目录

[多基音检测 3](#_Toc466884362)

[相关研究 3](#_Toc466884363)

[相关技术 3](#_Toc466884364)

[信号频谱分析 3](#_Toc466884365)

[基于非负矩阵分解的多基音检测算法 3](#_Toc466884366)

[多基音检测系统实现 4](#_Toc466884367)

[频谱计算模块实现 4](#_Toc466884368)

[非负矩阵分解模块实现 5](#_Toc466884369)

[系统评价 7](#_Toc466884370)

[算法评测 7](#_Toc466884371)

[系统评测 8](#_Toc466884372)

[乐谱跟踪 9](#_Toc466884373)

[乐谱跟踪算法流程 9](#_Toc466884374)

[Landmark 9](#_Toc466884375)

[指纹制作 9](#_Toc466884376)

[检索指纹 10](#_Toc466884377)

[筛选结果 11](#_Toc466884378)

[算法实现 11](#_Toc466884379)

[制作指纹实现 11](#_Toc466884380)

[结果筛选实现 13](#_Toc466884381)

[算法评测 14](#_Toc466884382)

[测试数据 14](#_Toc466884383)

[测试方法与指标 14](#_Toc466884384)

[测试平台 14](#_Toc466884385)

[测试结果 14](#_Toc466884386)

[总结与展望 15](#_Toc466884387)

[基于Android的钢琴演奏评价系统 16](#_Toc466884388)

[相关技术 16](#_Toc466884389)

[MusicXML文件格式 16](#_Toc466884390)

[系统设计 17](#_Toc466884391)

[系统实现 17](#_Toc466884392)

[开发及测试环境 17](#_Toc466884393)

[MusicXML处理模块实现 18](#_Toc466884394)

[乐谱显示模块实现 20](#_Toc466884395)

[乐谱播放模块实现 23](#_Toc466884396)

[录音模块实现 25](#_Toc466884397)

[录音数据解析模块实现 27](#_Toc466884398)

[系统评测 28](#_Toc466884399)

[测试环境 28](#_Toc466884400)

[测试结果 28](#_Toc466884401)

[总结与展望 28](#_Toc466884402)

[乐器识别 30](#_Toc466884403)

[相关技术 30](#_Toc466884404)

[支持向量机 30](#_Toc466884405)

[算法流程 31](#_Toc466884406)

[预处理 31](#_Toc466884407)

[特征提取 32](#_Toc466884408)

[训练和预测 32](#_Toc466884409)

[算法评测 32](#_Toc466884410)

多基音检测

多基音检测即检测声音信号中同时存在的多个音调，包括多个声源同时发声和单一声源同时产生多个音调两种情况。多基音检测被广泛应用于自动音乐转录、声源分离、旋律提取等研究领域。我们的研究内容为钢琴演奏中的实时多基音检测。

相关技术

基于非负矩阵分解的多基音检测算法

我们基于非负矩阵分解（Non-negative Matrix Factorization，NMF）算法实现钢琴演奏中的实时多基音检测。

NMF是一种信号处理技术，它将给定的非负矩阵V近似地表示为两个非负矩阵W和H的乘积，



其中，V的每一列表示音频的一个短时频谱，W的每一列表示一个音符的频谱，H的每一行则表示检测到的对应音符随时间变化的activation。

通过对H进一步处理，则可以得到多基音检测结果。可以使用支持向量机（Support Vector Machines，SVM）或隐马尔可夫模型（Hidden Markov Model，HMM），然而，大多数方法采用的是简单的阈值化技术。如果某一音符某时刻的activation高于阈值则认为该音符该时刻被演奏了，否则认为该音符该时刻没有被演奏。如果一段时间内，某一音符的activation由低于阈值变为高于阈值，则认为该时刻该音符对应的钢琴键刚被按下；如果activation由高于阈值变为低于阈值，则认为该时刻该音符对应的钢琴键刚被放开。阈值可以根据经验或手动设置。

多基音检测算法实现

我们实现的多基音检测算法的流程如图1所示：



图1 多基音检测算法流程图

首先，需要提取单个音符的频谱模板。对于只包含一个音符的音频，首先进行有无声区判断，检测出其中音符被演奏的时间段，去掉没有声音的开始和结束部分。然后采用NMF算法，提取出能代表该音符的频谱的模板。

将各个音符的频谱模板拼接起来，表示为W。对于实时输入的钢琴演奏的声音信号，对一帧采样数据进行短时傅里叶变换得到其振幅谱，表示为V。采用NMF算法，则可以计算出表示各音符activation的H。进一步进行阈值化处理，则可以得到多基音检测结果。

频谱计算实现

我们采用的时频表示为短时傅里叶变换得到的振幅谱。音频采样频率为44100Hz，帧长为4096个采样点，帧移为441个采样点，即每10ms取一帧采样数据，窗函数为Hamming窗，帧长为偶数时窗函数满足DFT偶对称，FFT长度为4096个采样点。由于振幅谱的对称性，只保留振幅谱的前半段。

FFT为数字信号处理中的常用方法，有比较成熟、高效的开源库实现。C语言中，应用较为广泛的开源FFT库有MIT开发的FFTW库[[1]](#footnote-1)、Kiss FFT库[[2]](#footnote-2)等。

FFTW（the Fastest Fourier Transform in the West）是一个快速计算DFT的标准C语言库。由MIT开发，完全开源免费。支持任意大小、任意维数数据的DFT计算，并且还支持离散余弦变换（DCT）、离散正弦变换（DST）、离散哈特莱变换（DHT），应用范围广。FFTW通常比目前其他开源DFT程序都要快，被称为是世界上最快的FFT。但是，FFTW库比较庞大，结构复杂，使用时配置过程比较繁琐，而且对跨平台移植的支持不是很好。

Kiss FFT库具有效率高、结构简单、支持跨平台移植等优点。

我们对FFTW库和Kiss FFT库进行了对比，如表1所示。采用的测试环境为Intel i5-4590 + 8G RAM + Win 10 + VS2015，测试数据为。综合考虑对比结果，我们选用Kiss FFT库来实现FFT，不仅能够显著提升FFT的计算效率，而且便于后期将系统导出为底层库移植到移动平台上。

表 1 FFT实现对比

|  |  |  |
| --- | --- | --- |
|  | FFTW | Kiss FFT |
| 对一帧采样数据进行FFT的平均耗时 | 1ms | 1~2 ms |
| 大小 | 7.8M | 一个.c文件和两个.h文件 |
| 使用方法 | 需要配置.dll和.lib文件的目录 | 把库文件添加到工程中即可 |
| 跨平台支持 | 在Linux平台有相应的版本，跨平台移植要考虑.dll和.lib文件的兼容性 | 随工程一起编译，便于导出。 |

非负矩阵分解实现

根据算法原理，通过已知的频谱模板信息，能够把音频的幅度谱分解。（Data = W \* H，已知Data和W来得到H。）分解之后经过叠加处理得到长度为88的数组H，将数组H中的值与阈值相比较，如果大于阈值的话表示相应位置的乐符被演奏，如果小于阈值的话表示相应位置的乐符没有被演奏。非负矩阵分解函数的声明如下：

/\*\*

\* @brief 非负矩阵分解函数nmf\_beta（data = W \* H，已知data和W来得到H）.

\* @param 输入参数data：要分解的矩阵

\* @param 输入参数W：矩阵分解的频谱模板

\* @param 输入参数H：矩阵分解的结果

\* @param 输入参数H0Flag：是否有H0输入，1为有H0，0为没有H0

\* @param输入参数 H0：H0数据

\*/

void nmf\_beta(double\* data, int rows, int column, double\* W, double\* H, int H0Flag, double\* H0, double beta, int maxiter);

在频谱分解算法计算开始之前，要设定初始的H数组：H0，当多基音检测算法独立运行的时候，数组H0的值采用随机初始化的方式来设定；当有了语音指纹算法的辅助之后，语音指纹算法会输出时间信息。编写computeH0函数来根据输出的时间信息以及标准乐谱的midi来计算初始H0数组。根据乐谱跟踪结果计算H0的函数声明如下：

/\*\*

\* @brief 根据给定的时间信息，查找midi文件来计算出H值.

\* @param 输入参数midiData midi信息

\* @param 输入参数Tscore 输入的时间信息.

\* @param H0 返回的计算出的H0值.

\*/

void computeH0(double\* H0, double\* midiData, int midiLen, int templateNum, double Tscore)

通过频谱分解函数得到H数组之后，将H数组中有多套模板的对应值相叠加，得到长度为88的H数组，数组的下标对应钢琴的88个乐符，将H数组每个值与阈值相比较，如果该数值大于阈值，则表示该数组下标对应的钢琴乐符被演奏。

在整个系统中，频谱分解模块是耗时最长的模块，基本占整个系统耗时的90%。

表 2 NMF算法耗时

|  |  |
| --- | --- |
| 测试数据： | MAPS\_MUS-bk\_xmas1\_ENSTDkAm.wav 总共3001帧 |
| 模板数据： | template.dat Double型2049\*265 |
| 平均每帧多基音检测算法耗时： | 77.992s / 3001帧 = 25.988 ms/帧 |
| nmf平均耗时： | 24ms |
| nmf占多基音检测算法耗时比重： | 92.3% |

频谱模板是一个浮点型的二维数组，数组的行列数都比较大，作为测试的模板数据行列为2049\*265。在频谱分解计算的过程中，涉及到多次对频谱模板的遍历操作，每次遍历整个数组都会花费很多的耗时。频谱分解算法涉及还到大量的矩阵相乘运算，耗时会更加的长。大量的浮点型数据的矩阵运算，是频谱分解算法主要的耗时点。

在系统的开发过程中，尝试了几种优化频谱分解耗时的方法，主要的优化思路在减少矩阵相乘耗时上。首先尝试将矩阵按列分解成多个子矩阵，然后使用跨平台开源线程库pthread库[[3]](#footnote-3)，开辟多个线程分别计算各个子矩阵与另一个矩阵相乘。但是考虑到系统开辟线程以及关闭线程需要时间，多线程对矩阵相乘运算的优化效果不明显；接着尝试使用了开源的矩阵运算库open-cblas[[4]](#footnote-4)，使用open-cblas库来做矩阵相乘运算。开辟多线程频谱分解耗时以及open-cblas库频谱分解耗时对比如表3所示。采用的测试环境为Intel i5-4590 + 8G RAM + Win 10 + VS2015。

表 3 矩阵运算实现耗时对比

|  |  |  |  |
| --- | --- | --- | --- |
|  | 未做处理 | pthread | open-cblas |
| 平均每帧耗时（ms） | 25.988 | 51.249 | 28.290 |

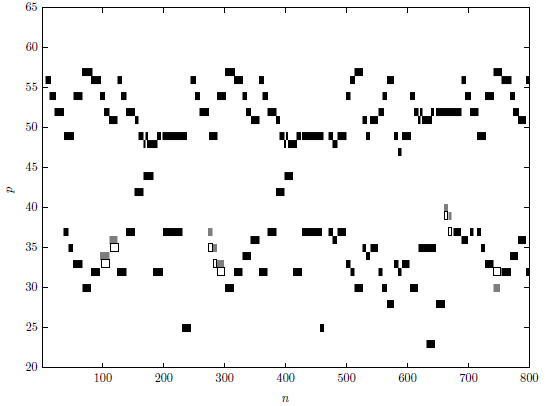
上述的几种方式都没有能够优化频谱分解模块的耗时，目前还在寻找新的思路来优化。

系统评价

算法评测

选用的数据集为MAPS (ENSTDkCl)，一个通用的钢琴多基音检测及音乐自动转录的数据集。每10ms进行一次基音检测。同时，约束一个音符的最短时长为60ms。对于检测到的音符，若其被按下的时刻与正确结果的偏差在±100ms内，则认为检测正确。

一个音乐片段的多基音检测结果如下图所示（横坐标为帧序号，每一帧为10ms，纵坐标为钢琴键的序号），其中，黑色表示检测正确了的音符，灰色表示没有检测到的音符或多检测到的音符。



多基音检测系统通用的评价标准之一为note level F-measure，计算方法如下

其中，nCorr表示检测正确的音符总数，nRef表示正确结果中的音符总数，nTot表示检测到的音符总数。

不参考乐谱信息时，我们实现的系统的note level F-measure为76.2%。

|  |  |  |  |
| --- | --- | --- | --- |
| 乐谱中的音符总数 | 检测到的音符总数 | 检测正确的音符总数 | F-measure |
| 8331 | 7367 | 5981 | 76.20% |

已知音频中演奏的乐曲对应的乐谱时，可以根据当前时刻应该被演奏的音符初始化H，从而使因式分解过程更快地收敛，更好地近似被分解的信号频谱。参考乐谱时，note level F-measure为82.4%。

|  |  |  |  |
| --- | --- | --- | --- |
| 乐谱中的音符总数 | 检测到的音符总数 | 检测正确的音符总数 | F-measure |
| 8331 | 6836 | 6249 | 82.40% |

实际应用中，为了知道当前时刻乐谱中应该被演奏的音符，需要乐谱跟踪技术。乐谱跟踪是实时检测当前演奏的内容对应于乐谱中的时刻的技术。我们采用音频指纹技术实现。可以借助乐谱跟踪技术获得的时间，以当前时刻乐谱中应该被演奏的音符初始化H，得到更准确的基音检测结果。

系统评测

多基音检测算法运行时间测试结果如表4所示。测试音频为MAPS\_MUS-bk\_xmas1\_ENSTDkAm.wav，总共3001帧。模板数据大小为2049\*265。采用的测试环境为Intel i5-4590 + 8G RAM + Win 10 64位。

表 4 多基音检测算法运行时间

|  |  |
| --- | --- |
| 测试平台 | 平均转录时间（ms/帧） |
| VS2013 | 54.7 |
| VS2015 | 25.988 |
| MATLAB | 22.9 |

乐谱跟踪

乐谱跟踪算法流程

乐谱跟踪算法是基于音乐指纹算法实现的，算法的输入为一段演奏音频和一段标准的乐谱音频，输出为待测音频出现在乐谱音频中的位置.算法的流程如下：

Landmark

经FFT之后的音频信号有大量的信息，考虑到频域中能量极大值点具有很强的抗干扰能力，这些能量极大值点称为Landmark。Landmark实际是抽取出每一时刻中有最大振幅的频率对应的点，综合了时域和频域信息，具有较强的代表性，这种抽取特征点的方法也有效地提高了算法的效率。

指纹制作

计算了音频所有的Landmark之后，会得到一个类似于星空分布的图，记录着每一时刻有着极大幅值的点对应的频率值，即一系列时间频率点对。对于每一个点，在它之后划定一块区域，区域内的每一个点与这个点一起构成了一个“指纹”，代表着音频中独特的索引值。

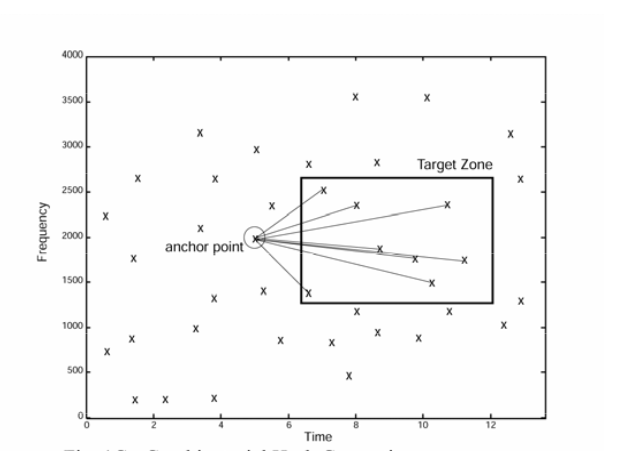


图 2‑2 指纹构造图

每一个指纹都记录着这两个点的信息，如图2-2所示，该指纹记录的信息是一个key-value键值对，t1时刻对应的频率f1、t2时刻对应的频率f2、两个点的时间差t2 – t1构成key，第一个点的时间t1为value。Key是一首歌曲的hash值，相似度很高的歌曲，它们能够匹配相等的hash值也越多。Value记录了指纹的当前时间信息，能够在匹配的时候给出关于匹配定位的信息。

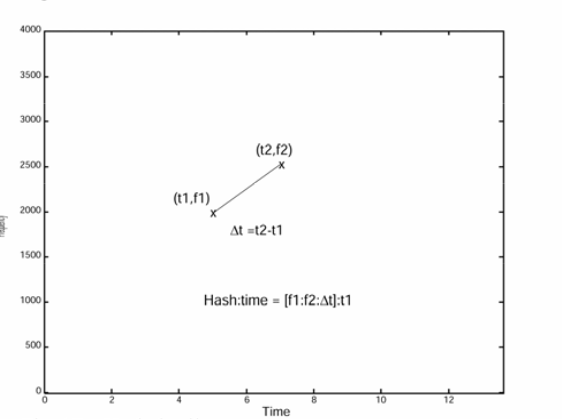


图 2‑3 一条指纹记录的信息

检索指纹

将制作的指纹在指纹数据库中检索，根据key值进行匹配，将匹配到的指纹对应的出现时间与当前指纹的出现时间相减，即记录，表示样本音频在匹配到的音频中的位置。检索完毕后，会出现很多，将这些排序，选择最大的即为匹配到的时间点。

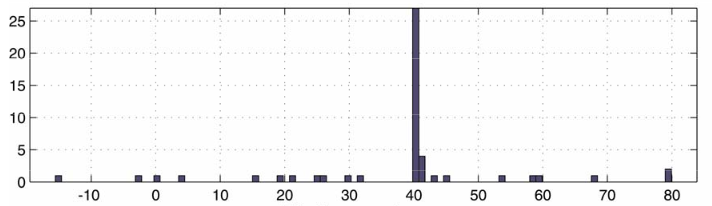


图2-4 匹配时间集合

筛选结果

在实际情况中，一段演奏音频会在多个位置有较高的匹配度，利用演奏的相对连续性，设置一定的筛选策略可以提高挑选的正确率.筛选需要考虑到多处匹配、连续演奏、跳跃等情况，需要合理利用历史匹配结果，又不能太依赖历史结果。



算法实现

制作指纹实现

制作指纹模块的核心是构造出一系列Landmark点。对于每一个WAV音频，首先采样读取并解析其信息，采样时间为10ms。将这个采样到的信号进行FFT变换，得到频域信息。选出频谱中两个极大能量值点并记录其时域的时间值，抽取两个点这样可以在一定程度上抗干扰。对于每一个时间频率点对，划定一个固定大小的窗口，每一个时间频率点对与其窗口中的每一个点构成了一个Landmark点，即一条指纹，对于某一个点a，及a对应的窗内的各点b，计算，其中, 分别为a, b所在帧对应的时间，分别为a, b对应的频率。Key值代表着音乐在该时刻的某种特征，但不是第一无二的特征。即使是在同一首音乐中，也会有很多时刻有着相同的key。在存储这些指纹的时候，是将key和一个存放value的集合，构造成一对键值对。由于数据库中有很多音频文件，指纹中还记录了文件名信息，一起放到集合中。最后，将所有的指纹存储到一个hashmap中。

 C语言中没有封装好的hashMap结构，使用链表可以构造出类似的结构，实现如下：

typedef struct HashNode {

int key;

dynamicArray\* value;

struct HashNode\* next; //相同hash取下一个节点

} HashNode;

typedef struct {

int size;

int item\_size;

HashNode\* head;

} HashTable;

Key值的存储：key的构成是，如果用字符串来表示的话，需要调用大量的字符串拼接函数，内存开辟空间也很大。考虑到频率和时间的都是较小的整数值，可以用位来表示，将32位的int拆分成三部分，分别存储，这样既可以减少耗时，又可以减少内存开销。

存储的实现过程：对于每一个key，通过hash算法映射到地址，遇到冲突就在同一位置用链表的方式链接。所以最后的存储形式如下：



在存储指纹的过程中，将key值映射到地址，在检索指纹的时候，就可以直接用地址查询到，当hash算法比较合理，可以大致均匀地映射到连续的地址时，指纹检索的复杂度就是O(1)。

结果筛选实现

1. 统计并排序。统计每一个时间点匹配到的次数k，将统计结果按照大小顺序排列。
2. 合并结果。记步骤一中得到的结果为 ，找出频数统计最大值max(k)对应的（），得到合并后的频数。对外的重复以上步骤，依次找到其他的合并匹配点。合并后，中的最小间隔为300ms。
3. 归一化。对于各个合并后的对应的新频数除以最大频数max(k)。归一化之后的值即为每一个时间点的权值，范围在0~1。经大量测试发现，正确的匹配结果的权值大于0.5，所以选取出所有k>0.5对应的。
4. 设置一个可伸缩的区间。原则是区间可以很大，但要包括正确的匹配位置，然后在这个范围中筛选候选值，计算候选值的匹配度，匹配度由上一次匹配结果和本次结果加权计算。设置一个偏差阈值，用来衡量当前结果和上一次结果的偏差，当偏差较大时，区间就扩大，当偏差较小时，区间就缩小。

算法评测

测试数据

本实验数据来源于MAPS(A piano database for multipitch estimation and automatic transcription of music)数据集。此数据集由HAL公开文档提供，用于乐谱MIDI对齐的研究。数据集中的数据是在多种不同乐器在多种环境下采集而来。

测试方法与指标

测试指标为匹配正确的帧数，帧耗时，以及回弹情况下检测恢复正常需要的时间，对每一帧检测的结果，在Ground truth中计算出正确的匹配结果，设置阈值为100ms，若，则检测正确。匹配正确的帧数除以总的帧数表示匹配正确率。考虑实际情况的回弹一般是前后小节跳跃，测试跳跃时间设置为3秒。

测试平台

Windows10，Visual Studio 2015

测试结果

匹配正确率结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 音频编号 | 音频时长 | 匹配帧数 | 总帧数 | 正确率 |
| 1 | 0:28 | 186 | 187 | 99.5% |
| 2 | 0:40 | 303 | 303 | 100% |
| 3 | 0:50 | 337 | 338 | 99.7% |
| 4 | 1:18 | 654 | 665 | 98.3% |
| 5 | 1:26 | 531 | 553 | 96.0% |
| 6 | 1:39 | 841 | 841 | 100% |
| 7 | 1:46 | 671 | 755 | 88.9% |
| 8 | 2:17 | 1103 | 1196 | 92.2% |
| 9 | 2:21 | 1227 | 1264 | 97.1% |
| 10 | 2:24 | 1209 | 1305 | 92.6% |
| 11 | 2:59 | 1802 | 1852 | 97.3% |
| 12 | 4:18 | 2197 | 2338 | 93.9% |
| 总计 | ——— | 10694 | 11291 | 94.7% |

挑选了三首音频，一共测试了9次回弹，回弹时间设置为3秒，最短时间为3.2秒，最长时间为8.0秒，平均时间为4.9秒。

内存占用情况：算法在pc上只是测试版本，最终要移植到移动端，算法内存的消耗主要来自于音频数据和指纹数据，其中音频数据是用作制作指纹，不用放在内存里，所以内存的瓶颈就在于指纹库的内存开销了。目前采用的方式是先分配一个较小的空间，不够就再次分配。测试了两段音频，30秒和3分钟的，在release模式下，帧平均耗时为27毫秒左右和32毫秒左右，指纹内存开销为2.5M左右、5.4M左右。

总结与展望

在不回弹的情况下，算法的匹配正确率可以高达95%，在应对一定条件的回弹，可以在一定时间做出反应，每一帧的处理耗时也比较理想。但是还存在一些问题，算法定位的思路是在一个较大的范围里筛选合适的匹配结果，而音频的节奏改变会对破坏指纹特征，还有大幅度频繁地跳跃演奏会让算法偏离正确的定位，这些情况都可能让定位偏离，偏离程度达到一定程度，就会跳出指纹检索的范围，之后很难回到正确的位置。

基于Android的钢琴演奏评价系统

系统概述

本文基于Android实现了一个用于评价用户钢琴演奏正误的实时系统。用户首先需要导入准备弹奏的正确乐谱，导入完成后乐谱会绘制在屏幕上，进而点击开始录音按钮后即可开始演奏。演奏过程中，进度线会根据乐谱的正确演奏时间实时推进，而评价结果则会在计算完毕后第一时间以将被评价音符改变颜色（演奏正确标记为绿色，否则不改变颜色）的方式显示在屏幕上，由于计算这一过程所耗费的时间在硬件条件足够的情况下可以忽略不计，因而在整个演奏过程中，用户可以实时地看到自己的演奏进度及演奏结果。演奏完毕后，用户可以选择加载新的乐谱或刷新原始乐谱，以便开始下一次演奏。

相关技术

MusicXML文件格式

MusicXML是一种基于XML、专为记谱、分析、检索及乐谱显示等相关应用程序设计的音乐交互格式。MusicXML由Recordare开发，对Finale等其他用于快速交互的程序所使用的本地文件格式进行了补充与完善，且允许任何拥有公共许可证的用户自由使用。

如今，MP3文件已然成为音乐交互过程中的通用格式，MusicXML在乐谱交互过程也具备着相同的地位。据不完全统计，迄今已有超过220个应用程序利用MusicXML格式实现了乐谱的记录、显示及分享等诸多功能。MusicXML采用与XML类似的标签用以记录相关乐谱信息，一段简单的MusicXML格式文件及对应音符（下图中标记为绿色的音符）如下所示：

<note>

<pitch>

<step>A</step>

<octave>4</octave>

<alter>0</alter>

</pitch>

<duration>192</duration>

<type>half</type>

<dot/>

<staff>1</staff>

</note>

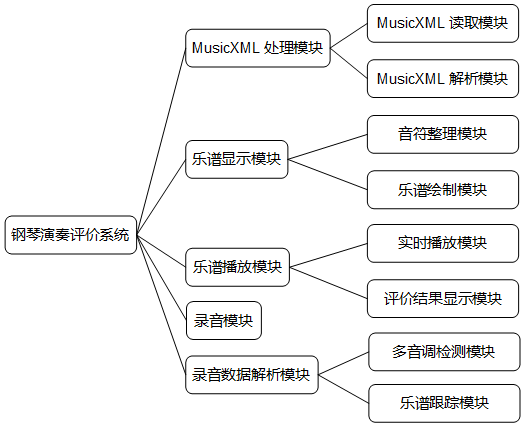


在该段文件中，首先用<note></note>标签定义了一个音符，该标签内部为音符的具体信息。<pitch></pitch>为音符音高，包含了音阶<step></step>、八度<octave></octave>以及<alter></alter>音调变化等信息，上图中音阶为A，八度为4，即对应乐谱中的a1。<duration></duration>为该音符的持续时间，是一个相对值，具体到演奏时间时需要进行换算。<type> </type>表示音符的类型，如“whole”表示全音符、“half”表示二分音符、“4th”表示四分音符等，上图中的音符即为二分音符。<dot/>表示该音符包含附点。<staff></staff>则用于确定该音符所属谱表（高音谱表或低音谱表）。

MusicXML3.0中定义了近700个类似的标签，每个标签都具备不同的含义，综合起来便能够实现对一段乐谱的完整描述。本系统目前仅支持以下标签的解析：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标签名称 | 标签信息 | 标签名称 | 标签信息 | 标签名称 | 标签信息 |
| work\_title | 作品题目 | line | 线值 | time | 节拍 |
| creator | 创建者 | step | 音阶 | clef | 谱号 |
| divisions | 分割 | octave | 八度 | pitch | 音高 |
| fifths | 五度环 | duration | 持续 | rest | 休止 |
| beats | 节拍 | type | 类型 | dot | 附点 |
| beat\_type | 节拍类型 | accidental | 临时记号 | note | 音符 |
| clef | 谱号 | staff | 五线谱 |  |  |
| sign | 符号 | beam | 符杠 |  |  |

系统设计



在钢琴演奏评价系统运行过程中，首先利用MusicXML处理模块获取相应的MusicXML文件内容并解析出后续模块需要的信息，并将这一信息以多个独立对象的形式传递给乐谱显示模块。乐谱显示模块根据这些对象来完成乐谱的绘制工作，并将其适配至ListView中。

绘制工作完成后，录音模块即可开始工作。考虑到内存容量问题，录音模块运行在一个独立的进程中，在完成录音工作的同时，为录音数据解析模块保存相关参数，并在满足条件时调用该模块的相应算法。录音数据解析模块在得到处理结果后，即可将其传递给乐谱播放模块，由乐谱播放模块根据结果进行相应的展示。

系统实现

开发及测试环境

编程开发语言：Java(jdk 1.8.0)

集成开发环境：Android Studio 2.2

Android模拟器：Genymotion 2.6.0

模拟器手机型号：Custom Phone – 768×1280

测试机手机型号：MI 2S / 荣耀 V8

MusicXML处理模块实现

MusicXML处理模块主要用于读取MusicXML格式文件内容，并从内容中解析出系统需要的标签信息，为乐谱显示模块提供数据基础。该模块由MusicXML读取模块以及MusicXML解析模块两个子模块组成。

MusicXML读取模块实现

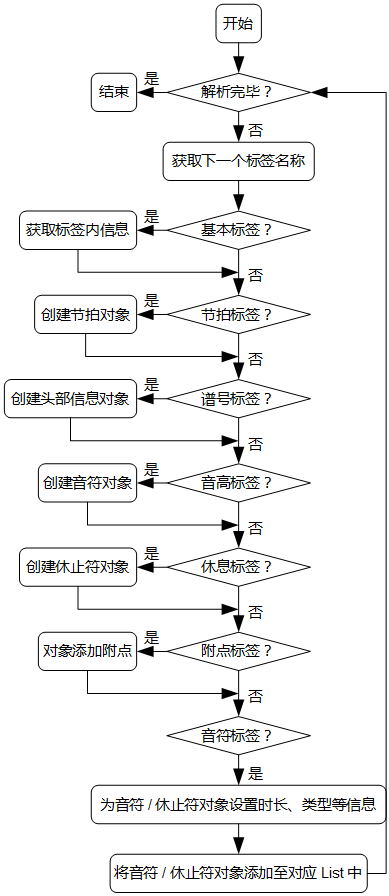
系统对于相关MusicXML文件的处理方式暂时为直接存放在assets文件夹下，并打包至apk中。系统运行时直接通过getResources().getAssets().open(“fileName”)方法打开该文件，并将文件内容逐行保存至一个String对象中，文件读取完毕后，需要关闭BufferedReader以释放被占用的内存。

MusicXML解析模块实现

在获取到MusicXML文件内容后，需要对其中特定的标签进行解析，以为乐谱显示模块提供相应的绘图信息。这里采用XmlPullParser对象完成解析工作，调用该对象的getEventType()方法获取当前解析状态，若尚未解析完毕，则调用nextText()方法获取相应标签下的信息。

而对于解析后信息的储存，系统采用的处理方式为，将整个乐谱中每一个音符或休止符都抽象为一个独立的对象，这些对象中包含了对应音符或休止符的所有信息，如音阶、八度、类型、附点、时长等等；而乐谱中的其他信息，如曲名、作者等则直接以独立的String对象进行传递。

整个解析及保存流程如下所示：



乐谱显示模块实现

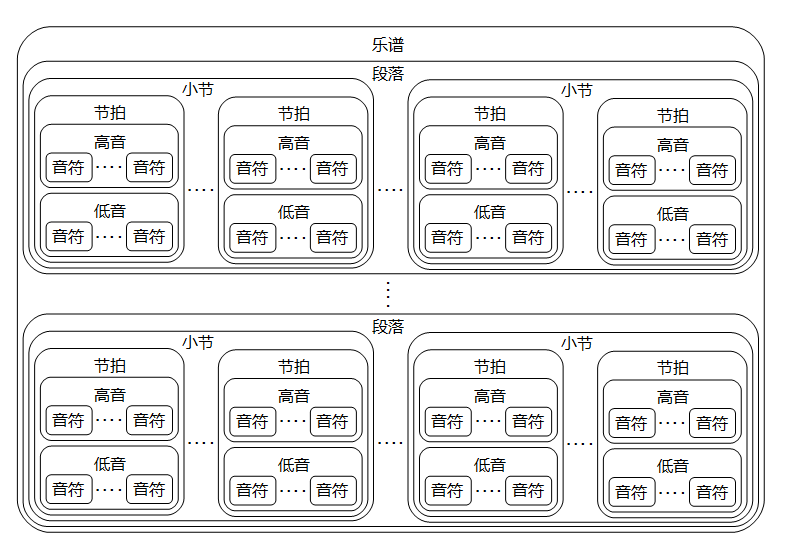
乐谱显示模块主要用于将从MusicXML文件中解析到的信息以绘图的形式呈现给用户，让用户能够看到一张完整且符合标准的乐谱。该模块由音符整理模块以及乐谱绘制模块两个子模块组成。

音符整理模块实现

由于在实际乐谱中，每小节内所包含的音符个数往往是不等的，而手机屏幕宽度是一定的。因此，如何合理设置每小节的宽度，使得这一段的所有音符都能够均匀分布，是系统在进行乐谱绘制前的必要准备工作。而另一方面，对于包含大谱表的乐谱而言，还需要保证高音谱表与低音谱表中在相同时刻演奏的音符应当在乐谱中处于同一垂直线上。音符整理模块便是用于解决上述两个问题的。

在为每小节分配宽度时，系统首先固定了每段的小节总数为4节，之后对于单谱表乐谱而言，以各小节内的音符数作为权值；对于大谱表乐谱而言，则选取两谱表对应小节中包含音符较多的音符数作为权值。进而以该权值来分割屏幕宽度。

因此，为了能够进行小节音符数的计算，音符整理模块需要将获得的音符按照节拍、小节、段落逐一存放如特定的数据结构中，以供绘制模块计算和使用。这一特殊数据结构如下所示：



乐谱绘制模块实现

整个乐谱的绘制是基于一个ListView控件实现的，ListView的一个item承载乐谱的一段（四小节，包含高音谱及低音谱）。因此，乐谱的绘制需要首先根据乐谱信息为ListView创建合适的adapter。而音符整理模块已经根据绘制需求建立好了合适的数据结构，因此，乐谱绘制模块只需根据传入的数据结构计算并分配每小节的宽度，再将根据各音符信息生成的View放入相应的小节内即可。绘制完成的效果如下图所示：

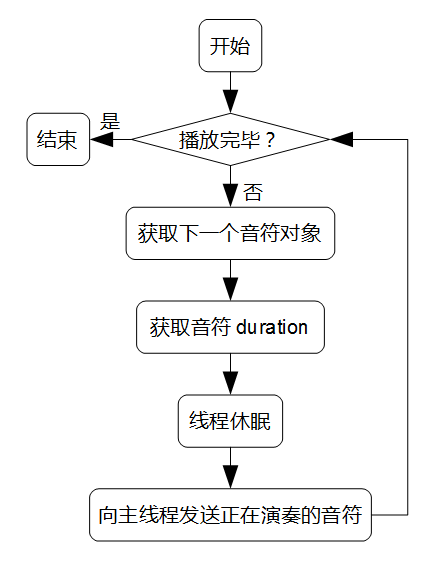


乐谱播放模块实现

乐谱播放模块主要作用为显示乐谱的动态变化。该模块由实时播放模块及评价结果显示模块两个子模块组成。其中，实时播放模块用于在系统运行时显示用户的演奏进度；而评价结果显示模块则会在接收到录音数据解析模块返回的数据后，将这一数据以直观的方式（如音符颜色改变等）呈现给用户。

实时播放模块实现

实时播放模块会在用户开始演奏时被置于后台运行，由于各音符的duration属性均和其实际演奏时间成一定比例，因此该模块可以通过计算某个音符的实际演奏时长，通过线程休眠的方式模拟演奏，并在休眠结束后通知UI线程该音符已经演奏完毕，进而UI线程便可将演奏进度推进至下一个音符。具体流程如下：



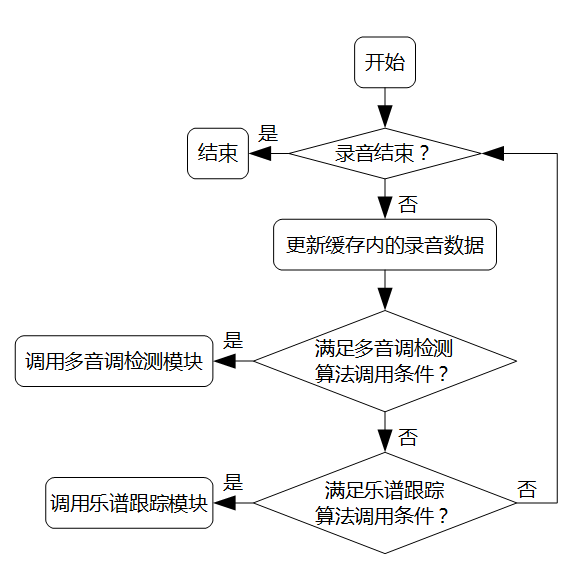
评价结果显示模块实现

评价结果显示模块主要用于当录音数据解析模块执行完算法并返回结果后，在乐谱中进行该结果的显示，如当返回结果为当前音符演奏正确时，评价结果显示模块则会将乐谱中相应音符的颜色变为绿色。

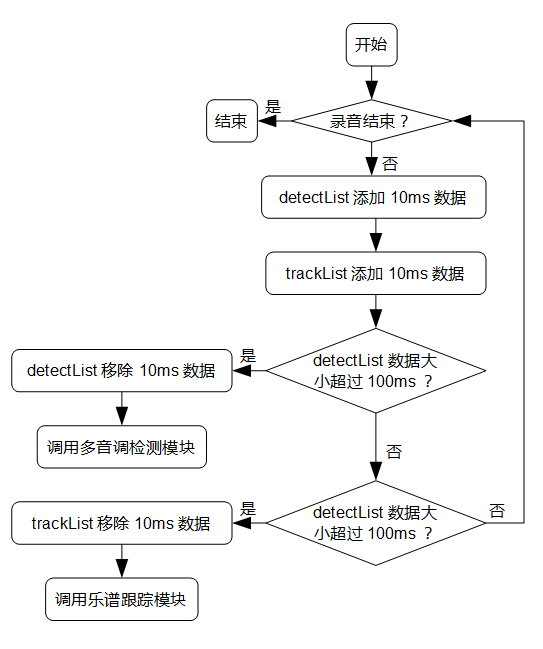


录音模块实现

录音是本系统音频数据的唯一来源。系统开始运行——即开始进行钢琴演奏效果评价时，需要进行持续的录音以获取实时音频数据。考虑到Android系统本身对进程会有内存限制，而录音数据解析模块的算法都需要一定内存的支撑才能运行，因此这里采用多进程的方式将录音模块置于独立进程中运行。其中，多进程采用service的方式实现。同时，录音模块会为录音数据解析模块中两个算法准备参数，且当满足调用条件时，会在主进程中开辟后台线程以完成录音数据的解析工作。具体流程如下：



其中，两算法的调用条件为：系统首先会为两算法分别准备一个用于缓存参数的List，记为detectList和trackList。detectList大小限定为100ms音频数据，而trackList则为8s。录音开始后，上述两个List会以10ms为间隔去获取最新的音频数据，并在二者达到大小限定时，系统会每隔10ms调用一次多音调检测算法，并将当前的detectList作为参数传入；而乐谱跟踪算法则会以100ms的间隔被调用，并传入trackList这一参数。具体流程如下：



录音数据解析模块实现

录音数据解析模块是钢琴演奏评价系统的核心。该模块主要用于调用多音调检测算法及乐谱跟踪算法，并将算法返回结果进行一定处理后发送给主线程中的评价结果显示模块，供其呈现给用户。

录音数据解析模块下的两个子模块——多音调检测模块以及乐谱跟踪模块——均采用Jni的方式实现算法的调用，即创建一个Jni类，类内首先在static方法中利用System.loadLibrary加载C/C++库，进而定义了需要调用的库函数接口，最后即可利用该接口实现相关函数的调用。

系统评测

测试环境

系统测试环境详细参数如下：

|  |  |  |
| --- | --- | --- |
| 环境 | 参数名 | 参数值 |
| 模拟器 | 手机型号 | Custom Phone – 768×1280 |
| 系统版本 | Android 5.0.0(API 21) |
| 真机（低配） | 手机型号 | MI 2S |
| 系统版本 | Android 5.0.2 |
| 真机（高配） | 手机型号 | 荣耀 V8 |
| 系统版本 | Android 6.0 |

测试结果

MusicXML处理模块 & 乐谱显示模块

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 测试方法 | 预期结果 | 测试结果 |
| 1-1 | 选择需要显示的MusicXML文件 | 显示选中文件对应的乐谱 | 通过 |
| 1-2 | 完成一次演奏评价后刷新乐谱 | 乐谱回归未被标记的状态 | 通过 |

乐谱播放模块 & 录音模块 & 录音数据解析模块

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 测试方法 | 预期结果 | 测试结果 |
| 2-1 | 生成乐谱后点击播放按钮 | 播放进度线根据实际演奏时间实时推进 | 通过 |
| 2-2 | 点击开始录音并播放该乐谱的正确音频 | 乐谱音符被实时标记为绿色 | 通过 |
| 2-3 | 点击开始录音并播放该乐谱存在部分错误的音频 | 乐谱中被正确演奏的音符被实时标记为绿色，错误演奏音符则无变化 | 通过 |

算法执行效率

|  |  |  |
| --- | --- | --- |
| 环境  运行速度  算法 | 真机（低配） | 真机（高配） |
| 多音调检测算法 | 约350~450毫秒/次 |  |
| 乐谱跟踪算法 |  |  |

总结与展望

迄今为止，该系统已然能够实现乐谱的绘制以及针对简单乐谱的演奏评价等功能，但系统仍旧处在Demo的尴尬位置，还未能达到应用级别。导致这一局面的问题有如下几点：

首先，系统在乐谱绘制这一功能上仍存在不完善的问题，一方面，系统目前只适配于五线谱，而简谱等其他类型的乐谱系统则无法实现其相关功能；另一方面，对于复杂的乐谱或乐谱符号，如连音符等，系统还无法绘制。

其次，由于系统所调用的两个算法在运算时均会占据一定的内存，而且调用频率很高，导致同一时间内会有多个算法同时执行。这就导致系统对运行环境的要求较高，就系统测试用到的两部真机而言，配置较高的荣耀 V8可以做到实时显示评价结果，而配置较低的MI 2S则会有较大延迟，影响用户的使用。

再次，系统目前对用户的操作有一定要求，用户体验不佳。这一问题对于一款应用而言是较为致命的。

最后，由于在系统设计阶段考虑不够周全，目前的系统结构并不是十分清晰，仍存在多处可以改进的地方。这一问题同样是亟待解决的，否则会对未来系统的扩展带来很大的麻烦。

从上述问题可以看出，系统重构势在必行。一是要对现有的系统进行优化，增强系统的流畅性和稳定性；二是要完善系统架构的设计，增强系统的可扩展性；三是要以应用为目标，提升系统的用户体验。这一系列工作将是未来一段时间钢琴演奏评价系统开发的核心所在。

乐器识别

国内外研究现状

从广义上来说，乐器可以分成西方乐器和民族乐器两个类别。本文的研究内容主要基于西方乐器。根据乐器发声方式的不同，西方乐器又可以分为三大类，分别是弦乐器、木管乐器和铜管乐器。不同种类的乐器在声音上有较大差别，但同一种类的乐器其声音差别就很小了，常人是很难分辨出来的。即使是专业人才，也难以分辨出所有乐器的声音，其精度和准度也是有限的。不过，在机器学习技术的帮助之下，乐器的识别工作变得相对简单，识别的精度也有了很大提高。

在1990年以前，音乐领域中关于乐器识别的研究少之又少，不过在这之后，这个方向有了较大的突破，涌现了一大批相关的专业研究成果。1999年，Martin使用感知特征对27种乐器进行了识别，乐器的个体识别率为71%。2001年，Eronen使用Mel频率、Δ倒谱系数、LPCC等特征对16种西方管弦乐器进行识别，最高准确率没能超过80%。2004年，Krishna在KNN和GMM分类器上，通过使用MFCC、LPCC等特征对10多种单音乐器进行训练，其最高识别率达到90%。同年，Essid使用波谱特征在二次分类器上进行训练，得到27种个体乐器的识别率为92.81%。在这些关于乐器识别的研究中，一直围绕着特征抽取和机器学习算法这两个重点，而本文也旨在总结前人研究的基础上寻找更好的乐器识别特征和机器识别算法的组合。

相关技术

支持向量机

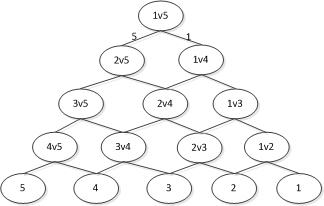
支持向量机(Support Vector Machine, SVM)是 Corinna Cortes 于 1995 年首先提出的，它在解决小样本、非线性及高维模式识别中表现出许多特有的优势，并能够推广应用到函数拟合等其他机器学习问题中。支持向量机方法根据有限的样本信息在模型的复杂性和学习能力之间寻求最佳折衷，以获得最好的推广能力。SVM成功应用于文本分类、人脸识别、语音识别和乐器识别等多种领域。

SVM是一种典型的两类分类器。而现实中要解决的问题，往往是多类的问题。一次性考虑所有样本，并求解一个多目标函数的优化问题，一次性得到多个分类面，多个超平面把空间划分为多个区域，每个区域对应一个类别，给一个样本，看它落在哪个区域就知道了它的分类。只可惜这种算法还基本停留在纸面上，因为一次性求解的方法计算量实在太大，无法实用。目前，存在的方法主要有：

（1）“1-V-R方式”，就是每次仍然解一个两类分类的问题。比如我们有5个类别，首先把类别1的样本定为正样本，其余的样本合起来定为负样本，得到一个两类分类器，它能够指出新的样本是不是第1类的；然后我们把类别2的样本定为正样本，把1、3、4、5的样本合起来定为负样本，得到一个分类器，如此下去，最终可以得到5个这样的两类分类器。这种方法的好处是每个优化问题的规模比较小，而且分类的时候速度很快（对于k类问题，把其中某一类的n个训练样本视为一类，所有其他类别归为另一类，因此只有k个分类器）。但有时可能会出现两种特殊情况，某样本属于多个类别（分类重叠现象）或者是某样本没有判别为任何类别（不可分类现象）。

（2）“1-V-1方式”，也就是我们所说的one-against-one方式。这种方法把其中的任意两类构造一个分类器，共有(k-1)×k/2个分类器。虽然分类器的数目多了，但是在训练阶段所用的总时间却比“一类对其余”方法少很多。最后预测中如果出现分类重叠现象，可以采用竞争方式（各个分类器向k个类别投票，取得票最高类）。但是如果类别数非常大时，要调用的分类器数目会达到类别数的平方量级，预测的运算量不可小觑。

（3）“有向无环图（DAG-SVM）”，该方法在训练阶段采用1-V-1方式，而判别阶段采用一种两向无环图的方式。

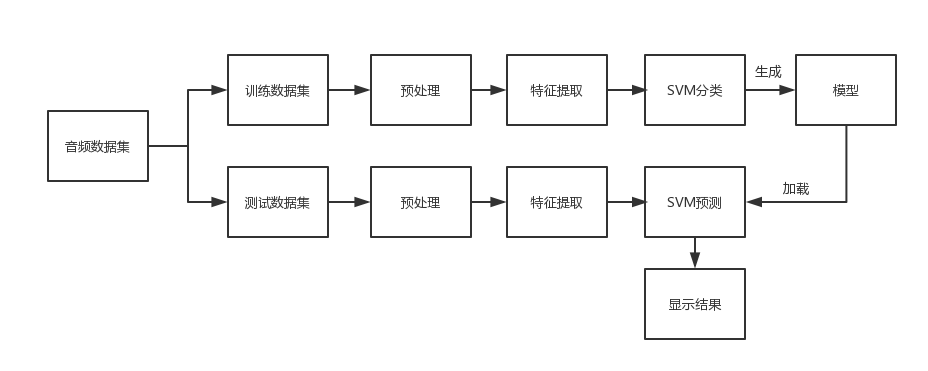


如果类别数是k，则只调用k-1个分类器即可。但是如果开始的分类器回答错误，那么后面的分类器是无论如何也无法纠正它的错误的，其实对下面每一层的分类器都存在这种错误向下累积的现象。也有一些方法可以改善整体效果，我们总希望根节点少犯错误为好，因此参与第一次分类的两个类别，最好是差别特别大，或者取在两类分类中正确率最高的那个分类器作根节点，或者我们让两类分类器在分类的时候，不光输出类别的标签，还输出一个类似“置信度”，当它对自己的结果不太自信的时候，我们就不光按照它的输出走，它可会按照一定的概率走向另一分支。

LibSVM采用的是1-V-1方式，因为这种方式思路简单，并且许多实践证实效果比1-V-R方式要好。

算法流程

基于libSVM的乐器识别的流程：



预处理

预处理主要是音频解析读取和静音处理。

(1) 音频解析

数据集音频采用的是wav文件。按照wav的文件格式对音频进行解析。对于多声道音频默认取第一声道。

(2) 静音处理

这里静音检测使用的方法是能量阈值法。提取有声区段进行后面的特征提取。

特征提取

这里选取的是一些通用的声音特征。

中时特征：zcr（过零率）、spectral（频谱流量）、spectral rolloff（频谱衰减）和MFCC（基于Mel频谱的倒谱系数）

MPEG-7：HSC（谐波谱质心）、HSD（谐波谱偏差）、HSV（谐波谱变化）、LAT、ASF

训练和预测

SVM训练和预测使用的是libSVM的库（有C实现的版本，可以在Android中进行本地调用）。

算法评测

测试集

测试集来自于IOWA数据库（共有乐器单音音频2963个），其中的乐器种类有19种，如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| woodwind | Flute |  | Alto saxophone |
|  | Alto flute |  | Soprano saxophone |
|  | Bass flute | Strings | Violin（2012） |
|  | Oboe |  | Viola（2012） |
|  | Eb clarinet |  | Cello（2012） |
|  | Bb clarinet |  | Double bass（2012） |
|  | Bass clarinet |  |  |
|  | Bassoon |  |  |
| Brass | Horn |  |  |
|  | Bb trumpet |  |  |
|  | Tenor Trombone |  |  |
|  | Bass trombone |  |  |
|  | Tuba |  |  |

其乐器格式为16/ 24bit、44.1 kHz、单声道/立体声、AIFF**。**文件命名规则为乐器.演奏技术.动态水平（pp、mf、ff）.音调.立体声。

测试结果

将数据集中的数据按8：1划分用于训练和测试，测试得到的识别准确度为76.0496%。（低于相关研究中的结果，可能是选取了无关的特征影响了识别，也可能是SVM算法的局限，后面打算只用MFCC和MPEG特征做训练看看结果，再对比GMM训练和预测的结果看看）

1. <http://www.fftw.org/> [↑](#footnote-ref-1)
2. <https://sourceforge.net/projects/kissfft/> [↑](#footnote-ref-2)
3. https://computing.llnl.gov/tutorials/pthreads [↑](#footnote-ref-3)
4. http://www.openblas.net/ [↑](#footnote-ref-4)