Computational Linguistics Final Project - Ayden Lamparski - lamparsa@bc.edu

Project Description

Just as Natural Language Processing (NLP) models predict the next word in a sentence based on context, this Java project predicts the next musical note based on melodic history.

This project utilizes various trigram-based algorithms to generate melodies, trained on a transcription of Oscar Peterson's "C Jam Blues".

How to Use

Compilation

Compile all Java files in the `src` directory:

```
javac src/*.java
```

Running the Generator

Run the main program from the root directory. The program will automatically run all 5 generation algorithms and save the results to the `output/` directory.

```
java -cp . src.Main
```

Output

The program generates files in the `output/` directory:

- `melody.mid`: The MIDI file of the original input melody (for comparison).
- `generated_melody_*.txt`: The raw text representation of the melody for each algorithm.
- `generated_melody_*.mid`: The MIDI file which can be played by converting it to an MP3 online.
- `trigrams.txt`: Statistical data about the different trigram frequency maps of the different algorithms.

Listening to Results

If you want to actually hear the generated melodies, look at the `sample-output/` directory and play the MP3 files. To hear the original input melody, listen to `sample-output/melody.mp3`.

Project Structure

The project is organized into the following directories:

`data/`

Contains the input training data.

- `melody.txt`: The source melody (Oscar Peterson's "C Jam Blues") represented in the custom text format. It includes chord symbols (e.g., `| C7`) and notes with their octave number and a value for their duration (e.g., `c5:4` for a quarter note, `r:8` for an eighth note rest).

`output/`

Contains all artifacts generated by the program. * `melody.mid`: A MIDI conversion of the input `melody.txt` file. * `generated_melody_[algorithm].txt`: The raw text output of a specific generation algorithm. * `generated_melody_[algorithm].mid`: The playable MIDI version of the generated melody. * `trigrams.txt`: A text file containing statistics about the learned trigram frequency maps.

`src/`

Contains the Java source code for the application.

- `Main.java`: Handles parsing input, running different generation algorithms, and saving results.
- `TrigramBuilder.java`: Responsible for building trigram maps from input data. Five different algorithms were used to do this, and the details of each are described later.
- `MelodyGenerator.java`: Uses frequency maps to probabilistically generate new melodies.
- `MidiWriter.java`: Utility to convert custom text format into standard MIDI files.
- `MusicUtils.java`: Helper library for music theory operations (note-to-MIDI, note duration math, etc.).

`sample-output/`

Contains MP3 files for melodies generated from the output by each of the algorithms, as well as for the input melody in `melody.mp3`. These were created from the generated MIDI files. Note: The Java program generates MIDI files. The MP3 files in this directory were created by converting the MIDI output using an external tool.

Computational Linguistics Concepts

1. N-Grams and Markov Chains

All of the generator models in this project are **Trigram Models**. In linguistics, a trigram model predicts the probability of a word appearing given the previous

two words.

I build a frequency map from the input text and calculate the probability distribution of every possible next note from every unique sequence of two preceding notes. This allows the generator to pick up on melodic and rhythmic patterns in the original melody.

2. Tokenization

The input music is "tokenized" into a custom string format: `[pitch][octave]:[rhythm]` (e.g., `c5:4` represents a C note in the 5th octave with a quarter-note duration).

Rests are treated as special tokens (`r:4`), functioning like punctuation or pauses in speech.

The 5 different algorithms used tokenize the notes differently. The default algorithm tokenizes notes as one string containing pitch, octave, and rhythm information, while other algorithms drop certain information like octave or rhythm, which has to be generated algorithmically later.

3. Syntax

Language has syntactic rules (grammar). Music has rhythmic rules (meter).

The generator enforces a "grammar" where every measure must sum to exactly 4 beats. If a generated "sentence" (measure) is incomplete, the model continues generating until the syntactic requirement of the notes and rests filling exactly 4 beats is met.

Algorithm Details

Understanding the Statistics

The statistics provided for each algorithm offer insight into the complexity and predictability of the generated music.

- **Total unique tokens**: The number of distinct musical elements (notes, rhythms, etc.) found in the input data.
- **Possible 2-note combinations**: The theoretical maximum number of unique bigrams (sequences of 2 tokens) possible within the constraints of the algorithm. This is calculated by determining the total set of possible tokens (e.g., all possible notes * all possible rhythms) and squaring it. This gives us an idea of the size of the "search space" for the algorithm.
- **Actual bigrams with followers**: The number of unique bigrams that actually appeared in the input melody and have a subsequent note.
- **Bigrams with multiple options**: The number of bigrams where the model has a choice for the next note (i.e., the sequence appeared followed by different notes in the source).

- **Average options for third note**: A measure of unpredictability. Higher values mean the model has more choices on average. This represents the average number of next-note possibilities for any given bigram.

1. Standard Algorithm (Exact Match)

- **How it works**: Direct application of the Trigram model. Treats "c5:4" as a unique token.
- **Pros**: Captures very specific style and melodies from the input melody.
- **Cons**: There are many possible unique bigrams in this model. For example, `c5:8 d5:8` is considered a different bigram from `c4:8 d4:8` which is different from `c5:4 d5:4`.
- **Stats**:
  - **Theoretical Calculation**:
    - **Pitches**: 7 note names (A-G) $\times$ 3 accidentals (flat, natural, sharp) $\times$ 4 octaves (2-5) = 84 pitches.
    - **Rhythms**: 3 durations (quarter, eighth, dotted-quarter).
    - **Total Tokens**: (84 pitches $\times$ 3 rhythms) + 3 rests = 255 unique tokens.
    - **Possible 2-note combinations**: $255^2 = 65,025$.
  - **Observed Stats** (from `output/trigrams.txt`):

```
--- Standard - Standard Stats ---
Total unique tokens: 49
Actual bigrams with followers: 168
Bigrams with multiple options: 37
Average options for third note: 1.37
Bigram with most options ('r:4 r:8' -> 6 options):
  [a3:8: 1, ab3:8: 1, c4:8: 2, g3:8: 1, g4:8: 1, g5:4: 1]
```

For each bigram in the frequency map, there are an average of 1.37 unique next notes in the map. This means that the output of this algorithm is very predictable compared to the others. Also, the possible amount of unique bigrams is extremely large.

- **Results**: Listening to the results, it is clear that this melody sounds the most like the original, with many phrases appearing identically as they do in "C Jam Blues". This is also probably the best sounding melody generated in this project.

2. Octave-Ignorant Algorithm

- **How it works**: Separates "Pitch Class" (C, D, Eb) from "Octave" (3, 4, 5). Bigrams in this model look like `c:8 d:8`. After generating notes with pitch class and duration, we later assign octaves in such a way as to minimize melodic jumps.
- **Pros**: More variable than the original algorithm.
- **Cons**: The contours of the melodies seem unnaturally "smooth", and sometimes large melodic jumps are indeed intended and desirable in melodies.
- **Enharmonic Equivalents**: In this algorithm (and others that use pitch names), enharmonic equivalents like `c#` and `db` are treated as distinct tokens. This is a pro in that different enharmonic equivalents often serve different melodic purposes in a piece of music (e.g., `c#` leading to `d` vs `db` leading to `c`), but a con in that it creates more possible next tokens.
- **Stats**:
  - **Theoretical Calculation**:
    - **Pitches**: 7 note names $\times$ 3 accidentals = 21 pitch classes.
    - **Rhythms**: 3 durations.
    - **Total Tokens**: (21 pitches $\times$ 3 rhythms) + 3 rests = 66 unique tokens.
    - **Possible 2-note combinations**: $66^2 = 4,356$.
  - **Observed Stats** (from `output/trigrams.txt`):

```
--- Octave-Ignorant - Octave-Ignorant Stats ---
Total unique tokens: 25
Actual bigrams with followers: 118
Bigrams with multiple options: 43
Average options for third note: 1.75
Bigram with most options ('c:8 g:8' -> 6 options):
  [a:8: 1, c:8: 1, f#:4: 1, f:8: 4, g#:8: 1, r:8: 1]
```

This algorithm has far fewer possible unique bigrams than the standard algorithm (4,356 rather than 65,025), but the trigram map has only slightly more bigrams with multiple next-note options (37 vs 43). The average options for the third note is higher than the original algorithm, making its output less predictable.

- **Results**: Listening to the melody generated, it sounds much less like the input melody than the standard algorithm, and the phrases seem to wander up and down unnaturally. This is because the algorithm does not learn what direction melodies move in, only which notes they move

between and the duration of the notes. It sounds significantly less like "C Jam Blues" than the original algorithm.

3. **Relative Scale Degree Algorithm**

- **How it works**: Tokenizes notes as intervals relative to the root of the current chord (e.g., `0:4` for a root note). This allows the model to learn harmonic functions.
- **Pros**: Generalizes melodies across different chords.
- **Cons**: Can produce awkward notes if the chord progression has non-major chords (one reason I chose a blues song like "C Jam Blues" is that it uses mostly major chords).
- **Stats**:
  - **Theoretical Calculation**:
    - **Intervals**: 12 semitones (0-11).
    - **Rhythms**: 3 durations.
    - **Total Tokens**: (12 intervals $\times$ 3 rhythms) + 3 rests = 39 unique tokens.
    - **Possible 2-note combinations**: $39^2 = 1,521$.
  - **Observed Stats** (from `output/trigrams.txt`):

```
--- Relative Scale Degree - Relative Scale Degree Stats ---
Total unique tokens: 24
Actual bigrams with followers: 126
Bigrams with multiple options: 50
Average options for third note: 1.79
Bigram with most options ('0:8 7:8' -> 7 options):
  [0:8: 1, 4:8: 1, 5:8: 2, 6:4: 1, 8:8: 1, 9:8: 2, r:8: 1]
```

This algorithm has slightly more average options for a next-note for a given trigram in the map than the octave-ignorant algorithm. This suggests that the input melody may have more patterns when notes are viewed relative to the current chord than when considered as their absolute pitch.

- **Results**: The produced melody sounds out of key at times. This is likely a result of the fact that playing common intervals based around a non-tonic chord will often produce notes that are out of key, especially if the chord is not simply a major chord. Strangely, this algorithm also seems to have more repeated notes than the others.

4. Separated Rhythm-Pitch Algorithm

- **How it works**: Uses two separate trigram models: one for rhythm (duration) and one for pitch (note + octave). The rhythm is generated first, then pitches are assigned.
- **Pros**: Can generate very coherent rhythmic structures independent of pitches.
- **Cons**: The pitch sequence might not match the rhythmic emphasis (e.g., a long note on a leading tone).
- **Stats**:
  - **Theoretical Calculation (Rhythm)**:
    - **Tokens**: 3 durations.
    - **Possible 2-note combinations**: $3^2 = 9$.
  - **Theoretical Calculation (Pitch)**:
    - **Tokens**: 7 notes $\times$ 3 accidentals $\times$ 4 octaves + 1 rest = 85 tokens.
    - **Possible 2-note combinations**: $85^2 = 7,225$.
  - **Observed Stats** (from `output/trigrams.txt`):

```
--- Separated Rhythm-Pitch - Rhythm Stats ---
Total unique tokens: 3
Actual bigrams with followers: 6
Bigrams with multiple options: 4
Average options for third note: 1.83
Bigram with most options ('8 8' -> 3 options):
  [4: 16, 4.5: 1, 8: 193]
--- Separated Rhythm-Pitch - Pitch Stats ---
Total unique tokens: 35
Actual bigrams with followers: 140
Bigrams with multiple options: 44
Average options for third note: 1.51
Bigram with most options ('r r' -> 8 options):
  [a3: 1, a5: 1, ab3: 1, c4: 2, g3: 2, g4: 1, g5: 1, r: 10]
```

As you can see, the amount of unique bigrams for the rhythm trigram map is very small. This is because the input piece has only 3 different note durations: Quarter note (4), eighth note (8), and dotted-quarter note (4.5). The trigram map for pitches has 7,225 possible unique bigrams, and of bigrams in the map, there are an average of 1.51 options for the next note. This is slightly more than the original algorithm, but slightly less than the octave-ignorant algorithm. This means that generated melody lines (considering their pitch only) are relatively predictable.

- **Results**: The rhythm feels very natural, and this algorithm produces some of the best melodies in this project in my opinion.

5. Separated Rhythm-Pitch (Octave-Ignorant) Algorithm

- **How it works**: Similar to the separated algorithm, but the pitch generation ignores octaves (pitch class only), assigning octaves later to minimize jumps.
- **Pros**: Combines the separated model with the smaller possible bigram space of the octave-ignorant approach.
- **Cons**: Loss of specific melodic contours from the original piece.
- **Stats**:
  - **Theoretical Calculation (Rhythm)**:
    - **Tokens**: 3 durations.
    - **Possible 2-note combinations**: $3^2 = 9$.
  - **Theoretical Calculation (Pitch)**:
    - **Tokens**: 7 notes $\times$ 3 accidentals + 1 rest = 22 tokens.
    - **Possible 2-note combinations**: $22^2 = 484$.
  - **Observed Stats** (from `output/trigrams.txt`):

```
--- Separated Rhythm-Pitch (Octave-Ignorant) - Rhythm Stats ---
Total unique tokens: 3
Actual bigrams with followers: 6
Bigrams with multiple options: 4
Average options for third note: 1.83
Bigram with most options ('8 8' -> 3 options):
  [4: 16, 4.5: 1, 8: 193]
--- Separated Rhythm-Pitch (Octave-Ignorant) - Pitch (Octave-Ignorant) Stats ---
Total unique tokens: 15
Actual bigrams with followers: 85
Bigrams with multiple options: 41
Average options for third note: 2.06
Bigram with most options ('c g' -> 6 options):
  [a: 1, c: 1, f: 4, f#: 1, g#: 1, r: 1]
```

As you can see, the Rhythm trigram map is identical to the previous algorithm's. The octave-ignorant pitch trigram map has only 484 possible unique bigrams, and the highest amount of average options for third notes than any other trigram map in this project. This makes it the most variable.

- **Results**: Still less coherent than the standard algorithm. The melodies are in-key and interesting without sounding too much like the input

melody. However, the contour of the melody suffers from the same problem of the octave-ignorant approach where melodies wander through a wide range of octaves.

Relationship to Previous Work

This project is based on the "Text Generator" assignment which generated random sentences using trigrams with "Sam I Am" as input data.

Reused & Adapted Core Logic

- `computeFrequencies`: Adapted to map musical tokens instead of words. A version of this function is written for each of the algorithms.
- `getNextWord`: This remains largely the same.
- `generateText`: Adapted to generate melodies of specific duration (amount of measures) instead of sentence length.

Limitations

None of the algorithms take into account where one is in the measure (aside from being forced to generate 4 full beats of notes at a time).

The input melody is only 40 measures long, and there is not an easy way to automatically get my custom text-based melody representations from an audio file. Additionally, melodies must be monophonic (one note at a time) and only use eighth, quarter, and dotted quarter notes to work within this project.