Faculty of Natural and Mathematical Sciences
Department of Informatics

King's College London
Strand Campus, London,
United Kingdom

7CCSMPRJ

Individual Project Submission 2021/22

Name:	Hao Xu
Student Number:	21036108
Degree Programme:	Artificial Intelligence
Project Title:	Structural change for the automatic evaluation of music generation systems
Supervisor:	Jacopo de Berardinis
Word Count:	10021

[✓] I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: Hao Xu

Date: March 17, 2023

Department of Informatics
King's College London
United Kingdom

7CCSMPRJ Individual Project

Structural change for the automatic evaluation of music generation systems

Name: **Hao Xu**
Student Number: 21036108
Course: Artificial Intelligence

Supervisor: Jacopo de Berardinis

This dissertation is submitted for the degree of MSc in Artificial Intelligence.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Jacopo de Berardinis, for his patience and attentiveness. Although the course required two personal meetings, he gave much more resources and guidance, even at his busiest times. He was willing to share his PhD thesis, he was willing to help me set up the code environment, and he was willing to share the unpublished article and a part of his code. Without his generous help, I would not have been able to complete this project. Words alone cannot express my sincere gratitude. Secondly, I would like to acknowledge and give my warmest thanks to my parents, who supported me and gave me selfless love and encouragement from the other side of the world. Last but not least, I am extremely grateful to my friends Wei Lu, Yujie Xie and Xinyi Wang for taking care of me physically and mentally when I had COVID and for all the entertainment and support during the year.

Abstract

Automatic music generation has been a hotly debated topic in recent years. It did not emerge in the recent past but only to be pushed back into the limelight with the increasing level of computer hardware and technology. As machine learning becomes the future of mainstream computer technology, one after another architecture has been proposed, which makes the result of the generation more and more sophisticated. Despite this, machines cannot compose music in the same way that humans can. Due to the nature of architectural design, music generated by machines lacks long-term structure. This is one of the challenges faced by automatic composition today. However, analysing the structure of music requires both subjective judgment and the expertise of the people involved, which is time-consuming. Thus, an automatic way to evaluate music structure complexity is needed.

This thesis addresses this problem by introducing a method to automatically evaluate structural change in three different music features. First, a framework is introduced to measure the structural change in different time scales. This is followed by the validity of my implementation, which includes visualising the structural change by heatmap and a waveform visualisation for the structural change summary. This implementation is of significance to the problem of learning long-term dependencies in music generation, as it will give some insight into both short-term and long-term dependencies in a single music track. Then, to test the framework's functionality, an experiment is set up to apply the framework to different music genres. The framework was found to successfully distinguish music from various subsets. Finally, the thesis concludes with a discussion of the contribution of the work and future research directions.

Nomenclature

AMC	Automatic music composition
AMT	Automatic music transcription
ANNs	Artificial neural networks
AR	Autoregressive
BPTT	Back propagation through time
BSC	British computer society
FP	Fluctuation pattern
HMM	Hidden markov model
IET	The institution of engineering and technology
KS tests	Kolmogorov–Smirnov tests
LSTM	Long-short term memory networks
MIDI	Musical instrument digital interface
MIR	Music information retrieval
MSA	Music structure analysis
NLP	Natural language processing
PCA	Principle component analysis
RNNs	Recurrent neural networks
SSCS	Summary of structural change summary
SSM	Self-similarity matrix
STFT	Short-time fourier transform

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Project aims . . . 2
1.3 Structure of the thesis . . . 2

2 Background **4**
2.1 The challenge of automatic music composition . . . 4
2.2 Autoregressive models for music generation . . . 4
2.3 Music structure analysis . . . 7
2.4 Chroma, rhythm and timbre . . . 8

3 Methodology **11**
3.1 Overview . . . 11
3.2 Feature extraction . . . 11
3.3 Structural change formulation . . . 12
3.4 Window, summary and divergence functions . . . 13

4 Experiments and results **14**
4.1 Music dataset . . . 14
4.2 Preliminary validation of the framework . . . 19
4.3 Structural change of different music subsets . . . 23

5 Legal, social, ethical and professional issues **24**

6 Conclusion **28**
6.1 Critical analysis . . . 28
6.2 Future directions . . . 29

References **30**

A Appendix **35**
A.1 Source code overview . . . 35
A.2 Key files . . . 36
A.2.1 sonification.py . . . 36
A.2.2 noise_generation.py . . . 38
A.2.3 shuffle_beats.py . . . 39
A.2.4 divergence_function.py . . . 40
A.2.5 structural_change.py . . . 40
A.2.6 test.ipynb . . . 44
A.2.7 get_sc_change.ipynb . . . 45
A.2.8 bivariate_KDE.ipynb . . . 47
A.2.9 pairwise_comparison.ipynb . . . 51

List of Figures

1 Architecture of a traditional RNN 5
2 Different RNN sequence types 6
3 Chroma Feature 9
4 Rhythm in music: notes 10
5 Implementation flowchart 11
6 Simulated power spectral densities as a function of frequency for various colors of noise 18
7 Visualisation of structural change for the song 'Lucky' by Britney Spears, Mauch and Levy's implementation 20
8 Visualisation of structural change for the song 'Lucky' by Britney Spears, my implementation 21
9 structural change summary visualisation for (a) Pink noise (b) Around the world (c) Prelude in C Major 22
10 Pairwise statistical analysis of the music subsets for each principle component (yellow represents statistical difference) 25
11 Distribution of chroma SSCS for each subset after bivariate KDE 26
12 Distribution of rhythm SSCS for each subset after bivariate KDE 27
13 sonification_1 36
14 sonification_2 37
15 noise_generation 38
16 shuffle_beats 39
17 divergence_function 40
18 structural_change_1 40
19 structural_change_2 41
20 structural_change_3 42
21 structural_change_4 43
22 test 44
23 get_sc_change_1 45
24 get_sc_change_2 46
25 bivariate_KDE_1 47
26 bivariate_KDE_2 48
27 bivariate_KDE_3 49
28 bivariate_KDE_4 50
29 pairwise_comparison_1 51
30 pairwise_comparison_2 52
31 pairwise_comparison_3 53
32 pairwise_comparison_4 54
33 pairwise_comparison_5 55
34 pairwise_comparison_6 56
35 pairwise_comparison_7 57
36 pairwise_comparison_8 58

1 Introduction

1.1 Motivation

Deep learning methods have been widely and successfully used in computer vision. However, since the beginning of this century, its applications in the music field have been restricted for technical and hardware reasons. With more advanced techniques and enhanced process power, people have started to apply deep learning methods in the field of music. Despite using state-of-the-art deep learning architectures and training techniques, problems that are not encountered in computer vision still arise.

Automatic music composition (AMC) is a novel way of applying deep learning. Although machine composition has so far been unable to enter the real music market, it can be used to augment entertainment, for example, by producing music for video games. Right now, big companies such as Google and Spotify are still seeking the potential of machine composition.

However, most of the music produced by the AMC lacks long-term structure. It is therefore important to study musical structure, and the complexity of musical structure has been one of the main concerns of researchers. J.-P. Briot and F. Pachet [1] proposed four key challenges in AMC: control, for example, how to control the number of repeated notes. Structure, how to compose the music without giving a sense of direction. Creativity, how to prevent plagiarism. Finally, interactivity, how to deal with the automated single-step generation problems. When it comes to the last point, with the advent of Transformer [2] models, the generated music is proved to be better on a longer time scale than that generated by Long-short Term Memory Networks (LSTM) [3]. Despite this, most of the music produced today lacks an organised structure, as it does not convey a coherent and meaningful musical idea to the listeners, and there are problems with variations in the music such as connecting and repetition [4]. One reason for this is that when training the model, the audio music being processed was usually sampled at 44.1 kHz, which resulted in a huge long input sequence. Another reason is that other music attributes such as timbre are also considered compared to symbolic music, making composition considerably more difficult.

To deal with the problem mentioned above, it is essential to study the structural complexity of the music. Nevertheless, there lacks a standard systematic approach to objectively assessing the music structure complexity [5]. Most evaluation methods can be concluded as music modelling evaluation, statistical comparisons, composition evaluation, and listening tests [6]. Some of these methods of evaluation do not elaborate well on the complexities of generating musical structures, others are too personal and subjective [7].

In conclusion, there is a lack of a systematic and standardised method to assess the music structure complexity, which could be a potential limitation in that there is no consensus on assessing and comparing music generated from different models. Therefore, this project aims to find a way to objectively assess the music structural complexity and see the results when it applies to different music classes.

1.2 Project aims

As motivated by the previous section, the project seeks to find a way to evaluate the music structural complexity automatically. The key objectives of this project are:

- Implement a framework to objectively evaluate the music structural complexity;
- Test the validity of the framework;
- Establish a dataset with different music genres, where each subset has a different collection of songs;
- Apply the framework to the dataset, then perform a data analysis between different music subsets.

1.3 Structure of the thesis

The structure of this thesis is organised as follows:

Chapter2 (Background) introduces the fundamental concepts behind the generation of music and the structural analysis. In particular, this chapter will first illustrate the challenge of automatic music composition and introduce state-of-the-art architectures and models for generating music. As a music structure analysis project, previous methods will also be included in this chapter. Apart from this, this chapter will briefly mention three basic music features.

Chapter3 (Methodology) describes my method of assessing music structural complexity. First, how to extract the three basic musical features will be introduced. Then, the thesis will explain the structural change in a formulaic way. Finally, the chapter will conclude with an introduction to the window and an explanation of the 'summary', followed by talking about other divergence functions that can substitute the one currently used.

Chapter4 (Experiments and results) starts by describing the dataset. Then, to test the validity of my implementation, a graphical comparison will be introduced between the result of my framework and the older version written in C++ that inspired this project. In addition, the structural change summary is visualised to see if the results are as expected. After that, an experiment is set up to test my implementation in all the subsets. Finally, some statistical analysis will be carried out to show if there are some differences in these results across the distinct music groups.

Chapter5 (Legal, social, ethical and professional issues) gives a reasoned discussion about legal, social, ethical and professional issues within the context of my project.

Chapter6 (Conclusion) concludes this project's contribution and achievement and provides some ideas for possible future improvements.

2 Background

This chapter covers the essential elements that support this project. First, the challenge of automatic music composition is introduced. The chapter then presents recurrent neural networks, which is how three of my music subsets were generated. This section will also refer to modern state-of-the-art structures for generating music. Afterward, we focus on previous work on music structure analysis, as this has the potential to improve long-term music structure learning and allow for higher quality music to be produced. Finally, this chapter will introduce three basic musical features used in my framework.

2.1 The challenge of automatic music composition

Sequential modelling has always been an interesting topic in unsupervised learning. For example, the autoregressive (AR) model is capable of predicting future values based on previous observations, which can be of great value in the stock market. The general idea of sequence modelling is that the probability of each sequence/event $p(\mathbf{s})$ is the product of all the sequences/events that happened before. The formula is shown below:

$$p(\mathbf{s}) = \prod_t p(s_t|\mathbf{s}_{<t}), \tag{2.1}$$

where $\mathbf{s}_{<t} = (s_1, ..., s_{t-1})$ represents the history of prior sequences/events.

Modern natural language processing regards words or phrases as sequences/events as mentioned above, and is also one of the significant research directions in computer science today. When it comes to music, notes can be treated as sequences. Considering that audio files are complicated to represent as data, sequence models usually deal with symbolic music, in which music is stored in a notation-based format (e.g. MIDI, which stands for Musical Instrument Digital Interface). Like the general case of sequence modelling, music models can predict the following note, given all the notes played previously. The complexity of music leads to different levels of difficulty in prediction tasks. For instance, the diversity of chords and instruments can increase the difficulty of prediction to some extent [6]. Music modelling can be applied in many areas, such as automatic music transcription (AMT) [8], music recommendation systems [9], and other applications that could help composers to augment their creativity. However, as sequence modelling is time-dependent, the question of how to 'remember' old sequences becomes a considerable problem. Therefore, generating music with a long-term structure is a challenge for AMC. Music structure analysis has the potential to help solve this problem, which will be mentioned in 2.3.

2.2 Autoregressive models for music generation

As mentioned above, since the sequence model and the music model are complementary, improving the predictive capabilities of the sequence models will significantly improve the music models. This brings us to artificial neural networks (ANNs) [10]. Inspired by the neurons and nerves of the human brain, the ANNs are considered one of the best predictive models in the world, and with them comes a range of networks derived from ANNs, all with good results in their respective fields. In fact, one specific type of ANN, called recurrent neural networks (RNNs) [11], allows the networks to process sequential data or time series data. This specific architecture benefits solving temporal problems, such as translation, natural language processing (NLP) etc. Unlike traditional feedforward neural networks, RNNs utilise the prior inputs to influence the inputs and outputs at the current state, while traditional ANNs assume those inputs and outputs independently. Figure 1 shows the architecture of a traditional RNN, where a^t stands for the output from the previous time step, x^t represents the input of the current state, and y^t indicates the output from the current state, $t \in 0, 1, 2, \ldots$ which stands for the time sequence.

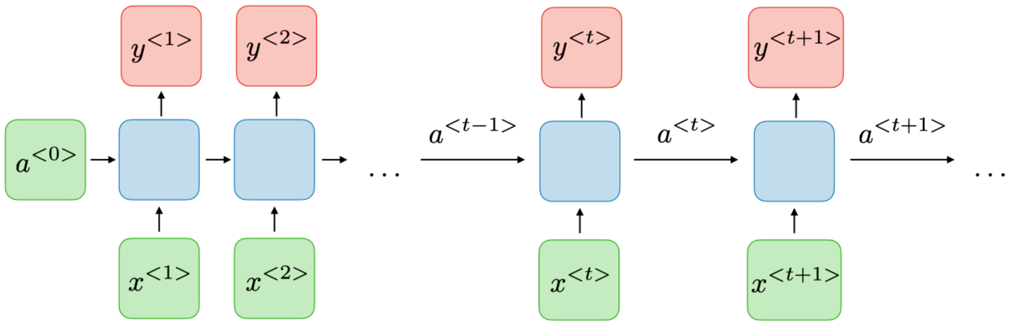


Figure 1: Architecture of a traditional RNN

RNNs use back propagation through time (BPTT) [12] to determine the gradient, which is also slightly different compared to the back propagation of traditional ANNs. In short, BPTT sums error terms at each time step, whereas traditional ANNs do not since they do not share the parameters across each layer. This also results in RNNs being more prone to have gradient exploding/vanishing problems than ANNs. Feedforward networks map an input to an output, and since the visualisation of RNNs is shown above, it is not difficult to see that RNNs do not have this limitation. Instead, their inputs and outputs can vary in length, and different types of RNNs are used for various use cases. For example, music generation, sentiment classification and machine translation. The different types of RNNs are often expressed in Figure 2.

Many RNN architectures have been developed nowadays, the most classic being Bidirectional recurrent neural networks (BRNN) [13] and Gated recurrent units (GRUs) [14]. Long short-term memory (LSTM) is an architeture that played a dominant role in music modelling and generation in the last few years. Though LSTM alleviates the gradient exploding and vanishing problems, which the vanilla RNN has, experiments and the nature of design have demonstrated that this architecture can still lose information if a long

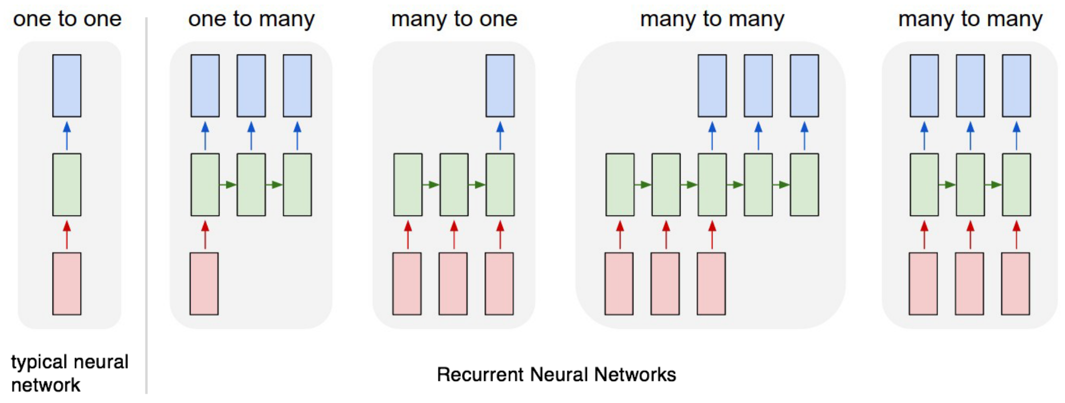


Figure 2: Different RNN sequence types

sequence is being processed. Recently, self-attention transformers successfully tackled the problem and innovated the NLP domain. With positional encoding or embedding, the entire sentence is processed. In this project, some of the music in my dataset was generated by different LSTMs, namely the lookback LSTM and the attention LSTM [15]. Other methods of automatically composing music are described below.

The state-of-the-art method to generate music is developed by the Google Magenta team [16], which mimics human behaviour of revisiting and rewriting songs using Gibbs sampling. A random subset of music is removed in each step, and the removed part is refilled with a sample generated from the probability distribution model. This process is repeated over time to form new music. The result shows that the samples generated using block Gibbs yield better than those generated by traditional ancestral sampling.

To deal with the short-term generation problem, the same team proposed Music Transformer [17]. This particular transformer replaced the pairwise distance [18] to store relative position information by relative global attention, which is a memory-efficient way to address long sequences. This modified structure enables one minute long composition and shows a good performance on Maestro dataset [19]. Enlightened by variational autoencoder (VAE) [20], a hierarchical decoder [21] was proposed. This decoder addresses the short-term problem by generating embeddings for each subsequent input. Alternatively, Musenet [22] utilised sparse kernel [23] to train models on text-based music notations. The long-short term universal transformer (LSTUT) [24] combined the transformer and RNN, outperforms models that can learn features at different time scales. Recent work from Wu et al. [25] and Huang and Yang [26] both applied Transformer-XL [27] to capture the long term dependencies. Inspired by the structures mentioned above, Muhamed et al. [28] provided a way to generate symbolic music with Generative Adversarial Networks (GANs) [29], which made use of the attribute that the discriminator can judge whether the music is real or fake. The network consists of a generator, Transformer-XL, and a discriminator, Bidirectional Encoder Representations from Transformers (BERT) [30]. The result shows that the quality of music generated by this Transformer-GANs is better than that generated by Musenet, both by humans and a self-proposed discriminative metric evaluation.

2.3 Music structure analysis

The objective of music structure analysis (MSA) is decomposition or segmentation, breaking down a given musical performance into patterns or temporal units corresponding to musical parts, then grouping these sections into musically meaningful categories [4]. From this definition, music structure analysis has two tasks. One is to detect the boundaries of a piece of music, and the other is to group these pieces after segmentation and detect the presence of some intrinsic relationships and common features.

To talk about the first problem–segmentation, three principles were identified in 2010 [31], which are homogeneity, which segments the music through repeated fragments; novelty, which segments the music through sudden changes in time, such as key change; and repetition, which segments the music through the repetition of sections. Then, in 2011, another principle was presented as regularity [32], which segments the music by a degree of regularity. For instance, two equally labelled sections are often separated by an integer ratio number of beats. Music can be split into different pieces under different criteria. However, even under the same criteria, the separated music sections are also not identical because this process is highly listener-dependent and time-consuming.

For computer scientists, discovering an algorithm to segment music automatically can be very challenging. Thus, music information retrieval (MIR) is a key focus for related researchers. A checkerboard kernel technique is beneficial based on the homogeneity principle described above, which uses a kernel with a tessellated structure that convolves on the main diagonal of the self-similarity matrix (SSM) [33], resulting in a novel curve highlighting abrupt changes in the selected musical features, from which boundaries can be extracted by identifying the prominent peaks. A more recent technique is based on the principle of homogeneity and repetition, namely structural features [34], which can be extracted from lag matrix [35], a more sophisticated approach based on homogeneity. This way, a novelty curve can also be produced to extract the boundaries. Many approaches to music segmentation have been inspired by the methods mentioned above. Recently, an architecture [36] combines deep convolutional neural networks and checkerboard kernel techniques to achieve state-of-the-art results.

The second sub-task, as described above, is known as structural grouping. Quantifying the segment similarities and grouping them together can be problematic. As a variation of the nearest neighbour search, Wang et al. [37] propose a supervised approach which uses Gaussian mixture models. Another approach [38] uses a variation of the nearest neighbour search algorithm, which aims to seek some similarity of timbre features on a multi-dimensional Gaussian space. In 2014, Nieto and Bello proposed a method that projects the harmonic features into 2D Fourier Magnitude Coefficients (2DFMC) [39], allowing further labelling of the resulting fragments with k-means. Finally, Serrà et al. propose a method [34] that combines structural features and cover song recognition techniques [40] to achieve a state-of-the-art estimating boundaries effect.

Some methods, however, can perform both boundary detection and structural grouping so that if an error occurs in the first step of MSA, the following steps will not be affected, which are more robust to use. Levy and Sandler propose an approach which uses a hidden Markov model (HMM) [41], where audio frames can be encoded as states; therefore, both boundaries and labels can be obtained. Another work from Paulus and Klapuri [42] calculates the likelihood of a specific segment. The search space is greatly reduced by adopting a greedy search algorithm. McFee and Ellis [43] enhance the repetition in music pieces by using spectral clustering, and by doing so, smaller segments can also be discovered. Interestingly, this is the first time that hierarchical segmentation has drawn the public's attention. In the last five years, Sargent et al. [44] have achieved this by limiting the size of the final segmentation, and Shibata et al. [45] have done so by using a Bayesian approach.

In summary, many MSA algorithms have been proposed over the last two decades, and computational efficiency has improved considerably. However, with this have come challenges. According to Nieto et al. [7], three key challenges remain unsolved and should be addressed in the next decade, which are subjectivity, where different people may have different views on segments of the same song; ambiguity, where one person may agree on multiple interpretations; and hierarchy, where the structure may exist on multiple time scales. In any case, MSA is still of high research value in the future.

2.4 Chroma, rhythm and timbre

The chroma feature usually represents twelve different classes of pitches and is a powerful means of analysing music. The basic observation is that if two musical pitches are one octave apart, humans will perceive them as similar in colour. Based on this observation, a pitch can be divided into two parts: tone height and chroma. Assuming an equal-tempered scale, chroma values can be represented by the following set: {C, C$^\sharp$, D, D$^\sharp$, E, F, F$^\sharp$, G, G$^\sharp$, A, A$^\sharp$, B}. It is important to note that some literature will notate $C\sharp$ as $D\flat$, but acoustically, they both represent sonic vibrations with a frequency of 311.13 Hz in the tuning of a standard instrument. Figure 3 shows the chroma features of a C-major scale in different musical representations. Subfigure (a) shows the symbolic music representation, i.e. the notes are recorded on the pentatonic scale. (b) shows the chromagram obtained by the score, and (d) represents the chromagram obtained from a real audio recording. (c) is the audio representation of the C-major scale played on a piano.

Chroma is a characteristic associated with music harmony, regardless of how the timbre changes. This is why almost every chord recognition program relies on chroma representation. Indeed, chroma features have already played a crucial role in MIR tasks, such as cover song identification [46] and audio matching [47]. MIR tasks often require processing music data, which often involves converting audio recordings into chromagrams. The most common way to do this is Short-time Fourier transform (STFT) [48].

The simplest definition of music rhythm is the arrangement of notes of different lengths,

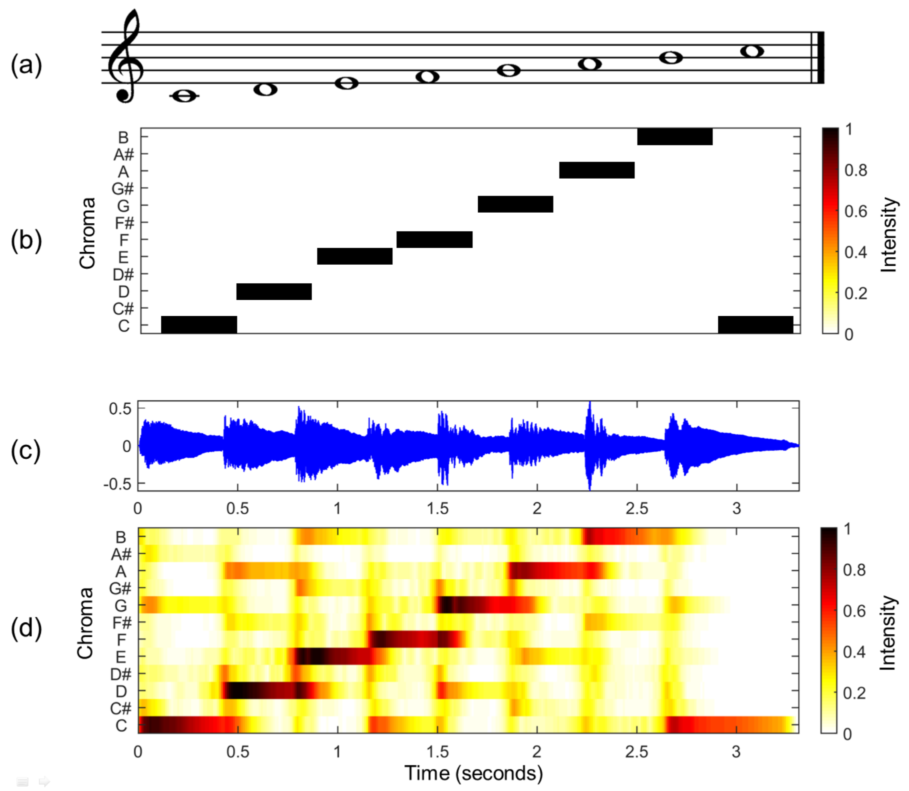


Figure 3: Chroma Feature

measured in terms of the length of time the notes sound in a piece of music. For most people, rhythm is the speed of the music or the length of the notes. In a narrow sense, indeed, several types of notes and rests are clearly defined in Western music as shown in Figure 4. However, in a broad sense, beat, tempo, and ostinato are all included in rhythm. Different lengths of notes and how they are arranged can give people a different "feel" for the music. Therefore, rhythm is the bone of music. As rhythm is a field that has a tiny intersection with computer science, the literature on rhythm extraction is sparse. Leve et al. [49] use dynamic programming to prove that long and intensity extractions are good choices for rhythm extraction.

Timbre, another music feature known as tone quality and tone colour, often distinguishes

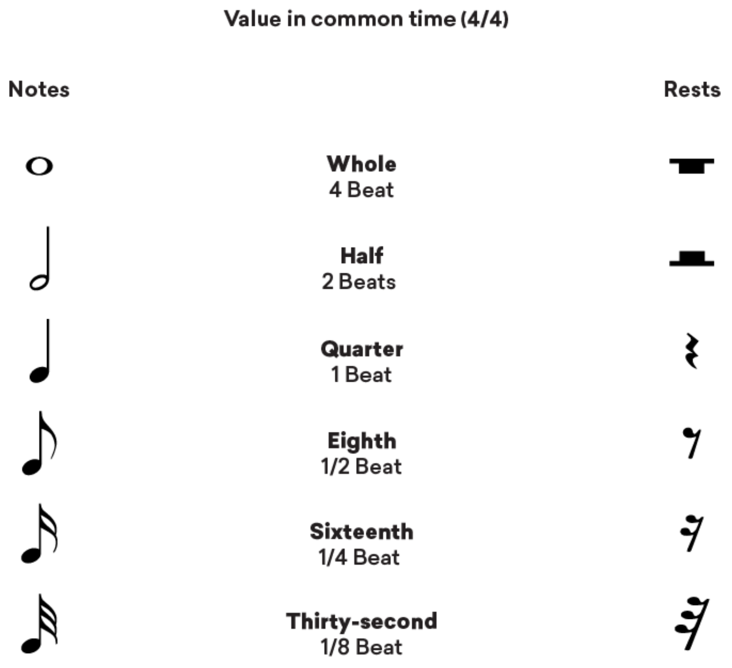


Figure 4: Rhythm in music: notes

the different types of sound production, such as musical instruments. In short, timbre makes one person's voice different from another's and makes one instrument sound unlike another. The physical characteristics of sound that determine timbre perception include frequency spectrum and envelope. The former is the frequencies of some signal along a time series and is often used in research to extract timbre data. The latter describes how a sound varies over time. It may relate to elements such as amplitude (volume), frequency (using filters) or pitch.

In conclusion, music contains chroma, rhythm, timbre, and several other features. Each of these features determines the difference in the music played. This project will only focus on these three features to extract music information.

3 Methodology

In this chapter, the idea behind the framework is explained. Specifically, the first section gives an overview of the framework. Then, the second section illustrates how chroma, rhythm, and timbre feature vectors are extracted from MIDI files. The third section formulates a mathematical description of the structural change. Finally, hyper-parameters that affect the result are presented.

3.1 Overview

This section gives an overview of my implementation, which covers chapter 3 and chapter 4.

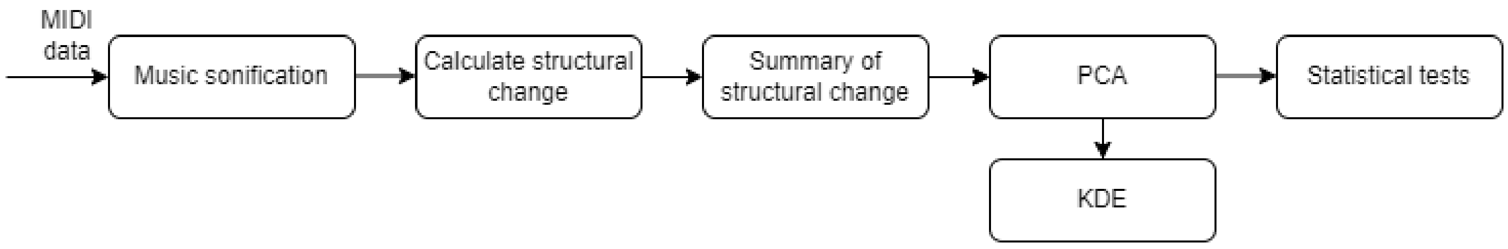


Figure 5: Implementation flowchart

As shown in Figure 5, MIDI data is the input in my framework. With music sonification, structural change can be computed. This part will be illustrated in chapter 3. Then the dimension of structural change is reduced by calculating a summary of the structural change followed by PCA. Kernel density estimation is used to plot the principle components. Finally, some statistical tests are performed between different music subsets to distinguish if they are significantly different. This way, a new music track can be summarised similarly until PCA. The structural complexity of this track can be obtained by observing the location of the principle components in the structural complexity plane made by my dataset. This approach gives a new way of estimating structural complexity and thus contributes to some extent to the development of the MSA field.

3.2 Feature extraction

From a raw MIDI file, the first step is synthesising an audio file using FluidSynth and the FR3 General-MIDI soundfont [50]. All the songs are sampled at 44100 kHz. A modern Python library Librosa [51] provides practical implementations of a large number of common functions used throughout MIR. The following implementations of feature extraction all use this library.

Chroma features are extracted using STFT, which transforms audio into a chromagram from a waveform or power spectrogram.

The rhythm feature is calculated by transforming the audio sequence into an STFT signal and taking its absolute value. Then, a transformation is performed by converting this amplitude spectrogram to a dB-scaled spectrogram. Finally, the conversion is made to a one-dimensional vector to mitigate the effects of timbre. STFT is calculated using Hamming-windowed segments to summarise a window every three seconds with a step of 44100 samples (1 second). These windows are further divided into 256 frames with a length of 512 samples, as shown by N_FFT in the code.

Timbre features are extracted by utilising the Mel-spectrum, which firstly performs a discrete Fourier transform on the audio signal, taking the logarithm, and then maps the result onto spaced bins to human perception of pitch. The hop size is set to one second. The window size is set to three seconds, and the same segmentation method is used to match the window and step size in the rhythm feature extraction. For simplicity, thirty-six Mel-frequency bins are used in this project.

3.3 Structural change formulation

After feature extraction, an m-dimensional with N frames thus $m \times N$ feature vector can be obtained for each music feature. For example, since the chroma feature has twelve pitch classes, the m for the chroma feature vector is 12. N is a number related to audio length, window type, size, and hop length.

For each music track, the idea is to compute a structural change feature for each frame i, i.e. to compare the features in the k frames to the right and the features in the k frames to the left. For convenience, let us define "summary" as the features in the k frames to the right or the left. There can be various ways to calculate summaries. In this implementation, the summary is simply the average feature values in k frames.

The structural change is computed by using a non-negative divergence function, which maps two $m \times w$ dimension summary feature vectors into a non-negative real value, i.e. $\mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}^+$. This real value should conclude the structural change in all dimensions for each frame. Different window sizes are applied to the same music feature vector. With n being the window size, we can calculate an n-dimensional structural change vector. As a result, a single frame in this structural change feature vector should be a $n \times 1$ vector. Specifically, this vector can be denoted as $v_i = (v_i^1, ..., v_i^n)$, where i is the frame number. By using a divergence function mentioned above, a frame feature can be represented as

$$v_i^j = \begin{cases} d(s_{[i-w_j+1:i-1]}, s_{[i:i+w_j]}) & \text{if } w_j < i < N - w_j + 1 \\ 0 & \text{otherwise} \end{cases}, \tag{3.1}$$

where w_j is the j^{th} window size, s is the summary, i is the current frame number, and N is the total frame number. The window size can be set to some arbitrary numbers, depending on the task type and computation resources. For convenience, in this project, the window size is set as a power of two:

$$w_j = 2^{j-1}. \tag{3.2}$$

Although the method for calculating structural change features is simple, it requires significant computational resources. For example, in my implementation, a summary is computed by the mean of a feature vector spanned over the window length w_j, which requires $w_j - 1$ additions. Since a frame value needs two windows on the left and the right, and n windows need to be considered, $2n(W_j - 1)$ additions are required for each frame. Assuming that W is the average window length, the total additions of a given music feature is $2mnN(W-1)$, where m is the dimension of that feature after feature extraction, and N is the total frame number. Apart from this, the division used in each convergence function or other mathematical manipulations such as power and the square root is not included in the calculations here.

When the summary is calculated by the mean of window-size vectors, another approach can be used to mitigate this addition problem by calculating a cumulative matrix $C = (c_1, ..., c_N)$:

$$c_i = \sum_{i'=1}^{i} x_{i'}, \tag{3.3}$$

where x_1 is the feature value of the first frame. In this way, the value of each frame can be pre-calculated as the value of the raw feature of this frame plus the cumulative value of the previous frame. Although the size of the cumulative matrix is the same as the raw feature matrix, the number of additions is reduced to $2mnN$ since all the additions in the window are eliminated.

3.4 Window, summary and divergence functions

In this project, power-of-two window sizes are chosen. Since chroma features might not significantly change in the short term, a longer window length is set to alleviate this problem. Specifically, for rhythm and timbre features, $j = 1, ..., 6$, and for chroma features, $j = 3, ..., 8$.

The summary is calculated by averaging all frames within a given window length.

A divergence function is needed to compare two summaries to get each frame's structural change value. A symmetrized and smoothed version of the Kullback–Leibler divergence, Jenson-Shannon divergence, is applied in my implementation. For two summary vectors s_1 and s_2, Jenson-Shannon divergence can be computed by:

$$d(s_1, s_2) = \frac{KL(s_1||M) + KL(s_2||M)}{2}, \tag{3.4}$$

where M is the average of s_1 and s_2, and Kullback–Leibler divergence function is given by:

$$KL(x||y) = \sum_{i=1}^{n} x_i log(\frac{x_i}{y_i}), \tag{3.5}$$

4 Experiments and results

In this chapter, experiments and results are shown by firstly introduce the music dataset. Then, the differences between my framework and one written in C++ ten years ago are described. The aim is to ensure that my implementation achieves the same results or better results compared to his work. Finally, I demonstrate that my framework distinguishes well between different subsets of music and may therefore be of some relevance to solving the problem of short-term music generation.

4.1 Music dataset

The first step is to create a dataset that contains music audio with different levels of structural complexity. Overall, four types of subsets are contained: *real music* with high complexity, composed by real humans; *computer-generated music* with medium structural complexity, generated by neural networks; *random music* with low structural complexity, generated by a real music subset with beats swapping; and *noise* with no structural complexity. Whether the implementation can distinguish these four subsets with different music structural complexity can prove the effectiveness of my work. Thus, this experiment aims to show if there are statistical differences between those subsets. Another topic is to study the feature relationships between those subsets, for example, whether computer-generated and random music would lie in the middle of an upper bound and a lower bound are defined for real music and noise, respectively.

The dataset contains six MIDI folders. Each folder includes one hundred MIDI files. Different folders contain different music types with various genres: pop, jazz, classic (piano), and music generated by basic RNN, lookback RNN, and attention RNN. This MIDI format takes up less storage space than audio files. Hence a sonification step is required. The dataset also includes ten waveform folders. Six of them are the audio files synthesised from the MIDI folders described above by FluidSynth. Another four subsets are white, pink, brownian noise, and random music, respectively. In total, ten subsets, including 1000 audio files, make up the entire dataset, allowing me to test the robustness and generalisation of my implementation. To be noted, this project only provides six MIDI folders. Users need to synthesise music by themselves using the python files within the source code.

The sonification of the audio precedes the calculation of structural changes. It means a transformation from a piece of symbolic music to audio waveform music. As MIDI files for pop and jazz music are extracted from the internet, 100 songs within each sub-folder are of unequal length. To equalise the length with the other songs, I shrink the length of pop music to 3 minutes and the jazz music to 3 minutes and a half, as the pop music is usually over 4 minutes, but some of the jazz music is under 2 minutes and 45 seconds. The specific sonification details for these four subsets are explained below.

- **Real music**

  This subset contains three genres of human-composed music: pop, classic, and jazz. Classic music is a partition of Pianomidi [52], which is a well-known dataset containing hundreds of symbolic classical music composed by various composers. There are no readily available pop and jazz datasets online to use as a reference. Therefore, these two datasets were created by hand. The former is from a website called "freemidi" [53] and the latter from a website called "midiworld" [54]. The expectation of this subset is to represent the highest musical structural complexity, with sections, phrases, and motifs representing long-term, medium-term, and short-term structures, respectively.

- **Computer-generated music**

  This subset contains music generated by three different state-of-the-art neural networks: the *Basic RNN*, the *Lookback RNN*, and the *Attention RNN* [15]. Lookback RNN and the Attention RNN aim to improve the ability to learn the long-term structure and, as a result, the music produced by them has a more complex structural complexity. In addition, all three models are trained by the same symbolic music dataset, encoded in the same way, and use the same optimisation method. Thus, some learned parameters can be compared, and the fairness of the comparison can be ensured. The following parts are the descriptions of three RNN networks.

  The Basic RNN is a vanilla LSTM model that takes a one-hot vector of the previous token as input, with the label of the next token.

  The Lookback RNN improves the Basic RNN by following:

  - The input not only includes the previous token vector, but also adds the token from one and two bars ago, which allows the model to recognise more patterns.
  - The input also includes some signals indicating whether the last token is repeated from the tokens 1 and 2 bars ago. In this way, the training of the model can have information about whether a token is newly created or just something that repeats an existing melody, allowing the model to recognise repetitive or non-repetitive patterns more easily.

  - Another input is to add a binary representation of the current position within the measure, assuming 4/4 time. Each row is one of the beat inputs, and each column represents one time step. The true implementation uses a binary representation of $\{0, 1\}$ instead of $\{-1, 1\}$.

  The Lookback RNN gets its name from the labels for training data that represent a step to be repeated 2 bars ago or 1 bar ago. This step is labelled as a specific melody token if it is neither. In this way, the Lookback RNN reduces the complexity of the information the model must learn to represent.

  The Attention RNN can learn even a longer-term structure using attention [55], which is first developed to increase the performance of the encoder-decoder RNN model. With this mechanism, the model can access previous information without storing the relative music information in the RNN cell's state. Specifically, in this case, the LSTM cell's state. The model differs from the originally proposed model in terms of output. Instead of an encoder-decoder structure, this model adds an attention mechanism by 'looking' at the outputs of the last n steps when generating the output of the current step.

  The expectation of this subset is to have medium structural complexity, which stands between real and random music. Since the order of the three models described above illustrates the complexity of the models, it reveals the fact that the structural complexity of the generated songs also increases, starting from the basic RNN model.

- **Random music**

  This subset includes 100 songs that are artificially generated by randomly shuffling the beat-aggregated features. Specifically, there is a shuffle beats Python file in the source code to achieve this task, and the code should be easy to understand. First, the input and output folders are given, which can also be defined by the user. Here, I use real classic music as the input file, as each beat and note can be clearly heard from this subset. Then, for each track, the Librosa library encodes the audio into an array of features. This feature should have a length of $44100 \times L$, where L represents the length of the audio. The operation on features starts by estimating beats and then shuffling blocks of feature vectors between consecutive beats. In this way, given two beat numbers B_i, B_j in the beat set B = (B_0,...,B_{N-1}), where N is the number of beats detected in the song, there is a uniform probability to swap feature vectors X[$B_i \times SR$, $B_{i+1} \times SR$] with feature vectors X[$B_j \times SR$, $B_{j+1} \times SR$]. X here represents the audio feature vector with a length of $44100 \times L$, i, j are chosen by random seeds, and if two seeds are the same, another seed should be chosen to make the two seeds different. The SR is the sample rate of the original classic audio and can be detected during the step of detecting the beat. This is some sort of beat-level scrambling, which can destroy all structural information beyond the beat level. By swapping the beat feature and keeping the note, the expectation of the subset is to have the minimal structure within these music pieces.

- **Noise**

  This subset contains 300 noise audio files, with 100 white noise, 100 pink noise, and 100 brownian noise.

  The name of white noise is inspired by white light, which combines the lights of different wavelengths. White noise is actually a signal which has equal intensity at all frequencies. For instance, within a white noise signal, the audio power between 40 Hz and 60 Hz is equal to the audio power between 400 Hz and 420 Hz.

  However, the pink noise frequency spectrum is on a logarithmic scale. This means the audio power between 40 Hz and 60 Hz can have the same power between 4000 Hz and 6000 Hz. Interestingly, human perception of a double frequency (an octave) is the same regardless of the actual frequency. This noise is usually used as a reference signal in audio engineering. Unlike uniformly distributed white noise, the spectral power density of pink noise decreases by 3.01 dB per octave, proportional to $\frac{1}{f}$. For this reason, pink noise is sometimes called "$\frac{1}{f}$ noise".

  Brownian noise, also known as brown noise, is a signal whose power density decreases by 6.02 dB per octave (proportional to $\frac{1}{f^2}$). The name "brownian/brown" is derived from the Brownian motion. Also, this noise has another name, red noise, as pink is a colour between red and white.

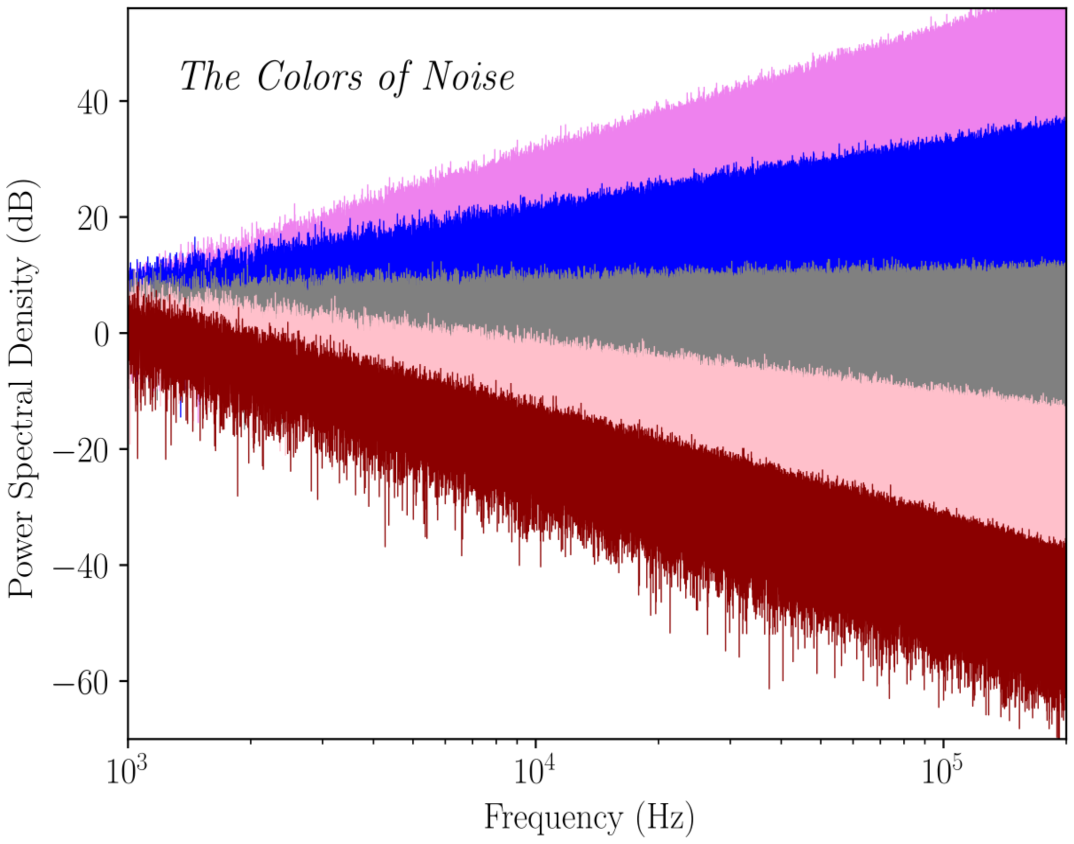


Figure 6: Simulated power spectral densities as a function of frequency for various colors of noise

Figure 6 simulates power spectral densities as a frequency function for various noise colours. From top to bottom are violet, blue, white, pink, and brownian noise. The power spectral density is arbitrarily normalised so that the values of the spectrum are approximately equal to around 1 kHz. The sonification of three types of noise is given in the Python file *noise_generation*, which uses a "colorednoise" library. In this file, a for loop is used to generate a given number of noise waveform files. Before that, some parameters need to be defined: the exponent of $\frac{1}{f}$, the sample rate, and the samples. The exponent is 0 for white noise, 1 for pink noise, and 2 for brownian noise. In this project, the sample rate is set to 44100 Hz, and the sample number is 44100×180, as each audio track takes 3 minutes. The expectation for this subset is to have minimal or zero structural complexity, which should be the bottom line of the comparison.

4.2 Preliminary validation of the framework

The validity of my framework is demonstrated in two parts. The first part compares my structural change with the results of the visualisation of a ten-year-old paper, "Structural change on multiple time scales as a correlate on musical complexity" [56]. The second part shows the expected and actual results of the summary of structural change features.

To illustrate the first part, the comparison is made under the heat map. Specifically, the heat map shows the feature vectors after the structural changes are computed. The heat map for each music feature is a six-dimensional feature vector, where each dimension represents a window size. Each value within that dimension is calculated from the divergence functions of the two summaries on the left and right. This is the first and most straightforward way to visualise a feature vector of structural change. The paper above gives an example of the visualisation of the structural change in the three basis music features for the song 'Lucky', performed by Britney Spears. Nevertheless, to be noted, there are some differences in feature extraction between my implementation and theirs. Their implementation for chroma feature extraction uses a Vamp plugin implementation written in C++, which firstly uses the discrete Fourier transformation to get a spectrogram. This spectrogram maps the spectral frame to a pitch space via linear transformation. Then, the chroma feature vectors are computed using weighted sums of the adjusted pitch space spectral bins. My implementation is just a chromagram representation from a waveform or power spectrogram derived from [57]. This method uses chroma analysis and chroma synthesis. The former uses instantaneous frequency estimates from the spectrogram to obtain high-resolution chroma profiles after generating a sequence of short-time chroma frames. The latter turns the former chroma representation back into an audio signal by using the 12 chroma values to modulate 12 sinusoids, covering one octave. Their implementation uses fluctuation patterns (FP) to extract the rhythm features, and mine uses STFT. Although both implementations use Hamming window segmentation, the length of the windows and the number of hops are the same, plus the same addition method is used to add the features from all bands to one band. There is also the question of whether STFT can be used to detect rhythms, which is partly included in the final chapter on future directions. The extraction of timbres is the same for both implementations.

Figure 7 shows an example of their visualisation, carried out to analyse the song "Lucky", performed by Britney Spears. It is pretty evident from the image that the horizontal axis represents the time and the vertical axis represents the six window sizes. The window length is increasing from bottom to up because for larger window length, there should be no structural change in some of the first frames, as it cannot be calculated. For better comparison, I also did this with my visualisation, as shown in Figure 8. It is clear that there are two drum stops before the first chorus and the first bridge in their implementation, marked by **a**. In my implementation, one of the drum stops is easy to spot, but the other one seems to be vague on the time scale, whether it can be **b** or the second white box in the rhythm structural change in Figure 8. Label **c** marks a section with very little musical movement: no real chord changes but plenty of vocal changes, including spoken vocal excerpts. Thus, it reflects in low chroma structural change but high in timbre structural change. Label **d** has an altered key change that heralds the arrival of the chorus. In my implementation, those changes in **c** and **d** can be clearly detected, and even the bridge boundaries can be spotted by seeing two clear peaks of timbre structural change. This could indicate that the second white box in rhythm structural change means the beginning of the second chorus, and label **b** could be the second drum stop. In conclusion, my implementation greatly reproduces the visualisation of the structural change of the framework ten years ago.

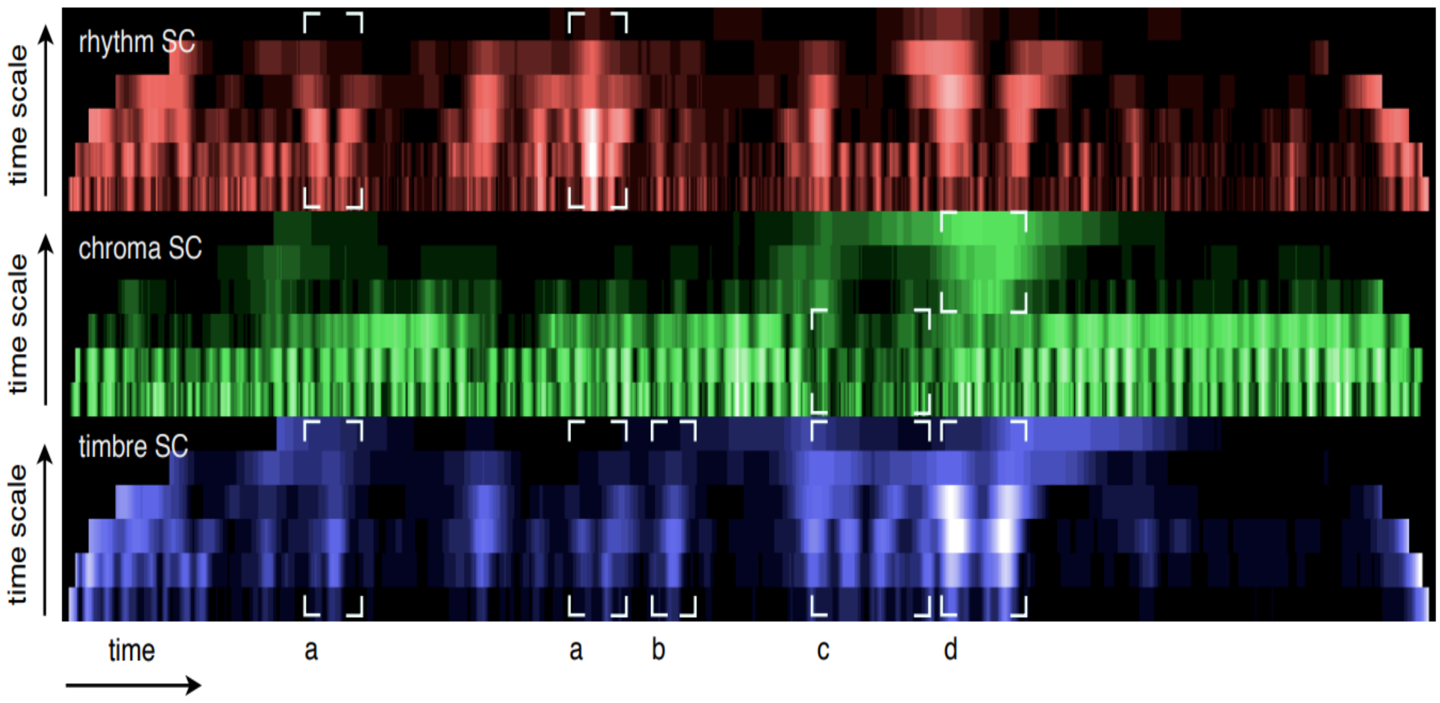


Figure 7: Visualisation of structural change for the song 'Lucky' by Britney Spears, Mauch and Levy's implementation

Secondly, after calculating the structural change, because three music features each have a multi-dimension feature vector, which is of a long sequence, a one-dimensional feature vector can be obtained by taking the average of all frames of the structural change vector, resulting in a 1×6 summary array, 6 for each different window size. The median is also used as a second summary indicator if the structural change is concentrated in a small proportion of the music. This prevents the misleading that a high mean represents a concentration of structural change while songs have relatively little structural change elsewhere. In this case, 12 values can represent a single music characteristic, where six

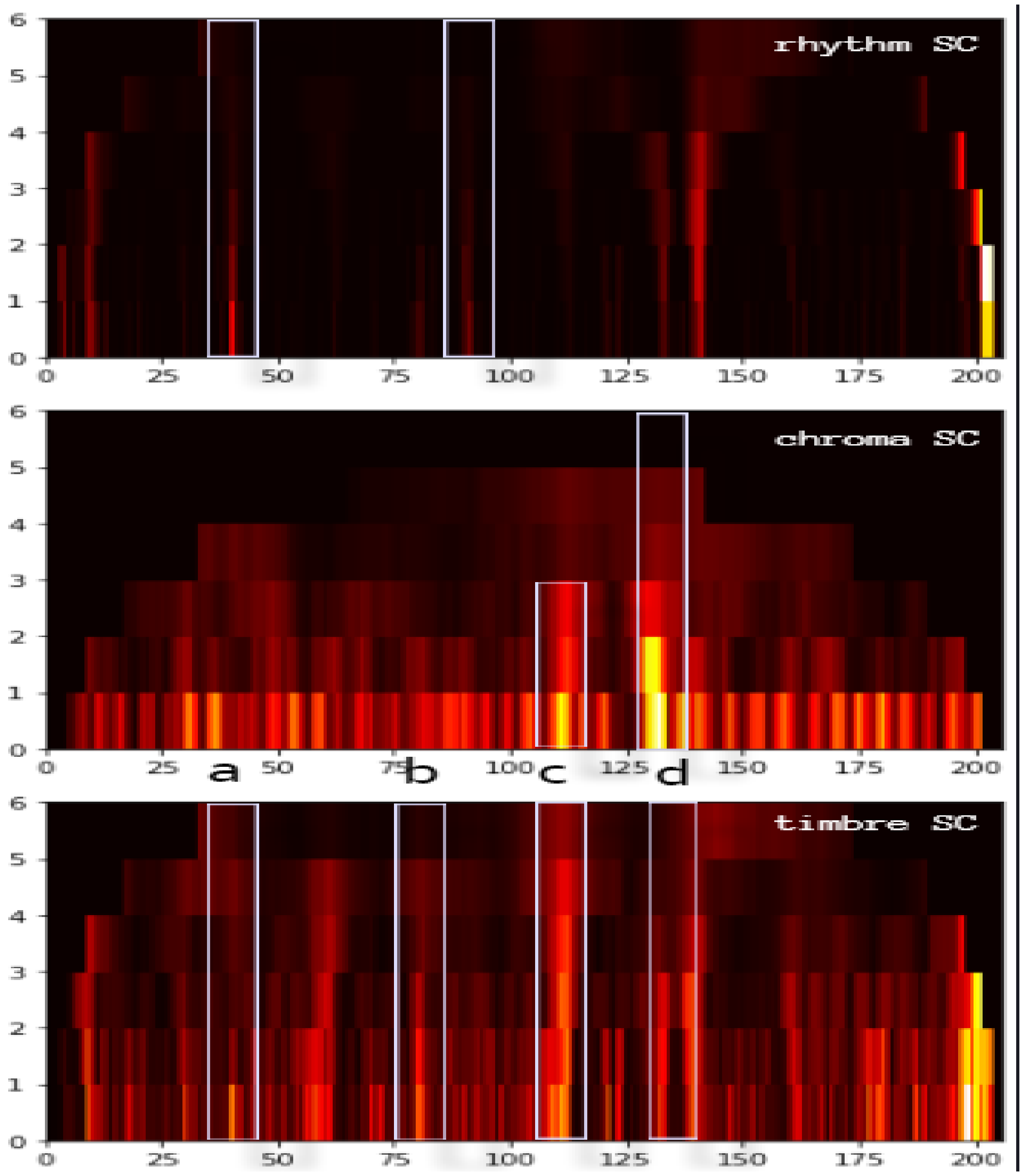
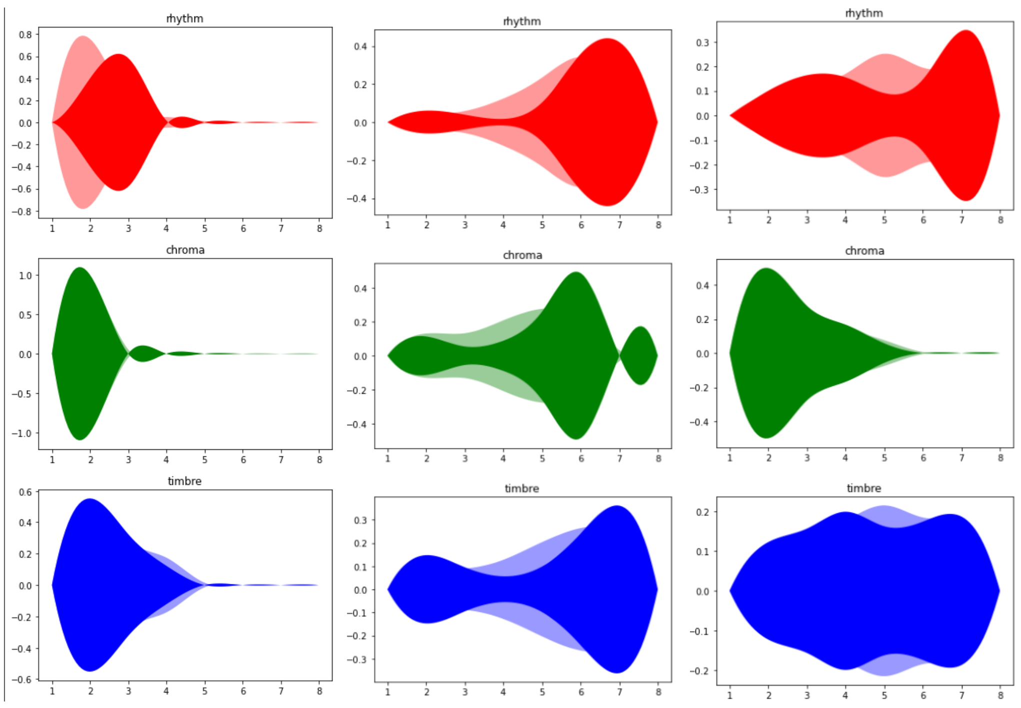


Figure 8: Visualisation of structural change for the song 'Lucky' by Britney Spears, my implementation

numbers are the mean, and the other six numbers represent the median. Thus, since we have three musical features, rhythm, chroma and timbre, each musical track can be summarised by $12 \times 3 = 36$ values. Showing these 36 values on a graph can be problematic in Python. Thus, three figures are drawn for each music feature. In each plot, the opaque part shows the more robust median value, and the translucent part represents the mean value. The graph is drawn by first plotting eight values on the horizontal axis, with two zeros at the beginning and end. The horizontal axis coordinates indicate window size getting larger from left to right. For instance, if the opaque part is wider on the left than on the right, there is more short-term variation in the song than long-term variation, and vice versa. A smooth line is then drawn by calculating an interpolation of 100 points between those eight values.



(a) (b) (c)

Figure 9: structural change summary visualisation for (a) Pink noise (b) Around the world (c) Prelude in C Major

This section does not compare my implementation and theirs in the structural feature change summary for two reasons. One is that music tracks are hard to find, and the paper only describes some features of the test music, for example, ”single major piano with different rhythmic sections”. The other reason is the normalisation; their implementation uses quantile normalisation relative to the entire song set but does not refer to the music dataset. In my implementation, for each music track, L1 normalisation is used. Figure 9 shows three audio structural change summaries using my implementation. Three audio each has its unique feature. For (a) pink noise, since the whole track is simple repetitive noise, there is no expected structural change in the long term. (b) is Around the world performed by Draft Punk. This track repeats the same sentence within 7 minutes, with some variations in the background from time to time. The expectation is to have a minor structural change in the short term and have some structural change in the mid-term or long term. (c) is a classic piano piece Prelude in C major performed by Lang Lang, which has a repeated melody in C major and a variation at the end. The expectation for this piece is that it should have little chroma structural change in the long term. The figure shows the result of the three songs all reaches my expectations, which proves the validation of my framework.

4.3 Structural change of different music subsets

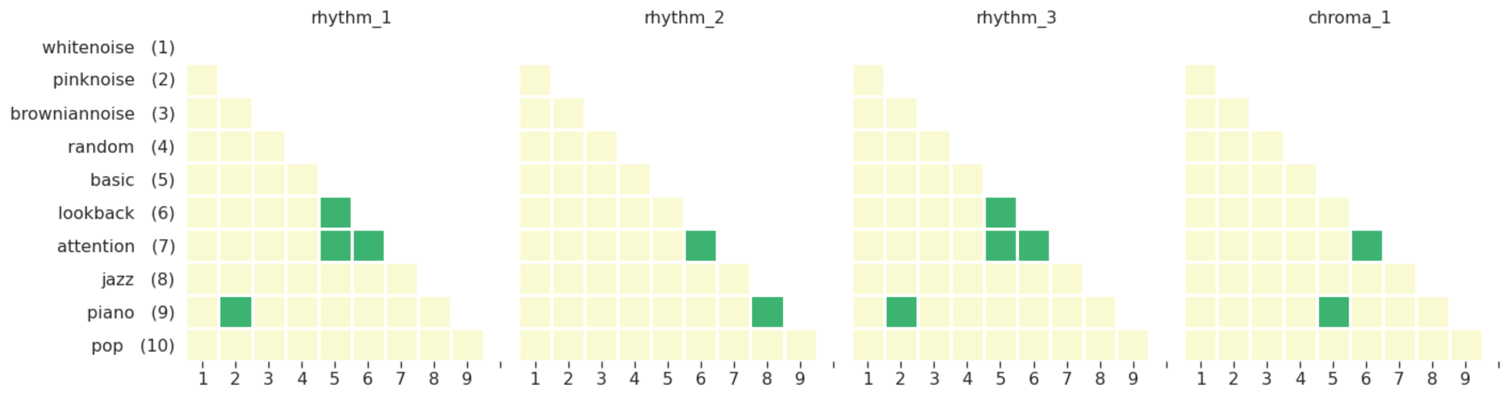
Following the summary of the structural change in the previous section, we apply this metric to all the music tracks in the dataset. This time, L1 normalisation is no longer used. Instead, quantile normalisation is used for a single track with respect to the entire dataset. Since each music feature summary has six dimensions, principle component analysis (PCA) is applied to shrink the dimensions to 2 to help visualisation better. The first two components of each feature structural change summary could explain 83% of the variance in rhythm, 94% of the variance in chroma, and 92% of the variance in timbre. Since the first two components are the summary of structural change summary, which is of confusing logic, below, the two components will be referred to by SSCS.

Since in my dataset, all the music tracks except noise are played using one instrument (mainly piano), timbre is not considered a significant music feature in the distribution comparison of SSCS. Figure 11 shows the comparison between different music subsets of chroma SSCS, and Figure 12 shows the comparison between different music subsets of rhythm SSCS. The boundaries of the four subsets mentioned in 4.1 are not clearly identified. As a result, Kolmogorov–Smirnov (KS) tests can be applied to those distributions to tell whether there are statistical differences between those groups. This type of test is chosen because it has no requirement for input data, unlike t-tests, which require the data to be normally distributed. As suggested by [6], Bonferroni corrections are used to prevent my results from being contingent, and p is set to 0.05. The KS tests are performed between each subset of music genres and for each principle component. To unify the amount of variance to be explained and balance the principle components of each music feature, 85% is set to be that amount of variance. Under this circumstance, rhythm has three principle components, and chroma and timbre have two principle components. Because of the large number of test pairs, ten heatmaps are used to indicate whether there are significant differences in the data between each subset of music in each principle component. As shown in Figure 10, the yellow squares show a significant statistical difference between those two subsets, and the green squares otherwise.

The figure reveals that these 10 subsets are statistically different from each other except for music generated by the lookback RNN and the attention RNN. Thus, 9 distinct clusters can be distinguished: music generated by the lookback RNN and the attention RNN, with other subsets of music forming their cluster. This contributes a lot to music structural analysis. Section 4.1 describes the four main classes of the dataset. The expected result of this experiment would be some statistical differences between real music, computer-generated music, random music and noise, with the structural complexity being decreased. However, the result proves that even within major categories of music, statistical differences could also be found between small subsets. In conclusion, SSCS allows discrimination of different music subsets in my dataset and can reveal their structural complexities by estimating the position of their principle components on the structural complexity planes.

5 Legal, social, ethical and professional issues

Special care has been taken to abide by the Code of Conduct & Code of Good Practice issued by the British Computer Society (BSC) and the computer science project and Rule of Conduct issued by The Institution of Engineering and Technology (IET) throughout the entire project. The application of the rules is all reflected in chapter 4. In 4.1, the generation of each subset is explicitly stated, and online sources for some of them are given. In 4.2, the paper that inspired the idea of structural change is outlined. Section 4.3 gives the paper referred to in the parameter selection section. All source code was implemented by myself, except for the heat map visualisation function in the pairwise subset comparison after SSCS extraction, which will be mentioned later in the appendix.



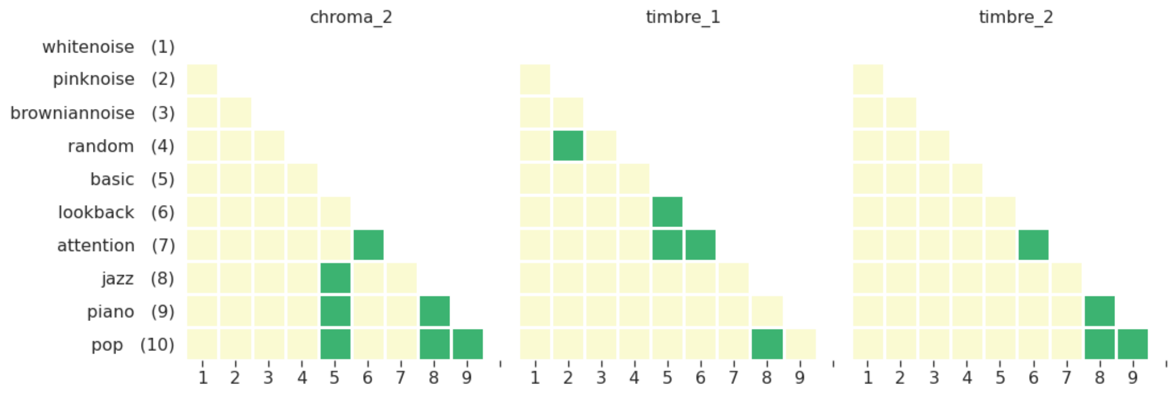
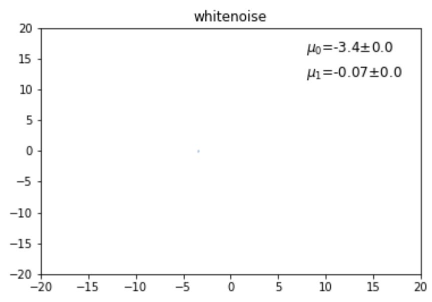
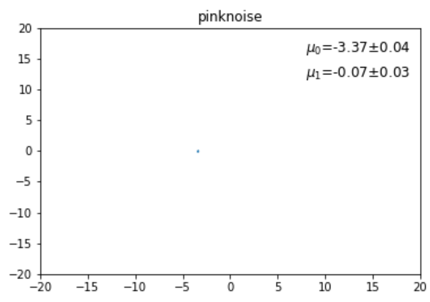


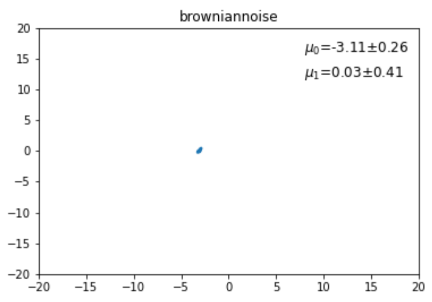
Figure 10: Pairwise statistical analysis of the music subsets for each principle component (yellow represents statistical difference)



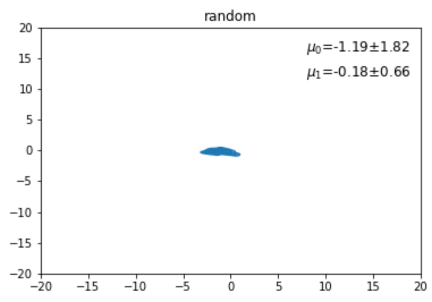
(a)



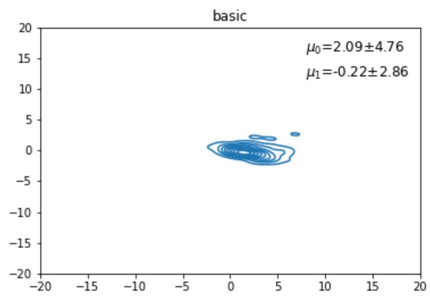
(b)



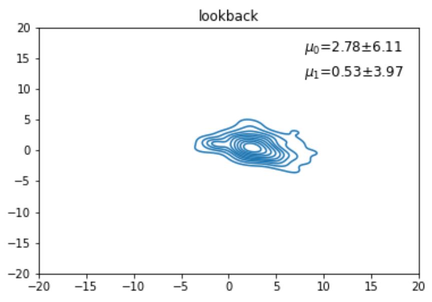
(c)



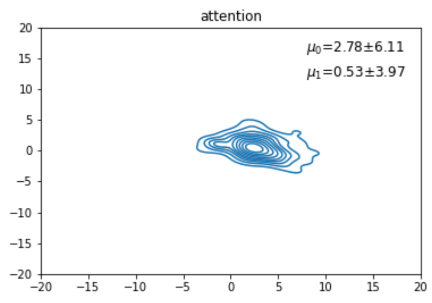
(d)



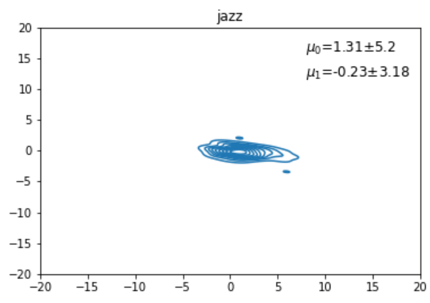
(e)



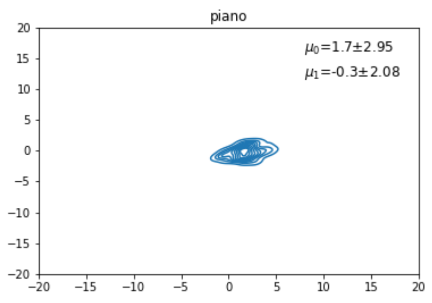
(f)



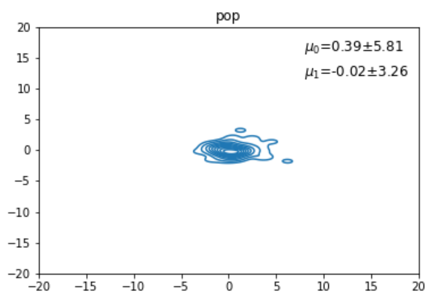
(g)



(h)

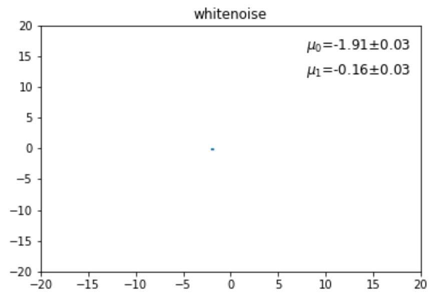


(i)

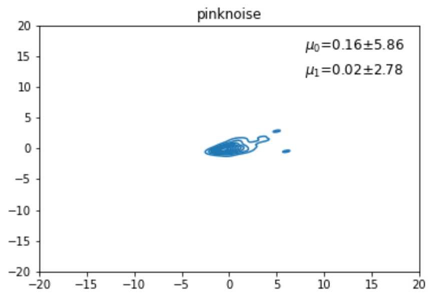


(j)

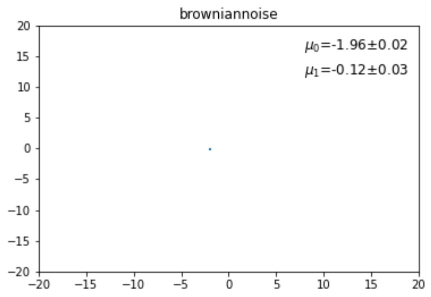
Figure 11: Distribution of chroma SSCS for each subset after bivariate KDE



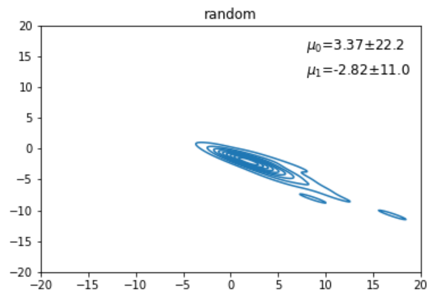
(a)



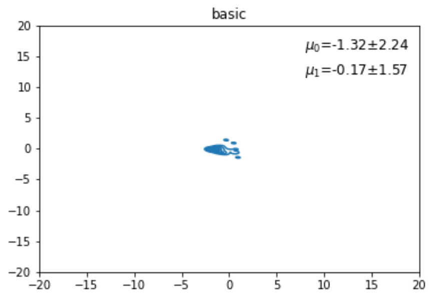
(b)



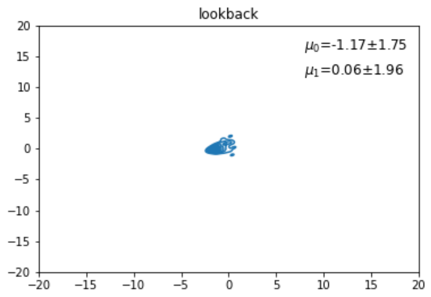
(c)



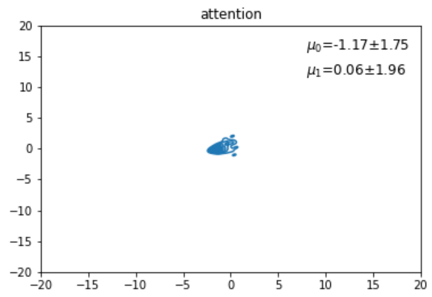
(d)



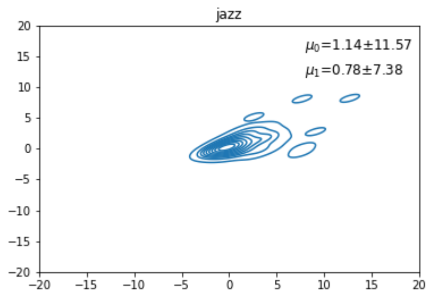
(e)



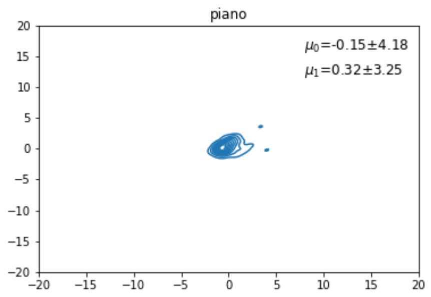
(f)



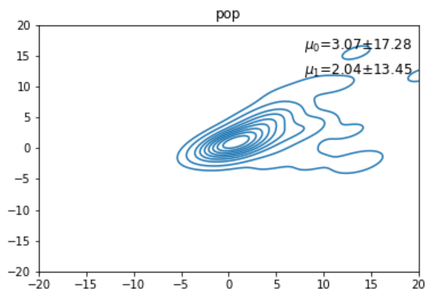
(g)



(h)



(i)



(j)

Figure 12: Distribution of rhythm SSCS for each subset after bivariate KDE

6 Conclusion

6.1 Critical analysis

The project addressed an open problem in computational automatic music structure analysis, which has a huge implication in evaluating music generation models and comparing different music generation systems. My implementation starts with sonification MIDI to audio files to form the dataset. Then, given a specific music track, a structural change vector is calculated by concatenating the structural change for all frames under six different window lengths of three basic music features rhythm, chroma, and timbre. This is followed by computing a summary of these features by means and medians to get an SSCS, a $3 \times 12 = 36$-dimension vector representing all the structural changes for a single track.

To test the ability of my implementation to characterise music structural properties, I applied the framework to different music subsets: real music, computer-generated music, random music and noise, which are of different music structural complexities. Specifically, PCA is used for each music feature to compress the 12-dimension vector to a lower dimension. KDE is applied to the principle components computed above to constitute distributions for each music subset. Finally, statistical tests are performed between those distributions to show if there are statistical differences between each group for each principle component. The results showed that different musical subsets indeed have different musical structural complexity. Thus, the musical structural complexity of an individual track can be obtained by finding the location of distributions after applying the principle components of KDE on the structural complexity planes, which can be formed by my 10 musical subsets.

This implementation of calculating structural change, computing SSCS, together with the PCA, KDE, and finding the position of the results on structural planes can provide a compact framework for evaluating the music structural complexity of a single audio track. This framework contributes to the MSA field by following: (1) it provides a simple way to use an array to represent the structural complexity of a music track; (2) it takes much less time to compute the SSCS that summarises the whole track than most of the feature extraction method in MSA; (3) it can reveal the structural complexity of music audio, using the method mentioned above; (4) my implementation did not subjectively define structural complexity for subsets but made an assumption that different music subsets could have different structural complexity. Besides, the audio of waveform and symbolic music can both be the input of my framework. To apply the framework on symbolic music, an additional sonification step is required, which is included in the source code.

6.2 Future directions

The implementation of extracting the feature change vector presented in chapter 3 is only one way of calculating the structural change. Alternatives could be to try different window sizes and divergence functions. The window size of power of two is chosen in my implementation. However, this could lead to a vast window length gap in the last few windows. For example, $2^6 - 2^5 = 32$ frames of window size could be neglected. Instead, further study can explore a more subtle change in window size by adding some interpolation between significant window length gaps. Apart from this, feature extraction of rhythm can be improved because STFT may not be a good way of extracting rhythm features. It is also interesting to explore more divergence functions such as Euclidean distance. The music dataset could also be enlarged by either adding more songs to each subset or adding more music genres. Analysing more songs enable us to find robust structural change planes, thus providing us with more meaningful criteria.

References

[1] J.-P. Briot and F. Pachet, “Music Generation by Deep Learning - Challenges and Directions,” *Neural Computing and Applications*, vol. 32, pp. 981–993, Feb. 2020. arXiv: 1712.04371.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention Is All You Need,” *arXiv:1706.03762 [cs]*, Dec. 2017. arXiv: 1706.03762.

[3] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. Conference Name: Neural Computation.

[4] M. Müller, *Fundamentals of Music Processing – Audio, Analysis, Algorithms, Applications.* Jan. 2015.

[5] L.-C. Yang and A. Lerch, “On the evaluation of generative models in music,” *Neural Computing and Applications*, vol. 32, May 2020.

[6] J. de Berardinis, A. Cangelosi, and E. Coutinho, “Measuring the Structural Complexity of Music: From Structural Segmentations to the Automatic Evaluation of Models for Music Generation,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 30, pp. 1963–1976, 2022. Conference Name: IEEE/ACM Transactions on Audio, Speech, and Language Processing.

[7] O. Nieto, G. J. Mysore, C.-i. Wang, J. B. L. Smith, J. Schlüter, T. Grill, and B. McFee, “Audio-Based Music Structure Analysis: Current Trends, Open Challenges, and Applications,” *Transactions of the International Society for Music Information Retrieval*, vol. 3, pp. 246–263, Dec. 2020. Number: 1 Publisher: Ubiquity Press.

[8] “Automatic Music Transcription: An Overview | IEEE Journals & Magazine | IEEE Xplore.”

[9] M. Schedl, “Deep Learning in Music Recommendation Systems,” *Frontiers in Applied Mathematics and Statistics*, vol. 5, 2019.

[10] B. YEGNANARAYANA, *ARTIFICIAL NEURAL NETWORKS.* PHI Learning Pvt. Ltd., Jan. 2009. Google-Books-ID: RTtvUVU_xL4C.

[11] J. J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities.,” *Proceedings of the National Academy of Sciences*, vol. 79, pp. 2554–2558, Apr. 1982. Publisher: Proceedings of the National Academy of Sciences.

[12] P. Werbos, “Backpropagation through time: what it does and how to do it,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990. Conference Name: Proceedings of the IEEE.

[13] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997. Conference Name: IEEE Transactions on Signal Processing.

[14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," Dec. 2014. arXiv:1412.3555 [cs].

[15] "Generating Long-Term Structure in Songs and Stories."

[16] C.-Z. A. Huang, T. Cooijmans, A. Roberts, A. Courville, and D. Eck, "Counterpoint by Convolution," *arXiv:1903.07227 [cs, eess, stat]*, Mar. 2019. arXiv: 1903.07227.

[17] C.-Z. A. Huang, A. V. J. Uszkoreit, N. Shazeer, I. S. C. Hawthorne, A. M. Dai, M. D. Hoffman, and M. D. D. Eck, "MUSIC TRANSFORMER: GENERATING MUSIC WITH LONG-TERM STRUCTURE," p. 15, 2019.

[18] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-Attention with Relative Position Representations," Apr. 2018. arXiv:1803.02155 [cs].

[19] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck, "Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset," *arXiv:1810.12247 [cs, eess, stat]*, Jan. 2019. arXiv: 1810.12247.

[20] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," *arXiv:1312.6114 [cs, stat]*, May 2014. arXiv: 1312.6114.

[21] A. Roberts, J. Engel, C. Raffel, C. Hawthorne, and D. Eck, "A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music," *arXiv:1803.05428 [cs, eess, stat]*, Nov. 2019. arXiv: 1803.05428.

[22] "Musenet : Music Generation using Abstractive and Generative Methods," *International Journal of Innovative Technology and Exploring Engineering*, vol. 9, pp. 784–788, Apr. 2020.

[23] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating Long Sequences with Sparse Transformers," *arXiv:1904.10509 [cs, stat]*, Apr. 2019. arXiv: 1904.10509.

[24] J. de Berardinis, S. Barrett, A. Cangelosi, and E. Coutinho, "Modelling long- and short-term structure in symbolic music with attention and recurrence," p. 11.

[25] X. Wu, C. Wang, and Q. Lei, "Transformer-XL Based Music Generation with Multiple Sequences of Time-valued Notes," *arXiv:2007.07244 [cs, eess]*, July 2020. arXiv: 2007.07244.

[26] Y.-S. Huang and Y.-H. Yang, "Pop Music Transformer: Beat-based Modeling and Generation of Expressive Pop Piano Compositions," *arXiv:2002.00212 [cs, eess, stat]*, Aug. 2020. arXiv: 2002.00212.

[27] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context," *arXiv:1901.02860 [cs, stat]*, June 2019. arXiv: 1901.02860.

[28] A. Muhamed, L. Li, X. Shi, S. Yaddanapudi, W. Chi, D. Jackson, R. Suresh, and A. J. Smola, "Symbolic Music Generation with Transformer-GANs," p. 9.

[29] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *arXiv:1406.2661 [cs, stat]*, June 2014. arXiv: 1406.2661.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, May 2019. arXiv: 1810.04805 version: 2.

[31] J. Paulus, M. Müller, and A. Klapuri, "Audio-based music structure analysis," *Proceedings of the 11th International Society for Music Information Retrieval Conference, ISMIR 2010*, pp. 625–636, Jan. 2010.

[32] G. Sargent, F. Bimbot, and E. Vincent, "A REGULARITY-CONSTRAINED VITERBI ALGORITHM AND ITS APPLICATION TO THE STRUCTURAL SEGMENTATION OF SONGS," *Poster Session*, p. 6, 2011.

[33] J. Foote, "Visualizing music and audio using self-similarity," in *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*, MULTIMEDIA '99, (New York, NY, USA), pp. 77–80, Association for Computing Machinery, 1999.

[34] J. Serrà, M. Müller, P. Grosche, and J. L. Arcos, "Unsupervised Music Structure Annotation by Time Series Structure Features and Segment Similarity," *IEEE Transactions on Multimedia*, vol. 16, no. 5, pp. 1229–1240, 2014. Conference Name: IEEE Transactions on Multimedia.

[35] M. Goto, "A chorus-section detecting method for musical audio signals," in *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, vol. 5, pp. V–437, Apr. 2003. ISSN: 1520-6149.

[36] M. C. McCallum, "Unsupervised Learning of Deep Features for Music Segmentation," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 346–350, May 2019. arXiv:2108.12955 [cs, eess].

[37] J.-C. Wang, H.-S. Lee, H.-m. Wang, and S.-K. Jeng, "Learning the Similarity of Audio Music in Bag-of-frames Representation from Tagged Music Data.," pp. 85–90, Jan. 2011.

[38] D. Schnitzer, A. Flexer, M. Schedl, and G. Widmer, "USING MUTUAL PROXIMITY TO IMPROVE CONTENT-BASED AUDIO SIMILARITY," *Poster Session*, p. 6, 2011.

[39] O. Nieto and J. P. Bello, “Music segment similarity using 2D-Fourier Magnitude Coefficients,” in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, (Florence, Italy), pp. 664–668, IEEE, May 2014.

[40] J. Serrà, X. Serra, and R. G. Andrzejak, “Cross recurrence quantification for cover song identification,” *New Journal of Physics*, vol. 11, no. 9, p. 093017, 2009. Publisher: IOP Publishing.

[41] M. Levy and M. Sandler, “Structural Segmentation of Musical Audio by Constrained Clustering,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 16, no. 2, pp. 318–326, 2008. Conference Name: IEEE Transactions on Audio, Speech, and Language Processing.

[42] J. Paulus and A. Klapuri, “Music Structure Analysis Using a Probabilistic Fitness Measure and a Greedy Search Algorithm,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 17, pp. 1159–1170, Aug. 2009.

[43] B. Mcfee and D. P. W. Ellis, “Analyzing Song Structurewith Spectral Clustering.”

[44] G. Sargent, F. Bimbot, and E. Vincent, “Estimating the Structural Segmentation of Popular Music Pieces Under Regularity Constraints,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 2, pp. 344–358, 2017. Conference Name: IEEE/ACM Transactions on Audio, Speech, and Language Processing.

[45] G. Shibata, R. Nishikimi, E. Nakamura, and K. Yoshii, “STATISTICAL MUSIC STRUCTURE ANALYSIS BASED ON A HOMOGENEITY-, REPETITIVENESS-, AND REGULARITY-AWARE HIERARCHICAL HIDDEN SEMI-MARKOV MODEL,” p. 8, 2019.

[46] D. P. Ellis and G. E. Poliner, “Identifying ‘Cover Songs’ with Chroma Features and Dynamic Programming Beat Tracking,” in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP ’07*, (Honolulu, HI), pp. IV–1429–IV–1432, IEEE, Apr. 2007.

[47] M. Müller, F. Kurth, and M. Clausen, “Audio Matching via Chroma-Based Statistical Features.,” 2005. Conference Name: Proceedings of the 6th International Conference on Music Information Retrieval (ISMIR 2005) Pages: 288-295 Publication Title: Proceedings of the 6th International Conference on Music Information Retrieval Publisher: ISMIR.

[48] M. A. Bartsch and G. H. Wakefield, “Audio thumbnailing of popular music using chroma-based representations,” *IEEE Transactions on Multimedia*, pp. 96–104, 2005.

[49] F. Levé, R. Groult, G. Arnaud, C. Séguin, R. Gaymay, and M. Giraud, “Rhythm extraction from polyphonic symbolic music,” in *12th International Society for Music Information Retrieval Conference (ISMIR 2011)*, (United States), pp. 375–380, 2011.

[50] “FluidSynth | Software synthesizer based on the SoundFont 2 specifications.”

[51] B. McFee, C. Raffel, D. Liang, D. Ellis, M. McVicar, E. Battenberg, and O. Nieto, “librosa: Audio and Music Signal Analysis in Python,” (Austin, Texas), pp. 18–24, 2015.

[52] “Classical Piano Midi Page - Main Page.”

[53] “Free Midi - Best Free High Quality Midi Site.”

[54] “MIDI files - Download for free :: MIDIWORLD.COM.”

[55] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” May 2016. arXiv:1409.0473 [cs, stat].

[56] M. Mauch and M. Levy, “Structural change on multiple time scales as a correlate of musical complexity,” in *In Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR’11*, 2011.

[57] “Chroma Feature Analysis and Synthesis.”

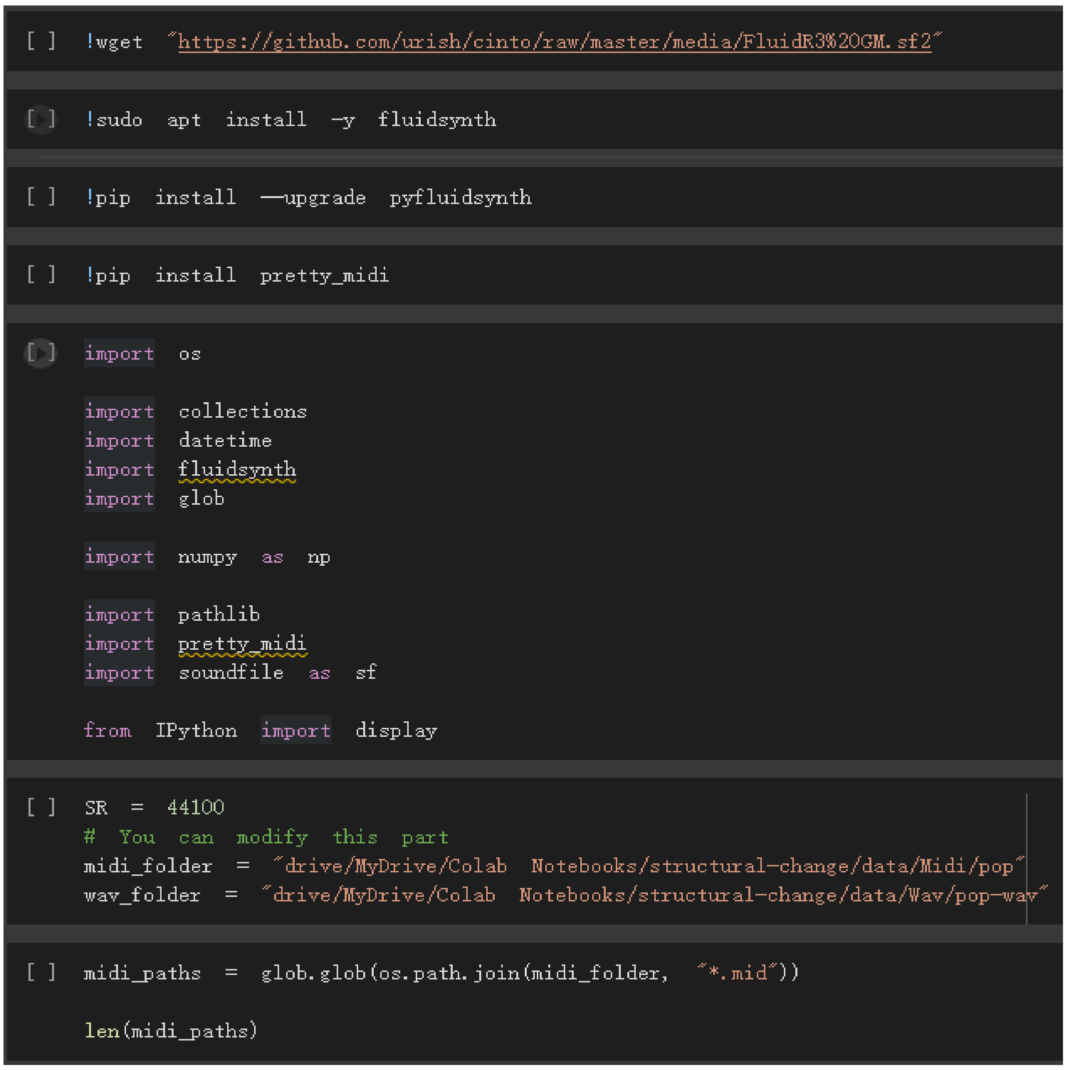
A Appendix

A.1 Source code overview

The structure of the source code is illustrated in this section. The source code includes 4 folders: data, src, result, and test. The data folder includes the dataset used in this project. MIDI stores all the subset MIDI files, and WAV is an empty folder waiting for the music to be generated. The result folder shows the result of this project. Specifically, the structural change vectors of all 10 subsets are stored in this folder. The results after PCA and KDE are visualised and stored as pictures. The pairwise comparison is shown as a figure. The test folder includes all the test tracks. The src folder contains all the files used in this project, which will be concretely presented in the below section. All the code are implemented by myself, exception three functions in pairwise_comparison, which are kindly provided by Jacopo de Berardinis.

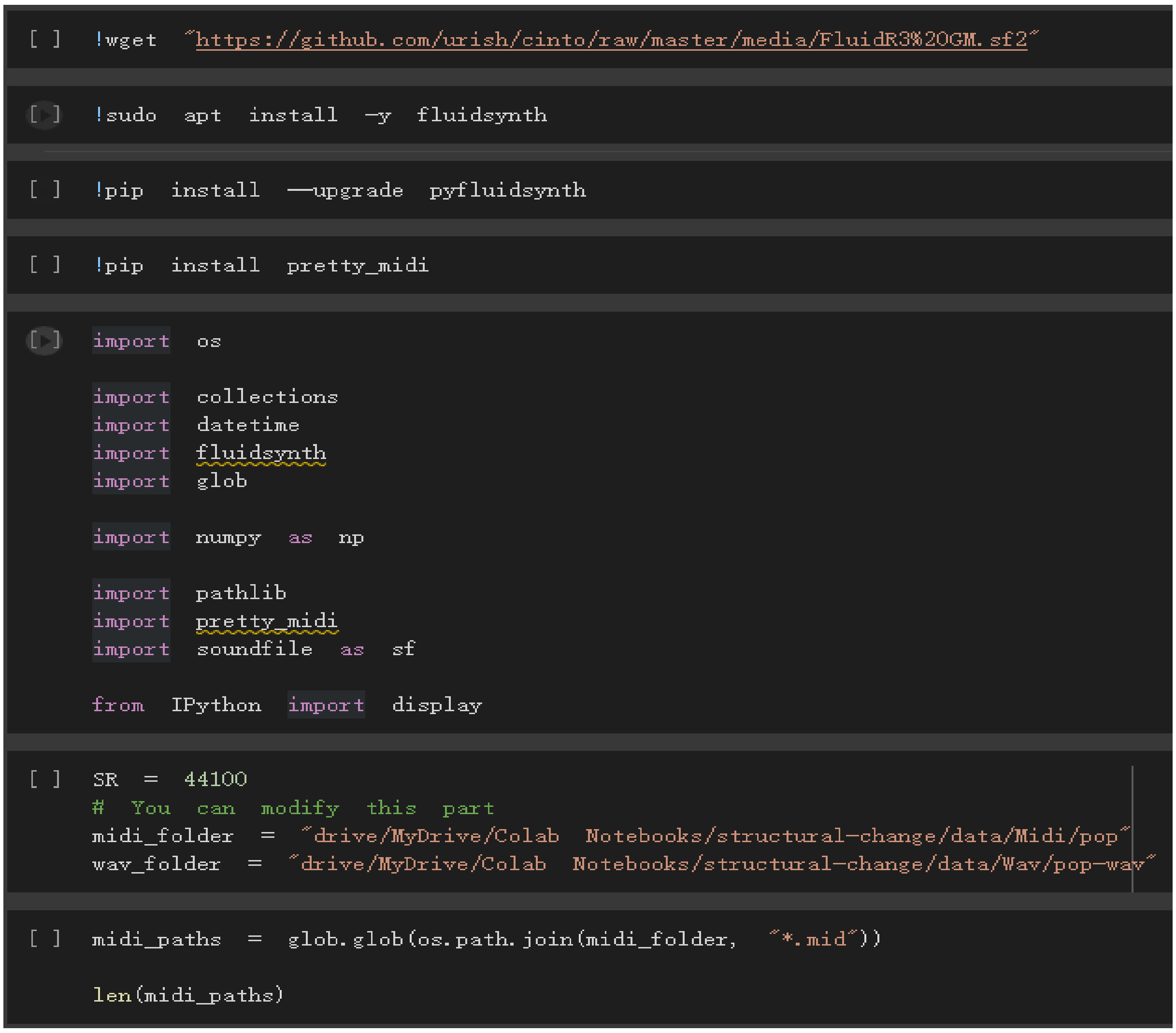
A.2 Key files

A.2.1 sonification.py



```
[ ]  !wget "https://github.com/urish/cinto/raw/master/media/FluidR3%20GM.sf2"

[ ]  !sudo apt install -y fluidsynth

[ ]  !pip install --upgrade pyfluidsynth

[ ]  !pip install pretty_midi

[ ]  import os

     import collections
     import datetime
     import fluidsynth
     import glob

     import numpy as np

     import pathlib
     import pretty_midi
     import soundfile as sf

     from IPython import display

[ ]  SR = 44100
     # You can modify this part
     midi_folder = "drive/MyDrive/Colab Notebooks/structural-change/data/Midi/pop"
     wav_folder = "drive/MyDrive/Colab Notebooks/structural-change/data/Wav/pop-wav"

[ ]  midi_paths = glob.glob(os.path.join(midi_folder, "*.mid"))

     len(midi_paths)
```

Figure 13: sonification_1



```
!wget "https://github.com/urish/cinto/raw/master/media/FluidR3%20GM.sf2"
```

```
!sudo apt install -y fluidsynth
```

```
!pip install --upgrade pyfluidsynth
```

```
!pip install pretty_midi
```

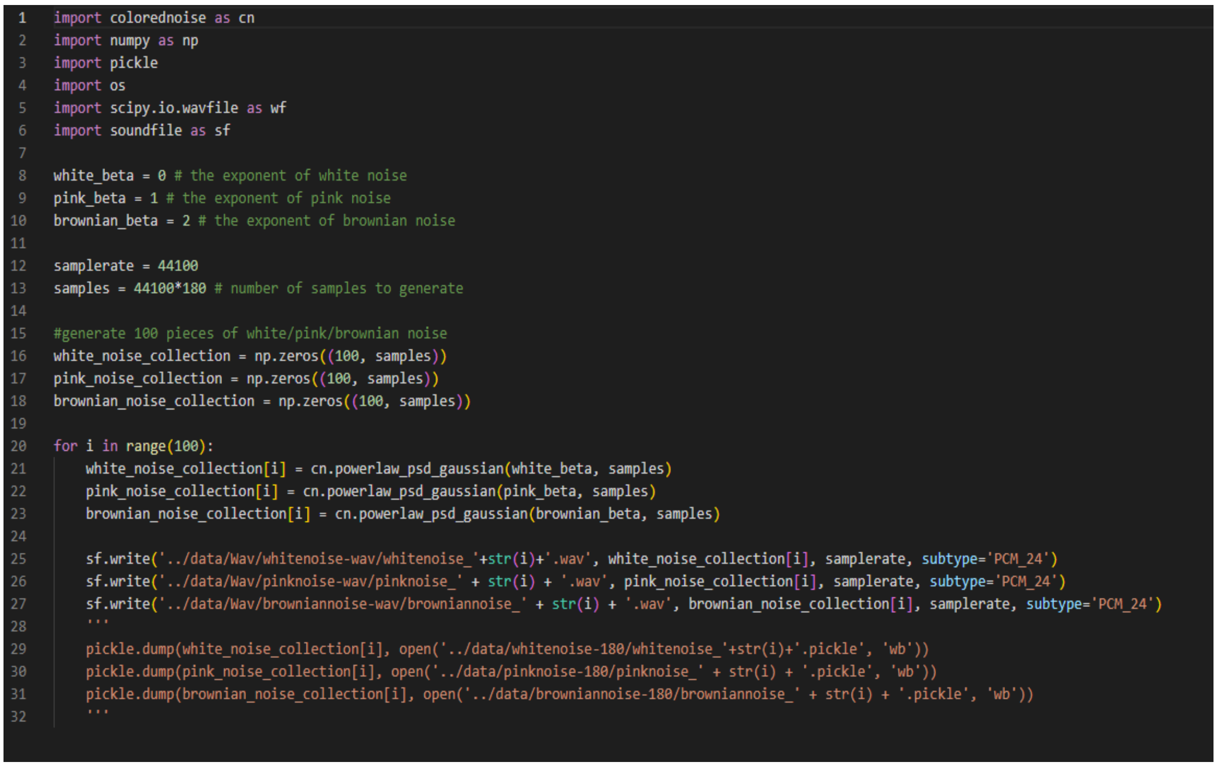
```
import os

import collections
import datetime
import fluidsynth
import glob

import numpy as np

import pathlib
import pretty_midi
import soundfile as sf

from IPython import display
```

```
SR = 44100
# You can modify this part
midi_folder = "drive/MyDrive/Colab Notebooks/structural-change/data/Midi/pop"
wav_folder = "drive/MyDrive/Colab Notebooks/structural-change/data/Wav/pop-wav"
```

```
midi_paths = glob.glob(os.path.join(midi_folder, "*.mid"))

len(midi_paths)
```

Figure 14: sonification_2

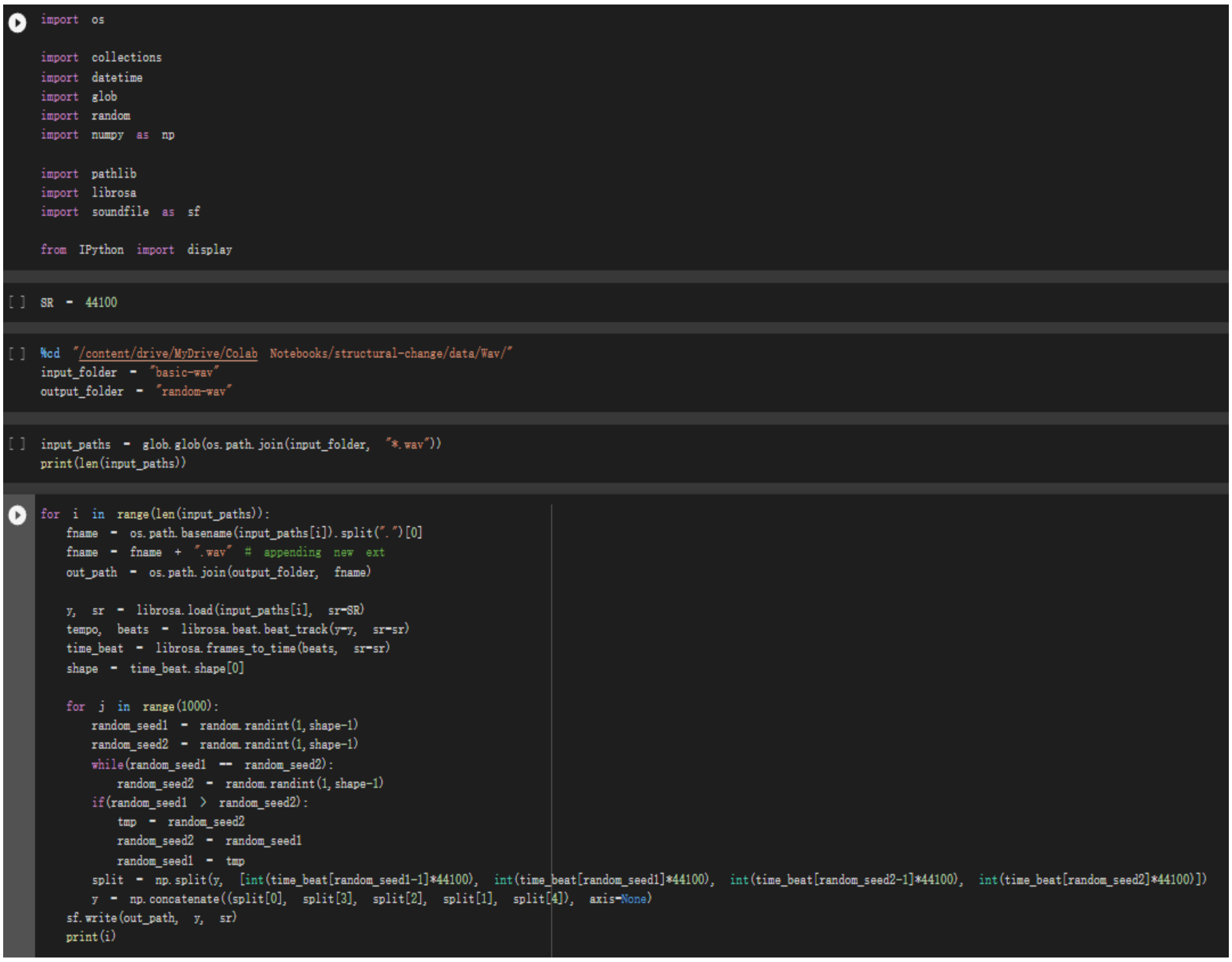
A.2.2 noise_generation.py



```python
import colorednoise as cn
import numpy as np
import pickle
import os
import scipy.io.wavfile as wf
import soundfile as sf

white_beta = 0 # the exponent of white noise
pink_beta = 1 # the exponent of pink noise
brownian_beta = 2 # the exponent of brownian noise

samplerate = 44100
samples = 44100*180 # number of samples to generate

#generate 100 pieces of white/pink/brownian noise
white_noise_collection = np.zeros((100, samples))
pink_noise_collection = np.zeros((100, samples))
brownian_noise_collection = np.zeros((100, samples))

for i in range(100):
    white_noise_collection[i] = cn.powerlaw_psd_gaussian(white_beta, samples)
    pink_noise_collection[i] = cn.powerlaw_psd_gaussian(pink_beta, samples)
    brownian_noise_collection[i] = cn.powerlaw_psd_gaussian(brownian_beta, samples)

    sf.write('../data/Wav/whitenoise-wav/whitenoise_'+str(i)+'.wav', white_noise_collection[i], samplerate, subtype='PCM_24')
    sf.write('../data/Wav/pinknoise-wav/pinknoise_' + str(i) + '.wav', pink_noise_collection[i], samplerate, subtype='PCM_24')
    sf.write('../data/Wav/browniannoise-wav/browniannoise_' + str(i) + '.wav', brownian_noise_collection[i], samplerate, subtype='PCM_24')
    '''
    pickle.dump(white_noise_collection[i], open('../data/whitenoise-180/whitenoise_'+str(i)+'.pickle', 'wb'))
    pickle.dump(pink_noise_collection[i], open('../data/pinknoise-180/pinknoise_' + str(i) + '.pickle', 'wb'))
    pickle.dump(brownian_noise_collection[i], open('../data/browniannoise-180/browniannoise_' + str(i) + '.pickle', 'wb'))
    '''
```

Figure 15: noise_generation

A.2.3 shuffle_beats.py



```
import os

import collections
import datetime
import glob
import random
import numpy as np

import pathlib
import librosa
import soundfile as sf

from IPython import display
```

```
SR = 44100
```

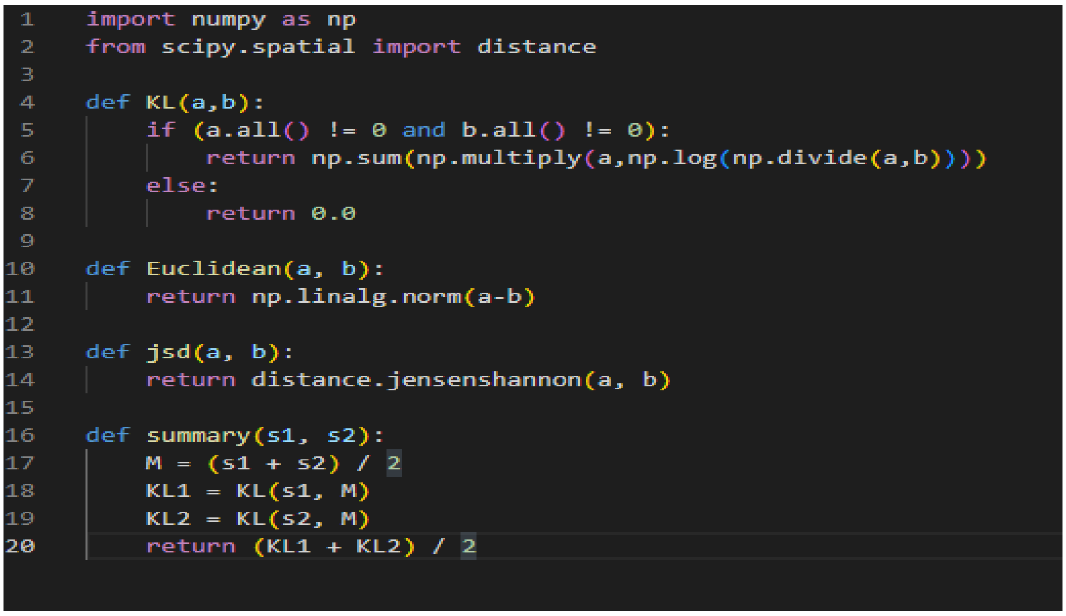
```
%cd "/content/drive/MyDrive/Colab Notebooks/structural-change/data/Wav/"
input_folder = "basic-wav"
output_folder = "random-wav"
```

```
input_paths = glob.glob(os.path.join(input_folder, "*.wav"))
print(len(input_paths))
```

```
for i in range(len(input_paths)):
    fname = os.path.basename(input_paths[i]).split(".")[0]
    fname = fname + ".wav" # appending new ext
    out_path = os.path.join(output_folder, fname)

    y, sr = librosa.load(input_paths[i], sr=SR)
    tempo, beats = librosa.beat.beat_track(y=y, sr=sr)
    time_beat = librosa.frames_to_time(beats, sr=sr)
    shape = time_beat.shape[0]

    for j in range(1000):
        random_seed1 = random.randint(1,shape-1)
        random_seed2 = random.randint(1,shape-1)
        while(random_seed1 == random_seed2):
            random_seed2 = random.randint(1,shape-1)
        if(random_seed1 > random_seed2):
            tmp = random_seed2
            random_seed2 = random_seed1
            random_seed1 = tmp
        split = np.split(y, [int(time_beat[random_seed1-1]*44100), int(time_beat[random_seed1]*44100), int(time_beat[random_seed2-1]*44100), int(time_beat[random_seed2]*44100)])
        y = np.concatenate((split[0], split[3], split[2], split[1], split[4]), axis=None)
    sf.write(out_path, y, sr)
    print(i)
```

Figure 16: shuffle_beats

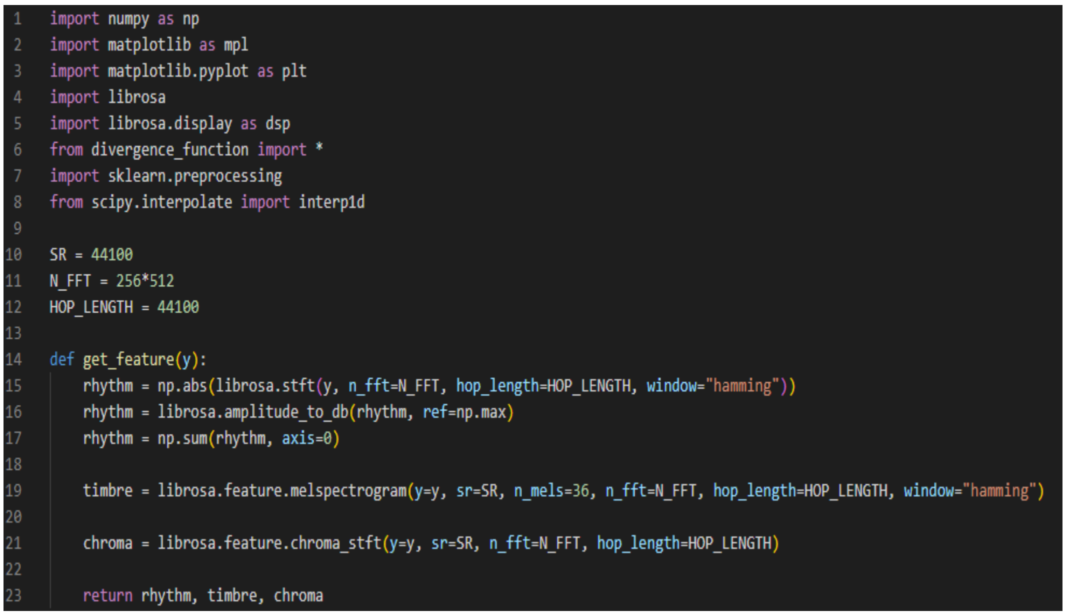
A.2.4 divergence_function.py



```python
import numpy as np
from scipy.spatial import distance

def KL(a,b):
    if (a.all() != 0 and b.all() != 0):
        return np.sum(np.multiply(a,np.log(np.divide(a,b))))
    else:
        return 0.0

def Euclidean(a, b):
    return np.linalg.norm(a-b)

def jsd(a, b):
    return distance.jensenshannon(a, b)

def summary(s1, s2):
    M = (s1 + s2) / 2
    KL1 = KL(s1, M)
    KL2 = KL(s2, M)
    return (KL1 + KL2) / 2
```

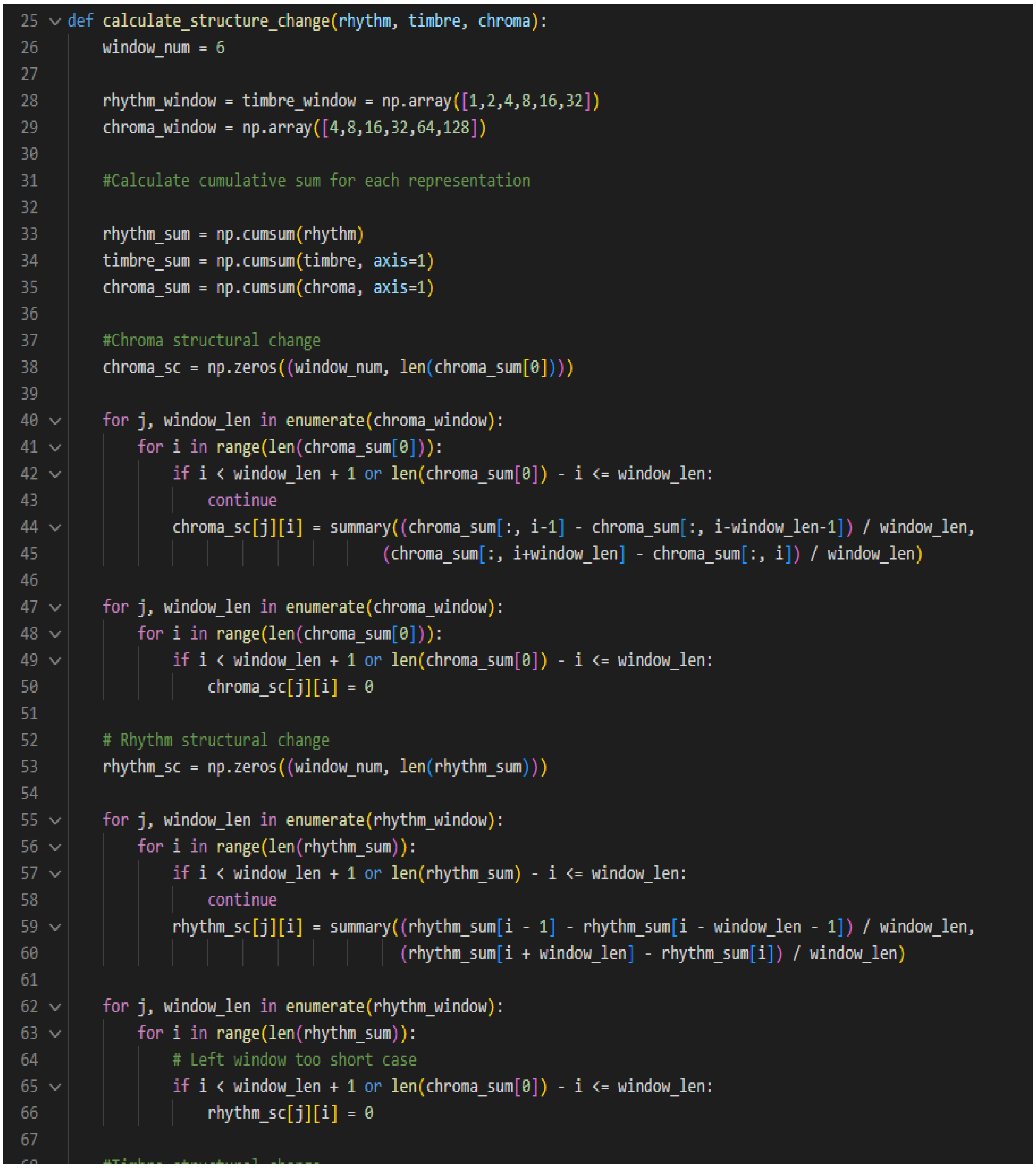
Figure 17: divergence_function

A.2.5 structural_change.py



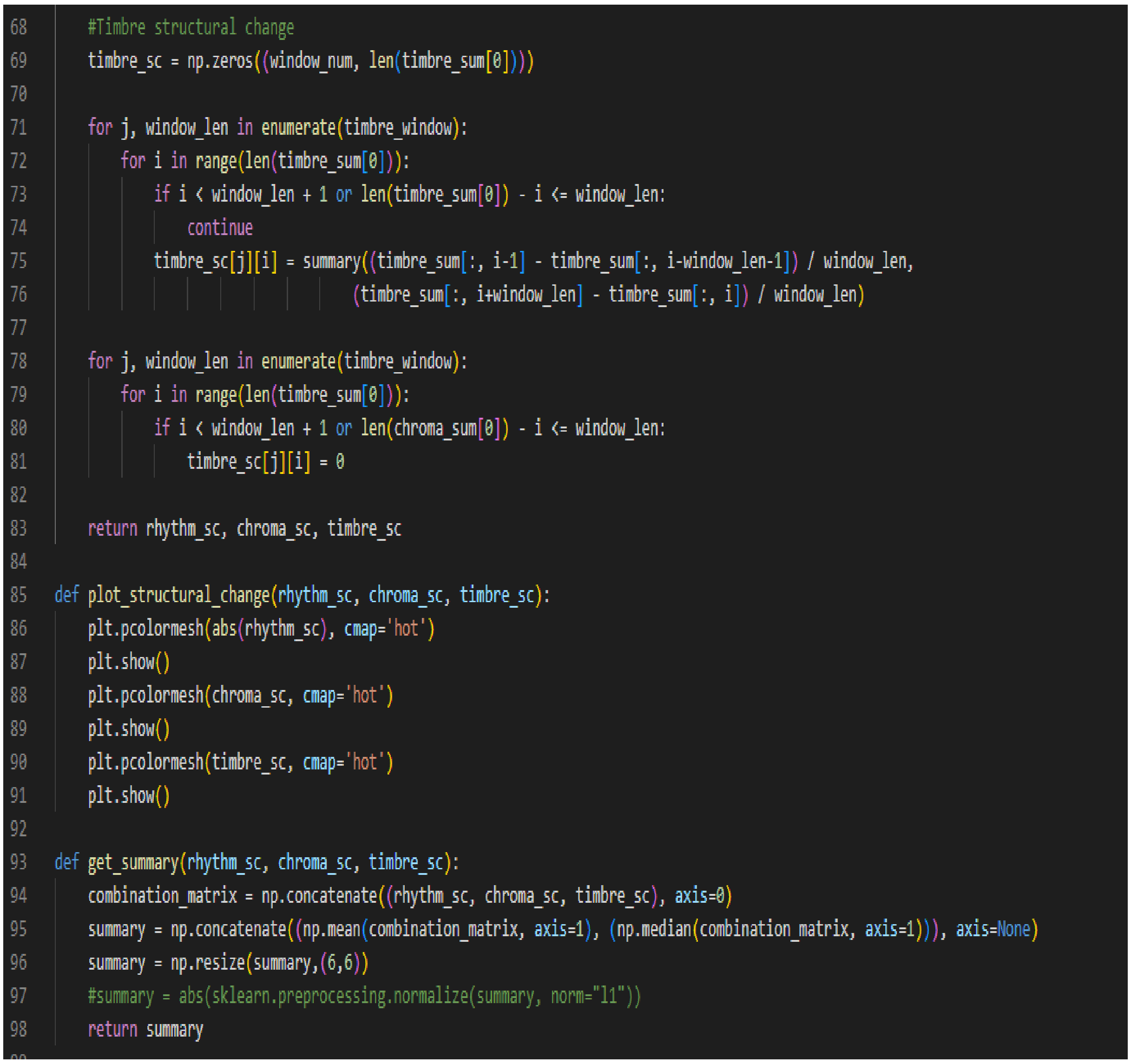
```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import librosa
import librosa.display as dsp
from divergence_function import *
import sklearn.preprocessing
from scipy.interpolate import interp1d

SR = 44100
N_FFT = 256*512
HOP_LENGTH = 44100

def get_feature(y):
    rhythm = np.abs(librosa.stft(y, n_fft=N_FFT, hop_length=HOP_LENGTH, window="hamming"))
    rhythm = librosa.amplitude_to_db(rhythm, ref=np.max)
    rhythm = np.sum(rhythm, axis=0)

    timbre = librosa.feature.melspectrogram(y=y, sr=SR, n_mels=36, n_fft=N_FFT, hop_length=HOP_LENGTH, window="hamming")

    chroma = librosa.feature.chroma_stft(y=y, sr=SR, n_fft=N_FFT, hop_length=HOP_LENGTH)

    return rhythm, timbre, chroma
```

Figure 18: structural_change_1



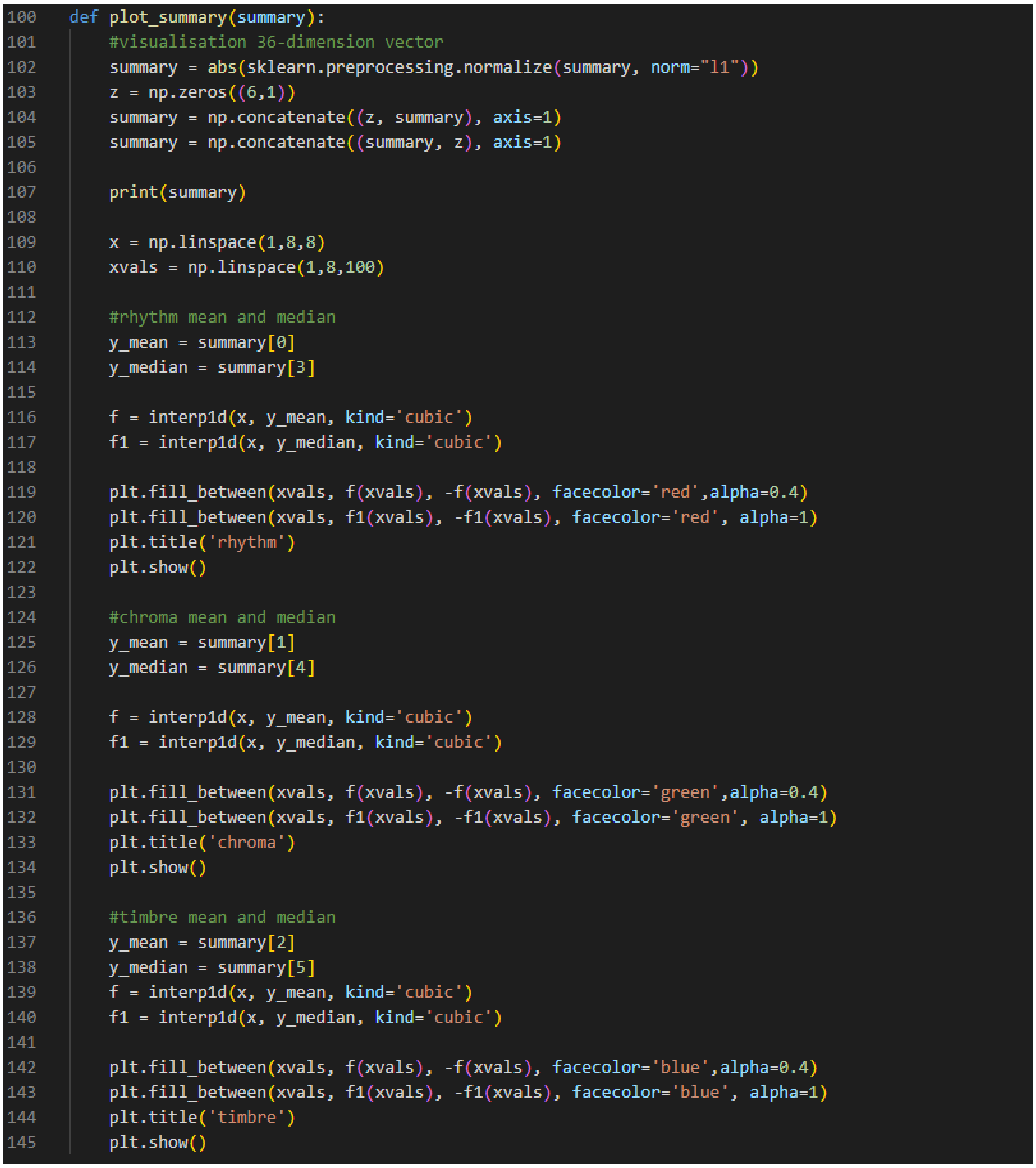
```python
def calculate_structure_change(rhythm, timbre, chroma):
    window_num = 6

    rhythm_window = timbre_window = np.array([1,2,4,8,16,32])
    chroma_window = np.array([4,8,16,32,64,128])

    #Calculate cumulative sum for each representation

    rhythm_sum = np.cumsum(rhythm)
    timbre_sum = np.cumsum(timbre, axis=1)
    chroma_sum = np.cumsum(chroma, axis=1)

    #Chroma structural change
    chroma_sc = np.zeros((window_num, len(chroma_sum[0])))

    for j, window_len in enumerate(chroma_window):
        for i in range(len(chroma_sum[0])):
            if i < window_len + 1 or len(chroma_sum[0]) - i <= window_len:
                continue
            chroma_sc[j][i] = summary((chroma_sum[:, i-1] - chroma_sum[:, i-window_len-1]) / window_len,
                                    (chroma_sum[:, i+window_len] - chroma_sum[:, i]) / window_len)

    for j, window_len in enumerate(chroma_window):
        for i in range(len(chroma_sum[0])):
            if i < window_len + 1 or len(chroma_sum[0]) - i <= window_len:
                chroma_sc[j][i] = 0

    # Rhythm structural change
    rhythm_sc = np.zeros((window_num, len(rhythm_sum)))

    for j, window_len in enumerate(rhythm_window):
        for i in range(len(rhythm_sum)):
            if i < window_len + 1 or len(rhythm_sum) - i <= window_len:
                continue
            rhythm_sc[j][i] = summary((rhythm_sum[i - 1] - rhythm_sum[i - window_len - 1]) / window_len,
                                      (rhythm_sum[i + window_len] - rhythm_sum[i]) / window_len)

    for j, window_len in enumerate(rhythm_window):
        for i in range(len(rhythm_sum)):
            # Left window too short case
            if i < window_len + 1 or len(chroma_sum[0]) - i <= window_len:
                rhythm_sc[j][i] = 0

    #Timbre structural change
```

Figure 19: structural_change_2



```
    #Timbre structural change
    timbre_sc = np.zeros((window_num, len(timbre_sum[0])))

    for j, window_len in enumerate(timbre_window):
        for i in range(len(timbre_sum[0])):
            if i < window_len + 1 or len(timbre_sum[0]) - i <= window_len:
                continue
            timbre_sc[j][i] = summary((timbre_sum[:, i-1] - timbre_sum[:, i-window_len-1]) / window_len,
                                      (timbre_sum[:, i+window_len] - timbre_sum[:, i]) / window_len)

    for j, window_len in enumerate(timbre_window):
        for i in range(len(timbre_sum[0])):
            if i < window_len + 1 or len(chroma_sum[0]) - i <= window_len:
                timbre_sc[j][i] = 0

    return rhythm_sc, chroma_sc, timbre_sc

def plot_structural_change(rhythm_sc, chroma_sc, timbre_sc):
    plt.pcolormesh(abs(rhythm_sc), cmap='hot')
    plt.show()
    plt.pcolormesh(chroma_sc, cmap='hot')
    plt.show()
    plt.pcolormesh(timbre_sc, cmap='hot')
    plt.show()

def get_summary(rhythm_sc, chroma_sc, timbre_sc):
    combination_matrix = np.concatenate((rhythm_sc, chroma_sc, timbre_sc), axis=0)
    summary = np.concatenate((np.mean(combination_matrix, axis=1), (np.median(combination_matrix, axis=1))), axis=None)
    summary = np.resize(summary,(6,6))
    #summary = abs(sklearn.preprocessing.normalize(summary, norm="l1"))
    return summary
```

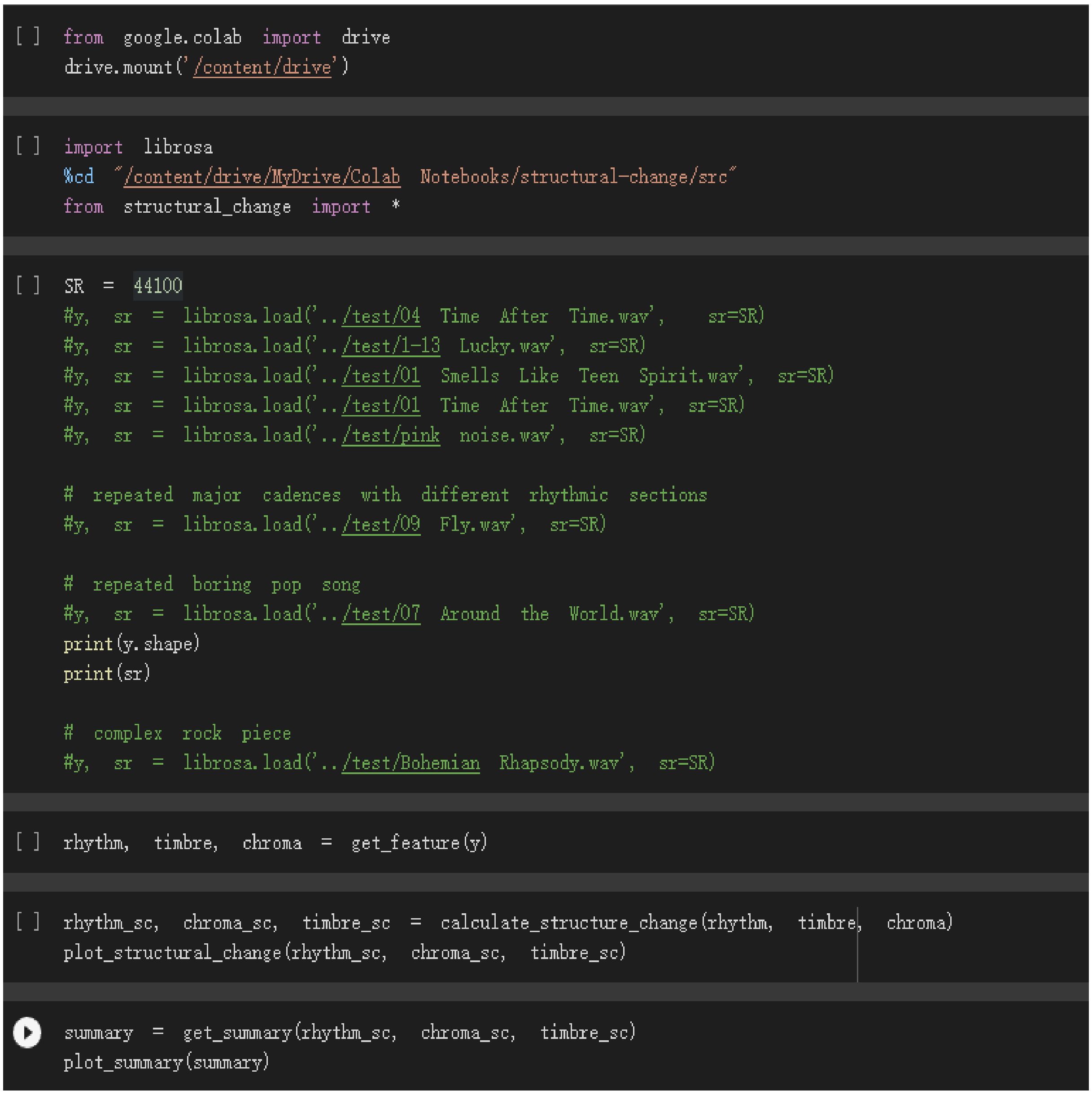
Figure 20: structural_change_3



```python
def plot_summary(summary):
    #visualisation 36-dimension vector
    summary = abs(sklearn.preprocessing.normalize(summary, norm="l1"))
    z = np.zeros((6,1))
    summary = np.concatenate((z, summary), axis=1)
    summary = np.concatenate((summary, z), axis=1)

    print(summary)

    x = np.linspace(1,8,8)
    xvals = np.linspace(1,8,100)

    #rhythm mean and median
    y_mean = summary[0]
    y_median = summary[3]

    f = interp1d(x, y_mean, kind='cubic')
    f1 = interp1d(x, y_median, kind='cubic')

    plt.fill_between(xvals, f(xvals), -f(xvals), facecolor='red',alpha=0.4)
    plt.fill_between(xvals, f1(xvals), -f1(xvals), facecolor='red', alpha=1)
    plt.title('rhythm')
    plt.show()

    #chroma mean and median
    y_mean = summary[1]
    y_median = summary[4]

    f = interp1d(x, y_mean, kind='cubic')
    f1 = interp1d(x, y_median, kind='cubic')

    plt.fill_between(xvals, f(xvals), -f(xvals), facecolor='green',alpha=0.4)
    plt.fill_between(xvals, f1(xvals), -f1(xvals), facecolor='green', alpha=1)
    plt.title('chroma')
    plt.show()

    #timbre mean and median
    y_mean = summary[2]
    y_median = summary[5]
    f = interp1d(x, y_mean, kind='cubic')
    f1 = interp1d(x, y_median, kind='cubic')

    plt.fill_between(xvals, f(xvals), -f(xvals), facecolor='blue',alpha=0.4)
    plt.fill_between(xvals, f1(xvals), -f1(xvals), facecolor='blue', alpha=1)
    plt.title('timbre')
    plt.show()
```

Figure 21: structural_change_4

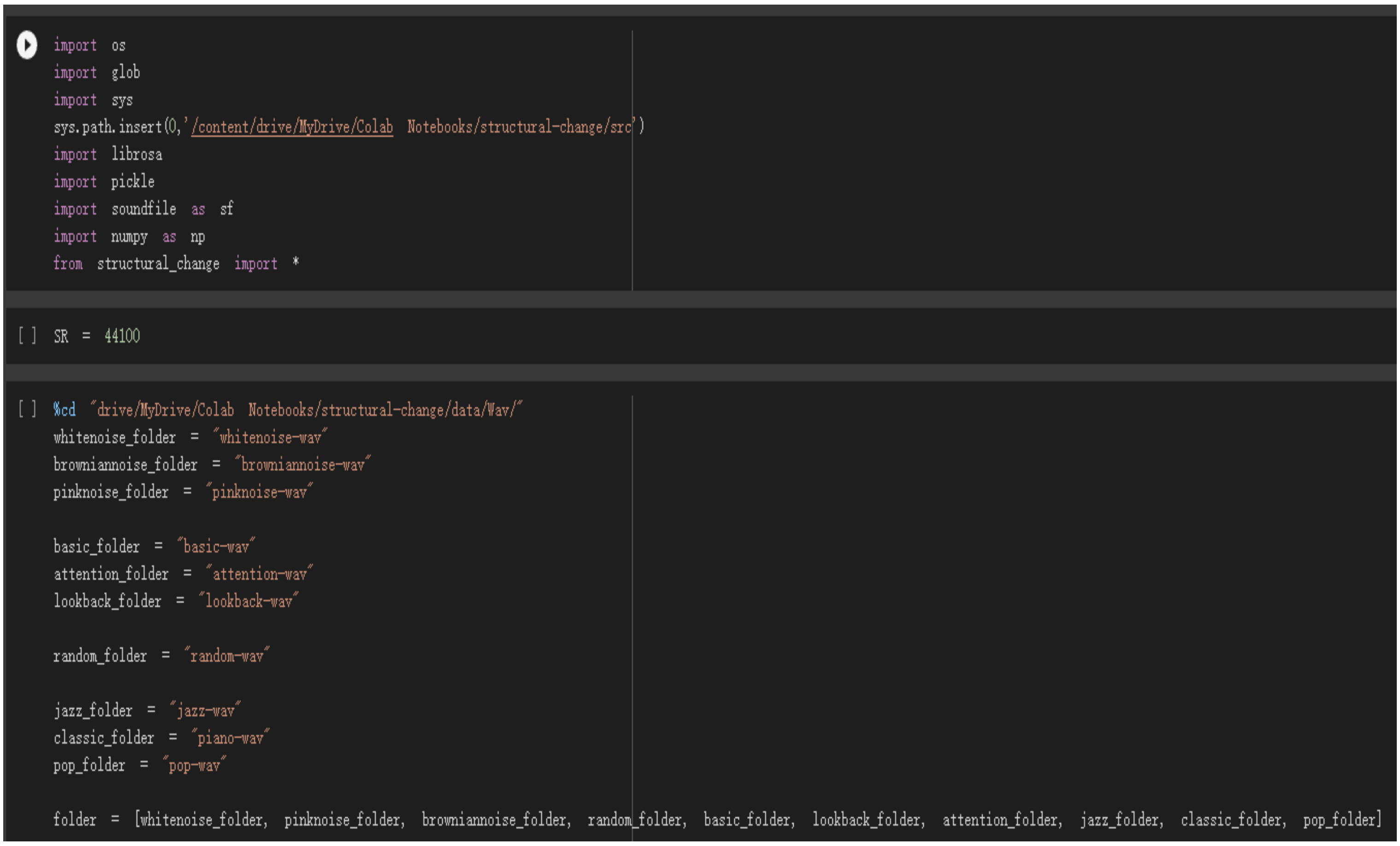
A.2.6 test.ipynb



```
[ ] from google.colab import drive
    drive.mount('/content/drive')

[ ] import librosa
    %cd "/content/drive/MyDrive/Colab Notebooks/structural-change/src"
    from structural_change import *

[ ] SR = 44100
    #y, sr = librosa.load('../test/04 Time After Time.wav',   sr=SR)
    #y, sr = librosa.load('../test/1-13 Lucky.wav', sr=SR)
    #y, sr = librosa.load('../test/01 Smells Like Teen Spirit.wav', sr=SR)
    #y, sr = librosa.load('../test/01 Time After Time.wav', sr=SR)
    #y, sr = librosa.load('../test/pink noise.wav', sr=SR)

    # repeated major cadences with different rhythmic sections
    #y, sr = librosa.load('../test/09 Fly.wav', sr=SR)

    # repeated boring pop song
    #y, sr = librosa.load('../test/07 Around the World.wav', sr=SR)
    print(y.shape)
    print(sr)

    # complex rock piece
    #y, sr = librosa.load('../test/Bohemian Rhapsody.wav', sr=SR)

[ ] rhythm, timbre, chroma = get_feature(y)

[ ] rhythm_sc, chroma_sc, timbre_sc = calculate_structure_change(rhythm, timbre, chroma)
    plot_structural_change(rhythm_sc, chroma_sc, timbre_sc)

    summary = get_summary(rhythm_sc, chroma_sc, timbre_sc)
    plot_summary(summary)
```

Figure 22: test

A.2.7 get_sc_change.ipynb

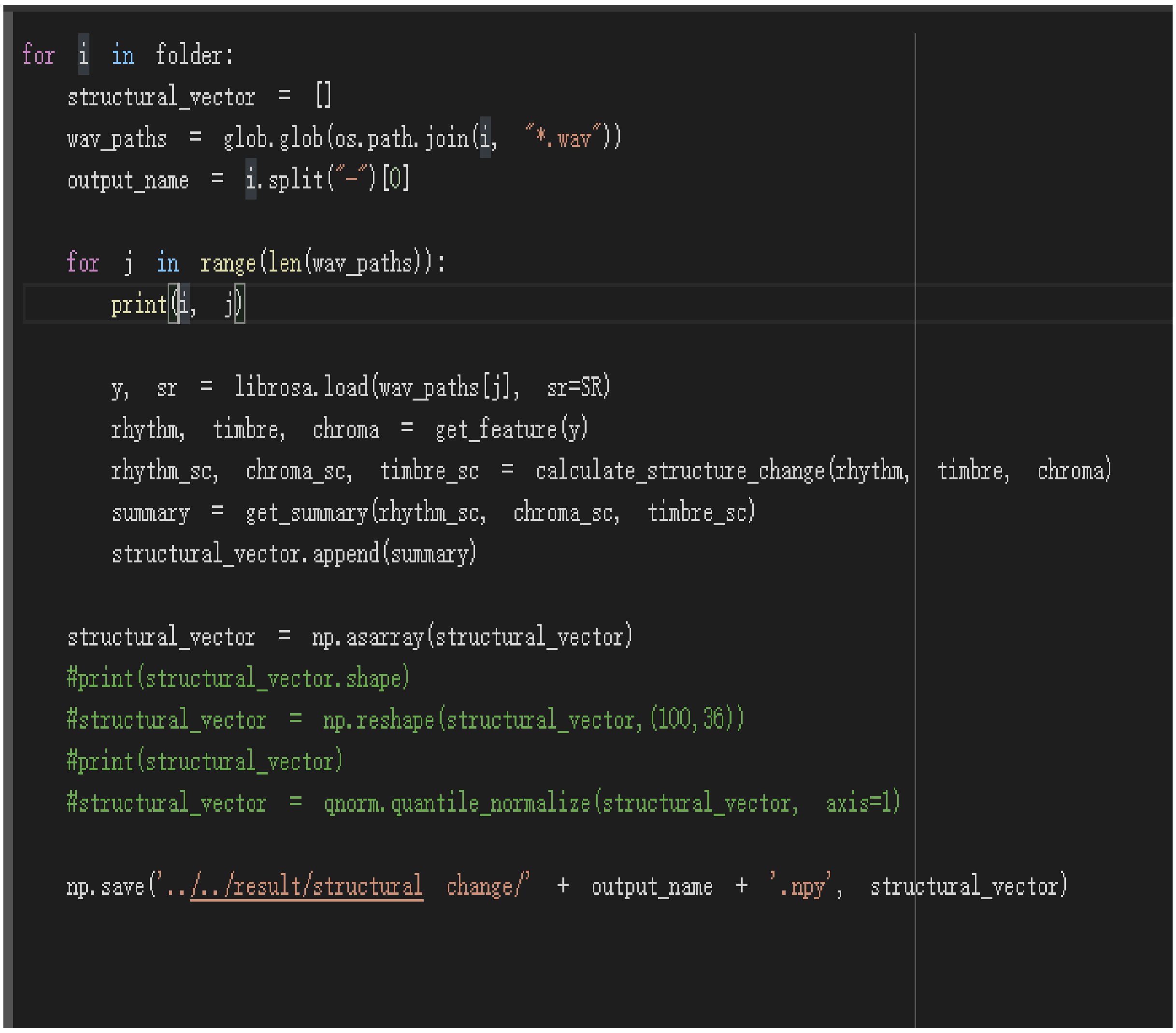


```
import os
import glob
import sys
sys.path.insert(0,'/content/drive/MyDrive/Colab Notebooks/structural-change/src')
import librosa
import pickle
import soundfile as sf
import numpy as np
from structural_change import *
```

```
SR = 44100
```

```
%cd "drive/MyDrive/Colab Notebooks/structural-change/data/Wav/"
whitenoise_folder = "whitenoise-wav"
browniannoise_folder = "browniannoise-wav"
pinknoise_folder = "pinknoise-wav"

basic_folder = "basic-wav"
attention_folder = "attention-wav"
lookback_folder = "lookback-wav"

random_folder = "random-wav"

jazz_folder = "jazz-wav"
classic_folder = "piano-wav"
pop_folder = "pop-wav"

folder = [whitenoise_folder, pinknoise_folder, browniannoise_folder, random_folder, basic_folder, lookback_folder, attention_folder, jazz_folder, classic_folder, pop_folder]
```

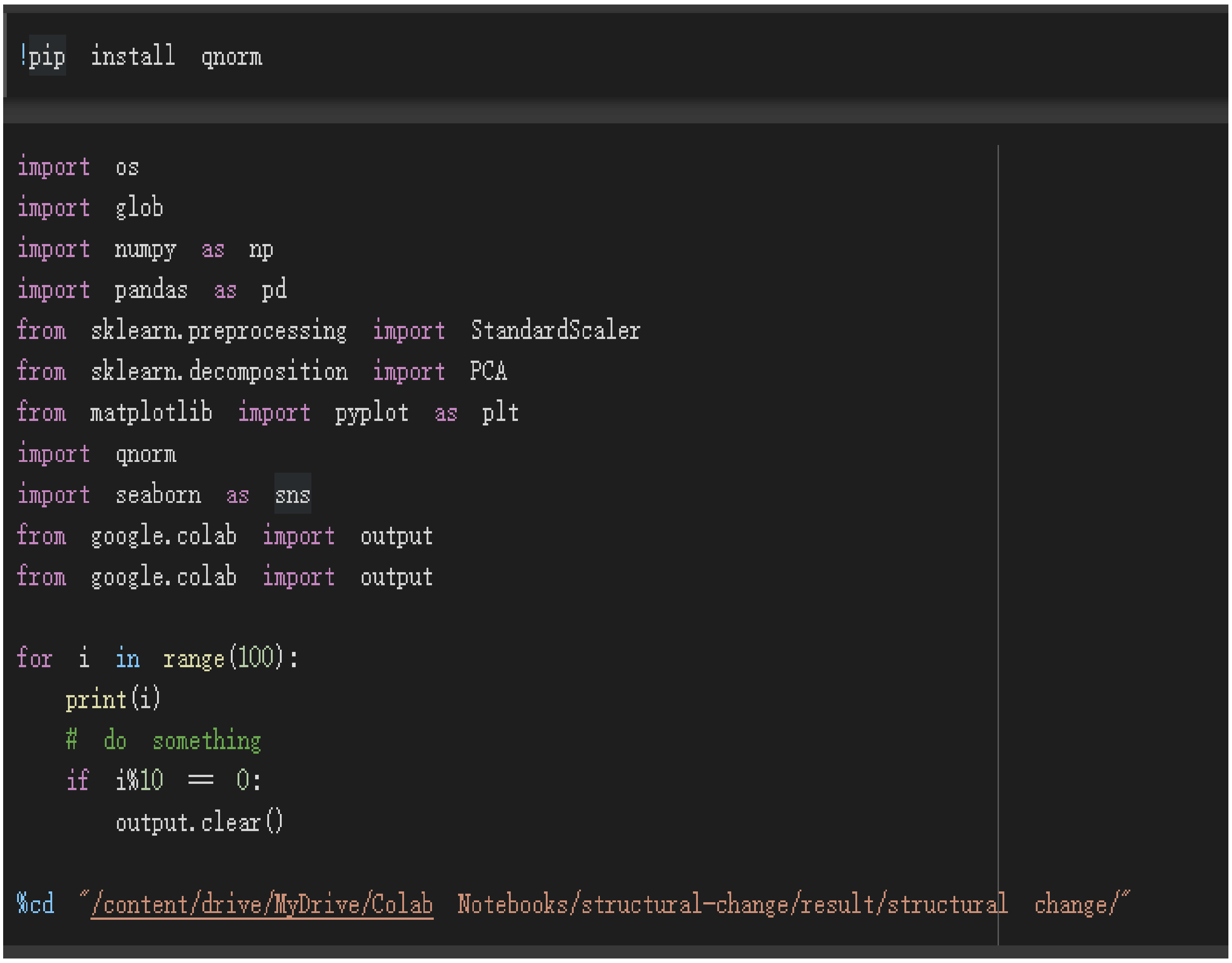
Figure 23: get_sc_change_1



```
for i in folder:
    structural_vector = []
    wav_paths = glob.glob(os.path.join(i, "*.wav"))
    output_name = i.split("-")[0]

    for j in range(len(wav_paths)):
        print(i, j)

        y, sr = librosa.load(wav_paths[j], sr=SR)
        rhythm, timbre, chroma = get_feature(y)
        rhythm_sc, chroma_sc, timbre_sc = calculate_structure_change(rhythm, timbre, chroma)
        summary = get_summary(rhythm_sc, chroma_sc, timbre_sc)
        structural_vector.append(summary)

    structural_vector = np.asarray(structural_vector)
    #print(structural_vector.shape)
    #structural_vector = np.reshape(structural_vector,(100,36))
    #print(structural_vector)
    #structural_vector = qnorm.quantile_normalize(structural_vector, axis=1)

    np.save('../../result/structural change/' + output_name + '.npy', structural_vector)
```

Figure 24: get_sc_change_2

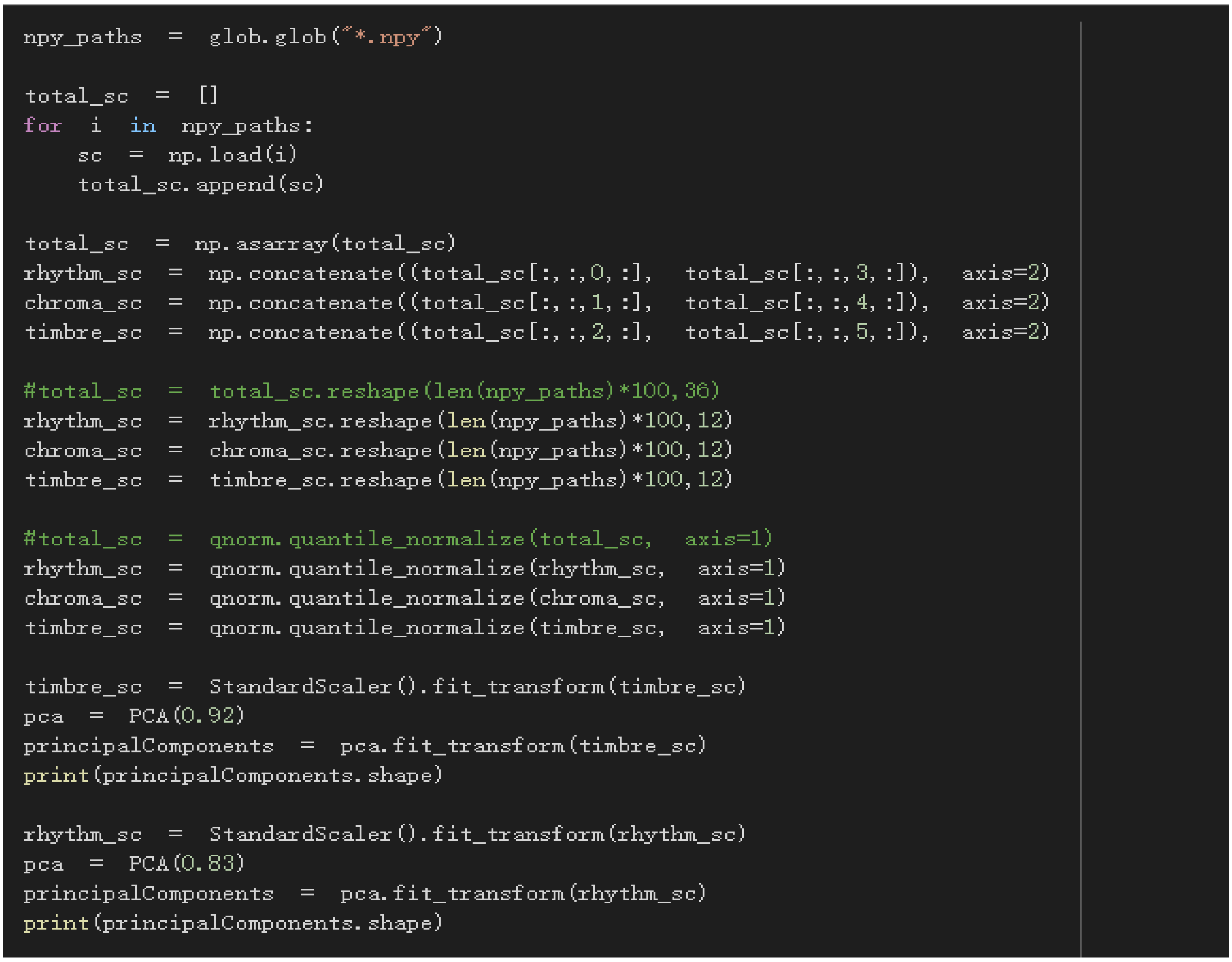
A.2.8 bivariate_KDE.ipynb



```
!pip install qnorm
```

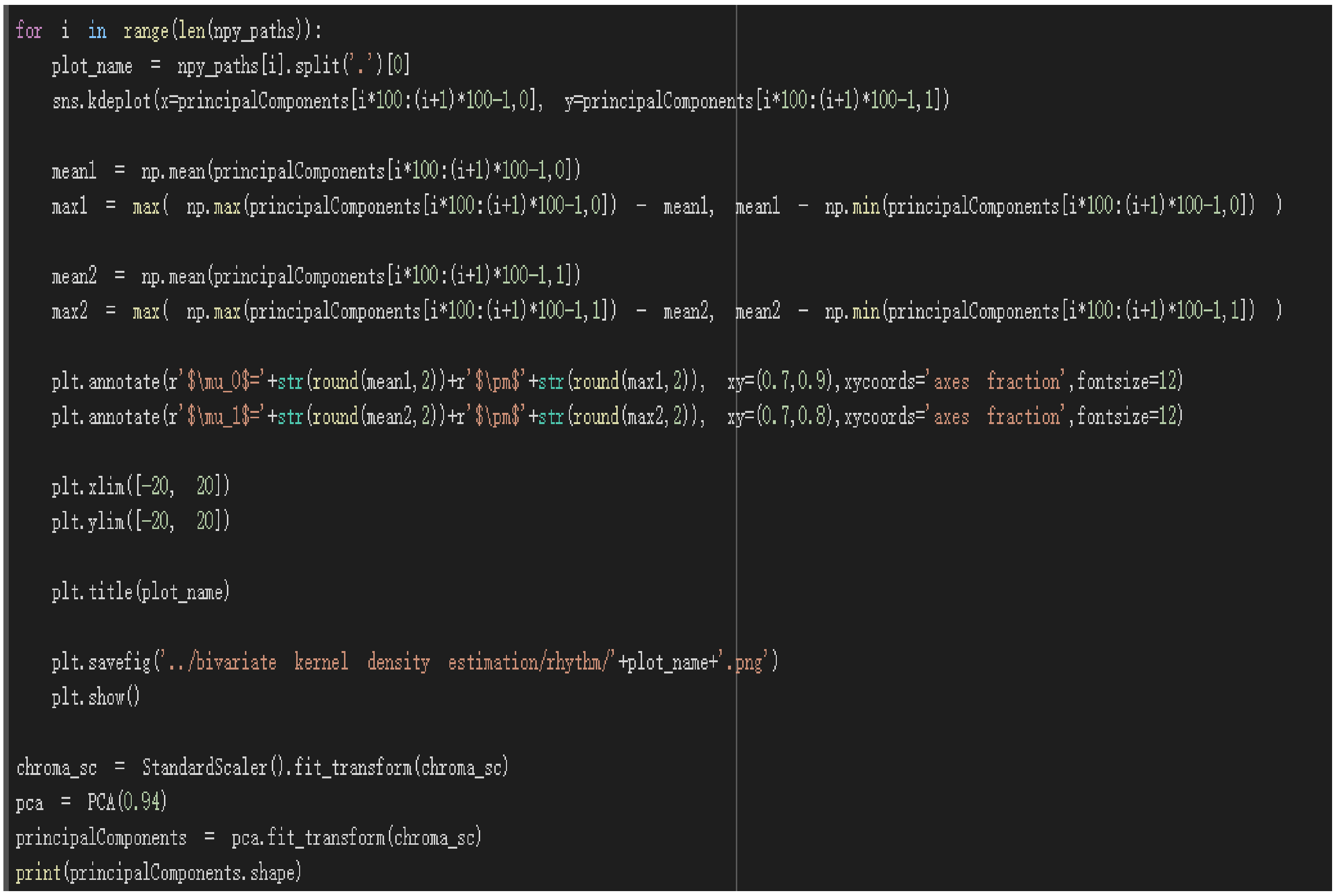
```
import os
import glob
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from matplotlib import pyplot as plt
import qnorm
import seaborn as sns
from google.colab import output
from google.colab import output

for i in range(100):
    print(i)
    # do something
    if i%10 == 0:
        output.clear()

%cd "/content/drive/MyDrive/Colab Notebooks/structural-change/result/structural change/"
```

Figure 25: bivariate_KDE_1



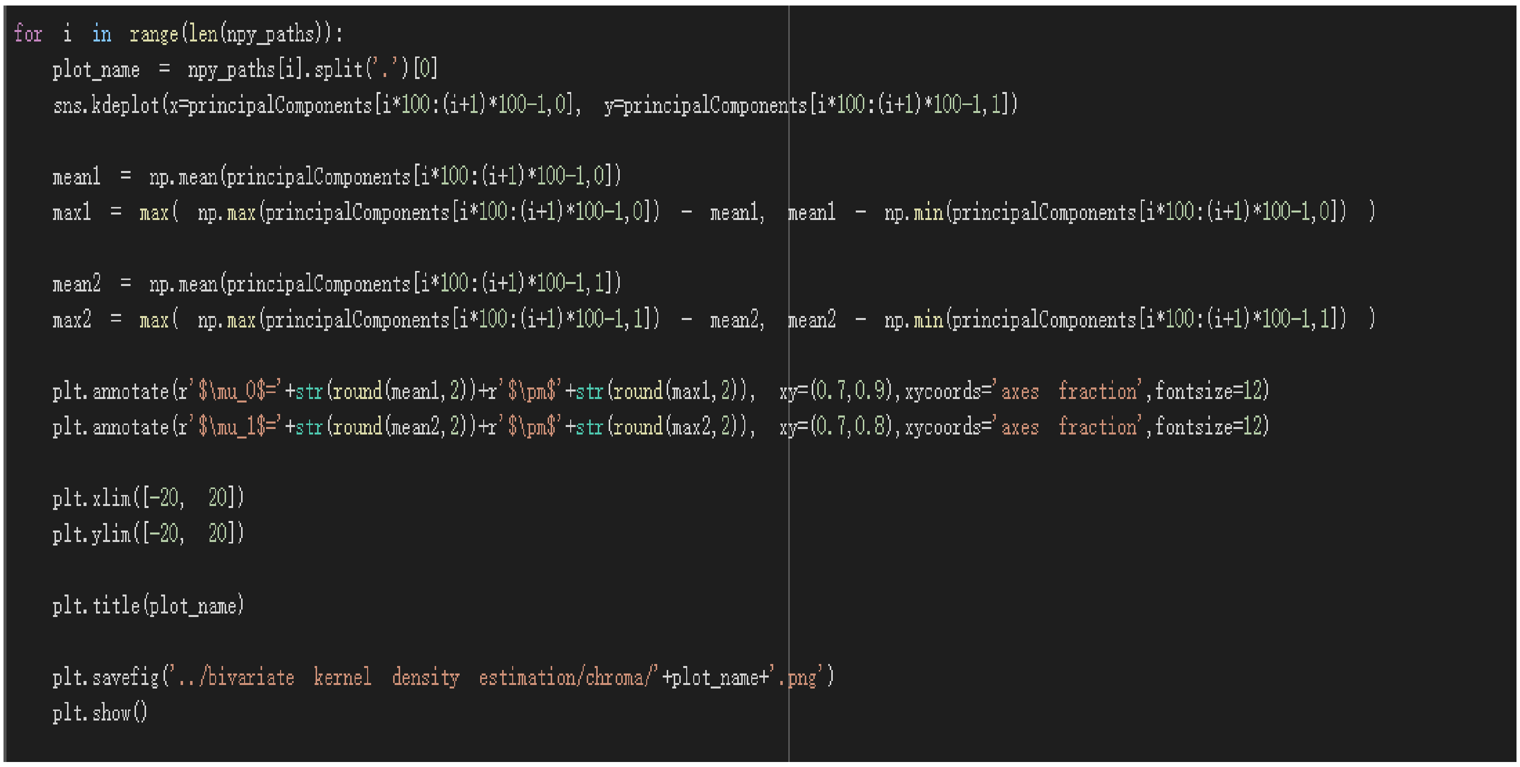
```
npy_paths = glob.glob("*.npy")

total_sc = []
for i in npy_paths:
    sc = np.load(i)
    total_sc.append(sc)

total_sc = np.asarray(total_sc)
rhythm_sc = np.concatenate((total_sc[:,:,0,:], total_sc[:,:,3,:]), axis=2)
chroma_sc = np.concatenate((total_sc[:,:,1,:], total_sc[:,:,4,:]), axis=2)
timbre_sc = np.concatenate((total_sc[:,:,2,:], total_sc[:,:,5,:]), axis=2)

#total_sc = total_sc.reshape(len(npy_paths)*100,36)
rhythm_sc = rhythm_sc.reshape(len(npy_paths)*100,12)
chroma_sc = chroma_sc.reshape(len(npy_paths)*100,12)
timbre_sc = timbre_sc.reshape(len(npy_paths)*100,12)

#total_sc = qnorm.quantile_normalize(total_sc, axis=1)
rhythm_sc = qnorm.quantile_normalize(rhythm_sc, axis=1)
chroma_sc = qnorm.quantile_normalize(chroma_sc, axis=1)
timbre_sc = qnorm.quantile_normalize(timbre_sc, axis=1)

timbre_sc = StandardScaler().fit_transform(timbre_sc)
pca = PCA(0.92)
principalComponents = pca.fit_transform(timbre_sc)
print(principalComponents.shape)

rhythm_sc = StandardScaler().fit_transform(rhythm_sc)
pca = PCA(0.83)
principalComponents = pca.fit_transform(rhythm_sc)
print(principalComponents.shape)
```

Figure 26: bivariate_KDE_2



```
for i in range(len(npy_paths)):
    plot_name = npy_paths[i].split('.')[0]
    sns.kdeplot(x=principalComponents[i*100:(i+1)*100-1,0], y=principalComponents[i*100:(i+1)*100-1,1])

    mean1 = np.mean(principalComponents[i*100:(i+1)*100-1,0])
    max1 = max( np.max(principalComponents[i*100:(i+1)*100-1,0]) - mean1, mean1 - np.min(principalComponents[i*100:(i+1)*100-1,0]) )

    mean2 = np.mean(principalComponents[i*100:(i+1)*100-1,1])
    max2 = max( np.max(principalComponents[i*100:(i+1)*100-1,1]) - mean2, mean2 - np.min(principalComponents[i*100:(i+1)*100-1,1]) )

    plt.annotate(r'$\mu_0$='+str(round(mean1,2))+r'$\pm$'+str(round(max1,2)), xy=(0.7,0.9),xycoords='axes fraction',fontsize=12)
    plt.annotate(r'$\mu_1$='+str(round(mean2,2))+r'$\pm$'+str(round(max2,2)), xy=(0.7,0.8),xycoords='axes fraction',fontsize=12)

    plt.xlim([-20, 20])
    plt.ylim([-20, 20])

    plt.title(plot_name)

    plt.savefig('../bivariate kernel density estimation/rhythm/'+plot_name+'.png')
    plt.show()

chroma_sc = StandardScaler().fit_transform(chroma_sc)
pca = PCA(0.94)
principalComponents = pca.fit_transform(chroma_sc)
print(principalComponents.shape)
```

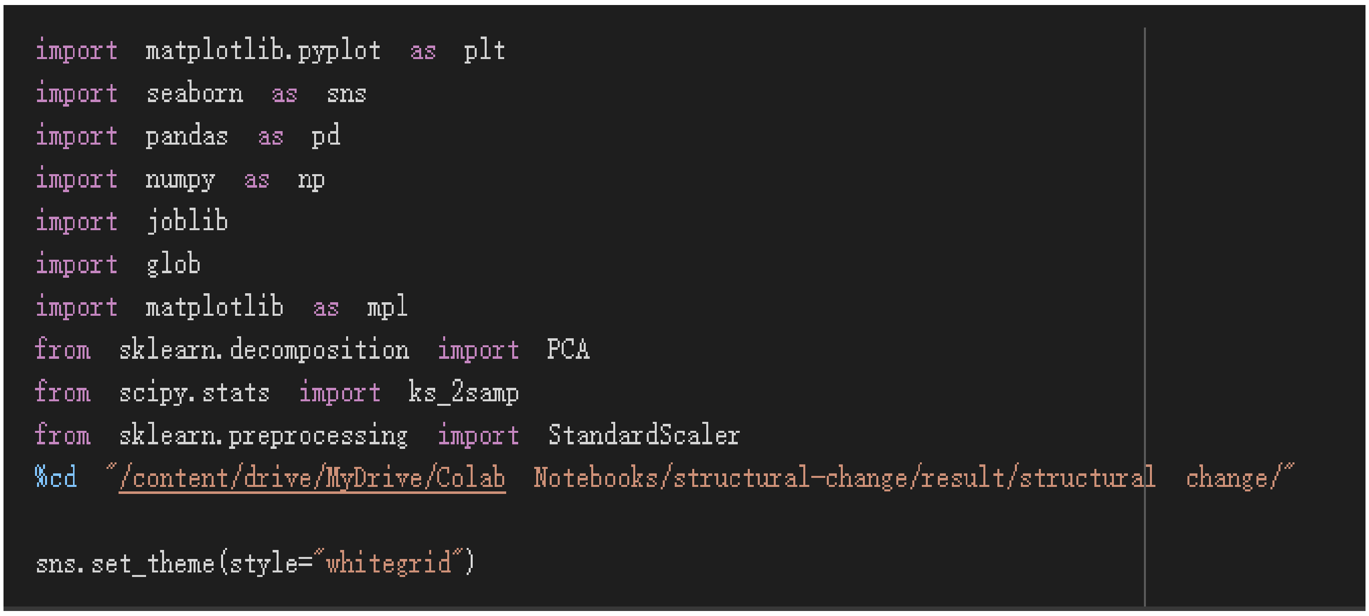
Figure 27: bivariate_KDE_3



```
for i in range(len(npy_paths)):
    plot_name = npy_paths[i].split('.')[0]
    sns.kdeplot(x=principalComponents[i*100:(i+1)*100-1,0], y=principalComponents[i*100:(i+1)*100-1,1])

    mean1 = np.mean(principalComponents[i*100:(i+1)*100-1,0])
    max1 = max( np.max(principalComponents[i*100:(i+1)*100-1,0]) - mean1, mean1 - np.min(principalComponents[i*100:(i+1)*100-1,0]) )

    mean2 = np.mean(principalComponents[i*100:(i+1)*100-1,1])
    max2 = max( np.max(principalComponents[i*100:(i+1)*100-1,1]) - mean2, mean2 - np.min(principalComponents[i*100:(i+1)*100-1,1]) )

    plt.annotate(r'$\mu_0$='+str(round(mean1,2))+r'$\pm$'+str(round(max1,2)), xy=(0.7,0.9),xycoords='axes fraction',fontsize=12)
    plt.annotate(r'$\mu_1$='+str(round(mean2,2))+r'$\pm$'+str(round(max2,2)), xy=(0.7,0.8),xycoords='axes fraction',fontsize=12)

    plt.xlim([-20, 20])
    plt.ylim([-20, 20])

    plt.title(plot_name)

    plt.savefig('../bivariate kernel density estimation/chroma/'+plot_name+'.png')
    plt.show()
```

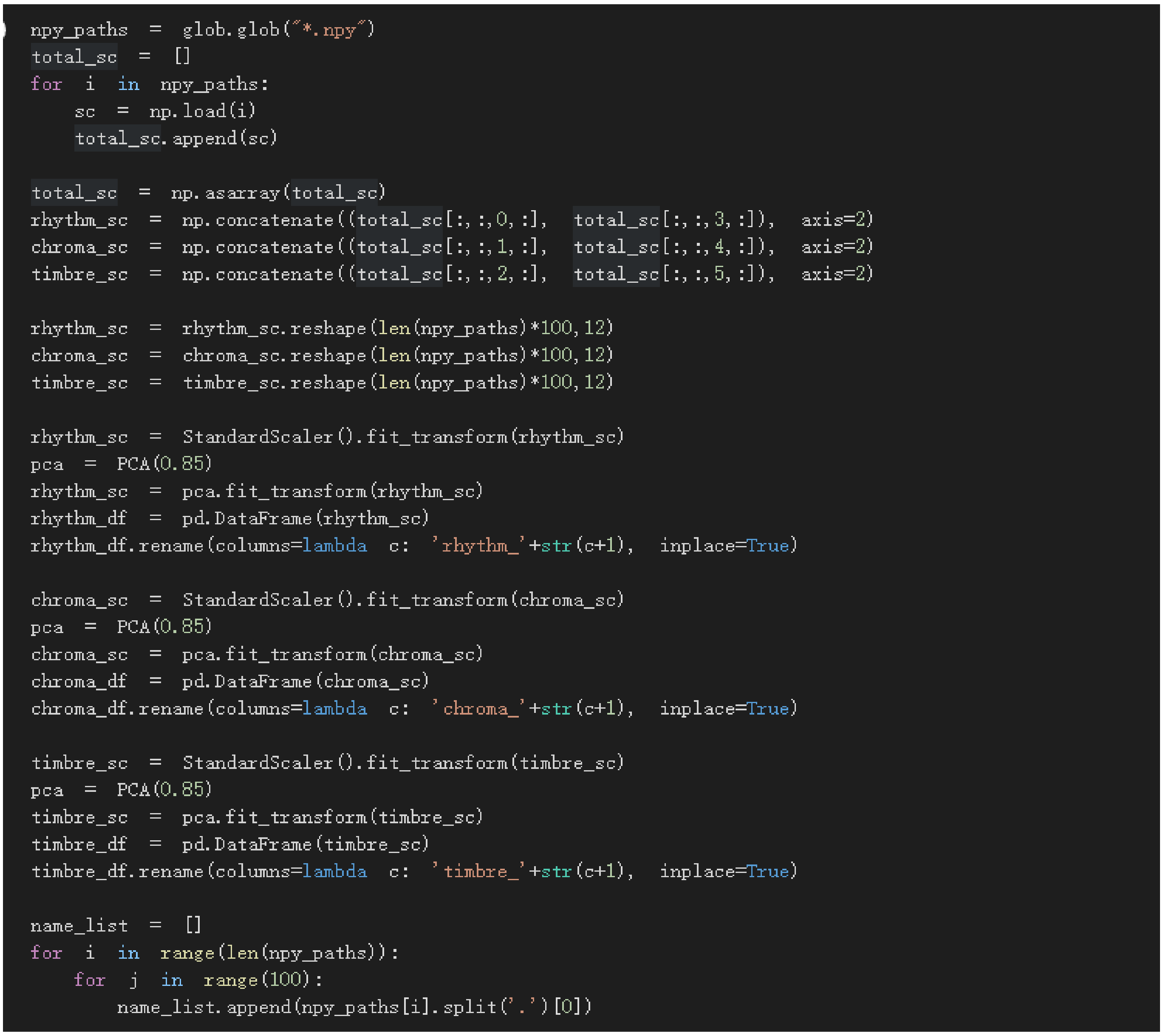
Figure 28: bivariate_KDE_4

A.2.9 pairwise_comparison.ipynb



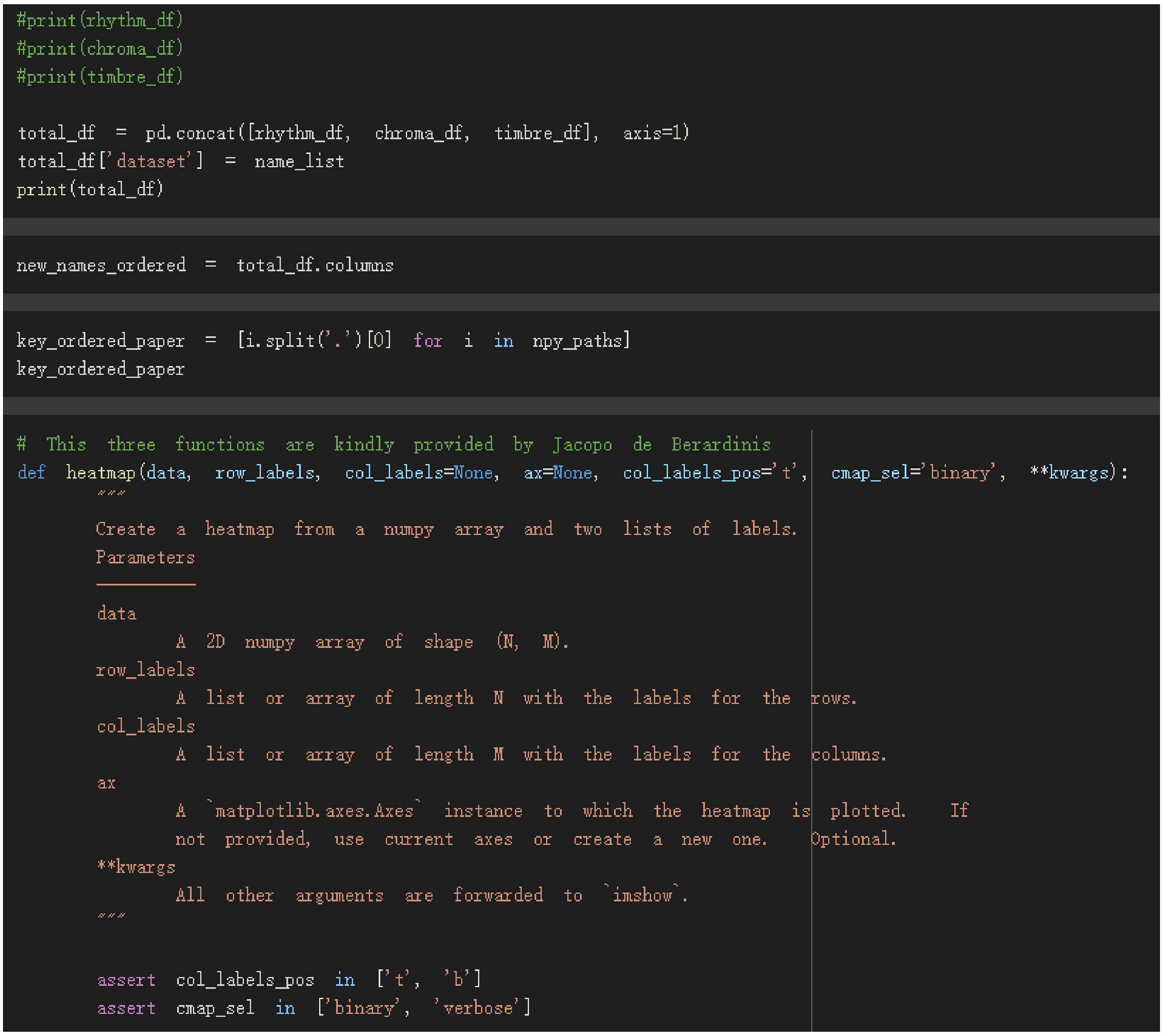
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import joblib
import glob
import matplotlib as mpl
from sklearn.decomposition import PCA
from scipy.stats import ks_2samp
from sklearn.preprocessing import StandardScaler
%cd "/content/drive/MyDrive/Colab Notebooks/structural-change/result/structural change/"

sns.set_theme(style="whitegrid")
```

Figure 29: pairwise_comparison_1



```
npy_paths = glob.glob("*.npy")
total_sc = []
for i in npy_paths:
    sc = np.load(i)
    total_sc.append(sc)

total_sc = np.asarray(total_sc)
rhythm_sc = np.concatenate((total_sc[:,:,0,:], total_sc[:,:,3,:]), axis=2)
chroma_sc = np.concatenate((total_sc[:,:,1,:], total_sc[:,:,4,:]), axis=2)
timbre_sc = np.concatenate((total_sc[:,:,2,:], total_sc[:,:,5,:]), axis=2)

rhythm_sc = rhythm_sc.reshape(len(npy_paths)*100,12)
chroma_sc = chroma_sc.reshape(len(npy_paths)*100,12)
timbre_sc = timbre_sc.reshape(len(npy_paths)*100,12)

rhythm_sc = StandardScaler().fit_transform(rhythm_sc)
pca = PCA(0.85)
rhythm_sc = pca.fit_transform(rhythm_sc)
rhythm_df = pd.DataFrame(rhythm_sc)
rhythm_df.rename(columns=lambda c: 'rhythm_'+str(c+1), inplace=True)

chroma_sc = StandardScaler().fit_transform(chroma_sc)
pca = PCA(0.85)
chroma_sc = pca.fit_transform(chroma_sc)
chroma_df = pd.DataFrame(chroma_sc)
chroma_df.rename(columns=lambda c: 'chroma_'+str(c+1), inplace=True)

timbre_sc = StandardScaler().fit_transform(timbre_sc)
pca = PCA(0.85)
timbre_sc = pca.fit_transform(timbre_sc)
timbre_df = pd.DataFrame(timbre_sc)
timbre_df.rename(columns=lambda c: 'timbre_'+str(c+1), inplace=True)

name_list = []
for i in range(len(npy_paths)):
    for j in range(100):
        name_list.append(npy_paths[i].split('.')[0])
```

Figure 30: pairwise_comparison_2



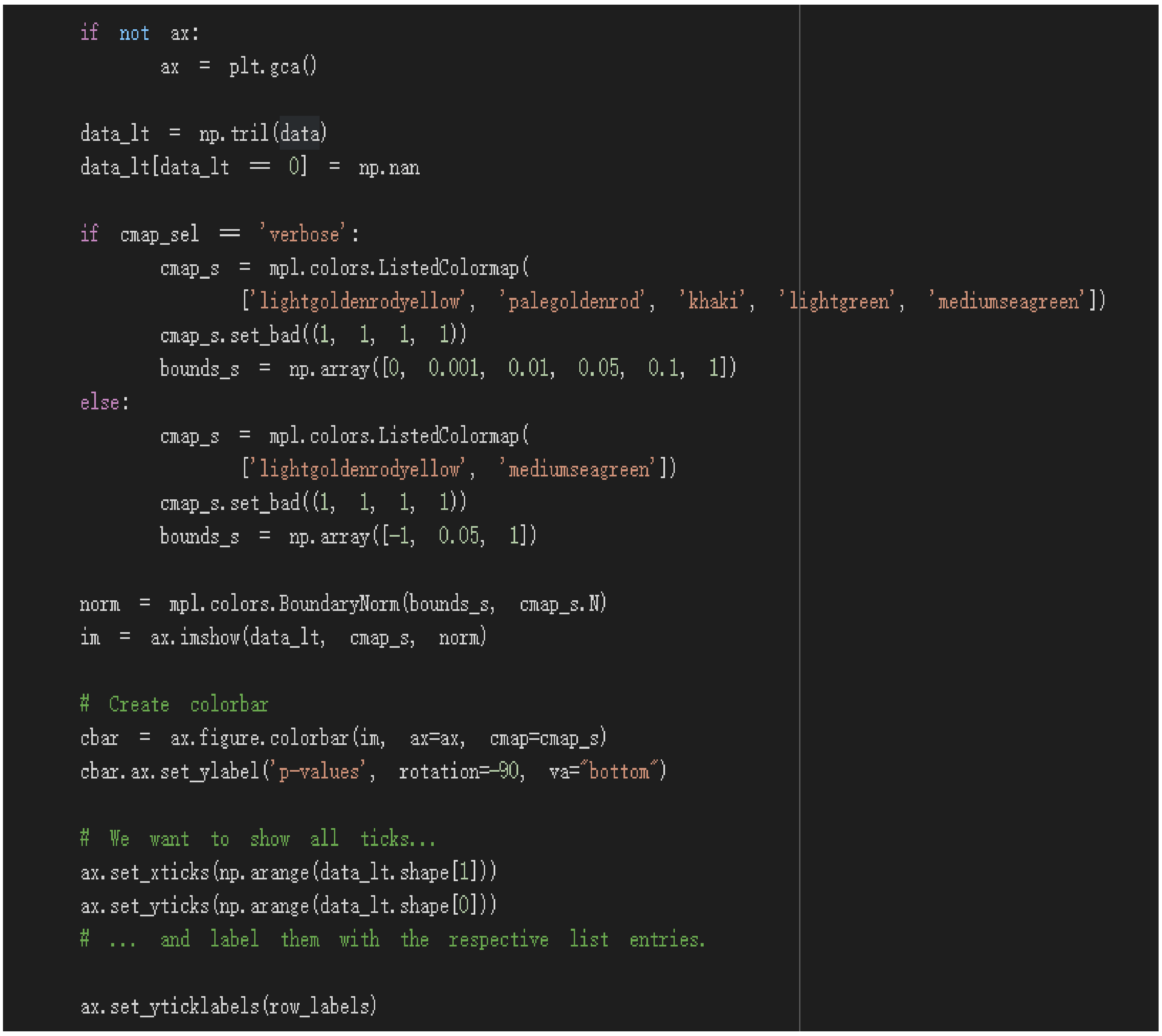
```python
#print(rhythm_df)
#print(chroma_df)
#print(timbre_df)

total_df = pd.concat([rhythm_df, chroma_df, timbre_df], axis=1)
total_df['dataset'] = name_list
print(total_df)
```

```python
new_names_ordered = total_df.columns
```

```python
key_ordered_paper = [i.split('.')[0] for i in npy_paths]
key_ordered_paper
```

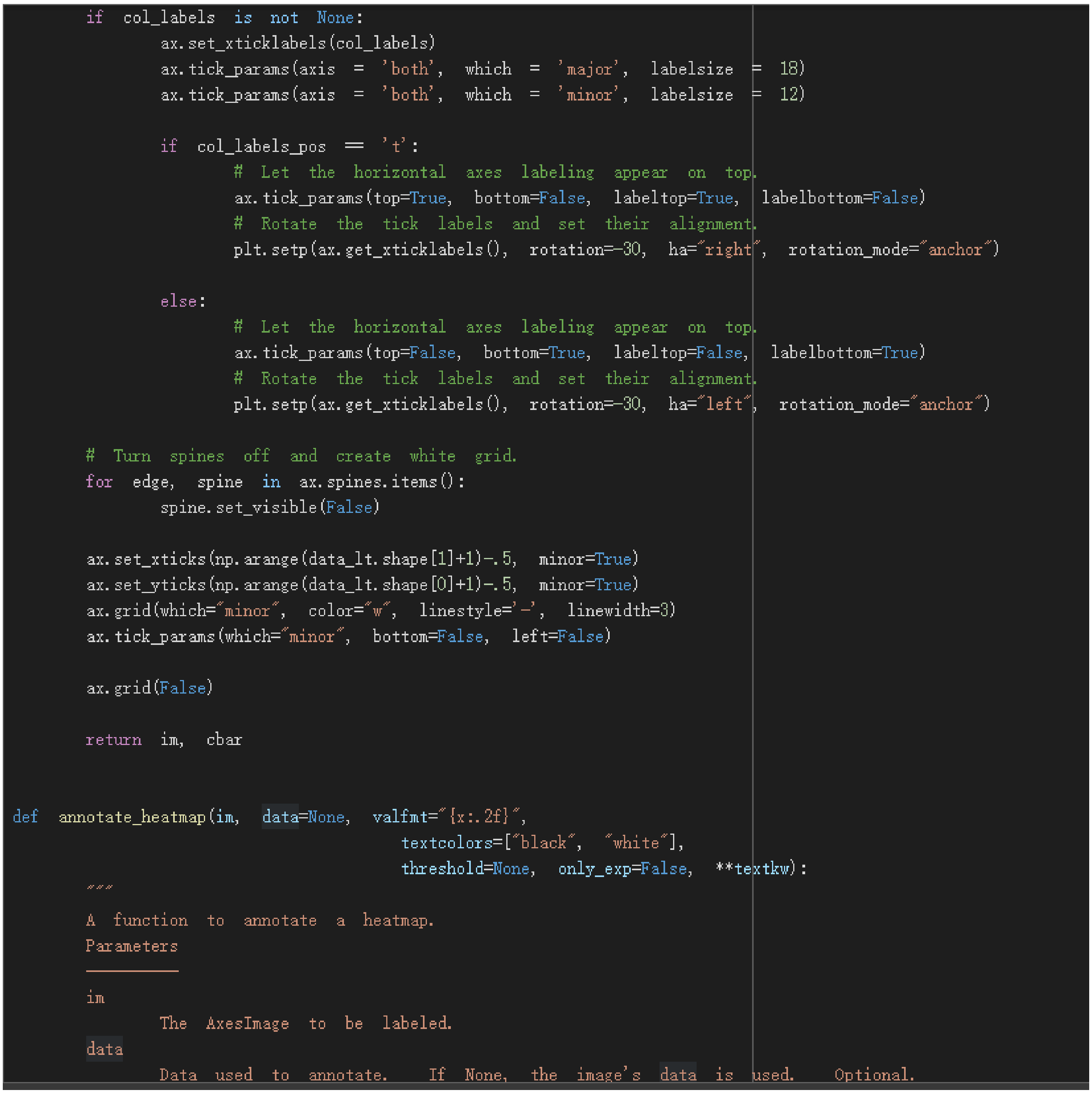
```python
# This three functions are kindly provided by Jacopo de Berardinis
def heatmap(data, row_labels, col_labels=None, ax=None, col_labels_pos='t', cmap_sel='binary', **kwargs):
    """
    Create a heatmap from a numpy array and two lists of labels.
    Parameters
    ----------
    data
        A 2D numpy array of shape (N, M).
    row_labels
        A list or array of length N with the labels for the rows.
    col_labels
        A list or array of length M with the labels for the columns.
    ax
        A `matplotlib.axes.Axes` instance to which the heatmap is plotted.  If
        not provided, use current axes or create a new one.  Optional.
    **kwargs
        All other arguments are forwarded to `imshow`.
    """

    assert col_labels_pos in ['t', 'b']
    assert cmap_sel in ['binary', 'verbose']
```

Figure 31: pairwise_comparison_3



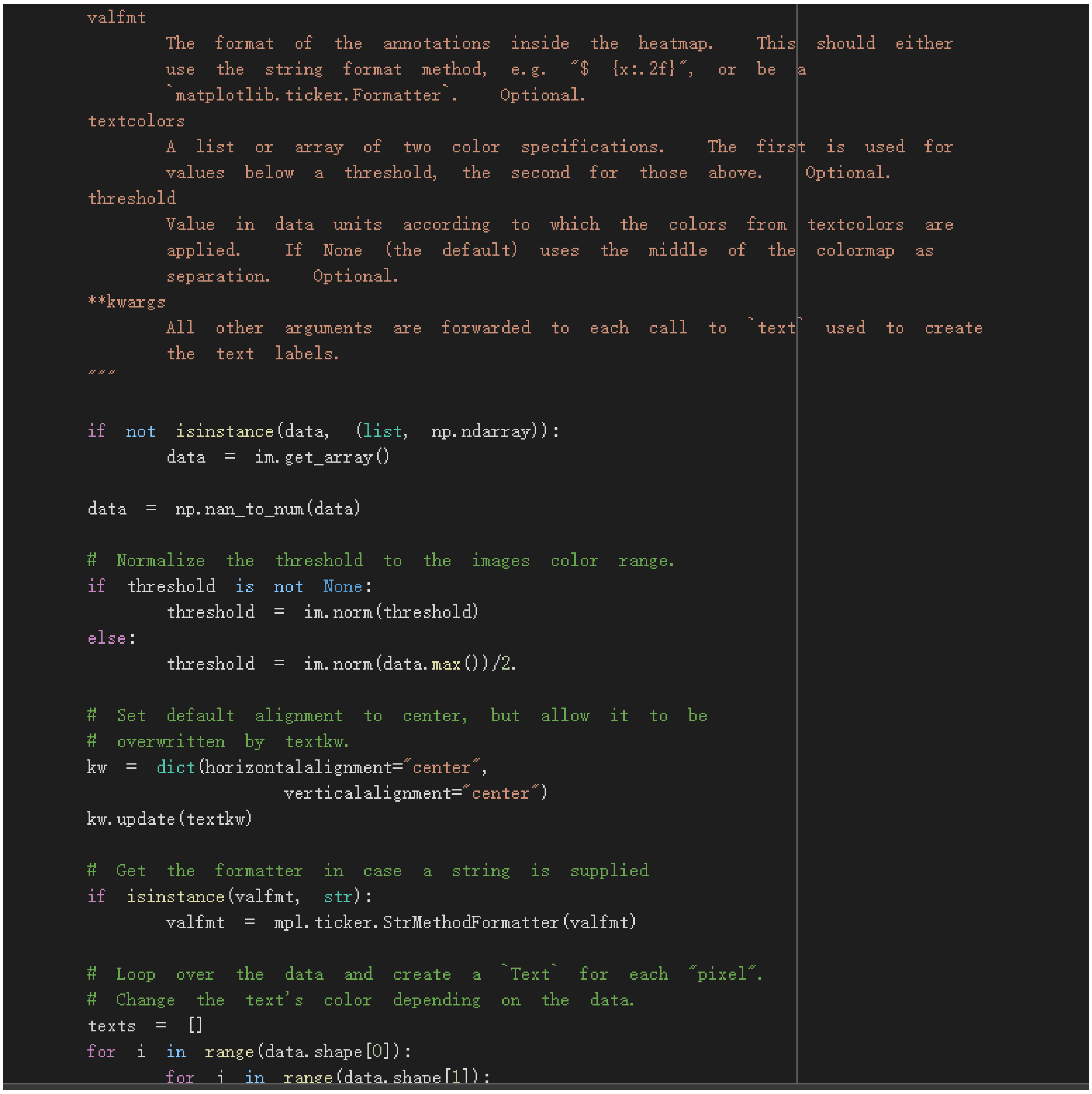
```
    if not ax:
        ax = plt.gca()

    data_lt = np.tril(data)
    data_lt[data_lt == 0] = np.nan

    if cmap_sel == 'verbose':
        cmap_s = mpl.colors.ListedColormap(
            ['lightgoldenrodyellow', 'palegoldenrod', 'khaki', 'lightgreen', 'mediumseagreen'])
        cmap_s.set_bad((1, 1, 1, 1))
        bounds_s = np.array([0, 0.001, 0.01, 0.05, 0.1, 1])
    else:
        cmap_s = mpl.colors.ListedColormap(
            ['lightgoldenrodyellow', 'mediumseagreen'])
        cmap_s.set_bad((1, 1, 1, 1))
        bounds_s = np.array([-1, 0.05, 1])

    norm = mpl.colors.BoundaryNorm(bounds_s, cmap_s.N)
    im = ax.imshow(data_lt, cmap_s, norm)

    # Create colorbar
    cbar = ax.figure.colorbar(im, ax=ax, cmap=cmap_s)
    cbar.ax.set_ylabel('p-values', rotation=-90, va="bottom")

    # We want to show all ticks...
    ax.set_xticks(np.arange(data_lt.shape[1]))
    ax.set_yticks(np.arange(data_lt.shape[0]))
    # ... and label them with the respective list entries.

    ax.set_yticklabels(row_labels)
```

Figure 32: pairwise_comparison_4



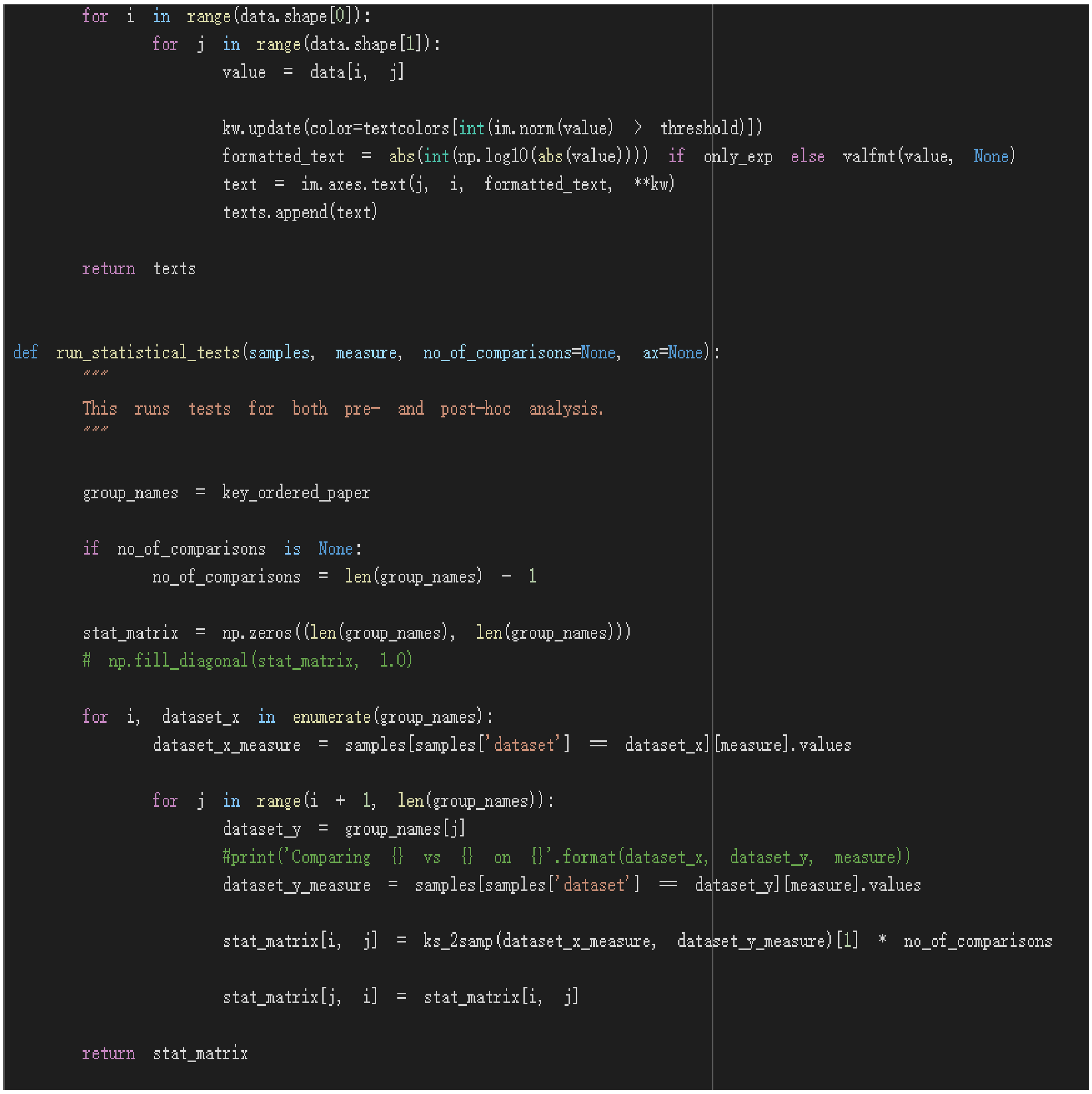
```
    if col_labels is not None:
        ax.set_xticklabels(col_labels)
        ax.tick_params(axis = 'both', which = 'major', labelsize = 18)
        ax.tick_params(axis = 'both', which = 'minor', labelsize = 12)

        if col_labels_pos == 't':
            # Let the horizontal axes labeling appear on top.
            ax.tick_params(top=True, bottom=False, labeltop=True, labelbottom=False)
            # Rotate the tick labels and set their alignment.
            plt.setp(ax.get_xticklabels(), rotation=-30, ha="right", rotation_mode="anchor")

        else:
            # Let the horizontal axes labeling appear on top.
            ax.tick_params(top=False, bottom=True, labeltop=False, labelbottom=True)
            # Rotate the tick labels and set their alignment.
            plt.setp(ax.get_xticklabels(), rotation=-30, ha="left", rotation_mode="anchor")

    # Turn spines off and create white grid.
    for edge, spine in ax.spines.items():
        spine.set_visible(False)

    ax.set_xticks(np.arange(data_lt.shape[1]+1)-.5, minor=True)
    ax.set_yticks(np.arange(data_lt.shape[0]+1)-.5, minor=True)
    ax.grid(which="minor", color="w", linestyle='-', linewidth=3)
    ax.tick_params(which="minor", bottom=False, left=False)

    ax.grid(False)

    return im, cbar


def annotate_heatmap(im, data=None, valfmt="{x:.2f}",
                     textcolors=["black", "white"],
                     threshold=None, only_exp=False, **textkw):
    """
    A function to annotate a heatmap.
    Parameters
    ----------
    im
        The AxesImage to be labeled.
    data
        Data used to annotate.  If None, the image's data is used.  Optional.
```

Figure 33: pairwise_comparison_5



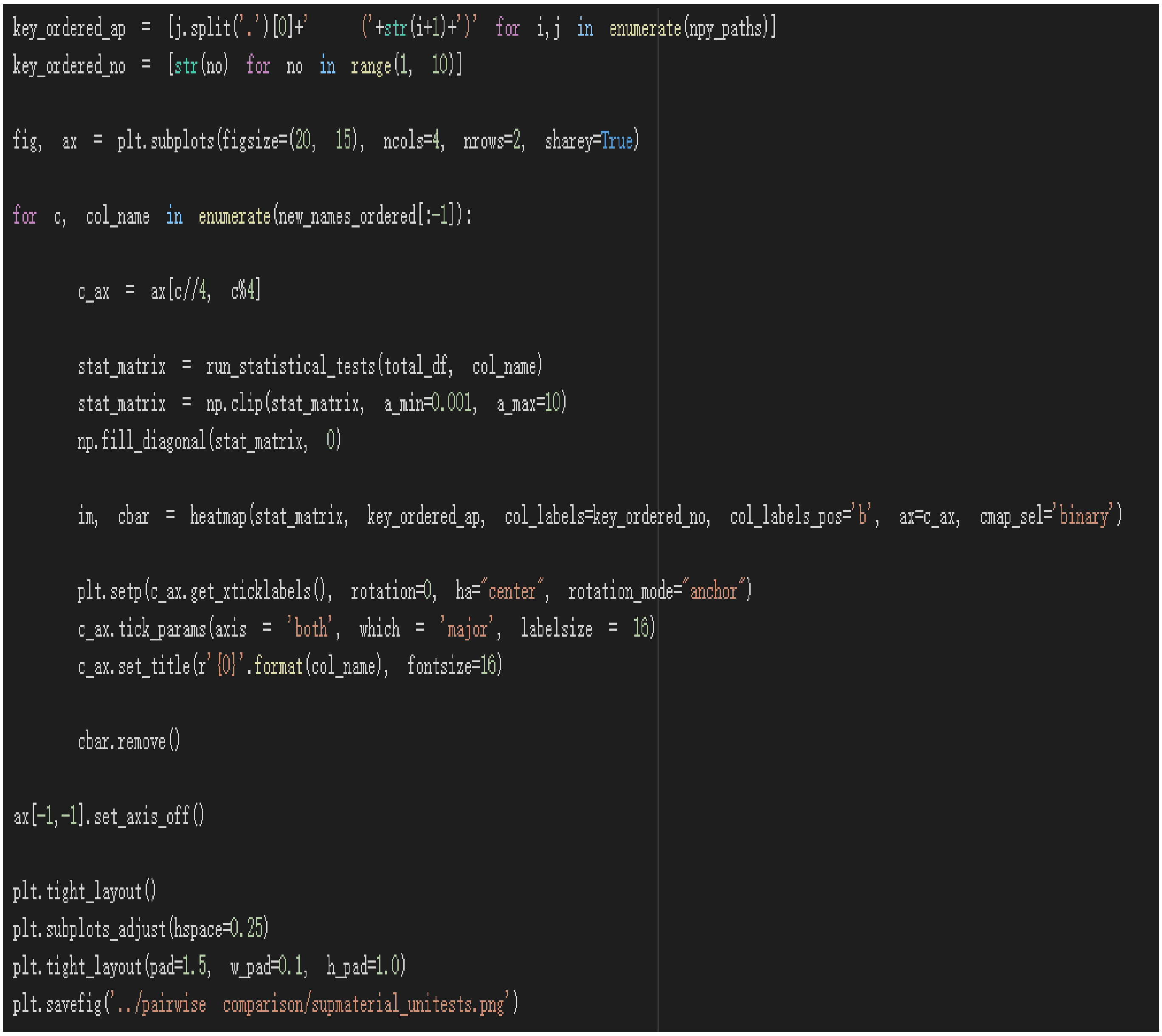
```
valfmt
    The format of the annotations inside the heatmap.  This should either
    use the string format method, e.g. "$ {x:.2f}", or be a
    `matplotlib.ticker.Formatter`.  Optional.
textcolors
    A list or array of two color specifications.  The first is used for
    values below a threshold, the second for those above.  Optional.
threshold
    Value in data units according to which the colors from textcolors are
    applied.  If None (the default) uses the middle of the colormap as
    separation.  Optional.
**kwargs
    All other arguments are forwarded to each call to `text` used to create
    the text labels.
"""

if not isinstance(data, (list, np.ndarray)):
    data = im.get_array()

data = np.nan_to_num(data)

# Normalize the threshold to the images color range.
if threshold is not None:
    threshold = im.norm(threshold)
else:
    threshold = im.norm(data.max())/2.

# Set default alignment to center, but allow it to be
# overwritten by textkw.
kw = dict(horizontalalignment="center",
          verticalalignment="center")
kw.update(textkw)

# Get the formatter in case a string is supplied
if isinstance(valfmt, str):
    valfmt = mpl.ticker.StrMethodFormatter(valfmt)

# Loop over the data and create a `Text` for each "pixel".
# Change the text's color depending on the data.
texts = []
for i in range(data.shape[0]):
    for j in range(data.shape[1]):
```

Figure 34: pairwise_comparison_6



```
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            value = data[i, j]

            kw.update(color=textcolors[int(im.norm(value) > threshold)])
            formatted_text = abs(int(np.log10(abs(value)))) if only_exp else valfmt(value, None)
            text = im.axes.text(j, i, formatted_text, **kw)
            texts.append(text)

    return texts


def run_statistical_tests(samples, measure, no_of_comparisons=None, ax=None):
    """
    This runs tests for both pre- and post-hoc analysis.
    """

    group_names = key_ordered_paper

    if no_of_comparisons is None:
        no_of_comparisons = len(group_names) - 1

    stat_matrix = np.zeros((len(group_names), len(group_names)))
    # np.fill_diagonal(stat_matrix, 1.0)

    for i, dataset_x in enumerate(group_names):
        dataset_x_measure = samples[samples['dataset'] == dataset_x][measure].values

        for j in range(i + 1, len(group_names)):
            dataset_y = group_names[j]
            #print('Comparing {} vs {} on {}'.format(dataset_x, dataset_y, measure))
            dataset_y_measure = samples[samples['dataset'] == dataset_y][measure].values

            stat_matrix[i, j] = ks_2samp(dataset_x_measure, dataset_y_measure)[1] * no_of_comparisons

            stat_matrix[j, i] = stat_matrix[i, j]

    return stat_matrix
```

Figure 35: pairwise_comparison_7



```
key_ordered_ap = [j.split('.')[0]+'     ('+str(i+1)+')' for i,j in enumerate(npy_paths)]
key_ordered_no = [str(no) for no in range(1, 10)]

fig, ax = plt.subplots(figsize=(20, 15), ncols=4, nrows=2, sharey=True)

for c, col_name in enumerate(new_names_ordered[:-1]):

    c_ax = ax[c//4, c%4]

    stat_matrix = run_statistical_tests(total_df, col_name)
    stat_matrix = np.clip(stat_matrix, a_min=0.001, a_max=10)
    np.fill_diagonal(stat_matrix, 0)

    im, cbar = heatmap(stat_matrix, key_ordered_ap, col_labels=key_ordered_no, col_labels_pos='b', ax=c_ax, cmap_sel='binary')

    plt.setp(c_ax.get_xticklabels(), rotation=0, ha="center", rotation_mode="anchor")
    c_ax.tick_params(axis = 'both', which = 'major', labelsize = 16)
    c_ax.set_title(r'{0}'.format(col_name), fontsize=16)

    cbar.remove()

ax[-1,-1].set_axis_off()

plt.tight_layout()
plt.subplots_adjust(hspace=0.25)
plt.tight_layout(pad=1.5, w_pad=0.1, h_pad=1.0)
plt.savefig('../pairwise comparison/supmaterial_unitests.png')
```

Figure 36: pairwise_comparison_8