Key Detection for Pop Music Supporting Modulation Point Locating Based on Transformer

Boning Wang bwang50@u.rochester.edu
Jiajun Wu jwu107@u.rochester.edu

Yichuan Wang ywang382@u.rochester.edu
Jiajun Chen jchen192@u.rochester.edu

ABSTRACT

This research aims to develop a transformer-based deep learning model for detecting music keys, and specifically addressing the challenge of modulation—key changes within a composition. Traditional methods often overlook this complexity, focusing only on identifying a single key. Our approach utilizes a dataset of 411 songs across various languages and genres, including songs with different key modulations. By using detailed preprocessing and a transformer model architecture, we demonstrate capabilities to detect keys and key changes within compositions. This advancement not only enhances key detection accuracy but also broadens the scope for future research in automated music analysis.

1. INTRODUCTION

Introduction

Key is one of the most essential features of a piece of music. It determines the patterns of notes involved and the color of emotion. Key detection plays a majority role in many tasks of contemporary music production, such as pitch correction and the reproduction of existing music. Currently, there exist a number of algorithms for key detection, and they are effectively implemented in standalone software, audio plug-ins, and music players. However, the majority of them are only able to output one result of key detection, but modulation is a common technique. Therefore, we decided to design an algorithm that can output several results that consist of all the keys appearing in the input audio file

Related Works

There'd been significant research done on the algorithms of music keys detection, and there are also software applications and websites developed that could recognize music keys. One of the methods that is commonly employed is Krumhansl-Schmuchler Profile[1]. This method is highly arbitrary: it is based on the subjective perception of music experts in musical psychology experiments, where after an incomplete tone scale is played, they judge and score the probability of suitability of each semitone in the next octave for preceding the scale. The scores constitute a probability profile, whereby the chroma vector of a piece of music is compared and the key information can be detected.

This method is effective because it is immuned from the negative impacts on the accuracy such as the lack of absolute tone sense and convoluted music theory knowledge. However, it is based too much on arbitrary judgment, which sometimes introduces errors, and it requires the engagement of extensive human labors.

Presently, there are also a number of key detection applications based on machine learning, which involve a large set of training data for the model to learn the spectral information of music audio wave files of different keys, then it can predict the key of any unseen audio clips. Methodologies include K-nearest Neighbors, Support Vector Machine, etc..[2] Such applications include Tunebat[3], Vocal Remover[4], apple music, tencent music, etc.. However, most of these applications can detect only one key given an audio file, ignoring the situation of modulation. As mentioned earlier, modulation is a frequently used technique in music composition, so musicians would prefer a key detection algorithm that could detect all the key and modulation information of a piece of music.

2. METHOD

2.1 Dataset and pre-processing

To train the neural network, we downloaded 411 songs as our training data including English songs, Mandarin Chinese songs, Cantonese songs, Japanese songs, and absolute music without the lyrics. 96 pieces of them include modulation. The common characteristic of them was that most of the notes existing are within the scale of local keys, and there were minor tonicizations and mixtures where notes outside the scales appear. The purpose was to obtain stable representations of the features of each key scale, and simultaneously ensure that those exceptions have also been captured. Next, we labeled all the keys that had appeared in a clip and their start time and end time. We used a matrix to represent all necessary information. There were three columns representing the start time of a key, end time of the key, and which key it was respectively. The rows were the sequence of the keys ordered chronologically. We used 0 to represent the key C, and with each increase in semitone, the key number was incremented by 1, whereby the 12 keys from C to B can be represented by integers from 0 to 11. For example, there was a clip with length 4:15. At the beginning, the key was C, and it modulated to C sharp at 2:45. For this clip, we converted the unit of the timestamps to seconds and created the following matrix to record the key information:

0	165	0
165	255	1

During the training process, we temporarily only focused on detecting a single key out of a given clip while considering the situation where a clip contained modulations. In order to extract more specific musical features, we further divided a song to multiple clips. Considering that the rhythm and the speed of songs varies remarkably, we assumed that dividing the songs in constant numbers of beats would provide equal information sums, so clips from a slow song would be longer than those from a fast song. To achieve this, we first applied a beat detection algorithm using the Librosa library[5]. Then, we set up a clip length in beats and divided the songs into clips. Finally, we compared the start time and the end time of each clip to the respective key information matrix mentioned above to add the key label to the clip. If a clip involved multiple keys, we would label the key that had the longest duration.

After we obtained the clips, we could further analyze them in the frequency domain. To obtain the temporal-spectral information, we applied the short-time Fourier transform (STFT) to the waveforms and got the spectrograms in terms of the frequency bins and the time frames. Given that the key is independent of the octave, that is, a note in different octaves plays the same role in the harmonic structure, we could load the frequencies into octave-independent classes. In this way, not only could the dimension of input data be diminished from thousands of frequency bins to several classes, but the relationship between the frequency bin widths and the notes could be linearized instead of being in logarithm as well. Therefore, both the efficiency and the validity of the input data would be boosted significantly.

For the STFT process, we had to choose a window length that guarantees a high enough frequency resolution so that the low notes could be intactly represented in corresponding pitch class bins in the later process. We were using a window length of 4096 samples, whereby the frequency resolution would be 10.67Hz given that the sample rate was 44100 Hz. We also applied a three-times zero padding to interpolate the spectrum so that the true frequency peaks could be more precisely located. The window we chose is Blackman-Harris window, which has the advantage of low sidelobe amplitude that would alleviate the problem of frequency leakage to other bins in later processes.[6] The hop size which determines the time resolution was 512 samples.

To load the keys to octave-independent classes, we used a method called harmonic pitch class profile (HPCP)[7]. Since the high frequencies above approximately 4000 Hz, within which reside almost merely the compositions of non-musical elements such as cymbals, breaths of voice, and ambiance textures, and below approximately 150 Hz, which are dominated by the kick drums, barely contain harmonic information, we limited our analysis within the frequency range between 150 Hz to 4000 Hz, whereby the computational cost was optimized and the non-harmonic interference was reduced. Next, we took the peaks of the spectrum within the range. There should be a threshold only above which we took the peaks, so that those due to the noise and frequency leakage could be eliminated. After that, we set up a reference frequency, for which we picked that of the middle C, 261.63 Hz.[8] Although an octave contains 12 semitones, we could use a larger number of bins to increase the accuracy. For example, we set 36 bins, so each bin corresponds to ⅓ semitones. Now, all the frequencies of the bins within the reference octave could be calculated from the reference frequency, and we could find the distance in semitones between all the peaks and the bins in the reference octave. Remainder with respect to 12 should be taken so that the distance would be octave-independent.

For the sake of further improving the accuracy and avoiding imprecise calibration, we applied weights, that is, each peak would contribute to not only one bin but a sequence of adjacent ones. The weight for the bins was computed from a single-cycle cosine square function, which takes only the center cycle of the cosine function and anywhere else is zero. The angle speed parameter of the cosine function was determined by a width we set up, for which we use 4/3 semitones, meaning that each peak will contribute to 4 bins. Finally, the contribution of each peak could be found by the weight and the square of the peak amplitude, and we summed them up for all peaks. More detailed procedure can be found in the codes in the appendix. After the procedure elaborated above, we obtained an HPCP matrix with the size of 36 pitch class bins times the number of time frames in the clip, which was the input to our model.

2.2 Data Processing before into Model

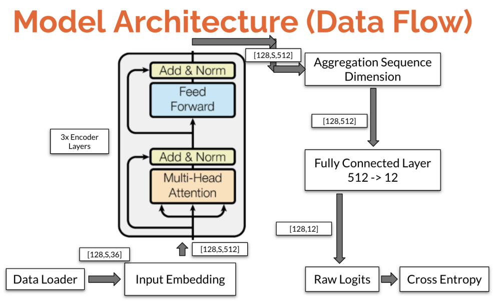
The data obtained from pre-processing are Excel files. In total there are 8077 Excel files as feature data, and 1 single Excel file as the label. Each feature data file is in the shape of [Sequence, 36-class], where sequence varies from 1700 to 2200, 36-class represent 36 keys - each key is divided into three sub keys for better classification, values in the matrix are normalized energy values.

The model will be developed and trained in Google Colab, so all feature files and the label file are loaded from Google Drive into Google Colab notebook by importing "os" and using "drive.mount". Then all files are read with panda's "read_excel()". After loading, features and labels are put into a DataSet class for DataLoader. After all data is put into the DataSet's class instance, the instance is split into 80% training data, 10% validation data, and 10% test data. All data is then loaded into the DataLoader, with a batch size of 128.

In order to load the data with different shapes, because the sequence varies, each feature file is zero-padded with "pad_sequence" from "torch.nn.utils.rnn".

2.3 Model

Transformer is used in this case. Inputs are fed into Input Embedding, to three Transformer Encoder Layers, to one Fully Connected Layer, to Aggregation, and eventually output as a vector of 12 classes, with raw logits as the value. The aggregation process aggregates output from the Fully Connected Layer of shape [batch_size, sequence, 12-class] into shape [batch_size, 12-class], by calculating the mean over the sequence.
For parameters, in this case, input dimension is 36 to represent 36 pitch classes, d_model is 512 as default, n_head is 8, number of encoder layers is 3, number of output classes is 12, and drop out is 0.1.
For more information about model structure and the data flow. See the following figure.



2.4 Optimizer and Criterion

Adam optimizer is used with a constant learning rate of 0.001. Cross Entropy Loss from torch.nn is utilized as the loss function, since our output are 12-class raw logits and the label value is the index of the ground truth class.

3. EXPERIMENT

3.1 Training

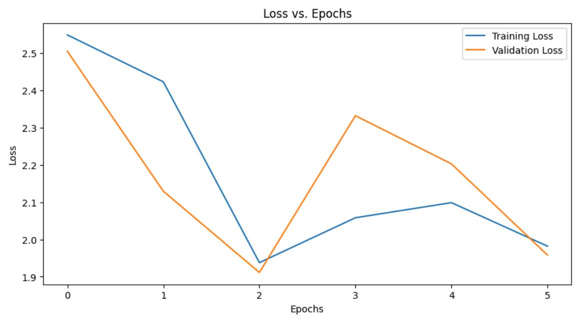
Due to the fact that the dataset is relatively large, the whole training process is divided into three parts: Test Run, 6-Epoch training, and 3-Epoch training. The Test Run took three hours, generating an average training loss of 3.4215, validation loss of 2.45, and accuracy of 12.62%. Here the validation loss is lower than the training loss because validation losses are calculated after the Test Run training. Then the model is saved and reloaded for the next training.

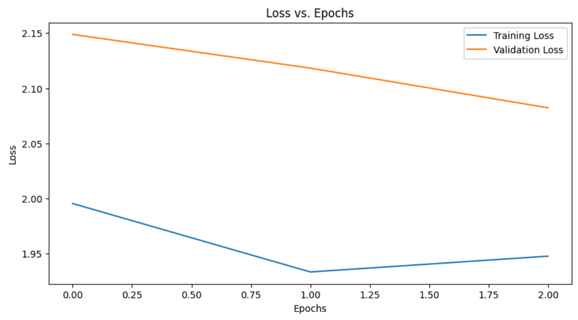
In the second training, 6 epochs are assigned, but this time, training time for each epoch is reduced to 1 hour. Result: training loss of 1.9818, validation loss of 1.9580 and accuracy of 24.38%.

In the third training, 3 epochs are assigned, and the training time required for one epoch is also around 1 hour. Result: training loss of 1.9477, validation loss of 2.0823 and accuracy of 25.87%.

Eventually our model reaches a state of around 25% accuracy, around 3 times the random guessing accuracy, which is around 8%.

Following are two training graphs with training loss and validation loss versus number of epochs.





After that, 25 more epochs are assigned, but the training does not really improve the model's accuracy. Training loss stays around 1.9 but the validation loss varies from 1.9 to 2.7. Reflection will present the potential problem.

3.2 Prediction

Although the model does not reach the expectation well, if the model was well-trained, we could use it to detect the keys in unseen songs. We first applied the same preprocessing methodology as that for the training data to compute the HPCPs for the clips. Next, we used the model to find the key probabilities array which contained the probabilities of the 12 keys that a clip belonged to. We first took the key with maximal probability for each clip. If we find that the key of a clip is different from that of the previous one, we assume there is a modulation happening between the two clips. Therefore, we took all the clips that had the fixed length with a hop size of 4 beats whose starting time was between the middle of the former clip and that of the current clip and used the model to detect the probability arrays. As the probability of the current key reaches a maxima, we could assume that the entire clip belonged to the key after the modulation, and the start time of the first clip that the maxima was reached should be the modulation point. Practically, the probability would fluctuate, so we set a threshold for the difference in the probability for each moving. When the difference was below the threshold, we could assume that the maxima was reached.

4. CONCLUSION

4.1 Reflection

There are several potential improvements that can be made to optimize the performance of our algorithm. First, non-harmonic elements, such as the percussion instruments, may affect the accuracy of the HPCP. The interference may be significant because the loudness is higher than that of other harmonic instruments. Possible solution includes applying a percussion eliminating algorithm before the generation of HPCP.

In addition, It is relatively a hard task to distinguish keys with dominant relationship, such as the C key and the G key, whose scales have only one different note: F sharp. This requires more training and a better model. Tonicization and mixture, which are temporary changes in key, is also another problem we need to consider, and it requires a more specific definition to differentiate between them and modulation.

The performance can also be improved by fine tuning the preprocessing and training parameters, such as the threshold for the peaks, pitch class weight, and learning rate. We should try more combinations of parameters so that the best one can be found.

For the Model part, the problem might come from not using positional encoding, aggregation of output sequence. Both of these two actions will lead to sequential information loss.

4.2 Future Work

For the model part, one way to improve is to first aggregate the sequence of each feature file, feed the whole song as a sequence into the encoder part with positional encoding, to the decoder part and generate the key in order. The second way is to add multiple fully connected layers with ReLU or Sigmoid activation function to increase the nonlinearity of the model.

5. TASK ASSIGNMENT

Boning Wang: Raise of topic, design of preprocessing method, design of prediction method, creation of dataset

Jiajun Wu: Design of data loading for training, Design of neural network architecture

Yichuan Wang: Fine tuning of parameters and program running, paper formatting

Jiajun Chen: Implement of algorithm, design of presentation slides

6. REFERENCE

[1]Y. Rou, H. Yang, H. Xu, and Y. Zhou, "Music Tonality Detection Based on Krumhansl-Schmuckler Profile," 2019.[1]Y. Rou, H. Yang, H. Xu, and Y. Zhou, "Music Tonality Detection Based on Krumhansl-Schmuckler Profile," 2019.

[2]S. Campbell, "Automatic Key Detection of Music Expert from Audio," McGill University, Aug. 2010.

[3]"Tunebat," Tunebat, [Online]. Available: https://www.tunebat.com. [Accessed: 7-April-2024]

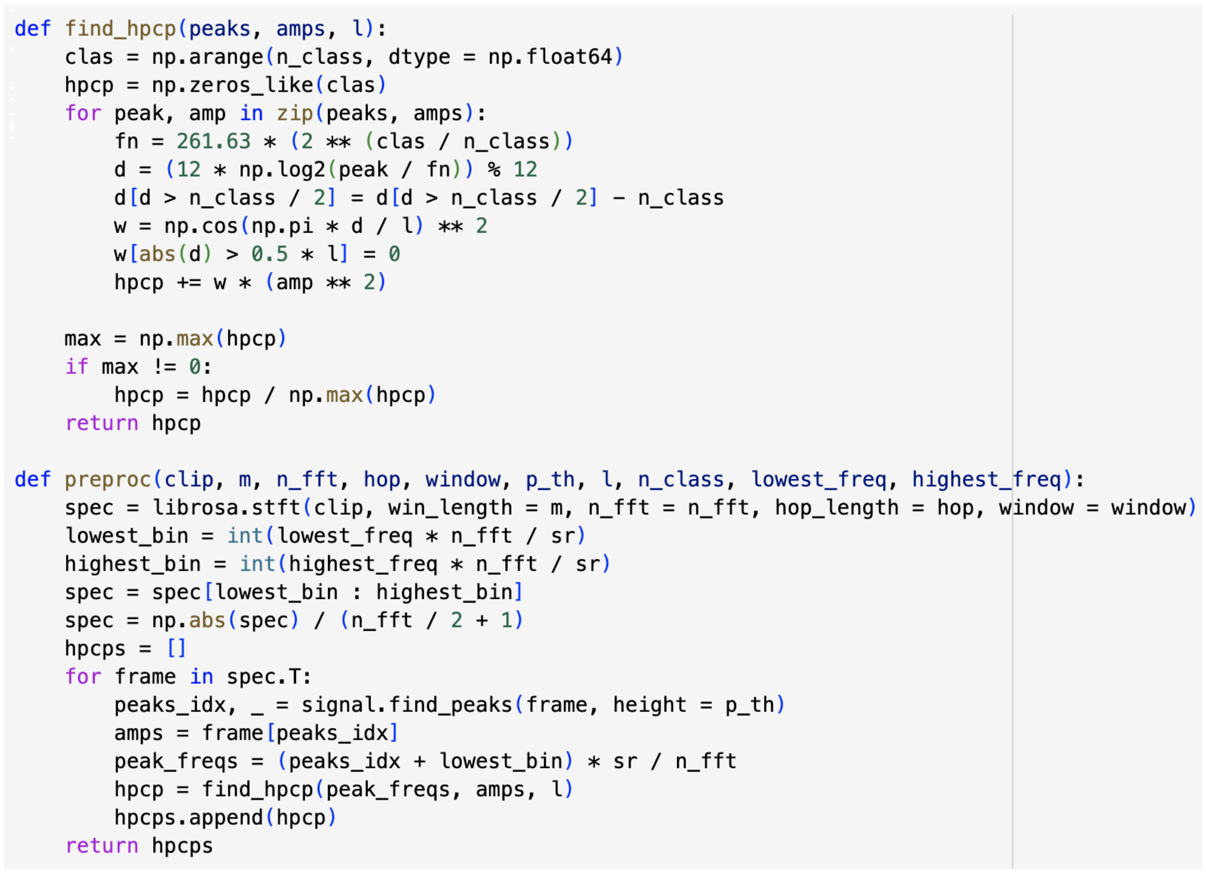
[4]"Song Key and BPM Finder," Vocal Remover, [Online]. Available:
https://vocalremover.org/key-bpm-finder.[Accessed: 7-April-2024]

[5]B. McFee, C. Raffel, D. Liang, D. P. W. Ellis, M. McVicar, E. Battenberg, and O. Nieto, "librosa: Audio and music signal analysis in Python," in Proc. 14th Python in Science Conf., 2015, pp. 18-25.

[6]MathWorks, "blackmanharris," [Online]. Available: https://www.mathworks.com/help/signal/ref/blackmanharris.html. [Accessed: 20-April-2024]

[7]E. Gómez, "Tonal Description of Polyphonic Audio for Music Content Processing," INFORMS Journal on Computing, University Pompeu Fabra, Aug. 2006.

[8]robrt60, "Ultimate Guide to Musical Frequencies," iDrumTune, May 8, 2021.



```
def find_hpcp(peaks, amps, l):
    clas = np.arange(n_class, dtype = np.float64)
    hpcp = np.zeros_like(clas)
    for peak, amp in zip(peaks, amps):
        fn = 261.63 * (2 ** (clas / n_class))
        d = (12 * np.log2(peak / fn)) % 12
        d[d > n_class / 2] = d[d > n_class / 2] - n_class
        w = np.cos(np.pi * d / l) ** 2
        w[abs(d) > 0.5 * l] = 0
        hpcp += w * (amp ** 2)

    max = np.max(hpcp)
    if max != 0:
        hpcp = hpcp / np.max(hpcp)
    return hpcp

def preproc(clip, m, n_fft, hop, window, p_th, l, n_class, lowest_freq, highest_freq):
    spec = librosa.stft(clip, win_length = m, n_fft = n_fft, hop_length = hop, window = window)
    lowest_bin = int(lowest_freq * n_fft / sr)
    highest_bin = int(highest_freq * n_fft / sr)
    spec = spec[lowest_bin : highest_bin]
    spec = np.abs(spec) / (n_fft / 2 + 1)
    hpcps = []
    for frame in spec.T:
        peaks_idx, _ = signal.find_peaks(frame, height = p_th)
        amps = frame[peaks_idx]
        peak_freqs = (peaks_idx + lowest_bin) * sr / n_fft
        hpcp = find_hpcp(peak_freqs, amps, l)
        hpcps.append(hpcp)
    return hpcps
```