Jazz Melody Generation and Recognition

Joseph Victor

December 14, 2012

Introduction

In this project, we attempt to use machine learning methods to study jazz solos. The reason we study jazz in particular is for the simplicity of the song structure and the complexity and diversity of the solos themselves. A typical jazz song has an introduction, followed by each musician taking a turn improvising a melody over a fixed set of chord changes. The chord changes constrain what note choices a soloist can make, but also guide the melodic evolution of the solos. Abstractly, a jazz musician is a machine which reads in a set of chord changes and non-deterministically outputs a melody[1].

There are then two obvious problems one can attempt to solve computationally. The first is generation; can a computer be a jazz musician? The second is classification; given a solo, can a computer guess the chord changes which inspired them? One should believe the first question is possible, although admittedly hardworking musicians should always be able to beat it. One should believe the second question is possible because if one listens to a soloist without the accompaniment one can still "hear the changes", especially with the jazz greats, so long as the soloist isn't playing anything too wild. The hope is that a computer can statistically hear the changes at least as well as a toe-tapper.

The basic philosophy is this: fine a bunch of reasonably structured jazz melodies (that is, 4/4 having the format described above), chop them up so we have a bunch of chord-progression/melody-snippet pairings of the same length and analyze them to solve the two problems above. The attempted solution to the first problem was a markov chain model, while the attempted solutions to the second were (in order of success) a tree-metric-space model, a Naive-Bayes model and a hidden markov model.

Data Accumulation

There are many educational websites that offer free PDF files containing transcriptions of jazz solos from great recordings by famous artists. It is easy to write scripts that grab all the jazz PDFs from said websites. Once you have these PDF files, there is a program called PDFToMusic Pro which reads in PDFs that have nice enough sheet music on them and outputs a parse-able format called musicXML. As far as I can tell, the PDFToMusic Pro software is good at getting all the notes and their values, but less good at getting the rhythms correct. It was very difficult to fix all the output of this data, and almost half of the songs I began with needed to be thrown out (including all songs which had non-standard formats or, for simplicity, were not in 4/4 time). There were many "systematic" issues with the rhythms that could be found and

[1]This is drastic simplification of course, in reality the improvised melody is being influenced in real time by the choices of the other musicians, who themselves are responding to the soloist. I only model the soloist here. I also do not model "soul".

fixed automatically, and when songs did not pass the sanity checks after these fixes they were eventually thrown out. This part of the project was highly frustrating, but eventually I got a dataset I believed in.

The harmonic data, that is, what chords were being improvised over in each measure, was more-or-less impossible for the PDFToMusic Pro software to reliably extract from the chord information. The chord information came from Ralph Patt's "Vanilla Real Book", which is a relatively comprehensive ascii index of chord progressions from famous jazz songs. By doing about a minute's worth of semi-manual work per song (ugg), I was able to match the chord-progression data to the solo transcriptions (the issues were things like solos not starting on the first measure or songs being transcribed in a different key). Luckily, many of the songs were multiple pages, so once the chord data was matched at the beginning of the song, I would many measures worth of data.

In the end, my dataset consisted of pairs of lists of chord changes and lists of notes from the solos. My dataset came from 81 songs and contained 3013 measures of music.

Some Definitions

A NoteValue is one of the 12 note values, for instance C or G or Bb. A Note is an octave (which is an integer), a duration (also an integer) and a NoteValue. A Modality is a type of chord. Modalities include 7, Major7 (denoted M7), minor7 (-7 or m7), m7b5, dim, augmented, altered and a few more. I do not consider what musicains all "higher extensions", so many modalities end up being the same, for instance, I do not differentiate a major chord and a major 7 chord, or a dominant 13th chord and a 7th chord[2]. A Chord is a duration (an integer which records the number of beats this chord is played for), a NoteValue and a Modality. A BbM7 chord played for a measure is then the triple (Bb,M7,4). A Interval is the distance between two noteValues, and is between 0 and 11 inclusive. A D is 2 above a C and 10 below (because of the wraparound, the direction matters). We can subtract NoteValues to get an interval or add an interval to a NoteValue, Note or Chord to get a new NoteValue, Note or Chord (respectively). A datapoint is a list of Chords and a list of Notes which has been be played over them. A melody or solo is just a list of notes, and we will use these phrases interchangeably. The level of detail of a datapoint is the length of musical time of that datapoint, which ranges from 4-measure-phrases to single beats.

Data Normalization

Say that a musician plays some lick over, say, a C7 chord. If I transposed each note up a bit and played it over a D7 chord, it should be just as good. The logic is that if the computer "learns" to play over some chord progression, a shift of that progression shouldn't throw it off [3]. Thus, for any data point (that is, list of Chords and list of Notes), we can fix some NoteValue n and only consider only the Intervals between the Chords in our list and n, and the Notes in our list and n. We say that the datapoint in question is normalized with respect to n. In a certain sense, we can think of this as multiplying the size of our data set by 12, since transposed chord progressions are isomorphic. The prior of seeing, say, a C7 is now the same as seeing a D7, although the prior of seeing a C7 or a CM7 can still be different, as can the prior of certain

[2]I am trying to be light on the music theory, it is not that important, just think of it as an attribute from some list.

[3]There is a likely anachronistic story about the great musician Charlie Parker coming to a jam session having learned a certain song in Bb and the leader saying to play the song in B, just to throw the young musician off.

combinations of chords or certain chords held for certain lengths of time, and all this depends on how finely we choose to chop-up our dataset.

Markov Chains for Solo Generation

One can use Markov Chains to generate convincing-sounding jazz solos pretty easily. Since, when soloing, the chord progression is known, one cay say the state is the current chord and the last k-beats. Each step you advance the state by picking from the dataset a one-beat note selection consistent with that state and look up the fixed next chord. Of course, it is somewhat easy for this method to get stuck, especially for large k, so when it does I have it insert a few beats of rest (this is a good thing!) and then start again. The results sound convincing and the structure of the underlying chord progression is easily detectable. Unfortunately, I have no way of quantifying how well it worked.

The values of k for which I felt the midi-playback sounded best were between 4 and 8 For smaller values of k the note-choices were very in my opinion random-sounding, while for larger values of k it got lost to often.

While this was by far the simplest part of my project and least scientific, it is certainly the most entertaining and fun to show off, and possibly the biggest success.

Metric Structure on Melodies

Musical data has a natural tree structure. A phrase is divided into measures, a measure into beats, a beat into eighth notes, or perhaps triplets, etc. Storing a solo, phrase, measure, half-measure, or whatever level of detail is desired in this format is rather easy. The level of details I considered were the phrase, the half-phrase, the measure, the half-measure and the beat, each of which is twice as long as the next. The edit distance for two trees is defined as the length of the shortest path of deletions of nodes, insertions of nodes and changes of labels of nodes to get from one tree to another. More generally, one can assign a cost function to insertions, deletions and changes of labels, which could depend on the tree being operated on. I implemented Shasha and Zhang's edit distance algorithm in hopes that it would reasonably tell me how different two solos were. The real hope is that solos coming from the same progression, or snippets of solos coming from similar progressions, would have lower edit distances.

The first thing to try is of course say that all edits are equal. For various levels of detail, I created the matrix of edit distances between each datapoint in my dataset representing that level of detail, normalized with respect to the first chord in that datapoint. This approach failed pretty drastically at all levels of detail. The hope would be that the k-nearest neighbors would contain solos coming from the same chord progression, but this did not happen. On average, the value of smallest k for which the kth nearest neighbor of a datapoint came from the same chord progression was the very close to random, except for the beat level-of-detail, where the datapoints were just too close together.

The next thing to try is modifying the cost functions. Intuitively, the number of notes played (which is somehow measured more by the insertions and deletions) is less important that the note choices themselves, and inserting or deleting an eighth note is less important than a quarter note or half note, (in fact, it seems it should be half as important). Unfortunately, this did not change the output of my tests. I wanted to change the cost function of note-changing so that it fit my dataset better, but at the time when the tree-edit-distance is being computed you know only the notes, and, effectively, every note is as good as every other, only relative note choices matter, so this method was abandoned.

Naive Bayes

Since trying to say how different things are based on the tree-like structure didn't work out, we want to ask the following question: based on a bag-of-notes, what Chord or Chords inspired this. Of course, there are some details to consider, like a quarter note counts twice as much as an eighth note, etc.

In this version of Naive Bayes there are many more buckets than, say, in spam classification, because the things we are trying to classify between are chord progressions of a given length. There are also many less "tokens" in each document in each "document", since the tokens are NoteValues and the documents are melodies of a certain fixed length. Luckily, everything ended up being quite tractable, with interesting results.

For training, pick a level of detail and chop up and normalize the dataset, and for each mini chord progression make a histogram (using Laplace smoothing) of all the notes seen, using assigning points to the histogram based on the note duration. This is good for figuring out what each φ should be, but has a bit of an issue which I will discuss later. All of the apparent priors should be divided by 1/12, since each datapoint can be transposed into any of the 12 keys with equal probability, but there is no need to explicitly process each datapoint 12 times.

Interestingly enough, I found that there are only 22 harmonically unique measures in my dataset, 53 harmonically unique half-phrases and 106 harmonically unique half-phrases. There were only 6 and 9 harmonically unique beats and half-measures. These relatively small numbers boded well for my the run-time of my testing function.

For testing, one takes a snippet of a solo and sums up the total duration of each NoteValue in the measure. Then of each of the normalized chord progression observed and each of the 12 intervals, rank the combinations by probability. The reason I have it rank them rather than just say what the highest probability one is that there are many many options, and often the second or third or even fourth as the most likely was the correct answer (this is not surprising, as jazz musicians do things like "chord substitutions" in their solos, which means they intentionally imply the "wrong" chords for effect, but the histogram for the "wrong chords" will be similar enough to the histogram for the right one).

One interesting thing about testing was that rather than discreet, one-off tokens (like with documents), I have notes which play for a certain amount of time. In Naive Bayes, if you see the word "buy" a bunch of times you assume that the word was randomly selected that many times. However, the "duration" of a note is a very big number (for reasons unknown to me, a quarter note has duration 6720). Seeing a C played for a quarter note does not mean 6720 independent decisions were made. Again, for forming the histogram this does not matter, but for computing the relative probabilities one needs to compute the prior times that particular ϕ raised to the power of the "number of occurrences" (I had a nice LogFloat data-type which acted like a regular number but was internally represented as a log of a number, so I never had to think about changing multiplication to addition and exponentiation to multiplication). The effect here is that the prior would be less important, because the Naive Bayes algorithm would "think" it had over six thousand times more data than it actually did (a document with just two words, both of which are buy, is more likely to be spam than a document with just one word which is spam, regardless of the priors). My solution was to divide this exponent by 3360, the duration of an eighth note. This means that an eighth note counts as one independent choice, a quarter note counts as two choices, and so on. It also means, somehow, that a triplet counts as two-thirds a choice and a sixteenth note counts as half a choice. My results were good enough that this logical strangeness wasn't too bad a problem.

I did leave one out cross validation (implemented by "subtracting" entries from the histogram, rather than retraining the entire time), not testing datapoints with no notes. The

resulting number is the average ranking of the correct answer divided by the number of possibilities (so a score of 1 means the top ranked progression was always correct, 1/(22*12) at the level of detail of measures means it always ranked the correct answer last). My rankings were, phrase: 0.64, half-phrase: 0.7, measure: 0.69, half-measure: 0.56 beat: 0.31. Beat I think was particularly confusing for it because it had at most 3 or 4 notes to make its guess. Many, but not all, of the horribly misclassified datapoints were ones with far fewer notes than average, which could happen, say on the measure level, if a soloist is resting on that measure except on the very last beat, where they play a pickup to the next measure.

Hidden Markov Models

The last thing tried for chord progression classification was a Hidden Markov Models. I am very glad I stumbled upon them while Wikipedia-grazing late in my project, because they model the relevant question extremely well. A Hidden Markov Model is like a Markov Chain except that you do not know the current state. The idea is that there is a set of states and probabilities of transitioning between the states. At each step, an observable is emitted based on a probability distribution depending only on the state, and then the state randomly goes to a next state based on the transition function. The input of a Markov Model problem is the things observed (the solo) and the algorithm has to play detective and guess which sequences of states created that solo (the progression).

In my HMM, the states were the chord progression for the last k beats, the observables to be emitted were the one beat's worth of notes. The transition probabilities were just came from what was observed in the chord progressions in our dataset (taking whole songs at a time, rather than chopping the solo up into pieces, although we still do the normalization and 12-time covering). Like in Naive Bayes, we assumed that the note output over a one-beat period were independent and depended only on the state (that is, the harmony for the last k beats).

This is slightly different than the genie-and-urn description of HMM's, because in that description you have a complete list of observables and the probability of each observation given each state. In my case, I could just, given an observation, compute the probability I observed that given a state (this meant being unable to use HMM libraries). For testing, my first try is I use Viterbi's algorithm to compute the most likely sequence of states, which was straightforward to implement from Wikipedia's pseudo-code. This was remarkably effective in leave-one-out cross-validation. Often times the Viterbi output was spot-on, or could be off at a phrase or so. Other times it would be completely off.

I computed the average faction of chords "correctly" classified by the Viterbi output for various values of k in leave-one-out-cross-validation (again, being smart not to retrain the whole thing every time). My results were (it took a long time to run, so I only did a few values of k), for $k = 1$: 0.24, $k = 3$: 0.33, $k = 6$: 0.41, $k = 8$ 0.71, $k = 12$: 0.57. I think that k too large was confusing because it was able to have less data on each possible set of last-k-measures.

Data Stolen From (Thank You!)

1. Ralph Patt's Vanilla Real Book http://www.ralphpatt.com/VBook.html
2. Prof Bert Ligon's Transcriptions http://www.music.sc.edu/ea/jazz/transcriptions.html
3. jazztrumpetsolos.com
4. sokillingit.com