able to generate songs that were recognizably derived from the source material and mix them together smoothly. Examples/results are included for you to listen along to!

2 Methods (Theory and Pseudocode)

2.1 Music Representation and Preprocessing

Musical notes are fairly easy to represent numerically; in the MIDI files we used (standard electronic representation of music) each musical note is a number from 0-127. However, most MIDI files would not be suited for this project, since they would contain multiple instruments and multiple notes played simultaneously in melody and harmony. In theory, we could have expanded the Markov state space to include combinations of m notes, but that would have quickly become intractable (together with the more important n-gram featurization, discussed later, the number of states would be on the order of $O(127^m n)$).

Thus, a significant amount of preprocessing was needed. We identified popular, orchestral, and video game music which had clear separation between melody and harmony. We then manually edited those MIDI files so they were single track melodies. This was done over many hours in Logic Pro X, GarageBand, and Mido (Python). We include some in the attached files as a potential resource for future students.

2.2 State Representation - Nth order chains

It would be incredibly simple to create a DTMC in which there are 127 states, each representing a single musical note. However, there were multiple issues with this approach when it came to generating songs and mashups that sounded coherent.

First, songs are not made of single notes. They are made of motifs and phrases. While a DTMC with a state space of single notes might retain the 'character' of a certain song (perhaps jumping between common notes appearing in the song) it is very unlikely to recreate phrases which make a song identifiable to begin with.

Second, a smooth song mix/mashup doesn't just transition based on single notes. In an ideal situation, a good DJ would mix together songs based on longer phrases; otherwise, the songs would just abruptly transition into each other in an unappealing way.

The solution was an n-gram featurization. Rather than constructing a state space based off of single notes, our states were the value of the last n notes. This was inspired by extra reading we did on n^{th} order Markov Chains. In such an n^{th} order DTMC,

$$P(x_i|x_{i-1}, x_{i-2}, ..., x_1) = P(x_i|x_{i-1}, ..., x_{i-n})$$

Meaning that state x_i, although dependent on states $i-1, i-2, ...i-n$, is independent of states earlier than $i-n$. This slightly differs from the MC model explored in class, in which only the previous time step is considered. In doing this through an n-gram featurization, our state space became size $O(127^n)$. The theoretical utility of such an n^{th} order DTMC is that the effective 'memory' of the chain is expanded.

2.3 Song Generation

The n^{th} order model discussed in the earlier section is functionally equivalent to having a 1^{st} order model in which each state is an n-gram of the last n notes.

The model is trained given a collection of notes from a MIDI transcription. We empirically determine transition probabilities by considering every possible n-length sequence present in the song, recording a transition when one n-gram i transitions to another j, and dividing by the total number of transitions.

$$P_{ij} = \frac{\sum_{i=1}^{n}(y_i = j, y_{i-1} = i)}{s}$$

To generate a song, we then step through the chain as we would any other DTMC.

Basic methods for the object follow (for brevity, only descriptions of the basic methods are provided; the pseudocode proved too long, but a full implementation is attached).

```
class DTMC:
    def __init__(self, transition_matrix=None, state_space=None, ngram=4):
        #define a state space, transition matrix, and order of chain

    def initialize(self, state=None):
        #start chain at given state or choose from state space uniformly at random

    def update(self, notes):
        #given set of notes, parse all n-gram sequences into state space
        #update transition matrix according to method discussed above

    def step(self):
        #identify candidate states out of current state
        #choose candidate state based on transition matrix probability
        # if in 'dead state' which doesn't lead to anything
        #uniformly choose from state space for next state
        #update state, return value
```

In the end, a different DTMC is generated for each source song. In theory, we are now able to step through the DTMC to generate an original composition which is unlike, but still reminiscent of, the corresponding source song.

2.4 Mixing Part 1 - CTMC and Similarity Measurements

Stepping through a DTMC allows us to generate a unique sounding song. We now need a CTMC to mediate transitions between different DTMCs, such that we could go from one mood/song to another after some amount of time. We implement a basic CTMC as follows; again, basic methods are merely described for brevity.

```
class CTMC:
    def __init__(self, Q, pi_zero):
        #Q is transition matrix
        #pi_zero is a distribution over initial states
        self.current_state = np.random.choice(pi_zero)

    def step(self):
        row = row in Q corresponding to curr state
        times = [np.random.exponential(rate) for rate in row]
        next_state = np.argmin(times)
        trans_time = times[next_state]
        self.current_state = next_state

        return next_state, trans_time
```

The implementation is fairly straightforward; the difficulty is in determining the appropriate transition matrix to use. Each entry in the transition matrix Q_{ij} should reflect a rate of transition from song i to song j. We wanted to

transition between similar sounding songs more frequently, and we wanted the remix to transition after a reasonable amount of time. To that end, we needed to consider a similarity metric and expected transition time.

For our similarity metric, we simply examined the state space of each pairwise combination of songs and recorded the number of matching n-grams to construct a similarity matrix S. However, we cannot naively use entries in S as values in a transition matrix; while it could be a valid transition matrix, using those values as rates for an exponential RV would cause weird transition times. We needed to construct Q such that the transition times between songs were reasonable.

To accomplish this, we note that the diagonal entries Q_{ii} in the transition matrix represent the 'flow' out of a song. Expected transition time out of a given song is:

$$\text{E[transition from song j]} = \text{E}[min(X_1, X_2, ...X_n), i \neq j] = \text{E}[Exp(\lambda_1 + \lambda_2 + ... + \lambda_n) = \frac{1}{\sum_{i=1, i\neq j}^{n} \lambda_i}$$

Where X_i is an exponential RV distributed λ_i indicating transition rate to the next song. By calculating the expected transition time out of each song, we then pick a scaling factor and scale the matrix Q such that the fastest expected transition time is n seconds (we chose 5). This is a straightforward calculation.

All the above is in pseudocode below.

Algorithm 1 Generating a good transition matrix (cont.)

Data: List of n DTMCs
Result: A transition matrix with reasonable transition timings that prioritizes mixing together similar songs

S = n by n matrix of zeroes

for *each pair of DTMCs in provided list* **do**
  similarity = number of matching ngrams
  S[dtmc1][dtmc2] and S[dtmc2][dtmc1] = similarity
end

//set diagonal values appropriately to make valid transition matrix
for *each entry on the diagonal of S* **do**
  entry = -1 * sum(other entries in row)
end

//scale values to make transition matrix with reasonable transition times
fastest transition time = max(abs(entries on diagonal))
desired transition time = 5
good Q matrix = S * (1/desired time) * (1/fastest time)
return good Q matrix

2.5 Mixing Part 2 - Transposition

In order to smoothly transition between songs, we need to account for the fact that different songs can occupy vastly different note ranges. If there are no shared n-grams between songs, then we shouldn't expect "smooth" transitions since songs would just be randomly initialized after each transition independently of the previous song. Thus, we want to maximize the number of shared n-grams between songs.

Our solution is to transpose songs such that the number of matching n-grams between songs is maximized. To do this, we arbitrarily select a "reference" song. Given another song, we compute the number of matching n-grams for every possible transposition of the given song ranging from ±30 half-steps and select the transposition with the highest number of matching n-grams. We repeat this for every song (except for the reference song). In practice, we find that this fix is critical to allow for smooth transitions between most songs. The pseudocode for this is fairly trivial and will not be covered here; refer to the "sync pair" method in the source files.

2.6 Mixing Part 3 - Creating the Actual Mix

At first, this seems like a simple application of the above theory/classes/pseudocode. We generate one DTMC for each source song and then construct a CTMC based on those DTMCs. Then, for each step in the CTMC, we return a DTMC number and transition time. We step through the corresponding DTMC for the returned transition time, and repeat the process.

However, this is merely song concatenation - not song fusion! Even though DJ MARKOV is choosing to switch between similar sounding songs, it is not starting the next song at a similar place; the transition is still disconnected. To fix this, we initialize the next DTMC to the same n-gram as the final state of our current DTMC. This is only made possible by the earlier transposition step. Thus, each song transition can be connected by the same musical phrase. If that final state n-gram is not found in the next DTMC, then the next DTMC initializes at a random state.

Algorithm 3 Creating the mashup (cont.)

Data: List of n MIDI Files
Result: A song mashup

```
allDTMCs = []
for each MIDI File do
    notes = noteListFromMIDI(file)
    dtmc = new DTMC(notes)
    allDTMCs.append(dtmc)
end
//transposition subroutine from subsection 2.5
allDTMCs = transpose(allDTMCs)
//Q-matrix subroutine from subsection 2.4
Q = generateGoodTransitionMatrix(allDTMCs)
djMARKOV = new CTMC(Q)
mashupNotes = []
lastState = None
//mashup is variable length, but generally more steps in the CTMC = longer mashup. We arbitrarily pick 10 here.
for i in range 10 do
    song, time = djMARKOV.step()
    song.initialize(lastState)
    //assume a steady 196 bpm (3.26 notes per second); generate appropriate number of notes
    numSteps = 3.26*time
    mashupNotes.extend([allDTMCs[song].step() for i in range(numSteps)])
    lastState = mashupNotes[-1]
end
outputMidi = midiFromNotes(mashupNotes)
return outputMidi
```

3 Experiments

For our experiments, we simply fed different songs through the mixing pipeline (described from start to finish in section 2) and observed the results. We attempted to mix together songs from different genres (e.g. Bach, which is classical, and Pendulum, which is electronic/drum and bass), songs from different games, nursery rhymes, and whatever we thought would be interesting. We tried mixing together just 2 songs as well as multiple songs.

The only other variation we made was the parameter 'n' in the n-gram. We noticed that by adjusting 'n', DJ MARKOV could be made to improvise more or less. Intuitively, if n is large, DJ MARKOV tends to recreate the longer phrases in a source song, and if n is small, then the phrases tend to be shorter and less connected. If n were the length of the whole source song, DJ MARKOV would simply have memorized the song.

4 Results and Analysis

Most of these results are qualitative in nature and are best heard firsthand rather than discussed; we invite you to open these midi files on your Mac (GarageBand) or drag them into this app http://qiao.github.io/euphony/ to listen along! We specifically recommend listening to:

- The mash-up of assorted basic kid's tunes such as the birthday song, "Mary had a little lamb", "Old MacDonald", and others (simple.mid); this ended up sounding fairly repetitive and simple, which is indicative of the kinds of songs used to construct the MCs. But you can still hear smooth transitioning between different melodies particularly near the end of the mix
- The mash-up of Granite and Witchcraft, both by Pendulum (granitewitchcraft.mid); starting from 28 seconds into the clip, the transitions between the two songs are incredibly smooth. It likely helped a great deal that most Pendulum songs sound fairly similar, which indicates that the likelihood of shared n-grams is higher
- The mash-up of various themes associated with Pokemon routes (pokemonroutes.mid); the transitions throughout this mix are also pretty smooth. All of the Pokemon route themes are also somewhat similar in that they are designed to instill a sense of adventure, which is apparent in the final product
- The mash-up of Seven Nation Army by The White Strips and Barbie Girl by Aqua (sevenbarbiearmy.mid); the fact that these songs meshed together at all was pretty surprising, but the transitions are mostly only noticeable about 35 seconds into the mix. As it turns out, Seven Nation Army and Barbie Girl share a non-trivial number of n-grams which leads to slightly odd pairings like this one

- The mash-up of Bach's Cello Suites, Fantaisie Impromptu by Chopin, Witchcraft by Pendulum, All Star by Smash Mouth, and Numb by Linkin Park (bachfantasiewitchcraftshreknumbw+1.mid); this was a mash-up of five songs that still maintained smooth transitions. Surprisingly, the CTMC did not just end up jumping between the two most similar songs. This shows that the transposition procedure for moving songs into similar note ranges allows us to avoid contained loops and allows us to traverse the global state space across all songs reasonably well

In summary, our results were not bad - from listening to the music, you can tell that our approach can produce fairly interesting mash-ups with relatively smooth transitions, occasionally even across vastly different genre domains. There isn't much to quantitatively analyze, however.

5 Discussion and Limitations

Our mashups exceeded our expectations significantly; we were already pleasantly surprised by the result of the individual n-th order DTMCs, but the song fusions and mashups really impressed us. Though music is subjective, we thought they sounded pretty good!

However, there were significant limitations to our approach that prevented this project from reaching its full potential. The first had to do with computational tractability. We ignored chords and structures involving multiple notes at the same time; we believed they would explode our state space, as discussed in section 2.1. However, these multi-note structures are essential to music, and capture the essence of harmony against melody (rather than only melody). Secondly, we could not think of a satisfactory way to also consider rhythm and timing in music. The rhythm is intimately tied to the actual melodies harmonies themselves, and we could not figure out a way to consider both in tandem. Finally, we felt that the fixed length n-th order chain was limiting; some musical phrases/ideas are short while others are long. Given more time, we would have liked to implement a variable-order Markov model (aka context tree).

DJ MARKOV did its best under the limitations we gave it. Although the mashups themselves don't go that hard/aren't instant bangers, we hope DJ MARKOV's ability to spot connections between song motifs and melodies can aid a human in making a better mix one day.