# homework3_stub

May 1, 2025

## 0.1 Homework 3: Symbolic Music Generation Using Markov Chains

**Before starting the homework:**

Please run `pip install miditok` to install the MiDiTok package, which simplifies MIDI file processing by making note and beat extraction more straightforward.

You're also welcome to experiment with other MIDI processing libraries such as mido, pretty_midi and miditoolkit. However, with these libraries, you'll need to handle MIDI quantization yourself, for example, converting note-on/note-off events into beat positions and durations.

[ ]:
```
# run this command to install MiDiTok
# ! pip install miditok
```

```
Collecting miditok
  Downloading miditok-3.0.5.post1-py3-none-any.whl.metadata (10 kB)
Collecting huggingface-hub>=0.16.4 (from miditok)
  Downloading huggingface_hub-0.30.2-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: numpy>=1.19 in /opt/homebrew/lib/python3.10/site-
packages (from miditok) (1.26.3)
Collecting symusic>=0.5.0 (from miditok)
  Downloading symusic-0.5.7-cp310-cp310-macosx_11_0_arm64.whl.metadata (8.7 kB)
Collecting tokenizers>=0.13.0 (from miditok)
  Downloading tokenizers-0.21.1-cp39-abi3-macosx_11_0_arm64.whl.metadata (6.8
kB)
Requirement already satisfied: tqdm in /opt/homebrew/lib/python3.10/site-
packages (from miditok) (4.66.2)
Requirement already satisfied: filelock in /opt/homebrew/lib/python3.10/site-
packages (from huggingface-hub>=0.16.4->miditok) (3.13.1)
Requirement already satisfied: fsspec>=2023.5.0 in
/opt/homebrew/lib/python3.10/site-packages (from huggingface-
hub>=0.16.4->miditok) (2024.2.0)
Requirement already satisfied: packaging>=20.9 in
/Users/ryoandrewonozuka/Library/Python/3.10/lib/python/site-packages (from
huggingface-hub>=0.16.4->miditok) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /opt/homebrew/lib/python3.10/site-
packages (from huggingface-hub>=0.16.4->miditok) (6.0.1)
Requirement already satisfied: requests in /opt/homebrew/lib/python3.10/site-
packages (from huggingface-hub>=0.16.4->miditok) (2.31.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
```

```
/opt/homebrew/lib/python3.10/site-packages (from huggingface-
hub>=0.16.4->miditok) (4.10.0)
Collecting pySmartDL (from symusic>=0.5.0->miditok)
  Downloading pySmartDL-1.3.4-py3-none-any.whl.metadata (2.8 kB)
Requirement already satisfied: platformdirs in
/Users/ryoandrewonozuka/Library/Python/3.10/lib/python/site-packages (from
symusic>=0.5.0->miditok) (4.1.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/homebrew/lib/python3.10/site-packages (from requests->huggingface-
hub>=0.16.4->miditok) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in
/opt/homebrew/lib/python3.10/site-packages (from requests->huggingface-
hub>=0.16.4->miditok) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/opt/homebrew/lib/python3.10/site-packages (from requests->huggingface-
hub>=0.16.4->miditok) (2.2.1)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/homebrew/lib/python3.10/site-packages (from requests->huggingface-
hub>=0.16.4->miditok) (2024.2.2)
Downloading miditok-3.0.5.post1-py3-none-any.whl (158 kB)
Downloading huggingface_hub-0.30.2-py3-none-any.whl (481 kB)
Downloading symusic-0.5.7-cp310-cp310-macosx_11_0_arm64.whl (2.6 MB)
                         2.6/2.6 MB
25.4 MB/s eta 0:00:00
Downloading tokenizers-0.21.1-cp39-abi3-macosx_11_0_arm64.whl (2.7 MB)
                         2.7/2.7 MB
17.1 MB/s eta 0:00:00
Downloading pySmartDL-1.3.4-py3-none-any.whl (20 kB)
Installing collected packages: pySmartDL, symusic, huggingface-hub, tokenizers,
miditok
Successfully installed huggingface-hub-0.30.2 miditok-3.0.5.post1
pySmartDL-1.3.4 symusic-0.5.7 tokenizers-0.21.1

[notice] A new release of pip is
available: 25.0 -> 25.1
[notice] To update, run:
python3.10 -m pip install --upgrade pip
```

[2]:
```python
# import required packages
import random
from glob import glob
from collections import defaultdict

import numpy as np
from numpy.random import choice

from symusic import Score
```

```
from miditok import REMI, TokenizerConfig
from midiutil import MIDIFile
```

[ ]:
```
# You can change the random seed but try to keep your results deterministic!
# If I need to make changes to the autograder it'll require rerunning your code,
# so it should ideally generate the same results each time.
random.seed(42)
```

### 0.1.1 Load music dataset

We will use a subset of the PDMX dataset.

Please find the link in the homework spec.

All pieces are monophonic music (i.e. one melody line) in 4/4 time signature.

[ ]:
```
midi_files = glob('PDMX_subset/*.mid')
len(midi_files)
```

### 0.1.2 Train a tokenizer with the REMI method in MidiTok

[ ]:
```
config = TokenizerConfig(num_velocities=1, use_chords=False, use_programs=False)
tokenizer = REMI(config)
tokenizer.train(vocab_size=1000, files_paths=midi_files)
```

### 0.1.3 Use the trained tokenizer to get tokens for each midi file

In REMI representation, each note will be represented with four tokens: `Position, Pitch, Velocity, Duration`, e.g. `('Position_28', 'Pitch_74', 'Velocity_127', 'Duration_0.4.8')`; a `Bar_None` token indicates the beginning of a new bar.

[ ]:
```
# e.g.:
midi = Score(midi_files[0])
tokens = tokenizer(midi)[0].tokens
tokens[:10]
```

1. Write a function to extract note pitch events from a midi file; and another extract all note pitch events from the dataset and output a dictionary that maps note pitch events to the number of times they occur in the files. (e.g. {60: 120, 61: 58, …}).

`note_extraction()` - **Input**: a midi file

- **Output**: a list of note pitch events (e.g. [60, 62, 61, …])

`note_frequency()` - **Input**: all midi files `midi_files`

- **Output**: a dictionary that maps note pitch events to the number of times they occur, e.g {60: 120, 61: 58, …}

[ ]:
```
def note_extraction(midi_file):
    # Q1a: Your code goes here
    pass
```

```
[ ]: def note_frequency(midi_files):
         # Q1b: Your code goes here
         pass
```

2. Write a function to normalize the above dictionary to produce probability scores (e.g. {60: 0.13, 61: 0.065, …})

`note_unigram_probability()` - **Input**: all midi files `midi_files`

- **Output**: a dictionary that maps note pitch events to probabilities, e.g. {60: 0.13, 61: 0.06, …}

```
[ ]: def note_unigram_probability(midi_files):
         note_counts = note_frequency(midi_files)
         unigramProbabilities = {}

         # Q2: Your code goes here
         # ...

         return unigramProbabilities
```

3. Generate a table of pairwise probabilities containing p(next_note | previous_note) values for the dataset; write a function that randomly generates the next note based on the previous note based on this distribution.

`note_bigram_probability()` - **Input**: all midi files `midi_files`

- **Output**: two dictionaries:
  - `bigramTransitions`: key: previous_note, value: a list of next_note, e.g. {60:[62, 64, ..], 62:[60, 64, ..], …} (i.e., this is a list of every other note that occured after note 60, every note that occured after note 62, etc.)
  - `bigramTransitionProbabilities`: key:previous_note, value: a list of probabilities for next_note in the same order of `bigramTransitions`, e.g. {60:[0.3, 0.4, ..], 62:[0.2, 0.1, ..], …} (i.e., you are converting the values above to probabilities)

`sample_next_note()` - **Input**: a note

- **Output**: next note sampled from pairwise probabilities

```
[ ]: def note_bigram_probability(midi_files):
         bigramTransitions = defaultdict(list)
         bigramTransitionProbabilities = defaultdict(list)

         # Q3a: Your code goes here
         # ...

         return bigramTransitions, bigramTransitionProbabilities
```

```
[ ]: def sample_next_note(note):
         # Q3b: Your code goes here
```

```
    pass
```

4. Write a function to calculate the perplexity of your model on a midi file.

   The perplexity of a model is defined as

   $\exp(-\frac{1}{N}\sum_{i=1}^{N}\log(p(w_i|w_{i-1})))$

   where $p(w_1|w_0) = p(w_1)$, $p(w_i|w_{i-1})(i > 1)$ refers to the pairwise probability p(next_note | previous_note).

`note_bigram_perplexity()` - **Input**: a midi file

- **Output**: perplexity value

```
[ ]: def note_bigram_perplexity(midi_file):
         unigramProbabilities = note_unigram_probability(midi_files)
         bigramTransitions, bigramTransitionProbabilities =␣
      ↪note_bigram_probability(midi_files)

         # Q4: Your code goes here
         # Can use regular numpy.log (i.e., natural logarithm)
```

5. Implement a second-order Markov chain, i.e., one which estimates p(next_note | next_previous_note, previous_note); write a function to compute the perplexity of this new model on a midi file.

   The perplexity of this model is defined as

   $\exp(-\frac{1}{N}\sum_{i=1}^{N}\log(p(w_i|w_{i-2},w_{i-1})))$

   where $p(w_1|w_{-1},w_0) = p(w_1)$, $p(w_2|w_0,w_1) = p(w_2|w_1)$, $p(w_i|w_{i-2},w_{i-1})(i > 2)$ refers to the probability p(next_note | next_previous_note, previous_note).

`note_trigram_probability()` - **Input**: all midi files `midi_files`

- **Output**: two dictionaries:
  - `trigramTransitions`: key - (next_previous_note, previous_note), value - a list of next_note, e.g. {(60, 62):[64, 66, ..], (60, 64):[60, 64, ..], ...}
  - `trigramTransitionProbabilities`: key: (next_previous_note, previous_note), value: a list of probabilities for next_note in the same order of `trigramTransitions`, e.g. {(60, 62):[0.2, 0.2, ..], (60, 64):[0.4, 0.1, ..], ...}

`note_trigram_perplexity()` - **Input**: a midi file

- **Output**: perplexity value

```
[ ]: def note_trigram_probability(midi_files):
         trigramTransitions = defaultdict(list)
         trigramTransitionProbabilities = defaultdict(list)

         # Q5a: Your code goes here
         # ...
```

```
    return trigramTransitions, trigramTransitionProbabilities
```

[ ]:
```
def note_trigram_perplexity(midi_file):
    unigramProbabilities = note_unigram_probability(midi_files)
    bigramTransitions, bigramTransitionProbabilities =␣
 ↪note_bigram_probability(midi_files)
    trigramTransitions, trigramTransitionProbabilities =␣
 ↪note_trigram_probability(midi_files)

    # Q5b: Your code goes here
```

6. Our model currently doesn't have any knowledge of beats. Write a function that extracts beat lengths and outputs a list of [(beat position; beat length)] values.

   Recall that each note will be encoded as `Position, Pitch, Velocity, Duration` using REMI. Please keep the `Position` value for beat position, and convert `Duration` to beat length using provided lookup table `duration2length` (see below).

   For example, for a note represented by four tokens `('Position_24', 'Pitch_72', 'Velocity_127', 'Duration_0.4.8')`, the extracted (beat position; beat length) value is `(24, 4)`.

   As a result, we will obtain a list like [(0,8),(8,16),(24,4),(28,4),(0,4)...], where the next beat position is the previous beat position + the beat length. As we divide each bar into 32 positions by default, when reaching the end of a bar (i.e. 28 + 4 = 32 in the case of (28, 4)), the beat position reset to 0.

[ ]:
```
duration2length = {
    '0.2.8': 2,  # sixteenth note, 0.25 beat in 4/4 time signature
    '0.4.8': 4,  # eighth note, 0.5 beat in 4/4 time signature
    '1.0.8': 8,  # quarter note, 1 beat in 4/4 time signature
    '2.0.8': 16, # half note, 2 beats in 4/4 time signature
    '4.0.4': 32, # whole note, 4 beats in 4/4 time signature
}
```

`beat_extraction()` - **Input**: a midi file

- **Output**: a list of (beat position; beat length) values

[ ]:
```
def beat_extraction(midi_file):
    # Q6: Your code goes here
    pass
```

7. Implement a Markov chain that computes p(beat_length | previous_beat_length) based on the above function.

`beat_bigram_probability()` - **Input**: all midi files `midi_files`

- **Output**: two dictionaries:

- `bigramBeatTransitions`: key: previous_beat_length, value: a list of beat_length, e.g. {4:[8, 2, ..], 8:[8, 4, ..], ...}
- `bigramBeatTransitionProbabilities`: key - previous_beat_length, value - a list of probabilities for beat_length in the same order of `bigramBeatTransitions`, e.g. {4:[0.3, 0.2, ..], 8:[0.4, 0.4, ..], ...}

[ ]:
```
def beat_bigram_probability(midi_files):
    bigramBeatTransitions = defaultdict(list)
    bigramBeatTransitionProbabilities = defaultdict(list)

    # Q7: Your code goes here
    # ...

    return bigramBeatTransitions, bigramBeatTransitionProbabilities
```

8. Implement a function to compute p(beat length | beat position), and compute the perplexity of your models from Q7 and Q8. For both models, we only consider the probabilities of predicting the sequence of **beat lengths**.

`beat_pos_bigram_probability()` - **Input**: all midi files `midi_files`

- **Output**: two dictionaries:
  - `bigramBeatPosTransitions`: key - beat_position, value - a list of beat_length
  - `bigramBeatPosTransitionProbabilities`: key - beat_position, value - a list of probabilities for beat_length in the same order of `bigramBeatPosTransitions`

`beat_bigram_perplexity()` - **Input**: a midi file

- **Output**: two perplexity values correspond to the models in Q7 and Q8, respectively

[ ]:
```
def beat_pos_bigram_probability(midi_files):
    bigramBeatPosTransitions = defaultdict(list)
    bigramBeatPosTransitionProbabilities = defaultdict(list)

    # Q8a: Your code goes here
    # ...

    return bigramBeatPosTransitions, bigramBeatPosTransitionProbabilities
```

[ ]:
```
def beat_bigram_perplexity(midi_file):
    bigramBeatTransitions, bigramBeatTransitionProbabilities = 
 ↪beat_bigram_probability(midi_files)
    bigramBeatPosTransitions, bigramBeatPosTransitionProbabilities = 
 ↪beat_pos_bigram_probability(midi_files)
    # Q8b: Your code goes here
    # Hint: one more probability function needs to be computed

    # perplexity for Q7
```

```
    perplexity_Q7 = None

    # perplexity for Q8
    perplexity_Q8 = None

    return perplexity_Q7, perplexity_Q8
```

9. Implement a Markov chain that computes p(beat_length | previous_beat_length, beat_position), and report its perplexity.

`beat_trigram_probability()` - **Input**: all midi files `midi_files`

- **Output**: two dictionaries:
  - `trigramBeatTransitions`: key: (previous_beat_length, beat_position), value: a list of beat_length
  - `trigramBeatTransitionProbabilities`: key: (previous_beat_length, beat_position), value: a list of probabilities for beat_length in the same order of `trigramBeatTransitions`

`beat_trigram_perplexity()` - **Input**: a midi file

- **Output**: perplexity value

[ ]:
```
def beat_trigram_probability(midi_files):
    trigramBeatTransitions = defaultdict(list)
    trigramBeatTransitionProbabilities = defaultdict(list)

    # Q9a: Your code goes here
    # ...

    return trigramBeatTransitions, trigramBeatTransitionProbabilities
```

[ ]:
```
def beat_trigram_perplexity(midi_file):
    bigramBeatPosTransitions, bigramBeatPosTransitionProbabilities =␣
 ↪beat_pos_bigram_probability(midi_files)
    trigramBeatTransitions, trigramBeatTransitionProbabilities =␣
 ↪beat_trigram_probability(midi_files)
    # Q9b: Your code goes here
```

10. Use the model from Q5 to generate N notes, and the model from Q8 to generate beat lengths for each note. Save the generated music as a midi file (see code from workbook1) as q10.mid. Remember to reset the beat position to 0 when reaching the end of a bar.

`music_generate` - **Input**: target length, e.g. 500

- **Output**: a midi file q10.mid

Note: the duration of one beat in MIDIUtil is 1, while in MidiTok is 8. Divide beat length by 8 if you use methods in MIDIUtil to save midi files.

[ ]:
```
def music_generate(length):
    # sample notes
    unigramProbabilities = note_unigram_probability(midi_files)
    bigramTransitions, bigramTransitionProbabilities = 
 ↪note_bigram_probability(midi_files)
    trigramTransitions, trigramTransitionProbabilities = 
 ↪note_trigram_probability(midi_files)

    # Q10: Your code goes here ...
    sampled_notes = []

    # sample beats
    sampled_beats = []

    # save the generated music as a midi file
```