homework1_stub

April 14, 2025

1 Homework 1: Sine wave generation and binary classification

1.1 Part A - Sine Wave Generation

1.1.1 Setup

To complete this part, install the required Python libraries:

[1]:
```
import numpy as np
from scipy.io import wavfile

import numpy as np
import glob
from mido import MidiFile
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
import math

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

[2]:
```
# (installation process may be different on your system)
# You don't need to use these libraries, so long as you implement the specified
 ↪functions
# !pip install numpy
# !pip install scipy
# !pip install IPython
# !pip install glob
# !pip install scikit-learn
# !pip install mido
```

1. Write a function that converts a musical note name to its corresponding frequency in Hertz (Hz)

`note_name_to_frequency()` - **Input**: A string `note_name` combining a note (e.g., `'C'`, `'C#'`, `'D'`, `'D#'`, `'E'`, `'F'`, `'F#'`, `'G'`, `'G#'`, `'A'`, `'A#'`, `'B'`) and an octave number (`'0'` to `'10'`) - **Output**: A float representing the frequency in Hz - **Details**: - Use A4 = 440 Hz as the reference frequency - Frequencies double with each octave increase (e.g., A5 = 880 Hz) and halve with each decrease (e.g., A3 = 220 Hz)

- **Examples**:
  - 'A4' → 440.0
  - 'A3' → 220.0
  - 'G#4' → 415.3047

[3]:
```
SAMPLE_RATE = 44100

def note_name_to_frequency(note_name):
    # Q1: Your code goes here
    return frequency
```

2. Write a function that linearly decreases the amplitude of a given waveform

`decrease_amplitude()` - **Inputs**: - `audio`: A NumPy array representing the audio waveform at a sample rate of 44100 Hz - **Output**: A NumPy array representing the audio waveform at a sample rate of 44100 Hz - **Details**: - The function must linearly decrease the amplitude of the input audio. The amplitude should start at 1 (full volume) and decrease gradually to 0 (silence) by the end of the sample

[4]:
```
def decrease_amplitude(audio):
    # Q2: Your code goes here
    pass
```

3. Write a function that adds a delay effect to a given audio where the output is a combination of the original audio and a delayed audio

`add_delay_effects()`
- **Inputs**:
- `audio`: A NumPy array representing the audio waveform, sampled at 44,100 Hz - **Output**:
- A NumPy array representing the modified audio waveform, sampled at 44,100 Hz - **Details**: - The amplitude of the delayed audio should be 30% of the original audio's amplitude - The amplitude of the original audio should be adjusted to 70% of the original audio's amplitude - The output should combine the original audio (with the adjusted amplitude) with a delayed version of itself - The delayed audio should be offset by 0.5 seconds behind the original audio

- **Examples**:
  - The provided files (input.wav and output.wav) provide examples of input and output audio

[5]:
```
# Can use these for visualization if you like, though the autograder won't use␣
↪ipython
#
# from IPython.display import Audio, display
#
# print("Example Input Audio:")
# display(Audio(filename = "input.wav", rate=44100))
#
# print("Example Output Audio:")
# display(Audio(filename = "output.wav", rate=44100))
```

[6]:
```python
def add_delay_effects(audio):
    #Q3: Your code goes here
    return delayed_audio
```

4. Write a function that concatenates a list of audio arrays sequentially and a function that mixes audio arrays by scaling and summing them, simulating simultaneous playback

`concatenate_audio()` - **Input**: - `list_of_your_audio`: A list of NumPy arrays (e.g., `[audio1, audio2]`), each representing audio at 44100 Hz - **Output**: A NumPy array of the concatenated audio - **Example**: - If `audio1` is 2 seconds (88200 samples) and `audio2` is 1 second (44100 samples), the output is 3 seconds (132300 samples)

`mix_audio()` - **Inputs**: - `list_of_your_audio`: A list of NumPy arrays (e.g., `[audio1, audio2]`), all with the same length at 44100 Hz. - `amplitudes`: A list of floats (e.g., `[0.2, 0.8]`) matching the length of `list_of_your_audio` - **Output**: A NumPy array representing the mixed audio - **Example**: - If `audio1` and `audio2` are 2 seconds long, and `amplitudes = [0.2, 0.8]`, the output is `0.2 * audio1 + 0.8 * audio2`

[7]:
```python
def concatenate_audio(list_of_your_audio):
    #Q4: Your code goes here
    pass
```

[8]:
```python
def mix_audio(list_of_your_audio, amplitudes):
    #Q4: Your code goes here
    pass
```

5. Modify your solution to Q2 so that your pipeline can generate sawtooth waves by adding harmonics based on the following equation:

   $\text{sawtooth}(f,t) = \frac{2}{\pi}\sum_{k=1}^{19} \frac{(-1)^{k+1}}{k}\sin(2\pi k f t)$

- **Inputs**:
  - `frequency`: Fundamental frequency of sawtooth wave
  - `duration`: A float representing the duration in seconds (e.g., 2.0)
- **Output**: A NumPy array representing the audio waveform at a sample rate of 44100 Hz

[9]:
```python
def create_sawtooth_wave(frequency, duration, sample_rate=44100):
    #Q5: Your code goes here
    return wave
```

1.2 Part B - Binary Classification

Train a binary classification model using `scikit-learn` to distinguish between piano and drum MIDI files.

Unzip MIDI Files Extract the provided MIDI datasets:

```
unzip piano.zip
unzip drums.zip
```

- `./piano`: Contains piano MIDI files (e.g., `0000.mid` to `2154.mid`)

- `./drums`: Contains drum MIDI files (e.g., `0000.mid` to `2154.mid`)
- Source: [Tegridy MIDI Dataset] (https://github.com/asigalov61/Tegridy-MIDI-Dataset)

These folders should be extracted into the same directory as your solution file

[ ]:

```

```

6. Write functions to compute simple statistics about the files

`get_stats()`

- **Inputs**:
  - `piano_file_paths`: List of piano MIDI file paths‘
  - `drum_file_paths`: List of drum MIDI file paths‘
- **Output**: A dictionary:
  - `"piano_midi_num"`: Integer, number of piano files
  - `"drum_midi_num"`: Integer, number of drum files
  - `"average_piano_beat_num"`: Float, average number of beats in piano files
  - `"average_drum_beat_num"`: Float, average number of beats in drum files
- **Details**:
  - For each file:
    * Load with `MidiFile(file_path)`
    * Get `ticks_per_beat` from `mid.ticks_per_beat`
    * Compute total ticks as the maximum cumulative `msg.time` (delta time) across tracks
    * Number of beats = (total ticks / ticks_per_beat)
  - Compute averages, handling empty lists (return 0 if no files)

[10]:

```python
def get_file_lists():
    piano_files = sorted(glob.glob("./piano/*.mid"))
    drum_files = sorted(glob.glob("./drums/*.mid"))
    return piano_files, drum_files

def get_num_beats(file_path):
    # Q6: Your code goes here
    mid = MidiFile(file_path)
    # Might need: mid.tracks, msg.time, mid.ticks_per_beat
    return nBeats

def get_stats(piano_path_list, drum_path_list):
    piano_beat_nums = []
    drum_beat_nums = []
    for file_path in piano_path_list:
        piano_beat_nums.append(get_num_beats(file_path))

    for file_path in drum_path_list:
        drum_beat_nums.append(get_num_beats(file_path))

    return {"piano_midi_num":len(piano_path_list),
            "drum_midi_num":len(drum_path_list),
```

```
            "average_piano_beat_num":np.average(piano_beat_nums),
            "average_drum_beat_num":np.average(drum_beat_nums)}
```

7. Implement a few simple feature functions, to compute the lowest and highest MIDI note numbers in a file, and the set of unique notes in a file

`get_lowest_pitch()` and `get_highest_pitch()` functions to find the lowest and highest MIDI note numbers in a file

- **Input**: `file_path`, a string (e.g., `"./piano/0000.mid"`)
- **Output**: An integer (0–127) or `None` if no notes exist
- **Details**:
  - Use `MidiFile(file_path)` and scan all tracks
  - Check `msg.type == 'note_on'` and `msg.velocity > 0` for active notes
  - Return the minimum (`get_lowest_pitch`) or maximum (`get_highest_pitch`) `msg.note`

`get_unique_pitch_num()` a function to count unique MIDI note numbers in a file

- **Input**: `file_path`, a string
- **Output**: An integer, the number of unique pitches
- **Details**:
  - Collect `msg.note` from all `'note_on'` events with `msg.velocity > 0` into a set
  - Return the set's length
- **Example**: For notes `["C4", "C4", "G4", "G4", "A4", "A4", "G4"]`, output is 3 (unique: C4, G4, A4)

[11]:
```
def get_lowest_pitch(file_path):
    #Q7-1: Your code goes here
    pass

def get_highest_pitch(file_path):
    #Q7-2: Your code goes here
    pass

def get_unique_pitch_num(file_path):
    #Q7-3: Your code goes here
    pass
```

8. Implement an additional feature extraction function to compute the average MIDI note number in a file

`get_average_pitch_value()` a function to return the average MIDI note number from a file

- **Input**: `file_path`, a string
- **Output**: A float, the average value of MIDI notes in the file
- **Details**:
  - Collect `msg.note` from all `'note_on'` events with `msg.velocity > 0` into a set
- **Example**: For notes `[51, 52, 53]`, output is `52`

[12]:
```python
def get_average_pitch_value(file_path):
    #Q8: Your code goes here
    pass
```

9. Construct your dataset and split it into train and test sets using `scikit-learn` (most of this code is provided). Train your model to classify whether a given file is intended for piano or drums.

`featureQ9()`

Returns a feature vector concatenating the four features described above

- **Input**: `file_path`, a string.
- **Output**: A vector of four features

[13]:
```python
def featureQ9(file_path):
    # Already implemented: this one is a freebie if you got everything above␣
↪correct!
    return [get_lowest_pitch(file_path),
            get_highest_pitch(file_path),
            get_unique_pitch_num(file_path),
            get_average_pitch_value(file_path)]
```

10. Creatively incorporate additional features into your classifier to make your classification more accurate. Include comments describing your solution.

[14]:
```python
def featureQ10(file_path):
    #Q10: Your code goes here
    return []
```

[ ]:
```python

```