Music Composition with Markov Chains

Github Repository

Allison Ferner, Nehir Ozden, Harry Lynch

Probabilistic Systems Analysis - Professor Tajdini

December 4, 2025

Table of Contents 2

I. Abstract

Algorithmic music composition explores whether structured statistical methods can capture the stylistic patterns of human-composed music. In this project, we investigate and analyze Markov chains as a method of artificially composing music. Using two large symbolic music datasets, the Nottingham folk melody corpus and the POP909 modern pop dataset, we construct a complete pipeline for data preprocessing, probabilistic modeling, sequence generation, and evaluation. We also compare effectiveness of model variants by implementing both first-order and higher-order Markov chains for music generation.

In preprocessing, MIDI files are parsed using *music21* to extract pitch and rhythmic information. This data is then converted into state sequences representing musical events. These sequences form the basis for training our Markov chain models, under two conditions: pitch-only modeling, as seen in our initial model, and combined pitch-and-rhythm modelling.

We implement robust evaluation methods using both quantitative and qualitative metrics to assess the model. Quantitatively, we compute Negative-Log-Likelihood (NLL) and log-likelihood statistics per each sequence. These metrics allow us to judge generalization, compare model orders, and examine how dataset characteristics, such as genre or size, influences statistical performance. Qualitatively, we convert model outputs back into MIDI files for real-time listening and human feedback. We conduct informal human-subject evaluations where we ask listeners to attempt to distinguish between real compositions and Markov-generated sequences. We also ask them to rate their confidence in distinguishment, which helps us gauge how “realistic” the Markov-generated sequences are. Our findings indicate that while listeners can often detect randomness, especially within the first-order generated sequences, higher-order

models with rhythm do sometimes produce sequences that are difficult to distinguish from human-composed material.

Our work provides a comprehensive analysis of Markov-based music composition. Our results highlight both the strengths and weaknesses of stochastic modeling, and demonstrate that Markov chains remain a relevant and interpretable baseline for symbolic, algorithmic music composition. We conclude by discussing the limitations of Markovian approaches, their implications for musical structure learning, and directions for future work such as more complex, hybrid rule-based methods or richer state representations.

I. Introduction

Algorithmic composition is the idea that music can be generated using formal procedures or rules. Composers have historically used rule-based schemes, combinatorics, and chance operations to structure music. One of the first landmark works was Illiac Suite (1957) by Lejaren Hiller and Leonard Isaacson[1]. Using the limited computing power of the ILLIAC I, a massive machine weighing five tons, Hiller and Isaacson were able to complete a four-movement composition based on the output of a computer program. Since then, there have been several branches of algorithmically produced music. Some are purely stochastic, while others combine rules and probabilistic elements. For this project, we are focusing on stochastic methods, specifically Markov chains, to statistically compose music.

II. Literature Review

Stochastic models are a particularly common approach for algorithmic music composition. In early computer-based composition, stochastic processes looked like “random

[1] https://distributedmuseum.illinois.edu/exhibit/illiac-suite/

walks", or simple probabilistic transitions. In the Illiac Suite, some movements used probabilistic pitch generation via weighted transitions to favor consonant or small-interval progressions[2]. These stochastic techniques are good for formalizing overall tendencies in music; they can capture statistical regularities that commonly see in music, such as which notes follow which, common rhythms, and common chord progressions. However, many details are still left to the randomness that characterizes the process and introduces variation. Overall, stochastic methods are an appealing way of artificially composing music, and provide accessible ways of testing if generated sequences can resemble human compositions. For these reasons, we chose to generate music stochastically using Markov chains for this project and analyze the outputs.

Beyond composition, statistical and probabilistic methods (including Markov chains) have been used to analyze music. A study done in 2025 used higher-order Markov chains, time-delayed mutual information, and mixture transition distribution analysis to probe statistical dependencies in classical sonatas and quartets[3]. They found that higher-order models often fit composers like Beethoven, Haydn, and Schubert better than first-order models. This indicates that their music exhibits longer-range dependences, rather than immediate note-to-note transitions. Thus, we will be implementing music composition with higher-order Markov chains in addition to first-order chains. Comparing the musical output of first-order and higher-order Markov chains, using corpuses of different genres, will help us understand how statistical memory and predictability vary across compositions.

There are several alternative computational approaches to algorithmic music composition which we will not be exploring. Beyond Markov-based systems, there are hybrid systems that combine probabilistic methods with rule-based heuristics. These systems avoid some common

[2] https://sandred.com/texts/Revisiting_the_Illiac_Suite.pdf
[3] https://arxiv.org/abs/2509.24172

weakness of stochastic systems, such as the lack of variation, or overly random output. Deep learning and neural network approaches have also been used for polyphonic music composition. Amadeus[4] is a music generation framework that combines an autoregressive model for note sequences with reinforcement learning to produce melodies, chord progressions, and contrapuntal textures (multiple independent melodic lines woven together). It is clear that these advances in artificial music composition increasingly leverage the strengths of statistical learning and rule-based modelling to generate coherent and pleasing music. While it is interesting to explore what different methods of algorithmic music composition exist, we chose to focus on Markov chains because they remain relevant and are easily interpretable. They also provide a good baseline for comparing more sophisticated methods.

III. Markov Chain Theory Overview

In this project, we will algorithmically compose music using Markov chains. A Markov chain is a mathematical model describing a sequence of events, or states, where the probability of transitioning to the next state depends only on the current state rather than the full history of previous states. Formally, the Markov assumption is:

$$P(X_{n+1} = x \mid X_n = x_n, X_{n-1}, \ldots) = P(X_{n+1} = x \mid X_n = x_n)$$

given that X_n is the random variable representing the state at time *n*. The dynamics of a Markov chain are captured in a transition matrix *M*, whose *(i, j)*-th entry M_{ij} represents the probability of transitioning from state *i* to state *j*. The transition matrix will be learned from various training corpuses (detailed in *Dataset Description* section). Then, music generation will proceed by

[4] https://arxiv.org/pdf/2508.20665

randomly sampling next states according to the probability distribution defined by the current state's row in the matrix.

Additionally, we will be implementing different model variants. The simplest variant is a first-order Markov chain, where the next state depends only on the immediately preceding state. The corresponding transition matrix for this model will be size *N x N*, if there are *N* possible states. For our initial model and simple, baseline experiments, we will use first-order chains for chords / note pitches. An issue with first-order chains is that they often fail to capture longer-range patterns that commonly appear in the music corpus. Thus, we will also implement higher-order Markov chains.

Higher-order Markov chains condition the next state on the previous *n* states. We will be focusing on output from third-order and fifth-order chains (in addition to first-order). These model variants can better approximate patterns that involve longer sequences of notes. Musical patterns such as motifs and rhythmic patterns can be replicated. Ideally, these models would produce more stylistically coherent and pleasant-sounding music, though there is the added cost of many more possible conditioned states.

IV. Dataset Description

A. Music Source Material

In order to successfully generate music, we needed a high-quality set of data to train our model on. From research online, we found that the Nottingham dataset and the POP909 dataset are both strong candidates for building our training corpus. The Nottingham dataset consists of roughly 1,200 folk tunes with clear melody lines and chord annotations. It is especially useful because many of the pieces are monophonic, which provides a clean and accessible starting point for learning basic pitch and rhythm transitions using a Markov chain. Because it is folk music, the Nottingham dataset tends to emphasize stepwise motion, simple rhythms, and strong tonal centers. Thus, the Nottingham corpus is an ideal dataset for early experimentation with training the Markov model.

The POP909 dataset is a collection of 909 pop songs with aligned melody, harmony, accompaniment, and structural annotations. It offers a more modern and complex stylistic domain than the Nottingham data, with richer harmonic motion, syncopated rhythms, and multi-instrument arrangements. This additional complexity enables us to test how well our model generalizes to more diverse musical structures. Being pop music, this dataset also includes more instances of repeated motifs and choruses. This stylistic contrast in genre will be useful in comparing the model's behavior under different musical distributions.

Both datasets include MIDI files. MIDI files are ideal for our model because MIDI is symbolic, rather than audio, allowing clean extraction of pitch and duration. This also eliminates the need for special audio feature extraction or machine listening. Preprocessing the data will consequently be standardized as we can cleanly define states from each musical sequence.

B. Preprocessing the Data

After downloading these datasets from their respective github repositories, we created a script that turned these MIDI files, a *.mid* file that contains collections of instructions describing musical events (such as notes and timing)[5], into state sequences suitable for our model. We implemented a modular preprocessing pipeline that uses recursive file pattern matching to discover all the MIDI files in our project's directory. We then cleaned and parsed the data to extract musical events and construct our state sequences. We leveraged the *music21* library (Cuthbert & Ariza 2010) for MIDI file parsing and musical structure analysis to provide robust

[5] https://www.loopcloud.com/cloud/blog/5260-What-is-MIDI-and-How-is-it-Used-in-Making-Music-#:~:text=Fundamentally%2C%20MIDI%20works%20by%20transmitting,you%20might%20have%20loaded%20up

handling of various MIDI formats and music. This library is very expansive, so we utilized the converter, note, and chord classes to handle various file formats and structures in the preprocessing part of this project.

1. **Loading the MIDI files**

First, we located and loaded all the MIDI files. This was done in the *load_midi_files()* function which performs four main tasks: recursive directory scanning, path aggregation, directory traversal, and alphabetical sorting. This function also took an optional parameter to limit the number of files being processed for testing. After locating the MIDI files by recursively scanning the provided directories, we created a list to store the MIDI file paths. Then, we searched through the directories and appended each file path to the list of MIDI file paths. Last, we sorted the file paths alphabetically to ensure a deterministic processing order, which was critical for reproducible experiment results. This was because when the search for these file names happens, the order of the file names can change in the process, so we always wanted to sort it at the end to avoid this.

2. **Extracting Musical Events**

Second, we needed to extract the notes from all the MIDI files. We made a function called *extract_notes_from_midi()*, which accepts as parameters a file path, an optional track name, and quantization settings, and returns a structured list of musical events represented as *(pitch, temporal_offset, duration)* tuples. Using the *music21.converter* module, we parse the MIDI files into musical objects, or a score.

3. **Track Selection**

Next, a track is selected. It is important to note that in this function, we handled both multi-track and single-track MIDI files by checking if the musical object score had a parts attribute. If it does, and a *track_name* is specified to the function, then that track is used to extract notes from. If a *track_name* is not specified, then the first track is autoselected. Otherwise, the score is a single track file, and is used to extract notes from. An example of this track selection is from a POP909 MIDI file, which are all usually multi track files. If the *track_name* "MELODY" is passed into this function, then the function will extract notes from just the melody of the song. This ensures a robust handling of datasets with varying track naming conventions.

4. Event Extraction

A list was created, called *notes_data,* to store the extracted notes. Single notes are extracted as *(pitch, temporal_offset, duration)* tuples, where pitch is encoded as MIDI notes numbers (0-127), temporal offset represents the note's onset time in beats, and duration is measured in quarter-note units. There is also an optional quantization parameter, which if true rounds durations to standard musical values (eighth, quarter, half notes, etc.) to reduce state space dimensionality. This is needed because some MIDI files will have durations that are very precise, so we created a *quantize_duration()* function that maps any duration to one of the 8 standard values: [0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0, 4.0] using nearest-neighbor matching.

5. Chord Simplification

For chords being extracted and saved into the *notes_data* list, they are simplified to their lowest pitch to maintain a manageable state space. This means that instead of storing the full chord in our program (like C major chord = C, E, G), we use only the root pitch (so the C note). This simplification trades harmonic richness for computational tractability, though future work could explore multi-pitch representations. Again, if the quantization option is set to true, then that function is called and used. The *notes_data* list is then returned to be used.

6. State Sequence Construction

Lastly, we created a function called *preprocess_sequences()*, which converts the temporal note data (the *notes_data* list) into a sequence of states. Each state represents a musical event. By including rhythm, we are able to model both the melodic and rhythmic patterns, which is more musically interesting than just pitch alone. We begin by sorting the *notes_data* by each event's start time (its temporal offset) to ensure the model is trained on musically accurate sequences. Because MIDI extraction can produce events out of order, this sorting step is essential for preserving the correct melodic progression that the Markov chain depends on.

After sorting, we construct the state sequence by extracting either the pitch alone or both pitch and duration from each event, depending on whether rhythm modeling is enabled. If rhythm is included, each state is a *(pitch, duration)* pair. If not, each state consists only of the pitch value from the *sorted_notes* list. With this, the musical data has successfully been preprocessed, and the final state sequences for the Markov chain have been generated.

Finally, we implement the wrapper function *load_dataset()*, which orchestrates the complete preprocessing pipeline, from file discovery through note extraction to state sequence generation. It calls both the *extract_notes_from_midi()* and *preprocess_sequences()* functions to extract the notes and convert the notes to state sequences (either pitches or pitch–duration pairs) in the order they appear in the song. Each resulting sequence is then appended to a master list, *all_sequences*, which forms the complete training corpus for our model.

C. Dataset Limitations

While the Nottingham and POP909 datasets provide valuable and stylistically diverse material for training our Markov chain models, they also introduce some limitations that may affect the expressiveness of our generated music. For example, both datasets are firmly rooted in

Western tonal traditions. Despite these corpuses being different genres, they both reflect tonalities that are common in Western music, such as diatonic pitch collections, simple repetitive rhythmic structures, and standard harmonic progressions. As a result, our generated musical sequences will be biased towards these musical norms.

Second, both datasets are symbolic MIDI representations. We chose to find datasets containing MIDI files, as this format makes sense for preprocessing and normalizing the data, but it does sacrifice certain performance attributes. Some expressive nuances such as dynamics, articulation and timbral variations may be omitted. Thus, the Markov chain learns a reduced representation of musical behavior. Neither the files in the original dataset, nor the generated output can reproduce expressive qualities fundamental to human performances of musical artistry.

V. Initial Model

The initial model implements a first-order Markov chain to generate new chord sequences based on a small dataset of Beatles chord progressions. The underlying idea mirrors natural language modeling: just as words form sequences with statistical structure, chords in popular music also follow patterns that can be learned from data. The model treats each chord as a state and assumes that the probability of the next chord depends only on the current chord.

To construct the model, the chord corpus is first transformed into bigrams, where each bigram represents an observed pair of consecutive chords. This allows the system to estimate how frequently specific chords follow others. For any given chord, the model identifies all bigrams that begin with that chord and counts their occurrences. These counts are then normalized to create a probability distribution over all possible next chords. This distribution reflects the statistical tendencies of the Beatles' harmonic language: chords that frequently follow a given chord in the dataset receive higher probabilities, while rarer transitions receive lower ones.

During generation, the model begins by choosing an initial chord, which is either specified by the user or selected from the corpus. It then repeatedly samples the next chord using the computed probability distribution, producing a new sequence one chord at a time. This stochastic process ensures that each generated progression is different, but still stylistically anchored to the patterns in the training data. Finally, the generated chord sequence can be converted into a playable MIDI file using the *music21* library, allowing musical inspection and evaluation of the model's output.

This initial Markov model is intentionally simple, but it establishes the foundational approach: learning transition probabilities from real musical data and using them to generate new, stylistically plausible harmonic progressions. Starting with a small dataset and a simple model helped us develop our understanding for the tasks that we would need to complete for the final model. Additionally, we were able to play the model's output (a sequence of chords, with no duration information) on a guitar. We noticed the output (unsurprisingly) sounded very similar to *Yesterday* by the Beatles!

VI. Markov Chain Final Model

A. Model Flow and Structure

The final Markov chain model is implemented as a Python class that encapsulates all necessary functionality for training on musical sequences and generating new compositions. The model architecture follows a modular design pattern, separating concerns between data representation, probability learning, and sequence generation. This design enables flexible experimentation with different model configurations while maintaining a consistent interface for training and evaluation.

The overall workflow of the model proceeds through several distinct phases. First, preprocessed musical sequences are loaded into the system, where each sequence represents a complete musical piece encoded as a list of states. These states can represent either pitch values alone or pitch-duration pairs, depending on the experimental configuration. The model then processes these sequences to learn transition probabilities, building an internal representation of the statistical patterns present in the training corpus. Once trained, the model can generate new sequences by sampling from the learned probability distributions, producing novel musical compositions that reflect the stylistic patterns of the training data.

B. Model Definition and Order

The class maintains four primary data structures that together define the model's learned knowledge. The order parameter specifies the memory length of the Markov chain, determining how many previous states influence the prediction of the next state. The transition matrix stores the empirical counts of state transitions observed during training, organized as a nested dictionary structure where outer keys represent contexts (sequences of previous states) and inner dictionaries map possible next states to their occurrence counts. The state counts dictionary maintains the total number of times each context has been observed, enabling probability normalization. Finally, the *all_states* set tracks every unique state encountered during training, providing a fallback vocabulary for handling unseen contexts during generation.

Our implementation supports arbitrary order specification through a single parameter, allowing systematic comparison of model variants. For our experiments, we focus on orders 1, 3, and 5, representing progressively longer memory windows. The number of possible contexts grows exponentially with order. For a vocabulary of V unique states, a first-order chain has at most V possible contexts, while an *N-th* order chain has V^N possible contexts. This exponential growth means higher-order models require substantially more training data to estimate reliable probabilities for each context, and may suffer from data sparsity issues where many contexts are observed only once or not at all.

Our model handles this trade-off through its context extraction mechanism. The *_get_state_sequence()* method extracts the appropriate context for any position in a sequence based on the model's order. This helper method uses list slicing to retrieve the previous N states, where N is the order parameter. For a sequence at position i, the method computes the starting index as *max(0, i - order*) and extracts the subsequence from that index up to position i. The result is converted to a tuple, which is necessary because tuples are hashable and can serve as dictionary keys in the transition matrix.

For positions near the beginning of a sequence where insufficient history exists, the context is padded with *None* values to maintain consistent dimensionality. For example, in a second-order chain processing position 1 of a sequence, only one previous state exists. The method pads this single-element context with *None*, producing $(None, state_0)$ as the context tuple. This padding ensures that the model can process sequences of any length while maintaining the semantic meaning of the order parameter. The padding also creates distinct context entries for sequence beginnings, allowing the model to learn which states tend to appear early in musical pieces.

C. Transition Probability Learning

The transition matrix is implemented as a nested dictionary structure using Python's *defaultdict* data type. The outer dictionary maps context tuples to inner dictionaries, which in turn map possible next states to their observed counts. This structure enables efficient lookup and automatic initialization of new entries. When a new context is encountered, the defaultdict automatically creates an empty inner dictionary; when a new next state is observed for an existing context, it automatically initializes the count to zero before incrementing. This automatic initialization eliminates the need for explicit key existence checks throughout the code, simplifying the implementation while maintaining efficiency.

During training, the model iterates through each sequence in the training corpus. For each position *i* in a sequence (starting from position equal to the order), the model extracts the context consisting of the previous *N* states and identifies the next state at position *i*. It then increments the count for this transition in the transition matrix and updates the total count for the context in the state counts dictionary. This process builds an empirical distribution of transitions that reflects the statistical patterns present in the training data. The training method also populates the all_states set by iterating through each state in each sequence, ensuring that the model maintains a complete vocabulary of observed states.

Sequences shorter than *order + 1* states are skipped during training, as they cannot provide valid transitions. For an *N-th* order chain, we need at least *N* states for the context plus one state to predict, hence the minimum length requirement. This filtering ensures that all counted transitions have valid, complete contexts. The training method prints progress information upon completion, reporting the number of unique contexts learned, total transitions observed, and unique states encountered. These statistics help diagnose potential issues such as insufficient training data or unexpectedly large state spaces.

The conversion from counts to probabilities occurs dynamically during generation through the *_get_transition_probabilities()* method. For a given context, this method retrieves the dictionary of next state counts and the total count for that context from *state_counts*. Each probability is then computed as

$$P(nextState \mid context\,) \;=\; \frac{count(context \rightarrow nextState)}{count(context)}.$$

This maximum likelihood estimation provides an unbiased estimate of the true transition probabilities, assuming the training data is representative of the underlying musical style. The method returns two parallel lists: one containing the possible next states and one containing their corresponding probabilities. This parallel structure facilitates weighted random sampling during generation.

After computing raw probabilities, the method performs a normalization step to ensure the probabilities sum to exactly 1.0. While the ratio of counts should theoretically produce properly normalized probabilities, floating-point arithmetic can introduce small errors. The explicit normalization step guards against these numerical issues, ensuring that the probability distribution is valid for sampling.

A critical design decision in our implementation concerns the handling of unseen contexts. When the model encounters a context during generation that was never observed during training, it falls back to a uniform distribution over all states seen in training. This fallback ensures that generation can always proceed, though it introduces randomness that may not reflect the musical style of the training corpus. The *all_states* set maintained during training provides the vocabulary for this fallback distribution. The uniform fallback represents a maximum entropy assumption: in the absence of any information about a context, we assume all states are equally likely. While this assumption is musically naive, it prevents the generation process from failing when encountering novel state combinations.

D. Generation Algorithm

The generate method produces new musical sequences by iteratively sampling from the learned transition probability distributions. This stochastic process creates sequences that reflect the statistical patterns of the training data while introducing variation through random sampling, ensuring that each generated sequence is unique.

Generation begins with context initialization. If no starting context is provided, the model randomly selects a context from those observed during training using *NumPy's* random integer function to select an index into the list of available contexts. This random selection ensures variety in the generated outputs while guaranteeing that the initial context has known transition probabilities. Alternatively, users can specify a starting context to guide generation toward particular stylistic features or to continue from a known musical fragment. This capability enables applications such as interactive composition, where a user provides a melodic seed and the model continues the piece.

The core generation loop repeats for the specified sequence length. At each iteration, the model retrieves the probability distribution for the next state given the current context by calling *_get_transition_probabilities*. If transitions exist for the context, it samples from this distribution using *NumPy's* random choice function with the computed probabilities as weights. The sampling uses index-based selection to properly handle states that may be tuples: rather than passing the states directly to random.choice, the method generates a random index and uses it to select from the parallel lists of states and probabilities. If no transitions are available (the context was never seen during training), it falls back to uniform random selection from all training states.

After sampling a next state, the model updates the context by shifting the window forward. For a second-order chain with context *(A, B)* generating state *C*, the new context becomes *(B, C)*. This sliding window approach maintains the Markov property throughout generation, ensuring that each prediction is conditioned on exactly the specified number of previous states. The generation algorithm also supports temperature scaling, a technique borrowed from neural network language models. Temperature controls the randomness of sampling by adjusting the probability distribution before sampling. The transformation works in log space: given probabilities *P*, the algorithm first computes *log(P)*, then divides by the temperature *T*, and finally exponentiates and renormalizes. Mathematically, the scaled probabilities are $P' = e^{(log(P)\,/\,Z)} / T$, where *Z* is a normalization constant ensuring the probabilities sum to 1. Temperature $T = 1.0$ leaves the distribution unchanged, as dividing by 1 in log space has no effect. Temperature $T > 1.0$ flattens the distribution, making all outcomes more equally likely and increasing randomness. This occurs because dividing log-probabilities by a number greater than 1 compresses them toward zero, reducing the differences between high and low probability states. Temperature $T < 1.0$ sharpens the distribution, making the most likely outcomes even more probable and producing more deterministic, conservative sequences. Dividing by a number less than 1 amplifies the differences between log-probabilities, exaggerating the distinction between likely and unlikely states. This parameter allows users to explore the trade-off between faithfulness to learned patterns and creative variation. Lower temperatures produce more predictable output that closely follows the most common patterns in the training data, while higher temperatures introduce more surprising transitions that may yield novel musical ideas. A small epsilon value (10^{-10}) is added to probabilities before taking logarithms to prevent numerical errors from *log(0)*. This smoothing ensures numerical stability without meaningfully affecting the probability distribution, as the epsilon is negligible compared to any realistic probability value.

The model includes methods for computing log-likelihood, which measures how well the model predicts a given sequence. The *calculate_log_likelihood()* method iterates through each position in a sequence, extracts the context, retrieves the probability of the actual next state, and accumulates the log of this probability. The *calculate_negative_log_likelihood* method extends this computation to multiple sequences, computing the average negative log-likelihood across a validation set. This metric enables comparison between different model configurations: lower NLL indicates better generalization to unseen data. The method filters out sequences that are too short to evaluate (those with fewer than order + 1 states) and sequences that produce infinite log-likelihood values, ensuring that the average is computed only over valid, finite contributions.

Model persistence is handled through save and load methods that serialize the trained model to disk using Python's pickle module. The save method creates a dictionary containing all model states: the order parameter, transition matrix, state counts, and all_states set. The transition matrix and its nested defaultdicts are converted to regular dictionaries before serialization, as defaultdicts with lambda functions cannot be pickled directly. The *all_states* set is converted to a list for the same reason. The load method reverses this process, reading the pickled dictionary and reconstructing the model state. The transition matrix is rebuilt as a nested defaultdict structure by iterating through the loaded dictionary and populating the defaultdict entries. The state counts are similarly reconstructed as a defaultdict, and the *all_states* list is converted back to a set. This persistence capability enables sharing of trained models between sessions, comparison of models trained on different datasets, and avoids the computational cost of retraining for generation experiments. For higher-order models trained on large datasets, where training can take considerable time, this persistence is particularly valuable.

E. Final MIDI File Generation

We also created a file called *midi_generator.py* that handled turning the sequences generated by the *generate()* function into MIDI files to be used by the user to actually hear the generated music. This file contains the *sequence_to_midi()* function which handles this functionality by leveraging the *music21* library. On a high level, the function creates a musical stream object called score using the *music21* library, and sets the tempo (120 BPM) and time signature (4/4) to it, and iteratively processes each state in the sequence. How it does this processing is by first checking if the state is a pitch integer or *(pitch, duration)* tuples. If the state is just a pitch integer, a default duration of one quarter note (1.0 beats) is applied, whereas if the state is a tuple of *(pitch, duration)* then we did not need to set the duration. With this information gathered from the current state, each state is then converted to a MIDI note object with its given pitch, duration, and temporal offset, and added to the score. The score, or the stream object, is written to a MIDI file format and eventually saved to an output folder later on in the pipeline of this project. This overall conversion process perceived the melodic and rhythmic information while maintaining compatibility with pitch-only models by setting default durations.

The modular design of the MarkovChain class facilitates integration with the broader music generation pipeline. The class exposes a clean interface with train, generate, and evaluation methods that accept and return standard Python data structures. This design allows the preprocessing and MIDI generation components to operate independently, connected only through the shared representation of state sequences. Changes to the MIDI parsing logic or output format do not require modifications to the Markov chain implementation, and vice versa. This separation of concerns simplifies debugging, testing, and future extensions to the system.

VII. Training and Evaluation

For training and evaluation, we created a *train_and_evaluate.py* file that carried out data loading, model training, evaluation, and music generation all in one workflow. This section will go into detail of each part of the pipeline.

The parts of this pipeline that we have already created in other files are imported in. This includes the data loading (imported *load_dataset* from *midi_loader.py*), model implementation (imported *MarkovChain* from *markov_chain.py*), and MIDI generation (imported *sequence_to_midi* from *midi_generator.py*).

A. Train and Validation Split

The first function in this pipeline is *train_test_split()*, which takes in the list of state sequences, a test ratio number, which is the percentage of data used for validation, and a random seed for reproducibility. We first split the data at the sequence level to ensure the validation pieces are completely seen during training. This means that the data will be split by songs, not notes, ensuring that notes from the same song don't end up in both the training and the evaluation. Otherwise, the model will be able to "know" a part of the song during the evaluation period, which would not produce fair results. We wanted to ensure that we split up the data in such a way that the validation pieces are never seen during training. Overall, splitting up the data like so provides a fair evaluation of generalization. Like we previously mentioned, we generated a random seed to ensure reproducible results, and then shuffled the sequences in order to randomize which sequences go to the train or validation sets. Shuffling is necessary to avoid bias from dataset ordering. Depending on the test ratio passed in, the split point is calculated to allocate the specified portion of data to validation, and then the dataset is partitioned into training and validation sets. Then, the function returns these lists, which are ready to be sent into the model.

B. Training Procedure

The *main* function in this file carries out the entire pipeline of the project. The steps are the following:

1. Parse command-line arguments: simply uses the *argparse* library to parse the command line arguments and get all necessary information for the model.
2. Load the dataset: calls the *load_dataset* function from the *midi_loader.py* file
3. Split into train/validation: calls the *train_test_split* function explained above to generate the train and validation data sets.
4. Train model(s): creates a *MarkovChain* object called model with the order specified by the user, and calls the train function from the *markov_chain.py* file on the object with the train data set created.
5. Generate sample music: calls the *generate* function from the *markov_chain.py* file on the *MarkovChain* object, in other words the model, to generate the music sequences. Then it calls the *sequence_to_midi* function from the *midi_generator.py* file to turn the generated sequences into a playable MIDI file.

After this pipeline has run, users are able to look into a folder called output, and play the MIDI files created on garageband or an online MIDI player. Additionally, we save the model used to generate these MIDI files for reproducibility and testing.

C. Evaluation Metrics

Another part of this pipeline is the model evaluation, which we performed by creating a function called *evaluate_model*. This function takes in the trained model and the validation set, and computes multiple metrics on the validation set to assess the generalization performance of the trained model passed in.

One of the first metrics we calculated was the Negative Log-Likelihood (NLL), which measures how surprised the model is by validation– lower values of NLL indicate better prediction by the model. The log-likelihood for each individual sequence is also calculated, which enables statistical analysis of model performance across individual pieces. Higher log-likelihoods are calculated from better model predictions, which allows us to understand how performance differs across pieces. The model may work better for some pieces over others. We calculate the average log-likelihood as well, which provides a summary of the model fit across validation sequences. Furthermore, if the variation of individual log-likelihoods is high, that means the model does better with some pieces than others, and if the variation is low that means the model is more generalizable. Lastly, we calculated the coverage, which is the percentage of validation states seen during training. A high coverage means the model has seen most validation patterns, indicating that our model has a better chance of making good predictions, and the training data represents the validation data well. These evaluation metrics are aggregated into a dictionary for reporting, which the function returns.

In the following results section, we report the NLL and coverage of the model and detail our other evaluation metrics.

VIII. Results

A. Model Performance

Before reading the results of our model, it is important to understand what parameters we ran our model with. During the preprocessing, we enabled quantization with the values [0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0, 4.0] for the beats. For track selection, Nottingham took the first track, and POP909 used the “MELODY” track. The temporal offset was preserved from the original MIDI files. During the training, we did a 80/20 split for train/validation, our random seed was 42, and we did sequence level splitting. The model was configured to run orders 1, 3, and 5, and had pitch and pitch and rhythm configurations. For the MIDI generation, we set the tempo to 120 BPM, the time signature to 4/4, the temperature to 1.0, and the default duration for pitch-only sequences to 1.0 beats (quarter note). Our preliminary model ran on 200 files from both the datasets, and our final model used all files from the respective datasets.

1. **Preliminary results: 200 Files**

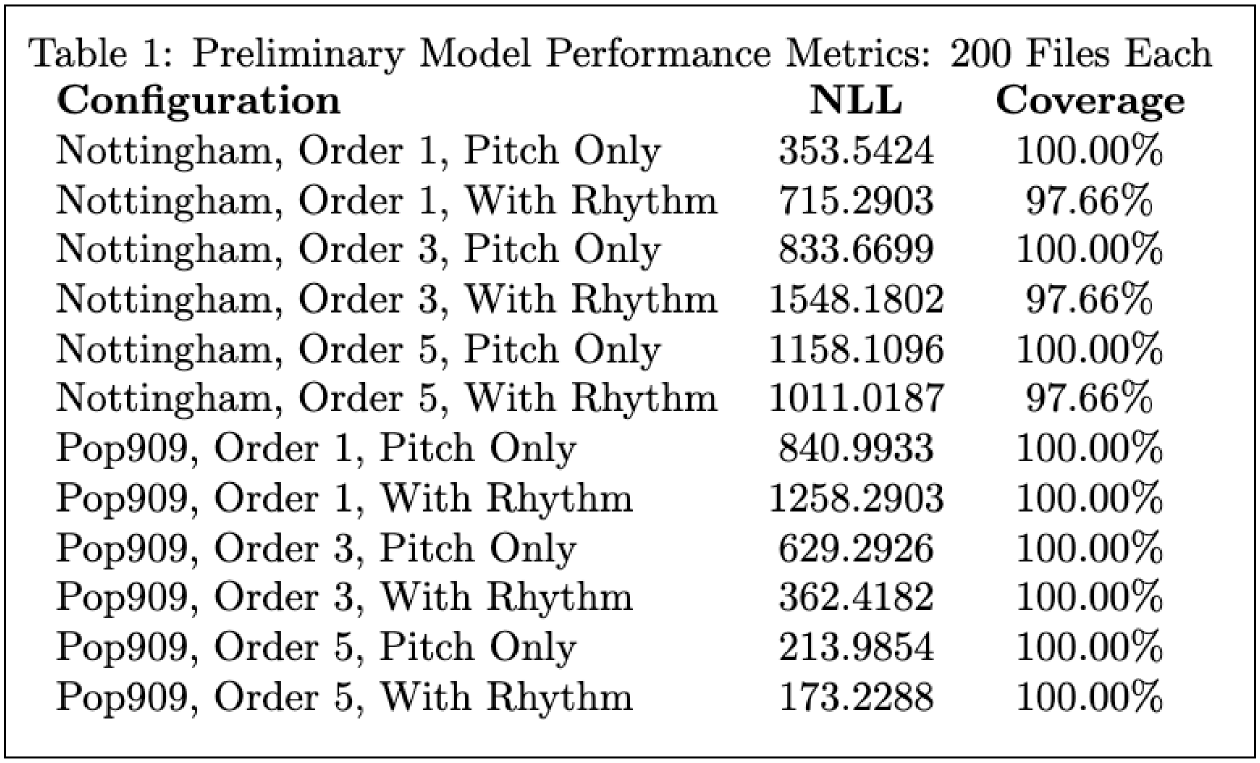

Table 1: Preliminary Model Performance Metrics: 200 Files Each

Configuration	**NLL**	**Coverage**
Nottingham, Order 1, Pitch Only	353.5424	100.00%
Nottingham, Order 1, With Rhythm	715.2903	97.66%
Nottingham, Order 3, Pitch Only	833.6699	100.00%
Nottingham, Order 3, With Rhythm	1548.1802	97.66%
Nottingham, Order 5, Pitch Only	1158.1096	100.00%
Nottingham, Order 5, With Rhythm	1011.0187	97.66%
Pop909, Order 1, Pitch Only	840.9933	100.00%
Pop909, Order 1, With Rhythm	1258.2903	100.00%
Pop909, Order 3, Pitch Only	629.2926	100.00%
Pop909, Order 3, With Rhythm	362.4182	100.00%
Pop909, Order 5, Pitch Only	213.9854	100.00%
Pop909, Order 5, With Rhythm	173.2288	100.00%

From Table 1 above, we can make some preliminary conclusions about our model. For the Nottingham dataset, the best model is *Order 1, Pitch Only*, which had a NLL of 353.54. The higher orders seem to perform worse, as well as models with the inclusion of the rhythm. For the

POP909 dataset, the best model is the *Order 5, with Rhythm* model, which has a NLL of 173.23. The higher orders seem to perform better than lower orders for the POP909 dataset, with rhythm helping its performance.

We can then infer why these preliminary results came out the way they did for the following reasons. Firstly, the Nottingham dataset is much simpler, so a low order model is sufficient enough for it to predict well. The higher orders overfit with the smaller dataset used for these preliminary results (200 files), and the inclusion of rhythm just complicates the model without enough data to support it. Secondly, the POP909 dataset is much more complex, meaning it would benefit from a high-order model, which it did. The 200 files given seems sufficient enough for the order 5 model to learn patterns, and the rhythm seems necessary for the model to capture the patterns in pop music.

Additionally, all models have from 97%-100% coverage, meaning our models represented the data well. Thus, the current results suggest that the simpler models work better for the Nottingham dataset, and complex models for POP909. However, by using a larger dataset for our final experiments, we can hope that both models will perform better.

2. Final results: All Files

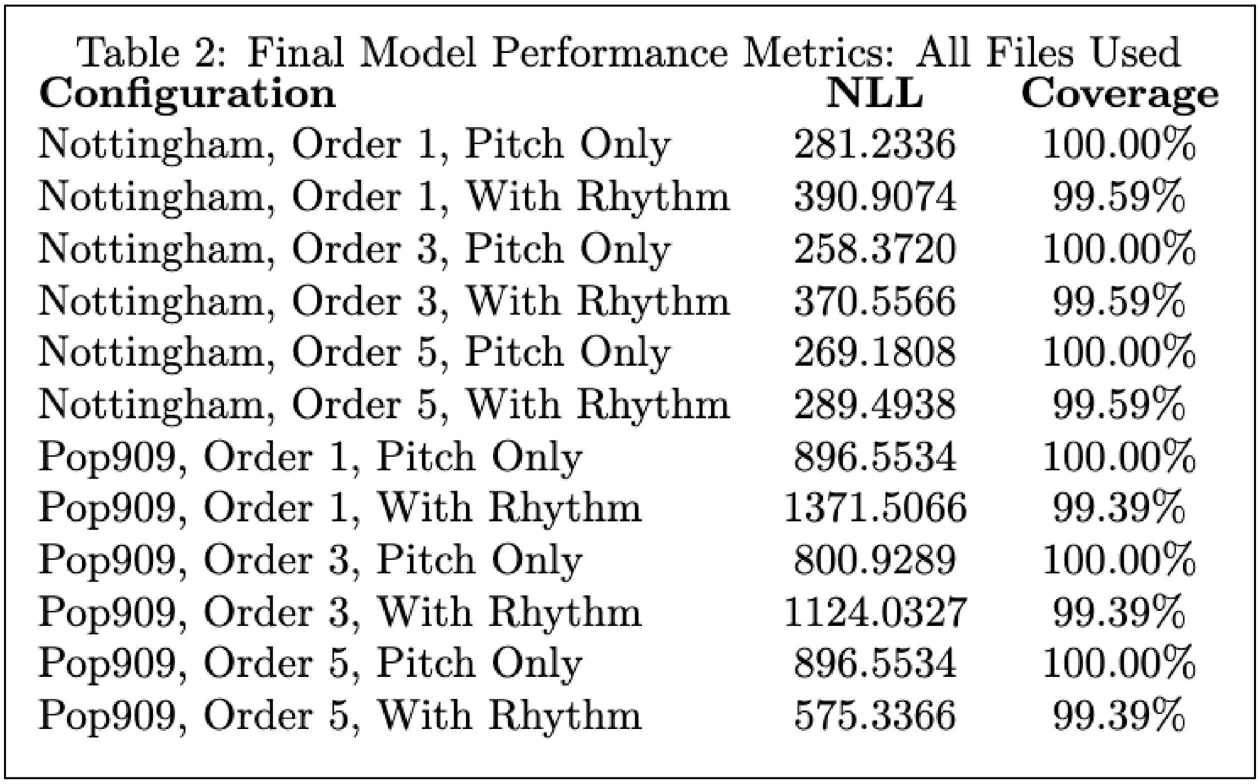

Table 2: Final Model Performance Metrics: All Files Used

Configuration	NLL	Coverage
Nottingham, Order 1, Pitch Only	281.2336	100.00%
Nottingham, Order 1, With Rhythm	390.9074	99.59%
Nottingham, Order 3, Pitch Only	258.3720	100.00%
Nottingham, Order 3, With Rhythm	370.5566	99.59%
Nottingham, Order 5, Pitch Only	269.1808	100.00%
Nottingham, Order 5, With Rhythm	289.4938	99.59%
Pop909, Order 1, Pitch Only	896.5534	100.00%
Pop909, Order 1, With Rhythm	1371.5066	99.39%
Pop909, Order 3, Pitch Only	800.9289	100.00%
Pop909, Order 3, With Rhythm	1124.0327	99.39%
Pop909, Order 5, Pitch Only	896.5534	100.00%
Pop909, Order 5, With Rhythm	575.3366	99.39%

The initial experiment using 200 files for the datasets provided baseline metrics that revealed patterns on how Markov chain order and rhythm inclusion affect the quality of the model. From the results from the preliminary run, we saw that higher-order models achieved different NLL values according to the configuration, with pitch-only models having varying performances. However, the limited dataset size (160 training sequences for both) constrained the models' ability to learn robust transition patterns, especially for higher-order chains that require more diverse context examples.

In the final experiment, which was trained on the Nottingham dataset in its entirety, showed significant improvements. All configurations for the Nottingham dataset had substantially reduced NLL values, seeing a 20% reduction for the order 1 pitch-only model, 69% reduction for the order 3 model, and a 77% reduction for the order 5 model. Clearly, the increase in training data drastically enhanced the models' ability to capture the musical patterns in the Nottingham dataset. On the other hand, the POP909 dataset showed some mixed results, in which some of the configurations performed worse. The order 1 pitch-only's NLL went from 840.99 to 896.55, order 3 pitch-only's NLL went from 629.29 to 800.93, and the order 5 pitch-only's NLL went from 213.99 to 703.67. What this suggests is that there was potential overfitting happening, or that the extra files used in POP909 training introduced stylist diversity that made it hard for the model to learn the music's patterns. Although the performance worsened for POP909, the coverage still remained consistently high (from 99-100%), concluding that both datasets still provided comprehensive musical representation. Dataset size and quality is clearly very important in Markov chain music generation, with POP909 requiring more careful model selection to balance generalization and complexity while the Nottingham dataset showing clear benefits from increased data.

B. Our Listening Judgements

After we had successfully generated music output, it was time to take a listen! We listened to a few samples from the Nottingham dataset, as well as some of our generated output from the model trained on Nottingham. We definitely noticed an increase in stylistic quality with the higher-order generated samples. The first-order music sounded pretty random, both in pitch and note duration. In comparison to the Nottingham generated music, the POP909 generated music generally had a faster tempo and shorter note durations. There were also a frequent amount of many consecutively repeated notes. Similar to the Nottingham samples, the higher-order compositions sounded much more expressive and natural than the first-order compositions. It is also worth noting that we were listening to the extracted melodic line from the original songs. This made it harder to differentiate between human-composed and computer-generated samples, as removing the harmony and accompaniment reduces many of the musical features that contribute to a song's stylistic soundscape.

C. Human Feedback

We also sent out an informal survey to a group of humans with varying musical skill. Given an audio file they were asked whether they believed the audio clip was human-composed or computer-generated. Additionally, they were asked to rank their confidence in their answer on a scale of 1-5, where 1 represents being not confident at all and 5 being almost certain.

Nottingham Dataset

Condition	**% Accuracy (participant guessed correctly)**	**Avg Confidence (of those who were correct)**
Order 3 generated	47%	3.8
Order 5 generated	60%	3.2
Human-composed sample #1	80%	4.2
Human-composed sample #2	73%	3.9

In general, participants were able to identify real music samples correctly far more often (73–80% accuracy) than they identified computer-generated music (47–60% accuracy). This suggests that generated outputs were more difficult to distinguish from human compositions. Additionally, higher-order Markov models (order 5) were slightly easier for listeners to classify as AI compared to order 3, possibly reflecting more repetitive or predictable structure.

IX. Conclusions

A. Possible Improvements

There are a few ways we could improve this project, either in the preprocessing pipeline or in the music quality of our generated sequences. One option would be adding the option to enable or disable duration quantization, which is currently always set to true. Quantization simplifies the state space and reduces noise from MIDI timings. This was helpful for processing and data extraction, but it also may have removed musical nuance or expressiveness. Running controlled experiments with quantization enabled and disabled, and comparing them, would help determine if the mode's musicality could be significantly improved through rhythmic fidelity.

Another possible improvement we could implement is normalizing the tempos and temporal offsets of the MIDI files. The datasets have a wide variety of timing structures, which may have introduced inconsistencies in the transition matrix. Standardizing tempo could yield more consistent transition probabilities and more coherent output.

B. Summary

In this project, we investigated whether Markov chain models can generate musical sequences that resemble human compositions and how effectively listeners can distinguish between real and computer-generated melodies. Using the Nottingham and POP909 datasets, we built a complete preprocessing pipeline to convert MIDI files into symbolic pitch and rhythm sequences, constructed first- and higher-order Markov models, and generated new musical samples. Ultimately, we were able to produce convincing music with Markov models and even fooled some of our friends through our human evaluation survey.