Automating Music Composition and Melody Generation

Armen Aghajanyan, Souren Aghajanyan

Email: armen.ag@live.com

Many believe that the epitome of human creativity is the ability to create art irregardless of the art form itself. One of the more popular art forms, music, has been viewed as an intellectual exercise and composers such as Bach and Chopin are viewed as geniuses in our modern world. The inherent question to ask was, could software even begin to mimic the creativity of these composers and their compositions. And that is the question this paper will introduce and hopefully answer.

Keywords: Artificial Intelligence; Machine Learning; n-grams; Markov Chains; Graphs; Neural Networks; Clustering; Mathematical Optimization, Genetic Algorithms

Received 00 Month 2009; revised 00 Month 2009

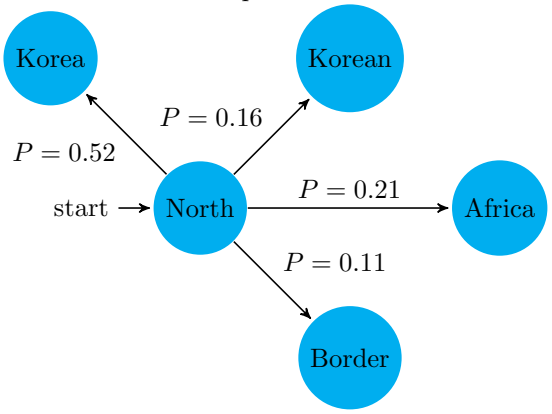
1. INTRODUCTION

To start off I would like to explain the format of the rest of this paper. We will split the task of music composition and melody generation into two sub tasks. First, we will create a statistical model of the training data, which is a collection of different songs in the popular midi format. Second, creating operators that would act upon our model in a way to creating a variety of aesthetically pleasing melodies. We try to keep the math to a complete minimum, and will only be used for formal definitions. This is a challenging problem, and all our challenges will be presented with their respective solutions in a rather detailed matter. An understanding of n-grams, k-means, and the concept of genetic searches is needed to understand this paper, while knowledge of Artificial Neural Networks is also helpful.

2. GENERATING OUR STATISTICAL MODEL

2.1. Generating Basic Model

From the beginning of this project it became apparent that the generation of music paralleled to the task of auto generation of research papers (AGP), which has been researched extensively in the field of Natural Language Processing (NLP). [**?**] The first step was creating a generic framework that would support the bare basic of AGP. We started off by building a a generic representation of an n-gram, and the ability to build an ordered collection of any sized n-grams in the exact order of which they appeared. From here surfaced two types of n-gram collections; homogeneous and heterogeneous. The difference is quite simple, a homogeneous collection of n-grams would contain all the n-grams of this same size (n), while heterogeneous would support n-graming ranging from n_i *to* n_f. Next we needed to implement a data structure that would allow us to flow from one n-gram to another, and which would preserve the probabilities of transitions. The obvious choice would be an implementation of a Markov Chain through a weighted directed graph. [**?**] Below is an example of a homogeneous n-gram Markov chain with a $n = 1$ that was produced from our framework.



Keep in mind that we have a strict constraint that the

$$\sum_{P} p = 1 \tag{1}$$

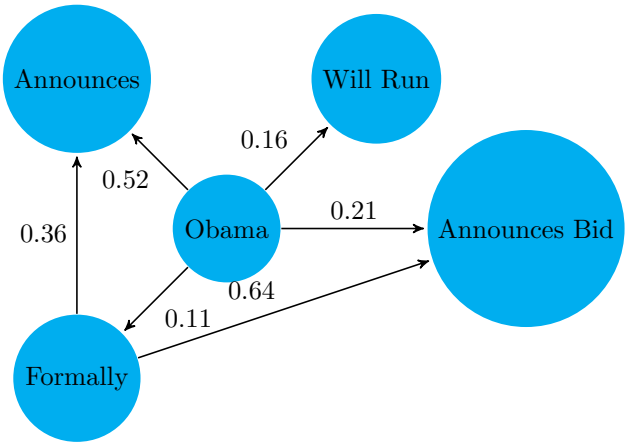
as is enforced by the original definition of a Markov Chain. The construction of a homogeneous n-gram chain is very simple. We begin by first constructing a distribution of n-gram appearances. For example if the n-gram $\{North\}$ appeared a total amount of a times in our n-gram collection, we would store a key-value where the key would be the $\{North\}$ and the value would be a. Therefore finding the weight or the probability of a

single transition would simply be.

$$\frac{F(node)}{\sum_{i=0}^{n} F(vertex_i)} \tag{2}$$

Where the function $F(node)$ represents our count n-gram distrubution function.

From here we could do random walks and generate semi-coherent (and sometimes extremely funny) sentences of variable length. We continued on the theme of building n-gram Markov chains and extended our framework to allow building not only homogeneous Markov chains but also heterogeneous Markov chains. Below is a simple visualization of an heterogeneous n-gram Markov chain generated by our framework with a range $[1 \rightarrow 3)$. [1]



Although the two types of Markov chains seems very similar, heterogeneous Markov chains have a major advantage, they allow movement in between their sub constructs. A construct is a homogeneous Markov chain inside the heterogeneous Markov chain. Therefore for a heterogeneous with a size range $(n_i, n_f]$ we will inherently have $n_f - n_i$ constructs. Let me explain more in depth what I mean movement between sub constructs. For example, let us say that during our walk we end up on a node *Announces Bid*, but this node has no vertices. If this was a typical homogeneous Markov chain, we would be done, but with a heterogeneous Markov chain we can take our node, and lower it one construct down, so we were at the node *Announces* and continue with our sub node so to speak. This type of manipulation becomes increasingly more helpful as we continue with building our statistical model.

We finished the bare building of our informational model and now move on to creating a more fluid and intelligent transitioning system. I should note that we are finished with the AGP analogy and now will be continuing in a more musical fashion .

[1]The generic nature of our code allowed the random walk to extend to heterogeneous chains as well.

2.2. Adding Concept of Weight Assigners

So far, the type of walking we have used (random walkers) have all maintained the Markov property [**?**]. [2] We will know stray off from this property and because of this I will transition from referring to Markov chains to referring to graphs whose vertices's must add up to 1 as to keep my mathematical accuracy as high as possible.

Now I will introduce the concept of weight assigners. A weight assigner is a simply a function that takes in a binary tuple of collection of n-grams. The first item in the tuple will represent the n-gram chain that is being built, while the second item represents the possible states to transition to. And return a probability distribution. Mathematically it will appear like so.

$$F = W(\overrightarrow{n-gram}, \overrightarrow{n-gram}) \tag{3}$$

Where:

$$[\sum_{x=W(a)} F(x_{node})] = 1 \tag{4}$$

Or if we are extending this probability distribution to a continuous distribution.

$$[\int_{x=W(a)} F(x_{node})] = 1 \tag{5}$$

The weight assigner that I found to be the most effective was a size weight assigner (SWA). What the SWA would try to accomplish, was to mimic the true distribution of the n-gram size in some window of the chain that we were building. Mathematicly speaking.

$$F_{SWA} = TrueDistrubutionSize() -$$

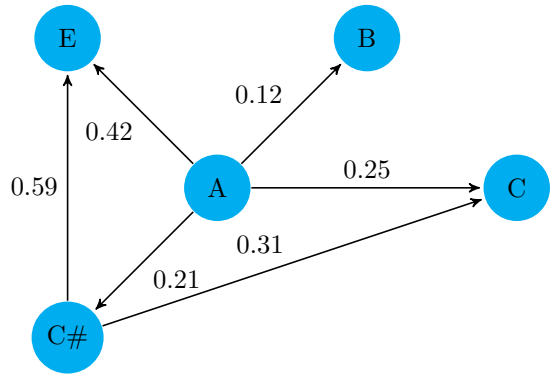
$$\mathrm{W}(\overrightarrow{\mathrm{ngram}} \rightarrow \overrightarrow{\|ngram\|}, \overrightarrow{\mathrm{ngram}} \rightarrow \overrightarrow{\|ngram\|}) \tag{6}$$

We have now finished building our statistical model and are know ready to create the necessary operators to build over it.

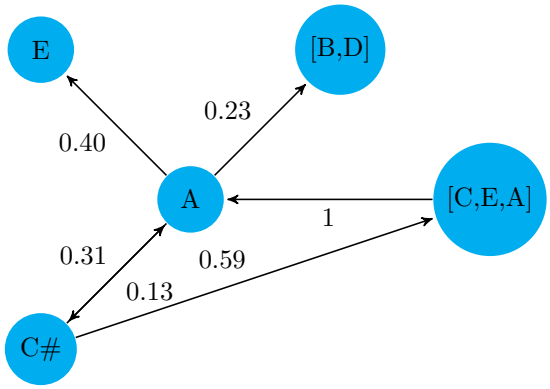
2.3. Quick Explanation of MIDI Information Extraction

We used an open source midi library for the basic extraction of ChannelMessages and other type of MIDI events.[**?**] The type of information we would extract involved a substantial amount of testing. We started off by extracting simply the notes, paying no attention to the time or the octave that it was on.

[2]The future nodes of the process depend only upon the present state



This showed no promising results even when paired when multiple weight assigners. We continued with pairing notes with their respective octave, and this showed a bit better results. We ended on naively extracting chords from the MIDI file, where the chords were a collection of notes with their respective octave. Right from the get-go this showed extremely promising results, giving us melodies that were almost aesthetically pleasing. Below is a small representation of a heterogeneous graph of chords.



3. BUILDING INFERENCES FROM OUR STATISTICAL MODEL

3.1. Initial Experience With Clustering

After experimenting around with the statistical model, creating and playing different n-gram chains it became apparent that certain type of melodies were being repeated with only minor variations. From here the technique of clustering seemed useful, in theory, if we defined a correct distance metric, after clustering, the centers of clusters would represent our melodies. We decided to implement two distance metrics, Squared Euclidean distance and Levenshtein distance. Below is how we calculated the Squared Euclidean distance.

$$\textit{Euclidean Squared} = d^2(\vec{p},\vec{q}) = \sum_{i=0}^{n}(p_i - q_i)^2 \quad (7)$$

Below is a recursive definition of our implementation of the Levenshtein algorithm.

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if}\min(i,j) = 0: \\ \min\begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} \end{cases} \quad (8)$$

Both of our distance metrics, only took into account the note-octave pair, no concept of time was introduced to either of our distance metrics.

For our clustering algorithm, we chose the widely used k-means algorithm. [**?**] In short, the k-means algorithm tries to minimize the within-cluster sum of squares. Mathematically speaking

$$[\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{a}_j \in S_i} \mathsf{distance}(a_j, \boldsymbol{\mu}_i)] \quad (9)$$

where μ_i represents a center of a cluster and a_j represents the vectors that belong to their respective cluster.

When building our heterogeneous n-gram graph, we used a total of 11 songs, 9 consisting of simple beats by the theme of the rap artist Eminem, and 2 consisting of heavy jazz improvisation in the styles of Louis Armstrong and Ella Fitzgerald. Our graph ended up consisting of exactly 1025 nodes, where each node had an average of 23 connections. For our clustering we ran through each node, and selecting 10 chains, with the same length of 25. [3] The clustering took a while because of the computational complexity of the k-means algorithm $\rightarrow$ $O(n^{dk+1}log(n))$ when d the dimension and n number of vectors stay constant, which they do in our scenario. Never the less, after the computation finished we did not achieve the extraction of main melodies, like we hypothesized before, regardless of the distance metric we used. Two possible causes explained this phenomenon, either we had a lacking distance metric or not enough sample points. Computationally it would be hard to include anymore sample points. We then decided to run the same experiment but now instead of running through every node in the graph and selecting n chains, we picked a root node, and completed a numerous amount of walks. After running these chains through the k-means clustering algorithm, we achieved a satisfactory result with respect to our earlier hypothesis. Our clustering now contained 3 main centers, whose melody's substantially varied and were pseudo pleasing to listen to.

[3] All of our selection used random walking paired with weight assigners.

3.2. Defining the Necessary Genetic Operators

From the start it is obvious from the way we framed this problem along the lines of an optimization problem. One of the more popular and more effective search methods is the genetic search algorithm. This is the search algorithm that we will use in our framework. The four necessary operators we need to define are

1. Random
2. Mutate
3. Cross
4. Fitness

The random operator simply returns a random object with respect to our search type, which in our case is a chain of n-grams of chords. The mutate operator takes in an entity, and randomly mutates it and returns the mutated entity. The cross operator inputs two entities with the same type, and returns an object which is the superposition of both its inputs. And lastly the fitness operator takes in an object and returns a real number which represents the fitness of that entity. Below I will go into further detail about the specific implementation of each individual function.

The implementation of the random operator for our n-gram chain is rather quite simple, and have in fact already implemented. We simply walk over the graph in a random walker. After a few experimental trials we concluded that the presence of a weight assigner did improve our results.

Before we start diving into the implementation of the mutate operator, let us define a few constants. $p_r \| \{p_r \in R \wedge 0 <= p_r <= 1\}$ *and* $p_m \| \{p_m \in R \wedge 0 <= p_m <= 1\}$. The constant p_r gives us the probability that we will replace a n-gram with a completely random n-gram, while p_m gives us the probability that we will replace the n-gram with one of its connections in our graph. The mutate functions purpose is not to generate complete randomness, but to add incremental randomness. Using this model it allowed us to add discrete amount of randomness to our mutate functions with respect to its purpose. After a numerous amount of experimental trials we concluded that $p_r = 0.05$ and $p_w = 0.1$ gave us the best results.

For our cross function we implemented two separate algorithms. The first algorithm had no concept of merging data but instead used random discrete selection. The second algorithm merged the two chains via the data inside each n-gram in the chain. The implementation of the first algorithm is quite simple, you simply, move along both chains and randomly select either the left chains n-gram or the right chains n-gram and add the respective n-gram into the new chain. For the second algorithm the implementation is quite similar to the first, but now instead of selecting the left or right chains n-gram we define a function merge which takes in 2 n-grams and returns the n-gram representing the superposition of both n-grams. The implementation of the merge function goes like so.

Merge:
$ngram\ temp = empty$
for $i := 0$ to n step 1 do
    $\text{temp}_i \leftarrow (\text{random}() < 0.5)\ ?\ left_i : right_i;$

Last but not least is our fitness function. The fitness function is hard to define as their is no ground truth we can utilize for defining what is good or bad in sense of the music. In the next subsection I will introduce our approach to implementing a general solution to a fitness function while lacking any ground truth.

Randomness is the key to adaptation.

3.3. Defining a Fitness Function Without Ground Truth

Creating a correct fitness function that is general enough to be extended with a lack of ground truth is a hard task to accomplish because of a pair of reasons.

- The concept of good and bad varies wildly due to lack of ground truth.
- Creating a continuous distribution of fitness having no labeled data.

Let me go into detail into both of these reasons. From a musical point of view what might seem as a perfect n-gram chain in rap lets say, will be bad in rock. Without a ground truth we are left with a free floating structure with absolutely no common area for generic rule extraction. This directly connects with our second reason. Not only do we not have labeled data, but the concept of using labeled data now seems pointless with respect to our prior statement. We needed a extremely versatile continuous interpolation algorithm that could dynamically learn to assign values to what is good and bad from a minimum and noisy training data set. The obvious choice was to train an Artificial Neural Network (ANN). [**?**]

The start would be dynamically defining a data set which we would feed to our ANN. The solution involved defining three types of chains; $Good = 1$, $Okay = ?$ and $Bad = 0$. The calculations involved in each one we kept to a very minimal. The window size we will refer to as n_w, while the training size we will refer to as n_t. For the *Good* collection, we would take a input of songs and retrieve windows of size n_w ranging from $[0, n_t - n_w)$, for the *Okay* collection we allowed a choice of using the same algorithm as in *Good* or just compiling a random amount of random walks across our statistical model.

The point of allowing the *Okay* to implement the *Good* algorithm was to allow for user melody customization. For example a user might decide that they prefer rock over pop, therefore we allow for labeling rock songs as *Good* while labeling pop as *Okay*. The *Bad* algorithm just aggregated together completely random n-grams from our graph.

A search algorithm has limits which directly correspond to the quality of its reward function.

The ANN had dimensions like this ? $\rightarrow$ 23 $\rightarrow$ 1.[4] The question mark at the beginning is because of the fact that the window size is dynamically chosen and therefore is not a static property. The number 23 came purely from testing, this number gave us the least amount of error and the quickest convergence. Our initial training used the Resilient Back-propagation algorithm, with the Sigmoid function ($\frac{1}{1+e^{-x}}$) as our activation function. [**?**] Although it showed promising results with some data-set's such as reducing the total network error to 2.14 with a total possible error of 6178. What this means is that the network out of 6178 melodies mislabeled only about 2 melodies. Although these results seem amazing and virtually perfect, that is not the case is about 80% of the data-set's we tested. The ANN seemed to fall into local minima's rather quick when producing the error of around 160 and sometimes around 60 which in our case is to high of an error to use as a fitness function. To counteract the convergence into local minima, we increased the *Learning Rate* of the back-propagation algorithm. After a few tries we zoned in one such a rate that gave us both low error results as well as a quick convergence.

3.4. Running the Genetic Search Defined with a Neural Network

The training we initiated was based on the same 11 songs that I talked about before in this paper. After about two hours of training we achieved an error rate just south of 4. For the *Good* collection we gave the 11 songs as input, for the *Okay* collection we let the random walk algorithm take over. We allowed our genetic algorithm to run exactly 100 epochs, before we stopped it and started analyzing the melody's we generated. As we hypothesised, the quality of the songs exponentially rose compared to the clustering technique we used prior. But to our surprise, the top individuals varied in virtually no way. It had seemed as though the genetic search got stuck in a deep local minimum. Due to the random nature of the genetic search we believed that running the algorithm again we give us varying results, but to our dissatisfaction it seemed that no matter the amount of times we ran the search the same top individuals appeared repeatedly. It became apparent that we would again need to introduce an algorithm to filter out in a shallow matter the non varying melodies.

3.5. Using a Hybrid K-Means to Filter in a Shallow Matter

Before I continue, let me explain in detail what I mean by "shallow". The aim of the majority of clustering is to find the smallest set of clusters which can successfully describe the given data. For shallow clustering, the intention is to be computationally quick, and to, of course, find clusters but not in the perfection mindset clustering algorithms are usually in. So a shallow clustering algorithm will, for example, take in 10000 vectors, and cluster them into 500 clusters. This is exactly what we need for filtering out each epochs population in our genetic population. Using standard clustering techniques was out of the question since we use the centers of the clusters as the surviving population, the ratio of survivors to random individuals would be to high for the genetic search to work as effectively as possible. We are still going to use the k-means algorithm, but we are going to morph it just a bit. Below are the three things we need to do.

- Start off with a relatively big number of clusters.
- Lower the amount of epochs we run.
- Add a source of randomness to the distance function.

The first two bullets points are rather obvious. In respect to the third bulletpoint, the reason I want to add a source of randomness is to steer away from the perfection of the clustering algorithm by adding noise into the distance function. Below I'm going to mathematically introduce a source of noise into our definition of the k-means algorithm. For the type of noise, we chose to use the standard Gaussian noise model. We used the popular Box-Muller Transformation to generate normally distributed random numbers. [**?**] Let us say our mean will always be $\mu = 1$, and we will assign the the variable σ to our standard deviation. We will also assign the function $BMT(\mu, \sigma)$ to the Box-Muller Transform, given a mean and a standard deviation we will get a uniformly distributed random number. Now that we have defined the necessary variables and functions let us begin. We will define a noisy distance function $distance(\overrightarrow{p_i}, \overrightarrow{p_q})$ as

$$\mathsf{noisydistance}(\overrightarrow{p_i}, \overrightarrow{p_q}) = \mathsf{distance}(\overrightarrow{p_i}, \overrightarrow{p_q})^{BMT(\mu,\sigma)} \quad (10)$$

Now on average to retain the distance function, as I stated previously we will fix our mean value μ to 1. We can now rewrite our equation as.

$$\mathsf{noisydistance}(\overrightarrow{p_i}, \overrightarrow{p_q}) = \mathsf{distance}(\overrightarrow{p_i}, \overrightarrow{p_q})^{BMT(1,\sigma)} \quad (11)$$

[4] Have exactly one hidden layer.

Now let us apply this to our definition of our k-means algorithm.

$$[\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{a}_j \in S_i} \mathsf{distance}(a_j, \boldsymbol{\mu}_i)^{BMT(1,\sigma)}] \qquad (12)$$

We now have successfully introduced a source of randomness to our k-means algorithm and are now ready to use this as an intermediate algorithm in our genetic search.

3.6. Introducing the Hybrid K-Means into our Genetic Search

All the operators of the genetic search stay as is, all we do is add an intermediate step in our algorithm. The intermediate step is placed after every epoch of the genetic search. Our hybrid k-means would take in the top 10% of the individuals and then randomly select from the bottom 90%. We did this for computational speed up, because even though we are clustering via a shallow method, therefore with a low epoch, each epoch takes some amount of time, that grows rather quickly as we introduce more and more samples. After running our new algorithm and playing the melodies, we in fact saw that the top individuals were very clean representations of the melodies we heard before. Not only that but we saw a substantial increase in the variety. We continued our experimentation, by completely changing our input songs, the melodies changed, and fit under the style of the songs we picked. [5]

4. CONCLUSION

[5]In respect to the 11 I've been referring though, we went from 3 melodies to a subjective 7.