```
import collections
import datetime
import glob
from typing import Optional
import pathlib
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
import fluidsynth
import pretty_midi
from IPython import display
import tensorflow as tf
```

✓ 0.0s

These are modules already built into python.

Data visualization libraries.

These are the relevant open-source libraries needed for this project. You'll need to download them.

```python
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)

# Sampling rate for audio playback
sample_rate = 16000

# Downloads the maestro dataset, containing around 1200 midi files.
data_dir = pathlib.Path('data/maestro-v2_extracted/maestro-v2.0.0')
if not data_dir.exists():

  tf.keras.utils.get_file(
      'maestro-v2.0.0-midi.zip',
      origin='https://storage.googleapis.com/magentadata/datasets/maestro/v2.0.0/maestro-v2.0.0-midi.zip',
      extract=True,
      cache_dir='.', cache_subdir='data',
  )

  filenames = glob.glob(str(data_dir/'**/*.midi'))
  #print('Number of files:', len(filenames))
# At this stage, output should be "Number of files: 1282"
```

✓ 0.0s

Ensures that any random operations performed by tensorflow or numpy are consistent every time we run the code.

16,000 Hz -> common rate for speech and some music datasets

Creates a path object for the dataset's directory.

If the data_dir doesn't exist, downloads the MAESTRO dataset as a zip file and extracts it.

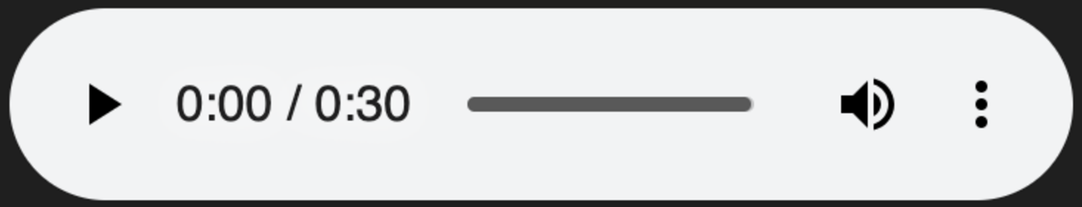
Finds all the midi files within the datset. Use the line below to ensure everything has run smoothly so far.

```python
# Practice using pretty_midi to parse a single MIDI file and inspect format of the notes
sample_file = filenames[1000]

pm = pretty_midi.PrettyMIDI(sample_file)

def display_audio(pm: pretty_midi.PrettyMIDI, seconds=30):
    waveform = pm.fluidsynth(fs = sample_rate)

    waveform_short = waveform[ :seconds * sample_rate]
    return display.Audio(waveform_short, rate = sample_rate)

display_audio(pm)
```

Takes a random file (in our case, the 1001st file) and creates an object (pm) which holds all info about MIDI data, including tracks, instruments, notes, timing, etc.

Converts the MIDI data into an actual digital sound file. We specify only 30 seconds of audio.

✓ 2.0s



```python
# Inspecting the instruments used in the file
print("Number of instruments:", len(pm.instruments))
instrument = pm.instruments[0]
instrument_name = pretty_midi.program_to_instrument_name(instrument.program)
print("Instrument name:", instrument_name)
```

✓ 0.0s

Finds the unique # of instrument tracks in a MIDI file.

MIDI instruments are defined by an integer 0-127. This code converts the # (instrument.program) to a readable instrument name.

```
Number of instruments: 1
Instrument name: Acoustic Grand Piano
```

Here we'll take a look at the first 10 notes of our song. Enumerate ensures the index # stays with each note's info.

```python
# Extract the notes from songs
for i, note in enumerate(instrument.notes[:10]):
    note_name = pretty_midi.note_number_to_name(note.pitch)
    duration = note.end - note.start
    print(f"{i}: pitch = {note.pitch}, note_name = {note_name},"
          f' duration = {duration:.4f}')
# For the first 10 notes in the song, output is the pitch, note name, and duration of each note
# Pitch is perceptual quality of sound as a MIDI note number
```

✓ 0.0s

Like program_to_instrument_name, this function converts the MIDI integer for pitch to a human readable name.

```
0: pitch = 49, note_name = C#3, duration = 0.6198
1: pitch = 77, note_name = F5, duration = 1.0130
2: pitch = 73, note_name = C#5, duration = 0.3438
3: pitch = 56, note_name = G#3, duration = 0.6341
4: pitch = 56, note_name = G#3, duration = 0.0755
5: pitch = 56, note_name = G#3, duration = 0.0872
6: pitch = 65, note_name = F4, duration = 1.2135
7: pitch = 61, note_name = C#4, duration = 1.2057
8: pitch = 68, note_name = G#4, duration = 1.5430
9: pitch = 56, note_name = G#3, duration = 0.4180
```

```python
#Extract notes from a MIDI file
def midi_to_notes(midi_file: str):
    pm = pretty_midi.PrettyMIDI(midi_file)
    instrument = pm.instruments[0]
    notes = collections.defaultdict(list)

    #Sort the notes by start time
    sorted_notes = sorted(instrument.notes, key=lambda note: note.start)
    prev_start = sorted_notes[0].start

    for note in sorted_notes:
        start = note.start
        end = note.end
        notes["pitch"].append(note.pitch)
        notes["start"].append(note.start)
        notes["end"].append(end)
        notes["step"].append(start - prev_start)
        notes["duration"].append(end - start)
        prev_start = start

    return pd.DataFrame({name: np.array(value) for name, value in notes.items()})
```

Notes gives us a default ditionary where each key has an empty list, allowing us to append values.

Sorts our notes based on start time. Prev_start is initialized to the start time of the first note. We'll later use this to calculate "step".

Now, each note in our song has a unique value for pitch, duration, and step.

```
raw_notes = midi_to_notes(sample_file)
raw_notes
```

✓ 0.1s

	pitch	start	end	step	duration
0	77	0.976562	1.989583	0.000000	1.013021
1	49	1.006510	1.626302	0.029948	0.619792
2	56	1.636719	2.270833	0.630208	0.634115
3	73	1.916667	2.260417	0.279948	0.343750
4	68	2.140625	3.683594	0.223958	1.542969
...	...	...	...	...	...
1496	61	273.015625	279.102865	0.015625	6.087240
1497	37	273.016927	278.799479	0.001302	5.782552
1498	56	273.022135	279.074219	0.005208	6.052083
1499	38	282.093750	282.264323	9.071615	0.170573
1500	45	282.264323	282.278646	0.170573	0.014323

1501 rows × 5 columns

This is a dataframe of the pitch, step, and duration of every note in the MIDI file!

```
get_note_names = np.vectorize(pretty_midi.note_number_to_name)
sample_note_names = get_note_names(raw_notes['pitch'])
raw_notes = raw_notes.assign(pitch = sample_note_names)
```

This function converts pitch # to a readable name.

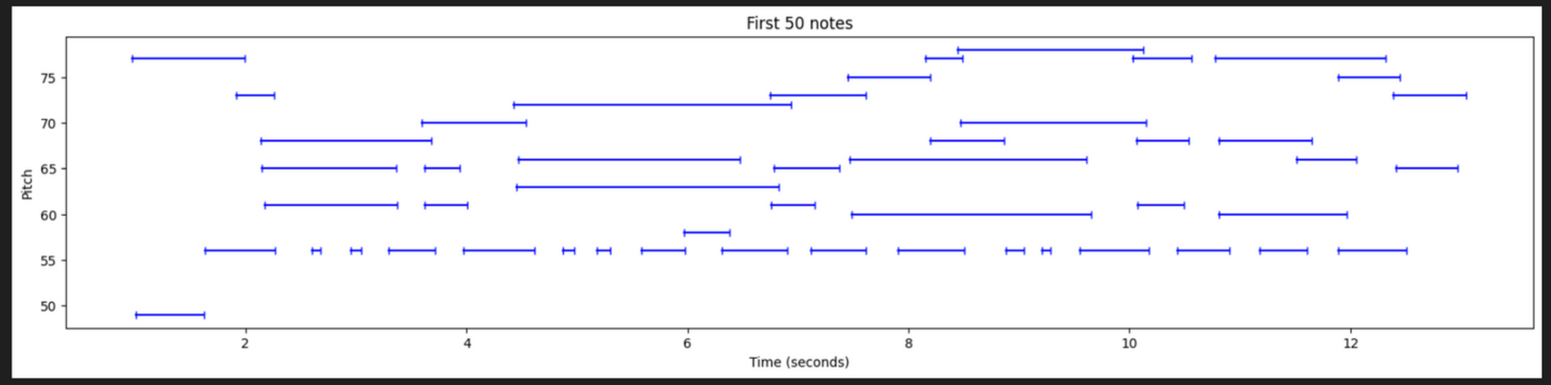
```python
def plot_piano_roll(notes: pd.DataFrame, count: Optional[int] = None):
  if count:
    title = f'First {count} notes'
  else:
    title = f'Whole track'
    count = len(notes['pitch'])
  plt.figure(figsize=(20, 4))
  plot_pitch = np.stack([notes['pitch'], notes['pitch']], axis=0)
  plot_start_stop = np.stack([notes['start'], notes['end']], axis=0)
  plt.plot(
      plot_start_stop[:, :count], plot_pitch[:, :count], color="blue", marker="|")
  plt.xlabel('Time (seconds)')
  plt.ylabel('Pitch')
  _ = plt.title(title)
```

If the optional count is provided, then only that # of notes will be displayed.

Stacks the columns twice along axis 0, creating a 2d array. The first copy is for the start time, the second is for the end time.

```
plot_piano_roll(raw_notes, count=50)
```

✓ 0.2s Python



Here's an example of the first 50 notes, plotted with respect to duration and pitch!

```python
#check distribution of each note variable
def plot_distributions(notes: pd.DataFrame, drop_percentile=2.5):
  plt.figure(figsize=[15, 5])
  plt.subplot(1, 3, 1)
  sns.histplot(notes, x="pitch", bins=20)

  plt.subplot(1, 3, 2)
  max_step = np.percentile(notes['step'], 100 - drop_percentile)
  sns.histplot(notes, x="step", bins=np.linspace(0, max_step, 21))

  plt.subplot(1, 3, 3)
  max_duration = np.percentile(notes['duration'], 100 - drop_percentile)
  sns.histplot(notes, x="duration", bins=np.linspace(0, max_duration, 21))
```

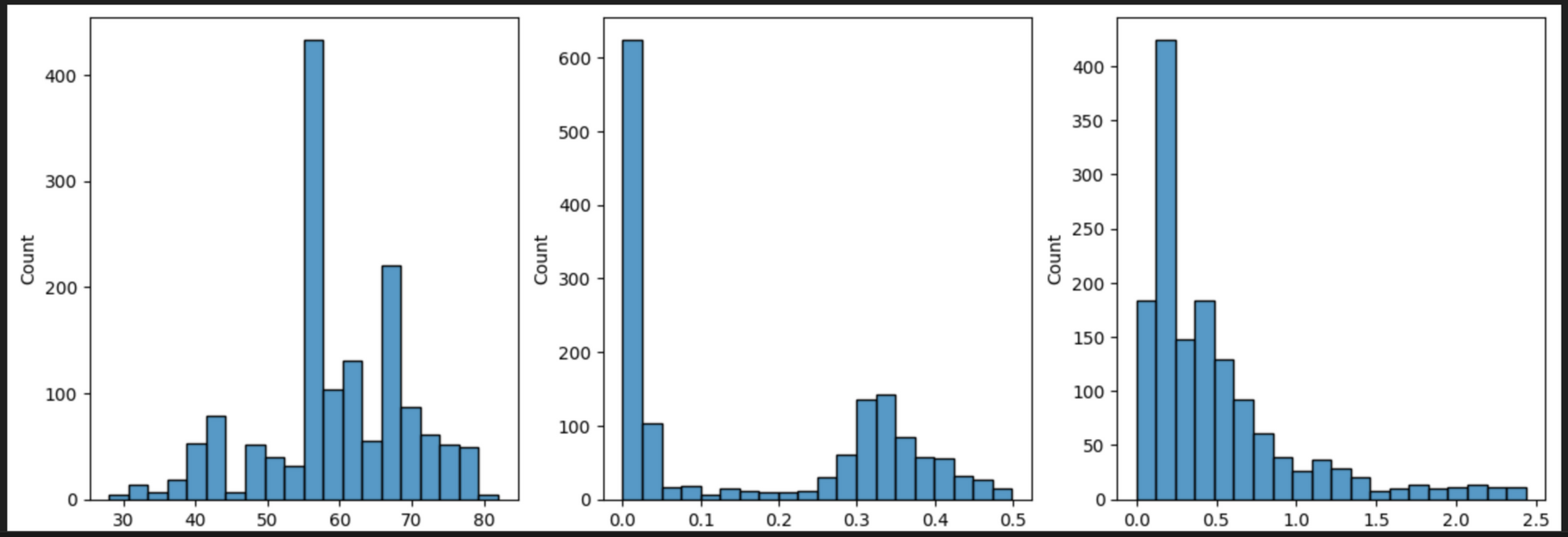
✓ 0.0s

Drop_percentile is used to drop outliers

Creating three histograms, at the 97.5th percentile, for pitch, step, and duration of our chosen song sample

```python
plot_distributions(raw_notes)
```

✓ 0.7s



```python
#create your own MIDI file
def notes_to_midi(
    notes: pd.DataFrame, out_file: str, instrument_name: str, velocity: int=100
    ):

    pm = pretty_midi.PrettyMIDI()
    instrument = pretty_midi.Instrument(
        program = pretty_midi.instrument_name_to_program(instrument_name))

    prev_start = 0
    for i, note in notes.iterrows():
        start = float(prev_start + note["step"])
        end = float(start + note["duration"])
        note = pretty_midi.Note(
            velocity = velocity, pitch = int(note["pitch"]), start = start, end = end
        )
        instrument.notes.append(note)
        prev_start = start

    pm.instruments.append(instrument)
    pm.write(out_file)
    return pm
```

Creates an instrument object that will be added to the MIDI file

.iterrows iterates through each row of the notes dataframe.

The created note ia appended to the notes list of the instrument object

Now that all notes are added, instrument is appended to the pm.instruments list

```
example = "example.midi"
example_pm = notes_to_midi(raw_notes, out_file = example, instrument_name = instrument_name)
```

✓ 0.1s

```
#play the generated MIDI file
display_audio(example_pm)
```

✓ 1.6s


```
#creating the training dataset
#extract notes from the MIDI files, starting with a small number of files
num_files = 5
all_notes = []
for f in filenames[:num_files]:
    notes = midi_to_notes(f)
    all_notes.append(notes)

all_notes = pd.concat(all_notes)
```

extracts data from a set of MIDI files and creates a dataset of notes to be used for training our model

Slices the filenames list to only include the first 5 MIDI file paths based on num_files. The notes for each file are appended to all_notes