Predictive Chord Generation with Machine Learning

Aditya Hariharan Rijul Raghu Varun Punater

Abstract

Music and Text share many common aspects, most notably the sequential nature of notes and words. Given our exposure to using RNNs, LSTMs, and Transformers for text-based tasks, we chose to use these techniques for the generation of chords based on a sequence of notes as an input. We wanted to train machine learning models to predict clear chord progressions from sequences of musical notes. We also chose to utilize a Naive Bayes model as a baseline along with the aforementioned architectures. We also chose to vary how our inputs were featurized, with initial testing relying on a padding, whereas further testing relying on using time based note tokenization. We also attempted methods using a more concise form of our input features, eliminating the Flats and Sharps in favor of base notes, and also looking only at the notes at each downbeat. Our findings indicate that LSTMs and Transformers perform the best, slightly above Naive Bayes and far above RNNs.

GitHub Repository: Project Code

1. Introduction

Recently, there has been a surge of progress in anything ML related since the birth of ChatGPT. As a result, there have been multiple organizations focused on creating AI tools for specific industries. One such example of this is Github Copilot, an AI suggestive software that is capable of predicting and writing code.

Musicians oftentimes find themselves struggling, and the sudden burst in progress in AI tooling seems to point clearly to some kind of musician's assistant. Oftentimes, it might be easy to come up with a melody, but finding harmonies and chords to play alongside that melody may prove to be more difficult. We focus on how melodies and chordal harmonies are closely related, hoping to create AI tooling for musicians that can help create chord progressions over pre-existing melodies.

Specifically, predicting chord progressions should have the following input and output:

Input: Musical content representable of a melody, which takes the form of MusicXML.

Output: Clear chord progressions that would be played for each measure, with either 1 or 2 chords per measure.

Final Results:

Naive Bayes: Train Acc = 20.5%, Dev Acc = 19.4%, Test Acc = 17.2%

RNN: Train Acc = 7.02%, Dev Acc = 7.47%, Test Acc = 5.44%

LSTM: Train Acc = 21.69%, Dev Acc = 19.62%, Test Acc = 19.05%

Transformer: Train Acc = 20.5%, Dev Acc = 17%, Test Acc = 18.5%

2. Related Work

2.1. Bidirectional Long Short Term Memory (BLSTM)

2.1.1. Paper Discussion

This method used a BLSTM model in comparison to Hidden Markov Models (HMMs) to see how well chord prediction would work (Lim et al., 2017). They originally worked with lead sheets, preprocessing the data into a matrix that maps time signatures, the measure number, the note itself (one of 12 pitch classes), the chord associated with that note, and the duration of the note as well.

For their model they had 2 hidden layers, using the hyperbolic tangent as an activation function alongside the softmax function for the output layer. It seems that they produced 24 time units to represent which chords are played for a particular measure (12 pitch classes, major/minor) – this means that they lack the consideration of more complex chords such as seventh chords or dominant chords, which they said could increase their user rating results.

The main challenge they face then it seems is not just input preprocessing (which is tedious, considering the conversion from lead sheets) but also attempting to simplify the data enough for the model. They simplified not only the types of chords used but also simplified it so that each measure only

used the first chord from the measure, leading to a loss of some data that could prove pivotal in creating a model for real music.

The most interesting thing, however, is the realization of why a BLSTM model works rather than a simple LSTM: they state that the model should be able to detect similar patterns even if the music is played backwards, allowing for more long term structure as the model can look both forwards and backwards.

2.1.2. Comparing it with our approach

- Since we used the pytorch LSTM model, we also used the same non-linearity function which is the hyperbolic tangent function.
- The paper chose to use softmax for the final output whereas we don't use softmax, and we instead use our final linear layer output as the output of our model. This design choice was made considering how our model did not require the softmax function since the outputs were being sent into the Cross-Entropy Loss function, which applies softmax as well.
- Another similarity is in terms of which chords of the measure they choose to use. Both approaches use only the first chord of a measure as a guide.

2.2. Machine Learning in Automatic Music Chords Generation

2.2.1. Paper Discussion

This method sought to assign chords to different measures of a melody in order to generate 'pleasant' musical compositions (Chen et al., 2015). The researchers collected 43 lead sheets, each of which feature scale tone chords. Important to note, in a lead sheet, every measure typically has one chord. After preprocessing, which involved several techniques including the removal of measures without any chords/notes, they selected features based on the notes in a measure, beats of the notes, and their duration. The following models were used:

- Logistic Regression
- Naive Bayes
- Support Vector Machine (SVM)
- Random Forest
- Boosting
- Hidden Markov Model (HMM)

Their results indicated that machine learning is effective in music generation; however, it largely depends on the choice of model, with Random Forest and Boosting performing the best for their dataset.

Some of the challenges they faced were:

- Imbalanced Data: Most of the labels were either 1 or 4, which could have impacted the performance of the models, making them potentially bias.
- Naive Bayes Modeling: Naive Bayes assumes that the features are conditionally independent; however, this may not be true because the notes in a melody might influence each other, making the assumption potentially problematic.

2.2.2. Comparing it with our approach

In terms of our dataset, we don't feature quite an issue with the labels being 1 and 4 for the most part. Their dataset was normalized to only be in one key signature (specifically in the key of C). However, we see key signature as an important feature that should be predicted (even though a simple transposition could account for any changes), and as a result we have much more chords to work with and to predict as well. Additionally, they restricted the chord types to only the 7 diatonic chords, and as a result our data might actually do worse as a result of this (in terms of accuracy, especially for Naive Bayes) as certain chords may become much less common in our case. We relate to this paper most with Naive Bayes, and we also face similar issues of how features might be conditionally independent.

2.3. Transformer-Based Seq2Seq Model for Chord Progression Generation

2.3.1. Paper Discussion

In the study (Li & Sung, 2023), the researchers designed a transformer-based sequence-to-sequence (Seq2Seq) model for generating chord progressions. This model was divided into a pre-trained encoder and decoder, allowing it to effectively capture and utilize the contextual information from melodies. This method showed superiority over traditional models like BLSTM, BERT, and GPT2, especially in handling long sequences and maintaining the harmony between chord progressions and melodies.

However, the study faced challenges in collecting diverse and high-quality musical data, which is crucial for training such models. This limitation impacted the model's ability to generate a wide range of unique chord progressions, highlighting a significant challenge in the field of automatic music composition. The research emphasizes the need for comprehensive datasets to enhance the generative capabilities of music AI models.

2.3.2. Comparing it with our approach

Our approach, using a Transformer model to predict chords from musical notes, primarily focuses on direct chord prediction based on input notes. In contrast, the study's approach involves a sequence-to-sequence (Seq2Seq) model with a pre-trained encoder and decoder for generating chord progressions. While both methods utilize Transformer architectures, the study's Seq2Seq model specifically targets the generation of sequences (chord progressions) in response to input sequences, likely capturing a broader context. Our model, on the other hand, is more focused on predicting individual chords, emphasizing the immediate context of each input note sequence. Furthermore, given that their goal was to generate chord progressions, while ours is a classification task, their entire model architecture differs from ours. They use an encoder-decoder structure while we just use encoding.

3. Dataset and Evaluation

3.1. Dataset

The dataset for this project is sourced from the 'Chord-Melody Dataset', which is available at https://github.com/shiehn/chord-melody-dataset. This dataset contains a collection of MusicXML files containing musical compositions. The Music21 library in Python is used to parse these XML files and extract essential musical information, including the (note, chord) pairings needed as input to the model. The dataset is organized to facilitate the training and evaluation of machine learning models in order to predict the chord progressions that would be laid on top of an existing melody. It consists of various songs, each of which is transposed into different keys in order to allow for a model to learn chordal relationships in various key signatures. With Music21 parsing, it makes it quite easy to get both ordinal data of which notes come first, alongside the chords associated with each note.

3.2. Data Split

There are a total of 5200 songs in this dataset, with a total of 158,796 measures after cleaning the data. To ensure a robust evaluation of the machine learning models, the dataset is split by the number of songs into the following sets:

- Training: A subset of 70% will be used to train the model. This will allow the model to learn from a broad range of compositions. There are about 3640 songs consisting of 112,329 measures of training data.
- Development: The dev set, containing 10% of the dataset, will be used to fine-tune model hyperparameters and validate the model to prevent overfitting. This includes 520 songs consisting of 16,087 measures.
- Testing: The remaining 20% of the dataset is used solely for model evaluation. It contains unseen data to assess the model's ability to generalize to new musical sequences. This includes 1040 songs consisting of 30,380 measures.

3.3. Dataset Evaluation

3.3.1. Chord Classes

The dataset contains 205 chord classes, some of which are much less common than others. The most common chords are all the major triads ($\sim$ 5000 - 6000 appearances on average for each major triad), then dominant seventh chords and minor triads ($\sim$ 1000 - 4000 appearances on average), then diminished triads and seventh chords ($\sim$ 100 - 1000 chords), and finally certain outliers that only have 4 - 5 appearances such as augmented sixth chords, major ninth chords, and other less common ones. As a result, predicting some of the less common chord classes is expected to be more difficult, but the models should ideally be accurate for most major chords. In addition, if we consider enharmonic equivalents to be the same (i.e. an A# major chord is the same as a Bb major chord), then there are really only 148 distinct chord classes. These chord classes are what our models will predict.

3.3.2. Vocabulary

The dataset contains 23 different notes (ranging from Ab to G#). These notes all appear about equally as most songs are transposed to every key, allowing each note to appear fairly equally. Additionally, when considering enharmonic equivalents, the number of distinct notes in our vocabulary goes down from 23 to 16 (i.e. A# is the same as Bb).

3.4. Model Evaluation

The performance of the machine learning model developed will be evaluated purely on accuracy. We define accuracy as the percentage of correct chord progressions for each song. Within this dataset, it is normalized to be in $\frac{4}{4}$ time with at most 16 notes per measure and guaranteed to have 2 chords per measure. As a result, the models predict 1 chord per measure, and an accurate prediction is defined by correctly predicting at least 1 of the 2 chords in the measure.

4. Methods

We had a goal of using three methods to predict a chord progression specific to some melody (i.e. a sequence of notes) in a song. We have implemented four different models as of now (three if you consider RNN / LSTM the same): Naive Bayes as our baseline, a Recurrent Neural Network (RNN) model, a Long Short-Term Memory (LSTM) model, and a Transformer Encoder Model. We will make these

predictions based on the data we are using, as referenced before.

4.1. Model: Naive Bayes (NB) as a Baseline

We have used Naive Bayes as a baseline to capture the interdependencies on a sequence of notes. Here, we chose to estimate how likely a certain measure of a melody and its associated chords are using the probabilistic approach of NB. We seek to classify each measure to a specific chord label.

The key challenge of this method is that we may not retain enough sequential information for the predictions to be accurate, hence its use as a baseline method.

Here is the step-by-step description of our implementation:

1. *Data Preprocessing*
    - Used the 'Music21' toolkit to parse our MusicXML files into objects, pickling those objects to save time.
    - The model then loads the pickle files and builds frequency dictionaries around these objects, storing them into JSON files at the end.
    - The training data is then loaded from our JSON files into an instance of our 'NaiveBayes' class.
2. *Feature Extraction*
    - For each measure in the musical score, we extracted the notes and chords.
    - We then counted the chord-note frequencies and chord frequencies needed for the probability equation P(Chord | Notes):

$$P(Chord|Notes) = \frac{P(Notes|Chord) \cdot P(Chord)}{P(Notes)}$$

3. *Model Training*
    - Trained our Naive Bayes classifier using the extracted features, using Laplace smoothing to handle zero-frequency issues. In this case x_i is one note in the measure and y_j is a potential chord class.

    Using Naive Bayes Assumption:

$$\prod_{i=1}^{n} P(x_i|y_j) \tag{1}$$

    This equation quickly becomes a summation when working in log space.

4. *Model Evaluation and Tuning*
    - Used our dev set to tune our lambda hyperparameter and prevent overfitting.

    Using Laplace Smoothing:

$$P(x_i|y_j) = \frac{count(x_i \cap y_j) + \lambda}{\sum_{k=1}^{n} count(x_k \cap y_j) + \lambda * \#notes} \tag{2}$$

5. *Model Testing*
    - Evaluated our model on our test set to get an unbiased estimation of performance.

4.2. Model: Recurrent Neural Networks (RNNs)

We have used RNNs since it captures the sequential links between the melodies and the chords. We have also chosen to use Long Short-Term Memory Models (LSTMs) for a more advanced approach using the same logical approach to predicting the chords for a sequence of notes. As mentioned in the related work, BLSTMs would also be another powerful way to model the data for more accurate predictions. However, for the sake of simplicity we stick to using LSTMs with some tweaks in how we treat the data.

Here is the step-by-step description of our implementation:

4.2.1. Model Architecture

1. Input Representation:
    - We added a <'REST'>Token to the vocabularies of notes and chords to give some kind of form to musical rests.
    - We utilized a maximum measure length of 16 notes.
    - We looked at the contents of each measure.
    - We generated each x_i as the following: for every note/rest we encountered in a measure, we added its pre-embedding value n times, where $n = duration \cdot quarter\_length$. Each final x_i tensor was of length 16.
      [*Note: This is a form of pre-padding to ensure that the model is provided with data with enough information and also does not have to pad the data with an arbitrary token.*]
    - We generated y_is as the following: For each measure we chose the first chord. We then used its index in our vocabulary and used one hot encoding to create a tensor. Each final y_i tensor was of length $chord\_vocab\_dims$.
    - We generated these (x_i, y_i) pairs for each measure we encountered in the test, dev, and train splits, creating our processed dataset.
2. Embedding Layer:

- We chose to embed the x_i inputs to the hyperparameter $embedding\_dims$. We were also aware of the dimensions of the vocab vector: $pitch\_vocab\_dims$.
- This embedding was the input to the next layer.

3. RNN/LSTM Layer:
   - The output of the embedding was passed into an RNN or LSTM of hidden dimension size $hidden\_dims$, which was a hyperparameter.
   - Since every sequence was of length 16, we now know that there were 16 hidden states.
   - The value of the last hidden state gets selected as the input to the next layer.
4. Linear Layer:
   - The value of the final hidden state was used as the input of the Linear Layer, with dimensions $hidden\_dims$.
   - The output dimensions of the linear layer was of size $chord\_vocab\_dims$, in our case a one hot encoded vector of size 205.
5. Output:
   - The output of the Linear Layer is then passed to an argmax function, which gives us the index of the chord within our $chord\_vocab$.

4.2.2. Training Architecture

- We trained the model for E epochs.
- We utilized batches of size B.
- We used the Adam Optimizer with learning rate L.
- We utilized Cross Entropy Loss for the loss function.
- We applied a gradient update depending on the loss value obtained on the batch.
- We tracked training, dev accuracy per epoch, storing the model state best dev accuracy as our final model.

4.2.3. Hyperparameter Selection:

- Model Hyperparameters:
  - $embedding\_dims$
  - $hidden\_dims$
- Training Hyperparameters:
  - Learning Rate: L
  - Number of Epochs: E
  - Batch Size: B

4.3. Input Modifications

While the original input representation is a potential choice, we were curious as to how changing the input would affect the model's performance. We experimented with modifying the input data in two specific ways:

1. **Removing Enharmonic Equivalents:**
   - Our original experiment treats notes such as G# and Ab (known musically as enharmonically equivalent notes) in different ways, when in reality these two notes have the same sounding pitch. We were curious to find out the effect removing these enharmonic equivalents has on model performance. As a result, we ran different experiments that trained different models: models that took in inputs that treat enharmonic equivalents as distinct and models that took in inputs that treat enharmonic equivalents as the same.
   - In order to treat enharmonic equivalents as the same, we essentially further pre-processed our data by replacing any instances of a flat note (notated in the data with the '-' character following the note name i.e. B-) with their enharmonic equivalent. This had to be done carefully as there is no hard and fast pattern to how this replacement happens (i.e. B- is replaced with A# but F- is replaced with E). We replaced enharmonic equivalents in not only the vocabulary but also the chord classes, reducing our vocabulary to a size of 16 and our chord classes to a size of 148 (as noted before in the Dataset section).
2. **Only Considering Notes on the Beat:**
   - The beat is defined as a note that occurs on any whole numbered beat (1, 2, 3, 4 in our case since all our songs are in $\frac{4}{4}$ time signature). Notes on the beat are often more important in music in determining what chord is actually played on that beat. Oftentimes, offbeat notes are typically passing tones, transitioning from one state to the next, and so would present noise when attempting to determine what chord should actually accompany a set of notes. As a result, considering only the notes on the beat is an important testing mechanism to see whether this could potentially improve model performance.
   - Since our inputs were standardized to a 16th note granularity (by duplicating notes, including the <'REST'>token based on their duration), we essentially can narrow our input down to every 4th note (using modulo operation). By doing this, we only consider notes on the beat and can analyze whether this makes any notable difference.

4.4. Model: Transformer

We developed a transformer-based model as it can effectively handle long-range dependencies and parallel processing. We expected this Transformer-based approach to surpass traditional RNN and LSTM methods, as it handles long-range dependencies and parallel processing. This capability is vital in music data, where understanding the context and complex patterns is key. Unlike sequential RNNs and LSTMs, the Transformer's simultaneous processing of all input data should yield more efficient and nuanced pattern recognition in musical compositions.

Here is the step-by-step description of our implementation:

4.4.1. Model Architecture

1. Input Representation:
    - We generate tensors in the exact same way as we generate for the RNNs. We convert musical notes and chords into pairings, picking the first chord in the measure as the output chord for the training pair (same as RNNs).
    - We then pad with <'REST'>tokens and to 16th note granularity like before in order to ensure standardized input. This would also allow the transformer to form more meaningful connections between the notes and the chords themselves, using multi-headed attention to understand this relationship.
    - Finally, we generated tensors from these padded representations of the measures, essentially allowing us to use the same input representation for both the Transformer and RNN models.
2. Model Architecture:
    - Embedding Layer:
        - The embedding layer makes use of the hyperparamter d_model to determine the number of features that the embedding contains.
        - This embedding was the input to the next layer, and it is essentially the same as the embedding layer from the RNNs.
    - Positional Encodings:
        - Using the same concepts as presented in the paper "Attention is All You Need," we used positional encodings (dependent on sin and cosine functions) to encode ordinal data. This is where our Transformer model really starts to differ from the RNN models we trained earlier.
        - The ordinal data is a crucial piece of information in music, as sequence is very important in coherent songs. As a result, this plays a pivotal role in our model's architecture.
    - Transformer Encoder Layer
        - The output of the positional encoding layer is passed to the Transformer Encoder layer, which consists of multiple parts.
        - We use the default TransformerEncoder from pytorch, allowing us to make full use of the benefit of Multi-Headed Attention that Transformers provide.
        - The hyperparamter $nhead$ is useful as it defines the number of heads we have for our Multi-Headed Attention
        - We utilize the dim_ff hyperparameter to determine how big the dimensions of the feedforward layers are.
        - Finally, the num_layers hyperparamter is used to decide how many encoder layers the Transformer consists of.
    - Linear Layer
        - We utilize a final linear layer that maps to the size of our chord classes. This allows us to associate the output with a specific chord, thereby serving as the prediction.

4.4.2. Training Architecture

- We trained the model for E epochs.
- We utilized batches of size B.
- We used the Adam Optimizer with learning rate L.
- We utilized Cross Entropy Loss for the loss function.
- We applied a gradient update depending on the loss value obtained on the batch
- We tracked training and dev accuracy per epoch, storing the model state's best dev accuracy as our final model.

4.4.3. Hyperparameter Selection

- Model Hyperparameters:
    - Embedding Dimensions: $embedding\_dims$
    - Number of heads: n_head
    - Feedforward Network Dimensions: dim_ff
    - Transformer Layers: num_layers
- Training Hyperparameters:
    - Learning Rate: L
    - Number of Epochs: E
    - Batch Size: B

5. Experiments

5.1. Results on Test Set:

5.1.1. Naive Bayes:

Our test set accuracy for Naive Bayes is 17.2%. This is likely due to the fact that Naive Bayes depends on the independence of features, whereas in our dataset, the notes within a chord are dependent on each other.

5.1.2. RNN:

Our test accuracy is 5.4%. This is quite low, but outlines the need for a more complex model to understand the data.

5.1.3. LSTM:

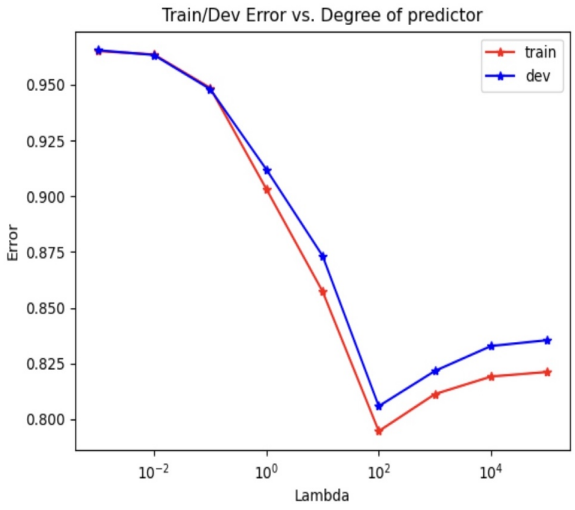
Our test accuracy is 19.05%. Surprisingly, this is quite low despite the use of an attention mechanism. Our training accuracy hit nearly 30% at one point (not the best model), but clearly that model was gravely overfit. As a result, the model does not perform as well on a test set.

5.1.4. Transformer:

Our test accuracy is 18.5%. Unfortunately, the model's training accuracy hovered around the 20% point, suggesting that it had reached what we assume to be a local minimum. As a result, the model could not perform as well when evaluated on the test set.

5.2. Results on Dev Set:

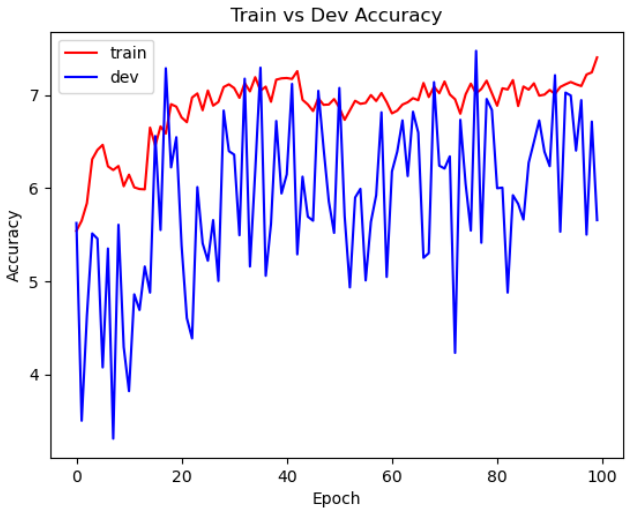
5.2.1. Naive Bayes:



Analysis:

1. Training Error Trend:
   The error starts off high at lower lambda values. As lambda increases, the error decreases sharply and then levels off after lambda reaches exactly 10^2.
2. Dev Error Trend:
   The dev error begins similarly high at low lambda values. As lambda increases, the development error decreases, but not as sharply as the training error. After reaching its lowest point, the development error starts to increase again slightly as lambda continues to increase. It seems like the dev error closely follows the training error curve.
3. Overfitting and Underfitting:
   At lower lambda values, both training and development errors are high, suggesting that the model is unable to truly give accurate predictions. Around lambda values of 10^2, the training error is minimized, but the development error starts to rise again. This suggests that as lambda increases beyond this point, the model might be starting to overfit to the training data. The optimal lambda value seems to be around 10^2. Unfortunately this seems to be a quite high lambda value, and certain issues with Naive Bayes become clear as we will explain in the analysis section.

5.2.2. RNN:



Hyperparameters: $E = 100,\ B = 64,\ L = 0.1$

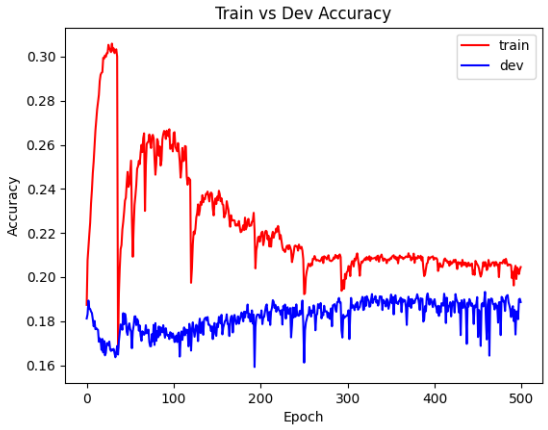
Analysis:

1. Training Accuracy Trend:
   The training accuracy is relatively consistend and increases during the first 10 epochs and after a slight dip hovers around 7% for the majority of the remaining epochs.
2. Dev Accuracy Trend:

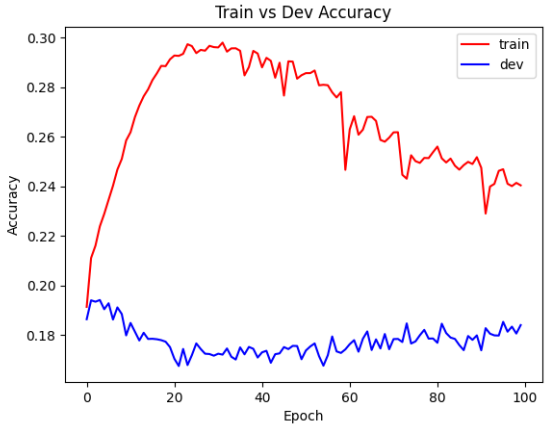
The dev accuracy has a very high variance relative to the training accuracy. No matter the epoch, we see highly fluctuating results ranging from as low as 3.5% to values as high as 7.5%.

3. Optimal Epoch Range:
   Based on the plot, we cannot isolate a certain epoch range for which to use as a guide for training. Even at higher epoch ranges, such as during training with 500 epochs we observed a very similar trend in terms of the train and dev accuracy values. For this reason, it appears that not much additional learning occurs at those high epoch values.

5.2.3. LSTM:



Hyperparameters: $E = 500,\ B = 64,\ L = 0.01$, removed flats



Hyperparameters: $E = 500,\ B = 64,\ L = 0.01$, only notes on downbeat

Analysis:

1. Training Accuracy Trend:
   - Looking at the first graph, this graph ran for 500 epochs, a fairly long time in this case. The training accuracy quickly increases to 0.3 in the first 50 epochs, and after that starts to see drastic changes. It drops suddenly, then rises, and then continues to drop, overall stabilizing only around the 300th epoch.
   - The second graph is much different. Instead, the training accuracy begins from a low point, peaking around the 25th or 27th epoch. From there, it continues to drop overall. This could be a result of using PyTorch's Adam Optimizer, which often performs really well at the beginnning, but is also prone to overfit after a point, leading it to a local minima that it does not seem to escape from.

2. Dev Accuracy Trend:
   - The dev accuracy in the first graph is a good bit different from the training. It peaks quickly, then drops and steadily increases, slowly stabilizing around the 400th epoch.
   - The second graph shows a similar trend, just much less sparse as it ran for much less epochs. It peaks quickly within the first 10 epochs, and then drops and continues to rise, never quite rising back up to the same height however.

3. Optimal Epoch Range:
   Given the plots, it seems that the model's best performance happens very quickly, within the first 10 epochs. Though it seems to steadily increase up to 500 epochs, running it for longer (1000 epochs) also proved to have no real gain, and as a result we choose to keep pretty much all our tests within 100 epochs (especially due to how long each epoch takes with smaller batch sizes and learning rates). Learning up to a 100 epochs allows the model a chance at least to overcome any local minima it gets stuck in during gradient descent, but the accuracy for the most part levels out and does not improve much.

Analysis of Tables 1 and 2 (pictured below):

- One important consideration is that we attempted to vary the in-model hyperparameters, but the choice of $embedding\_dims = 32$, $hidden\_dims = 256$ proved to be the most optimal.
- The choice of our training hyperparameters can be seen at the head of each table. Overall, we can see that a smaller batch size $B = 64$ performed better than the larger batch size $B = 512$.
- The model with the smaller batch size of $B = 64$ often reached its best epoch quicker than that of the larger batch size $B = 512$, in all the examples with $E = 100$.

Table 1. Results for following hyper-parameters with different experimentation methods: $E = 500,\ B = 512,\ L = 0.01$

		FLATS REMOVED	ONLY WHOLE BEATS
BEST EPOCH	7		
	9	×	
	12		×
	8	×	×
TRAIN ACCURACY	0.1945		
	0.1863	×	
	0.1944		×
	0.1965	×	×
DEV ACCURACY	0.1848		
	0.1760	×	
	0.1852		×
	0.1831	×	×
TEST ACCURACY	0.1274		
	0.0968	×	
	0.1018		×
	0.1027	×	×

- On the other hand, the larger batch size $B = 512$ reached its best epoch quicker than that of the smaller batch size in the one example that was shared across both with $E = 500$.

- We can see that the Dev Accuracy values remained similar across approaches, varying within a percentage point of each other.

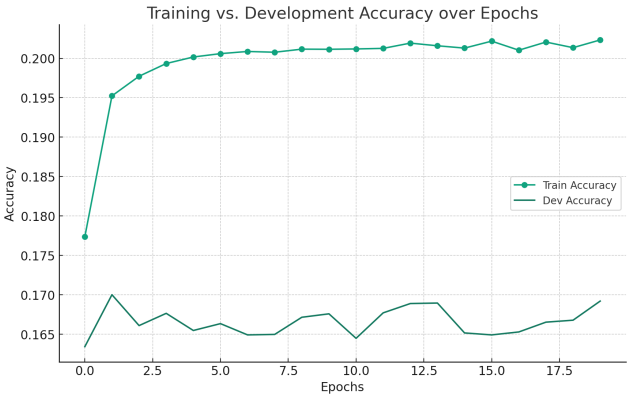
- Across both choices of hyperparameters we observe that the highest Test Accuracy is for the default model with no flats removed and not chosing only whole beats.

- One inference we can make on the basis of the Dev Accuracies is that their are very large variations in the Test Accuracy values. We arrive at the highest Test Accuracy for the model with the basic dataset with no input modifications. This may be due to the amount of information carried through the harmonically coded inputs over the standardized inputs, leading to greater generalizability.

Table 2. Results for following hyper-parameters with different experimentation methods: $E = 100,\ 500$ (for one of them), $B = 64,\ L = 0.01$

		FLATS REMOVED	ONLY WHOLE BEATS
BEST EPOCH	2		
	458	×	
	3		×
	3	×	×
TRAIN ACCURACY	0.2169		
	0.2068	×	
	0.2238		×
	0.2353	×	×
DEV ACCURACY	0.1962		
	0.1933	×	
	0.1941		×
	0.1894	×	×
TEST ACCURACY	0.1905		
	0.1368	×	
	0.1880		×
	0.1350	×	×

5.2.4. TRANSFORMER:



Hyperparameters: $embedding\_dims = 256,\ n\_head = 8,\ dim\_ff = 64,\ num\_layers = 4,\ L = 1e-4$

Analysis:

1. Training Accuracy Trend: The training accuracy starts at about 0.177 and shows a steady increase, reaching approximately 0.202 by epoch 19. As can be seen, however, the training accuracy hovered around 20% from the 4th epoch, suggesting that the model had reached a local minimum and was unable to learn further.

2. Development Accuracy Trend: The development accu-

racy, beginning at around 0.163, has major fluctuations throughout. It peaks at the second epoch with 17% accuracy. It generally lags behind the training accuracy though.

3. Optimal Epoch Range: The dev accuracy fluctuates considerably, so it is extremely difficult to determine the optimal epoch range. Important to note, the dev accuracy seems to increase from the 15th epoch onwards, suggesting that if the model were trained on a higher number of epochs, the improvements may have been seen. Furthermore, there isn't a significant gap between training and development accuracies, suggesting that overfitting has not yet become a prominent issue up to this point. Unfortunately, the model could not be run on a larger number of epochs due to extremely slow processing times.

Analysis of Methods:

- Performance Variation: All 4 methods perform differently as Naive Bayes performance is influenced by the choice of lambda. The lambda choice of a 100 does not really make much sense, implying that Naive Bayes essentially does better with a random guess rather than a true probabilistic approach. While the RNN and LSTM performance is impacted mainly by the number of epochs, batch size, and learning rate but also the other hyperparameters (it performs worse with different values of the model hyperparameters), the transformer is heavily influenced by the choices of the number of attention heads, layers, and learning rate.
- Best Performance: LSTM, with the new changes we incorporated, is our best performing model. However, our transformer implementation yields a test accuracy that is only 0.5% less than that of the LSTM.

6. Discussion

6.1. Naive Bayes

Naive Bayes was meant to be used as a baseline, and hence there were fairly low expectations of a model as crude as it is. We also faced issues in encoding satisfactory amounts of contextual information which impacted predictions greatly. After analyzing small samples of dev data around 6 measures at a time, it seems like the number of notes in a measure that pertain to a certain chord greatly affect the probability of predicting that chord negatively. As a result, even though a sequence of notes might be repeated with the same chords in multiple measures, the model sometimes predicts the wrong chord due to that wrong chord being more common overall (prior probability is higher) alongside the fact that the denominator is heavily dependent on the number of notes in a measure. As a baseline, Naive Bayes does fairly well. However, the choice of best lambda value is concerning, as a lambda value of 100 does not quite make sense. This implies that Naive Bayes actually does better as a random choice rather than a true probabilistic choice, and this makes sense since it would be hard for Naive Bayes to truly capture all the features and data present within the relationships between chords and melodies. However, a 17.2% accuracy for a somewhat random process is not bad and serves as a good baseline to test against the other models.

6.2. RNN & LSTM

Due to the limited architecture of our basic RNN and LSTM models, they were unable to perform extremely well. Despite the tuning of hyperparameters we did not achieve high levels of accuracy.

The RNN performed the worst out of any of the models we had chosen, arriving at a test accuracy value that was between a third or a quarter of the other methods' test accuracy values. One inference from this could be that the vanishing gradient problem greatly affects how the RNN learns, and hence we lose a lot of contextual information when considering our sequences of 16 notes. Moreover, since the chords we considered are played near the start of a measure, we might lose some of the most impactful information in terms of the notes to the chord played which would be at the start of the measure, which would have little importance in our current setup.

Additionally, when predicting a chord such as A major when in the key of A major, it would be more likely to predict A major if it saw the same notes that make up the chord A major. This is because those notes commonly appear whenever an A major chord is played. However, sometimes the chord played and melody on top do not line up exactly, instead resulting in passing tones that still work well musically (but maybe not as well in a mathematical and probabilistic world) creating noise that causes inaccuracies.

The LSTM did perform far better than the RNN, ending up being our most accurate model. We can observe how the use of an attention mechanism leads to far better performance and can help diminish the impact of the vanishing gradient problem seen earlier.

What is extremely interesting is how removing flats to simplify the data (and thereby reduce the number of distinct notes in the vocabulary for the input and the total number of possible classes) does not actually help, in fact harming the performance on the test set. While it does improve training accuracy ever so slightly, the test accuracy goes much farther down for any experiment involving the flats being removed. One potential reason for this is that we are essentially removing the number of features. By reduc-

ing the number of distinct notes and using the enharmonic equivalent of all flat notes / chords, we essentially make the dataset itself somewhat smaller and remove the key feature of a key signature. Each distinct note is a vital point of data that can help encode the whole new feature of a key signature, but now we remove those distinct notes. As a result, variance increases and therefore it becomes harder to find the best model in the model family that can form meaningful relationships between chords and notes.

Analyzing the data closely and examining examples that went wrong, we found that the frequency of chords can also potentially play a big factor into why the LSTM's accuracy is not quite as high as it could be. Our loss function (cross entropy loss) is highly probabilistic, and as a result chords that are less frequent are seen much less. When it comes time to make a prediction, sometimes our model would guess more likely chords for chords that are much less likely, leading to a lower accuracy. This makes sense, but other times, the model actually guesses the more unlikely chords instead of the more likely chord, which is a bit surprising. Unfortunately, this issue specifically has more to do with the passing tones issue that the RNNs faced, as notes within the same measure that are less related to the chord create noise that can cause this kind of incorrect prediction.

One final thing to note is the relatively similar performance of the LSTM with the Transformer. Practically, the transformer model architecture is far more powerful in text-based cases. In the case of notes, however, we see no large change in terms of accuracy. This could be due to inadequate tuning of hyperparameters for the transformer, and typically attempting to increase dimensions or features should result in a better fit. However, in our case it seems that increasing the model hyperparameters too much actually caused even train accuracy to go down, an interesting problem for the future.

Overall, we believe that the performance of the RNN and LSTM model architectures were not up to our initial goal of reaching an accuracy value above 50%. This is in spite of using multiple different architectures for the inputs and different experiments regarding those input formats while also performing a great deal of hyperparameter tuning.

6.3. Transformer

The performance of our Transformer model seems modest, but it is not fully optimized. Transformer models are known for their effectiveness in capturing complex patterns in sequence data, but achieving high accuracy often requires careful tuning and sufficient training.

Hyperparameters: Tuning hyperparameters such as the number of layers, the number of attention heads, the model's dimensions, and learning rates can significantly impact performance and we were unable to find another combination of hyperparameters that would yield more optimal results given our current setup.

Training Duration: Training the model for an extended number of epochs was ineffectual since the train and dev accuracies flattened after a limited number of epochs. Extended processing times also impacted the scope of training models for longer periods of time. Additionally, training the model was slow, with each epoch taking up to 35 seconds. With each epoch taking so long and only making incremental changes in accuracy, it makes it hard for the transformer model to really get up to par.

Dev Set Analysis: The model fits to the dev set immediately and from there onwards, overfits to the data, causing major variance in accuracy values.

Overall, the Transformer model faces many of the same issues that the LSTM faces when analyzing select examples. Lower frequency chords not being predicted is a big issue, and higher frequency chords are also often predicted incorrectly as some other higher frequency chord. This could simply be due to the fact that the attention mechanism gave more importance to certain note-chord pairs, resulting in certain notes weighting a chord in a certain direction (for example seeing rests and the note B made the transformer guess B major when the chord was actually G major).

In comparison to expectations, Transformer models often excel in sequence prediction tasks, but they do require computational resources to reach their full potential. The observed performance might improve with further input modifications and using more complex models.

7. Conclusion

Overall, the results for our more complex models did not match expectations. For the implementation of the transformer model, further tuning of hyperparameters followed by training for a more extensive period of time perhaps would have made a difference.

In the project, the performances of our RNN and LSTM models were highly constrained. The LSTM, inherently better at handling long-range dependencies due to its gating mechanism, shows noticeable improvement over the RNN. However, both struggled with the task at hand, which requires nuanced understanding of sequential patterns.

Despite these constraints, this project and overall exploration provided valuable insights into the application of advanced machine learning techniques in music analysis, opening avenues for further research and refinement. While it is unfortunate that the experiments did not work as intended, it shows how complicated music is and how hard of a task it truly is to do something most musicians find so

simple: put chords to a given melody that sound good and work in a musical area. Machine learning, a highly mathematical concept that relies on probabilistic ideas, may find it hard to come up with creative chords that compliment the melody, making it seem as though it might be still a distant future away before models arise that can accurately predict chords (much less come up with creative ones).

References

Chen, Z., Qi, J., and Zhou, Y. *Machine Learning in Automatic Music Chords Generation*. PhD thesis, Stanford University, 2015.

Li, S. and Sung, Y. Transformer-based seq2seq model for chord progression generation, 2023.

Lim, H., Rhyu, S., and Lee, K. *Chord Generation From Symbolic Melody Using BLSTM Networks*. PhD thesis, Seoul National University, Korea, 2017.