大数据综合处理实验 课程设计 实验报告

小组编号: 27
姓名：倪昀
学号：211294003
联系方式：scanocdii@outlook.com
导师：朱光辉
研究领域：大数据与智能计算
课程设计题目：音乐数据处理与分析

目录

1 摘要 **4**

2 研究背景 **4**

3 技术难点和解决的问题 **4**

4 主要方法和设计思路 **5**

5 详细设计说明 **8**

5.1 数据集预处理 8

5.1.1 预处理歌曲元数据 8

5.1.2 预处理歌词信息 10

5.1.3 预处理流派、用户信息 11

5.1.4 数据清洗 12

5.2 音乐数据特征分析 14

5.2.1 歌曲流行度分析 14

5.2.2 用户喜好分析 15

5.2.3 歌曲播放时长分布 18

5.2.4 歌词词频分析 19

5.2.5 流派信息挖掘 21

5.3 歌曲情感分析 23

5.3.1 模型训练 24

5.3.2 情感预测 25

5.3.3 效果优化 27

5.4 歌曲流派分类 28

5.4.1 模型训练 28

5.4.2 流派预测 29

5.4.3 效果优化 31

5.5 可视化系统设计 32

5.5.1 前端设计 32

5.5.2 后端设计 33

6 输入文件和输出结果说明 34

7 实验结果说明和分析 41

8 总结和展望 48

9 参考文献 50

1 摘要

本实验利用 Hadoop MapReduce 编程模型，对一个包含大量音乐信息的数据集进行了全面的处理和分析。我对数据进行了预处理，包括读取、清洗和转换。我对音乐数据进行了各项分析，包括对歌曲的流行度、用户喜好和歌词词频等的分析与总结。我应用机器学习算法，对歌曲的情感和流派进行了分类和预测。我将分析结果进行了可视化展示，设计可视化系统，以便更好地理解和数据特征和模型结果。

2 研究背景

随着数字音乐产业的快速发展，音乐数据的规模和复杂性日益增加。每天，音乐平台和流媒体服务都会生成大量用户行为数据、歌曲元数据和歌词文本。

这些数据蕴含了丰富的商业价值和研究潜力。一方面，大数据隐含着更准确的事实，通过对海量音乐数据的分析、总结，可以更准确地获得用户偏好、音乐流行程度等信息。另一方面，大数据将带来巨大的技术和商业机遇，通过对音乐的综合分析，可以完成过去小规模数据集上无法实现的任务，例如使用机器学习方法进行预测，从而进行更精确的推荐。

大规模的数据对计算模式提出了更高的要求，因此使用分布式计算来处理这些数据成为更好的选择。在本实验中，通过 Hadoop MapReduce 计算框架，调度不同节点上的计算资源来完成同一个任务，实现更高的分布性与扩展性，让大规模音乐数据的操作变得可行。

3 技术难点和解决的问题

本实验主要包含以下技术难点，并通过相应的方法加以解决：

1. **数据预处理复杂度高：**音乐数据集包含多种类型的数据，如歌曲元数据、歌词、流派和用户信息。尤其是歌曲元数据为 HDF5 格式的文件，读取复杂度较高，单机处理方式效率低下。而数据集中不同数据的读取操作是独立的，因此可以采用 MapReduce，将不同的文件内容分配

到不同 Mapper 节点上并行读取和处理，最终由 Reducer 节点汇总，提高处理效率。

2. **大规模数据分析处理：**在对庞大的数据集进行特征分析时，需要将同类信息汇总。收集同类信息需要扫描整个数据集，并对每一项数据进行收集。其中对不同数据的收集操作是独立的，因此可以利用 MapReduce，让不同的 Mapper 节点分别收集一部分数据，然后由 Reducer 节点汇总同类数据。

3. **机器学习模型训练和预测：**在对音乐进行情感分析和流派预测时，需要应用机器学习算法来构建模型。其中训练阶段需要利用训练集中的样本构建模型，对不同样本的处理之间可以并行处理，让不同的 Mapper 分别处理样本并统计数据，由 Reducer 汇总同类的模型特征。预测阶段需要将模型应用于测试集中的每一个样本，对不同样本的预测操作可以并行处理，让不同的 Mapper 分别对一部分样本进行预测。

4. **数据可视化：**为了将数据预处理、特征分析、训练和预测的操作过程和结果可视化，我利用 HTML、Flask 等相关技术设计可视化系统，让用户可以直接通过网页操作调用后端的计算服务，并查看计算结果。

4 主要方法和设计思路

在本节中，我将介绍本实验各部分的主要设计思路，具体设计细节将在第 5 节“详细设计说明”中介绍。

1. **数据集预处理：**

   (1) 处理歌曲元数据：每个 Mapper 分别处理一个 HDF5 文件，提炼其中信息，转化为相应格式后由 Reducer 统一汇总到文件 songs.txt。

   (2) 处理歌词数据、流派数据和用户数据：每个 Mapper 分别处理文件中的一行信息，将其转化为所需格式后由 Reducer 汇总到文件 lyrics.txt、genres.txt、users.txt。

(3) 清洗数据，去除冗余项：每个 Mapper 处理需要清洗的文件（songs.txt、lyrics.txt、genres.txt、users.txt 中的一个）中的一行，通过与其它文件匹配，决定是否保留这一行，将保留的行交给 Reducer 汇总。

2. **音乐数据特征分析：**

(1) 歌曲流行度分析：每个 Mapper 处理用户文件 users.txt 中的一行，通过连接歌曲文件 songs.txt 获取歌名，发送歌名与对应的播放次数。一个 Reducer 汇总一个歌名对应的播放次数总和。最后通过排序得到播放次数最高的 10 个音乐作品。

(2) 用户喜好分析：每个 Mapper 处理用户文件 users.txt 中的一行，通过连接歌曲文件 songs.txt 获取音乐家 ID，发送用户/音乐家 ID 与对应的播放次数。一个 Reducer 汇总一个用户/音乐家 ID 对应的播放次数总和。最后通过比较得到播放次数最多的用户/音乐家。

(3) 歌曲播放时长分布：每个 Mapper 处理歌曲文件 songs.txt 中的一行，根据播放时长将其归类到一个区间并发送。一个 Reducer 节点汇总一个时长区间的音乐数量。最后使用 seaborn 库绘制直方图。

(4) 歌词词频分析：每个 Mapper 处理歌词文件 lyrics.txt 中的一行，通过连接流派文件 genres.txt 获取对应流派，发送该流派与每个单词对在这一行中的出现次数。一个 Reducer 统计一个流派中一个特定单词的总出现次数。另外用一个 MapReduce 任务统计每一个流派的出现次数（每个 Mapper 处理歌词文件的一行并增加对应流派次数），最终使用 wordcloud 库生成出现次数最多的流派的词云图。

(5) 流派信息挖掘：每个 Mapper 处理歌曲文件 songs.txt 中的一行，通过连接流派文件 genres.txt 获取对应流派，发送该流派以及对应的各项特征值。一个 Reducer 处理一个流派，将这个流派的所有歌曲的特征值求平均。最后使用 matplotlib 库绘制折线图。

3. **歌曲情感分析：**

使用朴素贝叶斯分类对歌曲的情感进行预测，将 5000 个高频词在一个歌曲中出现的次数作为其特征向量。分为训练模型和预测情感两个阶段。

(1) 训练模型：每个 Mapper 处理 lyrics.txt 中被 sentiment_train.txt 标记了的一行，让该情感类别以及每个词语出现次数的频数加一。一个 Reducer 汇总一个情感类别或是词语出现次数的频数。

(2) 预测情感：每个 Mapper 处理 lyric1.txt 中待预测的一行歌曲的歌词信息，分别计算其被归类到每一个情感类别的概率，并取最高者输出。一个 Reducer 将一个歌曲预测的结果输出。

4. **歌曲流派分类：**

使用朴素贝叶斯分类对歌曲的情感进行预测，将每个词语在一个歌曲的歌名和作者名称中出现的次数作为其特征向量。分为训练模型和预测情感两个阶段。

(1) 训练模型：每个 Mapper 处理 unique_tracks.txt 中被 genres.txt 标记了的一行，让该流派类别以及歌名和作者名称中每个词语出现的频数加一。一个 Reducer 汇总一个流派类别或是词语出现频数。

(2) 预测情感：每个 Mapper 处理 unique_tracks.txt 中待预测的一行歌曲名称和作者名称信息，分别计算其被归类到每一个流派类别的概率，并取最高者输出。一个 Reducer 将一个歌曲预测的结果输出。

5. **可视化系统设计：**

使用 Website 技术设计用户交互系统，将后端计算与前端 GUI 分离。

(1) 前端设计：使用 HTML、CSS 和 JavaScript 渲染网页，通过 Ajax 向后端上传文件、调用后端服务以及从后端获取结果。

(2) 后端设计：使用 Flask 配置服务器，对前端不同路由的请求运行相应的 JAR 包，返回相应结果。

5 详细设计说明

在本节中，我将详细介绍本实验的算法、程序框架、主要类等的设计，具体从数据集预处理、音乐数据特征分析、歌曲情感分析、歌曲流派分类和可视化系统设计五个任务的设计做出说明。

5.1 数据集预处理

数据集预处理的任务包含对歌曲元数据、歌词信息、流派信息、用户信息的格式化以及对格式化后的数据进行清洗，根据 track_id、song_id 将各个文件中没有任何匹配的行删除。

5.1.1 预处理歌曲元数据

该部分任务包含一个 MapReduce 作业。

自定义输入格式类 FilenameInputFormat，让每个 Mapper 获得的输入是一个 HDF5 文件的路径。

Mapper 输出的键值对为 <song_id, features>，表示预处理后一首歌曲的各项特征。其中键 song_id 为 Text 类型对象，表示当前歌曲的 ID；值 features 为 Text 类型对象，格式为 song_id, track_id, title, release, artist_id, artist_name, mode, energy, tempo, loudness, duration, danceability, year，表示这个歌曲的各项特征，由逗号隔开。

Reducer 输出的键值对为 <null, features>，表示预处理后歌曲文件的一行。其中键是 NullWritable 类型对象；值 features 为 Text 类型对象，是 Mapper 输出的一首歌曲的特征。

以下是各个类主要算法设计的伪代码：

```
public class PrepSongsMapper {
   public void map(key, value=filePath) {
      try {
         // 将文件从 HDFS 复制到本地
         fs.copyToLocalFile(filePath, localTempPath);
         fileId = H5.H5Fopen(localTempPath);

         // 读取各字段值
```

```
            String song_id = readField(fileId, "/metadata/songs", "
                song_id");
            ...

            // 格式化输出字符串
            String features = song_id + "," + ...;

            // 输出键值对 <song_id, features>
            emit(song_id, features);
        } catch (Exception e) {
            ...
        }
    }

    private String readField(fileId, datasetPath, fieldName) {
        // 根据文件编号 fileId 和字段路径 datasetPath 找到字段的位置
        // 返回 filedName 指定的字段值
    }
}
```

PrepSongsMapper 类伪代码

```
public class PrepSongsReducer {
    protected void reduce(key=song_id, values=[feature]) {
        // 简单输出 feature
        write(null, feature);
    }
}
```

PrepSongsReducer 类伪代码

```
public class FilenameInputFormat extends FileInputFormat<LongWritable,
     Text> {
    // 自定义RecordReader类，用于读取文件路径
    public static class FilenameRecordReader extends RecordReader<
        LongWritable, Text> {
        // 读取下一个键值对
        @Override
        public boolean nextKeyValue() {
```

```
            if (!processed) {
                value.set(fileSplit.getPath().toString()); // 将文件路径设
                    置为值
                processed = true; // 标记为已处理
                return true;
            }
            return false;
        }
    }
}
```

FilenameInputFormat 类伪代码

5.1.2 预处理歌词信息

该部分任务包含一个 MapReduce 作业。

将包含单词表的文件放入缓存作为全局共享文件。

Mapper 的输入是歌词信息文件的一行，有效行的格式为 track_id, mxm_track_id, word_id1:count1, word_id2:count2 ...。

Mapper 输出的键值对是 <track_id, words>，表示预处理后一首歌曲的歌词信息。其中键 track_id 是 Text 类型对象，表示当前歌曲的 ID；值 words 是 Text 类型对象，格式为 track_id, mxm_track_id, [(word1:count1), (word2:count2) ...]，即一行歌词信息。

Reducer 输出的键值对为 <null, words>，表示预处理后歌词文件的一行。其中键是 NullWritable 类型对象；值 words 为 Text 类型对象，是 Mapper 输出的一首歌曲的歌词信息。

以下是各个类主要算法设计的伪代码：

```
public class PrepLyricsMapper {
    // 单词列表
    String[] wordList = setup();

    public void map(key, value=line) {
        // 处理有效的歌词行
        if (line[0] != '#' and line[0] != '%') {
            // 遍历当前歌曲的全部 word_id:count
```

```
            for (int i = 2; i < parts.length; ++i) {
                // 获取当前 word_id 对应的单词并添加到输出字符串
                words += "(" + wordList[wordId] + ":" + count + ")";
                if (i != parts.length - 1) {
                    words += ",";
                }
            }

            words += "]";

            // 输出键值对 <track_id, words>
            emit(trackId, words);
        }
    }
}
```

PrepLyricsMapper 类伪代码

```
public class PrepLyricsReducer {
    protected void reduce(key=track_id, values=[words]) {
        // 简单输出 words
        write(null, words);
    }
}
```

PrepLyricsReducer 类伪代码

5.1.3 预处理流派、用户信息

对流派、用户信息的预处理过程类似，均只需简单地将相应文件每一行格式化，以下以预处理流派信息为例。

该部分任务包含一个 MapReduce 作业。

Mapper 的输入是流派信息文件有效的一行，格式为 track_id, genre。

Mapper 输出的键值对是 <track_id, genre>，表示预处理后的一首歌曲的流派信息。其中键 track_id 是 Text 类型对象，表示当前歌曲的 ID；值 genre 是 Text 类型对象，表示对应的流派。

Reducer 输出的键值对为 <track_id, genre>，表示预处理后流派文件的一行。即简单输出 Mapper 输出的键值对，中间由逗号隔开。

以下是各个类主要算法设计的伪代码：

```
public class PrepGenresMapper {
    public void map(key, value=track_id,genre) {
        emit(track_id, genre);
    }
}
```

PrepGenresMapper 类伪代码

```
public class PrepLyricsReducer {
    protected void reduce(key=track_id, values=[genre]) {
        // 简单输出键值对 <track_id, genre>
        write(trackId, genre);
    }
}
```

PrepGenresReducer 类伪代码

5.1.4 数据清洗

该部分任务包含 4 个 MapReduce 作业，每个作业清洗预处理获得的一个文件。

每个作业的输入是待清洗的文件，并且将其余 3 个文件中可以与之连接的文件放入缓存作为全局共享文件。由于每个作业要读取另外 3 个文件，我选择先清洗规模较大的文件，以使得每个作业读取其它文件的时间开销最小化。

4 个作业的过程类似，以下以清洗歌曲文件 songs.txt 为例。

Mapper 的输入是预处理后用户文件的一行，格式为 song_id, track_id, title, release, artist_id, artist_name, mode, energy, tempo, loudness, duration, danceability, year。

Mapper 输出的键值对与预处理过程一致，为 <song_id, features>，其中键 song_id 为 Text 类型对象，表示当前歌曲的 ID；值 features 为 Text

类型对象，格式为 song_id, track_id, title, release, artist_id, artist_name, mode, energy, tempo, loudness, duration, danceability, year，表示清洗后保留的一首歌曲的各项特征，由逗号隔开。

Reducer 输出的键值对为 <null, features>，表示清洗后歌曲文件的一行。其中键是 NullWritable 类型对象；值 features 为 Text 类型对象，是 Mapper 输出的一首歌曲的特征。

以下是各个类主要算法设计的伪代码：

```
public class FilterSongsMapper {
    // 读取并保存其它三个文件中出现的 song_id 和 track_id
    Set<String> songIdSet = setup();
    Set<String> trackIdSet = setup();

    public void map(key, value=song_id,track_id) {
        // 如果 song_id 或 track_id 存在于集合中，则保留键值对 <song_id,
            features>
        if (songIdSet.contains(song_id) || trackIdSet.contains(track_id)
            ) {
            emit(song_id, features);
        }
    }
}
```

FilterSongsMapper 类伪代码

```
public class FilterSongsReducer {
    protected void reduce(key=song_id, values=[features]) {
        // 简单输出键值对 <song_id, features>
        write(song_id, features);
    }
}
```

FilterSongsReducer 类伪代码

5.2 音乐数据特征分析

音乐数据特征分析的任务包含歌曲流行度分析、用户喜好分析、歌曲播放时长分布、歌词词频分析以及流派信息挖掘。

5.2.1 歌曲流行度分析

该部分任务包含一个 MapReduce 作业。

将预处理后的歌曲文件 songs.txt 放在缓存作为全局共享文件。

Mapper 的输入是预处理后的用户文件 users.txt 的一行，格式为 user_id, song_id, play_count。

Mapper 输出的键值对是 <song_name, play_count>，表示一首歌曲及其播放次数。其中键 song_name 是 Text 类型对象，表示一首歌曲的歌名；值 play_count 是 IntWritable 类型对象，表示这首歌曲的播放次数。

Reducer 输出的键值对是 <song_name, total_play_count>，表示一首歌曲及其播放次数总和。其中键 song_name 是 Text 类型对象，表示一首歌曲的歌名；值 total_play_count 是 IntWritable 类型对象，表示这首歌曲的总播放次数。

MapReduce 任务结束后，通过排序得到 10 首播放次数最高的歌曲。

以下是各个类主要算法设计的伪代码：

```
public class PopularityMapper {
    // 读取缓存文件并建立歌曲 ID 到歌名的映射
    Map<String, String> titleMap = setup();

    public void map(key, value=line) {
        if (titleMap.containsKey(songId)) {
            // 将该歌曲 ID 对应到歌名
            String songName = titleMap.get(songId);

            // 输出键值对 <song_name, play_count>
            emit(songName, playCount);
        }
    }
}
```

PopularityMapper 类伪代码

```
public class PopularityReducer {
   protected void reduce(key=songName, values=[playCount1, playCount2,
        ...]) {
      int totCount = 0;

      // 累加所有的播放次数
      for (playCount in values) {
         totCount += playCount;
      }

      // 输出键值对 <song_name, total_play_count>
      write(songName, totCount);
   }
}
```

PopularityReducer 类伪代码

5.2.2 用户喜好分析

该部分包含两个 MapReduce 作业，分别用来统计用户听歌数量和艺术家作品被播放次数。

第一个作业统计用户听歌数量。

Mapper 的输入是用户文件的一行，格式为 user_id, song_id, play_count。

Mapper 输出的键值对是 <user_id, play_count>，表示一个用户的听歌数量。其中键 user_id 是 Text 类型对象，表示一个用户的 ID；值 play_count 是 IntWritable 类型对象，表示听歌数量。

Reducer 输出的键值对是 <user_id, total_play_count>，表示一个用户的听歌数量总和。其中键 user_id 是 Text 类型对象，表示一个用户的 ID；值 play_count 是 IntWritable 类型对象，表示总听歌数量。

MapReduce 任务结束后，通过比较得到听歌数量最多的用户。

以下是各个类主要算法设计的伪代码：

```
public class UserSongCountMapper {
    public void map(key, value=line) {
        // 输出键值对 <user_id, play_count>
        emit(userId, playCount);
    }
}
```

UserSongCountMapper 类伪代码

```
public class UserSongCountReducer {
    protected void reduce(key=userId, values=[playCount1, playCount2,
        ...]) {
        int totCount = 0;

        // 累加所有的播放次数
        for (playCount in values) {
            totCount += playCount;
        }

        // 输出键值对 <user_id, total_play_count>
        write(userId, totCount);
    }
}
```

UserSongCountReducer 类伪代码

第二个作业统计艺术家歌曲被播放次数。

将预处理后的歌曲文件 songs.txt 放在缓存作为全局共享文件。

Mapper 的输入是用户文件的一行，格式为 user_id, song_id, play_count。

Mapper 输出的键值对是 <artist_id, play_count>，表示一个艺术家歌曲被播放的次数。其中键 artist_id 是 Text 类型对象，表示一个艺术家的 ID；值 play_count 是 IntWritable 类型对象，表示歌曲被播放次数。

Reducer 输出的键值对是 <artist_id, total_play_count>，表示一个艺术家歌曲被播放的总次数。其中键 artist_id 是 Text 类型对象，表示一个用户的 ID；值 play_count 是 IntWritable 类型对象，表示总播放次数。

MapReduce 任务结束后，通过比较得到歌曲被播放次数最多的艺术家。

以下是各个类主要算法设计的伪代码：

```
public class ArtistSongCountMapper {
    // 读取缓存文件并建立歌曲 ID 到艺术家 ID 的映射
    Map<String, String> artistIdMap = setup();

    public void map(key, value=line) {
        // 如果歌曲 ID 存在于映射中，则输出艺术家 ID 和播放次数
        if (artistIdMap.containsKey(songId)) {
            // 将该歌曲 ID 对应到艺术家 ID
            String artistId = artistIdMap.get(songId);

            // 输出键值对 <artist_id, play_count>
            emit(artistId, playCount);
        }
    }
}
```

ArtistSongCountMapper 类伪代码

```
public class ArtistSongCountReducer {
    protected void reduce(key=artistId, values=[playCount1, playCount2,
        ...]) {
        int totCount = 0;

        // 累加所有的播放次数
        for (playCount in values) {
            totCount += playCount;
        }

        // 输出键值对 <artist_id, total_play_count>
        write(artistId, totCount);
    }
}
```

ArtistSongCountReducer 类伪代码

5.2.3 歌曲播放时长分布

该部分任务包含一个 MapReduce 作业。

Mapper 的输入是预处理后的歌曲文件 songs.txt 的一行，格式为 song_id, track_id, title, release, artist_id, artist_name, mode, energy, tempo, loudness, duration, danceability, year。

Mapper 输出的键值对是 <duration_gap, song_count>，表示一个区间的歌曲数量。其中键 duration_gap 是 Text 类型对象，表示一个区间；值 song_count 是 IntWritable 类型对象，这个区间的歌曲数量。

Reducer 输出的键值对是 <duration_gap, total_song_count>，表示一个区间的歌曲总数。其中键 duration_gap 是 Text 类型对象，表示一个区间；值 total_song_count 是 IntWritable 类型对象，这个区间的歌曲数量之和。

以下是各个类主要算法设计的伪代码：

```
public class DurationMapper {
    public void map(key, value=line) {
        // 获得当前歌曲播放时长
        String[] parts = line.split(",");
        double duration = Double.parseDouble(parts[10]);

        // 根据歌曲时长获取对应的分类区间
        int durationGap = getGap(duration);

        // 输出键值对 <duration_gap, song_count>
        emit(durationGap, 1);
    }

    private static String getCluster(double duration) {
        // 根据 duration 的值确定所属的区间
    }
}
```

DurationMapper 类伪代码

```
public class DurationReducer {
```

```
    protected void reduce(key=duration_gap, values=[song_count1,
        song_count2, ...]) {
        int total_song_count = 0;

        // 累加当前时长区间分类的歌曲数量
        for (song_count in values) {
            total_song_count += song_count;
        }

        // 输出键值对 <duration_gap, total_song_count>
        write(duration_gap, total_song_count)
    }
}
```

DurationReducer 类伪代码

之后根据输出的各区间歌曲数量，使用 seaborn 库绘制柱状图。

```
# 读取各个区间歌曲数量
gaps, counts = read_data('task23.txt')

# 使用Seaborn的barplot绘制柱状图，x轴为gaps，y轴为counts
barplot = sns.barplot(x=gaps, y=counts)

# 添加标签和标题等信息
plt.xlabel('播放时长', fontsize=14)
plt.ylabel('歌曲数量', fontsize=14)
plt.title('歌曲播放时长分布图', fontsize=18)
...
```

绘制柱状图伪代码

5.2.4 歌词词频分析

该部分任务包含 2 个 MapReduce 作业，一个用来统计每个流派的词频，另一个用来找出包含歌曲最多的流派。

第一个 MapReduce 作业统计词频。

将预处理后的流派文件放在缓存作为全局共享文件。

Mapper 的输入是预处理后的歌词文件的一行，格式为 track_id, mxm_track_id, [(word1:count1), (word2:count2) ...]。

Mapper 输出的键值对是 <<genre, word>, word_count>，表示一个流派中一个单词出现次数。其中键 <genre, word> 是 Text 类型对象，由流派名和单词名组合而成；值 word_count 是 IntWritable 类型对象，表示单词出现次数。

Reducer 输出的键值对是 <word, total_word_count>，表示一个单词出现次数。其中键 word 是 Text 类型对象，表示单词名；值 total_word_count 是 IntWritable 类型对象，表示单词出现总次数。使用 MultipleOutputs 指定输出到 Mapper 输出键规定的流派对应文件中。

以下是各个类主要算法设计的伪代码：

```
public class WordFrequencyMapper {
    // 从缓存文件获取 track_id 到流派的映射
    Map<String, String> genre_map = setup();

    public void map(key, value=track_id,words) {
        if (genre_map.containsKey(track_id)) {
            // 遍历当前歌曲中所有词语
            for ((word, count) in words) {
                // 对每个词语，输出键值对 <<genre, word>, word_count>
                emit(genre + "," + word, count);
            }
        }
    }
}
```

WordFrequencyMapper 类伪代码

```
public class WordFrequencyReducer {
    protected void reduce(key=<genre, word>, values=[word_count1,
        word_count2, ...]) {
        int total_word_count = 0;

        // 统计单词出现总次数
        for (word_count : values) {
```

```
            total_word_count += tword_count;
        }

        // 输出键值对 <word, total_word_count> 到 genre 文件
        multipleOutputs.write(word, total_word_count, genre);
    }
}
```

WordFrequencyReducer 类伪代码

第二个 MapReduce 作业统计每种流派的歌曲数量。

Mapper 的输入是流派信息文件的一行，输出 <genre, genre_count>，表示流派及其计数。

Reducer 汇总 Mapper 同一流派的计数，输出 <genre, total_genre_count>。

最后使用比较得到歌曲数量最多的流派。

然后使用 wordcloud 库生成词云图。

```
# 读取文件并统计词频
word_freq = read_data()

# 生成词云图
wordcloud = WordCloud(width=800, height=400, background_color='white')
    .generate_from_frequencies(word_freq)
```

绘制词云图伪代码

5.2.5 流派信息挖掘

该部分任务包含一个 MapReduce 作业。

将预处理后的流派文件放在缓存作为全局共享文件。

Mapper 的输入是预处理后的歌曲文件 songs.txt 的一行，格式为 song_id, track_id, title, release, artist_id, artist_name, mode, energy, tempo, loudness, duration, danceability, year。

Mapper 输出的键值对是 <genre, features>，表示当前歌曲流派以及各项特征值。其中键 genre 是 Text 类型对象，表示流派名；值 features 是

Text 类型对象，格式为 energy, tempo, loudness, duration, danceability，表示各项特征。

Reducer 输出的键值对是 <genre, avg_features>，表示一个流派以及各项特征值的平均值。其中键 genre 是 Text 类型对象，表示流派名；值 features 是 Text 类型对象，格式为 energy, tempo, loudness, duration, danceability，表示各项特征的平均值。

以下是各个类主要算法设计的伪代码：

```
public class GenreInfoMapper {
    // 从缓存文件读取 track_id 和流派之间的映射关系
    Map<String, String> genre_map = setup();

    public void map(key, value=line) {
        // 从一行输入中获取各项特征信息
        track_id, energy, tempo, loudness, duration, danceability = line
            .split();

        if (genre_map.containsKey(track_id)) {
            // 获得当前歌曲的流派
            String genre = genre_map.get(track_id);

            // 拼接所需的特征值
            String features = energy + "," + tempo + "," + loudness + ","
                + duration + "," + danceability;

            // 输出键值对 <genre, features>
            emit(genre, feature);
        }
    }
}
```

GenreInfoMapper 类伪代码

```
public class GenreInfoReducer {
    protected void reduce(key=genre, values=[features1, features2,
        ...]) {
        // 计算所有特征的平均值
        avg_features = avg(features1, features2, ...);
```

```
5
6           // 输出键值对 <genre, avg_features>
7           write(genre, avg_features);
8       }
9   }
```

GenreInfoReducer 类伪代码

然后使用 matplotlib 库生成折线图。

```
1  # 读取文件并获得各流派特征
2  genres, energy, tempo, loudness, duration, danceability = read_data();
3
4  # 绘制不同属性的折线图
5  plt.plot(genres, energy, marker='+', linestyle='-', color=palette[0],
       label='Energy')
6  plt.plot(genres, tempo, marker='s', linestyle='-', color=palette[1],
       label='Tempo')
7  plt.plot(genres, loudness, marker='^', linestyle='-', color=palette
       [2], label='Loudness')
8  plt.plot(genres, duration, marker='o', linestyle='-', color=palette
       [3], label='Duration')
9  plt.plot(genres, danceability, marker='x', linestyle='-', color=
       palette[4], label='Danceability')
10
11 # 设置图表标题和标签等
12 plt.title('流派属性平均值', fontsize=20, fontweight='bold')
13 plt.xlabel('流派', fontsize=16)
14 plt.ylabel('属性平均值', fontsize=16)
15 ...
```

绘制折线图伪代码

5.3 歌曲情感分析

该部分任务包含训练模型和情感预测两部分。将一首歌曲的词频作为特征向量，进行朴素贝叶斯分类。已经含有情感标签的歌曲作为训练集，待预测的歌曲作为测试集。

5.3.1 模型训练

该部分任务包含一个 MapReduce 作业。

将情感标签文件 sentiment_train.txt 设为全局共享文件。

Mapper 的输入是歌词文件的一行，格式为 track_id, mxm_track_id, word1:count1, word2:count2 ...。

Mapper 输出的键值对是 <item, freq>，表示某个特征的频率。其中键 item 为 Text 类型对象，有两种格式：(1)某个情感类别；(2)label:index:value，表示某个情感类别某一维特征向量取值为特定值。值 freq 为 IntWritable 类型对象，表示某个类别或是某一维特征向量取值为特定值的频率。

Reducer 输出的键值对是 <item, total_freq>，即将 Mapper 输出的值累加。

以下是各个类主要算法设计的伪代码

```
public class SentiBayesTrainingMapper {
    // 从标签文件读取训练集样本对应分类
    Map<String, String> sentiment_map = setup();

    public void map(key, value=line) {
        // 从一行中获取 ID 和词频数据
        track_id, words = line.split();

        if (sentiment_map.containsKey(trackId)) {
            String label = sentimentMap.get(trackId);

            // 类别标签计数加一
            emit(label 1);

            for ((word, count) in words) {
                // 属性索引和对应属性值计数加一
                emit(label + ":" + word + ":" + count, 1);
            }
        }
    }
}
```

SentiBayesTrainingMapper 类伪代码

```
public class SentiBayesTrainingReducer {
    protected void reduce(key, values=[freq1, freq2, ...]) {
        int total_freq = 0;

        // 累加某个属性的频数
        for (freq in values) {
            total_freq += freq;
        }

        // 输出一个属性的总频数
        write(key, total_freq);
    }
}
```

SentiBayesTrainingReducer 类伪代码

5.3.2 情感预测

该部分任务包含一个 MapReduce 作业。

将训练的模型文件设为全局共享文件。

Mapper 的输入是测试集歌词文件的一行，格式为 track_id, mxm_track_id, word1:count1, word2:count2 ...。

Mapper 输出的键值对是 <track_id, label>，表示某个歌曲的情感预测分类。其中键 track_id 为 Text 类型对象，表示歌曲 ID；值 label 为 Text 类型对象，预测的情感类别。

Reducer 输出的键值对是 <track_id, label>，即简单输出 Mapper 的输出结果。

以下是各个类主要算法设计的伪代码

```
public class SentiBayesClassificationMapper {
    Map<String, Double> label_freq = setup(); // 存储情感标签频率
    Map<String, Double> attr_freq = setup() // 存储属性频率

    public void map(key, value=line) {
        // 从一行中获取 ID 和词频数据
        track_id, words = line.split();
```

```
8
9         Double maxF = -1000000.0; // 最大的概率值，用对数表示
10        String maxF_label = "none"; // 最大概率值对应的标签
11
12        foreach (label) {
13            Double FXYi = 0.0;
14            Double FYi = label_freq.get(label);
15            for ((word, count) in words) {
16                // 乘以当前特征属性频率
17                FXYi += Math.log(attr_freq.get(label:word:count));
18
19                // 除以当前类别总频率
20                FXYi -= Math.log(label_freq.get(label));
21            }
22
23            // 更新最大F值和对应的标签
24            if (FXYi + FYi > maxF) {
25                maxF = FXYi + FYi;
26                maxF_label = label;
27            }
28        }
29
30        // 输出键值对是 <track_id, label>
31        emit(track_id, maxF_label);
32    }
33 }
```

SentiBayesClassificationMapper 类伪代码

```
1 public class SentiBayesClassificationReducer {
2     protected void reduce(key=track_id, values=[label]) {
3         // 简单输出结果
4         write(track_id, label);
5     }
6 }
```

SentiBayesClassificationReducer 类伪代码

5.3.3 效果优化

针对朴素贝叶斯分类，主要的优化思路有三种：

1. 拉普拉斯平滑

   朴素贝叶斯分类对于出现零概率的情况的处理是直接忽视，在特征向量稀疏的情况下会带来较大误差。拉普拉斯平滑通过在总类别中添加一个词频，将零概率的情况全部归类到这个词频，确保即使是未出现的单词也有一个小的概率。

```
if (attr_freq.containsKey()) {
    FXYi += Math.log(attr_freq.get(label:word:count));
} else {
    // 将零概率的情况视作总情况数以外的一种情况
    FXYi += Math.log(1 / total_track);
}
```

情感预测拉普拉斯平滑优化

2. TF-IDF

   像“the”这样的常见词会在所有文档中出现很多次，而对于分类没有太大意义。逆文档频率（Inverse Document Frequency, IDF）可以用来减少常见词的影响，并提高稀有词的权重。

   可以通过实验一中统计 IDF 的方法，统计每个词语的 IDF，并将其作为求乘积时的权重。

```
if (attr_freq.containsKey()) {
    FXYi += Math.log(attr_freq.get(label:word:count)) * IDF(word);
}
```

情感预测 TF-IDF 优化

3. 反比权重

   朴素贝叶斯分类对训练集中出现次数多的类别有更高的倾向性。通过给每个类别的权重添加一个反比系数，可以平衡训练集中类别出现频率的差异，使得分类器在处理新数据时不至于过度偏向于频率较高的类别。

```
// 除以当前类别总频率
FXYi -= Math.log(label_freq.get(label));
```

情感预测反比权重优化

5.4 歌曲流派分类

该部分任务包含训练模型和流派预测两部分。将一首歌曲的作者名称和歌名的词频作为特征向量，进行朴素贝叶斯分类。已经含有流派标签的歌曲作为训练集，待预测的歌曲作为测试集。

5.4.1 模型训练

该部分任务包含一个 MapReduce 作业。

将流派标签文件 genres.txt 设为全局共享文件。

Mapper 的输入是 unique_tracks.txt 文件的一行，格式为 track_id<SEP>song_id<SEP>artist_name<SEP>song_name。

Mapper 输出的键值对是 <item, freq>，表示某个特征的频率。其中键 item 为 Text 类型对象，有两种格式：(1)某个流派类别；(2)label:index:value，表示某个流派类别某一维特征向量（某个单词）取值（出现次数）为特定值。值 freq 为 IntWritable 类型对象，表示某个类别或是某一维特征向量取值为特定值的频率。

Reducer 输出的键值对是 <item, total_freq>，即将 Mapper 输出的值累加。

以下是各个类主要算法设计的伪代码

```
public class GenreBayesTrainingMapper {
   // 读取 ID 和流派的映射关系
   Map<String, String> genre_map = setup();

   public void map(key, value=line) {
      // 从一行中读取信息
      track_id, song_id, artist_name, song_name = line.split();

      if (genre_map.containsKey(track_id)) {
```

```
            String label = genre_map.get(track_id);

            // 类别标签计数加一
            emit(label, 1);

            for (word in artist_name and song_name) {
                // 词频计数加一
                emit(label + ":" + word, 1);
            }
        }
    }
}
```

GenreBayesTrainingMapper 类伪代码

```
public class GenreBayesTrainingReducer {
    protected void reduce(key, values=[freq1, freq2, ...]) {
        int total_freq = 0;

        // 累加某个属性的频数
        for (freq in values) {
            total_freq += freq;
        }

        // 输出一个属性的总频数
        write(key, total_freq);
    }
}
```

GenreBayesTrainingReducer 类伪代码

5.4.2 流派预测

该部分任务包含一个 MapReduce 作业。

将训练的模型文件设为全局共享文件。

Mapper 的输入是测试集 unique_tracks.txt 文件的一行，格式为 track_id<SEP>song_id<SEP>artist_name<SEP>song_name。

Mapper 输出的键值对是 <track_id, label>，表示某个歌曲的流派预测分类。其中键 track_id 为 Text 类型对象，表示歌曲 ID；值 label 为 Text 类型对象，预测的流派类别。

Reducer 输出的键值对是 <track_id, label>，即简单输出 Mapper 的输出结果。

以下是各个类主要算法设计的伪代码

```
public class GenreBayesClassificationMapper {
    Map<String, Double> label_freq = setup(); // 存储情感标签频率
    Map<String, Double> attr_freq = setup() // 存储属性频率

    public void map(key, value=line) {
        // 从一行中获取 ID 和词频数据
        track_id, words = line.split();

        Double maxF = -1000000.0; // 最大的概率值，用对数表示
        String maxF_label = "none"; // 最大概率值对应的标签

        foreach (label) {
            Double FXYi = 0.0;
            Double FYi = label_freq.get(label);
            for ((word, count) in artist_name, song_name) {
                // 乘以当前特征属性频率
                FXYi += Math.log(attr_freq.get(label:word:count));

                // 除以当前类别总频率
                FXYi -= Math.log(label_freq.get(label));
            }

            // 更新最大F值和对应的标签
            if (FXYi + FYi > maxF) {
                maxF = FXYi + FYi;
                maxF_label = label;
            }
        }

        // 输出键值对是 <track_id, label>
```

```
        emit(track_id, maxF_label);
    }
}
```

GenreBayesClassificationMapper 类伪代码

```
public class GenreBayesClassificationReducer {
    protected void reduce(key=track_id, values=[label]) {
        // 简单输出结果
        write(track_id, label);
    }
}
```

GenreayesClassificationReducer 类伪代码

5.4.3 效果优化

除了上一小节情感预测任务中提到的三种优化方法，此处还有一种优化方法，将作者名称的词频与歌曲名称的词频分开计算，从而更准确地描述一个歌曲的特征向量。

```
for (word in artist_name) {
    // 标记艺术家名称中词频计数加一
    emit(label + ":a:" + word, 1);
}

for (word in song_name) {
    // 标记歌曲名称中词频计数加一
    emit(label + ":s:" + word, 1);
}
```

流派预测优化 - 模型训练

```
// 乘以艺术家名字中词语频率
for ((word, count) in artist_name) {
    if (attrFreq.containsKey(label:a:word:count)) {
        FXYi += Math.log(attr_freq.get(label:a:word:count));
    }
    FXYi -= Math.log(label_freq.get(label));
```

```
}

// 乘以歌曲名字中词语频率
for ((word, count) in song_name) {
    if (attrFreq.containsKey(label:s:word:count)) {
        FXYi += Math.log(attr_freq.get(label:s:word:count));
    }
    FXYi -= Math.log(label_freq.get(label));
}
```

流派预测优化 - 流派预测

5.5 可视化系统设计

可视化系统设计包含前端和后端设计，前端渲染网页，向后端上传文件、调用后端服务以及从后端获取结果；后端提供计算服务。

5.5.1 前端设计

前端使用 HTML、CSS 和 JavaScript 渲染网页，通过 Ajax 向后端上传文件、调用后端服务以及从后端获取结果。

```
$.ajax({
    url: `http:\\${ip}:5000\status\genre`,
    type: 'GET',
    success: function(response) {
        // 更新状态
    }
});
```

从后端获取运行状态

```
$.ajax({
    url: `http:\\${ip}:5000\result\anal\popularity`,
    type: 'GET',
    success: function(response) {
        // 处理结果
    }
```

```
});
```

从后端获取结果

```
var xhr = new XMLHttpRequest();
xhr.open('POST', `http:\\${ip}:5000\process\genre`, true);

xhr.onreadystatechange = function() {
    if (xhr.readyState === XMLHttpRequest.DONE) {
        if (xhr.status === 200) {
            // 运行成功
        } else {
            // 运行失败
        }
        resultBox.classList.remove('hidden');
    }
};

xhr.send();
```

调用后端计算服务

5.5.2 后端设计

后端使用 Flask 配置服务器，对前端不同路由的请求运行相应的 JAR 包，返回相应结果。

```
@app.route('/process/anal/duration', methods=['POST'])
def process_anal_duration():
    try:

        # Hadoop 环境下运行 JAR 包
        subprocess.run([
            "hadoop", "jar",
            os.path.join("jars", "Duration.jar"),
            os.path.join(hdfs_path, "SongDataset"),
            os.path.join(hdfs_path, "result")
        ])

```

```
        # 运行 Python 进行绘图
        subprocess.run([
            "python3",
            os.path.join(os.getcwd(), "src", "task23.py")
        ])

        status_anal_duration = "Successful!"
        return jsonify({'message': 'Successful!'}), 200
    except Exception as e:
        # 异常处理
```

运行一个任务

6 输入文件和输出结果说明

这一节里我将给出各阶段输入输出片段及格式说明。

1. **数据集预处理：**

(1) 处理歌曲元数据：

- 输入为一系列.h5 文件，位于 data/SongDataset/songs。
- 输出为 songs.txt，位于 data/SongDataset，每一行格式为 song_id, track_id, title, release, artist_id, artist_name, mode, energy, tempo, loudness, duration, danceability, year。以下是输出文件的一个片段：

```
SOAAEHR12A6D4FB060,TRAYPFH128E07937C3,Slaves & Bulldozers,Badmotorfinger,AR5N8VN1187FB37A4E,Soundgarden,1,0,131.272,-9.306,415.81669,0,1991
SOAAFUV12AB018831D,TRAMPCN12903CC4E5B,Where Do The Children Play? (LP Version),Resistance,AR5ZGC11187FB417A3,Big Mountain,1,0,149.169,-7.885,216
SOAAHZS12A8C143A21,TRAFUGB128F92EDEDC,Rakkauden veteraani,Viimeiseen pisaraan,ARV69FI1187B9AA14F,Kari Tapio,1,0,103.244,-8.419,224.7571,0,2009
SOAANKE12A8C13CF5C,TRBDLDS128F429C171,Sam und Bo_ N.E.O.,So Deluxe So Glorious,ARB0WNP1187B9AC4AB,Samy Deluxe,0,0,114.072,-6.08,247.01342,0,200
SOAAQAB12A8AE4769F,TRAPXTV128F423A363,Wrote For Luck (12" - Remastered version),Bummed,ARPATQ21187B9ACAFC,Happy Mondays,0,0,101.831,-9.616,342.5
SOAASSD12AB0181AA6,TRAQHPK128F92E339B,Song From Moulin Rouge,16 Most Requested Songs Of The 1950s. Volume Two,ARKXLIJ1187B9A4C54,Percy Faith & H
SOAATRN12A6310E897,TRADSUA128C7196C7C,New York,Electrical Storm,ARUJ5A41187FB3F5F1,U2,0,0,125.088,-7.165,343.27465,0,2000
```

图 1: 处理歌曲元数据输出片段

(2) 处理歌词数据：

- 输入为一系列.txt 文件，位于 data/SongDataset/lyrics，每一行格式为 track_id, mxm_track_id，word_id1:count1, word_id2:count2 ...。以下是输入文件的一个片段：

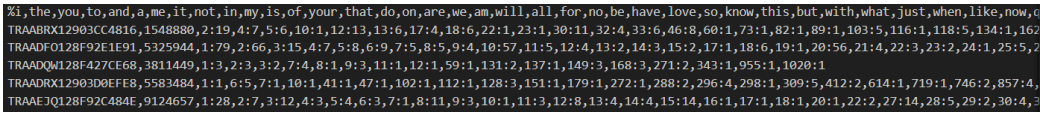


图 2: 处理歌词数据输入片段

- 输出为 lyrics.txt，位于 data/SongDataset，每一行格式为 track_id, mxm_track_id, [(word1:count1), (word2:count2) ...]。以下是输出文件的一个片段：

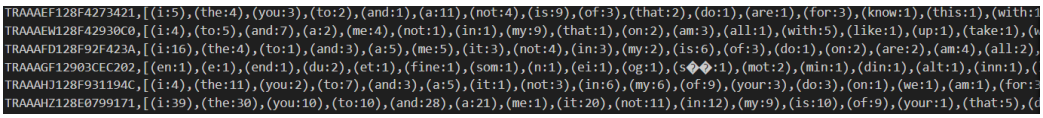


图 3: 处理歌词数据输出片段

(3) 处理流派数据

- 输入为一系列.txt 文件，位于 data/SongDataset/genres，每一行格式为 track_id \t genre。以下是输入文件的一个片段：

图 4: 处理流派数据输入片段

- 输出为 genres.txt，位于 data/SongDataset，每一行格式为 track_id, genre。以下是输出文件的一个片段：

图 5: 处理流派数据输出片段

(4) 处理用户数据

- 输入为一系列.txt 文件，位于 data/SongDataset/users，每一行格式为 user_id \t song_id \t play_count。以下是输入文件的一个片段：

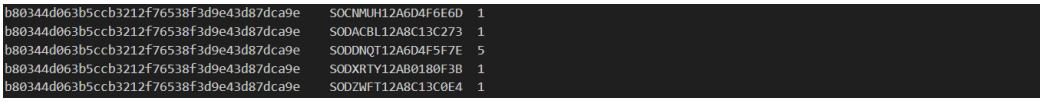


图 6: 处理流派数据输入片段

- 输出为 users.txt，位于 data/SongDataset，每一行格式为 user_id, song_id, play_count。以下是输出文件的一个片段：

图 7: 处理流派数据输出片段

(5) 清洗数据

- 输入为预处理生成的 songs.txt、lyrics.txt、genres.txt 和 users.txt。
- 输出为 filtered_songs.txt、filtered_lyrics.txt、filtered_genres.txt 和 filtered_users.txt，位于 data/SongDataset，格式与预处理生成的对应文件一致。

2. **音乐数据特征分析：**

(1) 歌曲流行度分析

- 输入为预处理后生成的 filtered_songs.txt 和 filtered_users.txt，位于 data/SongDataset。
- 输出为 task21.txt，位于 data/result，每一行格式为 song_name, play_count。以下是输出文件：

图 8: 歌曲流行度分析输出

(2) 用户喜好分析

- 输入为预处理后生成的 filtered_songs.txt 和 filtered_users.txt，位于 data/SongDataset。
- 输出为 task22.txt，位于 data/result，格式为 user_id, artist_id。以下是输出文件：

图 9: 用户喜好分析输出

(3) 歌曲播放时长分布

- 输入为预处理后生成的 filtered_songs.txt，位于 data/SongDataset。
- 输出包括一个文本文件 task23.txt 和一个图片 task23.png，位于 data/result，其中 task23.txt 每一行格式为 gap, song_count。以下是输出文件：

图 10: 歌曲播放时长分布输出

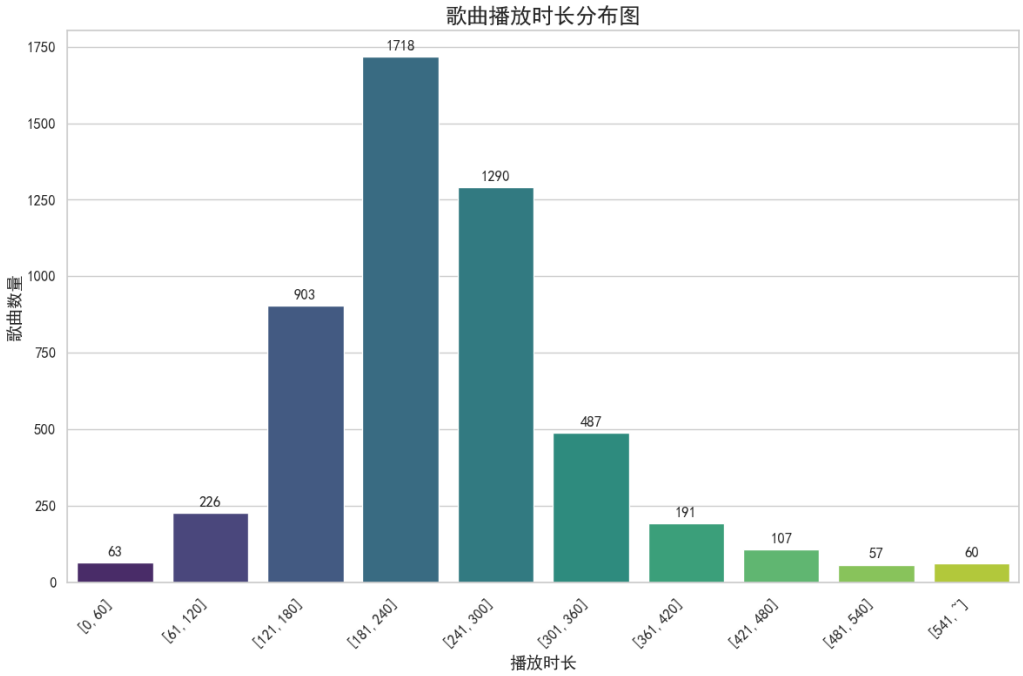


图 11: 歌曲播放时长分布直方图

(4) 歌词词频分析

- 输入为预处理后生成的 filtered_lyrics.txt 和 filtered_genres.txt，位于 data/SongDataset。
- 输出包括一个目录 task24，其中包含 15 个 <genre>.txt 文件，以及一个图片 task24.png，位于 data/result，其中 <genre>.txt 每一行格式为 word, word_count。以下是输出文件的片段：

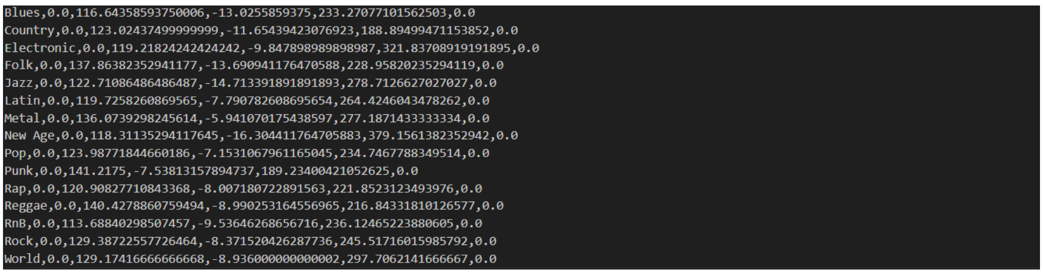
```
sunday,26
sung,1
sunlight,1
sunni,8
sunset,1
```

图 12: 歌词词频分析输出片段

图 13: 歌词词频分析词云图

(5) 流派信息挖掘

- 输入为预处理后生成的 filtered_songs.txt 和 filtered_genres.txt，位于 data/SongDataset。
- 输出包括一个文本文件 task25.txt 和一个图片 task25.png，位于 data/result，其中 task25.txt 每一行格式为 genre, energy, tempo, loudness, duration, danceability。以下是输出文件：



```
Blues,0.0,116.64358593750006,-13.0255859375,233.27077101562503,0.0
Country,0.0,123.02437499999999,-11.65439423076923,188.89499471153852,0.0
Electronic,0.0,119.21824242424242,-9.847898989898987,321.83708919191895,0.0
Folk,0.0,137.86382352941177,-13.690941176470588,228.95820235294119,0.0
Jazz,0.0,122.71086486486487,-14.713391891891893,278.7126627027027,0.0
Latin,0.0,119.7258260869565,-7.790782608695654,264.4246043478262,0.0
Metal,0.0,136.0739298245614,-5.941070175438597,277.1871433333334,0.0
New Age,0.0,118.31135294117645,-16.304411764705883,379.1561382352942,0.0
Pop,0.0,123.98771844660186,-7.1531067961165045,234.7467788349514,0.0
Punk,0.0,141.2175,-7.53813157894737,189.23400421052625,0.0
Rap,0.0,120.90827710843368,-8.007180722891563,221.8523123493976,0.0
Reggae,0.0,140.4278860759494,-8.990253164556965,216.84331810126577,0.0
RnB,0.0,113.68840298507457,-9.53646268656716,236.12465223880605,0.0
Rock,0.0,129.38722557726464,-8.371520426287736,245.51716015985792,0.0
World,0.0,129.17416666666668,-8.936000000000002,297.7062141666667,0.0
```

图 14: 流派信息挖掘输出

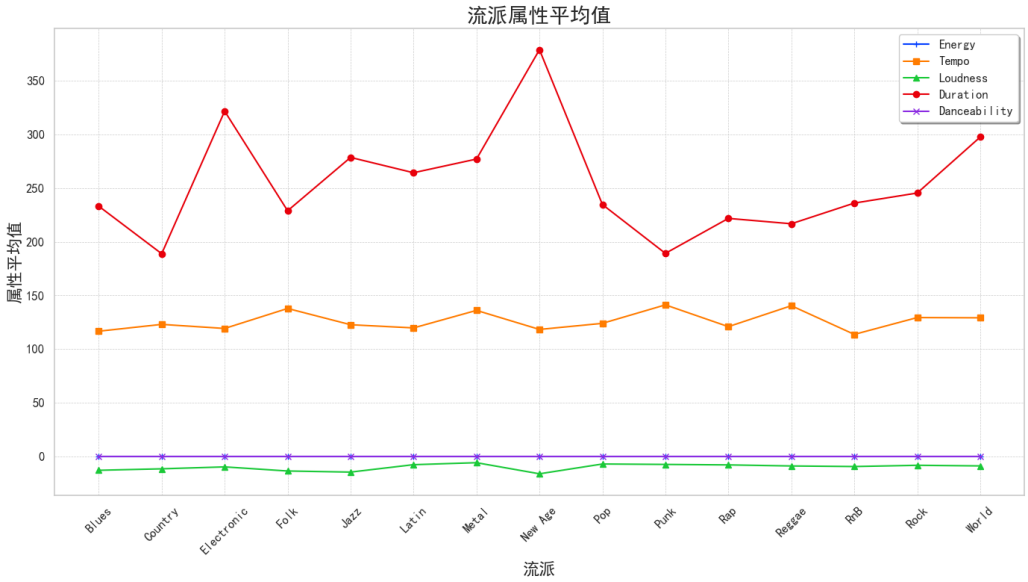


图 15: 流派信息挖掘折线图

3. **歌曲情感分析：**

- 输入包括样本标签 sentiment_train.txt 和歌词数据 lyrics/*.txt，位于 data/SongDataset。其中 sentiment_train.txt 每一行格式为 track_id, sentiment。以下是输入文件的一个片段：

```
TRAAAEW128F42930C0,negative
TRAAAFD128F92F423A,neutral
TRAAAGF12903CEC202,positive
TRAAAHJ128F931194C,positive
TRAAAHZ128E0799171,positive
```

图 16: 歌曲情感分析输入

- 输出为 task3.txt ，位于 data/result，每一行格式为 track_id, sentiment。以下是输出文件的一个片段：

```
TRAAETD128F92F89A3,negative
TRAAEWC128F427D19B,neutral
TRAAFDU128F426E91A,positive
TRAAFOY128F146CC17,positive
TRAAGMC128F4292D0F,positive
```

图 17: 歌曲情感分析输出

4. **歌曲流派分类：**

- 输入包括样本标签 genres.txt 和待分类数据 unique_tracks.txt，位于 data/SongDataset。其中 unique_tracks.txt 每一行格式为 track_id<SEP> song_id<SEP>artist_name<SEP>song_name。以下是输入文件的一个片段：

```
TRMMMCJ128F930BFF8<SEP>SOQQESG12A58A7AA28<SEP>Danny Diablo<SEP>Cold Beer feat. Prince Metropolitan
TRMMMBW128F4260CAE<SEP>SOMPVQB12A8C1379BB<SEP>Tiger Lou<SEP>Pilots
TRMMMXI128F4285A3F<SEP>SOGPCJI12A8C13CCA0<SEP>Waldemar Bastos<SEP>N Gana
TRMMMKI128F931D80D<SEP>SOSDCFG12AB0184647<SEP>Lena Philipsson<SEP>006
TRMMMUT128F42646E8<SEP>SOBARPM12A8C133DFF<SEP>Shawn Colvin<SEP>(Looking For) The Heart Of Saturday
```

图 18: 歌曲流派分类输入

- 输出为 task4.txt ，位于 data/result，每一行格式为 track_id, genre。以下是输出文件的一个片段：

```
TRAAAGW12903CC1049,Punk
TRAAAHD128F42635A5,Rock
TRAAAHE12903C9669C,Reggae
TRAAAHJ128F931194C,Metal
TRAAAHO128F423BBE3,Rock
```

图 19: 歌曲流派分类输出

7 实验结果说明和分析

1. **数据集预处理：**

   经过对数据格式预处理，得到歌曲信息 10000 行、歌词信息 237662 行、流派信息 133981 行以及用户信息 48373586 行。

   经过数据清洗，删除与其它数据没有关联的行后，剩余歌曲信息 5102 行、歌词信息 60116 行、流派信息 59336 行以及用户信息 772661 行。

2. **音乐数据特征分析：**

   (1) 歌曲流行度分析

   统计了播放次数最多的 10 首歌曲及其播放次数，其中播放次数最多的歌曲是 Nothin' On You [feat. Bruno Mars] (Album Version)，被播放了 87366 次

(2) 用户喜好分析

听歌次数最多的用户是 a3f5db66731126b173610095b831a216a0e27da4，听了 772 次。被播放次数最多的艺术家是 ARR3ONV1187B9A2F59，被播放了 110308 次。

(3) 歌曲播放时长分布

短时长的音乐数量较少，可能是一些简短的片段或过渡音轨，常见于专辑的开场白或过场音乐。

较长时长的音乐数量较少，可能是一些交响乐、某些概念专辑中的长篇曲目，或者现场演出版本。

中等时长音乐（121 - 300 秒）音乐数量最多，尤其是 181 - 240 秒的音乐，数量达到了 1718 首。这个时长段的音乐通常是常见的歌曲时长，大约在 3 到 5 分钟之间。

(4) 歌词词频分析

通过对不同流派的词频进行统计和分析，对 15 个流派归纳出以下特征：

- Blues 包含了一些常见的情感和个人经历的词汇，如“babe”,“love”,“down”，反映了其常见的情感表达和主题。
- Country 包含了一些常见的人称代词和情感词汇，如“you”,“love”,“am”,“she”，反映了其乡村和民俗音乐的特点
- Electronic: 包含了一些情感和状态的词汇，如“feel”,“love”,“up”，反映了涌动的情感。
- Folk 包含了一些较为传统和故事性的词汇，如“oh”,“down”,“come”，反映了其故事性和地方性的特点。
- Jazz 包含了一些舒缓情感表达相关的词汇，如“day”,“see”,“day”，体现了其即兴和慵懒音乐性质。
- Latin 包含了一些西班牙语的常见词汇，如“que”, “la”, “mi”，反映了其文化和语言的影响。
- Metal 包含了一些强调生活和情感的词汇，如“life”,“never”,“down”，表达对社会和自我状态的探索。

- New Age 包含了一些情感和灵性的词汇,如“love”,“way”,“feel”,反映了其寻求内心特点。
- Pop 包含了一些普遍和流行的词汇, 如 “babe”, “oh”, “feel”,反映了对情感和个人体验的表达。
- Punk 包含了一些反叛和社会意识的词汇,如“out”,“get”,“one”,反映了其对现实的批判和对自由的强调。
- Rap 包含了一些俚语, 如 “n-word”, “yo”, “caus”, 表现了嘻哈文化。
- Reggae 包含了一些情感和社会意识的词汇,如“dem”,“man”,“say”,以及对人生和社会状态的反思。
- RnB 包含了一些强调情感和个人体验的词汇,如“babe”,“yeah”,“let”,反映了对情感的感受。
- Rock 包含了一些强调强烈情感的词汇,如“oh”,“come”,“out”,反映了其狂热的感受。

(5) 流派信息挖掘

流派之间在节奏、响度和持续时间上有明显的差异。

New Age 和 Electronic 的持续时间最长, 可能因为这类音乐往往用于长时间背景音乐。Punk 和 Country 的持续时间较短, 可能因为这类音乐的曲风和表演方式决定了较短的歌曲长度。

Metal 的响度最高, 而 New Age 的响度最低, 这可能反映了不同流派的演奏风格和激烈程度。

节奏最快的是 Punk 和 Reggae, 而最慢的是 RnB 和 New Age, 反映了有些流派比较激昂, 有些则比较舒缓。

3. **歌曲情感分析:**

为了测试分析的准确性, 将 sentiment_train.txt 中标记的音乐按 4:1 的比例划分成训练集和测试集, 并测试在不同优化选项下的预测准确率, 结果如下表所示:

拉普拉斯平滑	TF-IDF	反比权重	预测准确率
否	否	否	74.52%
是	否	否	74.29%
否	是	否	74.91%
是	是	否	74.91%
否	否	是	74.87%
是	否	是	74.86%
否	是	是	74.93%
是	是	是	74.91%

表 1: 实验在三个优化选项分别取是或否时所有组合的准确率

根据结果，使用朴素贝叶斯分类算法对情感进行预测的准确率在 74% - 75% 左右。这三种优化选项对情感预测任务效果不明显，可能的原因如下：

- 拉普拉斯平滑通过给未出现的样本一个极小的频率，避免概率为零的情况，主要应对十分稀疏的数据。本实验中词频向量并不稀疏，因此效果有限。
- TF-IDF 优化通过给较少出现的词语添加更高的权重，避免因为高频词语过多导致区分不明显的问题，主要用于处理样本在许多维度上相似度过高的情况。根据之前对不同流派词频的分析，本实验中不同歌曲的词频有显著差异，因此 TF-IDF 效果有限。
- 反比权重优化通过平衡各个类别的样本，提升朴素贝叶斯模型在少数类别上的分类准确性。本实验中不同类别的样本数量相差仅仅几倍，数量级没有明显差别，不会因为不平衡导致少数类别难以被选择，因此反比权重的优化效果有限。

4. **歌曲流派分类：**

将 genres_.txt 中标记的音乐按 4:1 的比例划分成训练集和测试集，并测试在不同优化选项下的预测准确率，结果如下表所示：

拉普拉斯平滑	TF-IDF	反比权重	分离艺术家名和歌名	预测准确率
否	否	否	否	0.22%
是	否	否	否	79.37%
否	是	否	否	0.19%
是	是	否	否	71.36%
否	否	是	否	0.09%
是	否	是	否	67.46%
否	是	是	否	0.09%
是	是	是	否	69.37%
否	否	否	是	0.25%
是	否	否	是	82.13%
否	是	否	是	0.23%
是	是	否	是	75.72%
否	否	是	是	0.10%
是	否	是	是	78.55%
否	是	是	是	0.10%
是	是	是	是	74.11%

表 2: 实验在四个优化选项分别取是或否时所有组合的准确率

根据结果，使用朴素贝叶斯分类算法对流派进行预测的准确率在 70% - 80% 左右。在这四种优化方案中，拉普拉斯平滑的提升最为显著，其次分离艺术家名和歌名也带来比较明显的准确率提升，分析如下：

- 拉普拉斯平滑用来避免概率为零的情况，主要应对十分稀疏的数据。本实验中词频十分稀疏，因此拉普拉斯平滑是非常必要的。
- 分离艺术家名和歌名后，预测准确率有显著提升。说明艺术家名字和歌名的词频分布存在较大的差异，不适合当成一个整体一起分析。
- 本实验中不同歌曲的词频有显著差异，不存在因为高频词语过多导致区分不明显的问题，因此 TF-IDF 效果有限。

- 本实验中不同类别的样本数量相差仅仅几倍，不会因为不平衡导致少数类别难以被选择，因此反比权重的优化效果有限。

5. **可视化系统设计：**

   实现了一个网页用户界面，用户可以在这里上传数据集文件的压缩包，并实现一键预处理、分析、情感归类和流派预测。同时，用户也可以在网页端看到即时运行状态和运行结果，并且可以下载每一个步骤运行结果的压缩包。以下为可视化系统的部分效果：

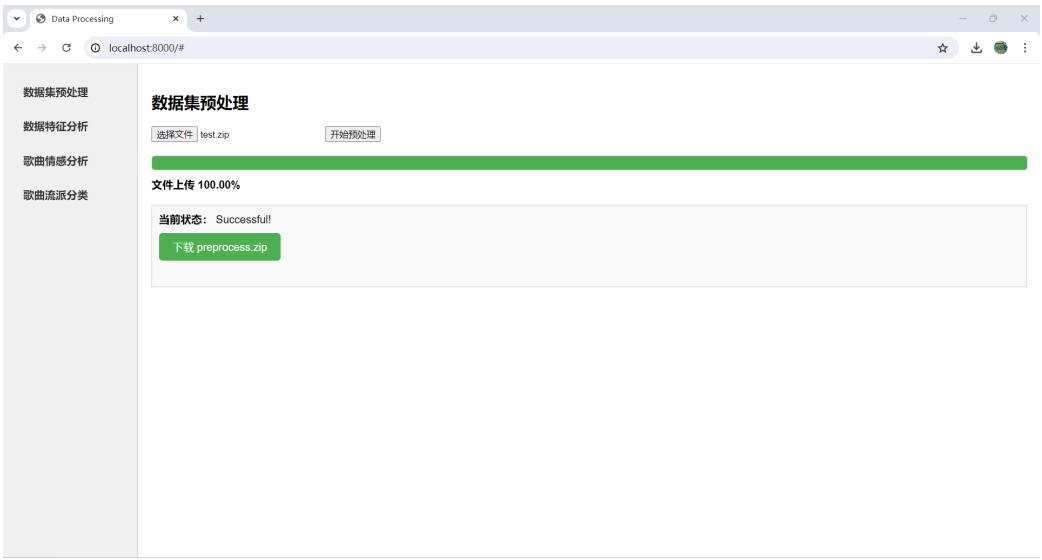


图 20: 可视化系统预处理效果

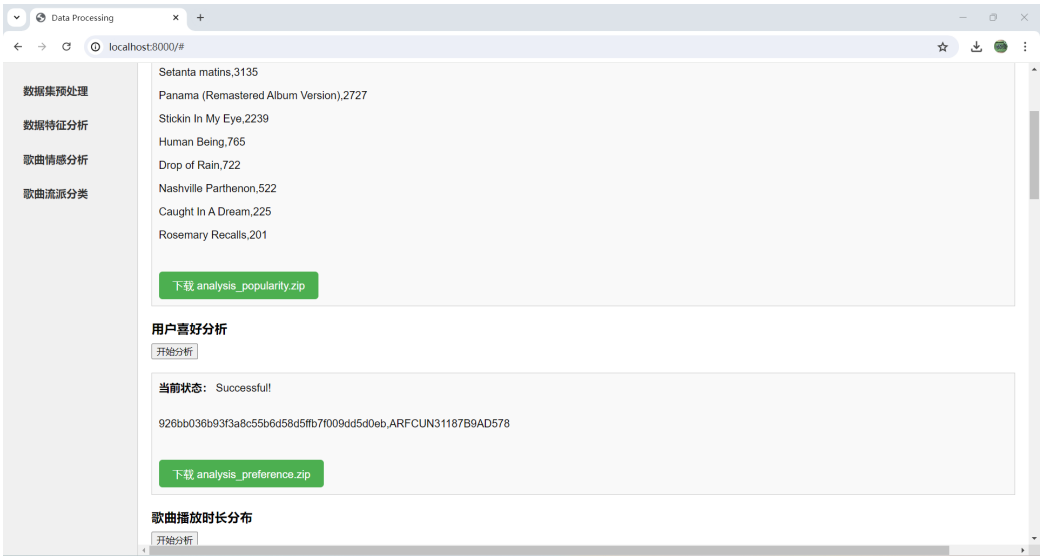


图 21: 可视化系统分析效果 - 1

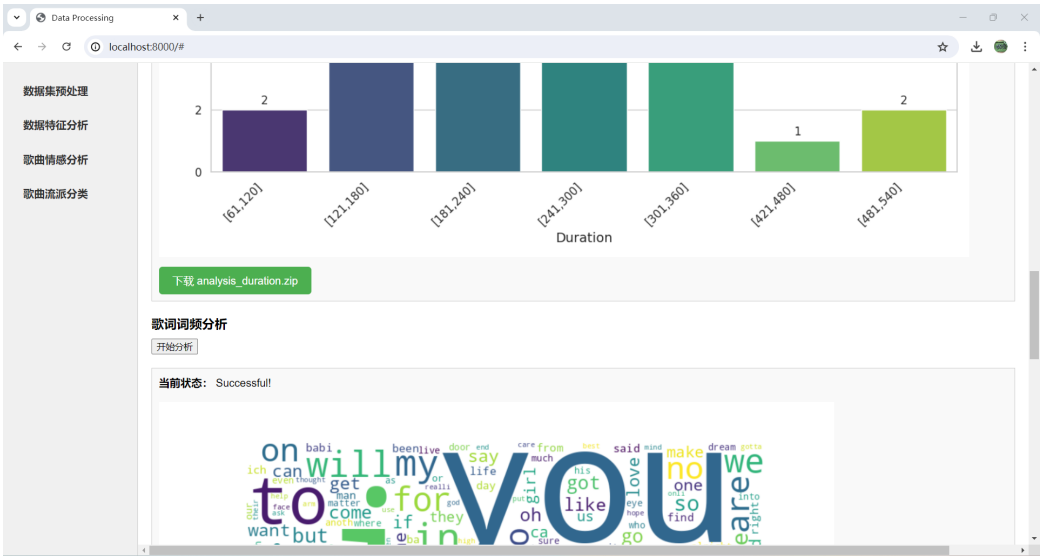


图 22: 可视化系统分析效果 - 2

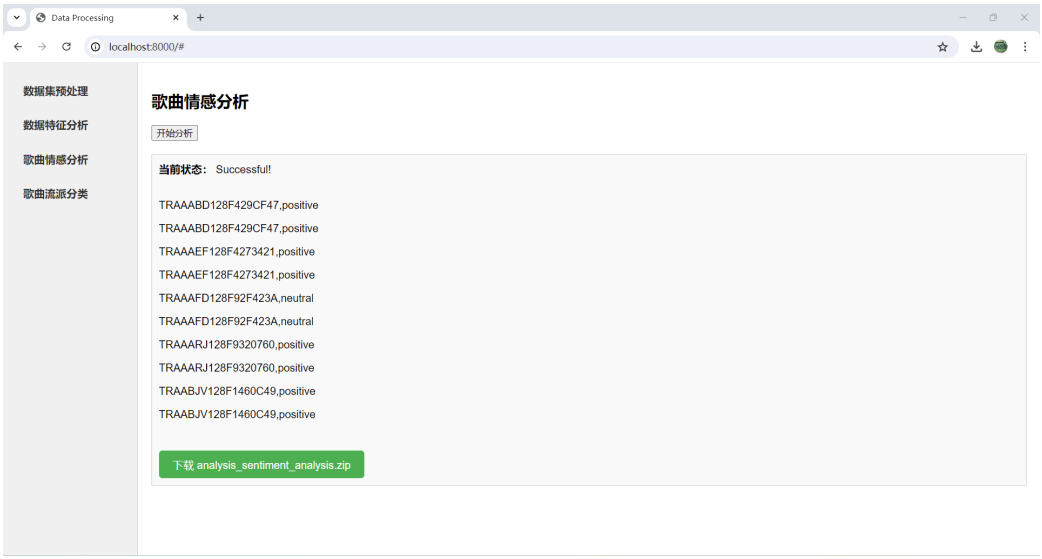


图 23: 可视化系统情感分析效果

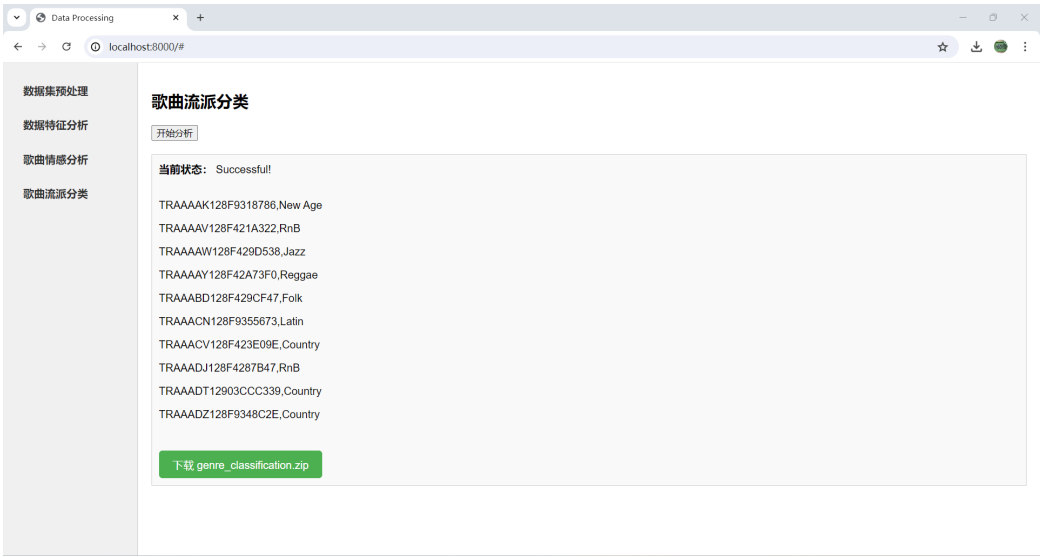


图 24: 可视化系统流派分类效果

8 总结和展望

本文通过对音乐信息的大数据分析，重点研究了歌曲情感分析和流派分类任务的处理与应用。通过实验和分析，得出了几个结论：

1. **数据预处理的重要性**：清理和预处理大规模音乐数据对保证分析准确性至关重要。通过去除无关数据和处理异常值，获得了更具代表性的数据集，提升了后续分析的有效性。

2. **音乐数据特征分析**：对歌曲流行度、用户偏好、歌曲时长和歌词特征等进行深入分析。这些分析有助于了解公众对音乐的喜爱程度和表现形式，以及揭示基于这些特征不同音乐流派之间的显著差异。

3. **歌曲情感分析**：利用朴素贝叶斯算法进行情感分析，实现对音乐情感的准确预测。不同的优化选项显示出算法准确率的稳定波动，表明特定的优化并不显著提高处理音乐文本数据时的性能。

4. **歌曲流派分类**：同样使用朴素贝叶斯算法，成功对音乐流派进行分类。拉普拉斯平滑和分离艺术家名字与歌曲标题作为优化策略显著提高了分类准确性，特别是在处理稀疏数据和区分艺术家与歌曲标题时。

5. **可视化系统设计与应用**：为了更好地展示分析结果并提供用户友好的界面，设计并实现了基于 Web 的可视化系统。该系统允许用户上传数据、运行分析，并查看结果，从而实现了数据分析的可视化和自动化。

展望未来，当前工作有几个扩展和改进的方向：

- **算法优化与模型集成**：尝试其他机器学习算法如支持向量机（SVM）或深度学习模型，以提高情感分析和流派分类的准确性。

- **跨模态数据集成**：结合音频特征分析与文本分析，实现更全面的音乐数据分析，例如将情感与音乐声学特征相关联。

- **个性化推荐系统**：基于用户偏好和情感分析结果开发个性化音乐推荐系统，提升用户体验和参与度。

- **实时数据处理与应用**：探索实时数据流分析技术，为音乐行业的决策者提供及时的见解和反馈。

通过持续的研究和技术创新，我们可以更好地利用大数据分析技术深入理解音乐背后的模式和趋势，为音乐产业的发展提供更多可能性和机会，同时增强音乐爱好者的体验。

9 参考文献

1. Analytics Vidhya, ”Naive Bayes Explained,” 2020.

2. Stack Overflow, ”How to use TFIDF with Naive Bayes,” 2016.