By: Marwan Harajli
Subject: Springboard Capstone 2 First Milestone Report
Last Updated: 03/02/2019

Problem Statement:

Given a data set of “music”, we investigate learning what constitutes music (using Deep Learning tools), and generating new unheard music. More specifically, we limit the scope to solo piano pieces (no other instrument).

Project Importance:

This project is a great way of getting exposed to and familiar with various neural network architectures (LSTM, ConvNet, GAN net, etc.). Furthermore, the project presents an opportunity in dealing with a problem that does not clearly fall within either of supervised or unsupervised learning. Quantifying performance is consequently complex and requires some thought (which I think separates it from the typical data science project where accuracy and F1 scores reign supreme). Finally, the project (subjectively) has a certain “cool” factor that I as an amateur musician am attracted to: a computer is creating music!

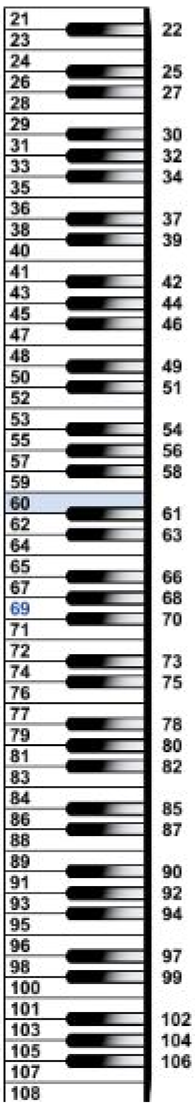
The Data:

We obtain piano midi tracks from this link. We use music21 to parse this data and obtain a representation of each midi file. This representation is called a stream, and is basically a list of music21 objects (either a note, chord or other event like an instrument definition). We concentrate on note and chord objects and list some of their relevant attributes:

- An offset: the time the note/chord started relative to the start of the stream (the start of the stream has offset 0).
- A duration, the time the note/chord lasts for.
- The pitch of the note (or pitch of each note in a chord) played. This indicates what key/keys on the piano is/are being played. Pitches are represented by a number from 1 to 127. The pitch number of each key in a piano is shown in figure 1 (note that a piano covers pitches from 21 to 108).
- A velocity: how hard the piano key is hit to play this note/chord.

We note that there are different types of music21 objects that we are not discussing. These objects enrich a musical piece and help define certain dynamic properties (like tempo of a song, time signature, and velocity); however we ignore them for simplification. With this simplification, all songs have the same tempo (which slows some and speeds others), and all notes have the same volume (no key is played softer/harder than any other).

Figure 1: MIDI pitch representation on a piano



Exploratory Data Analysis:

In this section we take a closer look at the chord and note events occurring for all the piano pieces in our dataset. The goal of this analysis is to understand how the attributes of notes and chords are distributed, and which ones play an important role in explaining what makes music sound as it sounds. This ultimately should help in the encoding of the midi files into a format that can be used to train deep neural nets (that will ultimately generate music). All code for this section can be found in the jupyter notebook named: *Milestone1Report.ipynb.*

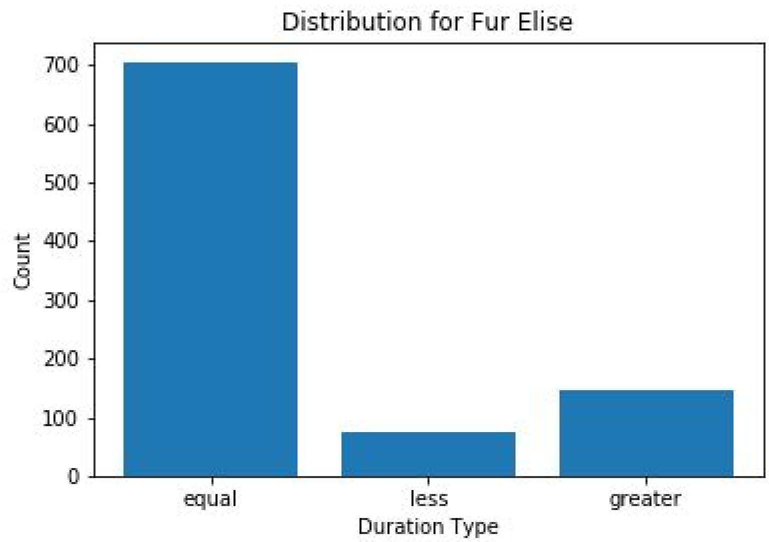
Since each note has an offset and a duration, one would think that one of the two attributes is redundant (the duration of the note could be obtained by subtracting the note's offset from the next event's offset, assuming both duration and offset have the same unit of time). That is in fact not true. There are two scenarios where this fails:

1. A note is played, then stopped. A rest (i.e. no sound) occurs for a short while, before a new note is played. This would give a duration that is smaller than the difference of offsets.
2. A note is played and held, and a new note is played as the note is still being held. This would cause the duration of the first note played to be larger than the difference of offsets.

We begin by observing the distribution of note duration types, where the the three types are: duration equal to the offset difference, duration less that the offset difference, and duration greater than the offset difference. We first look at how this distribution varies for different select songs, before obtaining the distribution over the entire dataset. In figures 2 3 and 4, the x-axis denotes whether the duration of the note is "equal" to, "less" than, or "greater" than the offset difference.

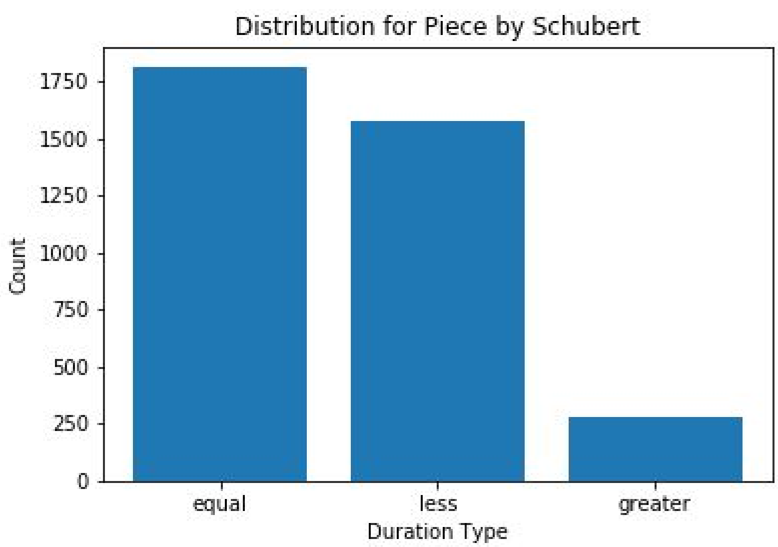
We begin by looking at Beethoven's Fur Elise. We get the following distribution:

Figure 2: Note duration types for Beethoven's Fur Elise



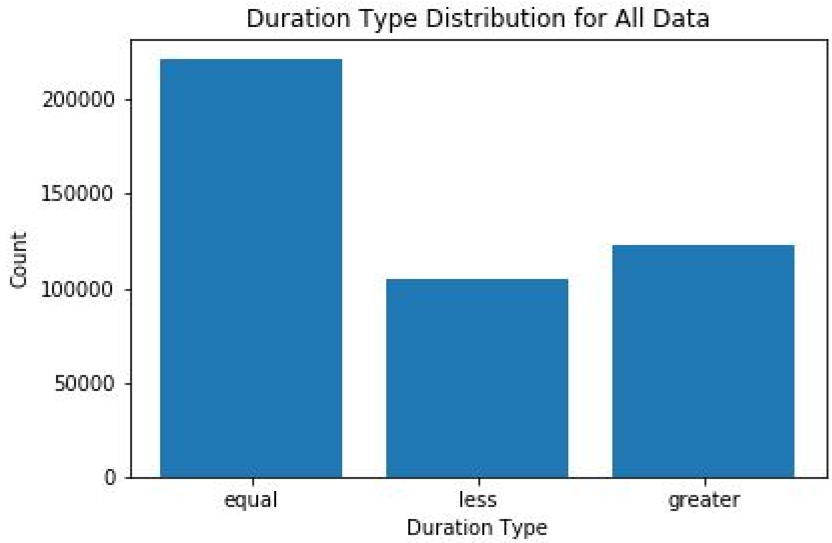
Next we look at a piano piece by Schubert:

Figure 3: Note duration types for piano piece by Schubert



As can be seen, the distribution varies between the songs. In Figure 4 we show the distribution over all the songs in our dataset.

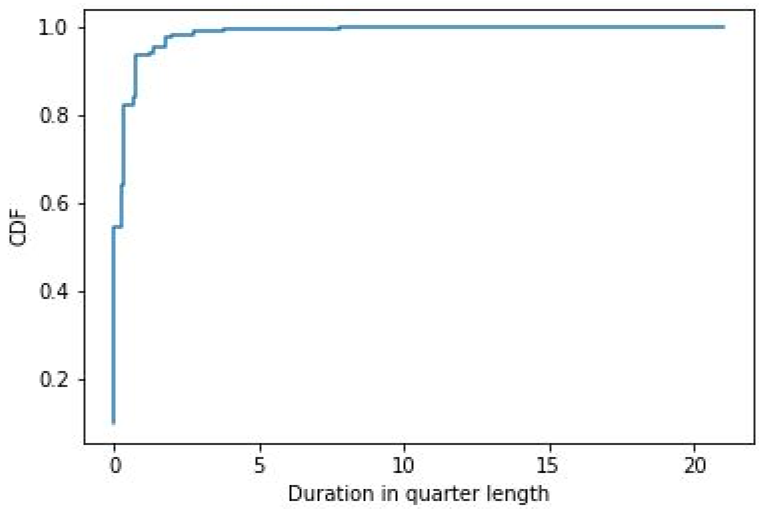
Figure 4: Note duration types for entire dataset



When looking at the entire dataset, we see that in fact the most common type of duration is that which is equal to the offset difference.

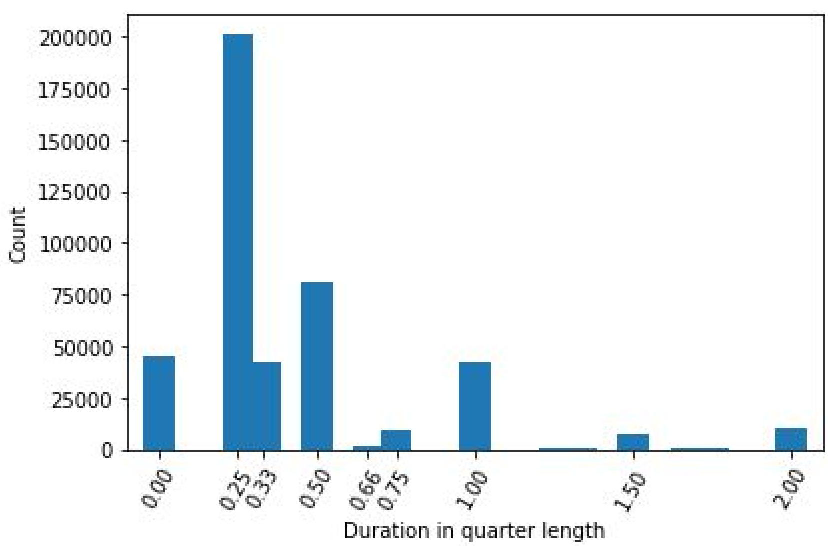
We next look at the distribution of note/chord durations. While we could use seconds (or any measure of absolute time) as a measure of duration, it is more useful to think of duration in terms of quarter notes. Quarter notes themselves are loosely defined temporally and depend on the beats per minute the songs utilizes. It it is more useful to think about duration relative to a song's inherent beat than in seconds. Below we show the cumulative distribution function of duration over all the dataset. The cumulative distribution is shown first (as opposed to the histogram), since it better handles the sparse space of occuring values.

Figure 5: Note duration types for entire dataset



From above, we see that the majority of notes/chords have durations less than 4 quarter lengths. Taking a closer look, Figure 6 shows a histogram for durations up to 2 quarter lengths.

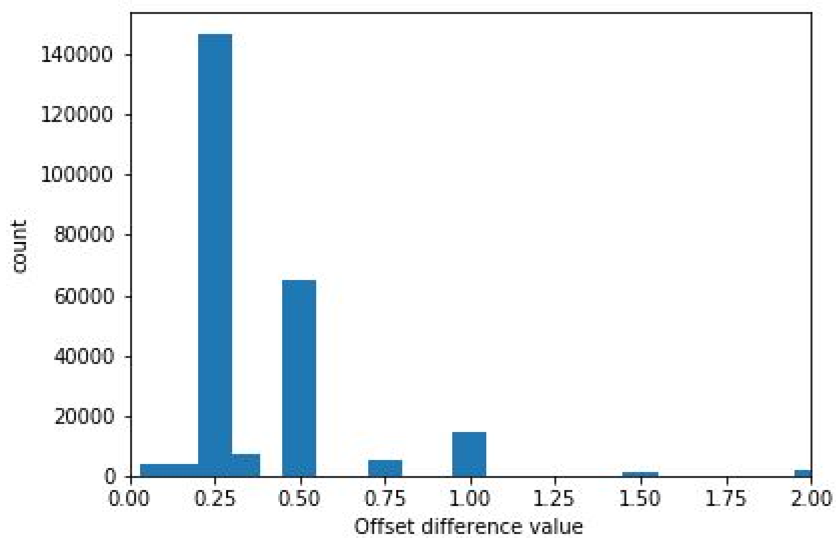
Figure 6: Histogram of durations for durations up to 2 quarter lengths



We can see from the above that most note are 0.25 quarter lengths (i.e. a sixteenth note). We also see the presence of notes of 0.33 quarter lengths (these are called triplets since we get 3 notes per beat).

Next we check the distribution of offset differences. In other words we obtain the distribution of the difference between an offset and the next event's offset. We note that more than one event may share an offset. Figure 7 shows the histogram of offset differences up to 2 seconds.

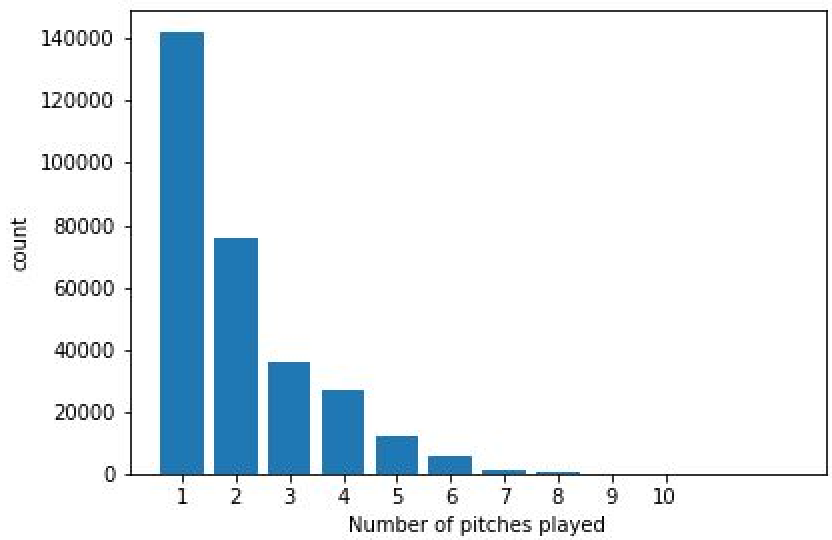
Figure 7: Histogram of offset differences up to 2 quarter lengths differences



We next check the distribution of the total number of pitches played at any given offset. We recall that a chord is a group of pitches played together. We also recall that more than one event can share the same offset. We thus obtain all the events for a given offset and count all the pitches played. Since all the music is piano music, and humans have 10 fingers, we expect that at most 10 fingers are used at any given offset and that most offsets involve events that use less than 10 pitches.

Figure 8 shows the distribution of pitches played at each offset.

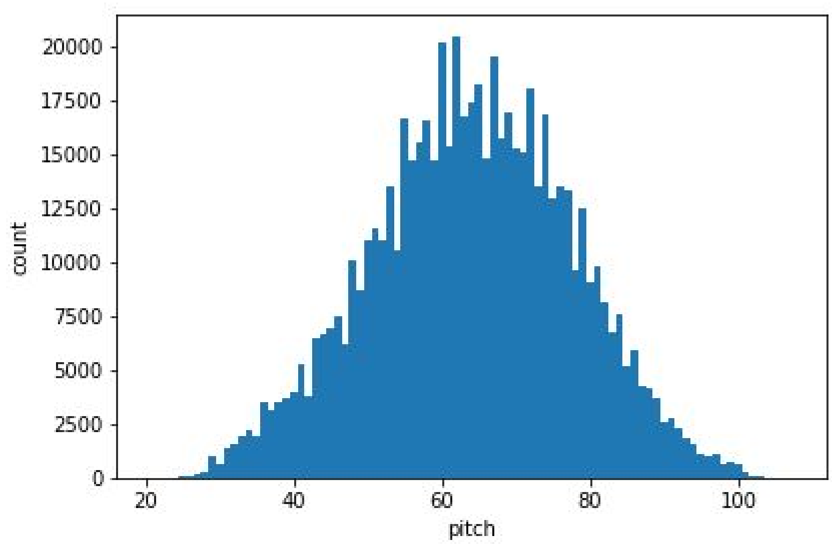
Figure 8: Histogram of number of pitches played at each offset



As can be seen in figure 8, the majority of offsets have one pitch being played.

Finally, we show the distribution of pitches used for the entire dataset. This will help us see if we can drop some unused keys when representing music, which could help us reduce the dimensionality of our feature space.

Figure 9: Histogram of pitches played



We can see from the above that the majority of the notes are centered about pitch 60 (which is called the middle C, also highlighted in figure 1). More so, we calculate that the 99% of the notes are between pitches 31 and 97.

Encoding Music:

In this section we discuss how to represent each song. We recall from the [proposal] that music should be expressed as a sequence of events. That way we can train a recurrent neural network to predict the next event. Since the majority of durations can be inferred from the offsets, we decide to drop durations from our representation and only encode offsets. This will change the music slightly but keep the general melody. Moreover we drop chords from the representation, since these can be expressed by note events sharing the same offset. With that the encoded sequence should have the following properties:

1. Be a sequence of events.
2. Each event should be a note event or a “goforward” event. Go forward events indicate the end of an offset and the start of a new one. A go forward event should have some indication how far forward to go (one quarter length, a quarter of a quarter length, etc.)
3. No more than 10 note events should occur between two goforward events.
4. No goforward event should be followed by another goforward event.

With the above, Beethoven’s Fur Elise become represented as follows:

[‘76’, 'gf0.25', ’75’, 'gf0.25', ‘76’, 'gf0.25', ‘75’, 'gf0.25', ‘76’, 'gf0.25', ‘71’, 'gf0.25', ‘74’, 'gf0.25', ‘72’, 'gf0.25', ‘45’, ‘69’...]

Above we highlighted the goforwards events to show where time jumps occur. We note that element that follow each other in the sequence above are not necessarily played after each other. In fact that is only the case if there is at least on “gf” event between them.

A couple of observations:

- The gf0.25 indicated a jump in time, where the jump is a quarter of a quarter length.
- More than one note can be played at a certain time, for example the 45,69 at the end of the shown list above.
- We only got goforwards (gf) events of duration 0.25. This is by chance, we will show below the encoding for other songs and see how there can be various gf events.
- When more than one note is played at one time offset, the notes are listed in increasing order. This might help as sequences have a more ordered structure.

We show the encoding for a song by Debussy:

['65', '68', 'gf0.5', '77', '80', 'gf2.0', '73', '77', 'gf1.5', '66', '69', 'gf0.5', '72', '75', 'gf0.5', '73', '77' ...]

In the jupyter notebook Miestone1Report, two functions: encode_song() and decode_song() encode and decode songs. The encode_song() function takes as input a midi and returns a list as shown above, while the decode songs takes in a list and a filename, and saves a new midi file in the location specified by filename.

We encode and decode Beethoven's Fur Elise and listen to it to make sure that the melody is still there after the encoding and decoding process. The midi file from that can be found in the samples folder in the midi file called: encoded_decoded_fur_elise.mid. As can be heard, the melody is intact albeit a bit sped up. This can be fixed by changing the tempo of the stream if necessary.

Generating Music:

To generate music 3 neural networks are trained. The first neural network predicts whether the next event in a sequence is gf event or a note event (call this network gf_or_note), the second neural network predicts the duration of the next gf event (get_gf), and the third neural network predicts the next note event (get_note).

With that, if we train the three neural nets mentioned above, the following scheme is used to generate music:

1. Begin with a sequence of Si for i in [i,i+1,i+2,...k], i<k.
2. If the last element is a gf event use get_note on the sequence to generate a note evenet. Go to step 5.
3. Otherwise, using gf_or_note decide if the next event is a gf or a note.
4. If the above returns gf, use get_gf on the sequence to obtain the appropriate gf event, otherwise generate a note event using get_note on the sequence.
5. Now consider Si for i in [i+1,i+2,...,k+1]. If you want to make the song longe, go to step 2, otherwise stop here.

Next Steps:

In the following milestone report, the above scheme is tested and its results assessed.