Markov Chains for Computer Music Generation

Ilana Shapiro

1 Introduction

Markov chains have long been useful in the generation of music, dating back to 1757 with the randomized "musical dice game." They have been vital to the field of computer music since 1957, when they were used to successfully compose a string quartet called the *Illiac Suite*. In mathematics, a Markov Chain is defined as a stochastic process (a collection of random variables) in which the probability of entering state X_{t+1} depends only on the previous state X_t. In other words, the distribution of each state depends only on the previous state, giving Markov chains a "memoryless" property. This paper will explore how a simple Markov chain, trained on an input piece, can produce music in the same style as the input.

Basic knowledge of musical representation (to the level of reading music) as well as fluency in Python is assumed.

2 Theoretical foundations of Markov chains

To give the formal definition of a Markov Chain, we first need to define *filtration*.

Let us first consider what a random variable is. We call X a *random variable* when it takes on values which are unknown, but for which we have partial information. A *stochastic process* is a collection of random variables.

Say that random variable X is measurable with respect to σ-algebra F if $\sigma(X) \subseteq F$. [2]

$\sigma(X)$ means the σ-algebra *generated* by X. Before discussing what this means, let us first define what a σ-algebra is. Let Ω be a set, and let G be a collection of subsets of Ω. We call G a σ-algebra when

1. G is closed under complement (i.e. $(\forall A \in F)(A^C \in F)$)
2. G is closed under countable unions (i.e. $(\forall A_1, A_2, \ldots \in F)(\bigcup_{i=1}^{\infty} A_i \in F)$) [2]

When a σ-algebra F is generated by a random variable X, we treat X as a function applied to a standard uniform random variable (i.e. this is shorthand for $X(U)$, where U is uniformly distributed over the set $[0, 1]$). F is all sets A such that if we know the value of $X(U)$, we can determine if U is in the set A, or not. [2]

More formally, this is saying the same thing as saying

$$\sigma(X) = \{X^{-1}(A) : A \in F\}$$

Now we are ready to define filtration.

A sequence of σ-algebras $F_0, F_1, \ldots$ is a filtration if $F_1 \subseteq F_2 \subseteq F_3 \subseteq \ldots$. The random variables $\{X_t : 0 \leq t \leq \infty\}$ are called *adapted* to the filtration F_t if, for every t, X_t is measurable with respect to F_t (i.e. we know whether the outcome at time t is (or is not) in every subset in F_t). [2]

Intuitively, if we know all the data in filtration F_t, we should be able to minimally determine X_t (i.e. X_t cannot be determined with less information).

The intuition behind filtrations is that the amount of information we know increases over time, and we assume no information is lost. Think of filtrations as record-keepers; as more information becomes available, our filtration size grows as we record the incoming information.

At time t, consider the corresponding filtration F_t. F_t will be able to (minimally) tell us anything that we could possibly want to know about the stochastic process in question at time t.

Let's now look at an example of a stochastic process with filtrations. Consider a stochastic process that consists of throws of a single 6-sided die. At time $t = 0$, before you have thrown the die at all, all the information we want to know is what the possible outcomes are, which are $\{1, 2, 3, 4, 5, 6\}$, as well as the question "did nothing happen" (corresponding to $\emptyset$). Therefore, $F_0 = \{\{1, 2, 3, 4, 5, 6\}, \emptyset\}$, a σ-algebra that can minimally answer any question you might have. [1]

After we have thrown the die once (so at $t = 1$), we now have more information. We have gotten a single result $d_1 \in \{1, 2, 3, 4, 5, 6\}$. We should be able to ask any question about the stochastic process now, including "was the result of the die roll a 1" (which corresponds to asking about the contents of the set $\{1\}$), "was the result even" (which corresponds to asking about the contents of the set $\{2, 4, 6\}$), "was the result less than or equal to 3" (which corresponds to asking about the contents of the set $\{1, 2, 3\}$), etc. Any question about the the stochastic process can now be answered minimally with the collection of subsets $\wp(\{1, 2, 3, 4, 5, 6\})$, where $\wp$ is the power set. [1]

Therefore, $F_1 = \wp(\{1, 2, 3, 4, 5, 6\})$, which is a σ-algebra that minimally answers all our questions. Note how more information gives us a larger σ-algebra. We have satisfied $F_0 \subseteq F_1$, and can continue this process indefinitely.

Now that we have defined filtration, we are ready to define Markov chains.

A stochastic process $X_0, X_1, \ldots$ is a Markov chain with respect to filtration F_t if for all t,

1. X_t is F_t measurable and

2. $[X_{t+1}|F_t] \sim [X_{t+1}|X_t]$.

Let's break down what each of these components mean.

First, consider the statement "X_t is F_t measurable." As we previously discussed, X_t is F_t measurable if $\sigma(X_t) \subseteq F_t$. $\sigma(X_t)$ refers to the smallest filtration against which X_t is measurable. [2]

Now consider the statement $[X_{t+1}|F_t] \sim [X_{t+1}|X_t]$. This means that X_{t+1} (the state of

the stochastic process X at time $t+1$), given F_t (the filtration at time t), has the same *distribution* as X_{t+1} given X_t (the state of the stochastic process X at time t). In other words, the distribution of the next state given the entirety of the history is the same as the distribution of the next state given only the previous state. This is the *memoryless* property of Markov chains mentioned at the beginning. [2]

We have not yet discussed what a distribution is in probability. A function $\mathbb{P} : F \to [0, 1]$ is a probability distribution if it has the following two properties:

1. Total probability

2. Countable additivity

Total probability means that for any $\Omega \in F$, $\mathbb{P}(\Omega) = 1$.

Countable additivity means that if we have $S_1, S_2, \ldots$ that are all subsets of Ω, such that $\forall i \neq j$, $A_i A_j = \emptyset$, then

$$\mathbb{P}(\bigcup_{i=1}^{\infty} S_i) = \sum_{i=1}^{\infty} \mathbb{P}(S_i)$$

In other words, countable additivity means that the probability of the union of all the S_i is the same as the sum of the probabilities of the individual S_i. [2]

Random variables by definition all have distributions associated with them. On a high level, the distribution of a random variable will tell you the possible values for the random variable as well as how the random variable is expected to fall amongst the data.

Let us now return to Markov chains. We know that X_{t+1} is a random variable. $[X_{t+1}|F_t]$ and $[X_{t+1}|X_t]$ are also random variables. Then the second statement in the above definition of the Markov chain essentially means that X_{t+1}, after already knowing F_t (i.e. the entire history), takes on the same set of values and has the same likelihood of distribution among those values as X_{t+1} does after knowing, instead, X_t: the previous value of X (i.e. the previous state of the process). [2]

A Markov chain $X_0, X_1, \ldots$ is considered to be *time-homogeneous* if for all t, $[X_{t+1}|X_t] \sim [X1|X0]$ (i.e. $[X_{t+1}|X_t]$ and $[X1|X0]$ have the same distribution). In other words, X_{t+1}

given X_t takes on the same set of values and has the same likelihood of distribution among those values as X_1 does after knowing X_0. [2]

Throughout this paper, assume all Markov chains discussed are time-homogeneous.

When a filtration is not given for a Markov chain, we assume the Markov chain is with respect to the *natural filtration*. Intuitively, this is a filtration associated with the stochastic process (in this case, the Markov chain), and this filtration keeps a record of the past behavior of the process at each time. Formally, for random variables $X_1, X_2, \ldots$, the natural filtration is the sequence $\sigma(X_1), \sigma(X_1, X_2), \sigma(X_1, X_2, X_3), \ldots$ So, the filtration at time t is $\sigma(X_1, X_2, \ldots, X_t)$, which is the smallest filtration against which $X_1, X_2, \ldots, X_t$ is measurable. [2]

Throughout this paper, assume all Markov chains discussed are with respect to the natural filtration.

3 Representing Markov chains

Markov chains can be represented in a variety of ways. They are commonly represented graphically with a *directed graph*. This is a collection of nodes and edges, in which each edge has a direction from one node to another. Each node represents a state in the Markov chain, and each edge has a probability associated with it that represents the likelihood the source node will transition to the destination node (and the source and destination node can be the same, which means the process remains in the same state). All the probabilities from the edges extending from a node must sum to one (if any edges are omitted, it is assumed that they represent a transition with probability zero).

An example of a simple Markov chain is shown in Figure 1:

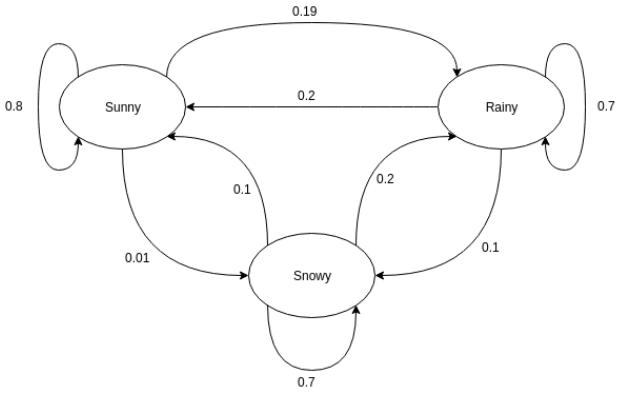

Figure 1: Markov Chain Example [3]

We can equivalently encode the Markov chain more compactly by defining the set of states S and the transition matrix M. In our above example, the set of states is $S = \{$Sunny, Snowy, Rainy$\}$, and the transition matrix M is shown in Figure 2

States	Sunny	Rainy	Snowy
Sunny	0.8	0.19	0.01
Rainy	0.2	0.7	0.1
Snowy	0.1	0.2	0.7

Figure 2: Markov Chain Transition Matrix Example [3]

Note that all rows in the transition matrix must sum to 1.

By starting at any given node, we can generate a sequence of states determined by the probabilities in the transition matrix.

4 Using Markov chains to generate music

Let us now explore how we can create a Markov chain from a given piece of music (the training data) and use it to generate music in the same style as that piece.

In order to generate music, we want the nodes in the Markov chain, or set of states,

to represent "sound objects" – entities that represent a single note or chord and contain information about its duration. Thus, each node will contain data about the note name or collection of notes in a chord (using note names A through F), accidental for each note (#, b, or nothing, indicating natural), octave for each note (0-8), and duration of the sound object (whole note, half note, quarter note, etc). One additional special case will be accounted for: the rest, which will be indicated with "R" and will also have a duration. The set of states will be determined by parsing the piece of music, a process which will be discussed shortly.

We can define our transition matrix by determining the probability that each sound object transitions next to every other sound object. In addition, we would also like to define an initial transition matrix I. This matrix tells us the probability that any given state is chosen, which allows us to pick a start state.

Consider the following Markov chain:

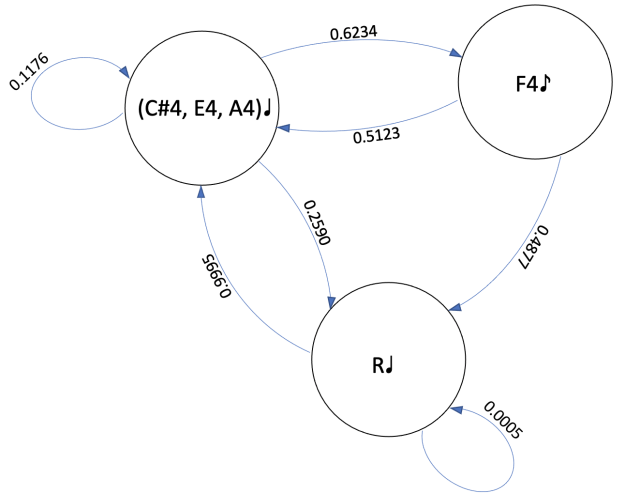

Figure 3: Musical Markov Chain Example

Here, our set of states is $S = \{$(C#4, E4, A4)♩, F4♪, R♩$\}$. In other words, we have a chord consisting of the notes C#4, E4, and A4 that has duration of one quarter note, a single note F4 with a duration of one eighth note, and a rest with a duration of one quarter note.

The transition matrix M is shown in the following figure:

State	(C#4, E4, A4)♩	F4♪	R♩
(C#4, E4, A4)♩	0.1176	0.6234	0.2590
F4♪	0.5123	0.0000	0.4877
R♩	0.9995	0.0000	0.0005

Figure 4: Musical Markov Chain Example

We want to pick a sound object to start with in our generation. We could simply choose the first sound object in the training data, or we could create an *initial transition matrix*. This tells us the probability that each sound object is encountered, which we determine by knowing how many total sound objects there are in the piece (including repetitions) and the number of times each individual sound object appears.

This data is not represented graphically. In the example above, let (C#4, E4, A4)♩ appear twice, F4♪ appear three times, and R♩ appear once in the training data. Then the initial transition matrix I is shown in the following figure:

(C#4, E4, A4)♩	F4♪	R♩
0.3333	0.5000	0.1667

Figure 5: Musical Markov Chain Example

The starting state is chosen using the initial transition matrix.

We now have the tools to generate music from the Markov chain in the same style as the training data.

5 Parsing the training data

Throughout this section, please refer to the file `parse_musicxml.py` in **Appendix** for the full Python code.

The training data (i.e. the input musical piece) will be given in a symbolic form called "musicxml." It is an XML-based file format for encoding musical notation.

We begin by creating a file called `parse_musicxml.py`. We will use Python's Element-

Tree library to parse the musicxml file and the NumPy libraryto build and manipulate matrices. We then create a class called `Parser` that will be used in later in the runner class `generate.py` to parse the input musicxml files. `Parser`'s constructor initializes some important information, such as filename, transition matrix, initial transition matrix, and states (the sound objects we will obtain from the input piece).

We initially extract the data that allows us to build our transition matrix. All the sound objects are extracted sequentially from the musicxml file (whether they are chords or individual notes), including duration, and stored in an ordered dictionary alongside with the number of times each sound object appears in the piece. A sound object is uniquely identified by its note(s), accidental, octave, and duration. At this time, we simultaneously save each sound object in an ordered list (i.e. the set of states) in the order it appears. Note that this ordered list, as it represents a set, does not contain repetitions. This process ensures that the dictionary and the list of states are in the same order, which will allow us to successfully create our transition matrix.

From this dictionary, the transition matrix is created using NumPy. If the list of states has length n, then the transition matrix has dimensions $n \times n$. The transition matrix is symmetric, with both the row and column order corresponding to the order of the state dictionary and the list of states. The transition matrix is built as follows:

1. The matrix is initialized to the known size of $n \times n$. We next build the matrix row by row.

2. Each entry i, j in the matrix is initialized to represent the number of times the i^{th} sound object in the list of states transitions to the j^{th} sound object in the list of states

3. Once all n^2 entries have been initialized, use NumPy to divide all the elements in row of the matrix by its row sum

4. Finally, use NumPy's `cumsum` function to (for each row), replace each entry with the sum of all the previous entries. This means that the first element in each row will

retain the value from the previous step, and all subsequent values will be greater. Note that because of what we did in the previous step, by applying `cumsum` we ensure that the final value in each row is now 1.

Imagine the i^{th} row representing a line, and each i, j entry representing a segment on that line. The i, j entry that corresponds to the longest line segment is the entry that sound object (or state) i has the highest probability of transitioning to. This process to transform the data into the line analogy is also known as *inverse transform sampling*.

We build the initial transition matrix in a similar way. This matrix has dimensions $1 \times n$, since we simply want to know the probability that each sound object is chosen at random. We therefore build the initial transition matrix as follows:

1. The matrix is initialized to the known size of $1 \times n$ (i.e. one row of length n)

2. Initialize the i^{th} entry in the matrix to represent the number of times the i^{th} note in the list of states appears in the piece

3. Once all n entries have been initialized, use NumPy to divide all the elements in the row by the row sum

4. Finally, use NumPy's `cumsum` function to replace each entry in the single row of the initial transition matrix with the sum of all the previous entries. This means that the first element retain the value from the previous step, and all subsequent values will be greater. Note that because of what we did in the previous step, by applying `cumsum` we ensure that the final value (i.e. n^{th} value) is now 1.

The line segment analogy applies here exactly the same way as before.

A final thing to note is that the last note in the piece is given a transition to a quarter rest, and a transition is then added from the quarter rest to the first note in the piece. This ensures that the Markov model contains no *absorbing* states, or states that once you enter, you cannot leave.

6 Generating new music

Throughout this section, please refer to the file `generate.py` in **Appendix** for the full Python code.

Now that we have a working parser that initializes all the elements we need for our Markov chain, we are ready to generate new music in the style of the training data.

We now create file called `generate.py` and import our parser file (`parse_musicxml.py`). We can instantiate the Parser class to create Parser objects (i.e. create Markov chains) for however many songs we want, so long as we have the corresponding musicxml files. In the code attached here, four parsers are created in a list. This allows us to loop through each the list and generate music for each Parser object.

In order to generate music from the Markov chain, we start by using NumPy to generate a random number that is from 0 to 1 inclusive. Now consider the initial transition matrix. Going back to the analogy of the line segment (i.e. inverse transform sampling), think of the generated random number as being a point on the line segment. This can be visualized in Figure 6:

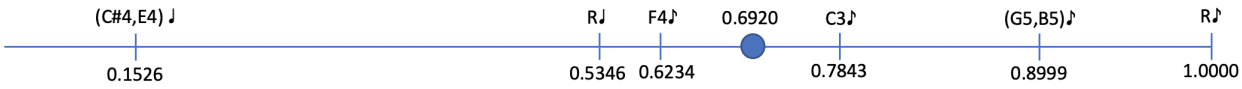

Figure 6: Inverse Transform Sampling Example

We will choose the next highest state (i.e. sound object) compared to the randomly chosen point we generated. In this example, our randomly generated point would give us the sound object C3♪. This allows us to choose the initial state of the Markov model.

We then generate a sequence of states (i.e. our generated music) from the model starting at this initial state. We follow the same method as above for choosing the next state to transition to, except we now use the transition matrix instead of the initial transition matrix. The length of the sequence is determined by the user's input. In the code in **Appendix**, the length chosen is 100 notes.

After generating the sequence of sound objects, the sequence is written out to a MIDI file, which is then loaded into the symbolic music software MuseScore for viewing and playing.

7 Results of the music generation

In this paper, I generate music trained on a piece I wrote. The piece is initially for flute and piano. I separated it into the flute part and the piano part and took a short excerpt from each in order to demonstrate a monophonic example (i.e. the solo flute part), and an example with harmony/chords (i.e. the piano part). The results of the music generated from these parts using their respective Markov chains are as follows.

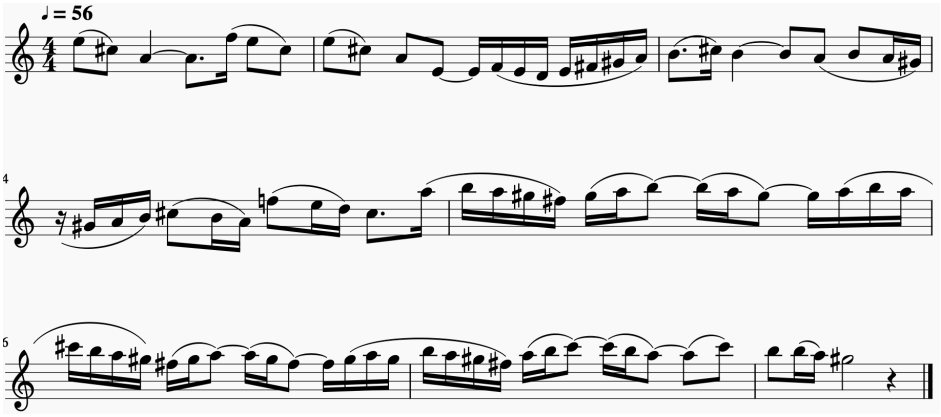

Figure 7: Original Flute Part

Figure 7 contains the original flute part (i.e. the training data)

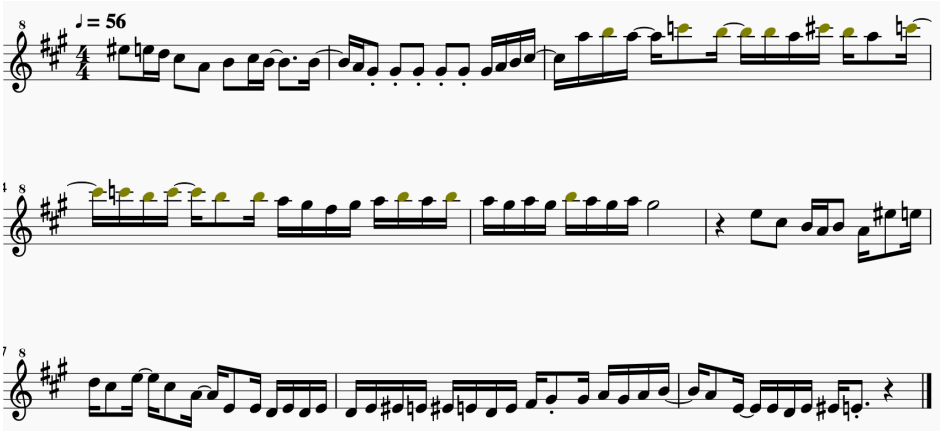

Figure 8: Generated Flute Part

Figure 8 contains the generated flute part

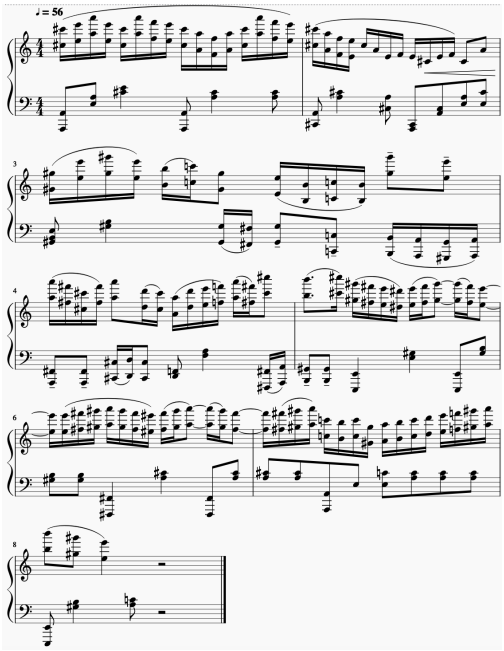

Figure 9: Original Piano Part

Figure 9 contains the original piano part (i.e. the training data)

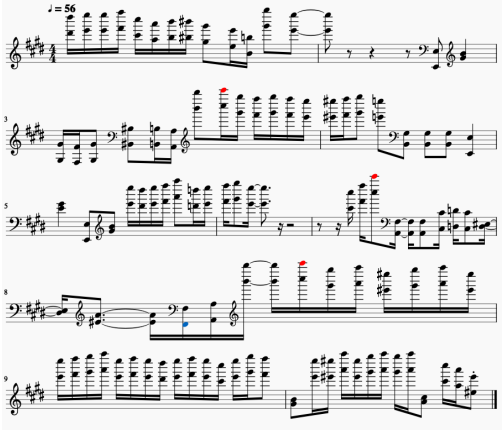

Figure 10: Generated Piano Part

Figure 10 contains the generated piano part. (Note that because of the way the MIDI file was generated, the generated piano part gets compressed into a single staff. This is not a result of the Markov chain, it is simply due to the MIDI formatting).

8 Conclusions

Using a simple Markov chain, music can be successfully generated in the style of the training piece. Rhythm, octave, pitch, and accidentals were accounted for. However, there are limitations to the current setup as well as many other avenues to be explored. Currently, the parser does not handle pieces with multiple voices within a single part, or a piece with multiple instruments. In addition, dynamics are not taken into account. Other statistical models, such as the Hidden Markov Model, may provide for interesting avenues of exploration. With the Hidden Markov Model, instead of generating a sequence of states, each state omits an *observable*, and the states themselves are hidden. The idea here is to use techniques such as dynamic programming to backtrack from the generated observables in order to determine the optimal sequence of hidden states that generated these observables. In our case, for instance, we could generate observable notes/chords, and use dynamic programming to uncover the optimal sequence of rhythms, or perhaps dynamics (whatever we choose to be the hidden states) based on the observables. Overall, it becomes evident that

statistical modeling for computer music generation has a strong potential.

References

[1] David High and William Morris. Example of filtration in probability theory, Aug 2019.

[2] Mark Huber. Lecture notes in stochastic processes.

[3] Alessandro Molina. Markov chains with python, Nov 2018.

9 Appendix

9.1 parse_musicxml.py

```
1  import xml.etree.ElementTree as ET
2  import collections
3  import numpy as np
4
5  class Parser:
6      def __init__(self, filename):
7          self.filename = filename
8          self.root = ET.parse(filename).getroot()
9
10         self.initial_transition_dict = collections.OrderedDict
               ()
11         self.normalized_initial_transition_matrix = None
12
13         self.transition_probability_dict = collections.
               OrderedDict()
14         self.normalized_transition_probability_matrix = None
15
16         self.states = []
17
18         self.smallest_note_value = None
19         self.tempo = None
20
21         self.order_of_sharps = ['F', 'C', 'G', 'D', 'A', 'E', '
               B']
```

```
22          self.key_sig_dict = {'C':'', 'D':'', 'E':'', 'F':'', 'G
               ':'', 'A':'', 'B':''}
23          self.parse()
24
25      def parse(self):
26          prev_note = None # the prev note (it may be part of a
               chord). but this variable itself NEVER stores a
               chord
27          sound_object_to_insert = None # either note or chord
28          prev_sound_object = None # either note or chord
29          in_chord = False
30          note = None
31          chord = None
32          prev_duration = None
33          first_sound_object = None
34
35          direction_blocks = self.root.find('part').find('measure
               ').findall('direction')
36          for direction_block in direction_blocks:
37              if self.tempo is None and direction_block.find('
                   sound') is not None and 'tempo' in
                   direction_block.find('sound').attrib:
38                  self.tempo = int(direction_block.find('sound').
                       attrib['tempo'])
39          self.instrument = self.root.find('part-list').find('
               score-part').find('part-name').text
40          if self.instrument == 'Piano':
41              self.instrument = 'Acoustic Grand Piano'
42          self.name = self.root.find('credit').find('credit-words
               ').text
43
44          for i, part in enumerate(self.root.findall('part')):
45              for j, measure in enumerate(part.findall('measure')
                   ):
46                  measure_accidentals = {}
47                  self.set_key_sig_from_measure(measure)
48
49                  for k, note_info in enumerate(measure.findall('
                       note')):
50                      duration = note_info.find('type').text
51                      note = None
52
53                      if note_info.find('pitch') is not None:
54                          value = note_info.find('pitch').find('
                               step').text if note_info.find('pitch
```

```
                ').find('step') is not None else ''
55              octave = note_info.find('pitch').find('
                    octave').text if note_info.find('
                    pitch').find('octave') is not None
                    else ''
56
57              accidental_info = note_info.find('
                    accidental').text if note_info.find(
                    'accidental') is not None else None
58              accidental = '' if accidental_info is
                    None else ('#' if accidental_info ==
                     'sharp' else 'b' )
59              note_for_accidental = value+octave
60              if accidental_info == 'sharp':
61                  accidental = '#'
62                  measure_accidentals[
                        note_for_accidental] = '#'
63              elif accidental_info == 'flat':
64                  accidental = 'b'
65                  measure_accidentals[
                        note_for_accidental] = 'b'
66              elif accidental_info == 'natural':
67                  accidental = ''
68                  measure_accidentals[
                        note_for_accidental] = 'n'
69              elif accidental_info is None and
                    note_for_accidental in
                    measure_accidentals:
70                  accidental = '' if
                        measure_accidentals[
                        note_for_accidental] == 'n' else
                         measure_accidentals[
                        note_for_accidental]
71              else:
72                  accidental = self.key_sig_dict[
                        value]
73
74              note = value + accidental + octave
75          elif note_info.find('chord') is None:
76              # means that note_info.find('rest') is
                    not None ----> so we are in a rest
77              note = 'R'
78
79          if note is not None:
```

```
80                          is_last_iteration = i == len(self.root.
                               findall('part')) - 1 and j == len(
                               part.findall('measure')) - 1 and k
                               == len(measure.findall('note')) - 1
81
82                          if note_info.find('chord') is not None:
83                              # currently, the duration is just
                                   going  to be that of the last
                                   note in the chord (can go back
                                   to change this later...)
84                              if in_chord:
85                                  chord.append(note)
86                              else:
87                                  chord = [prev_note, note]
88                                  in_chord = True
89                          else:
90                              prev_sound_object =
                                   sound_object_to_insert
91                              if in_chord and note_info.find('
                                   chord') is None:
92                                  in_chord = False
93                                  sound_object_to_insert = (tuple
                                       (sorted(chord)),
                                       prev_duration)
94                              else:
95                                  sound_object_to_insert = (
                                       prev_note, prev_duration)
96
97                              self.handle_insertion(
                                   prev_sound_object,
                                   sound_object_to_insert)
98                              if first_sound_object is None and
                                   prev_sound_object is not None
                                   and prev_sound_object[0] is not
                                   None:
99                                  first_sound_object =
                                       prev_sound_object
100                             if is_last_iteration:
101                                 # note that we're NOT in a
                                       chord (i.e. last sound
                                       object is NOT a chord)
102                                 self.handle_insertion(
                                       sound_object_to_insert, (
                                       note, duration))
103
```

```
104                     prev_note = note
105                     prev_duration = duration
106
107         if in_chord:
108             # handle the case where the last sound object was a
                    chord
109             final_chord = (tuple(sorted(chord)), prev_duration)
110             self.handle_insertion(sound_object_to_insert, (
                    final_chord, prev_duration))
111         else:
112             # final sound object was NOT a chord, it was a note
113             # set the last note/chord to transition to a
                    quarter rest, rather than nothing
114             # then add a transition from the rest back to the
                    first sound object
115             # this ensure that everything has a transition
                    defined for it
116             self.handle_insertion((note, duration), ('R', "
                    quarter"))
117             self.handle_insertion(('R', "quarter"),
                    first_sound_object)
118
119         self.build_matrices()
120
121     def set_key_sig_from_measure(self, measure_object):
122         key_sig_value = measure_object.find('attributes')
123         if key_sig_value is not None:
124             key_sig_value = key_sig_value.find('key')
125             if key_sig_value is not None:
126                 key_sig_value = int(key_sig_value.find('fifths'
                        ).text)
127
128                 for note in self.key_sig_dict:
129                     self.key_sig_dict[note] = ''
130                 if key_sig_value == 0:
131                     return
132                 elif key_sig_value < 0:
133                     for i in range(len(self.order_of_sharps) -
                            1, len(self.order_of_sharps)-
                            key_sig_value, -1):
134                         self.key_sig_dict[self.order_of_sharps[
                                i]] = 'b'
135                 else:
136                     for i in range(key_sig_value, len(self.
                            order_of_sharps)):
```

```
137                         self.key_sig_dict[self.order_of_sharps[
                               i]] = '#'
138
139     def build_matrices(self):
140         self.build_normalized_transition_probability_matrix()
141         self.build_normalized_initial_transition_matrix()
142
143     def build_normalized_initial_transition_matrix(self):
144         self.normalized_initial_transition_matrix = np.array(
               list(init_prob for init_prob in self.
               initial_transition_dict.values()))
145         # convert to probabilities
146         self.normalized_initial_transition_matrix = self.
               normalized_initial_transition_matrix/self.
               normalized_initial_transition_matrix.sum(keepdims=
               True)
147         # multinomial dist
148         self.normalized_initial_transition_matrix = np.cumsum(
               self.normalized_initial_transition_matrix)
149
150     def build_normalized_transition_probability_matrix(self):
151         # initialize matrix to known size
152         list_dimension = len(self.states)
153         self.normalized_transition_probability_matrix = np.
               zeros((list_dimension,list_dimension), dtype=float)
154
155         for i, sound_object in enumerate(self.states):
156             for j, transition_sound_object in enumerate(self.
                   states):
157                 if transition_sound_object in self.
                       transition_probability_dict[sound_object]:
158                     self.
                           normalized_transition_probability_matrix
                           [i][j] = self.
                           transition_probability_dict[sound_object
                           ][transition_sound_object]
159         self.normalized_transition_probability_matrix = self.
               normalized_transition_probability_matrix/self.
               normalized_transition_probability_matrix.sum(axis=1,
               keepdims=True)
160         self.normalized_transition_probability_matrix = np.
               cumsum(self.normalized_transition_probability_matrix
               ,axis=1)
161
```

```
162     def handle_insertion(self, prev_sound_object,
            sound_object_to_insert):
163         if sound_object_to_insert is not None and
                sound_object_to_insert[0] is not None:
164             if prev_sound_object is not None and
                    prev_sound_object[0] is not None:
165                 self.insert(self.transition_probability_dict,
                        prev_sound_object, sound_object_to_insert)
166             if sound_object_to_insert not in self.states:
167                 self.states.append(sound_object_to_insert)
168
169             if sound_object_to_insert in self.
                    initial_transition_dict:
170                 self.initial_transition_dict[
                        sound_object_to_insert] = self.
                        initial_transition_dict[
                        sound_object_to_insert] + 1
171             else:
172                 self.initial_transition_dict[
                        sound_object_to_insert] = 1
173
174     def insert(self, dict, value1, value2):
175         if value1 in dict:
176             if value2 in dict[value1]:
177                 dict[value1][value2] = dict[value1][value2] + 1
178             else:
179                 dict[value1][value2] = 1
180         else:
181             dict[value1] = {}
182             dict[value1][value2] = 1
183
184     def print_dict(self, dict):
185         for key in dict:
186             print(key, ":", dict[key])
187
188     def rhythm_to_float(self, duration):
189         switcher = {
190             "whole": 4,
191             "half": 2,
192             "quarter": 1,
193             "eighth": 1/2,
194             "16th": 1/4,
195             "32nd": 1/8,
196             "64th": 1/16,
197             "128th": 1/32
```

```
198         }
199         return switcher.get(duration, None)
```

9.2 generate.py

```
1  import parse_musicxml
2  import random
3  import numpy as np
4  import midi_numbers
5  from midiutil import MIDIFile
6  import sys
7  import re
8
9  # I did not write this function. Credit: Akavall on
       StackOverflow
10 # https://stackoverflow.com/questions/17118350/how-to-find-
       nearest-value-that-is-greater-in-numpy-array
11 def find_nearest_above(my_array, target):
12     diff = my_array - target
13     mask = np.ma.less(diff, 0)
14     # We need to mask the negative differences and zero
15     # since we are looking for values above
16     if np.all(mask):
17         return None # returns None if target is greater than
               any value
18     masked_diff = np.ma.masked_array(diff, mask)
19     return masked_diff.argmin()
20
21 def generate(seq_len, parser):
22     sequence = [None] * seq_len
23
24     # comment in for same start note as training data
25     note_prob = random.uniform(0, 1)
26     rhythm_prob = random.uniform(0, 1)
27     note_index = find_nearest_above(parser.
           normalized_initial_transition_matrix, note_prob)
28     check_null_index(note_index, "ERROR getting note index in
           initial transition matrix")
29     curr_index = 0
30
31     # comment in for seed
32     # sequence[0] = parser.states[0]
33     # note_index = 0
```

```
34      # curr_index = 1
35
36      while (curr_index < seq_len):
37          note_prob = random.uniform(0, 1)
38          rhythm_prob = random.uniform(0, 1)
39
40          note_index = find_nearest_above(parser.
                normalized_transition_probability_matrix[note_index
                ], note_prob)
41          check_null_index(note_index, "ERROR getting note index
                in probability transition matrix")
42
43          sequence[curr_index] = parser.states[note_index]
44          curr_index += 1
45
46      return sequence
47
48  def check_null_index(index, error_message):
49      if(index == None):
50          print(error_message)
51          sys.exit(1)
52
53  def get_note_offset_midi_val(note):
54      switcher = {
55          "C": 0,
56          "C#": 1,
57          "Db": 1,
58          "D": 2,
59          "D#": 3,
60          "Eb": 3,
61          "E": 4,
62          "Fb": 4,
63          "E#": 5,
64          "F": 5,
65          "F#": 6,
66          "Gb": 6,
67          "G": 7,
68          "G#": 8,
69          "Ab": 8,
70          "A": 9,
71          "A#": 10,
72          "Bb": 10,
73          "B": 11,
74          "Cb": 11
75      }
```

```
76      return switcher.get(note, 0)
77
78  def get_pitch(note):
79      octave_info = re.findall('\d+', note)
80      if len(octave_info) > 0:
81          octave = int(octave_info[0])
82          note = ''.join([i for i in note if not i.isdigit()])
83          base_octave_val = 12*octave + 24
84          note_val = base_octave_val + get_note_offset_midi_val(
                note)
85          return note_val
86      return None # this is a rest
87
88  if __name__ == "__main__":
89      parsers = [parse_musicxml.Parser('Cantabile_flute_excerpt.
            musicxml'), parse_musicxml.Parser('
            Cantabile_piano_excerpt.musicxml')]
90
91      for parser in parsers:
92          sequence = generate(100, parser)
93          track    = 0
94          channel  = 0
95          time     = 0.0    # In beats
96          duration = 1.0    # In beats
97          tempo    = parser.tempo if parser.tempo is not None
                else 80  # In BPM
98          volume   = 100  # 0-127, as per the MIDI standard
99
100         output_midi = MIDIFile(1)  # One track, defaults to
                format 1 (tempo track is created automatically)
101         output_midi.addTempo(track, time, tempo)
102         output_midi.addProgramChange(track, channel, time,
                midi_numbers.instrument_to_program(parser.instrument
                ))
103
104         time = 0.0
105         for sound_obj in sequence:
106             duration = float(parser.rhythm_to_float(sound_obj
                    [1]))
107             sound_info = sound_obj[0]
108             if type(sound_info) is str:
109                 pitch = get_pitch(sound_info)
110                 if pitch is not None: # i.e. if this is not a
                        rest
```

```
111                     output_midi.addNote(track, channel, pitch,
                            time, duration, volume)
112             else: #  type(sound_info) is tuple
113                 for note in sound_info:
114                     pitch = get_pitch(note)
115                     output_midi.addNote(track, channel, pitch,
                            time, duration, volume)
116             time += duration
117         with open(parser.filename + ".mid", "wb") as
                output_file:
118             output_midi.writeFile(output_file)
```