# Music Generation with Language Models

**Tyler Bateman**
batemat2@wwu.edu

## Abstract

This paper explores the application of Natural Language Processing to music, specifically the task of music generation. I propose the use of trained language models to accomplish this task. These experiments make use of three different language model architectures: N-gram models trained with various N-gram sizes, Markov chain generators with various state sizes, and a Recurrent Neural Network with various temperatures.

## 1 Introduction

The work described in this paper is based upon the premise that the language of music has many of the same features as natural language. It has even been argued that music is itself a natural language, some going so far as to propose methods of translating between the two (Schenkman, 1979). Given this premise, it follows that applications of Natural Language Processing would also apply to music. Indeed, there are numerous Natural Language Processing tasks with analogues in Music Processing. Some of the more obvious tasks that are common between the two areas of computing are classification tasks, where the object is to determine the author, or genre of a work. But there are less obvious analogues as well, for instance part of speech tagging is related to the musical task of chord function analysis.

The NLP task that I have chosen to apply to music for the purposes of these experiments is the use of generative models to create novel works. This task was chosen because it very easily translates to an NLP task. By translating sequences of chords into strings, pre-packaged language models can be used with little to no modification.

The goals of this music generation are two-fold: (1) to generate music which approximates classical music, and (2) to generate music which sounds good.[1] This second goal is included because whereas if a model generates text that is wholly unlike its training data, it is often completely unintelligible, but if music is generated that is wholly unlike its training data, it still has the potential to sound interesting, and it can give rise to new musical ideas and inspiration. Indeed, the most practical use of generative musical models is not to create music in antiquated styles, but to provide inspiration for composers to create new, innovative works. Therefore, even if the results of this experiment fail to capture the style of the pieces the models were trained on but are still musically interesting in other ways, this experiment will be considered a success.

## 2 Related Work

[1] The amount that these two goals overlap varies depending on who you ask.

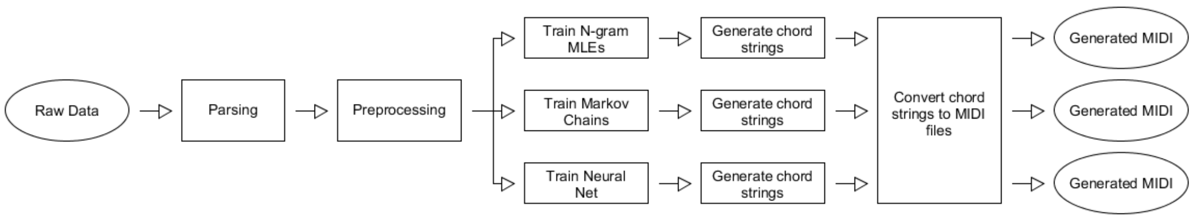

*Figure 1: Experiment setup*

The use of computational methods to generate music dates to 1957 with *The Illiac Suite,* the first piece of music to be composed by a computer (Hiller, et al., 1979). Since then, many methods of algorithmic composition have been proposed, using machine learning or otherwise. Modern music generation tasks have a wide range goals, methods, feature sets, and output formats. They are too numerous to repeat here but are described in detail in “A Functional Taxonomy of Music Generation Systems.” (Herremans, et al., 2017).

In my research, I have not found work that mirrors my method, one that converts pitch material at given moments into strings, effectively turning a music generation task into a text generation task, but many researchers have made similar assumptions to mine. (Phon-Amnuaisuk, Somnuk, et al., 2001).

## 3 Data

### 3.1 Data Set

The training and test data were obtained from the MusicNet dataset, a corpus of 300 recordings of classical music. Each recording is labeled with various data about the notes being played, including the instrument that performs each note pitch of each note, and the start and stop times of each note (Thickstun, et al., 2017). The labels are stored in an IntervalTree, which conveniently allows for retrieving information about each note that is being played at a given moment in the recording. It is from these labels that the training and test data were parsed. Since the results of these experiments were to be principally evaluated qualitatively by humans, 99% of the data were partitioned to be training data, leaving only 1% of the data left to be testing data for quantitative evaluation.

### 3.2 Parsing

The data were parsed from the MusicNet labels by extracting the pitches being played at a rate of 16 times per second with respect to the recordings, which corresponds to 16th notes at the tempo which Westergaard calls “Too fast to be useful”. It is assumed that harmonic information being skipped at this frequency would be rare. Chords were normalized by reducing the bass note to its pitch class and reducing the rest of the notes to an integer representation of their interval above the bass note compressed to one octave. Any duplicate pitch classes were then removed. A parsed musical phrase consists of a list of bass note transitions (i.e. the distance in half steps from the previous bass note) and a list of chord qualities (i.e. the set of intervals above the bass note). Because these experiments concern chord progressions, the transition between chords, any chords that had the same bass note and same quality as the previous chord were removed. Any moment of silence was treated as the end of a phrase in order to split the music into phrases, the musical equivalent of sentences.

### 3.3 Preprocessing

Because the models used in these experiments were designed for text generation, the lists of bass transitions and chord qualities that were parsed from the training data required conversion into strings. This was accomplished by treating each chord as a “word” where the first character represents the bass transition and the rest of the characters represent the chord qualities. In order to ensure that each note would occupy one character in the word, “T” and “E” were used in place of 10 and 11.

## 4 Language Model Architectures

Converting the data to strings as specified in the previous section conveniently allows for the use of models designed for text generation. Implementations of generative models for texts are aplentiful, so I made use of these rather than implementing them myself in order to save time.

### 4.1 N-gram Model

The first model used was the language model package from NLTK, which is an implementation of an N-gram model. It trains the model by counting the number of occurrences of each n-gram and the number of occurrences for each n-gram's context. Given these counts, it can calculate the probability of a token appearing following a given context, and by choosing a random token weighted by its conditional probability given the previous tokens, it can generate text.

For these experiments, I trained unigram, bigram, trigram, and quadrigram models and used each of them to generate text. To avoid data sparsity issues when evaluating the model, I used add-0.01 smoothing.

### 4.2 Markov Chain

Like the NLTK n-gram model, the Markov chain is a statistical model that depends only on the recent past. It consists of a set of states and the transitions between them, where the transitions between states are weighted with a certain probability. By choosing a random state to which to transition from the current state based on these edge weights, a sequence of states can be generated (Hayes, 2013). In the case of text generation, the states are n-grams, and sequence of states generated by traversing the graph becomes randomly generated text.

The implementation used for these experiments is the Markovify python package which allows for easy training and generation. I trained three Markov chains, with state sizes of 1, 2, and 3 respectively.

### 4.3 Neural Network

The third model used was a Recurrent Neural Network. The implementation used was the textgenrnn python package, which is built on Keras and TensorFlow. It consists of several layers of nodes and transition weights between the layers of nodes. The training process amounts to tuning the transition weights in order to more effectively capture the training data. For this application, I trained the network with 16 epochs, because any more than that would have taken too long.

When using this model to generate text, I used several temperatures. Higher temperatures mean the model is more likely to pick less likely sequences. In effect, this means that text generated at a higher temperature is more diverse and novel, but also more prone to error. For these experiments, I used temperatures of 0.5, 0.75, and 1.0.

## 5 Postprocessing

Not surprisingly, "T69 2258E E369 1258E" is not that interesting on its own. In order to achieve interesting results, the output of the language models need to be converted into something one could listen to. To achieve this, the mxm.midifile python package was used to convert the chord progressions into MIDI files. In terms of voicing these chords, the bass notes were placed between C3 and B3 and the rest of the notes were placed in the octave above the bass note. Any notes that were common between two successive chords were held over instead of being rearticulated.

## 6 Results and Discussion

### 6.1 Evaluation Metric

The evaluation of musical generative models is inherently different than that of text generative models because whereas generated text can be qualitatively said to be coherent or incoherent based on formal rules of sentence structure, music is entirely subjective. While there are rules and structures associated with the progression of chords in classical music, it is the flouting of these rules that make a piece of music interesting, whereas to flout rules of spelling or sentence structure would usually render a text incoherent. Therefore, for lack of a universal quantitative alternative, the primary metric used to evaluate

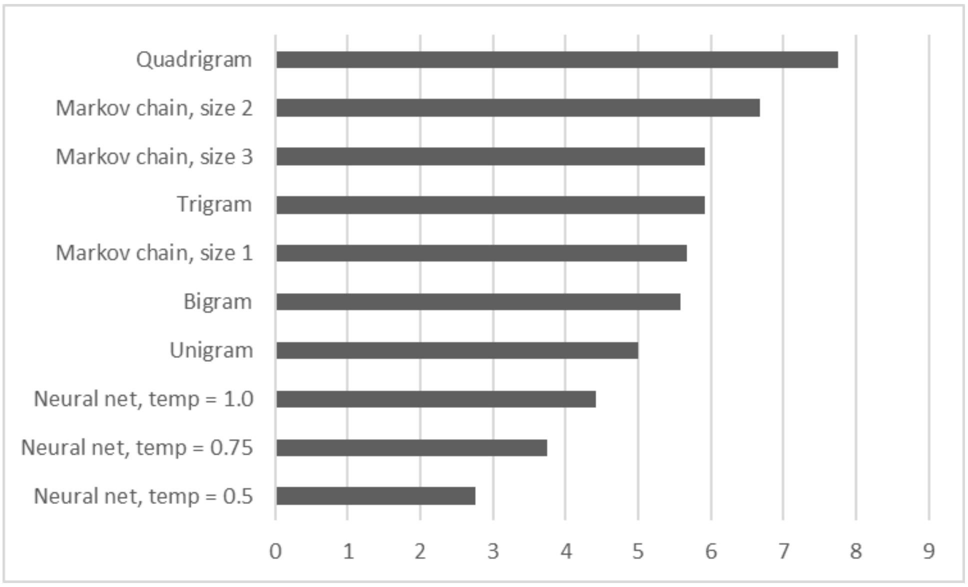

*Figure 2: Average audience scores for chords generated by each model on a scale from 1 to 10*

the results of these models is how good it sounds from a human perspective.

### 6.2 Results

The following is the average score on a 1-10 scale for chords generated by each model, collected from the opinions of family members.

#### 6.2.1 N-gram Model Results

The human evaluation results for the n-gram model generated chord progressions are as follows:

| N-gram Size | Average score |
|---|---|
| Unigram | 5.0000 |
| Bigram | 5.5833 |
| Trigram | 5.9167 |
| Quadrigram | 7.7500 |

The perplexity of each n-gram model over the test data are as follows:

| N-gram Size | Perplexity |
|---|---|
| Unigram | 95066.1553 |
| Bigram | 18267.51052 |
| Trigram | 24711.92238 |
| Quadrigram | 52429.90927 |

#### 6.2.2 Markov Chain Results

The human evaluation results for the Markov chain generated chord progressions are as follows:

| State size | Average score |
|---|---|
| 1 | 5.6667 |
| 2 | 6.6667 |
| 3 | 5.9167 |

#### 6.2.3 Neural Network Results

The human evaluation results for the neural network generated chord progressions are as follows:

| Temperature | Average score |
|---|---|
| 0.5 | 2.7500 |
| 0.75 | 3.75 |
| 1.0 | 4.4167 |

### 6.3 Discussion

By far, the model that generated the most pleasing chord progressions was the quadrigram model, with an average audience score of 7.75. However, the reason it did so well was because the model tended to generate chord progressions that were exact replicas of progressions from the training data. Indeed, when the perplexity of the test data was calculated with each n-gram model, the perplexity decreased with higher values of n as expected until it reached the quadrigram model, where it jumped up. Therefore, the high audience score was a result of overfitting, not of generating novel chord progressions.

The most surprising result was the poor performance of the neural network. Neural networks have become very popular in recent years as an alternative to statistical models, so I thought it would perform well, but the progressions it generated were mediocre at best, irritating at worst. The progressions generated at a low temperature were particularly bad. They tend to alternate between two or three simple chords, usually chords with only one or two notes. This is because at low temperatures, the model only picks the safest options, and simple chords that don't go anywhere are safe, albeit uninteresting. To improve outcomes of this model would entail additional parameter tuning, which I simply didn't have time to do given the six-hour training time.

## 7 Conclusion

In general, the music generated by these experiments did not sound very much like classical music, so it would be difficult to argue that the first goal (to generate music which approximates classical music) was met. The second goal, to generate music which sounds good, is a bit more arguable. It certainly did not generate good music consistently, but there were some phrases generated by the models that unambiguously sounded good and could reasonably be the inspiration for a more fleshed-out work. In this way, these models could be of use to a composer as mentioned earlier. Therefore, while these experiments did not meet the first goal to approximate classical music, they met the second goal to create interesting pieces of music, regardless of their similarity to classical music.

However, there are still many ways in which these models can be improved to create better music. First, some simple voice leading rules could be encoded the postprocessing in order to make the chords flow better and avoid large jumps. This would make the outputs more pleasing to the listener, but it would only superficially patch a major problem inherent in the entire approach to this experiment. The biggest shortcoming of my approach was that the parsing process unavoidably ignored a lot of integral musical information. By extracting only the pitch classes present at a given moment, other aspects of the music like melody, contour, voicing, dynamics, register, timbre, and instrumentation were lost, so the end results may have been interesting from a vertical harmony standpoint, but they lacked these key aspects of music that turn otherwise bland chord progressions into works of art. To generate music that preserves these aspects would be a task beyond the scope of what can be achieved with pre-packaged text generation.

## References

Chuan, C., & Chew, E. (2011). Generating and Evaluating Musical Harmonizations That Emulate Style. *Computer Music Journal,35*(4), 64-82. Retrieved June 11, 2020, from www.jstor.org/stable/41412947

Doornbusch, P. (2010). *Computer Music Journal, 34*(3), 70-74. Retrieved June 11, 2020, from www.jstor.org/stable/40963035

Hayes, B. (2013). Computing Science: First Links in the Markov Chain. *American Scientist, 101*(2), 92-97. Retrieved June 12, 2020, from www.jstor.org/stable/43707006

Herremans, D., Chuan, C.-H., & Chew, E. (2017). A functional taxonomy of music generation systems. *ACM Computing Surveys, 50*(5), 1–30. https://doi.org/10.1145/3108242

Hiller, L., Isaacson, L. M., & Hiller, L. (1979). *Experimental music: Composition with an electronic computer*. Greenwood Press.

Pai, A. (2020, January 21). Automatic Music Generation: Music Generation Deep Learning. Retrieved June 11, 2020, from https://www.analyticsvidhya.com/blog/2020/01/how-to-perform-automatic-music-generation/

Phon-Amnuaisuk, Somnuk, et al. (2001). The Four-Part Harmonisation Problem: A comparison between Genetic Algorithms and a Rule-Based System. Proceedings of the AISB'99 Symposium on Musical Creativity.

Schenkman, W. (1979). Language, Music, Musical Language Or Translating Latin into Music. *American Music Teacher*, *29*(1), 14-17. Retrieved June 12, 2020, from www.jstor.org/stable/43538218

Thickstun, J., Harchaoui, Z., & Kakade, S. (2017). Learning features of music from scratch. *ArXiv:1611.09827 [Cs, Stat]*. http://arxiv.org/abs/1611.09827

Westergaard, P. (1975). *An introduction to tonal theory*. Norton.