Computer-Aided Autocompletion of Cadential Harmony

Gabriel Lesnick

May 11, 2016

Abstract

The most popular Computer-Aided Algorithmic Composition systems are primarily algorithmic, having little human input to aid. We investigate the merits of more human-involved CAAC systems through analysis of our implementation of an autocomplete-type Hidden Markov Model that determines algorithmically the cadential harmony of a human-composed chord progression. We find that this system demonstrates potential to help humans compose music in creatively and stylistically consistently.

1 Introduction

Musicians have been quick to attempt to integrate computer processing techniques and electrical sound manipulations into composition and performance ever since the advent and subsequent development of computer and electrical engineering in the last century. Composers especially have taken a keen interest in technology as a vehicle for the Avant-Garde [6, 8, 9], although there is a strong following in musical analysis as well [7]. Composing with computers usually involves setting the composer's idea of music-making in the form of an algorithm in order to have the computer produce music as the composer would. Such purely algorithmic systems, whether based in automata or neural nets, whether deterministic or stochastic, are invariably limited, however, in that their composing is unsupervised. In other words, the computer is given a series of inputs and then outputs a completed work, but without any interaction with the human composer in between. Without human input, it is unclear to what degree composition algorithms, given the current progress in artificial intelligence, can actually compose [5]. However, whether it is the human programmer or the algorithm itself creating the music, the important point is that a purely contained algorithmic system that takes some input and then produces a musical output is strictly less powerful than a system that interacts with a human before outputting, simply because the external interaction gives the system a broader creative state-space in which to compose. It is of interest, then, to experiment with exactly how to set up a composition system with human interaction so as to make use of this expanded state-space and produce maximally creative music.

This idea of artist as union of human and computer can be reached equally well from the notion of musical complexity. With trends toward increasing complexity [11] and chaos [10] in contemporary music, there is more and more potential for computers to assist humans with the great tedium and increasingly unfathomable scope of immense, complicated scores. Such computer assistance could allow a human composer to focus more on form, development, and other important large-scale structural elements of a composition by automating lower-level necessities. A computer that could simplify and expedite the process of transcribing to the notated score the thoughts and ideas of the composer would increase the fidelity, and theoretically also the creative potential, of such ideas.

Between these two distinct but related pursuits of purely algorithmic composition and complex human composition lies a vast gulf of potential in Computer-Aided Algorithmic Composition (CAAC) that remains largely underdeveloped. Upcoming CAAC programs will allow human composers to achieve high degrees of complexity without unreasonable effort while also giving algorithmic composition systems the human supervision they need in order to achieve creative results [5]. Still, some extant CAAC software like OpenMusic [1] developed at IRCAM already show a lot of promise, with offshoot products like Composer's Assistant enabling Finale users to generate chords and fill in missing melodies [4].

Along this vein, we have implemented a CAAC system to perform autocompletion for chord progressions. We employ a Hidden Markov Model (HMM) to algorithmically finish (i.e. provide a cadential chord for) human-composed chord progressions. The hypothesis behind this is that in general, a good chord progression has only a few possible cadences so that a sufficiently well trained machine learning model should be able to find and suggest one of these few possibilities. This system is obviously only the beginning of what would be necessary to unite the compositional abilities of human and machine. However, the system we have created, rudimentary and low-level as it may be, gives evidence towards the immense creative potential available from such a union.

2 Implementation

Our CAAC program is written in Python (see Appendix) using music21 [2] to process the MIDI-formatted music data and the hmmlearn [3] branch of scikit-learn to power the HMM. The program consists of three overarching stages, namely those of data processing, HMM initialization, and finally cadence prediction.

In the data processing stage of the program, a selected corpus of music data is collated and made python-compatible, from which the final cadence of each piece in the corpus is extracted. In the next stage, system initialization and training, the number of states for the HMM is optimized and the system is trained on the cadence data. In the final stage, the trained HMM is used to suggest a cadential chord, either by using the HMM's score function to assess randomly generated cadences or by using the transition probabilities of the HMM to directly emit a probabilistically sampled cadence.

Setting up the training data well is one of the most important things to do to get good results from an HMM. To this end, our program interfaces with two large datasets from which we can extract thematic subsets to train the program. The first dataset is the corpus that comes installed with music21 and from which subsets can be generated by a simple keyword search. We used this corpus, for example, to generate a data subset of 471 cadences from various Bach

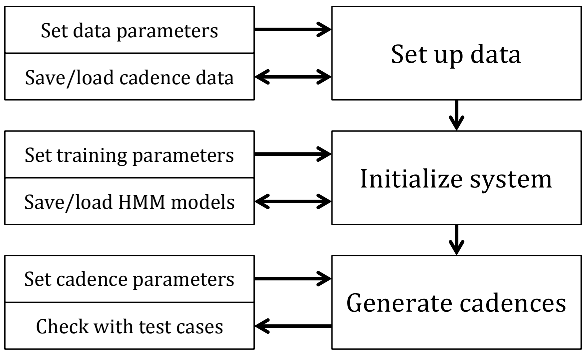

Figure 1: System block diagram.

pieces using the keyword “bwv”. The second dataset is from the website *kunstderfuge.com*, which hosts a wide range of sequenced MIDI files for pieces ranging from Baroque to Modern. This MIDI data can be easily manipulated with music21 and lets us create more contemporary-leaning training data than the music21 corpus.

After collecting a large and complicated data subset from one of these sources, the next step of the program is to extract cadential chord progressions from each piece in the dataset. For simplicity, and because we have access to so many pieces, we decided to extract just one cadence from each piece, namely the final cadence since it is the easiest to isolate. The pieces from which we extract the chords, however, are not necessarily very regular in that there may be passing chords, a variable number of notes per chord, and a different progression length in each piece. In order to make these progressions compatible with the HMM, then, we regularize them by ensuring that each progression has the same number of chords and that every chord has the same number of notes. We use the global parameters `CHORDS_PER_CADENCE` and `NOTES_PER_CHORD` to specify these numbers, and in addition we also use the `BEAT_STRENGTH_MINIMUM` parameter to pick chords only on strong beats to avoid passing chords. One final thing that must be done to ensure consistency is to normalize each chord progression to the same key. We use the key of the cadential chord, calculate by music21, for this purpose, subtracting the key and then finding the remainder modulo 12 to normalize each note to a value between 0 and 11. These methods for regularization were chosen for their ability to make the data compatible with the HMM, but it is certainly possible that there are better methods. This would be a good task for future work.

One final consideration for the training data is that in order for the HMM to “know” that the data it sees is cadential, the data must have some structure in addition to the note values of the chords. The additional structure we impose is an extra field designated to mark progress towards the cadence. In other words, if `NOTES_PER_CHORD` is set to 4, for example, then each data point fed to the HMM will actually consist of five values: the four note values and an extra value designating whether or not this chord is a cadence. For the sake of control, this cadence-marking ability can be configured, so that the `CADENCE_MARKER` value determines the weight of the cadence mark. If so desire, the `cadenceMark` function could be reset as well to implement any other algorithm for cadence marking.

Initialization of the system can take place once the training data has been processed. We instantiate a `GaussianHMM` object and then call its `fit` method on each of the six-chord cadential progressions to train it. Before we can train, however, we have to ensure the HMM is properly configured with an optimal number of states.

The program allows the user to either specify a number of states, or if a good number is not known, the program can be made to determine the optimal number of states.

The method used for automatically optimizing the number of states employs the training data to self-assess how accurately the HMM fits this data. Starting with a small number of states, we train a model HMM on half the training data. We then use this model system to score the other half of the training data. We repeat this training and scoring process, incrementing the number of states, until the score stops increasing. We want to find the number of states that best fits the data without overfitting - the number of states reached when the score stops increasing is a very good (and efficient) estimate of this optimal number.

Generating the output cadence becomes possible once the system is properly initialized. Given a complete cadential chord progression without a cadence, the HMM can algorithmically suggest the cadence. Our program actually offers two methods for suggesting this cadence, each tailored to a slightly different functionality. The first method is to search the space of all cadences and locally maximize the chord progression's score with stochastic gradient ascent while the second method is to emit a cadential chord using the transition probabilities of the HMM itself and the highest-likelihood sequence of states associated with the inputted chord progression.

The gradient ascent method makes the most sense when the cadence-marking option is enabled. Specifically, we can force this method to provide us with a cadential chord by constraining the state-space from which the potential chords are picked to the set of chords that are marked as cadential. In other words, since we are (stochastically) choosing chords to feed into the gradient ascent, we can generate these chords so that they are definitely cadences. This is as opposed to the second method, where we have no ability to constrain the chord that is generated. In the second method, however, we are more likely to get high-fidelity data since instead of just randomly searching the state-space of possible cadences for a good match, we use the preexisting structure of the HMM itself to decide what the most likely chord is. So while the first method is only guaranteed to find a local maximum, this method will theoretically find the global maximum for the best-fit cadential chord, assuming that such a global maximum exists.

3 Results

We used a number of test cases to see how our program fares in comparison to real human composers. Since this program is concerned primarily with new composition, we used mostly modern or contemporary test cases so as to test capacity for contemporary ideas. For completeness, though, we also include a sample from Beethoven.

All of these test cases are generated from an HMM trained on a corpus of Bach music. This dataset obviously gives the system a bias towards consonance and tonality, but for contemporary pieces that are quasi-tonal enough to be considered to have chord progressions, this is of interest. In Figures 2-6, the original cadential progression is followed by two generated cadences, the first generated by gradient ascent and the second from state emission.

Note that we did not yet get good data for these test cases from any training sets besides

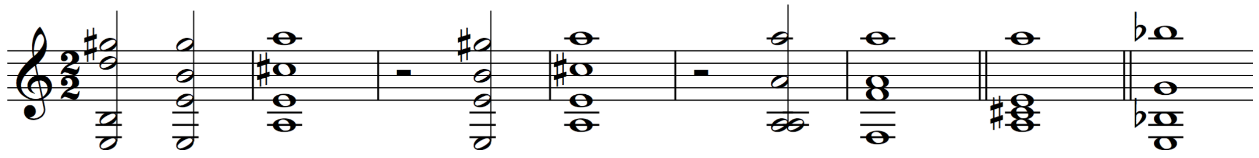
Figure 2: Beethoven - Symphony No. 9, cadence to Turkish March.

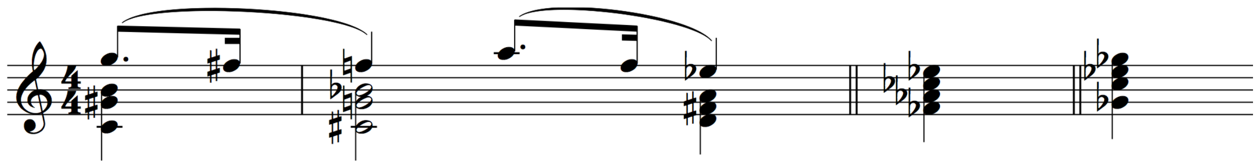
Figure 3: Schoenberg - Chamber Symphony No. 1, Adagio theme.

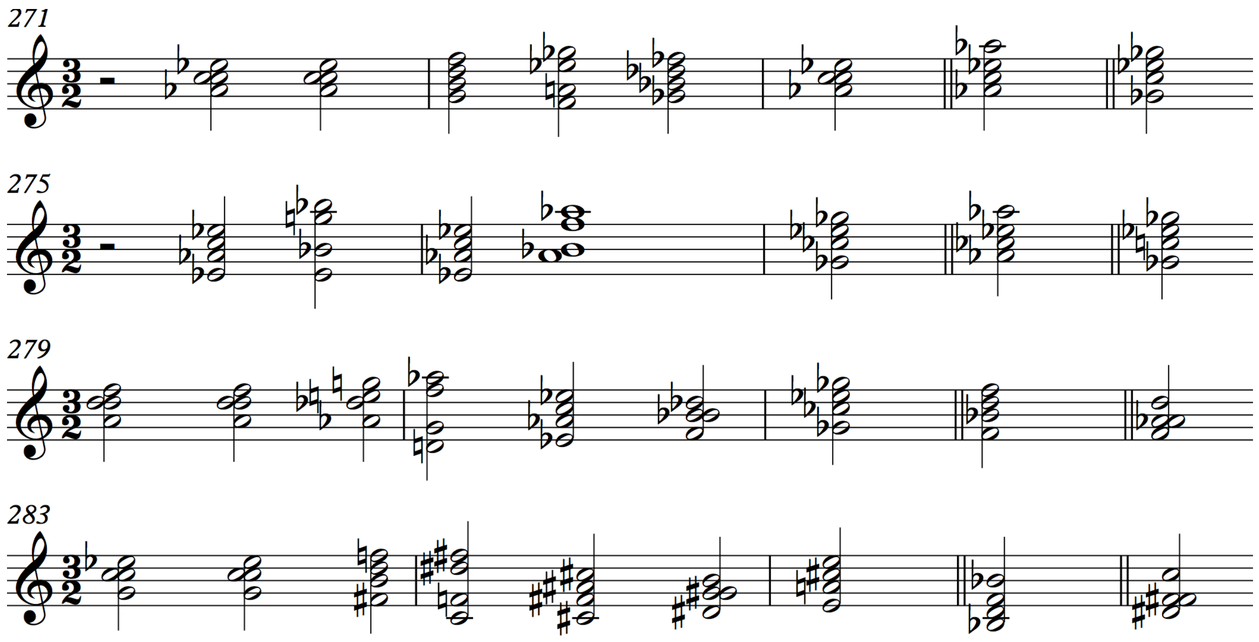

Figure 4: Lutosławski - Concerto for Orchestra, mm. 271-286.

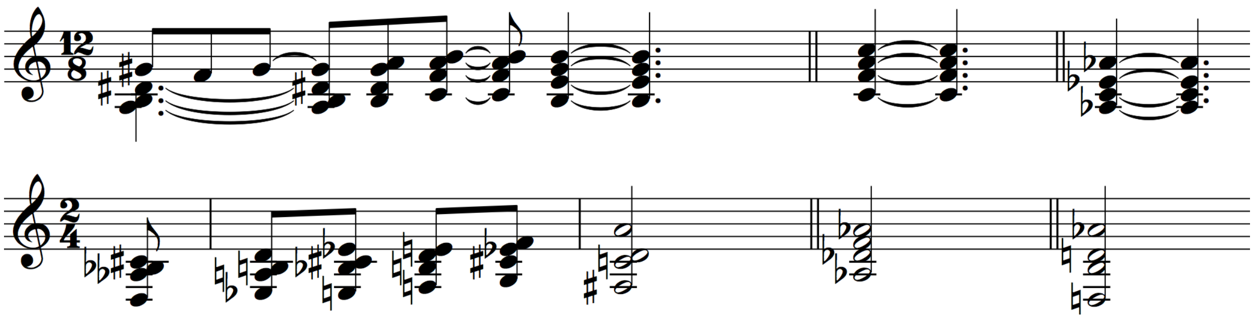

Figure 5: Messiaen - L'ascension, First and Third movements

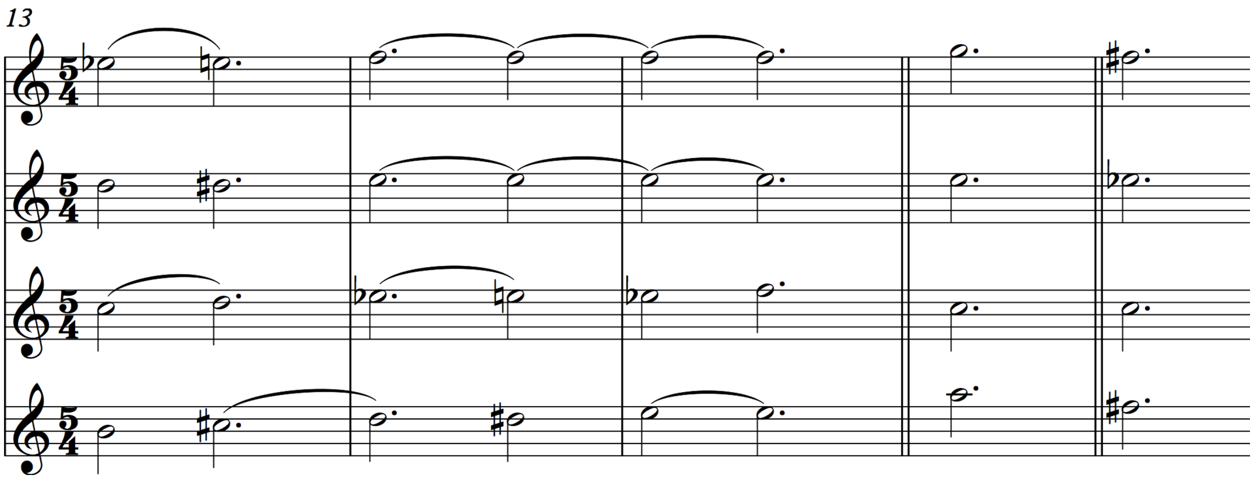

Figure 6: Ligeti - Hamburg Concerto, Hymnus mm. 13-15.

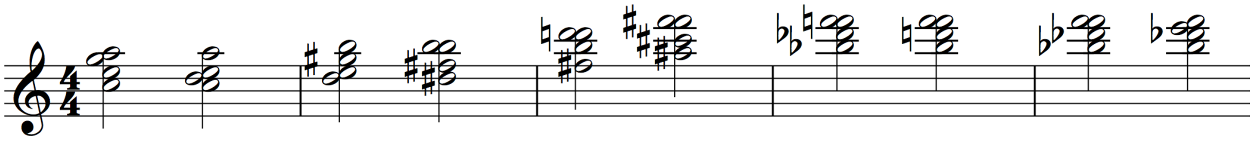

Figure 7: Chord sequence from random seed.

the Bach corpus. The *kunstderfuge.com* data is interesting to test the system with but is unfortunately not at a sufficient stage of development to enable meaningful results.

In any case, there is still plenty to be gleaned from just this Bach data, especially because the Bach data is easy to understand - the well-defined tonal nature of Bach's progressions makes it possible to distinguish between what the HMM is getting from the training data and what it understands about the style of each test case itself. It is clear, for example, that as the test cases go from triadic harmony (Beethoven, Figure 2) to pure atonality (Ligeti, Figure 6), the generated chords remain essentially triadic. This is of course a result of training the HMM on triadic data - it only produces chords of a kind that it has seen before. This is a good sign! It means the HMM is actually picking up relevant information from the training data. It would be good to see how this would work with a full compendium of contemporary data to make sure this training works universally - it is possible, although seemingly unlikely, that the current success of the training with Bach data is due to the specific constraints set up in the parsing of the data and that other data would not be as successful. Further testing would be necessary to completely understand this.

Continuation beyond the cadence is an obvious extension for a chord progression autocompleter. Even though the program is designed, in its current form, to produce only single chords, it is still intriguing to see what results from such a misuse. To do this, we began with a random seed chord produced by sampling the trained HMM (trained still with Bach) and then added "cadential" chords one at a time using the gradient ascent method. One such realization of this process is seen in Figure 7. Although this result is interesting, many other runs quickly stabilized around a single chord, usually a triad.

4 Conclusion

It is evident from the test cases that our program is able to emulate a lot of the ideas that human composers have about chord progressions. Elements like voice leading, triadic harmony, and the dominant-tonic cadential relationship clearly come through from these examples. However, it is hard to tell how much the system actually "understands." A lot of these factors are simple to approximate but very difficult to emulate exactly. Voice leading, for example, is easy to approximate heuristically with the simple rule that each voice should move by at most two semitones between each chord. However, leaps larger than a whole tone are obviously important in standard voice leading and this rule would leave them out. So it is necessary to assess just how much the HMM is able to get from the training data and on the contrary how much of the success of the program is due to the rule-based musical constraints we have added to it.

Ability to match human creativity is one way to assess the HMM, and comparing the generated cadences in the test cases to the original cadences is a great way to do this. A brief look at the test cases shows that at least voice leading is well emulated, and it is apparent that also the style of each test case is at least partially preserved in the generated cadences. One obvious thing that is missing in the generated cadences, however, is that they are never exactly the same as the original cadence, meaning that the HMM is not as powerful as the humans it is trying to emulate.

The success of the voice leading, although impressive, is also a bit deceptive. In order to make the HMM indifferent to key, the input chord progression is normalized to be rooted on C0. The notes of the generated cadence are then octave-transposed until they are as close as possible to those of the penultimate chord. So the octave positions of each chord are not determined by the HMM. Still, the fact that the generated chords are so often within a semitone or two of the preceding chord is encouraging, since this variation within the octave is indeed decided by the HMM.

Even more encouraging is the fact that the generated chords seem to have approximately the same style as the original cadences, indeed the same style as the rest of the chords in the test cases as well. In the Beethoven test case (Figure 2), for example, all the chords are triadic, so the generated cadences are triadic as well. Similarly in the Schoenberg test case (Figure 3), the chords are not as consonant but are still mostly triadic, so the generated cadences are again triadic. The Ligeti (Figure 6), being manifestly non-triadic, is completed with a much more condensed seventh chord, which despite being made from stacked thirds, is more dissonant than the pure harmonic cadences of the Beethoven or Schoenberg. The fact that all the generated cadences consist of stacked thirds might be disconcerting until you remember that we used Bach as training data. So with more contemporary (less triadic) training data, we would perhaps see a greater degree of variation in consonance or dissonance of the generated cadences depending on how consonant or dissonant the test case is.

Now the lack of exact replication of the original cadences by the HMM is troubling, since the HMM probabilistically considers most possible cadences and therefore rejects the original cadence as inferior to whatever cadence it comes up with. So further fine-tuning would be necessary before this program could more precisely emulate the original human composers of these test cases. However, we are looking for evidence of creativity in this system, not ability to copy humans exactly, so this is beside the point. A human composer given these test cases and asked to complete them with their own cadences would similarly have trouble recreating the exact originals but might still be able to come up with some creative solutions of their own.

Potential for new creativity is another good way to assess the HMM. Ignoring the original cadences in the test cases, what can be said about the ability of the HMM to be creative?

The most readily apparent shortcoming of this HMM is its inability to recognize global patterns like tonal center or position within the progression. In the Lutosławski test case (Figure 4), the generated chords are of the correct (triadic) sonority, but are in the wrong key. Now in some ways this is irrelevant; the HMM simply has a predilection for progressive tonality. But without an ability to generate chords based not just on immediate neighbors but also greater structure like tonal center or form, this HMM will be doomed.

In a smaller scale, though, without regard to larger patterns like this, the HMM performs spectacularly. In the Messiaen test case (Figure 5), the HMM generates novel, creative cadences that sound good despite being in the "wrong" key. Unless the listener already knew the piece or knew Messiaen's style very well, they would most likely hear the generated cadences as correct.

The HMM is clearly able to come up with interesting chords with good notions of voice leading and consonance and dissonance. Of course, there are still problems to be worked out. The HMM is not always this good; it often takes a few runs before a good chord is generated. Better training data would have to be collected to test its capability for more contemporary styles. The constraints and parameters of the program would have to be modified to fully optimize it. So it may not be perfect, but it definitely passes the mark for creativity.

5 Future Work

Although we have made significant progress in chord progression autocompletion, this is just a small feature with respect to the full potential of realizable CAAC systems. Even just within the subdomain of chord autocompletion, there is a lot more room for innovation. We added a lot of constraints, for example, to make the program easier to build, like all chords having the same number of notes, all training progressions having the same number of chords, and allowing only one cadential chord to complete each progression. It would be worth spending some time in the future unconstraining this program, or researching to see if there are perhaps alternative constraints that would be more conducive to creating cadences and would provide better results. It is doubtless that, given the inclination, a vast number of potential avenues for improvement of this program could be pursued; we will present just a few of these now.

Creating a good GUI for this program would make it utilizable for its intended purpose: helping humans compose. It is still useful as proof of concept in its current state, but it would only be useful in during composition if it were fleshed out with an interface. Ideally, the composer would use notation software like Sibelius or Finale to write music and would be able to invoke this progression auto-completion feature as a plugin with one click of a button. The plugin would suggest a cadential chord to the user and if it were to the user's liking it would then be added to the score. This functionality as described is a long way away. But it is good that we now have a proof of concept for this kind of plugin - we have taken the first step.

Training with more accurate data would help to make the HMM more accurate. However, this is much easier said than done. The act of accurately isolating and homogenizing data from a variety of different musical styles and eras is close to intractable. Even just within the Bach data, for example, it is exceedingly difficult to algorithmically decide which chords are part of the functional harmony, which notes are not passing tones, and where a chord progression starts and ends. It almost seems appropriate to create an entire HMM to process the music data.

Having a meta-system like this to process data to train the real system is impractical, but the end result of having good training data is necessary to get the full system working with high fidelity. Without high quality training data, there is no chance that the HMM will be able to pick up on the full complexity and nuance of music composition. The question now is how to get high quality data.

Generating more than one chord at a time would allow for more novel capabilities. Chord progression autocompletion would itself be benefitted by the possibility of multi-chord cadences since then chord progressions could be successfully completed in ways potentially more clever and creative than just appending a single chord. This idea could be extended further, however, to such novelties as modality bridging, that is, automatically generating a chord or set of chords to pivot between modalities, or even melody harmonization. Such higher level modes of chord generation would necessitate better and more fluid ways of representing chord data and would require reformulation, or removal, of some of the other constraints set in the current version of the program beyond just the number of chords generated at a time. For example, pivoting from a modality expressed in three-part harmony to one expressed in four-part harmony would only be possible if the constraint that all chords have the same number of notes were lifted.

Other machine learning models might be better suited to chord autocompletion than Hidden Markov Models. HMMs make intuitive sense since music often feels as if it proceeds from state to state, with recurring musical elements connected by transitional material in often well-defined ways. But it is entirely possible that such a heuristic idea misses some other sort of machine learning model that might behave better. Just because music might seem Markov-like does not mean that the best way to create music is with an HMM - at least, most human composers would balk at the idea of writing music in such a way. This is not to say that HMMs are badly suited for the task at hand of chord generation, but only that it would definitely be worth assessing the capabilities of other models as well.

A few small bugs still remain that should be fixed. In particular, the algorithm that trains the HMM conflicts with the cadence marking when this functionality is activated, often producing an error from degeneracy. We are currently getting around this by keeping the `CADENCE_MARKER` value small (a value of around 0.3 works) so as to minimize the degeneracy in the data and let the HMM training proceed. It would be nice, however, to be rid of this degeneracy altogether, but unfortunately this is work for the future.

References

[1] http://repmus.ircam.fr/openmusic/home.

[2] http://mit.edu/music21/.

[3] http://github.com/hmmlearn/hmmlearn.

[4] Products of interest. *Computer Music Journal*, 26(2):113–125, 2002.

[5] C. Ariza. The interrogator as critic: The turing test and the evaluation of generative music systems. *Computer Music Journal*, 33(2):48–70, 2009.

[6] A. R. Burton. Generation of musical sequences with genetic techniques. *Computer Music Journal*, 23(4):59–73, 1999.

[7] M. S. Cuthbert and C. Ariza. music21: A toolkit for computer-aided musicology and symbolic music data. In *International Society for Music Information Retrieval, 11th Conference*, 2010.

[8] S. R. Holtzman. Using generative grammars for music composition. *Computer Music Journal*, 5(1):51–64, 1981.

[9] K. McAlpine, E. Miranda, and S. Hoggar. Making music with algorithms: A case-study system. *Computer Music Journal*, 23(2):19–30, 1999.

[10] R. Steinitz. Music, maths and chaos. *The Musical Times*, 137(1837):14–20, 1996.

[11] B. Truax. The inner and outer complexity of music. *Perspectives of New Music*, 32(1):176–193, 1994.

6 Appendix: Python Code

```python
import numpy as np
from hmmlearn import hmm
import pickle
from music21 import *

# constants used in isolating chord progressions from corpus data
NOTES_PER_CHORD = 4
CHORDS_PER_CADENCE = 6
BEAT_STRENGTH_MINIMUM = 0 # set to nonzero to select for strong beats
CADENCE_MARKER = .3 # set to nonzero to use cadence marker
NUMBER_STATES = 13 # set to zero to use state optimization algorithm

def cadenceMark(data, lists=False, cadence=False):
    if lists:
        for chords in data:
            for chord in chords[:-1]:
                chord += [0]
            chords[-1] += [CADENCE_MARKER]
    elif cadence:
        data += [CADENCE_MARKER]
    else:
        for chord in data[:-1]:
            chord += [0]
        data[-1] += [CADENCE_MARKER]


# method to extract list of chord progressions from corpus
def getCorpusData(searchTerm):
    # try to load data from searchTerm
    try:
        dataFile = open('corpus_subsets.txt', 'r')
        try:
            dataDict = pickle.load(dataFile)
            if searchTerm in dataDict:
                data = dataDict[searchTerm]
                if CADENCE_MARKER != 0:
                    cadenceMark(data, lists=True)
                return data
        except EOFError:
            dataDict = {}
    except IOError:
        dataFile = open('corpus_subsets.txt','w')
        dataDict = {}

    corpusMetadata = corpus.search(searchTerm)
    corpusData = [piece.parse() for piece in corpusMetadata]
    chordData = [flattenChords(piece.chordify()) for piece in corpusData]
    data = [getCadentialHarmony(piece) for piece in chordData]
    data = [piece for piece in data if piece != []]

    # save completer with pickle
    dataDict[searchTerm] = data
    dataFile.close()
    with open('corpus_subsets.txt', 'w'): pass
    with open('corpus_subsets.txt', 'w') as dataFile:
        pickle.dump(dataDict, dataFile)

    if CADENCE_MARKER != 0:
        cadenceMark(data, lists=True)

    return data

# translate music21.chord to list of midi pitches
def chordToList(chord):
    return [pitch.midi for pitch in chord.pitches]

# translate list of midi pitches to music21.chord
# NOTE: must add and subtract 12 to keep correct octave
def listToChord(chordList):
    c = chord.Chord([note + 12 for note in chordList])
    return c.transpose(interval.Interval(-12))


# translate list of lists of midi pitches to music21.stream
def makeChords(chordList):
    s = stream.Stream()
    for midiChord in chordList:
        tempChord = listToChord(midiChord)
        tempChord.type = 'whole'
        s.append(tempChord)
    return s

# translate music21.stream to list of lists of midi pitches in progression
def flattenChords(chords):
    chordList = []
    # normalize chord data to root of first chord
    for chord in chords.recurse().getElementsByClass('Chord'):
        # filter out passing chords
        if chord.beatStrength >= BEAT_STRENGTH_MINIMUM:
            tempChord = chordToList(chord)
            root = chord.root().midi
            tempChord = getDistinctPitches(tempChord)
            chordList += [tempChord]
    return (chordList, root)

```

```python
def getDistinctPitches(chord):
    pitches = []
    for pitch in chord:
        p = pitch % 12
        if p not in pitches:
            pitches += [p]
    pitches.sort()
    return pitches

def getCadentialHarmony(piece):
    chordList = piece[0]
    root = piece[1]
    if len(chordList) <= 5:
        return []
    cadence = chordList[-6:]
    for i in range(6):
        diff = (NOTES_PER_CHORD - len(cadence[i]))
        if diff >= 1:
            cadence[i] = cadence[i] + [cadence[i][0]] * diff
        elif diff <= -1:
            cadence[i] = cadence[i][:NOTES_PER_CHORD]
        cadence[i] = [(note - root) % 12 for note in cadence[i]]
    return cadence

# preliminary method to train HMM system (completer) on music21.corpus data
def getCompleter(name = 'bwv'):
    print "Training␣HMM."

    chordData = getCorpusData(name)
    lengths = [len(piece) for piece in chordData]
    data = [note for chordNotes in chordData for note in chordNotes]

    # optimize number of states if NUMBER_STATES == 0
    if NUMBER_STATES == 0:
        print "Optimizing␣number␣of␣states..."

        dim = len(lengths) / 2
        trainLengths = lengths[:dim]
        trainData = data[:sum(trainLengths)]
        scoreData = chordData[dim:]

        score = -999999
        prevScore = -1000000
        numStates = 0

        while prevScore < score:
            numStates += 1
            completer = hmm.GaussianHMM(n_components = numStates)
            completer.fit(trainData, trainLengths)

            prevScore = score
            scores = map(completer.score, scoreData)
            score = sum(scores) / len(scores)

        numStates -= 1
        print "Optimal␣number␣of␣states␣is␣", numStates

    # otherwise use given number of states
    else:
        numStates = NUMBER_STATES

    completer = hmm.GaussianHMM(n_components = numStates)
    completer.fit(data, lengths)

    return completer

completer = getCompleter('bwv')

# emit next chord from HMM
def getNextChord(data, show=False, completer=completer):
    # normalize data to root
    root = listToChord(data[0]).root().midi
    chords = [[(note - root) % 12 for note in chordNotes] for chordNotes in data]
    states = completer.predict(chords)
    state = states[-1]

    completer.set_params(random_state = state)
    cadence = map(lambda note: int(round(note)),list(completer.sample()[0][0]))
    completer.set_params(random_state = None)

    # renormalize to account for "key"
    for i in range(NOTES_PER_CHORD):
        delta = (cadence[i] - data[-1][i] + root) % 12
        if delta >= 6:
            delta -= 12
        cadence[i] = data[-1][i] + delta

    data += [cadence]

    # display chord progression with cadence
    if show:
        chords = makeChords(data)
        chords.show()

    return data

# perform gradient ascent on cadence to find cadence that best fits data
def ascent(data, show=False, completer=completer):
    print "Performing␣gradient␣ascent..."

    # normalize data to root
    root = listToChord(data[-1]).root().midi
    chords = [[(note - root) % 12 for note in chordNotes] for chordNotes in data]

    if CADENCE_MARKER != 0:
        cadenceMark(chords)

    dim = NOTES_PER_CHORD
    delta = 1000
    prevScore = -1000
    prevCadence = chords[-1]

    # iterate gradient ascent until maximum is reached
    while delta >= .0001:
        score = prevScore
        cadence = prevCadence

        # check neighboring positions in chord-space stochastically
        moves = np.random.randint(-3, 4, (1000,dim))
        possibilities = [[prevCadence[i] + move[i] for i in range(dim)]
                         for move in moves]

        if CADENCE_MARKER != 0:
            for pos in possibilities:
                cadenceMark(pos, cadence=True)

        # find possibility with greatest score increase
        for tempCadence in possibilities:
            tempScore = completer.score(chords + [tempCadence])
            if tempScore > score:
                score = tempScore
                cadence = tempCadence

        delta = score - prevScore
        prevScore = score
        prevCadence = cadence

    print "Ascent␣complete.␣Score␣is", score

    # renormalize to account for "key"
    for i in range(NOTES_PER_CHORD):
        delta = (prevCadence[i] - data[-1][i] + root) % 12
        if delta >= 6:
            delta -= 12
        cadence[i] = data[-1][i] + delta

    output = data + [cadence[:NOTES_PER_CHORD]]

    # display chord progression with cadence
    if show:
        chords = makeChords(output)
        chords.show()

    return output

# chord progression test cases

test_beeth = [[52,59,74,80],[52,64,71,80],[57,64,73,81],[52,64,71,80],
               [57,64,73,81],[57,57,69,81],[53,65,69,81]]
test_schon = [[60,68,71,79],[61,67,70,77],[61,67,70,81],[62,66,69,75]]
test_luto1 = [[68,72,72,75],[68,72,72,75],[67,71,74,77],[65,69,75,78],
               [66,70,73,76],[68,72,72,75]]
test_luto2 = [[63,68,72,75],[70,63,79,82],[63,68,72,75],[68,70,77,80],
               [66,71,75,78]]
test_luto3 = [[69,74,74,77],[69,74,74,77],[68,73,76,79],[62,67,77,80],
               [63,68,72,75],[65,70,70,73],[66,71,75,78]]
test_luto4 = [[67,72,72,75],[67,72,72,75],[66,71,74,77],[60,65,75,78],
               [61,66,70,73],[63,68,68,71],[64,69,73,76]]
test_mess1 = [[57,59,63,68],[57,59,63,65],[57,59,63,68],[59,63,68,69],
               [60,65,69,71],[59,64,68,71]]
test_mess2 = [[50,56,58,61],[51,57,59,62],[52,58,61,63],[53,59,62,64],
               [55,61,63,65],[54,60,62,69]]
test_liget = [[71,72,74,75],[73,74,75,76],[74,75,76,77],[75,76,76,77],
               [76,75,76,77],[76,77,76,77]]
```