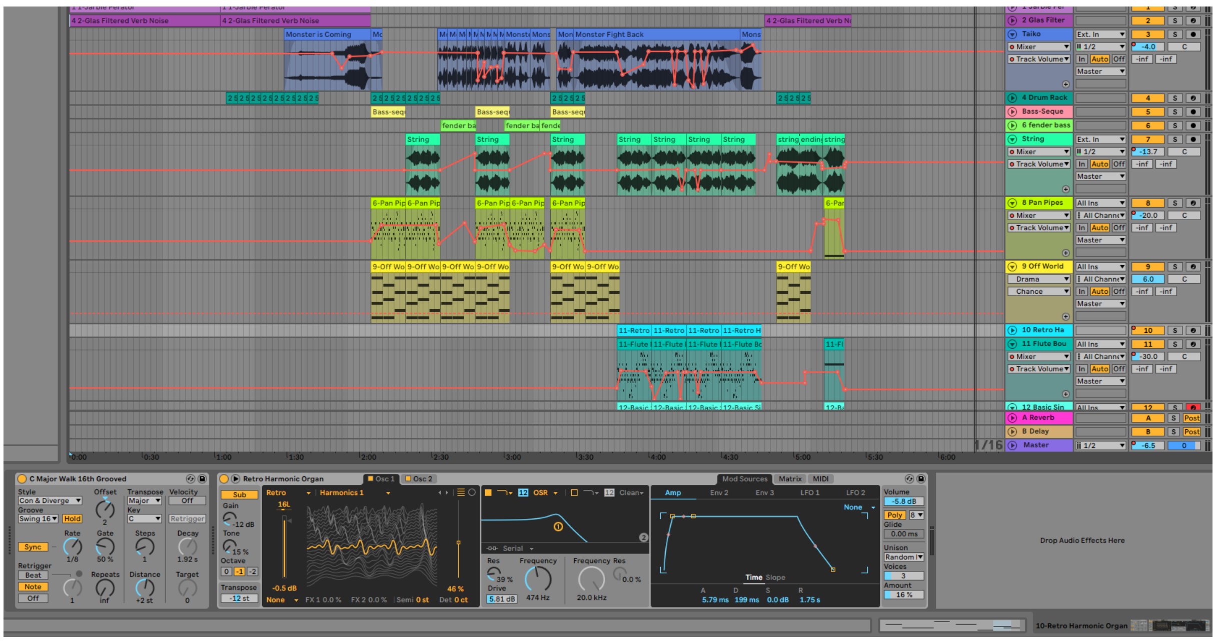
A Tool for Classifying Music Genres



Name: Yichen Lian

ID: 101066517

Course Code: COMP 4905 Honours Project

School: Carleton University

Supervisor: Anil Somayaji

Abstract

Nowadays, with the development of the Internet and information transmission technology, people are gradually stepping into an era of information explosion. Internet users are likely to find thousands of new posts, including the latest movies, videos, articles, and music, every minute on Tik Tok, Spotify and Instagram. Thus, the recommendation system intervenes when people have to face so much information to help them choose the content they prefer. Personalized recommendations originated in the 1990s. Based on machine learning algorithms, recommendation systems automatically recommend or provide similar content to users according to the content information characteristics of projects and users' personal interests and preferences. The most important part of the whole system is the information feature of the content. This is also known as music information retrieval (MIR). Due to the huge amount of data, it is difficult for audit staff to classify each piece of information separately through manual review. Instead, they must rely on computer technology to label the information and realize fit recommendations through more detailed and correct classification.

Based on the above background, this project will classify a given set of music and judge whether they belong to the same genre. The project involves audio conversion, eigenvalue extraction and clustering. Finally, the project aims to optimize the results of the recommendation system by better classifying a given song and assigning appropriate labels.

Acknowledgement

First and foremost, I would like to acknowledge my supervisor as well as mentor, Anil Somayaji who suggested an interesting and correct direction towards my honours project. Before settling down the project direction, Professor Anil Somayaji once asked me what I was interested in most besides computers, and I answered it is music and artificial intelligence. This is also one of the reasons why he suggested me choose this topic as my honours project. I must appreciate his effects in finding a project that would suit my interests and past work experience. I would also like to thank his meetings with me that kept me well organized and less stressed about the project.

Table of Content

1. Introduction

This report is a representation of my honours project at Carleton University. The report contains six parts with the first four parts discussing the implementation details of this project, including the code and algorithm explanation, and the last two parts aim to discuss the potential usage and the improvement in the future.

There are several kinds of different algorithms involved in the project. To help readers refer to the specific terms better, I used italics and different colors to identify the algorithms in this report. I also attached an appendix at the end of this report to help readers to cross-reference conveniently.

1.1 Overview

This project is a Python version of the music genre identifier. It takes one song as the training sample and uses this sample as the standard to cluster and classify other songs that have similar characteristics to this sample. The project then calculates the differences in the current song and the standard sample (as given before) and uses this value to determine the likelihood of whether these two songs belong to the same genre. The project is an implementation of re-designing and optimizing the existing clustering method. It uses *K-Means++* as the basic clustering method but improves the traditional clustering process to better fit the classification process of musical works. The algorithm also includes both supervised and unsupervised learning processes to shrink the randomization in the traditional *K-Means++* process (details in the *2. Algorithm and Code Implementation section*).

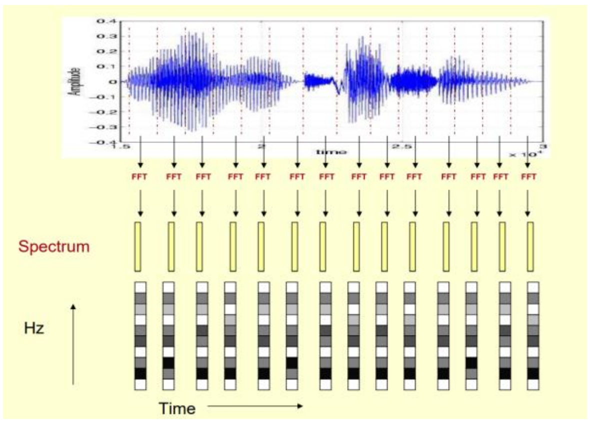
1.2 Background

This chapter includes two parts. I will introduce the key algorithms involved in this project in the previous four sections. And I will introduce the audio classification and recognition method that has been commonly used in industrial practice in the last section. It is important to note that the following part will only stay at the level of the algorithms and the corresponding code implementations but will not have the re-development of the theory of algorithm implementation and dependencies from a mathematical perspective in this report.

1.2.1 Mel-scale Frequency Cepstral Coefficients

The first step in an automatic speech recognition system is the extraction of feature values. This is because the general audio contains not only the speech signal but also useless information such as background noise and the speaker's emotion, which should be eliminated during speech recognition and analysis (Boxler, 2020, p. 30). Sound is emitted through object vibrations. In terms of humans, we make different sounds by changing the shape of the vocal tract. The shape of the vocal tract is shown in the envelope of the short-time power spectrum of speech. *Mel-scale Frequency Cepstral Coefficients* (*MFCC*) is a feature used to describe this envelope, which is widely used in automatic speech and speaker recognition. It is an algorithm based on the way the human ear recognizes the sound and the laws of auditory perception and simulates the characteristics of the human ear's processing of sound (Prahallad, n.d.). The hearing mechanism of the human ear has different auditory sensitivities to different frequencies of sound waves. In such a mechanism, the sound volume is not linearly related to the sound frequency but is approximately linearly proportional to the logarithm of that sound frequency. However, in *MFCC*, after calculation and conversion, the human perception of pitch is linear in the Mel frequency domain. *MFCC* does not depend on the nature of the signal, does not make any assumptions or restrictions on the input signal, and makes use of the results of auditory modelling. Therefore, it is more consistent with the auditory properties of the human ear and will remain a better recognition performance at low signal-to-noise ratios.

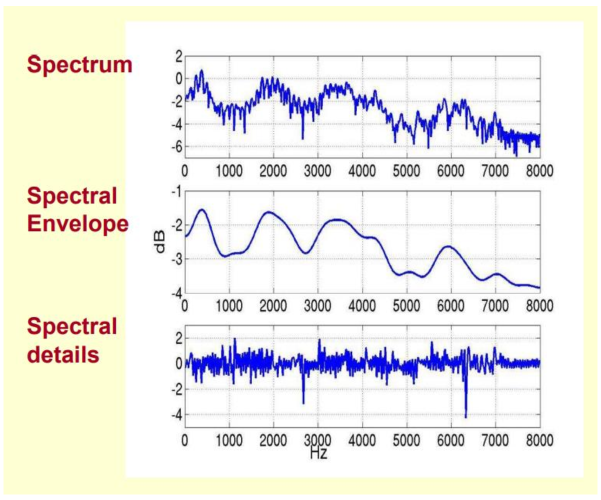
To understand *MFCC*, it is first necessary to understand how to describe a sound signal. Nowadays, much common music production software has functions like using waveform graphs to represent the frequency distribution direction of a song. But a simple waveform graph can only reflect the characteristics of one frame (not a segment) of speech. That is, it is a static feature at each point corresponding to its position (Prahallad, n.d.). Therefore, in order to represent the pattern of energy changes in sounds through time, speech is usually represented using a sound spectrum diagram (spectrograms). A spectrogram is derived from a waveform diagram. First, the speech is divided into many frames, and for each frame, a spectrum representing the frequency energy is calculated using fast Fourier transform (FFT). This step is used to improve the components with lower amplitude and to facilitate the observation of periodic signals of low-amplitude sounds. The figure below illustrates the process of the conversion, where the darker color represents the higher amplitude.



Figure[1] 1.1: Convert waveforms into time- and spectral-based sound spectrograms

[1] This image comes from Prahallad's lecture slides

The darkest part of each frame represents a formant, which is a peak in the spectrogram. Formants can be used to represent the main frequency components of the speech, and therefore, they can be considered as data that carries sound recognition properties. The envelope is a smooth curve connecting these peaks, which can be used to describe and identify the positions and transitions of the formants. Therefore, the original spectrum can also be considered as a combination of the envelope and the corresponding spectral details. In this case, the spectral details represent the characteristics of each point, and the original spectrum can be restored by multiplying the corresponding slope of the envelope on this basis.



Figure[2] 1.2: Spectrum, envelope, and spectral details

Human ears are only sensitive to specific frequencies. According to the shielding effect, when two sounds of different volumes act on the human ears, the presence of the louder frequency component affects the perception of the less loud frequency component. Only when the two frequency components differ by a certain bandwidth can they be distinguished, otherwise the person will hear the two different sounds like one. In general, low-frequency sounds tend to mask high-frequency sounds, and for sound signals, most of the signal is included in the low frequency and low amplitude components (Xu, 2018). Therefore, after obtaining the envelope it is also necessary to intercept the envelope according to the auditory frequency of the human ears. There is a set of band-pass filters from low to high frequencies in the order of bandwidth, from dense to sparse, to filter the input signal. These filters mimic the auditory perception of the human ears and will only allow sounds at specific frequencies to pass through, ignoring other frequencies. The filters are distributed more densely in the low-frequency region and more sparsely in the high-frequency region. In the *MFCC* algorithm, the underlying linear spectrum needs to be transformed into a Mel-scale non-linear spectrum by Mel filter first, and then the corresponding *MFCC* is obtained as the input feature of speech by taking logarithmic and inverse transform (usually it is

[2] This image comes from Prahallad's lecture slides

calculated by discrete cosine transform (DCT)) (Fayek, 2016). Among them, the cestrum parameters corresponding to the frequencies are as follows[3]:

$$Mel(f) = 2595 * lg(1 + \frac{f}{700})$$

The details of how to calculate the corresponding *MFCC* of a sound signal will be presented in *2.1.1 Mel-scale Frequency Cepstral Coefficients*.

1.2.2 Principal Component Analysis

Principal Component Analysis, also abbreviated as *PCA*, is a common way of data analysis, often used for downscaling high-dimensional data and simplifying data sets so as to extract the main feature components of the data. *PCA* is an unsupervised linear dimensionality reduction method (Lee, 2015). When it comes to multi-factor analysis, data often carry hundreds of different dimensions at the same time. A very intuitive example is the database. In a typical relational database, each row simultaneously carries multiple columns of directional data corresponding to it, with each column representing a different attribute. There is a lot of content that is not used or is redundant when analyzing the data in the database. The increase in dimensionality leads to a simultaneous increase in spatial volume, resulting in a sparse spatial distribution of data and a weakened relationship between data points (see *3.2 Processing of High-Dimension Data*).

Principal Component Analysis is one of the solutions to these dimensionality reduction methods. In addition to linear dimensionality reduction that increases the sample density by rounding off some information, *PCA* has the following two features in this project. First, *PCA* reduces the noise of audio files, which is especially suitable for live or recorded audio of songs with poor sound quality. This is because when the data is affected by noise, the feature vector corresponding to the minimum eigenvalue is often related to the noise. Noise does not usually appear as a major part of such song audio files. Therefore, the minimum eigenvalues discarded in *PCA* can have a noise reduction effect to a certain extent. However, the problem of important information being incorrectly discarded may also occur in the process of dimensionality reduction. This is because *PCA* discards information based on the eigenvalues, but it may cause some information that “seems” useless but is actually important features to be discarded incorrectly. For this reason, *PCA* should be used to reduce the dimensionality of the data to ensure that it retains a high level of information about the original data and should not discard the details of the information for the sake of low dimensionality (Lee, 2015). But in turn, the relevant features and other important features are more likely to be represented in the data after dimensionality reduction. Please recall the content from the last section. In the project, *MFCC* will retain a matrix with each frame is represented

[3] This formula comes from Fayek, 2016

by 39-dimension values. The computation of clustering using high-dimensional data is meaningless, so the data should first be reduced in dimensionality before clustering. In this project, the code related to *PCA* is located in the file *genre.py*.

PCA finds a set of mutually orthogonal axes from the space composed of the original data (Lee, 2015). The first new axis is chosen in the direction of the largest variance in the original data, and the second axis is chosen as the diagonal line with the largest variance in the plane orthogonal to the first axis. Starting from the third axis, a slope with the largest variance in the plane orthogonal to the first two axes is selected and so on up to all n axes. With the gradual acquisition of new axes in this way, the first K axes contain most of the variance, and the next n-K axes contain almost zero variance. In this case, the dimensionality reduction of the data features can be achieved by retaining only the first K dimensional features that contain the vast majority of the variance and ignoring the feature dimensions that contain almost zero variance.

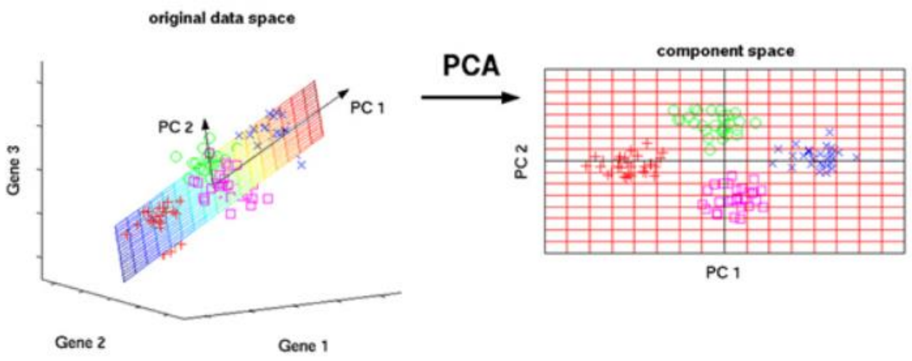


Figure 1.3[4]: Diagrammatic representation of the *PCA* principle.

The essence of *PCA* is diagonalization of the covariance matrix, which can be used to obtain the eigenvectors of the covariance matrix by calculating the covariance matrix of the data matrix and selecting the matrix of eigenvectors corresponding to the k features with the largest eigenvalues (variances) in descending order to transform the data matrix into a new space (Lee, 2015).

1.2.3 K-Means and K-Means++

Clustering analysis, also known as group analysis, is a statistical analysis method for studying sample classification problems and can also be used as a pre-processing step before analytical algorithms such as data mining (Qing, 2016). The execution of clustering algorithms often includes many patterns, which are a vector of measurements, or a point in a multi-dimensional space. The implementation of clustering algorithms can be based on similarity, partitioning a data set into different classes or clusters according to some specific criteria (e.g., Manhattan distance or Euclidean distance),

[4] This image comes from Lee, 2015

making the similarity of data objects within the same cluster as large as possible, while the difference (distance) of data objects not in the same cluster is as large as possible. This means that the data in the same class are clustered together as much as possible after clustering, and the different data are separated as much as possible. Patterns that are in the same cluster have more similarity to each other than patterns that are not in the same cluster. The clustering algorithm is also a type of unsupervised learning because it does not require information about the classification or grouping of data categories, and the algorithm itself does not need to use training data for learning. In clustering, the purpose of the procedure is not to identify, but to cluster similar data together (Wagstaff, Cardie & Schrödl, 2001, pp. 1). Therefore, the most important aspect of a clustering method is how to calculate the similarity of the data, and the algorithms can be broadly classified into hierarchical methods, partition-based methods, density-based methods, grid-based methods, and model-based methods according to the calculation method. *K-Means* is a partition-based method, and the *K-Means++* used in this project is an improved algorithm of *K-Means*, but the logic and principles of the two algorithms are almost the same. I will start with *K-Means* and then introduce the specific *K-Means++* improvements.

The basic principle of partition-based clustering is that given a dataset D including n data, K groups ($k \leq n$) are constructed within the dataset using a partitioning approach. At the same time, the data in each group must maintain the properties of clustering, that is, the distance between any two points of the same group is less than the distance between any two points of different groups. Based on this, the idea of *K-Means* is to divide the sample set into K clusters according to the size of the distance between samples for a given sample set, and for clusters C_1, C_2, C_3, …, C_k, the goal of *K-Means* is to satisfy the minimization of the squared error E[5]:

$$E = \sum_{i=1}^{K} \sum_{x \in C_i} (x - P_i)^2$$

Where P_i represents the centroid (mean vector) of each cluster C_i[6]:

$$P_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

Thus, when determining the total number of clusters as K, in the worst case, the enumeration of the cluster to the power of N is required, which poses an NP-hard problem to obtain the minimum squared error E directly through calculation. This is the reason why the implementation of *K-Means* requires the use of heuristic iterative methods.

[5] This formula comes from Qing, 2016

[6] This formula comes from Qing, 2016

1.2.4 Vector Quantization

There are two types of quantization: scalar quantization, which quantizes the sampled signal values one by one, and *Vector Quantization* (*VQ*), which quantizes more than two sampled values to form a vector in K-dimensional space. *VQ* is a lossy data compression method based on block coding rules. Its basic idea is that the K-dimensional input vector is given an overall quantization in vector space and mapped into another K-dimensional quantization vector, thus compressing the data with as little loss of information as possible (Makhoul, Roucos, and Gish, 1985). The set of quantization vectors is called a codebook, where each quantization vector is a code vector. The process of compression can be considered as an approximation. Similar to the process of approximation, taking a floating-point number as an example, the process of *VQ* is to approximate this number by an integer that is closest to this number. *Vector Quantization* plays a very important role in speech signal processing and is widely used in speech coding, speech recognition and speech synthesis. In addition, the step of calculating *VQ* is included in multimedia compression formats such as JPEG and MPEG-4. This is because, for multimedia, especially audio and video, the representation of these media information is usually an analog signal consisting of continuous values in an interval (Pluskid, 2009). However, since computers can only process discrete data, when processing multimedia files it is necessary to first convert the analog signal into a digital signal by replacing a whole interval of continuous values with an integer value at both ends of the interval. The distortion value represents the closest distance from a certain number to the corresponding codebook. Among other things, this approximation should be chosen to satisfy the minimum distortion value caused by quantization at a determined codebook size of K (recall that *VQ* is a data compression process so that the smaller the distortion the better).

1.2.5 Industrial Practices

The form of music storage has long since shifted from physical storage to information in computers because of the rapid technological advances. Large music platforms now offer catalogs containing millions of songs, which are often accessed by text tags (such as genre, artist, or title keywords). And this large amount of data is obviously difficult to maintain by a human. Therefore, dealing with such large amounts of data involves an application area in computing: music information retrieval (MIR) (Boxler, 2020, p. 2). As a basic task of MIR, it is used to manage and distinguish the content of music files. In general, this step involves the extraction and transformation of feature values of the music and speech signals contained in the audio file. Common features used to characterize audio content include time-domain features based on short-time energy (STE) and zero-over rate (ZCR); frequency-domain features including primary frequency, *MFCC*, and signal energy; and visual features based on spectrograms and waveform plots (Xu, 2018, p. 3). In fact, there are many algorithms and methods for classifying and identifying songs or audio files based on artificial intelligence, deep learning, and neural network algorithms. In this section, I will give a brief introduction to some of these common algorithms and discuss how these algorithms are applied in practical development.

Due to the diversity and continuity of music, identifying and classifying music cannot be done simply by a series of conditional if-else statements but with a more complicated method in the artificial intelligence field. These algorithms include Traditional Neural Networks (TNN), Convolutional Neural Networks (CNN), Support Vector Machines (SVM), and Logistic Regression, which was used earlier as well as the more popular algorithms include Decision Tree, K-Nearest Neighbor (KNN), and Multi-layer Perceptron (MLP). The classification accuracies corresponding to these algorithms are shown in the following chart:

ALGORITHM NAME	ACCURACY
TNN	51.4%
CNN	40.5%
SVM	72%
LOGISTIC REGRESSION	88.9%
DECISION TREE	88.9%
KNN	92.2%
MLP	83.%

Chart 1.1[7]: Common audio classification algorithms and the corresponding classification accuracy

Instead of presenting all the algorithms mentioned above, I will briefly outline how some of them are practiced. In a convolutional neural network-based project, the feature matrix obtained based on *MFCC* extraction is used as input to a CNN neural network, which is trained to obtain an optimized classifier for the audio signal as input. The network itself consists of several convolutional layers, and the output is aggregated several times and sent to two fully connected layers. Here, the audio file is truncated into short and long frames, which are sequentially processed through the spectrum to obtain n different feature subgraphs. Each graph represents the information corresponding to the frequency and energy of a different time. In the last layer, all obtained partitions and corresponding labels are merged. Since different regions correspond to different labels, the probabilities of all samples are summed to perform the mean-pooling to obtain the probabilities and predictions for each classification of the current music in the last step. The feedback data from the neural network needs to be returned to the training and further optimized by a backpropagation algorithm (Boxler, 2020, p. 35). When it comes to KNN, the algorithm is similar to the idea of *K-Means*. KNN is a supervised learning algorithm that discriminates the properties of new data based on the K known data closest to the data to be predicted. This means that when new data (the data to be predicted) comes, the algorithm compares this data with the existing data. The new data is assigned to the corresponding classification based on

[7] This experiment data comes from KevinLu10's code posted on GitHub

the characteristics of the new data and the similarity of the rest (Xu, 2018, p. 10).

As can be seen, most of these algorithms are more complex to implement, and it is a tedious process to understand the principles of the whole algorithm from scratch. For example, one needs to understand deep learning first before understanding the usage of KNN to facilitate the subsequent tuning of the model; one also needs to understand the concepts related to neural networks before learning MLP to further understand the meaning of the corresponding layers' functionalities. Second, these algorithms need numbers of labelled data sets for training. Machine learning is inherently an iterative optimization process, and without the amount of data to support the model it is difficult to obtain satisfactory prediction results in practice. Therefore, for small projects, these prerequisites amount to increased difficulty in implementation.

1.3 Project Structure

The project contains three Python files: *genre.py, clustering.py, and extract_features.py*. The implementation details of each file will be explained in the later section of this report (details in the *2. Algorithm and Code Implementation* section). There is also a folder called dataset in the project directory that is used for training and testing. The folder contains different sub-directories that are divided according to the purpose. Specifically, *training* contains a single blues song that will be used for generating the *Vector Quantization* (*VQ*) label. Other folders including SingleGenre, MixedGenre, CyclicBlues, and Subgenre corresponds to the four test cases with the relevant test sets. For details of the test, please refer to *4.1 Subsequent tests and results*. Nevertheless, to shrink the total size of the project due to the limitation, some of the test sets has been removed from the project directory.

The project is coded with Python 3.6.1. In addition to the Python environment, the project also needs following the libraries: *NumPy*, *pyparsing*, *SciPy*, *librosa*, and *scikit-learn*. A *requirement.txt* file that lists all the libraries used in the program with the version is given in the project directory as well. The folder tree is as follow:

```
├─.idea
│  └─inspectionProfiles
├─dataset
│  ├─CyclicBlues
│  ├─MixedGenre
│  ├─SingleGenre
│  │  └─blues
│  │      ├─CloudMusic
│  │      └─GTZAN
│  ├─Subgenre
│  │  ├─rock
│  │  └─sub-genre
│  └─training
└─__pycache__
```

It is worth mentioning that the genre of the songs in the dataset is defined by the authoritative musical collection library GTZAN. I selected most of the songs randomly from this library to form the dataset.

1.4 Execution

The program can be executed with three different modes: run with a model, run without a model, and train the model. To run the program, simply change the terminal to the directory where the program exists and execute the command:

Python genre.py -flag_option [training_model] [target_folder]

Where ***training_model*** indicates the model or a song that will be used for training the model and ***target_folder*** means the folder of music that the user wants to classify. For example:

Python genre.py --tr dataset\\training\\blues.YesMan.wav dataset\\blue

The command will use the audio file blues.YesMan.wav for training as implement the trained model to the test set locates at the folder dataset\blue. There are three flags in total that correspond to the execution method mentioned above:

-r and **--run**

Run the program with an existing model. Users can apply the existing or pre-trained model as the classification standard to identify the genre. The program will then compare the characteristics of each song in the ***target_folder*** parameter with this standard. This mode is recommended because it adapts to most situations.

-tr

This flag means to train and run. This mode will be time-consuming as the program will extract features and train the sample song before classifying. Thus, users are encouraged to avoid redundant training if there is already a qualified model exists.

-t and **--train**

This mode will train the model without executing the classification. When the training is complete, the data will be stored in a *.npy* file (generated by *NumPy* to store the associated ndarray data). The filename of this model will consist of the song itself. Under this situation, the last parameter (***target_folder***) can be omitted.

1.5 Target Readers

This project is an implementation of the acoustic and artificial intelligence-based algorithm. In order to better understand the whole clustering process and the algorithm details, readers who have a certain algorithm and linear algebra foundation are a prefer. It will be favorable if readers have the basic knowledge of the supervised and unsupervised training techniques and processes in machine learning. Since the whole project is built with Python, readers are also expected to have a basic knowledge of Python and relevant libraries.

2. Algorithm and Code Implementation

In this section, I will explain in detail the implementation of the code for this project. I will first introduce the logic of the whole project, briefly describing how a song goes through a series of operations during the program's runtime and is eventually used to compare it to the classification criteria. I will start with the algorithm and combine it with code to describe how I implemented the functionality of the algorithm. Finally, I will explain some of the functions in the main function to help the readers understand the logic and implementation of the entire program.

The program can be divided into four main parts: eigenvalue extraction, data processing, model training, and clustering evaluation. The following diagram shows the execution process of this program.

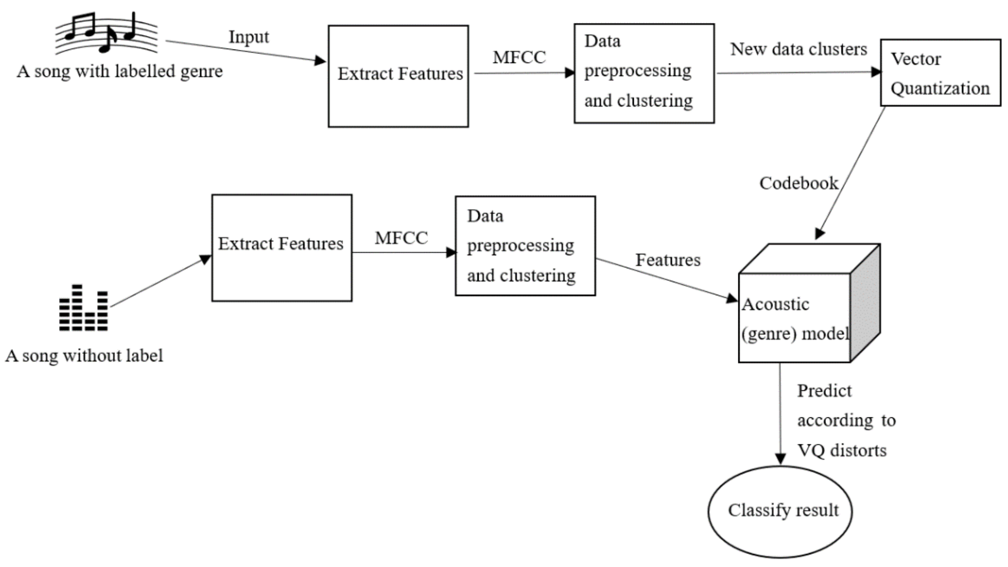


Figure 2.1: Flow diagram of the project.

First and foremost, the project needs a labelled song as a criterion for classification, which is what makes this project different from other projects based on deep learning and neural networks to predict genre. Common artificial intelligence projects require numbers of labelled datasets to train and optimize the model. The data from the test set is then fed into the generated model for predicting the results. Therefore, a common deep learning project requires a long period of optimization and a large amount of data accumulation to get the optimal training results and higher recognition accuracy. This project is based on the idea of vector distance and regression equation, which determines whether two songs belong to the same genre by determining whether the difference between genre and auditory feature values of the two songs is within a specific range.

As can be seen in the figure, the program first transforms the dimensionality and extracts the eigenvalues of all input songs (for both training and test sets), which corresponds to the eigenvalue extraction in the process part. The songs are transformed from acoustically expressed sounds to numerically expressed feature values, also known as *Mel-scale Frequency Cepstral Coefficients* (*MFCC*). *MFCC* is widely used as a feature value in speech recognition and speaker recognition represents the acoustic feeling of an audio clip. *MFCC* converts audio into a number of 39-dimensional audio feature points in the form of frames, specifically, each point represents a window of approximately 30 milliseconds of sound clips on the original audio. However, due to the high dimensionality of these frames, it is hard to obtain an ideal clustering result in the ensuing *K-Means* clustering process (see *3.2 Processing of High-Dimension Data*). Therefore, before performing clustering, these feature points first need to be downscaled. The process of dimensionality reduction is similar to the projection of a three-dimensional object viewed from a two-dimensional plane in the real life, and the same principle applies to high-dimensional data points. This project uses *Principal Component Analysis* (*PCA*) to reduce the dimensionality of high-dimensional frame data and extract the principal feature components of the data. As a reference, the dimensionality of the target is not determined randomly but is a value chosen after several tests. In this project, the 39-dimensional data will be reduced to 8 dimensions for subsequent computation and classification. At this point, the second step of data processing is officially completed.

The last two parts, model training and clustering evaluation both involve the use of *VQ*, but their difference lies in the different benchmarks for calculating vector distance. In terms of model training, the program first uses *K-Means++* to group the data that have undergone dimensionality reduction in the previous step into 160 groups. At the same time, the cluster centers (centroids) of these 160 clusters are used as the return value. *VQ* takes the centroids as the benchmark and computes the shortest distance to this benchmark for each frame in *MFCC*. This is because *VQ* performs data compression by forming a vector with scalar data sets and then giving an overall quantization in vector space. The path that satisfies the *MFCC* where every point to the cluster center is the shortest represents the best match between the current set of vectors and the codebook used for template matching because its sum of the vector distance is the smallest. The amount of distortion obtained from this *VQ* calculation is saved and used as a criterion for the acoustic model classification for the next step, genre classification.

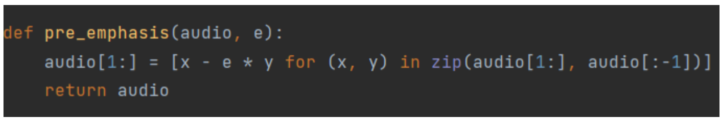
During the *VQ* calculation of the analysis step, the codebook should not be the clustering result of the test song, but still based on the model's centroids (the model data just saved), and the shortest distance from the *MFCC* of the target audio to the model's centroids is calculated based on the model's centroids. Similar to the above principle, the return value of *VQ* represents the shortest distance from the *MFCC* of the target audio to the model. The set of this distance difference indirectly reflects the distance between two spatial vectors, i.e., how well they fit each other. This means that when the distance difference between two spatial vectors or matrices is zero, they overlap completely because at this point there is no difference between them. And the larger the sum of this distance difference the larger the difference between the two vectors. The last step of this project is to determine the current target song attribution based on the threshold value. A larger sum of the distance difference indicates that the two songs are more unlikely to belong to the same category. The selection of this threshold is not random but is determined by the sum of distortions generated during the training. When the sum of the distance differences obtained during the cluster analysis is less than the threshold value, the two songs can be considered as belonging to the same genre, and vice versa.

2.1 Algorithm Implementation

2.1.1 Mel-scale Frequency Cepstral Coefficients

The meaning of *MFCC* can be seen as convert a normal spectrum into the Mel-scale coefficients by filtering the data to the Mel filter banks. According to the background section mentioned previously, the steps to compute *MFCC* can be divided into the following steps: pre-emphasis; frame blocking; windowing; computation of Mel frequency filter set; computation of *MFCC*; and average optimization. The specific steps and the corresponding code implementation for each part will be described below.

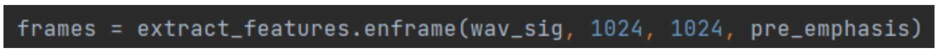
First, pre-emphasis is required for the input audio to amplify the high frequencies and balance the entire spectrum. The sound emitted will suppress some of the high frequencies because of the construction of the vocal folds and lips. Pre-emphasis is equivalent to adding a high-pass filter to boost the high-frequency part so that the spectrum of the signal becomes flat and stays in the whole band from low to high frequencies. At the same time, this also highlights the resonant peaks in the high frequencies. According to the formula, the pre-emphasis is achieved by subtracting the corresponding timing, i.e., $x(t) - \alpha x(t - 1)$ to obtain the updated value. Here, α (*e* in the code) represents the filter coefficient, which generally takes values between 0.9 and 1, and is usually taken as 0.95 or 0.97 (Fayek, 2016).



```
def pre_emphasis(audio, e):
    audio[1:] = [x - e * y for (x, y) in zip(audio[1:], audio[:-1])]
    return audio
```

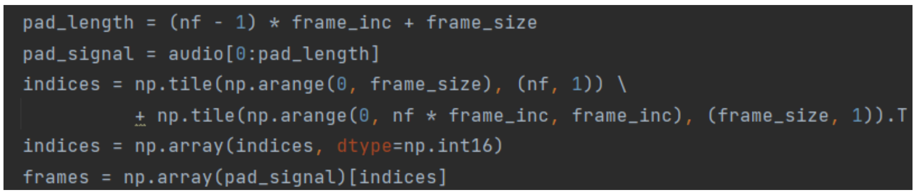
Figure 2.2: Pre-emphasis for the audio before calculating *MFCC*.

Frame blocking. Since the frequencies in a signal change over time and the Fourier transform is only suitable for analyzing smooth signals. Framing is based on the idea of differentiation, i.e., the assumption that the frequency of the signal is constant over a quite short period of time. The input audio is decomposed into several short audio sessions of about 30 milliseconds depending on the settings (Fayek, 2016). To increase the continuity and reliability of the data, overlapping frames and analyzing adjacent frames in series can have a better result. In the code, the first three parameters are the audio stream, the length of a frame, and the length of each frame increasement. In this project, I did not set the overlap between frames, so the length of each frame and the distance of the frame shift are equal in the function call (1024 samples for each). The total number of samples in the function is not set arbitrarily. Usually, the length of a frame is around 30 milliseconds, and the number of sampling points should be guaranteed to be an exponent of two. The time (frame length) is equal to the sampling frequency (22.05 kHz set at the time of reading, see *2.2 Other Details*) divided by the number of samples. After calculations I determined the total number of samples per frame should be 1024.



```
frames = extract_features.enframe(wav_sig, 1024, 1024, pre_emphasis)
```

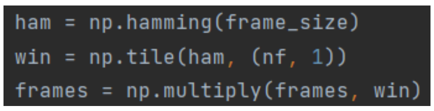
Figure 2.3: Framing function of speech signal.



```
pad_length = (nf - 1) * frame_inc + frame_size
pad_signal = audio[0:pad_length]
indices = np.tile(np.arange(0, frame_size), (nf, 1)) \
          + np.tile(np.arange(0, nf * frame_inc, frame_inc), (frame_size, 1)).T
indices = np.array(indices, dtype=np.int16)
frames = np.array(pad_signal)[indices]
```

Figure 2.4: The framing operation is implemented through the *tile()* and *array()* functions in NumPy. The process is like cutting a long string on demand and storing it in its original order using *np.array()* with the corresponding index. Two variables *pad_length* and *pad_signal* ensure that the length as well as the total number of sample points of all frames is the same.

It is also necessary to perform the windowing to each frame after framing. This is because it is difficult to ensure signal continuity at both ends of the frame after splitting and the windowing is done to eliminate as much as possible the factors that can cause signal discontinuity at both ends of each frame (Fayek, 2016). There are many ways to do windowing while the one used in this project is the Hamming window. The Hamming window is a weighted cosine function that smoothed or tapers the discontinuities at the beginning and end of the sampled signal to achieve this purpose. At this point, the original speech signal has been processed and the corresponding spectrum for each frame can be obtained using the Fourier transform. According to the formula, I calculated the spectrum and energy in *spectrum()* using the Fourier transform functions *fft()* and *rfft()* in *NumPy*. The spectrum obtained here will be fed into the Mel filter banks to calculate the *MFCC*.



```
ham = np.hamming(frame_size)
win = np.tile(ham, (nf, 1))
frames = np.multiply(frames, win)
```

Figure 2.5: The program generates the Hamming window by calling the function *np.hamming()* and merges all the frames after adding the window with the functions

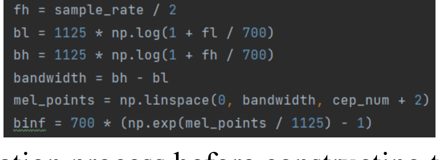
tile() and *multiply()*. The variable *frames* is an array representing the set of all framing result, which will be used as a return value and will be used for the next calculation operations.

Construct the Mel filter banks. Recall that there is a non-linear relationship between human ears and heard frequencies, but a logarithmic relationship (see *1.2.1 Mel-scale Frequency Cepstral Coefficients*). Thus, the corresponding voice needs to be measured in a Mel-scale by using a Mel filter banks (triangular bandpass filters). This set of filters smoothen the spectrum to highlight the formant peaks of the original speech (see *1.2.1 Mel-scale Frequency Cepstral Coefficients*). The formula for converting the frequency to a Mel- scale is as follows (this is equal to the equation mentioned earlier in *1.2.1 Mel-scale Frequency Cepstral Coefficients* with proper mathematical transformations):

$$Mel(f) = 1125 * ln(1 + \frac{Hz(f)}{700})$$

Based on the formula, I calculated the Mel frequency corresponding to the lowest frequency (0 in this project) and the cutoff frequency (half of the sample rate), respectively. This range is equally divided according to the bandwidth under the Mel-scale, and this Mel-scale values are inverted back to the Hertzian frequency (corresponding to the variable *mel_points* and *binf* in the code, respectively). Based on this Hertzian frequency and the following equation[8], the triangular bandpass filters, i.e., Mel filter banks, can be constructed.

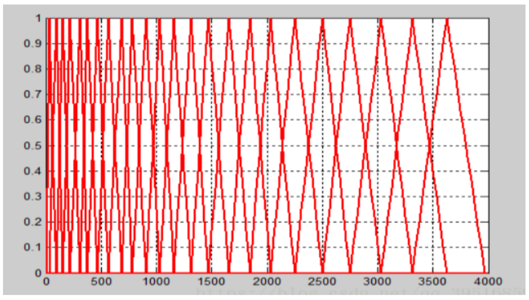
$$\begin{cases} 0 & k < f(m-1) \\ \dfrac{k - f(m-1)}{f(m) - f(m-1)} & f(m-1) \le k < f(m) \\ 1 & k = f(m) \\ \dfrac{f(m+1) - k}{f(m+1) - f(m)} & f(m) < k \le f(m+1) \\ 0 & k > f(m+1) \end{cases}$$



```
fl = 0
fh = sample_rate / 2
bl = 1125 * np.log(1 + fl / 700)
bh = 1125 * np.log(1 + fh / 700)
bandwidth = bh - bl
mel_points = np.linspace(0, bandwidth, cep_num + 2)
binf = 700 * (np.exp(mel_points / 1125) - 1)
```

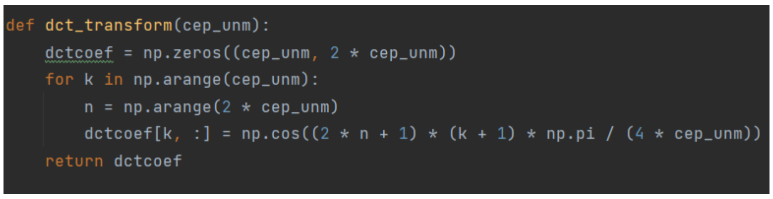
Figure 2.6: Initialization process before constructing the Mel filter banks.

[8] This formula comes from Fayek, 2016



Figure[9] 2.7: How typical Mel filter banks look like. Because the frequency is not linear distributed in normal Hertzian scale, the filters are concentrated on the low-frequency part but loosen in the high-frequency.

Computation of *MFCC*. The cepstrum coefficients obtained by discrete cosine transform for each frame are the original *MFCC* values. Generally, the initial *MFCC* will discard the first value and keep it from the second one. Here I have omitted this step to simplify the process. The function *dct_transform()* will first calculate the DCT coefficients at the corresponding positions, and this coefficient will be multiplied by the matrix obtained through the Mel filter to obtain the inverse spectral coefficients based on the Mel frequencies. Note that, in this step, it multiplies the n * 26-dimensional Mel data with the 13 * 26-dimensional DCT coefficients, and the final matrix obtained is 13 * n-dimensional. I performed an additional inverse spectral lifting in the program. The function *lift_window()* will improve the high-frequency part of the DCT cepstrum obtained in the previous step. Since the high-frequency energy is lower compared to the low-frequency. This is equivalent to raising the energy of the high-frequency part of the spectrum so that the high-frequency part is also valued. The *MFCC* of the current audio signal is obtained after the cepstrum is merged.



```
def dct_transform(cep_unm):
    dctcoef = np.zeros((cep_unm, 2 * cep_unm))
    for k in np.arange(cep_unm):
        n = np.arange(2 * cep_unm)
        dctcoef[k, :] = np.cos((2 * n + 1) * (k + 1) * np.pi / (4 * cep_unm))
    return dctcoef
```

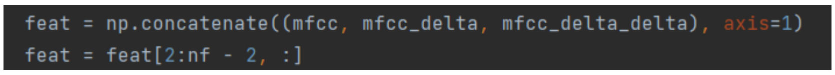
Figure2.8: Calculate the corresponding coefficients according to the DCT formula.

However, this *MFCC* is not the final target value in this project; the 39-dimensional *MFCC* is composed of 13 static coefficients, 13 first-differentiate coefficients and 13 second-differentiate coefficients. Among them, the original *MFCC* obtained by DCT calculation in the previous step is only 13-dimensional describing the envelope content of the speech signal, which is used for speech recognition. It also needs to take into account the frame-to-frame relationship. Therefore, the relationship between frames, energy, and time should also be merged using first order and second order differencing before outputting the results. This operation is also one of the reasons why the dimensionality of *MFCC* eigenvalues is typically 13-, 26-, or 39-dimensions. This

[9] The image comes from: https://blog.csdn.net/qq_39516859/article/details/80815369

coefficient is used to describe the value of the dynamic feature, i.e., the variation of the acoustic feature between two frames as:

$$c(t)' = \frac{c(t+1) - c(t-1)}{2}$$



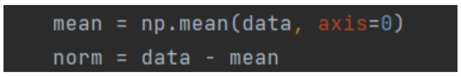
```
feat = np.concatenate((mfcc, mfcc_delta, mfcc_delta_delta), axis=1)
feat = feat[2:nf - 2, :]
```

Figure2.9: Combine the differentiate result.

2.1.2 Principal Component Analysis

In general, the process of *PCA* algorithm implemented based on eigenvalue decomposition covariance matrix includes the following four steps: normalization of the original data; calculation of covariance matrix, corresponding eigenvalues, and eigenvectors; principal component sorting; and calculation of the reduced dimensional data set.

Raw data normalization. In the *PCA* calculation, if one of the features (a column of the matrix) of the data has a particularly large value, it will have a large weight in the error calculation. After projecting such data into a low-dimensional space, the entire projection will try to approximate the largest feature and ignore the features with smaller values to make the low-dimensional data approximate the original data (Giraud, 2014). The importance of individual features is unknown until *PCA* is performed. Thus, this is likely to result in a large amount of missing information. Therefore, it is necessary to first normalize each feature and calculate the normalization so that the size of the data is in the same range. Here, I first used the mean function in NumPy and set axis=0 to calculate the mean of each column of the matrix. After that, I normalized the input matrix data by subtracting the matrix with the mean of each column from the original matrix according to the laws of matrix operations.



```
mean = np.mean(data, axis=0)
norm = data - mean
```

Figure 2.3: The implementation of matrix normalization

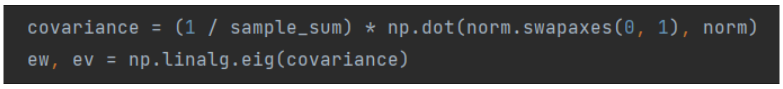
Calculating the covariance matrix, the corresponding eigenvalues, and eigenvectors. This step can be expressed in the formula[10] as:

$$C = \frac{1}{n} \cdot X \cdot X^T$$

Where n represents the number of samples and X represents the data set matrix. The

[10] This formula comes from Lee, 2015

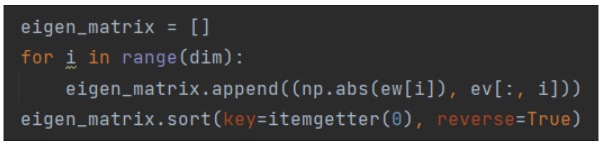
first thing that is used here is *swapaxes()*. This function swaps the rows and columns of the corresponding matrix, i.e., it obtains the transpose of the corresponding matrix. Another function that is used is *dot()* in *NumPy*, which is used to perform multiplication between matrices. After obtaining the covariance matrix, I directly called *NumPy*'s *linalg.eig()* to compute the eigenvalues and eigenvectors of the matrix.



```
covariance = (1 / sample_sum) * np.dot(norm.swapaxes(0, 1), norm)
ew, ev = np.linalg.eig(covariance)
```

Figure 2.4: Obtaining the eigenvalues and eigenvectors of the covariance matrix

Principal component sorting. Since the feature values and the feature vector obtained in the previous step are one-to-one, I do not want to spread them out, but first store the two values as tuples through a loop and put all the "feature" tuples in a new list for the last step to generate a new feature vector. I also need to sort the feature values from largest to smallest in this step. Nevertheless, using the *sort()* function alone does not work here, because the feature values and feature vectors are stored as tuples, and the sorting criterion should be the eigenvalues of the feature vectors (recall that according to the principle of *PCA*, the dimensionality reduction is retained for the principal components, i.e., the data with larger eigenvalues). Therefore, I call the *itergetter()* function to get the first element, the eigenvalue, of each tuple, as the element of comparison. Also, I set *reverse=True* to get the ranking from the largest to the smallest.

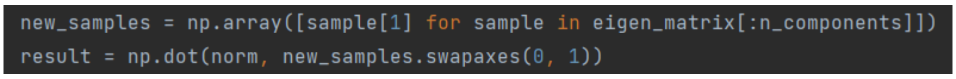


```
eigen_matrix = []
for i in range(dim):
    eigen_matrix.append((np.abs(ew[i]), ev[:, i]))
eigen_matrix.sort(key=itemgetter(0), reverse=True)
```

Figure 2.5: Component prioritization based on the magnitude of eigenvalues

Generation of dimensionality reduction results. Since the purpose of *PCA* is to downscale the data, only the top K values (the K elements with the largest eigenvalues) of the list after the previous sorting step will be kept. Where K represents the target dimension. Finally, the new data set N can be calculated by (E represents the matrix composed of the top K elements with the largest eigenvalues):

$$N = E^T \cdot X$$



```
new_samples = np.array([sample[1] for sample in eigen_matrix[:n_components]])
result = np.dot(norm, new_samples.swapaxes(0, 1))
```

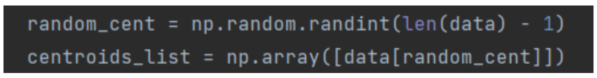
Figure 2.6: Calculating the downscaled-dimensional matrix

The variable *result* in this code represents the data after dimensionality reduction.

2.1.3 K-Means and K-Means++

This project uses clustering to classify the frames of *MFCC* for the next calculation of spatial vector distances. Rather than classifying directly using *VQ* without using cluster analysis, *MFCC* feature values will first be grouped by clustering into about 160 groups, and then *VQ* will be calculated based on the cluster centers of each cluster (group) as a codebook. The advantage of doing so is that each frame of data is first classified before the formal classification. Since the distribution of eigenvalues in *MFCC* is uneven, it is difficult to map and compress each frame individually in *VQ* to achieve the desired effect. Instead, the classification is performed using the clustering method before calculating the *Vector Quantization*, which can use a centroid value to represent the information of surrounding data points in the same cluster. This can increase the corresponding representation of a point to generalize more data from one point. Besides, using clustering to group the data before calculating *VQ* is also equivalent to compressing the data first, which facilitates subsequent calculations and improves the operational efficiency of the project. Specifically, the realization of *K-Means* is divided into the following steps: initial centroids selection; classifying the remaining points according to the distance; updating the positions of the centroids; iterative optimization.

First, the initial centroids are chosen randomly. Since *K-Means* is an unsupervised learning method, I first store all the data points in a list and find K centroids randomly by generating random numbers. Here, K represents the total number of clusters that can be partitioned as expected by the user. In this step, the initial centroids should be generated directly from the current set of points. This is to prevent the possible bias caused by the generated random numbers, which can lead to local optimization of the whole clustering process. The *centroids_list* in the code represents the set of all centroids. The variable *random_cent* represents a randomly generated index, indicating that the element in the *centroids_list* is obtained, and this element (data point) is used as an initial centroid.



```
random_cent = np.random.randint(len(data) - 1)
centroids_list = np.array([data[random_cent]])
```

Figure 2.7： Find a random point in the data list.

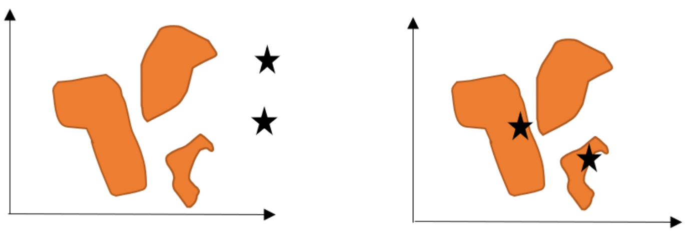
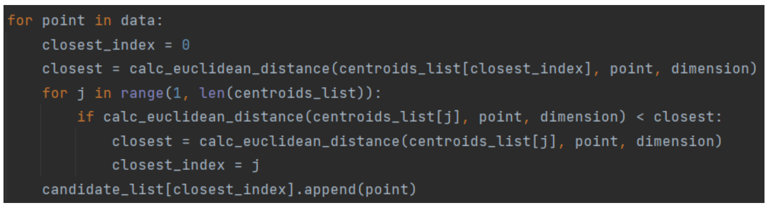


Figure 2.8: This figure shows the difference between generating the initial centroids by random numbers (left) and randomly selecting points among the data points as the

initial centroids (right). The orange area represents the data points distributed, and the black star represents the initial centroids. When the initial centroids are generated by random numbers, they may appear anywhere in the given range. As shown in the left figure, the centroids obtained at the end of the clustering may still be located on the rightmost side of the graph, making the results meaningless, while the initial centroids are obtained by drawing among the data points, they are still making sense for the current point and the surrounding neighborhood even if they are currently located at "remote" area. In contrast to the right figure, the left figure is likely to have a situation where no data points appear within a certain range around the initial centroids.

Based on the centroids selected in the previous step, the remaining points need to be classified into the corresponding clusters based on their distances to the centroids. Therefore, it is necessary to first find the distances of each point to the different cluster centers. In this step, I use a nested for-loop to calculate the Euclidean distance between two points. A simple value comparison determines the centroid closest to each point, i.e., the cluster to which each point belongs in this classification. I used a two-dimensional list and added data points to this list using the index of the nearest centroid as the position in the *candidate_list*. This index corresponds to the index of the list of centroids so that the cluster centers can be matched to points in the same cluster by querying the position index.



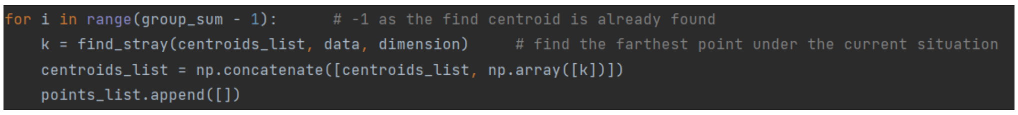
```
for point in data:
    closest_index = 0
    closest = calc_euclidean_distance(centroids_list[closest_index], point, dimension)
    for j in range(1, len(centroids_list)):
        if calc_euclidean_distance(centroids_list[j], point, dimension) < closest:
            closest = calc_euclidean_distance(centroids_list[j], point, dimension)
            closest_index = j
    candidate_list[closest_index].append(point)
```

Figure 2.9: Classification of the remaining points according to distance

It is worth mentioning that during the calculation of *K-Means*, the process should first consider the composition of the data when calculating the Euclidean distance and cannot use the Euclidean measure for all types of data in general. All calculations of distance in *K-Means* assume that the data can be measured using the Euclidean distance (the smaller the distance, the more similar the two data are). When some data, such as data allowed to be measured along the curve surface only, cannot be measured using Euclidean distance, it is necessary to first convert the data into a Euclidean metric. Otherwise, the obtained distances will become meaningless (Qing, 2016).

Updating the positions of the centroids. This step first requires calculating the mean of each cluster. I use the *NumPy.mean()* function and set *axis=0* to calculate the mean of each cluster along each column. The data involved in this project are all high-dimensional, and the numbers in the same column represent the size of the same axis. The mean value of each axis calculated will be used as a new centroid corresponding to the size of the axis. After the mean calculation, this new centroid will be stored in the *centroids_list* instead of the original cluster center. There is a special case in this step. If there is a cluster without a centroid, that is, there are no elements in this cluster, this cluster will be removed directly. Finally, the whole algorithm needs to repeat the above steps of data classification and centroid update, and iterative optimization. Usually, the current clustering is optimized when the position of the centroids no longer changes. However, to reduce the amount of computation and increase the efficiency of the program, I set the total number of iterations to 30 in this project. After experimentation, this number of iterations is sufficient to ensure optimal clustering for the dimensionality of the data and the total number of clusters in this project.

The biggest drawback of *K-Means* is that it tends to cause local optimality in the results when the data set is too large. This is because the K values (total cluster numbers) need to be predetermined before the algorithm starts and randomly selected based on the size of K to generate the initial centroids. while their selection is sensitive because it is the basis for subsequent iterations of the algorithm. *K-Means++* is a modified version of *K-Means*, but the implementation of the clustering process is the same for both methods. The only difference is in the way the initial centroids are selected, while the basic idea behind the *K-Means++* algorithm is that the initial cluster centers should be as far away from each other as possible. Like *K-Means*, *K-Means++* will select a random point from the input set of data points as the first centroid. However, for each remaining point p in the data set, the distance D from the selected centroid is calculated, and the selection of new data points as new centroids should ensure that the points with larger D have a higher probability of being selected (Kapoor and Singhal, 2017). This process will be repeated K - 1 time (recall that the first centroid is selected randomly). After that, the algorithm will run the standard *K-Means* algorithm using these K initial centroids. The implementation of this process of selecting the initial points is shown below.



```
for i in range(group_sum - 1):       # -1 as the find centroid is already found
    k = find_stray(centroids_list, data, dimension)     # find the farthest point under the current situation
    centroids_list = np.concatenate([centroids_list, np.array([k])])
    points_list.append([])
```

Figure 2.10: The *find_stray()* function will find the farthest point from the current cluster center among all the points except the one that has been selected as the centroid, ensuring that the distance between all the initial centroids is as large as possible.

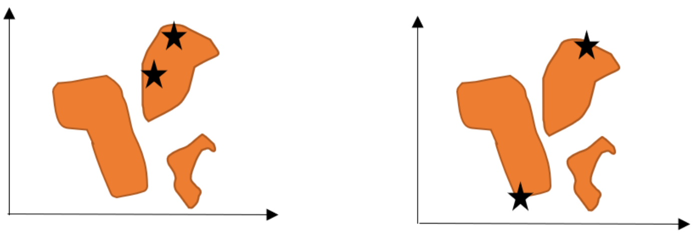


Figure 2.11: This figure shows the difference between *K-Means* (left) and *K-Means++* (right). The orange areas represent the data points distributed in, and the black stars represent the initial centroids. When the initial cluster centers are further

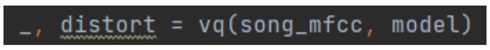
away from each other, they are also more independent from each other. This is because in normal *K-Means*, the initial centroids may be next to each other according to the principle of randomness, and the idea of *K-Means* is to group similar or neighboring points in the same class. Therefore, these points close to each other should probably be grouped into the same class instead of existing as two separate centroids. This may cause the final clustering results to become less reliable.

Clustering does not change the dimensionality of the original data, but only classifies the data points according to the given computational rules. The final output of this algorithm is an array of n * K consisting of all cluster centers, representing the spatial location where each cluster center is located.

2.1.4 Vector Quantization

In this project, each frame of the song audio signal is a vector. The part about *VQ* processing can be summarized as follows: train the codebook from the *MFCC* of the songs in the training set. Given the set of *MFCC* feature values of a song in the test set, a codebook as a control criterion, and the number of quantized values (total number of cluster centers), then find a codebook with the minimum average distortion and the partition of the space.

In practice, the implementation of *VQ* is based on the LBG-VQ algorithm and is divided into the following five steps. Firstly, divide the initialized reference codebook (the set of centroids in this project) into N subsets according to the nearest neighbor condition; Secondly. compute distortion and the new code vector; Thirdly. compare the current distortion improvement with the distortion threshold; Fourthly. iterative optimization; and fifthly. output the results. LBG-VQ is an iterative optimization algorithm. This algorithm requires alternating the codebook and the partition of the space so that the distortion continuously converges to its local minimum (Makhoul, Roucos, and Gish, 1985). The calculation of *VQ* in this project is done by the *cluster.vq()* function in the *SciPy* library. This function takes two *ndarray* as arguments, representing the observations and the codebook, where the observations are the set of eigenvalues of the *MFCC*, and the codebook is the aggregation of the centroids obtained by the clustering algorithm. The function assigns the codes in the codebook to each observation and compares each observation vector in the observation array with the centroids in the codebook and assigns the code with the closest centroids to it. In this project, the purpose of *VQ* is not to compress the training audio data, but to classify the songs by obtaining the distortion of the songs in the training and test sets and classifying the songs based on this distortion value. Therefore, this function does not need to return the codebook after training.



```
_, distort = vq(song_mfcc, model)
```

Figure 2.12: The *vq()* function takes the *MFCC* of the current song and the comparison model (training set centroids) as input parameters. This function will return two values, the trained model codebook, and the distortion value.

As a condition for comparison, the set of centroids obtained by clustering in the training set is used as a codebook for calculating the *VQ* distortion during both model training and classification, instead of calculating the distortion of the respective *MFCC* and the centroids of the respective songs separately. This is because eventually, the distortion values of the training and test sets need to be compared. Therefore, both should first be compared based on the same criterion to calculate a meaningful distortion. Since the cluster centers are obtained by clustering based on the *MFCC* feature values, and one of the properties of clustering is that the distance from each point in the same cluster to the centroids of this cluster is smaller than the distance to the centroids of other clusters. It can be inferred that for a song its *MFCC* eigenvalue is the closest distance to its own centroids. Therefore, the distortion value computed by *VQ* should be minimal at this point. This distortion value will be used to compare with the values of the rest of the songs. If the eigenvalues of *MFCC* and the eigenvalues of centroids are considered as two curves respectively, the distortion can be considered as the difference between these two curves. When the two curves are closer, the difference is smaller, the distortion value is also smaller. And when the distortion value is zero is when the two curves overlap. When using the model to compare with other test set songs, it is equivalent to comparing the distance between the two. As the distance (*VQ* distortion) between the two vectors is less than a certain degree it can be inferred that the two songs are similar aurally (have similar *MFCC*) and therefore it can be further obtained that the two songs should belong to the same genre. Otherwise, the two songs will be considered to belong to different genres.

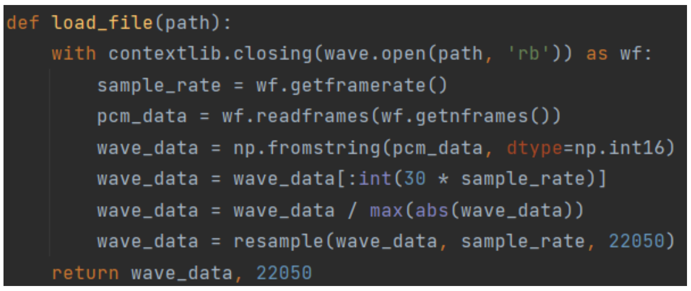
2.2 Other Details

The above section has described the different algorithms and the corresponding code implementations. This section will introduce some code details of the program and the implementation of other design features.

Firstly, the program reads the input flags and path arguments through the *sys* module. There is a list *argv* in this module, which records all the running arguments by the order of input and spaces. But also, for this reason, users should take care of when naming audio files to avoid using filenames with spaces. For paths, the single character "\" is interpreted as the conversion symbol in Python rather than a backslash for the path. *argv* takes arguments that are first replaced by a series of *replace()* functions, preventing the path from being read unproperly due to the unhandled “\”. In addition, when reading and writing files, the program will use UTF-8 encoding by default to ensure that all special characters are encoded properly.

The program opens an audio file in binary form with *wave()* and the corresponding flag *rb*. The *contextlib.closing()* in this place is to create a content manager to ensure the class can open the file correctly using *with* even if the *__enter__()* and *__exit__()* methods are not implemented. Next, I use the functions from *wave* to get the sample frequency and audio frames and use *fromstring()* to convert the binary stream *pcm_data* into a one-dimensional array. The pcm format here is the pulse-code modulated audio format, which simply means that the sound (analog signal) is turned into a series of digital signal consisting of 0's and 1's and recorded.

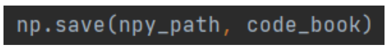
The program only retains the first 30 seconds of audio data when reading the song, rather than processing the entire song. This is because the genre identity of a song can in most cases be determined from the first few seconds while the genre itself is unlikely to be changed during the song. Therefore, it is not necessary to obtain the *MFCC* of the whole song to have enough data to support and make a judgment. The amount of computation needed can be reduced by performing this step before the calculation. However, due to the limitation of the code implementation method, the length of the read audio must be longer than thirty seconds, otherwise, the program will crash. In the future improvement, I will compare the song length with the 30 second limit by using the *min()* function in the later improvement to ensure that the index does not overflow by taking the smaller value. Here, I truncate the array obtained from the previous step with only keep the data that the first 30 seconds of the audio indicates. And this length equals to time multiply by the sample rate. The last two lines will re-format the audio sample so that the sample rate will remain 22050. This is used to standardize the audio file to make sure all the files in the dataset have the same format, to better compare between different songs. Otherwise, the total number of the *MFCC* eigenvalue will be different and will lead to a meaningless result. There are two ways to achieving the conversion, either by using *ffmpeg* or relevant functions. Considering *ffmpeg* needs additional installations of the plugins, I chose the latter in the project. The function *resample()* comes from *librosa* that will change the give audio data into the designated sample rate (in this project, it is 22050). As the program have already done the conversion before return the audio data array, the function returns 22050 directly instead of returns the variable itself.



```python
def load_file(path):
    with contextlib.closing(wave.open(path, 'rb')) as wf:
        sample_rate = wf.getframerate()
        pcm_data = wf.readframes(wf.getnframes())
        wave_data = np.fromstring(pcm_data, dtype=np.int16)
        wave_data = wave_data[:int(30 * sample_rate)]
        wave_data = wave_data / max(abs(wave_data))
        wave_data = resample(wave_data, sample_rate, 22050)
    return wave_data, 22050
```

Figure 2.13: This function will intake the path to an audio file and convert the audio to the binary-string format. The binary string will be executed with a series of proper operations before return for the future use.

In the last step of calculating the song categorization by *VQ* distortion value, this distortion value is not simply obtained by comparing the eigenvalues of each *MFCC* and adding the distortion values, but after the calculation, it is also necessary to divide this sum by the total number of *MFCC* points to calculate the average of distortion values. This is because in the process of calculating *MFCC*, the sampling rate and time value of the song, etc., have an impact on the selection of *MFCC* feature points. This leads to the fact that the overall number of *MFCC* selected in the process is uncertain. And the sum of distortion values of songs with more *MFCC* frames will be increased accordingly. All the test set data in this project has been preprocessed, so this step is not obvious. However, when using this project to solve real-world problems, such as using randomly downloaded songs from a website as the input sample set (and these songs are likely to have different formats and attributes, especially sample rate), this processing allows all calculated distortion values to be compared based on the same criteria, making calculated distortion value meaningful.



```
np.save(npy_path, code_book)
```

Figure 2.14: Format of a *NumPy.save()* call

In terms of storing and reading the data model, the program generates two files at the same time. First, the result of the clustering (in this project, a one-hundred-and-sixty 8-dimensional data point, i.e., a *ndarray* of shape 160 * 8) will be saved as a model codebook in a *.npy* file using the *save()* function that comes with *NumPy*. At the same time, the distortion of the training *MFCC* and codebook comparison is stored in a *.txt* file. This is because the distortion calculated during training will be used later as a criterion for classification in the process of prediction and classification. To prevent unnecessary calculations and to ensure data usage, this value will also be stored locally and read as needed. For ease of finding and management, the file names of both model files will be kept the same as those of the training audio, replacing only the file extensions. when the model is loaded, the user only needs to enter the filename of the .npy file to complete the loading of both files.

3. Challenges and Solutions

I also encountered a lot of difficulties in the development of the project. This project was more of a combination of algorithms, and almost all of the algorithms involved were new to me. Therefore, most of the time spent on this project was allocated to algorithm research and learning. The most difficult part of the project was also the selection of algorithms. Due to the lack of reference material on the Internet, I had to analyze other people's codes or look up literature to improve the process. One of the biggest problems was that whenever I considered an implementable method, I was not sure if it was valid or meaningful. In the next part of this section, I will list the challenges I experienced in completing this project and the corresponding solutions.

3.1 Feature Definition and Extraction

The first difficulty I encountered was not knowing how to convert a song into data that a computer could understand. As humans, we ourselves rely on our auditory perception when listening to a song to get information about it. This means that we collect sound waves through our ears and process the information obtained in our brain. Here, we will not explore how the brain processes the collected information, but more importantly, how this information is collected. We know from our own experience that while listening to a song our eyes are not used to "seeing" the frequency, volume level, beat, etc. of the song, but it is all done by our ears (imagine that we can still hear the music with our eyes closed and even infer information related to the song). Therefore, for us humans, auditory perception is the most intuitive and direct way to get information about a song.

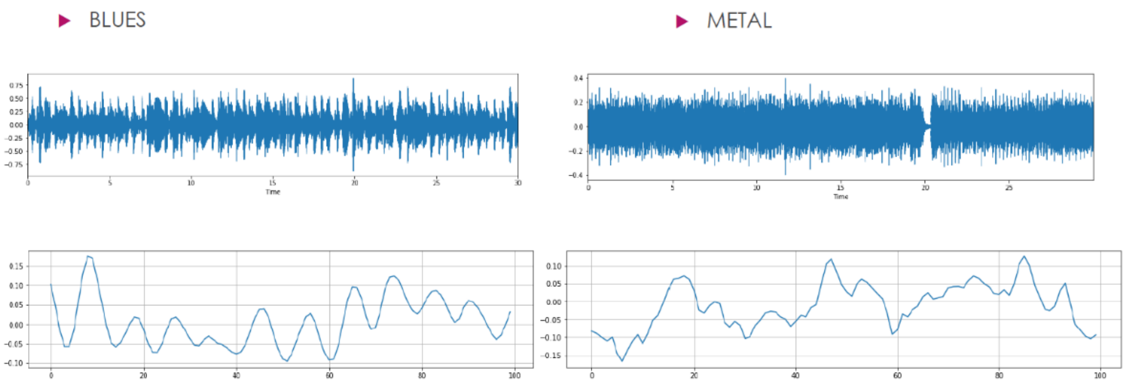
Based on the above ideas, I learnt that I need to convert auditory content to visual content, or rather, convert to information that could be expressed in numbers and equations. In my own experience, hearing a song and recognizing the genre it belongs to is more by the combination of musical features such as rhythm, beats and lyrics, in addition to experience. For most listeners, they may not be able to accurately distinguish between sub-genres (take house music as an example, there are ambient house, UK house, Chicago house, acid house, and so on), but they are able to directly distinguish between hip-hop, jazz, and classical music. Because the beats and rhythms of these genres are completely different, this leads to a significant difference in the styles of these pieces from an aural perspective. Besides, there is also a big difference in the instruments used between the different genres. For example, it is rarely (even impossible) to find the electrical guitar in classical music. These attributes mentioned above are the musical characteristics of a song. Therefore, I first treat the musical characteristics of a genre as a genre's eigenvalue.

I looked on the web to find libraries for processing music in Python. In addition to python's library for reading audio files, I also found *music21* and *essentia*. *music21* is a Python library developed by MIT for processing audio and performing music analysis. The library includes a variety of functions and utilities for audio analysis and even music production, and it contains a lot of content based on music theory. Thus, it is very powerful and allows me to use code to extract notes, chords, and chords progression. I also tried to use *essentia* to do a beat analysis of the song (Boxler, 2020, p. 32). Specifically, I used the beat detection, BPM (beat per minute), and beat loudness features included in the rhythm descriptor. I expected, as mentioned before, to isolate these features from a song that represents the characteristics of the genre, and to classify the music according to these genre-based features.

But as I extracted musical features from some songs, I found that classification by these musical features works for our ears but not for computers. This is because all the musical features or the collections of musical features obtained are not representative enough. Even if two songs belong to the same musical style, their beats and rhythms are different. Therefore, it is difficult to define genre numerically by its musical characteristics. If blues are defined with BPM ranging from 50 to 100, then the program will not be likely to recognize a blues with a BPM of 101 correctly when it encounters it. This is because the definition of genre is very vague and there is no hard and absolute definition. This makes it difficult in practice to classify styles directly by judging whether the data is in an interval or not. Another reason why musical features are not applicable here is that a song involves too many musical features, including chords, melody lines, lyrical emotions, maximum and minimum volume, etc. It is impossible to decide which features are worth keeping and which can be omitted during the whole extraction process. From the practical point of view of this project, it is more effective to use the volume level to distinguish rock music, but it may not play an important role in other genres' judgement. Therefore, I gave up using musical features as a direct basis for genre classification, but I think some of the features can still be used as an auxiliary basis for classification (see *5.2.1 Align with the musical features*).

We know from physics that the foundation of sound is a sound wave produced by the vibration of an object. Therefore, to extract the characteristics of music, I considered starting from the waveform diagram of sound waves and finding a regression equation that could express the approximation of the sound waves or frequencies of songs of this genre for a different genre. But this idea was quickly dismissed. First of all, the most important problem is that there is almost no regularity in the curve of the diagram, which makes it very difficult to find a function approximation. Secondly, the characteristics of different genres are not well distinguished from each other. This approach may work between fast and slow songs, but it is difficult to get enough recognition for songs that sound similar. Such a simple calculation of the waveform graph of the regression equation is likely to lead to two or more possible classifications for the same song, especially for slow songs like balled and soul, which have almost the same waveform graph. Additionally, the process of calculating regression equation involves selecting sample points in the image and selecting reference points is unfriendly for an image with almost no logic in its development like a waveform graph because the total number of sample points and their selection is difficult to define and implement. This will absolutely make the whole procedure more difficult to implement and computationally intensive to run.

Finally, by looking up documents, I learned about the idea of *Mel-scale Frequency Cepstral Coefficients*, which transforms an otherwise complex, illogical audio waveform into a development trend (see *2.1.1 Mel-scale Frequency Cepstral Coefficients*).



Figure[11] 3.1: *MFCC* depicts the trend of the corresponding waveform plots. This figure shows the waveform diagrams of blues and metal and their corresponding *MFCC* spectrum, respectively. From this figure, it can be seen that there is a big difference in the spectrum between different genres due to the difference in style characteristics. *MFCC* is a way to describe the aural characteristics of a song by converting the waveform into a spectrum that shows the trend, which is equivalent to amplifying the trend of the waveform so that the characteristics can be more clearly reflected. Since songs of the same genre are also more likely to be similar in spectrograms, comparing the distances between different spectrograms is theoretically a feasible way to identify and classify song genres.

Compared to the idea of classification by waveform graph mentioned above, *MFCC* is equivalent to an optimization. It obtains the trend of the waveform while retaining detail and resolution. This value is more meaningful than simply performing the calculation of the regression equation because the loss during the graph change is substantially reduced. In addition, each frame is segmented during the calculation of *MFCC*, which facilitates the subsequent clustering and the process of calculating the spatial distances. Otherwise, the traditional way of calculating the regression function also needs to select enough, representative points in the equation to compare with other regression equations. Therefore, I think this coefficient is very suitable as an eigenvalue of a song, and eventually, *MFCC* was used as the input eigenvalue of music and genre in this project.

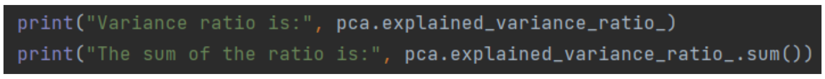
[11] This image comes from Yoğurtcuoğlu and Çokça, 2020

3.2 Processing of High-Dimension Data

When I was just learning *K-Means* I looked up a lot of information online, all of which had one thing in common when they explained that they only clustered two-dimensional arrays and generated corresponding images based on this result. Thus, when I first started learning the clustering algorithm and methods, I mistakenly thought that *K-Means* could only be used for clustering in two dimensions due to my lack of understanding of the algorithm itself. That is, each input parameter to *K-Means* can only be represented by two values with x-axis and y-axis. So, I thought of putting the eigenvalues of a song on a two-dimensional coordinate and representing the position of a song's eigenvalues in the space in the form of a point. After that, *K-Means* was executed to obtain the classification results of the songs in the test set. Based on this idea, I started to look for ways to obtain the average value of *MFCC*, which is obtained as a set of the total number of frames multiplied by the dimensionality of each frame. In the function that obtains *MFCC*, the dimension of each returned frame is up to 39 dimensions. Therefore, the returned value of *MFCC* can be seen as a n * 39 matrix. It is worth noting that here I still simply assume that I only need to turn the data of this set into the form of a two-dimensional coordinate point. I tried methods like calculating the mean of the points and the median of the points within the matrix, calculating the spatial distance between the points, calculating the vectors of rows and columns of the matrix, and even selecting a random point from within the matrix to represent this matrix. These methods did yield two-dimensional data, but I found that the data obtained could not represent the matrix itself, i.e., the data obtained from these calculations were meaningless. At the same time, I also realized the problem that reducing the n * 39 matrix to a simple point must have seriously affected the details of this matrix. Not only can this point not represent the matrix, but the values obtained by this transformation can no longer be reduced to the original song music characteristics.

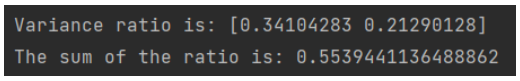
After further study of *K-Means*, I understand that it can be used to perform clustering of high-dimensional data. High-dimensional data can be obtained by calculating the Euclidean distance between two points. But the dimensionality of the data is too high to be meaningful in clustering, as the space between the points of high-dimensional data makes the definition of density and distance meaningless. This is also called the curse of dimensionality. According to the study, all the points are at a similar distance one from the others, so the distance relationship between data points will be weakened. As the dimensionality increases, the volume of the space will increase exponentially and rapidly, so that the available data becomes sparse (Giraud, 2014). This sparsity can cause some difficulties in statistical significance since the amount of data required to support the results in order to obtain a statistically reliable result tends to increase exponentially with the increase in dimensionality. In the case of *K-Means*, for example, the cluster of the data usually relies on the detection of the distance between data points. However, in high-dimensional data, all objects appear to be sparse and dissimilar in many ways, meaning that the amount of data needed to support and compute doubles with each additional dimension.

I started to look for a way to map high-dimensional data to low-dimensional and found *Principal Component Analysis*, which maps the 39-dimensional data features of the original *MFCC* to the selected K-dimension, and this reconstructed K-dimension data is the new orthogonal feature that can be used to generalize the original 39-dimensional data. I initially kept only two dimensions of data for each frame due to the influence of “two dimensions”. However, I found that excessive dimensionality reduction would affect the integrity of the data by calculating the corresponding component ratios in the code. After a series of experiments and comparisons, I set the dimensionality of the downscaled data to eight. This dimension size does not cause the data to be too scattered. At the same time, the result after dimensionality reduction will retain about 95% of the original information and will not result in a lot of missing information.



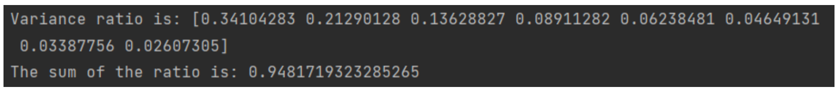
```
print("Variance ratio is:", pca.explained_variance_ratio_)
print("The sum of the ratio is:", pca.explained_variance_ratio_.sum())
```

Figure 3.1: The first line will calculate the proportion of the information of each new explained variance to the total information amount of the original data. Explained variance indicates the information carried on each new feature vector. The larger the variance, the more important this feature is. The second line calculates the proportion of the new-retained data to original data through the sum of explained variances.



```
Variance ratio is: [0.34104283 0.21290128]
The sum of the ratio is: 0.5539441136488862
```

Figure 3.2: Compressing data from 39 dimensions to 2 dimensions only preserves approximately 55% of the original contents.



```
Variance ratio is: [0.34104283 0.21290128 0.13628827 0.08911282 0.06238481 0.04649131
 0.03387756 0.02607305]
The sum of the ratio is: 0.9481719323285265
```

Figure 3.3: Compressing data from 39 dimensions to 8 dimensions retained approximately 95% of the information

It is worth mentioning that even if the data is downscaled, the *MFCC* of a song still corresponds to several hundred frames (bars) of data, which reflect the changing pattern of the waveform of a song, and the data are related to each other in some way. It would be meaningless to transpose the matrix and downscale the data from hundreds of frames to one frame again, and the information contained in the data would be lost as well. Therefore, the project uses *VQ* instead of *K-Means* for the classification process.

3.3 Identification Standard

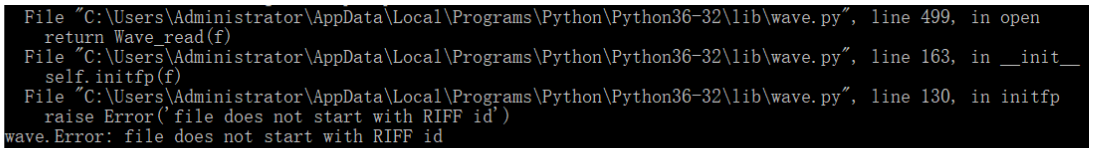
One of the original thoughts of this project was to give a test set containing only blues and use *K-Means* to classify the data to determine if there are any outliers. This leads to the problem that the clustering results obtained from *K-Means* are not intuitive in determining which group is blues and which are not because *K-Means* requires a random selection of points from the dataset. But as a user, we do not know whether the program is selecting the target data or the outlier data. This may lead to the final presentation of the classification results where the two types of data are well separated, but it is not clear which is really needed, and which is the outlier and should be discarded. Another problem is that the eigenvalue of *MFCC* includes hundreds of frames. Even if the data of each frame has been downscaled, it is not feasible to merge hundreds of data into one by *PCA* because the details of the data are still lost greatly in the process. So even if *K-Means* could be used for classification, finding the only one that can replace the whole song among the hundreds of data representing a song is also a big headache. For the above two reasons, I made a small change at the end of this project by abandoning the direct use of *K-Means* to classify songs and, instead, calculating the distance of the difference between the eigenvalues of two songs (the target song and the song that already has a genre tag) to determine whether the two songs belong to the same genre classification.

4. Outcomes and Reflections

In this section, I will describe each of the several methods I used to test this project. I will describe the testing methodology and the results of the tests in *4.1 Subsequent tests and results*. I will also analyze and evaluate the results and, if possible, examine the reasons for the situations that non-compliance with the expected results in *4.2 Result Analysis*. But I will not elaborate on the future expectations and possible improvements for this project until *5. Applicable Fields and Optimizations*.

4.1 Subsequent tests and results

The project already includes several trained models and audio files that can be used for training and testing. Users are also encouraged to use their own audios and audios downloaded from other sources to execute the project and test its applicability. Although I mentioned earlier in this report that the program would only intercept the first 30 seconds of audio, I had pre-processed the songs for the test set. To prevent the total size of the project from increasing rapidly due to a large number of media files, I truncated the songs to reduce the total sample size of the test set. However, the truncated songs were still longer than 30 seconds. So, to some extent, the program still trims the song length when reading the audio. Please note that since I rewrote the method of reading the audio file, all input audios should be guaranteed to be in .wav format when using this program. Using other formats (such as .mp3, .flac, and .aac) may cause the project to report errors. Please refer to section *1.4 Execution* for more information on how to run the project.



```
  File "C:\Users\Administrator\AppData\Local\Programs\Python\Python36-32\lib\wave.py", line 499, in open
    return Wave_read(f)
  File "C:\Users\Administrator\AppData\Local\Programs\Python\Python36-32\lib\wave.py", line 163, in __init__
    self.initfp(f)
  File "C:\Users\Administrator\AppData\Local\Programs\Python\Python36-32\lib\wave.py", line 130, in initfp
    raise Error('file does not start with RIFF id')
wave.Error: file does not start with RIFF id
```

Figure 4.1: A WAV file is an example of the Resources Interchange File Format (RIFF). It usually consists of three chunks (the basic units of RIFF), the first of which is the RIFF chunk. However, a MP3 file does not conform to RIFF, so there is no special chunk added at the beginning of the file. When running, the program reports an error because the RIFF chunk of the current audio file cannot be read properly.

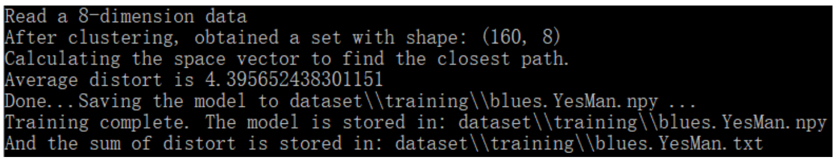
For the input audio files (both training and test sets), the naming of the files needs to conform to the given format:

[labelled genre].[filename].wav

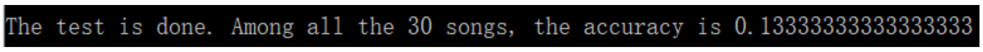
This is used to ensure that the program can accurately identify and compare the annotated genre of the two songs. It is worth noting that this naming rule is adapted to better determine the recognition accuracy, rather than obtaining the corresponding genre of the audio by the filename directly. in the program, this labelled genre is compared with the genre predicted by the program to determine the accuracy of the program's classification result. The corresponding result of each test case will be added in the appendix.

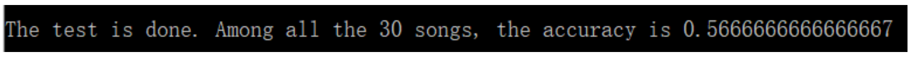
4.1.1 Single Genre Test

The program satisfies my expectation when classifying a set of music that belongs to the same genre. As the title suggests, the test set for this test will only include songs from a single genre. Here, I chose a test set with only blues music. This test is to determine the program's ability to distinguish and classify well when faced with a single genre. The test will be divided into two parts. Both parts will use the same song as a comparison standard for classification, i.e., the program will be run based on the same model in both parts. The only difference between the two parts is the choice of songs for the test set. In the first part, the songs in the test set will consist of only thirty randomly selected blues songs from the GTZAN library. In the second part, the songs of the test set will be selected from online resources that from the blues category of Cloud Music (a similar application to Spotify), and this part will also include the same number (thirty) of blues song clips. Specifically, the testing process will be as follows: first, I will use a blues found from the Internet as input data for model training. Here, the song chosen is *Yes Man* by B.B. King (*blues.YesMan.wav* in *dataset\training*). This song will be different from both GTZAN and the song included in Cloud Music. After the model is trained, I will use this model to classify the songs in each of the two subfolders (folders containing songs from two different sources) under the folder *dataset\SingleGenre\blues*. The classified result of each song will be represented by discrete data in the form of blues or not blues. After classifying thirty songs in each of the two test sets, the program will calculate the accuracy of the classification for each of the two test sets.

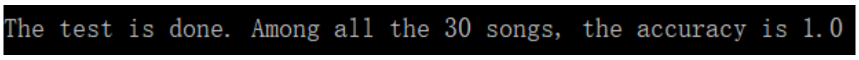

```
Read a 8-dimension data
After clustering, obtained a set with shape: (160, 8)
Calculating the space vector to find the closest path.
Average distort is 4.395652438301151
Done...Saving the model to dataset\\training\\blues.YesMan.npy ...
Training complete. The model is stored in: dataset\\training\\blues.YesMan.npy
And the sum of distort is stored in: dataset\\training\\blues.YesMan.txt
```
Figure 4.2: Training in progress

The test results are as follows:


```
The test is done. Among all the 30 songs, the accuracy is 0.13333333333333333
```
Figure 4.3: Classification results' accuracy of GTZAN

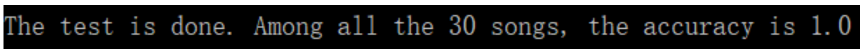

```
The test is done. Among all the 30 songs, the accuracy is 0.5666666666666667
```
Figure 4.4: Classification results' accuracy of Cloud Music

After performing the first-round tests, I found that the choice of the training set model for the test seemed to be somewhat flawed. This was due to the large accuracy gap between the two test sets. Therefore, I reselected a song (*blues.00055.wav*) from GTZAN for training and obtained the second prediction based on the model of this song. And after replacing the training model, the accuracy of both folders reached 100% this time.



```
The test is done. Among all the 30 songs, the accuracy is 1.0
```

Figure 4.5: GTZAN's second test result with *blues.00055.wav*

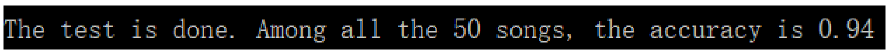


```
The test is done. Among all the 30 songs, the accuracy is 1.0
```

Figure 4.6: Cloud Music's second test result with *blues.00055.wav*

4.1.2 Cyclic Blues Test

The program has a good classifying result in a test of identifying 100 blues recursively. In this test, I take turns training the 100 blues provided in GTZAN, comparing, and classifying each of them with the remaining 99 blues according to their corresponding models. For each song used as a model for classification, I will calculate and record the accuracy of the classification result. After all the 100 songs have been identified and classified, I will first compare the difference between the correctness and find the model with the highest and lowest correct rates. After that, I will use the model with the lowest accuracy to run the *mixed-genre test* (details in *4.1.3 Mixed-genre Test*) and use it to determine if the song has a higher classification accuracy in other genres. This test uses a Python script *test_blues_recursively.py* that has been included in the project folder as well. It loops through the functions of genre.py and parts of the main program to perform the training of the model and the classification procedures. To facilitate the analysis of the final results, I record the correctness of each model prediction in a file called *results.csv* in the program folder. Please refer to this file for more details. After performing all 100 songs, the average of all correct rates was 76.3%, which was basically what I expected at the beginning. I found that *blues.00001.wav* had the lowest prediction rate, with an only 4% of songs among 99 blues are classified correctly. Therefore, as planned before this test, I reused the model with this song as the input training data and performed the mixed-genre test. In this test, the prediction accuracy of the model generated from *blues.00001.wav* reached 94%. The relationship between the prediction accuracy and the corresponding genre is as follows:



```
The test is done. Among all the 50 songs, the accuracy is 0.94
```

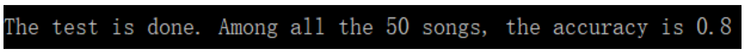
Figure 4.7: Results after using the outlier blue to classify other genres

	Classical	Country	Disco	Jazz	Reggae
Total #	10	10	10	10	10
Correct Prediction	10	10	10	7	10
Accuracy	100%	100%	100%	70%	100%

Chart 4.1: The corresponding correctness ratio of each genre in the mixed-genre test.

4.1.3 Mixed-genre Test

When it comes to a test set with the mixed genre, the program shows a shortcoming when dealing with the threshold and the classified result. The purpose of this test was mainly to explore the effectiveness of this project in classifying songs when encountering different genres. The test set will consist of 5 different genres and 10 songs from each genre. The songs used for the test were not only from GTZAN but also from Cloud Music. All 50 songs will be mixed in the same folder. The songs in the test set will be as different as possible from those used in the *4.1.1 Blues-only Test*. In the model training section, I will randomly select one of the five genres and find an additional song in that genre (so make sure it is different from the songs in the test set) to train the model. In the actual test, the song used as the comparison model is a jazz song called *jazz.ComeRainOrComeShine.wav* and the model is called *jazz.ComeRainOrComeShine.npy* in *dataset\training*, respectively. In the implementation phase, I will execute the ***-tr*** command of the program directly. First, the program trains a song model of the training set and classifies 50 songs based on this model. Finally, the program executes the accuracy in this case based on the classification results. The purpose of this test is to mock the complex situations that the program may face in practice. In a real recommendation system, the program often encounters many different types of data that are mixed. Therefore, I expect the program to perform well in the face of complex, multiple types of data to show the effect in the practical application.



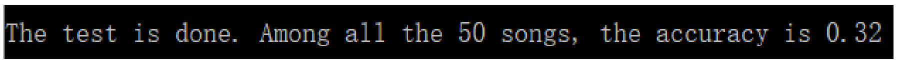
```
The test is done. Among all the 50 songs, the accuracy is 0.8
```

Figure 4.8: The test result of the mixed-genre test

	Classical	Country	Disco	Jazz	Reggae
Total #	10	10	10	10	10
Correct Prediction	10	10	10	0	10
Accuracy	100%	100%	100%	0%	100%

Chart 4.2: The accuracy of each genre in the mixed-genre test.

It can be seen from the result that I was using jazz as the comparison model while the accuracy for predicting this genre is 0. Since the results of the first experiment were not satisfactory, I found other songs with different genres from GTZAN that re-trained the model with another jazz (*dataset\training\jazz.00019.wav*) and a disco (*dataset\training\disco.00063.wav*) respectively to execute the test again. The results of the runs and the correct rates corresponding to each genre are as follows:

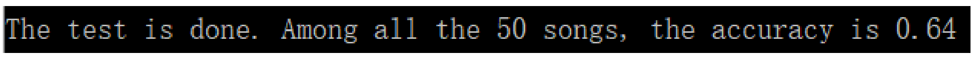


```
The test is done. Among all the 50 songs, the accuracy is 0.32
```

Figure 4.9: Accuracy for using disco as the prediction model

	Classical	Country	Disco	Jazz	Reggae
Total #	10	10	10	10	10
Correct Prediction	0	3	6	2	5
Accuracy	0%	30%	60%	20%	50%

Chart 4.3: The accuracy of each genre when using disco as the model.



```
The test is done. Among all the 50 songs, the accuracy is 0.64
```

Figure 4.10: Second mixed-genre test with a different jazz model

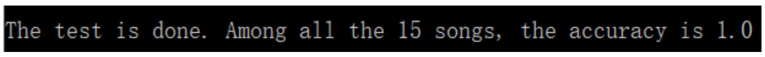
	Classical	Country	Disco	Jazz	Reggae
Total #	10	10	10	10	10
Correct Prediction	1	8	9	5	9
Accuracy	10%	80%	90%	50%	90%

Chart 4.4: The accuracy of each genre in the mixed-genre test with a jazz model.

4.1.4 Sub-genre Test

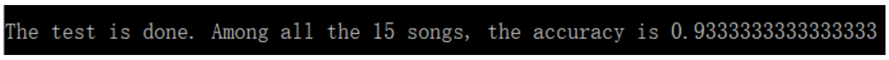
I also found that the program cannot distinguish the corresponding sub-genre for the general genres in this test. This sub-genre test can be thought of as a breakdown of the *4.1.1 Single Genre Test*. In this test, I will classify piano-based rock and roll, vocal group (doo-wop), and guitar-based rock and roll in early rock. I will find 5 corresponding songs under each type of sub-genre, for a total of 15 songs. I will first select and train an additional rock (any rock and roll songs that are not limited to these three types) as a model and use this model to classify these fifteen songs and calculate the accuracy of the classification. I will then select one song from each of these 15 songs to represent this sub-genre, and I will use this selected song as the input audio to train a model that will determine whether the remaining 14 songs can also be correctly identified and classified. It is worth mentioning that the definition of the genre has been blurred, and for most songs, the sub-genre here is a more subjective concept. Therefore, when selecting songs for the training and test sets, I tried to select songs with more obvious features as input data to ensure that the labelling of the data in the test set is as accurate as possible. For the genre labelling of the test set, I used the specific sub-genre as the filename that functions as the basis for determining the correctness. Since the purpose of this test is to determine whether a sub-genre can be classified as well, and not just to distinguish a general genre as before, the logic to be followed in this test is that songs of the same genre but belonging to different sub-genres will be considered as two different genres.

To start with, when training the model with a random rock (*rock.00036.wav*), the classification accuracy reached 100% for the songs within the fifteen-song test set. To prevent contingency, I randomly selected another rock song (*rock.00092.wav*) and ran the same test again. In the second test, the model could still predict the genre in the same test set with high accuracy. Based on the result of two test cases, the three sub-genres were selected in a way that generally matched the musical characteristics of the rock. Then I performed the sub-genre test as designed above. According to the previous description, I performed three tests separately and gain the corresponding results that are shown in the table below.



```
The test is done. Among all the 15 songs, the accuracy is 1.0
```

Figure 4.11: For the first time to use a rock song for training the model and gain the classification accuracy.



```
The test is done. Among all the 15 songs, the accuracy is 0.9333333333333333
```

Figure 4.12: For the second time to use a different rock song for training the model and gain the classification accuracy.

Model	Overall Accuracy		piano-based	vocal group	guitar-based
guitar.Helicopter.wav (guitar-based)	28.6%	Total #	5	5	4
		Correct Prediction	0	0	4
		Accuracy	0%	0%	100%
vocal.ShoutShout.wav (vocal group)	78.6%	Total #	5	4	5
		Correct Prediction	5	1	5
		Accuracy	100%	25%	100%
piano.FuzzyWuzzyHoney.wav (piano-based)	21.4%	Total #	4	5	5
		Correct Prediction	3	0	0
		Accuracy	75%	0%	0%

Chart 4.5: The corresponding results of three types of sub-genre.

4.2 Result Analysis

In practical tests, I found that most of the program's execution time was spent on selecting the 160 initial centroids, especially in computing and finding the new centroid farthest from all existing cluster centers. *K-Means++*, while improving on *K-Means* by reducing the possibility of local optima, comes at a rapidly increasing time cost. For ordinary *K-Means*, this algorithm only requires generating K ($K \leq n$) random numbers and extracting the corresponding initial centroid from the set of all cluster cores according to the index in O(n) time. In contrast, for *K-Means++*, each newly generated initial centroid requires recalculating the distance from the remaining points to all the currently existing centroids and deciding the next initial cluster center that should be selected based on the distance. Therefore, this step requires a time complexity of $O(n^n)$. With the computation of the initial centroid in the project, it can also be found that for the first 100 initial centroids, they are generated very fast due to the small amount of data. And from about 120th centroid onwards the program runs significantly slower. For this project, the total amount of data involved in *MFCC* is not very large. In particular, after training the model for one audio, it is not necessary to perform the same *K-Means* computation again for the same audio file but can read the model-related data directly from the local files. Therefore, I think the time sacrifice is worthwhile for such a small amount of data, and *K-Means++* is also suitable for this project due to this reason.

I also realized that the training set and the corresponding threshold settings were critical in this project. The feedback results from the *4.1.1 Single Genre Test* show that different training models lead to completely different prediction results for the same test set. In the first test case, the program predicted only 13% and 56% of the songs from the two sources correctly, respectively. Especially for the songs from the GTZAN library, this correct rate is not competent for the needs that this program is expected to achieve. However, after simply replacing the comparison model, the correct prediction rate of songs from both sources improved to 100%. This phenomenon reflects two issues. First, the selection of the training set is very important. When choosing the test set, these songs are supposed to contain obvious genre features as much as possible so that the musical information contained in the waveforms can be better translated into feature curves when computing *MFCC*. Since the *MFCC* values of each song in the test set need to be compared with the training model, clustered with the *MFCC* feature points, and calculated the distortion distance, using a song with more genre prevalence as the model can reduce the difference between the songs and further reduce the distortion. From the test, simply replacing the songs in the training set substantially improves the correctness, which proves that the correctness of the program prediction and the model song has a strong connection. Since it is difficult to find a typical song to represent the genre, I think it is also possible to stitch together many music clips with genre features as a whole song into the program for training. This approach ensures that the correctness can be improved without a lot of modifications to the program.

In addition to improving the song samples for training, a better calculation scheme for the classification threshold is needed. The basis for determining and classifying songs genre in this project is the distance between the test song and the model. Only when this distance is less than a certain value will the songs be considered to be in the same category. In the current practice, this value is fixed at 2.35 times the *VQ* distortion obtained during training. My initial idea was that since the distance from the training song's *MFCC* to its centroids should have the smallest distortion. Then all values that satisfy this distortion value within a certain range can be regarded as belonging to the same genre. With several tests executed; I chose 2.35 as the threshold factor. After the single genre test, I also compared the values of the two models, hoping to find a common pattern in them. I found that there is a big difference between the distortion values of the two models. The distortion of the training song *blues.YesMan.wav* is only 4.39, while the distortion of *blues.00055.wav* is 7.36. Therefore, after multiplying the same threshold coefficients, the difference between the two corresponding distortion ranges reaches about 7. In order to reduce the gap between the two, I tried to add 8.5 directly to this value instead of multiplying by a threshold factor when calculating the threshold for the trained model. Afterwards, I re-ran the single genre test with the new calculation rule with the model generated based on *blues.YesMan.wav*. The accuracy was as follows:

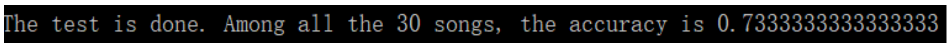

```
The test is done. Among all the 30 songs, the accuracy is 0.7333333333333333
```

Figure 4.13: The results of the third single genre test on GTZAN obtained after changing the threshold calculation rules.

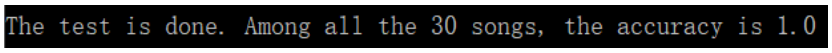
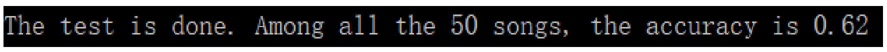

```
The test is done. Among all the 30 songs, the accuracy is 1.0
```

Figure 4.14: The results of the third single genre test on Cloud Music obtained after changing the threshold calculation rules.

As can be seen from the result, a simple change in the threshold setting can also improve the accuracy of the prediction. However, I think that simply adding a parameter may not be very informative for calculating the threshold. If the song itself has a high distortion during training, then adding a smaller number on top of that big value does not make sense either. In addition to simple mathematical operations, I think it might be possible to add a mapping relationship between the distortion value and the corresponding threshold so that the threshold can be set to better satisfy the relationship between the distortion value, the corresponding genre, and the classification threshold. I did not study the specific function of this mapping due to time constraints, but I think this relationship should be roughly satisfied for log images. The criticality of threshold setting can also be seen from the cyclic blues test. The accuracy of the prediction results in this test ranges from 4% to 100%. Among them, 23 songs all reached 100% accuracy when used as training models, and the total number of songs with accuracy above 90% is 47, accounting for almost half of the total number of songs. I re-analyzed the distortion values of the corresponding songs during training and found that the *MFCC* of these songs itself had a higher distortion value for the codebook, so this value was also further expanded after multiplying by the threshold factor. This confirms my thoughts above that the threshold calculation and setting should be changed as the distortion value changes during training, rather than being calculated or set to the same value.

The content and conditions of the second test (mixed-genre test) are closer to the actual application scenario than the first test (single genre test). In the first run, I selected a jazz song as the audio for the training set. What interested me most is that all the songs exceeded the distortion threshold when calculating the *VQ*. This is because the song itself has a distortion value of only 2.6, and the threshold is still quite small after multiplying by the corresponding coefficient. I double-checked the calculation history and find that the distortion values of the songs in the test set were all around 10, which is larger than the threshold. Therefore, all the jazz songs were judged as non-jazz. To figure out the reason, I listened to this training set song and the remaining ten jazz clips and found there were some differences. First, the volume of this song is less than the rest of the songs. Since volume is reflected by amplitude, this would cause the amplitude of this song to appear smaller on the waveform than the other songs. Secondly, this song has a cacophony in the background. I cannot identify if this sound is due to noise or an instrument (such as a cymbal). Although it has been mentioned before that *PCA* somehow reduces the noise of the song, the high volume of this cacophony is likely not to be completely removed. Therefore, this background sound may also be affected the *MFCC* samples and further affect the final result. Finally, the listening feeling of this song is not quite the same as the others. The main instrument of this clip is piano, not sax, drums or even vocals like the rest of the jazz songs. Besides, the melody sounds softer and slower than other songs. As mentioned earlier in this report, this project classified the two songs based on calculating the *VQ* by converting the audio waveforms into *MFCC*. Therefore, a large part of the classification was implemented based on how the songs sound like. This result reflects the importance of setting the threshold and selecting the training set. What I think this also reflects is the result of a bad selection of training models. When the songs in the training set are somewhat different from those in the test set, the songs in the test set are also not well predicted. Therefore, the selection of the training set for the model should consider the adaptability of the songs, or even splice together several different song fragments for the training of the model. Please refer to *5.2.2 Bigger dataset* for improvement details on the training set.

Based on the above question about threshold setting, I experimented again using the same model (*jazz.ComeRainOrComeShine.npy*) as in the first test. However, like before, I added 8.5 to the calculated threshold based on the *VQ* distortion and obtained the following results:



```
The test is done. Among all the 50 songs, the accuracy is 0.62
```

Figure 4.15: Result obtained after changing the threshold calculation rules.

	Classical	Country	Disco	Jazz	Reggae
Total #	10	10	10	10	10
Correct Prediction	1	8	9	4	9
Accuracy	10%	80%	90%	40%	90%

Chart 4.6: The accuracy of each genre with a jazz model and a different threshold calculation rule.

During the execution of the mixed-genre test, I also found that the distortion values for classical music were generally low, with an average value of around 10. By listening to each song, I found that the songs were more soothing or with low volume. But even when compared with fast-paced songs like disco, their distortions remained in the same range. So, I deduced that this could be due to the softer style of the song. However, I went on to test blues and jazz, which are also soothing, on the model but for these songs the distortions are higher than classical music. Thus, this is not entirely due to the speed of the beat or the strength of the rhythm. It is impossible to determine the exact cause, but based on the test audios, I inferred this result is a combination of the backing instrument, the volume level, and the characteristics of the genre itself. I guess that these factors affect the *MFCC* and further lead to a smaller distortion value in the *VQ* calculation. For example, in a lot of testing samples of classical music, there are segments where the violin plays along with the rhythm (similar to the strong or rhythmic beat in disco) and segments that unfold slowly from silence. This may result in classical music that in fact combines many different genre features, for example, a single song that contains clips that are fast and slow, strong and quiet, and a combination of instruments. The program begins with the conversion of the audio into *MFCC* feature points, and these combined features more or less affect the *MFCC*, making the eigenvalues more generalized. That is, these songs' *MFCC* is equivalent to a fusion of many different genres, leading to a smaller distortion value for all genre models when classifying.

Finally, in the sub-genre test, I think the program did not do well in classifying. While it was more accurate in classifying general genre (i.e., rock), it was somewhat less accurate in classifying sub-genre. I recognize that this is due to a problem with how the program is executed. The program does not process the details like many deep learning algorithms do, but rather performs distance measurement and classification based on the general waveform outline. I think this is one of the shortcomings of this program that is not being able to do a fine delineation. But as I mentioned before, the genre itself is a very ambiguous concept so that distinguishing sub-genre is not very meaningful in real life, and corresponding application scenarios are rare. The sub-genre test itself is more of a way to explore the "limits" of the project rather than to study its capabilities in practical applications. However, I believe that it is still possible to achieve some degree of sub-genre classification by iteratively optimizing the model and adjusting the corresponding classification parameters (such as the total number of clusters of *K-Means* and the *VQ* threshold). However, due to the limited application, I did not make further tests and improvements here.

In conclusion, I am satisfied with the overall performance of the program in terms of these test results. When faced with a single genre, the program was able to perform the classification task well. Although the accuracy of the model selection was occasionally low, it still achieved an accuracy rate of over 70% in about 70 test cases. In addition, training and testing the model using randomly selected songs from the Internet also demonstrated that the program is equally applicable to everyday real-world projects, rather than only obtaining a theoretically based reference value by executing well-processed data (e.g., all data are processed into a representative presentation format). However, this series of tests has likewise revealed some shortcomings of the program, that is, the program currently needs to be strengthened in its handling of test sets with mixed song styles. One of the most critical issues is how to set reasonable thresholds. The selection of thresholds is the basis for distance calculation and comparison by the program. The inaccurate results in the test are not all caused by problems in the *VQ* calculation during the test, but by obtaining a small distortion value during the model training, which leads to an unreasonable choice of the final threshold value. For example, in a mixed-genre test based on jazz as the model, there may be a situation where the jazz in the test sets have distortions that are in fact generally lower than the other genre. However, the selection in threshold leads to the classification result that all songs are considered as not belonging to jazz. The threshold setting may also have something to do with the genre itself. As I mentioned above, the distortion of classical is always low, no matter what genre of song is used as the model. Although the actual cause of this phenomenon is unknown for now, it may suggest a problem that various genres and corresponding features need to be taken into account when calculating the threshold. In future improvements, the relationship between the genre itself, the training set data and model, and the threshold setting should be considered. In this way, the accuracy of classification is expected to be improved.

5. Applicable Fields and Optimizations

5.1 Possible Applications

This project implements the function of classifying the music according to genres and acoustic feelings. This project is expected to be functional daily because its training data and test sets all come from daily life.

As stated at the beginning of this report, one of the motivations for this project is to better serve the recommendation system by enabling them to recommend content to users that match their preferences through potential content features and tags. Some of today's video and music sites tag uploaded content based on the category and description of the uploaders. In Sound Cloud, users are asked to select tags that match their genre in a given selection set for each upload. these tags are used to assist in categorizing songs and recommending them to users through the appropriate algorithms. However, these tags become useless when the user does not classify his songs correctly (it could be the situations like the user does not know the classification between different genres and incorrectly classifies his music in another genre). This is because a wrong tag not only does not help the recommendation system make the right judgment but also may degrade the user experience by pushing the wrong songs. I have encountered a similar situation with KuGou Music (again, users can select the genre and keywords of their work while uploading). It is undeniable that such a situation will affect the user's approval of the product. It is undeniable that such a situation will affect the user's recognition of the product. This program can be used to review and categorize these music websites and music players rather than relying on users to categorize the genre of music works while the program will save labor costs in such music sites and players compared to manual review. Another advantage is that these music sites often contain a large number of works, which helps in the preparation of the training set. Among them, the songs that have been identified will be involved in the training of the classification model as labelled data. This project has a small sample requirement for training and comparison, which may lead to bias in classification and prediction (see *5.2.2 Bigger dataset*). Musical collections such as music websites and music players will be able to provide a larger total number of samples and sample sets for training, which will also improve the efficiency and representative of model training and strengthen the training results.

In addition to this approach of feeding the recommendation algorithm through genre tags, this project can also aurally implement a recommendation system. The *MFCC*, which was introduced above, is a value that reflects the direction of a song's spectral features and can be used to simply summarize the auditory characteristics of a song. The program extracts the *MFCC* of each song as the feature value and thus obtains the auditory characteristics of the song, i.e., how a song sounds like, from this perspective.

Thus, as part of the recommendation system, the system can use the song that the user is currently listening to as the input, i.e., the training sample of the model, and use this value as a criterion to check the *MFCC* of other songs in the database. After that, the system selects potential recommended songs by calculating the difference between the feature values of the songs in the database and the currently playing music and uses the songs with feature value difference less than the threshold as the output of the system for a personalized recommendation.

Besides, the project can also help with the development and testing of small projects. This is because this program does not require a lot of tagging data and samples to train, as many large machine learning frameworks do. Instead, it uses the eigenvalues of a song as a benchmark to find all similar songs. From a small project perspective, this classification can reduce development and maintenance costs, as a large amount of labelled data usually can only be done manually. Therefore, the acquisition of such data often requires a high cost of time and resources. This project has somewhat reduced the need for these annotated data and thus will improve the efficiency of development and maintenance in small projects, such as the automatic classifiers of mobile phone audio and even can be used in machine-learning-based academic assignments.

Finally, this project can be used for data pre-processing prior to model training during audio-related deep learning. Take Natural Language Processing (NLP) as an example, the generation of language models relies heavily on the labelling of the training dataset, and training with a better dataset can increase the prediction accuracy and improve the training result of the model. Therefore, in the data pre-processing stage, this program can be run first for data cleaning and data integration. By simply identifying the audio content of the data and eliminating outliers such as noise-only samples, the training sample set can be more representative. For example, the beep sound that indicates the call is not answered can be used as the input sample in a phone call recognition project to quickly find all the recordings of calls that are not answered in the training set and remove them, ensuring that the remaining samples are meaningful and contain speech that can be used for training. In addition to data cleaning, this project can also be used to classify different usage scenarios in the pre-processing stage. For example, smart audio corresponds to different parameters in noisy and quiet environments. Using this project to classify the audio scenes before training the model can improve the training effect of the training samples and make the model more proficient in a specific application scenario. Due to the waveform for the quiet and noisy environment being different, it is easy to use *MFCC* to extract the environmental features. The reason why this project can also be extended to audio-related other sounds processing like NLP instead of being limited to music-related applications is that the algorithms involved in this project, especially *MFCC* for audio feature extraction, are not designed for music processing only; *MFCC* is also widely used in speech recognition and semantic analysis. Therefore, this project is equally applicable to NLP-related projects in general.

5.2 Future Improvements

Due to the data and time limits, there is not enough labelled dataset for me to realize the following improvements. However, I think the project can be optimized from the three aspects to gain a better classification result.

5.2.1 Align with the musical features

Genres are not only defined by the frequency spectrum but also involve emotional analysis and instrumental arrangements. Take blues as an example, this kind of music conveys the oppression faced by African American people with their emotions and feelings of life through the unique rhythm and lyrics. Thus, the word blues also indicates the sorrowful emotion among the African American people in the 1960s. Based on this background, it can be referred to the idea that blues is often associated with slow and lyric melodies while a song with fast and rapid beats is not likely to be blues. This involves an emotional assessment. Similar to sentiment analysis that is commonly used in NLP, this method requires the training with massive, labelled data, and predicts the emotional tendency of the input song based on the trained model.

Similar to the emotion analysis, the arrangement of beats and instruments is also a key feature to determine the genre characteristics. Specifically, rock and roll, especially heavy metal rock, often accompanied by drums and distorted electric guitars, emphasizes a wild feeling in the vocal part. Although hip-hop music also has strong beats and drums, it often incorporates a lot of electronic sounds and synthesizers effects, and the performance style is not as wild as rock music either. Both classical music and blues use the piano as an instrument, but classical music also incorporates strings and wind ensembles, while blues usually combines harmonica and guitar with a special shuffle rhythm. Although the definition of different genres is vague, a song can still be analyzed based on musical characteristics, which function as a complement to *MFCC* prediction.

Generally, this improvement amounts to the increasement of a predictive Bayesian classification algorithm at the end of the project. The above eigenvalues with musical features will be combined and cross-compared with the songs' *MFCC* values. Thus, this is not inconsistent with what I mentioned earlier about using musical features for classification. Considering specificity, for example, not only does blues music sound sad, but most blues have a shuffle rhythm, each feature and value will be assigned a different weight (i.e., the percentage in Bayesian classification). This improvement reduces the weight of *MFCC* in the whole process and combines different eigenvalues thus is expected to have a better classification effect. It is worth noting that the extraction algorithm of these musical features should not be used unconditionally but determined according to the situation. Otherwise, redundant features may reduce the operational efficiency of the whole project by underfitting (That is, multifarious input data and parameters cause the model to fail to capture the most critical features).

5.2.2 Bigger dataset

Currently, the project only needs one song as input for model training, and genres are differentiated according to this model. This excessively small sample of the training data set facilitates the development and testing of small projects to some extent (see *5.1 Possible Applications*) but may lead to the problem of overfitting. Overfitting refers to that the model incorrectly learns irrelevant data during training, or the amount of data is too small, resulting in poor generalization ability of the model and can only deal with the data given during the training. Such models usually perform well in training sets, but often fail to give accurate predictions in test sets. Figure 5.1 shows the difference between a normal model and an overfitting model. It is easy to find that every outlier is fully considered in the green line, which presents a lot of unnecessary twists that make its graph looks abnormal. While the blue line only calculates a regression equation, which is more likely to have a better generalization ability for the data.

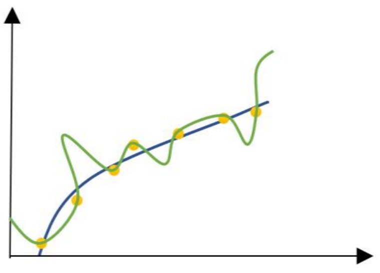


Figure 5.1: Comparison between a normal model and an overfitting model. The yellow dots indicate the samples, the green line indicates an overfitting model, and the blue line indicates a normal model.

In terms of this project, because the model summarizes a small amount of data, it is easy to fail to correctly classify different music pieces with the same genre. For instance, blues includes piano-based blues and guitar-based blues. However, selecting only piano-based blues as the sample for training may result in the program not being able to correctly deal with guitar-based blues songs, where a "bias" against genre arises. Therefore, overfitting can be reduced by increasing the proportion of training data to total data in future improvement, and the accuracy of program prediction can be improved further. It should be noted that the data set should not be too large either, or underfitting will occur.

5.2.3 Parameter Optimization

Due to time and resource constraints, some parameters of the project were only optimized briefly. Thus, fine-tuning the parameters is a significant process in the future to increase classification accuracy. For example, the program fixes the number of clusters of the *K-Means++* algorithm currently with the output to be exactly one-hundred-and-sixty eight-dimensional points. Since the nature and characteristics of different genres are not the same, these two parameters can only be used as a rough reference. In the future, these two parameters can be adjusted by testing with multiple test sets, to ensure that the algorithm's correctness while without losing or losing less information. Another parameter that should be considered is the threshold for the final *VQ* classification. The current threshold was tested in the project manually. Due to the small number of the training sets, the selection of this threshold may be haphazard and may make sense to the existing test sets only. Nevertheless, this value is the most important parameter used to determine the genre of a given song. In the later optimization, this parameter should be taken into account, and even a separate classification standard value should be established for different music styles to cater to the genre specificities.

6. Summary

To sum up, this project achieves reading, converting, and classifying untagged audio files by combining different algorithms. Thus, it meets my expectations for the project. As one of the motivations for the project, I wanted to be able to classify uncategorized or unlabeled songs based on the genre through an algorithm. Further, the recommendation algorithm can optimize the recommendation system by making personalized content based on such classified genre tags. In this project, I have used *MFCC*, *PCA*, *K-Means++*, and *VQ* to complete the process. The input songs to be classified will be compared by the distance between *MFCC* features to determine whether they are consistent with the current given standard song's genre and based on this, the songs will be simply identified and assigned different genre labels. As can be seen through the test, this project still has a lot of room for improvement due to time and resource constraints. For example, the accuracy of song recognition and the generation format of the final results. After graduation, I look forward to having the time to continue maintaining this project, so that it can be applied to real-world large-scale projects.

I have also gained a lot from building the whole project. First of all, the main algorithms involved in the project were all new to me, which not only expanded my knowledge base but also gave me a better understanding of how these algorithms are used in practice. Second, the process of finding and learning these algorithms also taught me to analyze an algorithm from scratch and learn to judge how to choose the right algorithm and apply it to the project. I have also seen how the knowledge I have learned over the years can be applied in the implementation of algorithms. Maybe like many students, I probably started out thinking that linear algebra, which I studied in my freshman year, was a subject that I hardly ever used. However, this project applied the concepts related to matrices in linear algebra. Finally, this project re-affirmed my interest in data science, especially big data management. I had taken a class related to artificial intelligence, but this project made me realize more clearly what is interesting about AI. Therefore, I would like to say that this project has more or less influenced my planning for future research and development directions, as I wish to have more chances to probe into the field of data science in the future. At first, I thought the honours project was just a simple summary of what I had learned in the past few years, but now I realize that this project is not only a test of my previous knowledge but also an improvement of my abilities. At the same time, the development of this Python-based classifier as my graduation design eliminated the gap between my academic knowledge and practical application and made me feel confident about my performance after graduation.

Appendix A

Terms listed in the alphabet order.

1. *K-Means* / *K-Means++*: A method of clustering based on the classification of similar points at close distances.
2. *Principal Component Analysis* / *PCA*: A method of dimensionality reduction for data.
3. *Mel-scale Frequency Cepstral Coefficients* / *MFCC*: A feature describing the short-time power spectrum envelope that can be used to characterize audio files.
4. *Vector Quantization* / *VQ*: A lossy data compression method based on block coding rules.

Appendix B

The chart for the test result. All the test cases are executed based on the same test set.

Model	**Overall Accuracy**		**Classical**	**Country**	**Disco**	**Jazz**	**Reggae**
blues.00001.wav	94%	Total #	10	10	10	10	10
		Correct Prediction	10	10	10	7	10
		Accuracy	100%	100%	100%	70%	100%
jazz.ComeRainOrComeShine.wav	80%	Total #	10	10	10	10	10
		Correct Prediction	10	10	10	0	10
		Accuracy	100%	100%	100%	0%	100%
jazz.ComeRainOrComeShine.wav (with a different rule to calculate the threshold)	62%	Total #	10	10	10	10	10
		Correct Prediction	1	8	9	4	9
		Accuracy	10%	80%	90%	40%	90%
dataset\training\jazz.00019.wav	64%	Total #	10	10	10	10	10
		Correct Prediction	1	8	9	5	9
		Accuracy	10%	80%	90%	50%	90%
dataset\training\disco.00063.wav	32%	Total #	10	10	10	10	10
		Correct Prediction	0	3	6	2	5
		Accuracy	0%	30%	60%	20%	50%

References

Listed in alphabet order

1. Boxler, D., (2020). *Machine Learning Techniques Applied to Musical Genre Recognition*. (Master’s thesis). Retrieved from Carleton University MacOdrum Library.
2. Fayek, H., (2016). *Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between*. Retrieved from: https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html
3. Giraud, C., (2014). *Introduction to High-Dimensional Statistics*. CRC Press.
4. Kapoor, A., and Singhal, A., (2017). *A comparative study of K-Means, K-Means++ and Fuzzy C-Means clustering algorithms*, International Conference on Computational Intelligence & Communication Technology (CICT), pp. 1-6, doi: 10.1109/CIACT.2017.7977272.
5. KevinLu10, (2020). *Recognize Speech Music*, GitHub. Retrieved from: https://github.com/KevinLu10/recognize_speech_music
6. Lee, C., (2015, March 4). *A Detailed Explanation of Principal Component Analysis (PCA)*. [Web log post]. Retrieved from: https://blog.csdn.net/zhongkelee/article/details/44064401
7. Makhoul, J., Roucos S., and Gish, H., (1985). *Vector Quantization in Speech Coding*, Proceedings of the IEEE, vol. 73, no. 11, pp. 1551-1588, doi: 10.1109/PROC.1985.13340.
8. Pluskid, (2009, January 13). *Ramblings about Clustering: Vector Quantization*. [Web log post]. Retrieved from: https://blog.pluskid.org/?p=57
9. Prahallad, K., (n. d.). *Spectrogram, Cepstrum and Mel-Frequency Analysis* [PDF]. Retrieved from: http://www.speech.cs.cmu.edu/15-492/slides/03_mfcc.pdf
10. Qing, Y., (2016 January 19). *K-Means Clustering Algorithm: Algorithmic Ideas*. [Web log post]. Retrieved from: https://zhuanlan.zhihu.com/p/20432322
11. Wagstaff, K., Cardie, C., Rogers, S., & Schrödl, S. (2001). *Constrained K-Means clustering with background knowledge*, ICML. vol. 1, pp. 577-584. Retrieved from: https://web.cse.msu.edu/~cse802/notes/ConstrainedKmeans.pdf
12. Xu, Y., (2018). *Research and Application of Music Classification Based on Convolutional Neural Network*. (Master’s thesis). Retrieved from: CNKI database.
13. Yoğurtcuoğlu, S., and Çokça, Ö., (2020). *Music Genre Classification and Clustering*. Retrieved from: https://github.com/ozgecokca/