Report Special Topics: Audio and Music Processing

Working with audio files: onset, tempo and beat detection

Lecturer: Rainer Kelz - University: Johannes Kepler University Linz - Semester: SS22

Work done by: Student Nina Braunmiller – User K11923286 – Team LiszteningIn

In this report will be dicussed the methods in Python. For this purpose, efforts and experiments will be descirbed. 'train' refers always to training data which all three tasks have in common. 'test' means the data for which no solution is given.

1. Onset Detection - Aiming for a onset detection function

1.1 Short Time Fourier Transform (STFT)

Idea: Get frequency information of a signal. Even better, get also an idea where which frquency appears how strong with creating different time windows (not on point but good interval idea). **Implementation:** with number_overlap=100 was used to create the STFT 2d matrix which can be disassembled into time windows on the columns. Each column gives us one Discrete Fourier Transform (DFT) over a certain amount of time/samples. The number_overlap describes that 100 samples of one time window/column are reused also in the upcoming time window. The rows describe the different frequency bins k of the single DFT. **Usage for Onset Detection:** The detection function shows us in from of high values whether there is a big difference between the frequency distributions of neighboring time windows/columns of the STFT. However, how do we get the detection function? Two approaches where tried out (only one is needed):

1.2 Complex Domain (L04, p.33-34)

Idea: try out this approach because it combines the idea of using phase and amplitude together to find the detection function.

Predicting next bin $\hat{X}_k[t] = |X_k[t-1]| \cdot e^{j(\varphi_k[t-1]-\Delta\varphi_k[t-1])}$,

where $|X_k[t-1]|$ is the magnitude of previous bin k (0<=k<=N, as single DFT entries),

From L04, p. 30 we know: $\Delta\tilde{\varphi}_k[t-1] = (\tilde{\varphi}_k[t-1] - \tilde{\varphi}_k[t-2]) - (\tilde{\varphi}_k[t-2] - \tilde{\varphi}_k[t-3])$

To get the onset detection function: $d_{CD}[t] = \sum_{k=1}^{N} |X_k[t] - \hat{X}_k[t]|$

Implementation: We need the phase values for the whole STFT matrix (each k bin of single DFT considered). **Attention:** For Euler formula we don't need the geometric angle but the algebraic (https://realpython.com/python-complex-numbers/)! We have for example $STFT[20][20] = (-1763.016 - 1299.2828j)$ in which $ReX_{t=20}[20] = -1763.016$ and $ImX_{t=20}[20] = -1299.2828$. --> $e^{j(\varphi_k[t-1])}$ by using `np.exp(1j*np.angle(STFT))` .

Now we are searching for the $\Delta\tilde{\varphi}_k[t]$. For this purpose we compare the different time slots but stay for each comparision in the same bin k. Use `np.angle(STFT)` to get all $\tilde{\varphi}_k$ and compare columnwise the horizontal neighboring values of each frequency bin k. --> $\Delta\tilde{\varphi}$.

Bringing all together to compute $\hat{X}_k[t] = |X_k[t-1]| \cdot e^{j(\varphi_k[t-1]-\Delta\varphi_k[t-1])}$ and then the detection function $d_{CD}[t]$.

1.3 Used alternative: LogFiltSpecFlux (LFSF) (L04, p. 58-61)

Idea: This approach should not be as noisy as the Complex Domain. Because the Complex Domain did not reach a high score in Onset Detection I decided to try out the LFSF.

1.3.1 Creating a mel filterbank

Sources: L04,52-57; http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/#eqn1; **Idea:** Good https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html oppurtunity to compress the STFT row information.

With help of the formula $m = 7959 \cdot log_{10}\left(\frac{1+frequency}{700}\right)$ I was able to define the lower and upper frequency border of mel scale. Then a different number of uniform distributed mel scale numbers were added. All these mel scale numbers were converted back into frequency domain (filters). Those were fitted to our STFT k frequency bins (rows) with $floor\left(\frac{(NumberKFrequencyBinsSTFT+1)\cdot melScaleInFrequency}{NumberSamplesPerTimeWindow}\right)$ and a formula describing where each filter has which amplitude. Finally, I took that filterbank to make a cross-correlation with the STFT magnitudes to see how much each single filter has a high score with each single DFT (time window). Last but not least, I took the log of the result - pseudo code: $MelfilteredMatrix = 10 \cdot log_{10}(result + 1)$.

1.3.2 Spectral Differentiation

The MelfilteredMatrix is now used as replacement of the STFT matrix. Find out how much the different time windows differ in the respective characteristic of single frequency bins compared to the upcoming window. In this way, we can observe changes in frequency and therefore changes in an audio file. To find the spectral difference on sequential time windows we simply need to substract from the column/time window at time t the column of time t+1, which means for all columns at once: `MelfilteredMatrix[:,1:]-MelfilteredMatrix[:,:-1]`.

1.3.3 Taking sum

Now sum up all the frequency bins within the single columns. The result is a function with only one value per time window. The onset detection function.

1.4 Post process detection function

1.4.1 Normalizing the detection function

Use .mean() and .std() to get the mean and standard deviation of a detection function. Then calculate `(detection_func-mean)/std` to normalize the function. With that method the detection functions of different audio become comparable.

1.4.2 Adaptive thresholding at the detection function

Sources: L04, p. 46

Aim: Finally, pick the peaks. However, we want to exclude points in the function which are not suitable as peaks. When we would simply fix a threshold (fixed thresholding) we could get problems with silent passages of the piece where are also true onset peaks but all values would lay below that threshold and therefore would not be selected. To handle this problem, I implemented adaptive thresholding. It is a smoothed version of the detection function.

Implementation: For each time point calculate an adaptive threshold (L04, p. 46): $\delta_t[t] = \delta + \lambda \cdot median(d[t-k])_{k=-WindowSize/2}^{WindowSize/2}$ where δ is the standard threshold value (fixed thresholding) and λ increases the meaning of the median of the current window (adaptive thresholding). I used $\delta = 0$ and $\lambda = 1$ for which I reached the highest score. When the current data point is smaller, it is set to 0 such that it won't be chosen as peak in the next step.

1.5 Peaks

1.5.1 Peak picking

Aim: Finding points/onsets in the onset detection function which show us that there is a high differenence in frequency distribution between two neighboring time windows of STFT even higher than for other nearby pairs (local maxima).

Implementation: To find the local maximum we fix a hyperparameter **max value window**. Furthermore, this maximum has to be higher than a mean value of a second hyperparameter **mean value window**. When we find a peak, the next peaks needs to have a **distance** next to it. To save time the peak picking function is a bit more complicated than simply looping over the whole onset detection function. With the array 'all_peaks' we want to look only at values which are bigger than 0 in the onset detection function. Furthermore, we not simply loop over the 'all_peaks function', also here we make jumps when **a)** through a found onset the minimal distance for the next potential onset has to be considered and **b)** when we have within the max value window several maxima. Therefore, we would jump to the next maximum with same value when the first was not able to be a onset because it could not top the mean of the mean value window. Several hyperparameter settings were tried out. In the table you can find examples where it worked best:

Best hyperparameter setting for peak picking:

max value window	8	14	14
mean value window	22	22	18
distance	48	48	48
train	0.3333	0.3338	0.334

Different peak picking approaches:

	Imported package	All values > 0	Implemented
train	0.36	0.039	0.334
test (reach in)	0.385		0.38

1.5.2 Transfer peaks from time windows into seconds

We can simply use $\frac{peaks \cdot hopSampleNumber}{sampleRate}$ to transfer the peaks into seconds where hopSampleNumber is a hyperparameter which decides how much samples do not overlap of one STFT time window with its following window; sampleRate was gained at the beginning by importing the audio file into the notebook with $sampleRate, data = scipy.io.wavfile.read(file)$.

1.6 Final results

	Complex Domain	LFSF
train	0.330	0.36
test (reach in)	0.332	0.38

2. Tempo Estimation - based on Onset Detection Function

Sources: L05, p.22-29;
https://resources.mpi-inf.mpg.de/departments/d4/teaching/ss2010/mp_mm/2010_MuellerGrosche_Lecture_MusicProcessing_BeatTracking_handout.pdf

2.1 Splitting onset detection function

Idea: The tempo might vary within a single music file. The handle the variation we will compute the 7 tempi for $spitter=7$ disjoint windows out of the onset detection function (**"split windows"**). I have chosen 7 windows because I got the best score in training out of it. Even defining splitting a file concerning its duration didn't work as well. I also tried out to work with the full detection function with worse results.

2.2 Autocorrelation of onset detection function (windows)

Next the detection function (windows) were autocorrelated. With autocorrelation we can observe periodicity patterns in the onset function. Simply multiply elementwise the detection function with itself in the first step and then sum the values up to get first entry of the autocorrelation function. After that we multiply $detection\_func1 = detection\_func1[1:]$ with $detection\_func2 = detection\_func2[:-1]$ and sum up again. Do this until the arrays are []. So, detection_func1 loses with each multiplication the begining element and detection_func2 loses every time the last element.

2.3 Considering only relevant tempi

For music we normally have a tempo range of 60 up to 200 bpm. It is enough to consider only that range. For this purpose calculate the place in time windows where 60 and 200 bpm ('givenBmp') lay on the x-axis of autocorrelation with $autocorrelationXValue_{givenBmp} = \frac{60 \cdot sampleRate}{splitter \cdot givenBmp \cdot hopSampleNumber}$ where we use 60 to mark 60 seconds because we want bpm (beats per MINUTE). Now we get for each autocorrelation of the 7 single time splitters the relevant lower and upper border indices of x-axis.

2.4 Estimate tempo

In the described window from above $autocorrelation[autocorrelationXValue_{200} + 1 : autocorrelationXValue_{60} + 1]$ we now search for the maximal value. At this value we get a index which we convert back to bpm: $tempo = \frac{60}{lagInSec} = \frac{60}{\frac{autocorrelationXValue_{maximalValue} \cdot hopSampleNumber}{SampleRate}}$ where lag describes the x-axis of the autocorrelation and says how much the autocorrelation array shift is. This tempo is the most indicating one out of this window! Caculate the tempo for each split part. Finally, I selected from the split windows the highest tempo. As option you can additionally choose the second highest y-value tempo within each split window. However, it did not work out well for the final submission. Here is a comparison of scores deciding which tempi to reach finally in where the windowed version with highest found tempo out of the split windows as result leads to highest training score:

Methods of selecting the tempo/tempi estimation which should be reached in:

	Windowed	Full cross-correlation train
median*		0.15
mean*		0.16
all 7 found tempi		0.21
[smallest_tempo, highest_tempo]		0.24
most indicating tempo**		0.13
two highest indicating tempi		0.16
highest tempo		0.27
	[smallest_tempo, biggest_tempo]	0.17
	most indicating tempo**	0.16

* out of 7 tempi out of 7 windows
** with the highest indication store like we see later

2.5 Final tempo best results

	Complex Domain	LFSF
train	0.24	0.270
test (reach in)	0.22	0.298

3. Beat Detection - based on Onset Detection Funtion and Tempi

Sources: L05, p. 29;
https://resources.mpi-inf.mpg.de/departments/d4/teaching/ss2010/mp_mm/2010_MuellerGrosche_Lecture_MusicProcessing_BeatTracking_handout.pdf
Implementation: Use the onset detection function and estimated tempi of the split windows. Cross-correlate a sinus/cosine curve with tempo information with the onset detection function to find the first onset where a beat could be. Then reuse the gained beat period to find the upcoming beats. **Challenge:** Working with the 7 different tempi ans 7 split windows at once while avoiding loops. Further advanced array thinking was needed.

3.1 Implementation of pulse

Firstly, we want to create a periodic pulse which has its first maximal value at the found tempo from above. Because I have found 7 different tempi we consider all of them within one function at once.

3.1.1 Convert the 7 tempi into seconds

Because we will compare the detection function with the pulse in seconds as x-axis, we need to convert each tempo index of autocorrelation window into a time value [seconds]. Therefore, it was needed to get the index of each tempo in autocorrelation function by calculating the $indexTempo = 0 + (autocorrelationXValue_{200} + 1) + autocorrelationXValue_{maximalValue}$ where 0 marks the starting index of each split window. Then convert it into seconds like before.

3.1.2 Calculating pulse

The cosine pulse shall make use of the full tempi seconds array at once. Additionally, it is needed that its $period = \frac{1}{foundTempoSeconds}$. Let's define the cosine cos(Bx) where $period = \frac{2\pi}{|B|}$ such that we get $B = \frac{2\pi}{period} = \frac{2\pi}{\frac{1}{foundTempoSeconds}} = 2\pi foundTempoSeconds$ (https://study.com/academy/lesson/how-to-find-the-period-of-cosine-functions.html). Instead of foundTempo we can use an array filled with 7 rows describing tempi of split windows. So, we get $abs(cos(2\pi foundTempiArraySeconds \cdot input))$; where input is a 2d array in which each row refers to one row of the foundTempiArraySeconds (no inner product but normal multiplication), the columns describe the current time window index for which the individual tempi account. Only absolute values were taken because there are no negative magnitude values in the detection function. Besides that, also the sinus curve was tried out but the training results were almost the same (difference < 0.01).

3.2 Finding first beat location

3.2.1 Cross-correlation

Cross-correlate the 2d array output of cosine with onset detection function split windows. **Challenge:** Its like the autocorrelation function from before but to avoid extra looping for every single value, I made the array multiplications in one step like before and then again simply summed up the values, but for each row (one cross-correlation per row) (axis=1). Then I added the gained column vector as new column to a big array where we get whole crosscorrelation function in rows for each windowed tempo.

3.2.2 Ignoring the first hill of crosscorrelation

Of course when tau/shift=0 for cross-correlation we get in general a high value. This one shall be ignored. Find the first valley where one value is smaller than both of its neighbours. I wanted to avoid a for-loop and introduced a filter which does the job:
f = np.asarray([[1, 0, 1]]) having a look at the two horizontal neighbours of a value
valley_indices = np.where(resultCrosscorrel < scipy.ndimage.minimum_filter((resultCrosscorrel, footprint=f, mode='reflect', cval=-np.inf))
I only want the first valley in the cross-correlation:
valley_inds_rows, ind_val = np.unique(valley_indices[0], return_index=True)
valley_inds_col = valley_inds[1][ind_val]
For each row in valley_indices we only search first valley. With the from np.unique() returned ind_val we can find out the indices within valley_indices where a new row starts, e. g. [0, 0, 1] where at index=0 first row starts and at index=2 the second row/crosscorrelation. Use it to filter out the column values (where in which row the first valley starts).
Set all the values whom indices are smaller than the [valley_inds_rows, valley_inds_col] to 0 such that they can't get chosen as single row maxima of single windowed cross-correlations.

3.2.3 Find first beat location

For each row of the resultCrosscorrelation array find its index with maximal value with np.argmax(resultCrosscorrelation, axis=1).

3.3 Compute the following beat locations within each of the 7 windows

3.3.1 Calculating the beat period

When assumming that the beats are periodically, the period is the index of first beat in every every single split window.

3.3.2 While-looping over onset split window length

We create a while-loop lasting in the range of one split window length. In this the upcoming beat indices are calculated. No beat_index is allowed to be bigger than the last index of the current split window. As soon as one of the beat indices leaves the window, remove it from the list of current beat indices. When we found the next beat index by $StartBeatIndex + c \cdot StartBeatIndex$ we convert the next beat index into seconds by using its its index + the index start of the split window to get the seconds refering to the whole onset detection function.

3.3.3 Find closest onset to found beat location in seconds

From the first task of onset detection we already have an array containing all estimated onsets in seconds. Have now a look which of them are close to our found beat location in seconds, close was defined as a distance of +/- 0.05 seconds. When several onsets were close, the closest was selected with help of np.argmin(). This onset is now also counted as beat location. When no onset was close, the beat location in seconds was taken as final beat location estimation.

3.3 Final results

	Complex Domain	LFSF
train	0.27	0.258
test (reach in)	0.16	0.198

4. Explanation scores

The resulting F-scores (no matter if train or test set) are never higher then 0.39. The explanation for this is quite simple. The onset detection function is not only used for onset estimation (0.38 test score) but also for tempo estimation (0.298) and beat detection (0.198). When the onset detection did not work that well, then the other tasks which build up on it can't be better but even worse because also new calculations and implementations are added which also do not work perfectly. Therefore, it is only understandable that tempo and beat estimation worked even worse.
The last question stays why no higher score was reached for the onset detection function. One reason could be that the STFT was directly created out of the audio file (no preprocessing). Noise was not considered in this case. Additionally, the contents of the different files were always treated in the same way. It was not distinguished between degree of noisyness, type of audio (music, talking, daily sounds, live music, studio music ...), genre of music (classic, rock, ...), used instruments (guitar, piano, ...) and so on. Last but not least, hyperparameter settings could increase the scores. E. g. peak picking influence on onset detection function result: imported 0.385 vs. self implemented 0.38