Music Score Generation

Kritin Dhoka **Rohan Gupta** **Zain Merchant**

Abstract

In this paper, we discuss applying NLP methods to the field of music, specifically score generation. We use the music21 library to train our models on a large corpus of classical music scores. We use three different models – an n-gram model for the baseline, a recurrent neural network (RNN), and a transformer model. We find that the RNN out performed the n-gram model and the transformer out performed the RNN. We evaluate these models on human evaluation as well as perplexity and BLEU scoring. We also discussed the importance of using more than just notes when attempting to generate scores and discuss the nuance of creating music acceptable to the human year.

1 Introduction

Music is an integral aspect of human culture, having existed in some shape or form for thousands of years. Over the course of musical history, it has evolved naturally but also with specific human intention through innovations like standardized musical notation. Just like language, music is a sequence of tokens from a given vocabulary, and can be analyzed using NLP techniques. Current NLP efforts surrounding music generally focus on Text-To-Music, the act of generating musical sound given a textual-prompt, or generating lyrics. There is comparatively less work on the topic of having NLP models tasked with completing existing or generating new musical scores. Central to our research are the questions: is music able to modeled using NLP methods? Given a dataset of several musician's transcribed works, can we use NLP models to generate melodies that are acceptable to the human ear in the style of that composer?

2 Existing Work

Some existing work include MusicGen and Museformer.

MusicGen is text-to-music rather than score generation, it uses a single language model over several streams of tokens representing music. Both automatic and human studies were used to evaluate the model and ablation studies were used to examine the importance of each component. (Copet et al., 2023)

Museformer is on symbolic music generation via a coarse and fine-grained transformer. The fine-grained attention has a token of a specific music bar attend to all tokens most relevant to the music structures, whereas the coarse-grained only has the token attend to summarizations. This allows the model to capture both musical structure and context, but also be able to model longer music sequences. (Yu et al., 2022)

3 Hypothesis

We believe that by applying NLP methods for sequential data, we can generate musical melodies that are acceptable to the human ear and that mimic the style of specific songs or artists.

4 Methodology

4.1 Phasing of Project

Over the course of this project, we received peer and instructor feedback in regards to our preliminary and intermediate results. Based on these, we increased the size of our dataset and modified the transformation we applied to it and we changed the evaluation metric we used afterwards as well. As such, the methodology and evaluation of this project can be split into two phases, Phase 1 and Phase 2, which is how we will be referring to the different general approaches we took. Phase 1 is the approach we took prior to receiving feedback in class, and Phase 2 is our attempt to incorporate that feedback and the results of that.

In Phase 1, our main approach was to train a model for each composer by only training that model on that composer's data and the data only tokenized the note feature, not any other part of each musical composition like timing or tempo. We used 4 composers for this, which we will specify in the upcoming sections. The feedback we received for this approach, based on our results for it, was that we used too little data, since each mode was only given one composer's data, and that it was overfitting to that composer based off that data. In Phase 2, we attempted to solve both issues by creating one model that used all the composer data in the hopes that it could generalize and effectively learn the differences between the different composers and generate scores that could still be in the style of a specific composer based on the input sequence given to it. We more than doubled the number of composers from phase 1 to 9 in total and then combined their data together and fed that into the model. We still used a Transformer, RNN, and N-gram as our models to train and test.

4.2 Data set

The dataset we used is the Core corpus of music21, a Python toolkit for computer-aided musicology made at MIT. All of the works in this corpus are from composers who lived centuries ago, so licensing fees are not an issue — all the works are accessible free of cost. These works are in the form of .mxl or .krn, both of which are file types that allow music to be read into a Music XML Reader. These files contain information about what notes or chords are being played and when they are being played. It's essentially a digital file format for sheet music.

In phase 1, We chose to focus on 4 historically significant composers and their works: Mozart, Monteverdi, Bach, and Beethoven. Each of those composers has a large enough data set to train on, as explained in section 4.3.

In phase 2, we expanded the existing composer choices and added Haydn, Josquin, Schubert, Weber and Palestrina, in addition to the previous composers. The complete makeup of this new dataset will also be explained in section 4.3

4.3 Data Transformation

In music21, each score contains a series of music21 objects corresponding to sheet music, such as Note, Chord, Measure, Part, etc. Our goal for phase 1 was to simply just get the sequence of core sounds, i.e. the notes, and simplify away the more complex music objects. We transformed our data from the sequence of music21 objects to just a sequence of strings, each one being from the "vocabulary" of music. This musical "vocabulary" is just the 12 notes that appear in all of music: A, A#, B, C, C#, D, D#, E, F, F#, and G. We performed the following transformations on the raw scores provided for each composer by music21:

- Transposed the score from its current key to C major
- Set the octave to 4 for each note
- Transformed each chord to its root note

The output of the transformation is now just a sequence of notes that represent how each score would have been played in the key of C major and octave 4. Each of these notes can now be considered our token. This transformation was made so that the general pattern of what the composer plays is emphasized rather than the musician's choice of key. This sequence of notes can be transposed to any other key, and the notes will shift by the same intervals, and given the prevalence of C major being so foundational in music theory, we chose to standardize it to this key. After standardizing the data, we combined all the notes from each song into one single array of notes. This can be thought of as all the notes ever written by the composer ordered in the sequence in which they are played in their scores. We then used this to produce our n-gram counts.

Retrieving and transforming the data takes about 2 min and 30 seconds in our Jupyter notebook in a Python 3.11 kernel.

For phase 2, we wanted to incorporate the duration of the note. Instead of our vocabulary being just the 12 notes, we took the timing for each note and appended it to the corresponding note name. Instead of simplifying each musical object in the sequence to a note string, we simplified it to a 'note-duration' string, such as 'G-quarter'. We got 102 unique tokens in our training set from doing this, while still doing the same type of music normalization we did in Phase 1.

The retrieval and transformation of this data set took around 5 minutes with the addition of the new composers on Google Colab CPU.

4.4 Train, Validation, and Test Data

For Phase 1, after we get the standardized and combined sequence of notes for each composer, we take the first 80% of notes for that composer as the training set, the next 10% as the validation set for hyperparameter testing, and the remaining 10% as the test set for the perplexity calculations.

For Phase 1, here are the number of notes in the train/validation/test sets for each composer:

- Mozart
  (16 compositions): 18,598/2,325/2,325
- Monteverdi
  (49 compositions): 35,087/4,386/4,386
- Beethoven
  (26 compositions): 123,003/15,375/15,376
- Bach
  (413 compositions): 89,608/11,201/11,201

In Phase 2, we increased the number of composers. Instead of combining all the compositions for each composer into one sequence of notes and doing the 80/10/10 split on that, we mapped each composition to its sequence of note-duration tokens and then took the first 80% of each composition as the training, the next 10% of that composition as the validation, and the final 10% of that composition as the test. This was better because it accounted for every composition rather than leaving out some compositions like phase 1 had. This approach produced the following train/validation/test breakdown per composer:

- Mozart
  (16 compositions): 18,593/2,324/2,331
- Monteverdi
  (49 compositions): 35,065/4,384/4,410
- Beethoven
  (26 compositions): 122,995/15,375/15,384
- Bach
  (413 compositions): 89,448/11,187/11,375
- Hadyn
  (9 compositions): 11,272/1,410/1,413
- Josquin
  (1 compositions): 2,708/340/342
- Schubert
  (1 compositions): 1,037/130/130
- Weber
  (1 compositions): 2,649/331/332
- Palestrina
  (500 compositions): 189,825/23,737/23,979

One note is that Music21 contained 1319 compositions by Palestrina, but attempting to use all of them as part of our dataset caused memory usage limit errors during dataset processing. Specifically, the remote server on Google Collab would crash regardless of runtime type. As such, we had to reduce the number of compositions taken from Palestrina to a random subset of 500 compositions.

When the composers were aggregated together, the dataset had the following train/validation/test split:

- Overall
  (1016 compositions): 473,592/59,218/59,696

The above data ended up being 5MB in size.

4.5 Approach

We first used a standard n-gram model, with n going from 1 to 5. Our vocabulary in phase 1 is just all the 12 notes in music from A to G. In phase 2, that vocabulary changed to 102 unique 'note-duration' tokens. In our n-grams, our sequences were the previous n-1 notes with the nth note being the current note. After we calculated the probabilities of each sequence given its context, we created two probability functions, one with and one without interpolation. Within the probability functions, to prevent log(0) from happening, we assume ϵ is a small value of 10^{-5}. If we don't do this, we get infinite perplexity since some sequences in the validation set aren't seen in the training set.

Given we have a probability function with interpolation and a perplexity computation function, our hyperparameters end up being the 5 lambdas and ϵ. We showcase the results of our hyperparameter testing in Figures 1 and 2 for Phase 1.

For our RNN, we mainly used the PyTorch library. The RNN itself consisted of 1 hidden layer and 1 output layer and was trained using cross-entropy loss. One important modification we had to make when moving from the n-gram to the RNN was how to encode the notes such that they could be fed into the RNN. We settled on using a unique Note to ID mapping, which was reflecting in our code as two dictionaries, one using the notes as keys to the their respective, unique integers and the other using integers as keys to a unique note. When training the model, we split the train data into pairs of a sequence of 40 consective notes and a single subsequent token.

For the Transformer, we used a similiar encoding structure to the RNN note to ID mapping. We used the out of the box PyTorch Transformer model with the Adam optimizer and cross entropy loss. For the encoder we used the nn.TransformerEncoder and for the decoder we used nn.Linear. We had 8 transformer heads, 6 encoder layers, 6 decoder layers, and feedforward networks of dimension 2048.

Between phase 1 and phase 2, our architecture for each model didn't change, only the data we passed in and the final layer output for the RNN and Transformer changed because the vocabulary changed.

4.6 Generating Scores

For each composer, we generated scores in 4/4 time signature that each have 16 measures, or 64 quarter notes.

For phase 1, to generate scores, we first randomly sample the first note by just picking one of the 12 notes in music. Then, we use that first note as the prefix for a musician's model to complete 16 measures for, which is 63 more notes. The generated notes are quarter notes appended to the sequence one by one to generate a musical score in the time signature of 4/4, which when opened through MuseScore 4, can be played with a variety of instruments and vocal ranges.

Through standardizing the generated score to be in 4/4, we can generate an outline for the user of what specific notes one could play, but leaves it up to the user on how to play them, which requires a lot more music features, like sustain and different kinds of notes, beyond just the note itself. In other words, this model generates "what to play", not "how to play", with regards to single notes.

In Phase 2, we tried to do melody completion tasks as our generations. We chose a specific composition, chose the first 20 note-duration tokens of that composition as the prefix/context, and then generated another 44 notes to end up with another 64 note/16 measure composition.

4.7 Evaluation

Our method of evaluation is based on two methods: objective NLP-metrics like perplexity and subjective feedback from people on the generated scores.

In phase 1, we look at the perplexity on the test split to see how we can change our hyperparameters to have the generated score match more closely with the target composer.

In phase 2, due to concerns of overfitting and perplexity showing little variation between the RNN and the transformer as per feedback recived from peers and instructor, we shifted to evaluating via the metric BLEU. BLEU is an n-gram based matching metric, which compares sets of sequences, one of which is model-generated and the other is treated as the reference. For our purposes, we gave the models the opening 10 notes in each work in the test set, and had them generate a completion to the work. The generated ending and the original ending were then used to calculate BLEU scores.

Another test for music generation models is a human listening test. For the purposes of this experiment, this is a subjective evaluation wherein 6 people are asked to score each generated sample on a scale from 1 to 10 based on personal preference, 1 being worst music sample and 10 meaning best music sample. In phase 1, it was hard to tell how good the music generated was between the generations since they all had the same timing. For phase 2, it was quite easy to tell that the generations were not as musically pleasant due to odd duration of notes, especially for the Transformer generations. That's why we chose not to collect any human evaluations for Phase 2 generations because it was quite obvious the generations were not of good quality.

5 Results

5.1 N-Gram

We have 3 tables to showcase our results across different hyperparameters for the Phase 1 n-gram model.

Figure 1 shows the values for interpolated and non-interpolated perplexity when we keep a fixed value for ϵ, which we set to 10^{-5}, and different values for each of the lambdas. We have 5 lambda values, corresponding to each of our 5 n-grams. From when n = 1 to 5, our first set of lambdas is [0.2, 0.2, 0.2, 0.2, 0.2], so each of the n-grams gets equal weight. In the same order, our second set of lambdas is [0.4, 0.3, 0.15, 0.1, 0.05], which emphasizes the n-grams where n is smaller. Our third set of lambdas is [0.05, 0.1, 0.15, 0.3, 0.4], which emphasizes the n-grams where n is larger. We compute the perplexities across all of these while fixing our epsilon for Figure 1.

Figure 2 shows the values for interpolated and non-interpolated perplexity when we keep a fixed value for our test set using the lambdas that led to the lowest perplexity for each composer. We fixed ϵ here to be 10^{-5}.

We found that when we vary the hyperparameter of ϵ between 10^{-5}, 10^{-6}, 10^{-7}, fixing the lambda set to all be equal, we get very minimal differences for the values of the perplexities using interpolation while the perplexities not using interpolation will vary as a function of that hyperparameter. This leads us to believe that the more significant result lies in the perplexities with interpolation as this hyperparameter is essentially just what we add to a probability to ensure it's not 0. It is one method of smoothing, and we will explore other smoothing techniques for the final report.

Figure 3 shows the average score given to each generated score from each musician based on asking 6 people their rating of the score from 1-10 after listening to it.

We saw high values for the perplexities when using the non-interpolated probability function, which just depends on our ϵ values. All it shows is that the non-interpolated probability dealt with sequences in the validation set that were not in the training set. Across the board, Bach was the composer with the smallest perplexity, which was true even for the non-interpolated perplexities. However, the perplexity wasn't that much higher for the other 3 composers in comparison.

For the different lambdas during interpolation, Mozart and Monteverdi achieved the lowest perplexity with the equal lambdas, while Beethoven and Bach achieved the lowest perplexity with the lambdas emphasizing larger values for n.

Validation Set Perplexity for Different Lambdas	Mozart	Monteverdi	Beethoven	Bach
Equal, Non-interpolated	136.745	278.575	88.712	11.905
Equal, Interpolated	6.911	9.308	9.161	6.149
Small Ns Emphasized, Non-Interpolated	136.745	278.575	88.712	11.905
Small Ns Emphasized, Interpolated	7.42	9.432	10.09	7.283
Big Ns Emphasized, Non-Interpolated	136.745	278.575	88.712	11.905
Big Ns Emphasized, Interpolated	7.16	10.27	9.12	5.587

Figure 1: Validation Set Perplexity for Different Lambdas

Test Set Perplexity for Each Composer	Mozart	Monteverdi	Beethoven	Bach
Interpolated	7.577	7.325	7.735	5.822
Non-interpolated	252.708	59.232	40.038	14.454

Figure 2: Test Set Perplexity for Each Composer

Composer	Score 1	Score 2	Score 3
Mozart	4.16	3.5	4.5
Monteverdi	3.57	4.5	4.33
Beethoven	3.83	3.33	4.16
Bach	4.5	4	4.67

Figure 3: Average Rating Per Score

5.2 RNN

In regards to RNN hyper parameter tuning, we looked at 3 main hyper parameters: the number of epochs the model trains for, the learning rate and the batch size. We varied the values of these parameters and examined the effects on the validation perplexity to assess the best set of hyper parameters to use. The following tables outline the results produced based on each variation of the hyper parameters at the given values:

# of Epochs	Validation Perplexity
5	1.000855
10	1.000722
20	1.000766
30	1.000648

Figure 4: Validation Perplexity by Number of Epochs

Based on this table, the trend of how the number epochs the models trains for in relation to the validation perplexity is clearly inverse, As the number of epochs increases, the validation perplexity decreases, with 30 epochs having the lowest validation perplexity. However, we chose not to continue increasing the number of epochs further for three main reasons. First the validation perplexity did not fluctuate more than .000005 with epochs higher than 30, which would indicate that an optima can be reliably found within 30 epochs. Second, concerns of overfitting if we were to arbitrarily increase the number of epochs without limit. Third and Finally, the limits imposed on us by our limited access to computational resources in terms of hardware and time. This made it much more practical to train for 30 epochs.

Batch Size	Validation Perplexity
8	1.000828
16	1.000653
32	1.000649
64	1.000618

Figure 5: Validation Perplexity by Batch Size

The results of batch size tuning were much like those of epoch tuning. A clear, inverse relationship between validation perplexity and batch size, with the highest value, 64, producing the lowest validation perplexity. Unlike epochs though, the only reason not to go further in testing batch size was purely resource based. Any attempt to utilize the next power of 2, 128, would lead to memory issues in the GPU allocation utilized to train the RNN. As such, the value for batch size was chosen to be 64.

Learning Rate	Validation Perplexity
0.001	1.000658
0.005	1.000664
0.01	1.000521
0.05	1.00053

Figure 6: Validation Perplexity by Learning Rate

The results for varying learning rate were slightly different. The learning rate that produced the lowest validation perplexity was 0.01, and both increasing and decreasing from that value would produce higher validation perplexities. This is likely because any smaller, and the model wouldn't step fast enough into an optima, but any larger, and the model would cycle around an optima during gradient descent.

Based on these resutlts, the RNN produced the following perplexities on the phase 1 test dataset's 4 composers.

Composer	RNN Test Perplexity
Mozart	1.0010
Monteverdi	1.0006
Beethoven	1.0007
Bach	1.0012

Figure 7: RNN Test Perplexity

All of the resultant perplexities were incredibly close to 1, which would could be an indicator for over fitting. That said, within those results, Bach has the highest, followed by Mozart, then Beethoven, then Monteverdi as the lowest. Monteverdi was also the lowest in the N-gram model, but all other composers have there order changed, likely due to the RNN's ability to learn the composers beyond simply n-gram representation.

Finally, human evaluation was carried out for the the RNN with the same methodology as earlier, with 3 generated musical scores for each composer. This produced the following average ratings for each composer as generated by the RNN.

Composer	RNN Mean Human Eval
Mozart	5.25
Monteverdi	4.917
Beethoven	4.5
Bach	5.417

Figure 8: RNN Human Evaluations

5.3 Transformer

For the Transformer model, we used most of the default parameters that came with the Pytorch model. Additionally, we used a batch size of 64 and a learning rate of 0.01. In general, we found that the more epochs, the lower the loss. We ended up doing 5 epochs. The following are the transformer test perplexities for each composer.

Composer	Test Perplexity
Mozart	1.0007
Monteverdi	1.0004
Beethoven	1.0005
Bach	1.0008

Figure 9: Model Test Perplexity

5.4 Phase 1 Comparison

We have 2 tables for a side-by-side model comparison. Figure 4 shows perplexity and Figure 5 shows human evaluation scores.

Composer	N-Gram	RNN	Transformer
Mozart	7.577	1.0010	1.0007
Monteverdi	7.325	1.0006	1.0004
Beethoven	7.735	1.0007	1.0005
Bach	5.822	1.0012	1.0008

Figure 10: Model Test Perplexity

Composer	N-Gram	RNN	Transformer
Mozart	4.053	5.25	5.417
Monteverdi	4.133	4.917	5.75
Beethoven	3.773	4.5	5
Bach	4.39	5.417	5.917

Figure 11: Average Human Feedback Score

In general, the RNN outperformed the n-gram model, and the transformer outperformed the RNN both in regards to test perplexity and human evaluations. Specifically, the N-gram produces values greater than 5, but the RNN and transformer produce values close to 1. We can see the transformer's worst results, on Bach, are still better than the RNN's results on Mozart. However, both the RNN and transformer have the same ordering of best to worst in regards to perplexity. Monteverdi, followed by Beethoven, then Mozart and finally Bach. Additionally, in regards to human evaluations, Bach performed the best across the board, and Beethoven perform the worst. This may speak to some of the underlying choices these musicians make, and how universally pleasing their songs are, which our model was trained off of.

5.5 Phase 2

Composer	N-Gram	RNN	Transformer
Mozart	0.181	0	0.044
Monteverdi	0.218	0	0.044
Beethoven	0.161	0.018	0.085
Bach	0.360	0	0.021
Haydn	0.176	0.012	0.031
Josquin	0.302	0	0
Schubert	0.140	0	0
Weber	0.125	0.029	0.040
Palestrina	0.261	0.003	0.020

Figure 12: Average Human Feedback Score

In regards to Figure 6, it should be noted that cells with with a reported BLEU score of 0 were actually non-zero. Instead, the scores produced were incredibly close to 0 and were of less magnitude than 10^{-100}. Specifically, these values ranged from 4.16×10^{-206} to 5.59×10^{-104}. As such, these values provided little information for analysis and comparison and we made the decision to replace these with 0.

So while N-grams have the highest BLEU score, there are more theoretical reasons it wouldn't be the best model to work on for music. First off, while the scores are better than the RNN and Transformer, they still aren't that high. Secondly, N-grams are still too simple and depend heavily on the choice of N, lambdas, and epsilon. Additionally, for our data, certain composers have more notes than others, so their sequences of notes will be more represented than other composers in the counts and probability maps. It is much more reasonable to continue to work on a Transformer since it would be more versatile, despite the current results showing low BLEU scores.

One reason for why the Transformer made poor choices for duration in its generations compared to the N-gram is due to the fact that if the Transformer happened to select an oddly timed next token, the token after that would have to attend to the poorly chosen token along with the previous tokens. So in other words, it could take just one poorly generated token to prevent the model from generating the better tokens afterwards.

6 Future Work

We effectively tried two main approaches in the work we did. First, we tried to model each composer by training one model for each set of data belonging to a composer. Our issue, as we later found with that approach, was that the library of music21 data that we used was fairly small, while just training one model, whether it was the Transformer, RNN, or N-gram, on just the target composer's previous works, overfit the model to just that composer. Another issues is that we just looked at the note being generated rather than the other aspects of the music, like timing. Our second approach used data with tokens for the note and timing and tried to create one model that was trained on multiple composers with the goal of having a model that could perhaps learn how each composer is different. We chose the works of several more classical composers, but the size of the training data set we ended up constructing was still just under 4 MB. Additionally, we used the same set of hyperparameters for testing both approaches across all the 3 models. Issues in both our approaches lead us to believe our issue is primarily a lack of enough data and potentially a more robust hyperparameter tuning on the different models.

Future work can effectively look at transforming the data we currently have so it can contain more features to train a model on. For example, MuseFormer was trained on much more data than we had and used more features too than simply timing and note, though it did take a similar approach to normalizing the data like we did. Our RNN ended up having a little more than 280K parameters, and our Transformer ended up having a little over 22.1M parameters. Museformer ended up having 16.1M parameters and MusicTransformer ended up having 16.6M parameters. While our Transformer has more parameters, we also probably did not feed it enough meaningful data. That would most likely mean we would have to increase the training time cost if we do use a dataset that's larger but contains more features. In either case, our future work would aim to solve our current issues of having too small of a dataset and too simple of a model by transforming our data to have more features and doing more training on larger and more complex Transformer models in particular. Additionally, our Transformer model may have to attend to more musically complex objects, such as Museformer attending to the bars of music rather than just the previous notes. In other words, we may have to modify a Transformer's architecture to be more specialized to the nuances of music rather than using a general purpose Transformer for the task.

One more change we will likely make is how we choose to represent these new features. We decided that expanding the vocabulary through the 'note-duration' tokens would have been a significant enough way to represent the duration feature in addition to the note. However, as mentioned earlier, the results of the Transformer generations were not musically pleasant with the timing of notes not sounding musical at all. We think that this is due to the probability mass being too sparsely spread out amongst the choices for the next most likely note. We would therefore think that having parallel inputs, where one input stream is just the notes, another input stream is just the duration, another input stream is another feature, etc. and having multiple heads output a note, duration, and other features for a single time step may do a better job of generating the next note since each feature has its own, smaller vocabulary and the probability mass may not be spread so sparsely over a larger vocabulary.

As mentioned in the last section, if the next generated token was one that was odd or not seen during training, then using that as part of the context for subsequent token generations means that the model wouldn't know how to deal with "incorrect" sequences. This is similar to the issues encountered during teacher-forcing where the model doesn't understand how to deal with an unseen or "incorrect" sequence as the context, so future work would also aim to solve this issue.

7 Bibliography

References

Jade Copet, Felix Kreuk, Itai Gat, Tal Remez, David Kant, Gabriel Synnaeve, Yossi Adi, and Alexandre Défossez. 2023. Simple and controllable music generation. *arXiv preprint arXiv:2306.05284*.

Botao Yu, Peiling Lu, Rui Wang, Wei Hu, Xu Tan, Wei Ye, Shikun Zhang, Tao Qin, and Tie-Yan Liu. 2022. Museformer: Transformer with fine-and coarse-grained attention for music generation. *Advances in Neural Information Processing Systems*, 35:1376–1388.