# Music Preprocessing

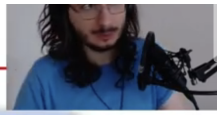
- KRN is another format for storing music but it is not widely used, unlike MIDI. It is just a form of music encoding.
- Humdrum refers to a system for representing and analyzing music in a symbolic format.

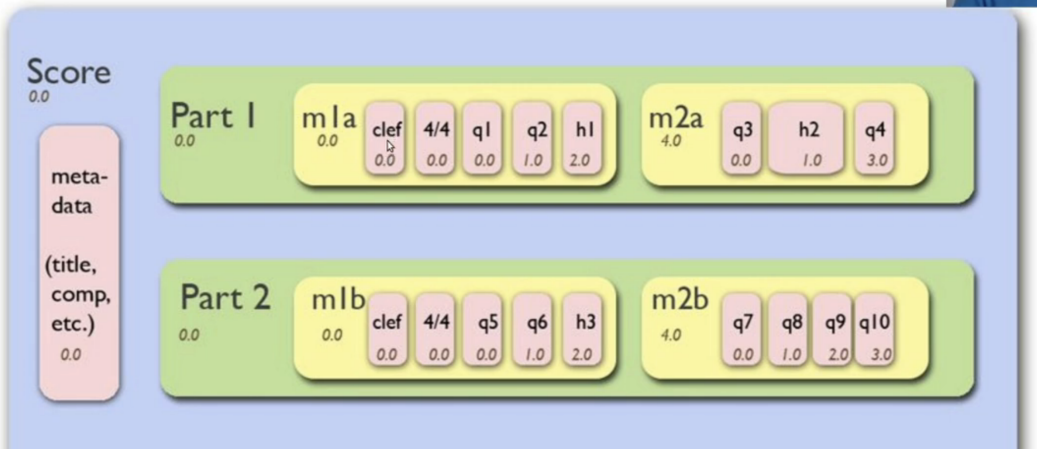
```
def preprocess(dataset_path):
    pass

    # load the folk songs

    # filter out songs that have non-acceptable durations

    # transpose songs to Cmaj/Amin

    # encode songs with music time series representation

    # save songs to text file
```

- By filtering out songs that have non-acceptable duration we mean to eliminate songs which have irregular time series in the number of beats. Like 3 bets, 5 beats, 6 beats per bar. We want beats to be consistent in a bar so that our model can learn it. music21 library is used for converting symbolic music data, representing it and manipulating it.

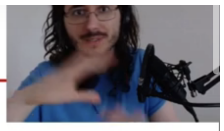
```
# kern, MIDI, MusicXML -> m21 -> kern, MIDI,...
```

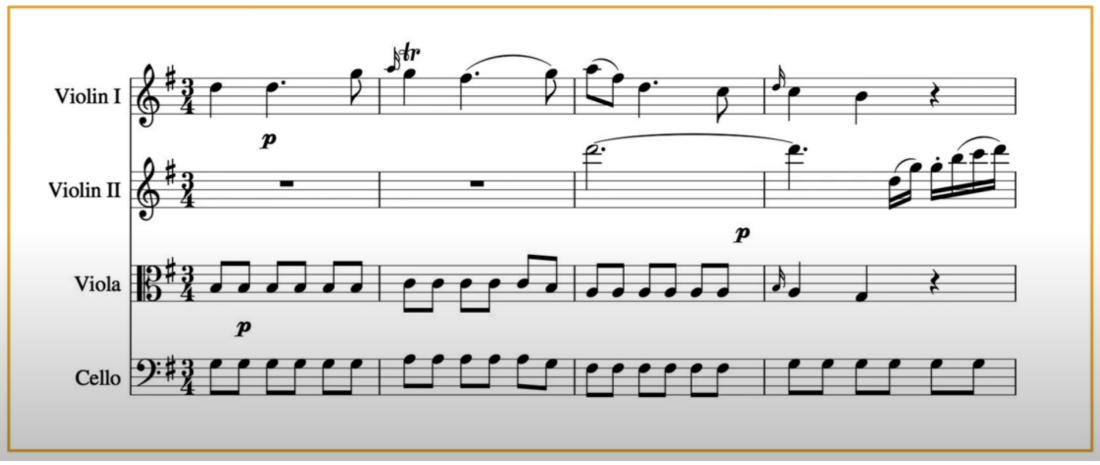
## Music21 score



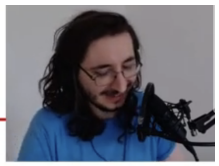


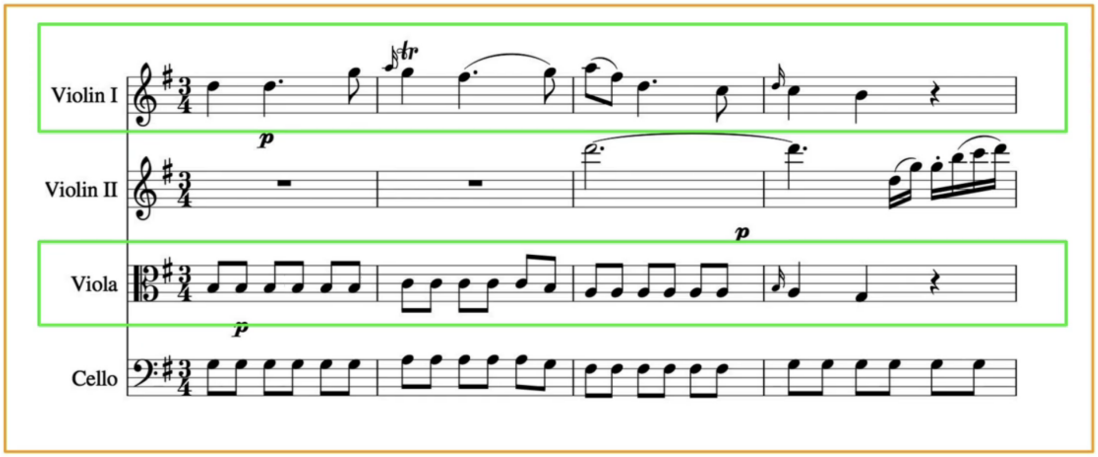

## Score



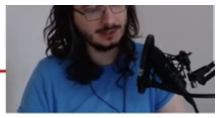


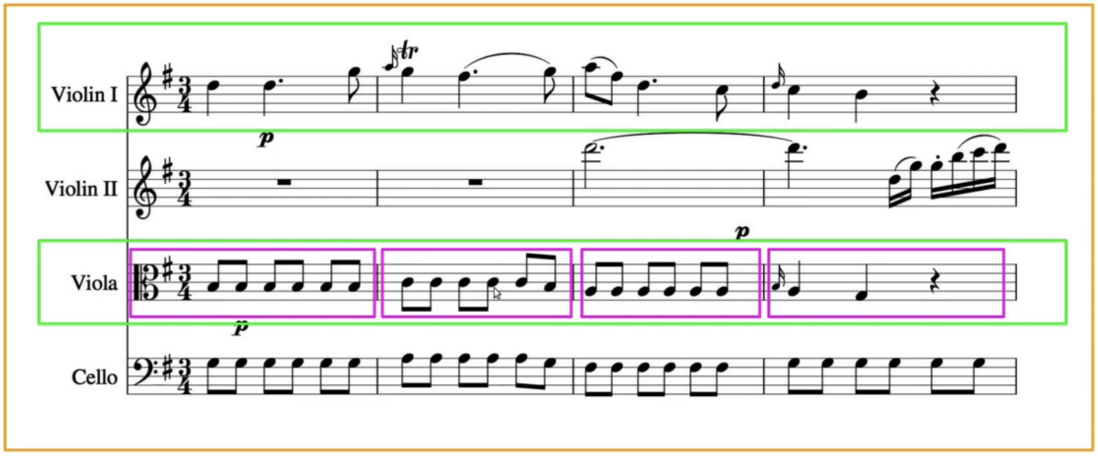
## Score + Parts





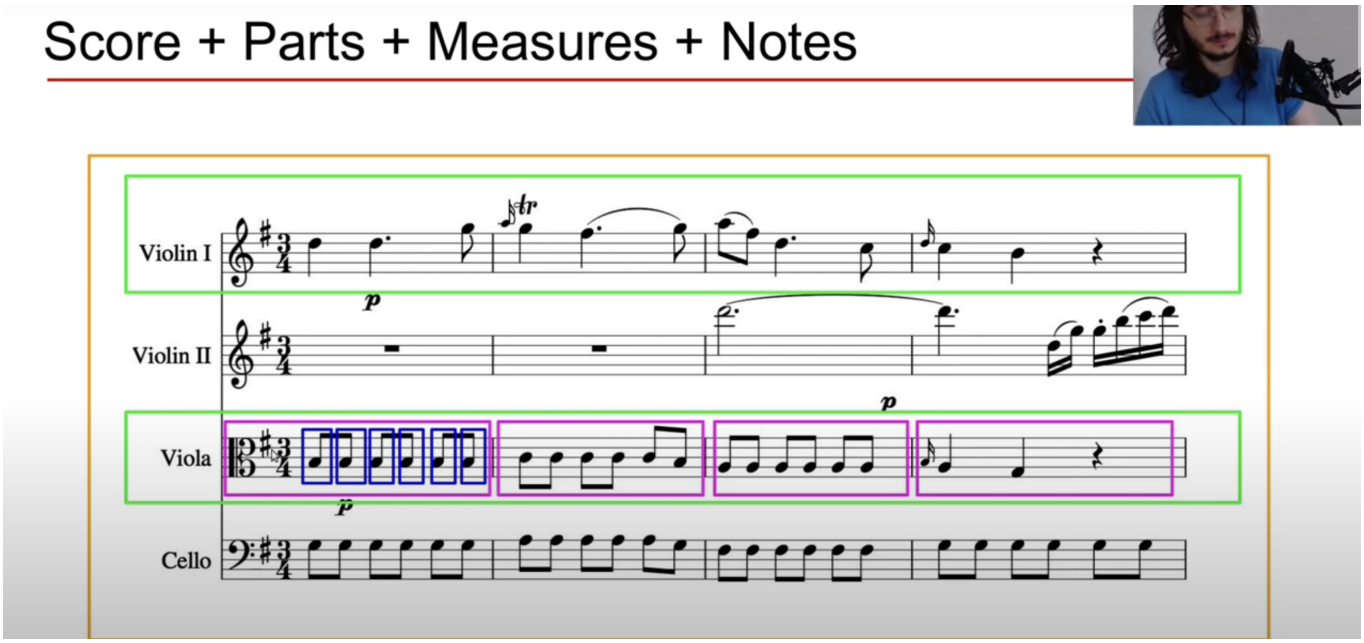

## Score + Parts + Measures






Here, measures are equal to bars.

Inside the measures, we have various notes.




# Loading Songs

```
def load_songs_in_kern(dataset_path):

    songs = []

    # go through all the files in dataset and load them with music21
    for path, subdirs, files in os.walk(dataset_path):
        for file in files:
            if file[-3:] == "krn":
                song = m21.converter.parse(os.path.join(path, file))
                songs.append(song)
    return songs
```

The function m21.converter.parse returns a music21 object which is of the type of score and contains every notes,

keys, time series etc.

```
if __name__ == "__main__":

    songs = load_songs_in_kern(KERN_DATASET_PATH)
    print(f"Loaded {len(songs)} songs.")
    song =songs[0]
    song.show()
```

# Filtering

- Let's make a boolean function that returns whether the song is acceptable or not.

```
ACCEPTABLE_DURATIONS = [
    0.25,
    0.5,
    0.75,
    1.0,
    1.5,
    2,
    3,
    4
]
```

- This is the representation of musical notes in quarter-length format. This is represented as 1/(no of beats); therefore 1/4,1/2,1/1 and so on.
- Some extra values are dotted note values.

```
def has_acceptable_durations(song, acceptable_durations):
    for note in song.flat.notesAndRests:
        if note.duration.quarterLength not in acceptable_durations:
            return False
    return True
```

- dotted notes is the same as a simple note but the only difference is that there is an addition of half of its value in its length.

# Transposing

```
def transpose(song):

    # get key from the song
    parts = song.getElementsByClass(m21.stream.Part)
    measures_part0 = parts[0].getElementsByClass(m21.stream.Measure)
    key = measures_part0[0][4]

    # estimate key using music21
    if not isinstance(key, m21.key.Key):
        key = song.analyze("key")

    # get interval for transposition. E.g., Bmaj -> Cmaj
    if key.mode == "major":
        interval = m21.interval.Interval(key.tonic, m21.pitch.Pitch("C"))
    elif key.mode == "minor":
        interval = m21.interval.Interval(key.tonic, m21.pitch.Pitch("A"))

    # transpose song by calculated interval
    tranposed_song = song.tranpose(interval)

    return tranposed_song
```

We transpose songs so that all songs follow one key only.

In the case of major mode, we transpose to C major and in the case of minor mode we transpose to A minor.

- The key is present in part 1 of the music21 score and specifically on the 4th index. And if the key is not present there then we estimate it using music21.
- Then, we find the interval between the current key and our aim key. The interval could be imagined as some length to shift the notes.
- Then, we transpose it.

## Why transpose?

- We have less data so the model won't be able to learn all the 24 keys.
- So we generalized our model to learn only the C major and A minor key.

```
def preprocess(dataset_path):
    pass

    # load the folk songs
    print("Loading songs...")
    songs = load_songs_in_kern(dataset_path)
    print(f"Loaded {len(songs)} songs.")

    for song in songs:

        # filter out songs that have non-acceptable durations
        if not has_acceptable_durations(song, ACCEPTABLE_DURATIONS):
            continue

        # transpose songs to Cmaj/Amin
        song = transpose(song)

        # encode songs with music time series representation

        # save songs to text file
```

- **Encoding Songs to Time Series Format**

```
def encode_song(song, time_step=0.25):
    # p = 60, d = 1.0 -> [60, "_", "_", "_"]

    encoded_song = []

    for event in song.flat.notesAndRests:

        # handle notes
        if isinstance(event, m21.note.Note):
            symbol = event.pitch.midi # 60
        # handle rests
        elif isinstance(event, m21.note.Rest):
            symbol = "r"

        # convert the note/rest into time series notation
        steps = int(event.duration.quarterLength / time_step)
        for step in range(steps):
            if step == 0:
                encoded_song.append(symbol)
            else:
                encoded_song.append("_")

    # cast encoded song to a str
```

```
    # cast encoded song to a str
    encoded_song = " ".join(map(str, encoded_song))

    return  encoded_song
```

# Save Song

```
# save songs to text file
save_path = os.path.join(SAVE_DIR, str(i))
with open(save_path, "w") as fp:
    fp.write(encoded_song)
```

# Making one file

- In the LSTMs, we want to pass a single dataset so we will combine all these songs with delimiters between them so that we can pass only one file to our LSTMs model as an input.

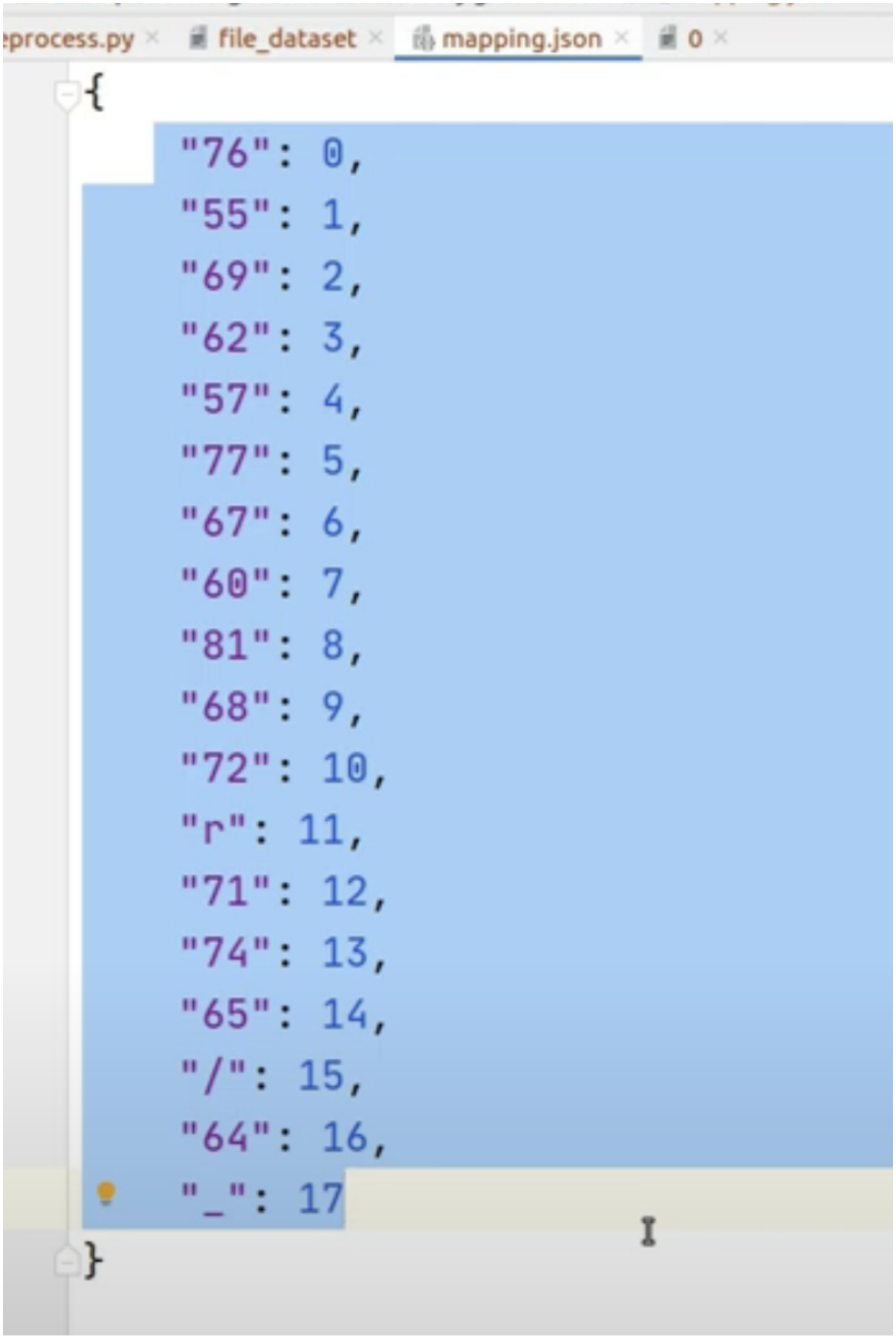
```
def load(file_path):
    with open(file_path, "r") as fp:
        song = fp.read()
    return song
```

```
def create_single_file_dataset(dataset_path, file_dataset_path, sequence_length):
    new_song_delimiter = "/ " * sequence_length
    songs = ""

    # load encoded songs and add delimiters
    for path, _, files in os.walk(dataset_path):
        for file in files:
            file_path = os.path.join(path, file)
            song = load(file_path)
            songs = songs + song + " " + new_song_delimiter

    songs = songs[:-1]

    # save string that contains all the dataset
    with open(file_dataset_path, "w") as fp:
        fp.write(songs)

    return songs
```

# Mappings

- Now, we have only one dataset and it contains ' _ ' symbols and 'r' for the rest symbols. NN don't take such symbols so we need to map them into numbers.
- We will map each symbol to an integer.

```
def create_mapping(songs, mapping_path):
    mappings = {}

    # identify the vocabulary
    songs = songs.split()
    vocabulary = list(set(songs))

    # create mappings
    for i, symbol in enumerate(vocabulary):
        mappings[symbol] = i

    # save voabulary to a json file
    with open(mapping_path, "w") as fp:
        json.dump(mappings, fp)
```


eprocess.py
file_dataset
mapping.json
0
{
"76": 0,
"55": 1,
"69": 2,
"62": 3,
"57": 4,
"77": 5,
"67": 6,
"60": 7,
"81": 8,
"68": 9,
"72": 10,
"r": 11,
"71": 12,
"74": 13,
"65": 14,
"/": 15,
"64": 16,
"_": 17
}

Now, we have the created mappings, now we will map these symbols in our dataset. So each symbol in our dataset gets converted according to this mapping.

## Convert Songs to int

```
def convert_songs_to_int(songs):

    # load mappings

    # cast songs string to a list

    # map songs to int
```

```
def convert_songs_to_int(songs):
    int_songs = []

    # load mappings
    with open(MAPPING_PATH, "r") as fp:
        mappings = json.load(fp)

    # cast songs string to a list
    songs = songs.split()

    # map songs to int
    for symbol in songs:
        int_songs.append(mappings[symbol])

    return int_songs
```

```
def generating_training_sequences(sequence_length):
    # [11, 12, 13, 14, ...] -> i: [11, 12], t: 13; i: [12, 13], t: 14

    # load songs and map them to int
    songs = load(SINGLE_FILE_DATASET)
    int_songs = convert_songs_to_int(songs)

    # generate the training sequences
    # 100 symbols, 64 sl, 100 - 64 = 36
    inputs = []
    targets = []

    num_sequences = len(int_songs) - sequence_length
    for i in range(num_sequences):
        inputs.append(int_songs[i:i+sequence_length])
        targets.append(int_songs[i+sequence_length])

    # one-hot encode the sequences
    # inputs: (# of sequences, sequence length, vocabulary size)
    # [ [0, 1, 2], [1, 1, 2] ] -> [ [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ],  []]
    vocabulary_size = len(set(int_songs))
    inputs = keras.utils.to_categorical(inputs, num_classes=vocabulary_size)
```

```
    # one-hot encode the sequences
    # inputs: (# of sequences, sequence length, vocabulary size)
    # [ [0, 1, 2], [1, 1, 2] ] -> [ [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ],  []]
    vocabulary_size = len(set(int_songs))
    inputs = keras.utils.to_categorical(inputs, num_classes=vocabulary_size)
    targets = np.array(targets)

    return inputs,
```