EQ2340 - Pattern Recognition
A.2: Feature Extraction - Song Recognition

Tianxiao Zhao
tzh@kth.se

Zheyu Zhang
zheyuz@kth.se

October 13, 2017

1 Working Inputs

In this assignment, we decide to design a feature extractor for the final song recognizer, and we will use the three recordings from `Songs.zip` as our inputs in the design process. To get a general understanding, plots of the pitch and intensity profiles of these melodies are attached below, see Figure 1 and 2.

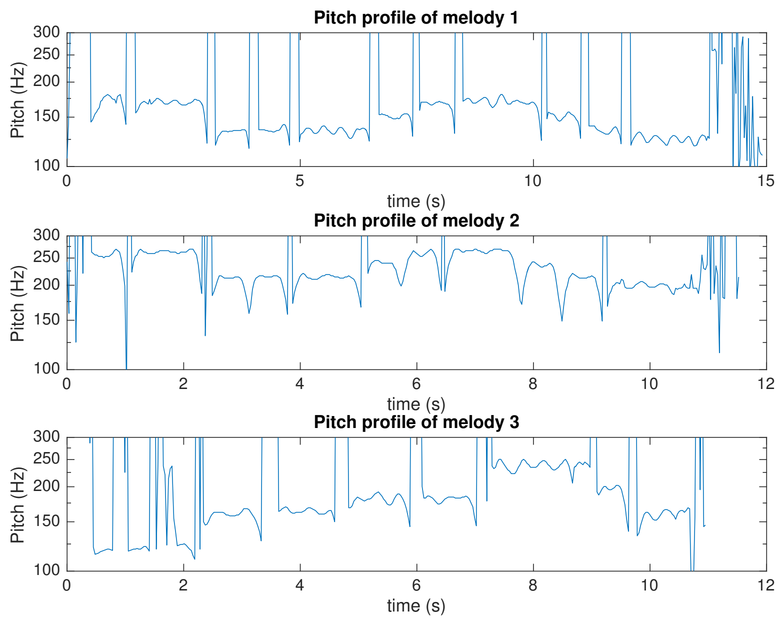

Figure 1: Pitch profiles of the three recordings from `Songs.zip`. The Y-axes are changed to logarithmic scales and limited to the frequency range 100-300 Hz in order to preserve a clearer view.

According to the instruction, we transform the Y-axes of these two plots into logarithmic scale since information in pitch and intensity track varies a lot. The major changes of pitch track (excluding noises) are around 100-300 Hz, so we also limit the frequency range of the first plot for easy observation. The unit of X-axes (number of windows) is transformed into seconds. From Figure 1, we can recognize from pure observation that melody 1 and 2 are from the same melody while melody 3 is from another one because melody 1 and 2 seem to share a similar quotient between different frequencies but melody 3 doesn't. Those abrupt jumps represent noises and pauses, which are not taken into account when estimating the quotient. The MatLab code for generating these plots is included below:

```
% read in audio files
[Y1, FS1] = audioread('melody_1.wav');
[Y2, FS2] = audioread('melody_2.wav');
[Y3, FS3] = audioread('melody_3.wav');
```

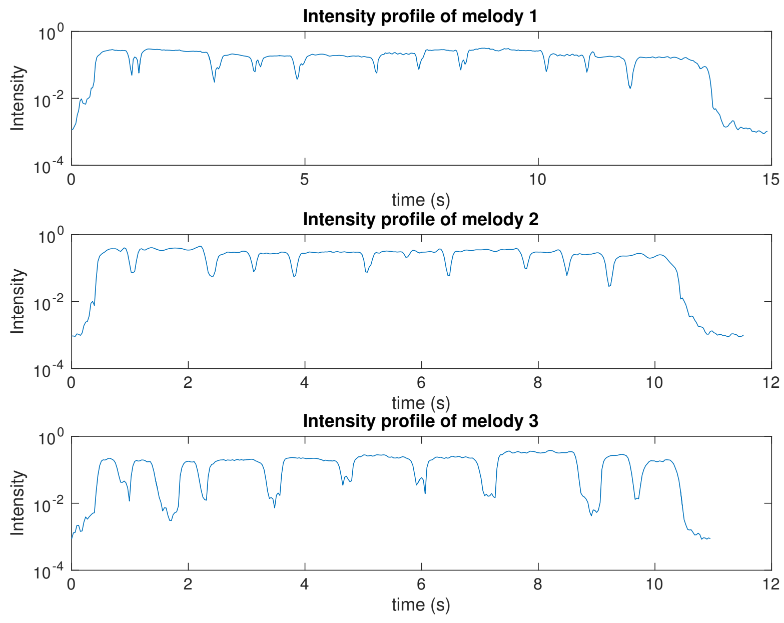

Figure 2: Intensity profiles of the three recordings from `Songs.zip`. The Y-axes are changed to logarithmic scales to preserve a clearer view.

```

% extract feature matrix from source
winlen = 0.03;
frIseq1 = GetMusicFeatures(Y1, FS1, winlen);
frIseq2 = GetMusicFeatures(Y2, FS2, winlen);
frIseq3 = GetMusicFeatures(Y3, FS3, winlen);

% convert numbers of window into time
t1 = 0 : winlen : (size(frIseq1, 2) - 1)*winlen;
t2 = 0 : winlen : (size(frIseq2, 2) - 1)*winlen;
t3 = 0 : winlen : (size(frIseq3, 2) - 1)*winlen;

% pitch profile
figure;
ax1 = subplot(3, 1, 1);
plot(t1, frIseq1(1, :));
title('Pitch profile of melody 1');
xlabel('time (s)'); ylabel('Pitch (Hz)');
ax2 = subplot(3, 1, 2);
plot(t2, frIseq2(1, :));
title('Pitch profile of melody 2');
xlabel('time (s)'); ylabel('Pitch (Hz)');
ax3 = subplot(3, 1, 3);
plot(t3, frIseq3(1, :));
title('Pitch profile of melody 3');
xlabel('time (s)'); ylabel('Pitch (Hz)');
set([ax1 ax2 ax3], 'YScale','log');
set([ax1 ax2 ax3], 'YLim', [100 300]);

% intensity profile
figure;
ax1 = subplot(3, 1, 1);
plot(t1, frIseq1(3, :));
title('Intensity profile of melody 1');
xlabel('time (s)'); ylabel('Intensity');
ax2 = subplot(3, 1, 2);
plot(t2, frIseq2(3, :));
title('Intensity profile of melody 2');
xlabel('time (s)'); ylabel('Intensity');
ax3 = subplot(3, 1, 3);
```

```
plot(t3, frIseq3(3, :));
title('Intensity profile of melody 3');
xlabel('time (s)'); ylabel('Intensity');
set([ax1 ax2 ax3], 'YScale','log');
```

2 Design Specification

Since combinations of semitones form the basis of all Western music, the design of our feature extractor is based on this octave and semitone mechanism. As is mentioned in the instruction, the most fundamental relationship between two pitches is that one is double the frequency of the other. This interval is known as an octave. Suppose an octave starts with pitch p_0 (also with a semitone of 1) and ends with pitch p_e, then the relation $p_e = 2 \cdot p_0$ holds. Since an octave on logarithmic scale is equally divided into 12 parts, we can assign any pitch p_k in this octave a continuous semitone k by

$$\log p_k = \log p_0 + (k-1) \cdot \frac{\log p_e - \log p_0}{12} = \log p_0 + (k-1) \cdot \frac{\log 2}{12} \Leftrightarrow \log \frac{p_k}{p_0} = (k-1) \cdot \frac{\log 2}{12} \tag{2.1}$$

$$k = 12 \cdot \frac{\log(p_k/p_0)}{\log 2} + 1 = 12 \cdot \log_2 \frac{p_k}{p_0} + 1 \tag{2.2}$$

By equation 2.2, we could transform frequency information in pitch track into semitones. Here, we choose to use this continuous k as our post-processed feature. The reason why we choose a continuous form over a discrete one lies in the fact that it could also smoothly represent out-of-tune notes which may appear in hums. On the other hand, if discrete feature is used, those out-of-tune notes will be first converted into a decimal number and then rounded to the closest integer. In this process, part of the information about out-of-tune notes will be lost, and this rounding operation also does not take the dependency between consecutive notes into account. Therefore, we choose the continuous form over the discrete one.

As for pause and noise, we implement a naive filtering operation before converting pitch into semitones. The general idea is to set several thresholds for different tracks to separate notes and non-notes in the input. First, those regions with a low intensity and a low correlation coefficient are for sure noisy or silent. We set the thresholds based on the mean values of the intensity series and correlation coefficients. Besides, we also pay certain attention to the pitch estimate as very high or very low pitch frequencies indicate the presence of silent or unvoiced segments in this case. So we set another two thresholds, one upper and one lower, to the pitch series. Figure 3 illustrates the thresholds of melody 1 as an example. The code responsible for thresholding is attached here for reference:

```
% post-process features
p = log(frIseq(1, :));
r = frIseq(2, :);
I = log(frIseq(3, :));

% detect noisy and silent region
p_thresh_pos = mean(p) + std(p);
p_thresh_neg = mean(p) - std(p);
r_thresh = mean(r);
I_thresh = mean(I);
noise = (p < p_thresh_neg) | (p > p_thresh_pos) | ((I < I_thresh) & (r < r_thresh));
```

After the noisy and silent regions are recognized, we can convert pitch information into semitones according to equation 2.2. And random values within the range [0, 0.5] are returned for noises and pauses. Code for the above part can be found below:

```
% convert pitch information into semitone (continuous)
base_p = min(p(find(noise == 0)));
semitone = 12*log2(p/base_p) + 1;

% return random values around 0 for noise and pause
semitone(find(noise == 1)) = 0.5*rand(size(find(noise == 1)));
```

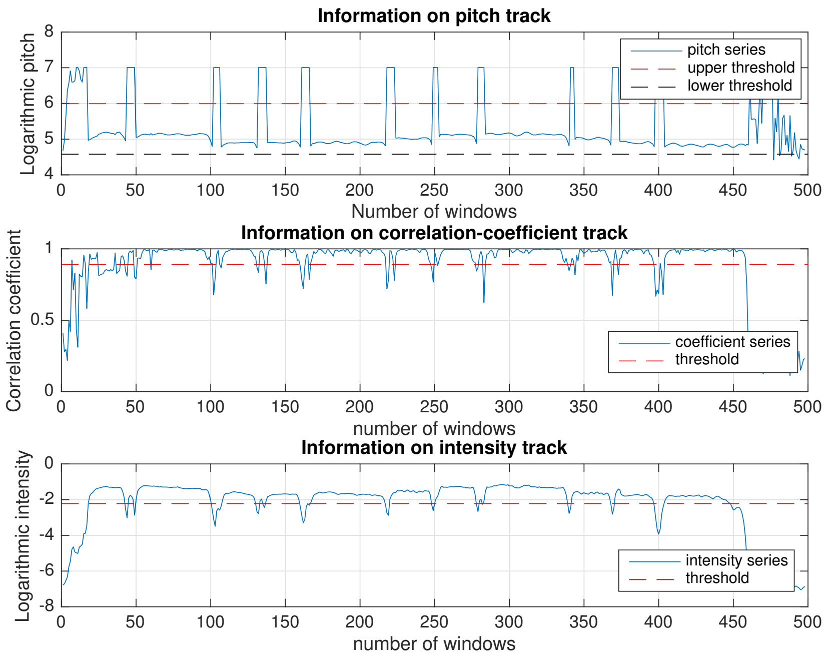

Figure 3: Features series of melody 1 along with their thresholds. In the first subplot, the red dotted line is the upper threshold while the black one is the lower one. Series between them are regarded as notes. The red dotted lines in the following two subplots are thresholds under both of which are regarded as silent or noisy segments.

3 Requirements Check

Similar to the output of assignment 1, our feature output here could be generated by a bunch of `GaussD` sub-sources. Specifically in an octave, the continuous output should mainly be around integers 0-12 as the transition between 12 semitones and pause. Therefore, using `GaussD('Mean', k, 'StDev', 0.5)` for $k = 0, 1, ..., 11, 12$ could theoretically give out a similar feature output. Note our design could also handle the situation that the pitch of song spans over several octaves in one snippet. The converted semitones will be larger than 12 and the range mainly depends on how many octaves the pitch spans over. For instance, suppose that the pitch of song spans over two octaves, then we need 25 sub-sources with output distribution `GaussD('Mean', k, 'StDev', 0.5)` for $k = 0, 1, ..., 23, 24$ respectively to generate a similar feature output.

Our design can be verified analytically to meet the requirements in the instruction:

1. **They should allow distinguishing between different melodies, i.e., sequences of notes where the ratios between note frequencies may differ.**
   According to 2.2, the combination of semitones is decided by the ratios between note frequencies within a melody. Therefore, different melodies can be easily distinguished.

2. **They should also allow distinguishing between note sequences with the same pitch track, but where note or pause durations differ.**
   Similarly, when notes or pauses last longer or shorter while the ratios between note frequencies remain the same, one of the feature plots of the two melodies will just be a time-domain rescaled version of the others but they will share a similar shape.

3. **They should be insensitive to transposition (all notes in a melody pitched up or down by the same number of semitones).**
   Suppose for a note sequence $\{p_1, p_2, ..., p_n\}$, we pitch up all notes by a value of α (α could be larger or smaller than 1). The new sequence then becomes $\{\alpha \cdot p_1, \alpha \cdot p_2, ..., \alpha \cdot p_n\}$. According to 2.2, $k = 12 \cdot \log_2 \frac{\alpha \cdot p_k}{\alpha \cdot p_0} + 1 = 12 \cdot \log_2 \frac{p_k}{p_0} + 1$, which means that our design is insensitive to transposition.

4. **In quiet segments, the pitch track is unreliable and may be influenced by background noises in your recordings. This should not affect the features too much, or how they perceive the relative pitches of two notes separated by a pause.**
   For quiet segments, double thresholds are set for the pitch track to filter out too large or too small values. Also another threshold of correlation coefficient can help filter out those segments with a weak correlation with note sequences.

5. **They should not be particularly sensitive if the same melody is played at a different volume.**
   As we all know, volume is only related to intensity. In our case, intensity is used for thresholding noises and pauses. The threshold is set based on the mean value of the intensity series, and it won't affect the result of noise detection when the same melody is played at a different volume constantly. Since our feature output is mainly decided based on pitch track information, we believe that our design is safely insensitive to volume changes.

6. **They should not be overly sensitive to brief episodes where the estimated pitch jumps an octave (the frequency suddenly doubles or halves). This is a common error in some pitch estimators, though it does not seem to be particularly prevalent in this melody recognition task.**
   This requirement is just a special case of requirement 3 when $\alpha = 2$ or $\alpha = 0.5$, which can be reliably handled by our feature extractor.

4 Performances

We put the feature outputs of the three melodies together and attach it below, see Figure 4. By observing these plots of feature sequences from the extractor, we get the intuition that feature sequence of melody 1 and melody 2 are quite similar to each other, whereas the third one is significantly different.

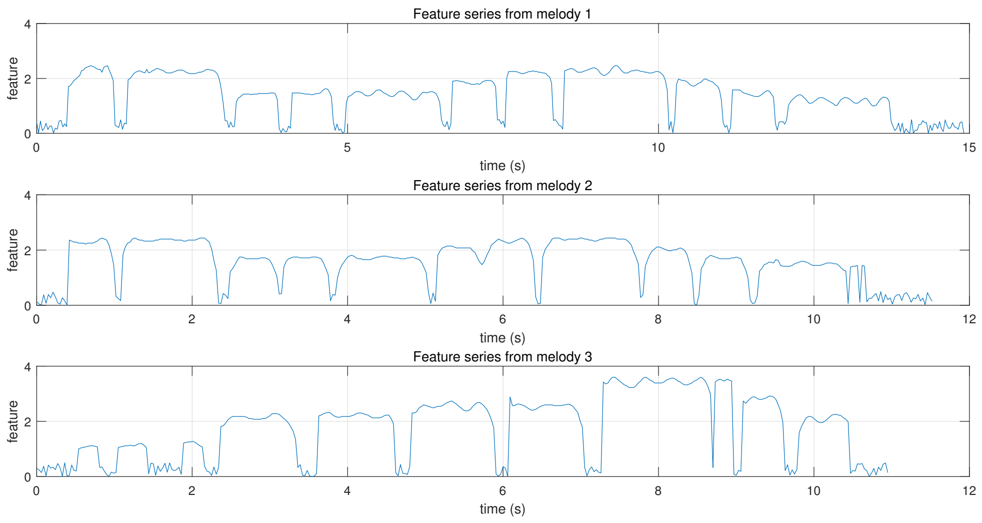

Figure 4: Extracted feature series from the extractor

Additionally, we calculate the similarity between two temporal sequences which may vary in length using *Dynamic Time Warping (DTW)* algorithm[1]. The smaller the distance between two sequences is, the more similar they are. The results below also support our argument above that feature sequence 1 and 2 are similar, however feature sequence 3 is different.

```
------------ sequence similarity ------------
Distance between feature seq 1 and 2: 15.4509
Distance between feature seq 1 and 3: 195.2649
Distance between feature seq 2 and 3: 147.5851
```

To prove that our feature extractor is transition-independent, we multiply the pitch track by a value of 1.5 and then feed it into our extractor. From the plot of feature output of melodies with transposed pitch track in Figure 5, it is obvious that scaling the pitch track does not change the outline of extracted feature sequences, which indicates that our design is transposition-independent.

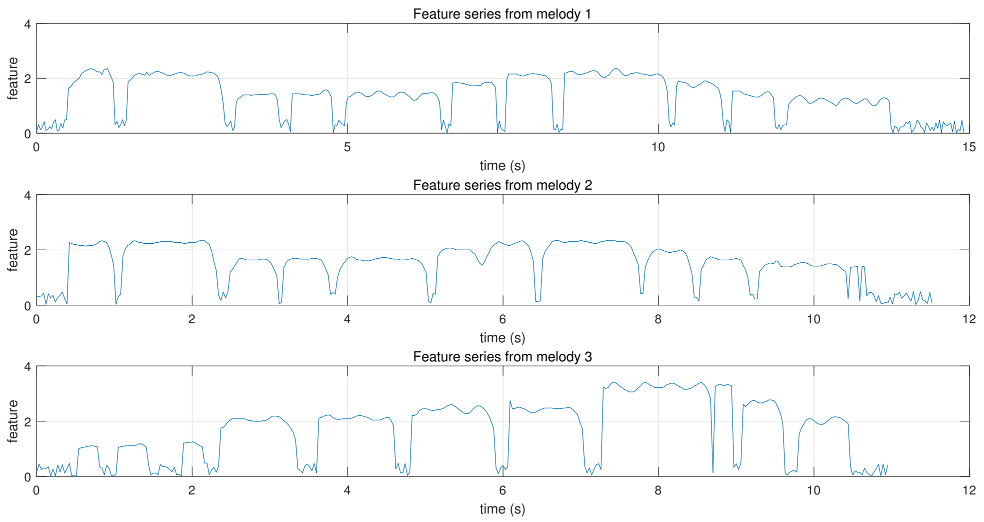

Figure 5: Feature series of melodies with transposed pitch track from the extractor

The code for these plots is attached here for reference. Note if the plot of feature series with transposed pitch track is desired, please change the variable `factror = [1; 1; 1];` to `factror = [1.5; 1; 1];` in *line 15* to rescale the pitch track.

```
1  clear;
2  addpath(genpath('GetMusicFeatures'));
3  addpath(genpath('Songs'));
4
5  [Y1, FS1] = audioread('melody_1.wav');
6  [Y2, FS2] = audioread('melody_2.wav');
7  [Y3, FS3] = audioread('melody_3.wav');
8
9  winlen = 0.03;
10 frIseq1 = GetMusicFeatures(Y1, FS1, winlen);
11 frIseq2 = GetMusicFeatures(Y2, FS2, winlen);
12 frIseq3 = GetMusicFeatures(Y3, FS3, winlen);
13
14 % rescale pitch track by a factor
15 factor = [1; 1; 1];
16 st1 = PostProcess(frIseq1.*repmat(factor, 1, size(frIseq1, 2)), true);
17 st2 = PostProcess(frIseq2.*repmat(factor, 1, size(frIseq2, 2)), true);
18 st3 = PostProcess(frIseq3.*repmat(factor, 1, size(frIseq3, 2)), true);
19
20 % convert numbers of window into time
21 t1 = 0 : winlen : (size(frIseq1, 2) - 1)*winlen;
22 t2 = 0 : winlen : (size(frIseq2, 2) - 1)*winlen;
23 t3 = 0 : winlen : (size(frIseq3, 2) - 1)*winlen;
24
25 % pitch profile
26 figure;
27 ax1 = subplot(3, 1, 1);
28 plot(t1, st1); grid on;
29 title('Feature series from melody 1');
30 xlabel('time (s)'); ylabel('feature');
31 ax2 = subplot(3, 1, 2);
32 plot(t2, st2); grid on;
33 title('Feature series from melody 2');
34 xlabel('time (s)'); ylabel('feature');
35 ax3 = subplot(3, 1, 3);
```

```
plot(t3, st3); grid on;
title('Feature series from melody 3');
xlabel('time (s)'); ylabel('feature');
set([ax1 ax2 ax3], 'YLim', [0 4]);

% similarity
Dist12 = dtw(st1, st2);
Dist13 = dtw(st1, st3);
Dist23 = dtw(st2, st3);
disp('------------- sequence similarity -------------');
disp(['Distance between feature seq 1 and 2: ' num2str(Dist12)]);
disp(['Distance between feature seq 1 and 3: ' num2str(Dist13)]);
disp(['Distance between feature seq 2 and 3: ' num2str(Dist23)]);
fprintf('\r');
```

5 Discussion and Conclusion

As for possible flaws of our design, how we detect and deal with noisy and silent segments is what we should pay more attention to. First of all, the thresholds are not fixed and are mainly decided by the mean value and standard deviation of the corresponding feature series. It remains unclear whether this strategy of choosing thresholds works in a more general case or not, and maybe more test cases should be involved to verify this strategy.

Besides, after setting the thresholds, we classify a segment to be silent or note-playing by combining several conditions with pitch track, correlation coefficient and intensity track together (see *line 11* of the first snippet of code in section: Design Specification). This way of combining conditions may be another thing that we can suspect. Only those regions that have a below-threshold correlation and a below-threshold intensity at the same time will be classified as non-note part, which may be too conservative under some circumstances. For instance, some noises may be assigned very low correlation coefficients but may have a large intensity, and therefore tend to be classified as non-noise (the thresholds of pitch track will help filter out segments like this, but there is a potential risk that a bad threshold of pitch track fails to rescue such a situation).

To sum up, our design could meet all the requirements of the instruction and produce continuous feature output sequences that can be generated by several `GaussD` sub-sources. For the test cases we try so far, both of the situations that similar sounds produce different feature outputs and the other way around never happen to our feature extractor. For improvements, we may need to come up with more clever ways to decide thresholds and combine them together in the future.

References

[1] Wikipedia. Dynamic time warping. `https://en.wikipedia.org/wiki/Dynamic_time_warping`, 2017. [Online; accessed 3-October-2017].