Mini-Project: MarkovMixer Option

EE 126

Spring 2015

1 Introduction

Over the past few weeks, you have been learning about Markov chains, which have many interesting properties as well as a number of useful applications in many different fields. There are many real processes which can be described, or at least approximated, completely by Markov chains [1]. If we have a model describing a random process, then assuming it is a good model, we can simulate the process by sampling its distribution! We're interested in seeing if we can come up with a good model for musical processes using Markov Chains. In particular, we explore the problem of trying to emulate a DJ mixing together different songs.

2 Project Overview

For this mini-project, you are free to choose any of our provided ideas (PageRank or this one) for your projects, or choose your own. You're also more than welcome to use these ideas as a starting point and take it in a different direction. We look forward to seeing your great ideas!

With this as a starting point, you could do a number of things to improve this model to better emulate music. In the last section, I talk about the motivations and assumptions made for this model, so that may be a way to get inspiration. We make a lot of simplifications when formulating this, so you could make the model a bit more accurate by addressing more realistic scenarios. One such case might be to consider the case where you can overlap songs.

There are a lot of different ways you could change the model to exhibit certain behaviors. You could try to make a Markov chain model that emulates a certain genre perhaps. In addition, you could try playing around with Markov Chain properties. You could look into or analyze the Markov chain for recurrent or transient components, analyze interesting cases of hitting times, or come up with some other kind of music related First Step Equation.

Feel free to adapt the MarkovMixer code however you like, and take it in whatever direction you want. If you like music, this could be a fun opportunity to mix your artistic creativity with the probability theory you've learned

[1] At least with respect to the quantities of interest.

in this class while also addressing modeling concerns when trying to consider appropriate musical models.

In addition to the code, you will need to submit a write-up of your approach/model/improvements as well as your results.

3 How to use MarkovMixer

MarkovMixer is a simple application that attempts to model and simulate the task of creating a mix of songs from a given playlist, i.e. it tries to emulate a DJ! As you can imagine from the name, it randomly generates a mix of given songs by using a Markov chain (1^{st} order in this case) to model a DJs decision making progress to create a mix or a mashup.

Here is a general overiew of the steps that the MarkovMixer takes:

- Load music files from a directory provided by the user
- Segment each song into audio segments containing some number of beats
- Calculate the cross-correlation (or rather, a suitable proxy) between segments of the songs, providing a similarity metric between songs
- Threshold the cross-correlations, keeping only the ones above the threshold. The corresponding segments now have a transition between each other in the Markov Chain
- Map the cross-correlations to probabilities
- Start up an output stream with pyaudio
- Provide pyaudio with a callback to provide frames upon request
- Everytime pyAudio requests a frame, make advances through a random walk through the Markov chain, starting at a random state initially. When we encounter transitions, we flip a coin to determine whether we follow one of the transitions, or to continue playing the current song
- Write the output of the current song and any transitions that may have occurred and return it from the callback

At this point, you can sit back and enjoy your randomized music experience! Here's some helpful information for getting started.

Dependencies

- Python 2.7
- `numpy`
- `scipy`

- `essentia` OR `librosa` and `scikits.samplerate`
- `pyAudio`

If you have any difficulties installing anything, check the installation instructions or let anyone on the teaching staff know.

For Windows users, pyAudio can be installed from these Windows binaries. The Essentia library is very difficult to install on Windows, so you should use `librosa` and `scikits.samplerate`, which can be installed through `pip`.

After you have all of the dependencies installed, Get together a few music files and put them into an empty directory.

(**IMPORTANT:** If you use more than 20 songs at once, you may need to replace some of the data structures with databases in some form. In addition, you may need to rework some of the cross-correlation calculations to avoid `MemoryError`s)

From your Terminal/Console, you can run:

```
./markovmixer.py ~/path/to/your/songs
```

This will run the aforementioned process. Note that the initial processing may take a bit of time (due to the beat detection and all of the cross-correlations to be calculated). Once you do the initial processing, the results of the beat detection and cross-correlation are saved in python `.pkl` files in the directory where your songs are located. Unless you set the `--recompute` command line flag, it won't recompute that information if it sees that it exists. You may see some warning messages caused pyAudio regarding audio drivers. It's a well known annoyance that doesn't really cause any issues. If you do have issues, consult the teaching staff, the Internet, or Piazza for help.

After it finishes processing everything, you should hear your songs start to play! You can play around with various parameters in the model (see the command line optional arguments), and then start working on improving, changing, or ripping apart the model.

Once you've played around with the base code, take a look at the last section, where we motivate the model, as well as reveal some shortcomings which may be opportunities for ideas for part of your project!

If you find any issues with the code, let us know! In addition, ny feedback would be very appreciated so can improve this for EE 126 classes in future semesters.

For reference, here is the help message for the script:

```
usage: markovmixer.py [-h] [--num_songs MAX_SONGS] [--beats BPS]
                      [--jumps MAX_JUMPS_PER_SONG] [--spread SPREAD]
                      [--offset XCORR_SHIFT] [--probscale PROB_SCALE]
                      [--recompute] [--forcerestart]
                      [--threshold XCORR_THRESHOLD]
                      song_dir

Markov Chain Song Mixer

positional arguments:
  song_dir              The directory where the songs are located.

optional arguments:
  -h, --help            show this help message and exit
  --num_songs MAX_SONGS, -n MAX_SONGS
                        The number of songs to be used, picked arbitrarily. If
                        unspecified, all songs are used.
  --beats BPS, -b BPS   Sets the number of detected beats per segment.
  --jumps MAX_JUMPS_PER_SONG, -j MAX_JUMPS_PER_SONG
                        Sets the maximum number of jumps per song. The default
                        number is 100.
  --spread SPREAD, -s SPREAD
                        This factor affects the spread of cross-correlation
                        when they are mapped to a probability via the logistic
                        function.
  --offset XCORR_SHIFT, -o XCORR_SHIFT
                        This factor shifts the center of the logistic used to
                        map cross-correlation to probabilities.
  --probscale PROB_SCALE, -p PROB_SCALE
                        Linearly scales the transition probabilities. Used to
                        tune the overall probability of staying in a state
                        when transitions are possible.
  --recompute, -r       When set, forces recomputation of beats and cross-
                        correlation values, regardless of if they have been
                        saved.
  --forcerestart, -f    When set, forces the Markov Mixer to randomly start
                        from a new song after reaching the end, rather than
                        terminating.
  --threshold XCORR_THRESHOLD, -t XCORR_THRESHOLD
                        The cross-correlation threshold with which we reject
                        transitions associated with insufficiently cross-
                        correlated segments.
```

4 Motivation for MarkovMixer

In this section, we describe the MarkovMixer, and the motivations for its design. How might we go about simulating a DJ with Markov chains?

4.1 High Level Model

First, let us formulate a high level (and highly simplified) model for how DJs creates a mix in real time. In this simple model we will restrict our attention to the case where only a single song can play at once. We assume that DJs have a library of songs, and that DJs arbitrarily pick a song to begin their set. Of course, part of the craft of DJing is blending together different songs to create something fresh and enjoyable, so we need to consider how the process of choosing another song occurs.

Throughout each song, they decide between crossfading to a different phrase in a different song, or allowing the current song to continue playing. Our DJ is likely to crossfade to another song if a particular section of this other song sounds is very similar to the what is currently playing. Because DJs prefer to maintain their credibility, if a part of the song does not sound sufficiently similar to what is currently playing, then they don't consider it. If a part of another song doesn't sound similar, it probably will not sound good and their fans will become displeased. For the sake of further simplification of our model, we assume that DJs have no time constraints, so they can perform for as long as they want. When they have no more suitable songs to crossfade to, they decide to play the song until the end and finish their set. Now that we have an idea of what our model of a DJ does, how do we translate this behavior into the states and transitions of a Markov chain?

4.2 Markov Chain States

We can safely say that the current state is the current position in the song playing now; this is where our model DJ makes all of their (random) decisions, which depend wholly on which part of the song is currently playing. Thus, the state space consists of all of the parts of all of the other songs in the library of songs. However, a DJ does not likely consider every single sample in one song to every sample in every other songs. On the other extreme, comparing entire songs at once is probably not practical, especially if we want to consider changing songs in the middle of the current song. Therefore, we can describe our states as intervals of some reasonable length in between a single sample and the entire song. We consider every grouping k "beats" of each song to be our states (possibly corresponding to a bar or measure or a). Making sure the beat is synchronized is important, so it seems reasonable to consider groupings of beats to be states. Great, but now how do we decide how to go between these states?

4.3 Markov Chain Transitions

Recall that we claimed that we are likely to transition to a part of a different song if it sounds sufficiently similar. Thus, we want to find some similarity metric, and some function to map it to a probability. There are numerous ways to describe similarity between two objects. For signals, a common similarity metric is to use cross-correlation, which is defined for real sequences as

$$f[n] \star g[n] = f[-n] * g[n] = \sum_{k=-\infty}^{\infty} f[k]g[n+k]$$

where n describes the *lag* of g behind f. Remember that our goal is to find parts of other songs that are sufficiently similar to the current time window, so we could find the cross-correlation of the time-window with the song. Since the current time window is presumably much shorter than the prospective song, only the region overlapping with the window will be non-zero when multiplied. In effect, we get:

$$w_i[n] \star s_j[n] = \sum_{k=K_0}^{K_1} w_i[k]s_j[n+k] = \sum_{k=K_0}^{K_1} w_i[k]w_j[k]$$

where $w_i[n]$ is the segment of the current song, $s_j[n]$ is the prospective song, and $w_j[k]$ is the segment of $s_j[n]$ that overlaps with $w_i[n]$. Notice that we effectively are taking the *inner product* of the two segments of the signal. You may recall that inner products [2] are often used as measures of similarity between elements in a vector space (remember that signals are vectors!).

With this in mind, recall that we are only considering disjoint time-intervals in our state space. It is worth noting that if we simply take the inner product of the current segment of the current song, and a segment of some other song, it is the same as looking at the value of the cross-correlation when the our current song segment overlaps with the other song. The significance of this is that since we're only concerned with segments that start on the beats, we can find the cross-correlation at that point by taking a inner-product. This saves us some computation, allowing us to sample the cross-correlation sequence without actually computing it. Cool!

In addition, since negative cross-correlation still implies a that the two segments are still correlated, we can take the absolute value to account for when the segments are similar when reflected. It is worth noting that since our time intervals contain a number of "beats", which are different for each song, and even within a song, it is not guaranteed that the two sequences will contain the same number of samples. We can still believe that the inner product between the two windows will give a measure of similarity between the segments. However, we still have the problem that the measure of similarity for other segments are not exactly on the same scale since they have a different number of samples.

[2] In addition, It turns out that cross-correlation operation is also known as a "sliding inner-product".

We might heuristically claim that this isn't too bad, since we would most likely trust shorter sequences less than longer sequences in terms of how much similarity we will have further in the future. Objectively though, this is a tradeoff that comes from non-uniform segmentation of each song.

Finally, we want to map our similarity measures (cross-correlation) to the range $[0, 1]$. This choice depends on how you want to model the probability. For the sake of simplicity, in this implementation, we arbitrarily choose the logistic function as a mapping for the transition probability:

$$P[X_{k+1} = j | X_k = i] = \frac{\alpha}{1 + e^{-k(|r_{ij}| - b)}}$$

where $|r_{ij}|$ is the magnitude of the cross-correlation. We constrain α to be in $[0, 1]$ so that the quantity is still a probability. While simple, this model gives us a few parameters that we can work with to tune our model more to our liking. α dampens the probability of transitioning, so that we generally transition less. b lets us shift the center of the logistic function. k lets us control the spread of the distribution.

One more thing to consider is the case when you have multiple possibilities for transitions from the current state. There are a number of ways we could formulate a model that accounts for all possible transitions from the current state. Again, for simplicity reasons, we just divide all outgoing transition probabilities by the total number of outgoing transition probabilities.

Great! We've come up with a Markov chain that follows our model for a DJs process of real-time mixing! As you may have noticed, and as was mentioned a few times, this is extremely simple model, so there is a lot of room for improvement! What are some ways you can think of improving this model?