Personal annotation of sounds using subjective tags

Andreas Näsman

Master's thesis in computer engineering
Supervisor: Professor Johan Lilius
Faculty of Science and Engineering
Information Technologies
Åbo Akademi University
April 30, 2020

Abstract

Sound designers use extensive sound libraries together with associated metadata management tools in their everyday life. However, there exists no standard telling how to organize or how to label the audio files. The maintainers of the libraries annotate the sounds with descriptive labels that specify the source of the audio, the materials used, or some other physical property about the samples. As this process is both an error-prone and a time-consuming task, research has taken place, producing automatic sound identification and categorization methods. An area far less covered is the consideration and application of subjectivity in these systems.

This thesis explores a theoretical and practical solution of subjective audio tagging in the form of a developed program, focusing on the individual. The proposed system demonstrates an application capable of finding similar groups of sound according to a specific user's perception and automatic tagging of these sounds. Continuous improvement applies to the system with the help of unsupervised machine learning run recursively. Publicly available audio packs simulate segregation of sound according to different persons, which help test and evaluate the proposed solution with various example use cases.

Keywords*: audio identification, audio classification, machine learning, metadata management, sound design, sound effects, tagging ontology, tagging recommendation*

Acknowledgment

Firstly, I would like to thank Professor Johan Lilius for functioning as my supervisor during the writing process and for suggesting the topic in the first place. Secondly, I thank Adjunct Professor Marina Waldén for helping me with administrative tasks and supplying deadlines for the project. Also, for providing reference material and general help when the completion of this thesis seemed far away. I would also like to thank Åbo Akademi University for providing accommodation and maintaining an excellent environment for studying and writing. Without those, I could not have found the focus to finish this project. Lastly, I give a warm thank you and a hug to my girlfriend, Melissa, for always being there and giving me emotional support ♡.

Contents

List of Abbreviations

AI	Artificial Intelligence
GUI	Graphical User Interface
IoSR	Institute Of Sound Recording
JSON	JavaScript Object Notation
ML	Machine Learning
MTG	Music Technology Group
SFX	Sound Effect
UI	User Interface

List of Figures

List of Listings

List of Tables

1 Introduction

Sound plays a more vital role than ever in today's media-influenced world. Sound effects (SFX), jingles, and other snippets of sound bring out a devised experience among listeners, either separately or together in forms of larger unities. Different applications may require distinct approaches and specializations to achieve the desired sound landscape, but the methods used for developing the needed sounds are mostly the same.

Sound design is the practice of bringing sounds to life intended for a specific production [5, p. 1]. Many sound designers choose to utilize sound libraries with predefined samples to speed up the work process. Since the size of the sound libraries can be quite massive, various management tools are available to improve the accessibility of the libraries, making it easier to organize and find wanted sounds. However, both the libraries themselves and the accompanying tools have flaws and drawbacks in the way they are structured and how they function.

1.1 Motive

The purpose of this thesis is to suggest an improvement to the current management tools for sound libraries. A shared shortcoming in all of the tools is the lack of adaptability to different users' understanding and judgment regarding sound perception. The proposed solution is a flexible and self-improving system empowering personal organization of sounds using subjective tags. To explain the approach more clearly, the author of the thesis has implemented a prototype application of the system that demonstrates a practical approach with minimal manual tagging effort needed from the user. The program establishes a frame of similar sounds according to a user's perception and eases the tagging process with automatic labeling. The solution forms around the hypothesis that the comprehension of sound differs enough among people that a system like this is feasible [19, p. 2]. Existing categorization systems and management tools can hopefully adopt the program and embellish it, therefore improving the craft for sound designers making use of them.

1.2 Thesis outline

This thesis consists of three introductory chapters, including the current one. The present chapter provides a brief background on the chosen subject, defines the purpose of the thesis, and maps out the overall structure of the paper. Chapter 2 extends on the background of the topic, describes relevant problems, and reviews related work. Chapter 3 provides detailed information on the methods applied to achieve the proposed solution.

Chapter 4 and chapter 5 explain the implemented application and how it performs in different scenarios. The former illustrates the design of the program and demonstrates the underlying code, and the latter specifies the evaluation process and measures the system using four example use cases.

Lastly, two chapters wrap up the thesis in the form of a discussion about the project in chapter 6 and a concluding summary, plus analysis on future work, in chapter 7.

2 Background

In the first part of this chapter, the content provides more background on the topic and exhibits the current state of the subject. The second section briefly presents associated literature and how it relates to the proposition in this thesis.

2.1 Problem description

The art of sound design involves planning, creating, and acquiring sounds destined for media productions or other purposes [5, p. 1]. Skillfully implemented audio integrates so tightly with other content forms, such as video and animation, that it often goes unnoticed [7, p. 1]. Sound is nevertheless a vital part of audiovisual production – movies, games, television programs, and the like – where up to 75 % of the SFX are part of the post-production stage [11, p. 1].

In general terms, there are two ways of obtaining samples of sound. The first approach is to reconstruct audio using objects that imitate the wanted sound. The props used in this routine, called *foley*, can be anything able to mimic the sonic properties of the desired sample [29, p. 1]. A sound designer can also generate sounds electronically for specific purposes, where programs can assist in the creation process [28]. These composition methods can be quite tedious and expensive since the sound producer needs time and resources when creating audio from scratch.

The second approach has the sound designer using sound libraries containing various predefined example sounds. These SFX libraries can contain thousands of sounds, often with variations of a particular sound through changes in intensity, material, duration, or similar. By mixing the sounds, the designer is also able to blend tracks to create new samples in endless combinations. This possibility can be particularly useful to, e.g., increase the dramatic effect of a sound. [7, p. 1] [5, p. 1]

Using sound libraries is not problem-free; it has challenges and complications. One of the predominant issues is the lack of a standardized taxonomy and universally agreed on vocabulary. Constructing and cataloging the libraries is also tiresome and error-prone [5, p. 1]. These disadvantages make it challenging to develop navigation systems for managing the sounds, which the sound designers need due to the sheer size of the data handled. Sound designers are also more interested in the sonic properties of a sound than the physical properties. Still, most of the categorization systems organize the sounds into geographical or physical categories. [29, p. 1]

A common aspect neglected in the sound libraries themselves and the supplementing tools is the absence of support for private annotation and subjective queries. Humans often describe sound with personal and biased words, which hints at the notion of a system adjustable for every individual being beneficial [7, p. 8]. However, implementing subjective tags is a problematic task. Subjectivity, by its nature, varies among people, making it impossible to create collective terms acceptable by everyone. On the contrary, if a system focused on each individual and adapted to the current operator and circumstances, it could produce better results. This inquest is the centerpiece of this thesis and what the author tries to solve with the proposed system.

2.2 Related work

The author has found no prior immediately similar or comparable work related to the proposed solution outlined in section 1.1 and further discussed in this thesis. Personal sound classification using subjective tags seems to be uncommon, as most previous studies focus on the mass and commonality while trying to establish universal audio categorization systems. Nonetheless, some related topics that the suggested system resembles have inspired extensive research and produced prominent content to a large extent.

Studies have concluded in the area of automatic sound identification and annotation of audio in great detail. These researches cover both fully automatic procedures by the computer alone, and semi-automatic methods were both humans and machines operate in conjunction. [47] shows a fully automatic approach to extract perceptual labels with reliable accuracy, and the more recent study in [29] describes a taxonomy based solely on the sonic properties of audio using feature selection, unsupervised learning, and hierarchical clustering. The results in [16] and [15] prove that human input, together with the appropriate algorithms, can be of great use to develop high-quality systems. Freesound and Google have launched large scale projects where everyone can help to develop automatic sound recognition by manually listening to and tagging sounds, thus generating correctly labeled training data for machines to utilize [33, 58].

Query methods in sound retrieval used for sound libraries are often text-based solutions formed on an ontology. The procedure for finding relevant sounds often searches for verbal descriptions associated with the sounds, e.g., the system developed in [13]. The textual verbs are usually definitions of sound itself, the sounding situation, or the sound impression [61]. These solutions inherit the imprecision and ambiguity problems of natural languages. This issue means that things like polysemy – where a single word can refer to multiple things – and synonymy pollute the systems [5, p. 2]. To remedy these problems, presented solutions like [5], [14], and [6] adapt semantic networks, say WordNet [57], to create relationships and connections between words, and eliminate language-specific uncertainties. Newer studies examine the possibility of including concepts such as domain-specific class definitions and relations [11], as well as knowledge elicitation and sound design ontology engineering as alternatives or extensions to the traditional text-based retrieval systems [7].

Typical for all of these systems is that they try to improve identification, annotation, and categorization of sounds, often in the context of sound libraries. People making use of sound libraries, e.g., sound designers, can take advantage of and benefit from the advancements. These goals also hold for the proposed system treated in this thesis, although the approach taken to achieve the intention varies slightly.

3 Methods

This chapter explains the methods needed for accomplishing the motives described in section 1.1. Establishing a sound analysis strategy, finding similar sounds, and predicting user categorization are some of the necessary components for the proposed system. Each section clarifies the choice of a particular procedure over comparable ones. As the author has no prior knowledge of the subject, the decision process progressed mostly through research and planning, with a great deal of trial and error involved.

3.1 Audio analysis

One objective required for the suggested system described in section 1.1 is the possibility to handle digital audio files in a programming environment. In order to interpret sounds programmatically, some form of conversion needs to take place, processing sound files to data applicable for a computer. There are different strategies for generating metadata from sounds, but the approach favored in this thesis is an interpretation with timbral characteristics.

Generally, timbre is a word describing perceptual attributes of a sound, excluding pitch and loudness, spatial and musicological descriptors, as well as higher-level cognitive properties [35, p. 7]. In more general terms, timbre is what enables humans – and probably other species – to distinguish one sounding object from another, even though the pitch, loudness, and duration remain constant. For example, if a violin and a piano play the same note at equal volume for a fixed period, it is still possible to differentiate the two instruments by their sound quality alone.

Many published methods are available that represent the timbre of a sound in different data-oriented ways, often in the form of an object containing metadata. The model used in [22] illustrates a way of analyzing sounds using analog equipment. As most work transpires digitally today, this process has advanced accordingly; [14] describes a computerized assembled counterpart. Numerous tools and libraries built on these concepts are available for analyzing and describing sounds. For instance, [25] presents a tool containing functions dedicated to the extraction of musical features from audio files, and [60] exhibits a web-based tool that performs spectral and roughness analysis on user-submitted sound files. A notable instance of an open-source library is *Essentia* [34, 4] developed by Music Technology Group (MTG), competent in audio analysis and audio-based music information retrieval to a high degree. Another popular library is *LibROSA* [26, 27], used for audio and music signal analysis. These systems are capable of much more than just producing timbral metadata, as they can track the beat of a song, work out the tempo, along with other features.

All of the described methods use one or more algorithms, either established or custom-made ones, to manage sounds and produce an output. Some projects keep the algorithms secret while others, like with *Essentia*, have them available to the public. Defining various adjective-based descriptions that function as dividers and designating each sound with a numeric value for every descriptor is another typical pattern among the systems. Combining words like 'roughness', 'sharpness', or similar with a number is something found in many of the methods for deriving timbral values. The studies in [49] and [35, p. 20] are examples of studies that provide comprehensive lists of words describing timbre.

The selected method for analyzing audio in this thesis is the *AudioCommons Timbral Models* package [21] developed by the Institute of Sound Recording (IoSR) for the AudioCommons project funded by the European Union [1]. It is part of the *Audio Commons Audio Extractor*, which *Freesound*, for instance, has chosen to integrate into their service [2, 10]. Interestingly enough, the implemented package is dependant on the previously mentioned *Essentia* and *LibROSA* libraries for some of the calculations, but only provide a fraction of the features that they have [36, p. 4][38, p. 5].

The timbral model package by IoSR can predict eight distinctive timbral characteristics from audio files of multiple formats, using a separate model for each characteristic. These eight models are all regression-based models, except for the classification model used to predict reverb qualities. This project omits the reverb model as it deviates from the other models' output and obstructs the sound similarity estimation method (see section 3.2).

One of the seven remaining used models produces a *hardness* value. The attack time, attack gradient, and spectral centroid of attack for a sound serve as parameters for a linear regression model, which generates said timbral characteristic. The model responsible for *depth* works by analyzing the spectral centroid, the energy proportion, and potentially the limit of lower frequencies of input audio. A representation of *brightness* models on a sound's spectral centroid variant and a spectral energy ratio of high frequencies, while *roughness* models the interaction of similar amplitude and frequency peaks within the frequency spectrum. The three remaining timbral values, *warmth*, *sharpness*, and *boominess*, are all implementations on previous models, which the associated papers describe and reference, but without any in-depth information. [36, 38]

The seven resolved characteristics produced by the timbral model conclusively portray a sound timbrally in a metadata format. This procedure means that every analyzed sound receives seven corresponding attributes represented numeric values, ranging from 0 to 100. Figure 3.1 shows an audio graph of an example sound, and table 3.1 shows the output of the same sound processed by the package.

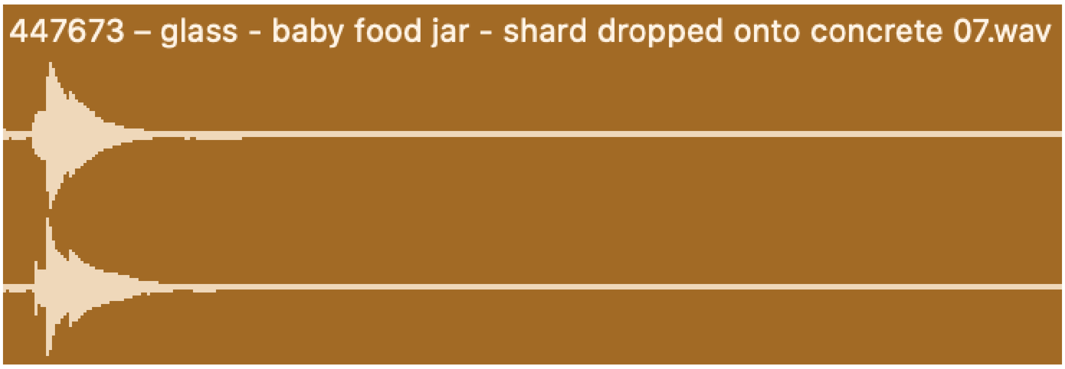


Figure 3.1: Audio graph of an example sound.

Table 3.1: Predicted timbral values for the same example sound.

hardness	*depth*	*brightness*	*roughness*	*warmth*	*sharpness*	*boominess*
67.71	3.41	88.3	54	31.16	100	5.07

The specified package is relatively straightforward to use, due to it being rather concentrated with only a few available scripts, which is one of the reasons for the author choosing to incorporate it into the developed system. Furthermore, the created set of timbral attributes seems to be of high-quality and covers a broad timbral spectrum. These aspects make it well suited as a building block for constructing prototypes, in which category the developed application falls, sequentially making it the chosen method for audio analysis in this thesis.

3.2 Sound similarity estimation

The goals in section 1.1 oblige a strategy for understanding a user's definition of similar sounds. Different people may have a distinct threshold when segregating sounds depending on their perception. A sound engineer could classify two closely resembled audio tracks as separate sounds, while a more inexperienced listener puts them in the same category. There are no wrongs or rights, as it is only a matter of taste and preference. By using the predicted timbral attributes in section 3.1 as a foundation, it is possible to conceptualize a strategy that materializes into a system that can distinguish sounds the same way as the current user.

Since the seven characteristics described in section 3.1 represent sounds in the proposed solution of this thesis, the similarity of two sounds is consequently the difference between their predicted timbral values. Two sounds which merely vary with 50 units of *hardness*, say if one sounds has a *hardness* value of 25 and the other of 75, are equally dissimilar in portrayed timbre as if the difference was with 50 units of *depth*. This formula builds on the axiom that all of the seven calculated timbral values are of equal significance. The difference sums of all timbral values between two sounds, like the *hardness* difference of 50 in the previous example, collectively make up the distance between the sounds. Even though all characteristics have equal influence, the tolerance of when two similar sounds split into separate sounds could alter among the attributes. This definition means that a user might perceive sounds differing with a value of 25 *hardness* as separate groups or classes, while only a difference of 15 *depth* achieves the same effect; one could say the user is more sensitive to *depth* changes. The suggested solution considers this with the support of multiple dimensions when searching for similar sounds, effectively meaning that each value corresponds to a point in a seven-dimensional coordinate system.

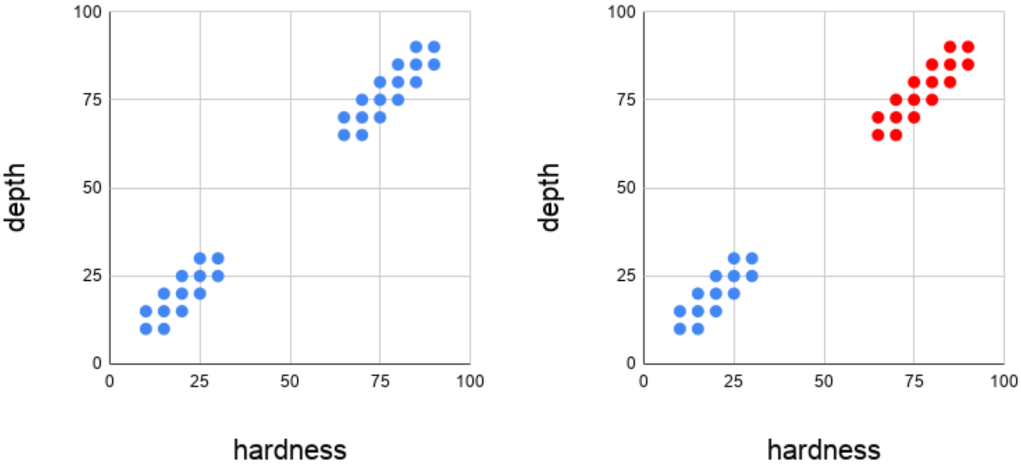


Figure 3.2: Example case were two users have categorized sounds differently. The blue color signifies once group and the red another.

When comparing sounds as audio or timbrally calculated values, there could be more deciding factors that affect a user's decision process when perceiving sounds as equal or not. Alongside differences within the same timbral characteristic, cross-characteristic discrepancies could also affect a user's judgment. In order to illustrate this scenario, the following examples will consider only *hardness*, *depth*, and *brightness*, or three dimensions. If two sounds have the same predicted values for *hardness* and *depth*, say 20 *hardness* and 30 *depth*, but different values for *brightness*, e.g., 85 versus 90, a user could still want to categorize them independently. Even though two sets of the same timbral values are identical and the remaining two only differ with a modest distance of 5, the cross-distance of *hardness–brightness* and *depth–brightness* also contrasts when comparing the example sounds. This variance from one characteristic to another could affect how a user segregates sounds.

In contrast, a user could perhaps say two sounds still belong to a single group even when all the distances between the same timbral characteristics are relatively large. This situation could perchance happen when the ratios are constant, like if the values for one sound were 20 for every defined attribute and 30 for the other sound. These kinds of variations are right to consider, as audio perception and sound segregation are quite abstract and dynamic, depending on the circumstances.

3.3 Machine learning

Part of the aimed for requirements outlined in section 1.1 demands a component or subsystem intelligent enough to mimic and adapt to different definitions of sound similarity. The threshold variance among users when segregating sounds is what the desired component should try to learn as precisely as possible. The discussed theories in section 3.2 set a foundation for such a subsystem, which this section expands upon and substantiates using machine learning (ML).

Machine learning is the act of programming a computer so that it learns from data without explicit instructions [12, p. 26]. In other words, the concept of machine learning is to let the algorithms define and output the rules of a problem when given the data and the answers as input [8, p. 28]. Many fields of technology make use of ML today to accomplish complex tasks. Speech recognition, spam filtering, and customizing web experiences are some areas that utilize ML extensively [12, p. 13]. As each problem is unique with its idiosyncrasies, no single algorithm can handle every task, often referred to as the 'No-Free-Lunch theorems' originating in [62, p. 12]. Therefore, it is first necessary to define the problem itself and all its challenges and quirks to know what style of ML best fits the task at hand.

The goal of the asserted subsystem is to categorize sounds based on similarity into the same groups as a specific person would, without regulation or influence from other users. With this intention specified, the problem falls into the category of classification-based problems. The idea builds on the motive that each user's perception is unique enough that neglecting the element of a unified definition for sound similarity based on shared assumptions is tolerable [19, p. 2]. As identical classifications of sounds among different persons could be quite common, this hypothesis is probably not entirely true. However, it is a generalization made in this thesis in order to achieve a functional system.

The subsystem should ergo initially operate without any presumed instructions, which also means it lacks directions on the concluding sum and division of classes. Evaluation for the component's accuracy comes from the users themselves when they interact with the proposed system planned in section 1.1. This valuation transpires when the user verifies categorizations designated by the ML algorithms in the developed program. With the adjustments, the number of correct classifications grows, and the subsystem receives more and more information and clues on how to segregate sounds according to the user's sound similarity definition. The continuous feedback from the user should improve the component accordingly. Finally, the size of the data collection of sounds to interpret can change, as the user can add more data to the developed application at any time, which the ML procedure should take into consideration.

In ML terms, these specifications signify that data collection and data preparation occur progressively and that all data is *unlabeled*, i.e., unprocessed, in the beginning. More samples are converted to *labeled* data as the user interacts with the system; in the end, when the user has verified every sample, all data is *labeled*. When the user includes new *unlabeled* data, it should undergo the same process until it is fully *labeled*. A single static ML algorithm is most likely insufficient, as the solution needs to improve gradually.

A dominant challenge of the described task is that it at least partially fits all the four major types of ML currently established [12, p. 30]:

- Supervised learning uses *labeled* data as a training set to discover the mapping between a set of inputs and outputs [12, p. 30]. A typical task solved with supervised learning is classification, which suits the problem at hand. A hindrance is that no *labeled* data is available, at least initially.

- Unsupervised learning tries to find usable patterns hidden within data without instructions, even when the input data is convoluted [12, p. 32]. These qualities make it suitable for the described task as *unlabeled* data is present at practically all stages of the suggested system.

- Semi-supervised is a mix of supervised and unsupervised learning. This combination means that semi-supervised algorithms function with *unlabeled* data, but require *labeled* data for training, although in a smaller scale than supervised algorithms [12, p. 35]. Since *labeled* data is unavailable at many times, as described earlier, the semi-supervised style has the same obstruction as the supervised type.

- Reinforcement learning deviates substantially from the other approaches, as it learns by itself using a positive and negative feedback system, an observer called ‘agent’, and a strategy, or ‘policy’, for scoring the most rewards over time [12, p. 35]. This method falls short as the end classification distribution is impossible to predict in advance, and due to the lack of scores to give the algorithms.

There are no doubt multiple algorithms and sequences of algorithms that manage to complete the outlined task. A combination of one or more unsupervised algorithms with some supervised or semi-supervised ones could probably work. One challenging bit, in that case, would be the interaction between the styles and understand which outputs to use in what scenarios. Reinforcement learning could also work if there were ways to construct the self-learning system that the algorithms demand.

The method that the author chose to implement uses two unsupervised learning algorithms: *Mean shift* and *K-means clustering*. The first creates a starting point for categorizing sounds based on similarity, and the other adjusts the estimation for every user interaction to better simulate the user’s perception. The implemented application relies heavily on *clustering*, or the act of detecting groups of elements with similar features, as it is the core concept for finding related sounds. Section 4.4 in the following chapter explains the two algorithms in more detail, how they synergy with each other, and how the proposed system uses them in a programming context.

4 Implementation

In order to present audio tagging with subjective labels in practice, the author of this thesis has built an application that strives to minimize the manual tagging effort needed by a user. Although simplistic, the application is a functional demonstration of how these problems could be solved using existing technologies, combining them into a solution. This chapter explains the design of the program, presents an overview of how the logic works, and examines implementation-specific problems.

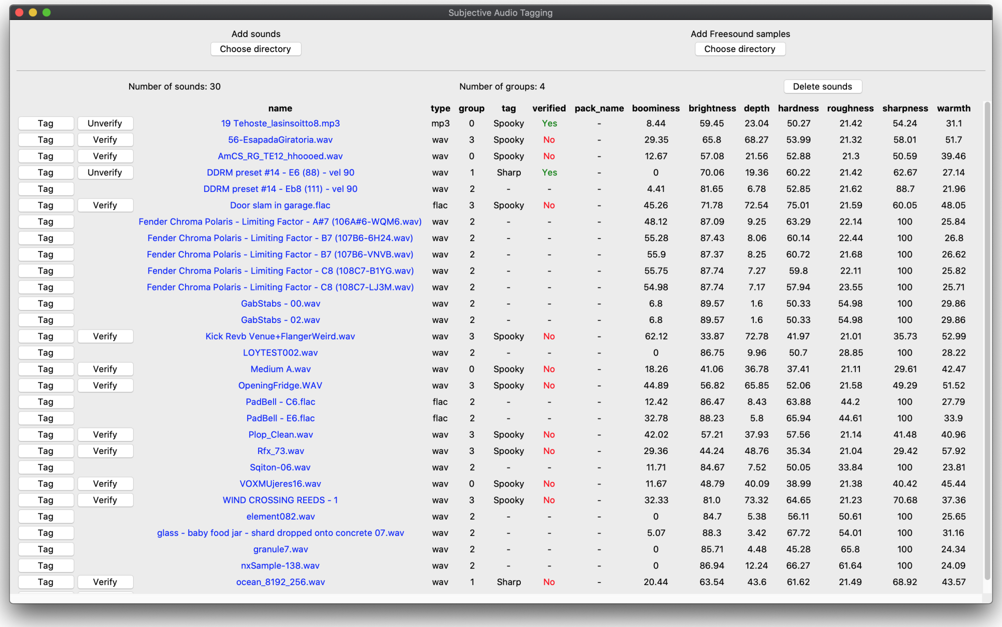
4.1 System architecture

All of the codebase for the application uses *Python* [43], mostly due to it being a widely-used programming language for scientific purposes [24], along with having an extensive number of ML and artificial intelligence (AI) packages accessible for use [63]. At the time of writing, version 3.8.1 is the latest stable release, which is also the version used throughout the codebase.

Some additional packages assist with certain functionalities:

- *AudioCommons Timbral Models* [21] analyzes sound input, calculates timbral attributes, and outputs the result to a usable data format.
- *scikit-learn* [52], along with *NumPy* [44], is used for the sound similarity estimation part of the application.
- *TinyDB* [56] handles elements related to databases.
- *pipenv* [45] installs and manages packages used in the project in a virtual environment.
- *autopep8* [17], *pylint* [41], and *rope* [48] deal with formatting, linting, and refactoring, respectively.

The straightforward, yet representative, user interface (UI) utilizes the *tkinter* [42] package as its infrastructure. This package is included in most *Python* distributions by default and offers a decent toolkit to make graphical user interfaces (GUIs).



Number of sounds: 30

Number of groups: 4

		name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	Unverify	19 Tehoste_lasinsoitto8.mp3	mp3	0	Spooky	Yes	-	8.44	59.45	23.04	50.27	21.42	54.24	31.1
Tag	Verify	56-EsapadaGiratoria.wav	wav	3	Spooky	No	-	29.35	65.8	68.27	53.99	21.32	58.01	51.7
Tag	Verify	AmCS_RG_TE12_hhoooed.wav	wav	0	Spooky	No	-	12.67	57.08	21.56	52.88	21.3	50.59	39.46
Tag	Unverify	DDRM preset #14 - E6 (88) - vel 90	wav	1	Sharp	Yes	-	0	70.06	19.36	60.22	21.42	62.67	27.14
Tag		DDRM preset #14 - Eb8 (111) - vel 90	wav	2	-	-	-	4.41	81.65	6.78	52.85	21.62	88.7	21.96
Tag	Verify	Door slam in garage.flac	flac	3	Spooky	No	-	45.26	71.78	72.54	75.01	21.59	60.05	48.05
Tag		Fender Chroma Polaris - Limiting Factor - A#7 (106A#6-WQM6.wav)	wav	2	-	-	-	48.12	87.09	9.25	63.29	22.14	100	25.84
Tag		Fender Chroma Polaris - Limiting Factor - B7 (107B6-6H24.wav)	wav	2	-	-	-	55.28	87.43	8.06	60.14	22.44	100	26.8
Tag		Fender Chroma Polaris - Limiting Factor - B7 (107B6-VNVB.wav)	wav	2	-	-	-	55.9	87.37	8.25	60.72	21.68	100	26.62
Tag		Fender Chroma Polaris - Limiting Factor - C8 (108C7-B1YG.wav)	wav	2	-	-	-	55.75	87.74	7.27	59.8	22.11	100	25.82
Tag		Fender Chroma Polaris - Limiting Factor - C8 (108C7-LJ3M.wav)	wav	2	-	-	-	54.98	87.74	7.17	57.94	23.55	100	25.71
Tag		GabStabs - 00.wav	wav	2	-	-	-	6.8	89.57	1.6	50.33	54.98	100	29.86
Tag		GabStabs - 02.wav	wav	2	-	-	-	6.8	89.57	1.6	50.33	54.98	100	29.86
Tag	Verify	Kick Revb Venue+FlangerWeird.wav	wav	3	Spooky	No	-	62.12	33.87	72.78	41.97	21.01	35.73	52.99
Tag		LOYTEST002.wav	wav	2	-	-	-	0	86.75	9.96	50.7	28.85	100	28.22
Tag	Verify	Medium A.wav	wav	0	Spooky	No	-	18.26	41.06	36.78	37.41	21.11	29.61	42.47
Tag	Verify	OpeningFridge.WAV	wav	3	Spooky	No	-	44.89	56.82	65.85	52.06	21.58	49.29	51.52
Tag		PadBell - C6.flac	flac	2	-	-	-	12.42	86.47	8.43	63.88	44.2	100	27.79
Tag		PadBell - E6.flac	flac	2	-	-	-	32.78	88.23	5.8	65.94	44.61	100	33.9
Tag	Verify	Plop_Clean.wav	wav	3	Spooky	No	-	42.02	57.21	37.93	57.56	21.14	41.48	40.96
Tag	Verify	Rfx_73.wav	wav	3	Spooky	No	-	29.36	44.24	48.76	35.34	21.04	29.42	57.92
Tag		Sqiton-06.wav	wav	2	-	-	-	11.71	84.67	7.52	50.05	33.84	100	23.81
Tag	Verify	VOXMUjeres16.wav	wav	0	Spooky	No	-	11.67	48.79	40.09	38.99	21.38	40.42	45.44
Tag	Verify	WIND CROSSING REEDS - 1	wav	3	Spooky	No	-	32.33	81.0	73.32	64.65	21.23	70.68	37.36
Tag		element082.wav	wav	2	-	-	-	0	84.7	5.38	56.11	50.61	100	25.65
Tag		glass - baby food jar - shard dropped onto concrete 07.wav	wav	2	-	-	-	5.07	88.3	3.42	67.72	54.01	100	31.16
Tag		granule7.wav	wav	2	-	-	-	0	85.71	4.48	45.28	65.8	100	24.34
Tag		nxSample-138.wav	wav	2	-	-	-	0	86.94	12.24	66.27	61.64	100	24.09
Tag	Verify	ocean_8192_256.wav	wav	1	Sharp	No	-	20.44	63.54	43.6	61.62	21.49	68.92	43.57

Figure 4.1: UI of the application.

NB Although the UI is an essential and necessary part of an application, the priority of it has been lowered in this project to allow more time spent on calculation-heavy parts of the program.

4.2 Overview

By looking at fig. 4.1, one can see that the application consists of an upper and a lower section. Clicking the buttons in the upper section adds sounds to the application in one of two different ways, processing them accordingly (described in section 4.3 and section 4.4). In the lower section, an info-panel displays statistics about all sounds added, plus a delete-all button, and a table underneath shows metadata for the sounds. Alongside the attributes shown, each row also lists buttons used for updating a particular sound. In conjunction with the possibility of listening to a sound by clicking its highlighted name, these buttons allow for tagging and verifying of sounds (section 4.5). The user can also choose to rename a tag or, after the program has assigned its tags, discard a tag, and relabel a sound entirely (section 4.6). Finally, some additional logic is required when adding more sounds to the application, but once done, the user can repeat all of the previously mentioned steps (section 4.7).

4.3 Adding sounds

The process of adding sounds to the application starts when the user clicks the upper leftmost button in fig. 4.1 and chooses a directory that contains audio files; the implementation supports many of the most commonly used audio formats. However, before the sounds appear in the UI where they can be tagged, they first have to be analyzed and converted to a usable format.

AudioCommons Timbral Models is a package that processes sounds and predicts eight timbral characteristics, which in turn helps managing sounds and enables measuring and comparison between them. The 'semantic annotation of non-musical sound properties' work package deliverables explain how these characteristics were developed and describe the implementation of the models, plus usage of the package in detail [35, 36, 37, 38, 39, 40]. Although developed to aid with automatic tagging of sounds [40, p. 4], in this project, the calculated attributes distinguish one sound from another regarding similarity.

The eight timbral characteristics are:

1. *booming*
2. *brightness*
3. *depth*
4. *hardness*
5. *roughness*
6. *sharpness*
7. *warmth*
8. *reverberation*

The seven first characteristics are all regression-based models. These models produce a numerical output ranging from 0 to 100; the `clip_output` parameter seen in listing 4.1 restrains the upper value, as it may otherwise exceed the range. The last feature, *reverberation*, is a classification model, producing a boolean value represented as 0 or 1. Since the output values serve as a measurement for how similar sounds are – computed by ML algorithms – each one of them is considered equally significant. However, as the *reverberation* model produces a different ranging result, the prerequisite for a homologous environment rules it out. Therefore, the program omits the *reverberation* value (shortened to *reverb* in the function output) when processing added sounds.

Listing 4.1: Function for processing (nested) sounds in a directory.

```python
def process_sounds(directory):
  """Processes sounds using AudioCommons Timbral Models."""
  result = []

  paths = Path(directory).rglob("*")
  for path in paths:
    try:
      path_str = str(path)
      sound = timbral_extractor(path_str, clip_output=True)
      del sound["reverb"]

      sound["id"] = uuid1().hex
      sound["name"] = Path(path_str).resolve().stem
      sound["pack_name"] = ""
      sound["path"] = path_str
      sound["tag"] = ""
      sound["type"] = Path(path_str).suffix.replace(".", "")
      sound["verified"] = False

      result.append(sound)
    except:
      continue

  return result
```

4.3.1 Freesound samples

As seen from fig. 4.1, there are two buttons in the upper section for adding sounds to the application. The alternative rightmost button accepts predefined *Freesound* [31] metadata files, i.e., JavaScript Object Notation (JSON) files with the timbral characteristics included. Because *Freesound* integrates with the *Audio Commons Audio Extractor* [2] tool (which uses the same timbral prediction as *AudioCommons Timbral Models*), almost all available sounds have an associated metadata file containing the calculated timbral attributes [10]. To browse and download these files, one can use the *Freesound API* [30] or the *Audio Commons Extractor Web Demonstrator* [3]. This feature makes it possible to add sets of biased data, namely where the wanted result is defined; the tests in the evaluation chapter (chapter 5) uses these constructed sets when measuring results. It also speeds up the process of experimenting with different sounds, as the analysis of sounds done by the *AudioCommons Timbral Models* package can be slow at times, especially for larger audio files.

Listing 4.2: Function for processing Freesound samples.

```python
def process_freesound_samples(directory):
  """Processes Freesound samples to a common format."""
  result = []
  relevant_properties = ["ac_analysis", "id", "name", "pack_name", "type"]

  paths = Path(directory).rglob("*.json")
  for path in paths:
    try:
      with open(path, "r") as reader:
        metadata = loads(reader.read())
        converted_metadata = {
            "path": str(path),
            "tag": "",
            "verified": False
        }

        for rp in relevant_properties:
          value = metadata[rp]

          if rp == "ac_analysis":
            for tc in timbral_characterstics:
              converted_metadata[tc] = value["ac_" + tc]
          else:
            converted_metadata[rp] = value

        result.append(converted_metadata)
    except:
      continue

  return result
```

4.3.2 Deleting sounds

A possibility to delete sounds is available in the application by clicking the pertinently named button displayed in fig. 4.1. This interaction opens a prompt where the user confirms the choice to remove all sounds. If approved, all databases purge their content, and the program returns to its initial state.

4.4 Sound similarity

The timbral characteristics described in section 4.3 estimates the similarity of sounds in this project. When comparing two sounds, the more closely resembling predicted timbral values there are between them, the more similar the sounds are. The ML logic uses this estimation of closeness when finding groups of sound, or *clusters*. All sounds in a cluster link to a single tag; multiple clusters can have the same tag.

4.4.1 Mean shift

Two different ML algorithms operate together in the application to find and maintain groups of sound. Both of them come from the *scikit-learn* package. The first algorithm searches for naturally occurring clusters in a set of data using the *Mean shift* [51] procedure. It operates on the timbral coordinates for all sounds, X, and produces an array of cluster centers, also called *centroids*. Since the final sum of tags a user chooses to use throughout the application is unknown, *Mean shift* works particularly well for creating a foundation of how many groups of sound could be satisfactory. It independently decides on the optimal number of centroids in a set of data, making it ideal for creating a starting point.

Listing 4.3: Function for finding initial clusters.

```python
def init_groups(sounds, X):
  """Searches for initial groups of sound using Mean shift."""
  ms = MeanShift().fit(X)
  centroids = ms.cluster_centers_.tolist()
  centroids.sort()

  return find_groups(sounds, X, numpy.array(centroids))
```

4.4.2 K-means

K-means [50] is the second ML algorithm used for calculating clusters among sounds. This algorithm is the method used to separate the sounds into different groups and marking them accordingly. It takes as input the current cluster center coordinates, which are either produced by the preparatory *Mean shift* procedure or by a previous iteration of the *K-means* clustering. These coordinates guide the algorithm when deriving renewed concluding centroids. It also needs a k value, which is the number of groups to split the data in, derived from the number of centroids. Lastly, the `n_init` parameter tells the algorithm to perform a single iterator on each seed, so the new cluster center coordinates do not strive too far from the previously calculated ones. Like the *Mean shift* algorithm, the *K-means* procedure operates on X, being coordinates for the timbral characteristics of all sounds. As the number of tags and groups grows with usage, this algorithm scales proportionately and, when executed, produces the correct updated number of centroids.

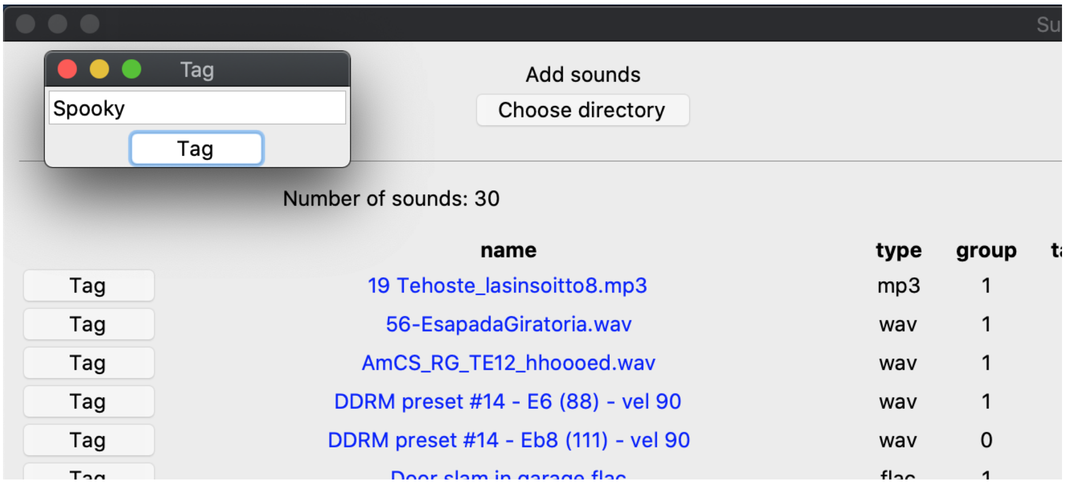
Listing 4.4: Function for finding groups of sound.

```python
def find_groups(sounds, X, centroids):
  """Finds groups of sound using K means."""
  k = len(centroids)
  km = KMeans(init=centroids, n_clusters=k, n_init=1).fit(X)

  updated_sounds = deepcopy(sounds)
  for sound, group in zip(updated_sounds, km.labels_):
    sound["group"] = int(group)

  updated_centroids = km.cluster_centers_.tolist()
  updated_centroids.sort()

  return updated_sounds, updated_centroids
```

4.4.3 Usage

Each time the user adds a set of sounds to the application, both the *Mean shift* and the *K-means* algorithm invokes in sequence. This step makes sure that every sound added belongs to a group, and if applicable, tags additional sounds with the existing group's tag. Whenever there is a need for a new group (described in section 4.6.2), the *K-means* algorithm is triggered separately.

4.5 Tagging



Spooky

Number of sounds: 30

name	type	group
19 Tehoste_lasinsoitto8.mp3	mp3	1
56-EsapadaGiratoria.wav	wav	1
AmCS_RG_TE12_hhoooed.wav	wav	1
DDRM preset #14 - E6 (88) - vel 90	wav	1
DDRM preset #14 - Eb8 (111) - vel 90	wav	0
Door slam in garage.flac	flac	1

Figure 4.2: Tagging a sound.

Once sounds have been processed, designated to a group, and added to the application, the user can start tagging them. To annotate a sound with a tag, the user clicks the corresponding button in the table and fills in the subsequent dialog, illustrated in fig. 4.2. The label the user chooses to enter in the dialog can be the same as an existing tag or a new one. The sound's *tag* property –

which is declared empty in both listing 4.1 and listing 4.2 – stores this input value. To decide what tag goes along with a sound, the user probably wants to listen to the sound first. The sound names listed in the table shown in fig. 4.1 are links that trigger this action, opening the matching audio file in the system default application.

NB The implementation limits tagging to allow only a single label per sound; multiple groups can have the same tag. This restriction scales down the logic and code needed for the application, especially in the ML parts.

4.5.1 Verification

For the user to be able to keep track of which sounds they have tagged, each sound has a *verified* property. This attribute is a boolean flag initially set to `False` for all sounds, established in listing 4.1 and listing 4.2. Whenever the user tags a sound, the *verified* status remains or changes to `True` for that sound, as opposed to the automatic tagging process by the program, which will not affect the value of the property. When a sound has a tag – assigned either manually or automatically – the user can choose to toggle the *verified* status by clicking the corresponding button in the table (see fig. 4.1). The *verified* property also determines how the automatic tagging behaves, judging the need to create new groups or carry on with the ones already defined.

4.6 Automatic tagging

One of the core functionalities in the application is the inclusion of an automatic tagging system. This procedure helps cut down the overall effort needed to label sounds and is flexible enough to cover most use cases adequately. For every sound manually tagged by the user, the program responds by automatically tagging similar sounds. The logic consists of two different operations, where the selection of the alternative to execute is dependent on the relevant sound's *verified* property. One option directly changes a group's tag, or in other words, performs a tag renaming. The other is slightly more complex, creating one or more new groups and modifying existing tags and groups. Both alternatives save the modifications to the application databases and repaint the screen with the revised sounds once done.

4.6.1 Tag renaming

The first and more straightforward of the two automatic tagging scenarios is a renaming of a group's tag where all ongoing groupings are kept intact. This event triggers when a *verified* sound is tagged, or when the user tags a sound belonging to a group where all sounds lack verification. The latter happens, for instance, when the user tags a sound for the first time since all sounds have their *verified* status initially set to `False`. In both cases, the tagged sound's *verified* status is kept or set to `True` by the code fragment responsible for renaming.

Listing 4.5: Logic for initalizing and renaming a tag.

```
# Keep current groupings.
group_sounds = db.search(where("group") == sound["group"])
if (sound["verified"] or all([not gs["verified"] for gs in group_sounds
    ])):
  db.update({"tag": input}, where("group") == sound["group"])
  db.update({"verified": True}, where("id") == sound["id"])
```

4.6.2 Modifying groups

Occasionally, the user might be dissatisfied with tags allocated by the application. By retagging a sound with the *verified* flag set to `False`, the user signals to the program that the current tag is incorrect, which, consequently, means supplementary groups are required. The only deviation from this is if all sounds in a group are unverified, as then a tag renaming is sufficient to accomplish the same outcome. It is important to note that the number of groups cannot exceed the number of sounds, and this exception handling protects the application from reaching that erroneous state.

Listing 4.6: Logic for modifying groups.

```
else:  # Modify groups.
    db.update(
        {"tag": input, "verified": True},
        where("id") == sound["id"]
    )
    modified_sound = db.search(where("id") == sound["id"])[0]

    db_sounds = db.all()
    X = ml.search(where("id") == "X")[0]["data"]
    centroids = ml.search(where("id") == "centroids")[0]["data"]

    sounds, centroids = modify_groups(
        db_sounds,
        X,
        centroids,
        modified_sound,
        input,
        db_sounds
    )
    sounds, X, centroids = review_groups(sounds, X, centroids)
    sounds, X = tag_groups(sounds)
    update_databases(sounds, centroids, X)
```

A retagging action by the user starts a series of events where groups are created and modified until the arrangement meets specific criteria. The group numbers displayed in fig. 4.1 show how the application internally connects a sound to a group. The relationship between a specific number and a group may change in the process of modifying sounds, which is acceptable, as it is only a conceptual representation of the division between groups.

Partly what makes this scenario more intricate than the renaming one, is the consideration of how reshaping groups affect sounds that are already *verified*. If a sound is *verified*, it means the tag assigned to it is correct and should, therefore, remain the same when groups are modified. This concern means some part of the program needs to check that only unverified sounds – sounds tagged by the application and untagged sounds – are altered. In other words, some logic needs to be present to prevent a ping-pong effect where completed sounds are 'stolen' between groups.

What orchestrates the adjustment of groups and solves accompanying problems is the `modify_groups` core function, listed in listing 4.7. It is a recursive method that keeps creating new groups as long as *verified* sounds are incorrectly tagged. The situation of *verified* sounds temporarily becoming mistagged is a side effect of the function's implementation with an optimistic pattern when reworking the clusters.

Listing 4.7: Functions for adding and modifying groups.

```python
def add_group(sounds, X, centroids, modified_sound):
  """Adds a new group of sound."""
  new_centroid = []
  for tc in timbral_characterstics:
    new_centroid.append(modified_sound[tc])

  new_centroids = deepcopy(centroids)
  new_centroids.append(new_centroid)
  new_centroids.sort()

  return find_groups(sounds, X, numpy.array(new_centroids))


def modify_groups(sounds, X, centroids, modified_sound, tag, db_sounds):
  """Modifies groups until sounds have their associated tag."""
  updated_sounds, updated_centroids = add_group(
      sounds, X, centroids, modified_sound)

  # Finds the new group.
  for updated_sound in updated_sounds:
    if updated_sound["id"] == modified_sound["id"]:
      new_group = updated_sound["group"]
      break

  # Tags sounds in the new group.
  for updated_sound in updated_sounds:
    if updated_sound["group"] == new_group:
      updated_sound["tag"] = tag

  # Checks that all verified sounds kept their tag.
  for db_sound, updated_sound in zip(db_sounds, updated_sounds):
    if (db_sound["verified"] and
        db_sound["tag"] != updated_sound["tag"] and
            db_sound["id"] != modified_sound["id"]):
      return modify_groups(updated_sounds, X, updated_centroids,
                           db_sound, db_sound["tag"], db_sounds)

  return updated_sounds, updated_centroids
```

First, the unverified sound tagged by the user serves as a reference point, having a new group created around it. As a reaction to adding a group, recalculating the position of current centroids is consequently necessary. This development could lead to less or more significant adjustments in existing groups depending on the situation. Calling the *K-means* algorithm (section 4.4.2) from `add_group` calculates new cluster center coordinates using the incremented number of groups. The newly formed groups are optimistic in the sense that the sounds they include receive tags with no consideration towards their verification status.

To remedy the possible alteration of *verified* sounds' *tag* property, the process of modifying groups repeats recursively. Each mistakenly changed sound has its erroneous tag corrected by, in turn, running `modify_groups` with itself as the argument. The function, again, creates a new group around the input sound plus revises present groups, tags sounds belonging to the newly constructed group, and loops through the modified data to review the updated tags. This procedure reruns until all *verified* sounds have their original correctly associated *tag* property.

After the step of modifying groups, there might be a potential issue of multiple tags per group, violating the enacted simplification of groups having only a single tag (section 4.5). For that reason, narrowing down tags to a single label per group takes place. As long as there are different contesting tags in a group, the `review_groups` function seen in listing 4.8 keeps creating new groups. This regulation splits groups into smaller subgroups – with the centroids adjusted accordingly – until a single tag remains in the groups.

Listing 4.8: Function for singularizing the number of tags per group.

```python
def review_groups(sounds, X, centroids):
  """Makes sure that there is only one tag per group."""
  updated_sounds, updated_X = sort_sounds(sounds)
  updated_centroids = centroids

  for a, b in zip(updated_sounds, updated_sounds[1:]):
    if (a["group"] == b["group"] and a["tag"] != b["tag"]
            and a["tag"] and b["tag"]):
      updated_sounds, updated_centroids = add_group(
          updated_sounds, updated_X, updated_centroids, b)
      return review_groups(
          updated_sounds, updated_X, updated_centroids)

  return updated_sounds, X, updated_centroids
```

Lastly, the code in listing 4.9 assures that every tagless sound belonging to a group with a tag is labeled accordingly. The `tag_groups` function sorts the entries – placing untagged sounds together – and labels the incomplete sounds with the group's tag. In some cases, more than one tag might exist in a group, and if so, the first tag found serves as the source.

Listing 4.9: Function for tagging incomplete sounds.

```python
def tag_groups(sounds):
  """Tags sounds that are missing a group's tag."""
  updated_sounds, updated_X = sort_sounds(sounds)

  for a, b in zip(updated_sounds, updated_sounds[1:]):
    if a["group"] == b["group"] and a["tag"] and not b["tag"]:
      b["tag"] = a["tag"]

  return updated_sounds, updated_X
```

In the end, the initially retagged sound is now verified and has a group of its own associated with it. The sum of groups has grown by at least one, up to a maximum of the number of sounds. Group numberings are presumably shuffled compared to what they were before, but each sound still links to a group. Some sounds have a new tag, which remains for the user to accept or neglect. Notably, all of the sounds that were already *verified* have remained unchanged and preserved their validated *tag* value.

4.7 Additional sounds

Including additional sounds introduces new challenges of how to continue the tagging progress made so far. When adding more sounds, the application should be able to tag new sounds that fit inside the current group boundaries, as well as identify outlying samples that belong to a group not yet established. The part of the program in charge of achieving this also needs to be aware of current group tags not to disrupt ongoing tagging advancement.

The method to expand the sound pool ultimately is the same as if all sounds were added together at the start, with the user performing the same interactions hitherto. Every time the user adds more sounds to the program, the entire set of sounds, both newly and previously added ones, is processed by the ML algorithms from scratch. The *Mean shift* algorithm (section 4.4.1) finds preliminary groups and passes the output to the *K-means* procedure (section 4.4.2) for further analysis. This computation disrupts the existing structure of groups, where the resulting groups may contain multiple sounds with different tags (violating the rule of a single label per group) and sounds that are missing the group's tag. These are the equivalent to the transpiring issues in section 4.6.2 when modifying groups. Hence, similarity alike, utilizing listing 4.8 neglects the former problem of multiple tags in a group, and listing 4.9 solves the latter issue of missing tags.

Depending on how the additional sounds correlate to the ones in the application, the sum of groups formed varies. In most cases, the total grows or stays the same. In some cases, however, the number of groups decreases. Added sounds may occur between two identically tagged groups, effectively joining them together, or 'filling in the gap' between the groups. This effect could also be an outcome of retagging all sounds from the beginning. Although the review logic theoretically replays tagging interactions by the user, it is not a one-to-to simulation, and the difference may fluctuate the concluding result.

5 Evaluation

This chapter explains the evaluation process of the developed application described in chapter 4. Predefined examples measure the performance of the program using sets of data with a determined solution and analyze the steps needed to achieve said result. The examples simulate different types of use cases, which, in effect, exposes the strengths and weaknesses of the implementation.

5.1 Subjectivity

As the application should be flexible enough to deal with varying levels of sound perception, the evaluation process needs multiple examples to test how the program performs in different situations. All of these test cases need a common established standard of subjectivity to evaluate the application objectively.

The approach chosen in this project to form this specification of subjectivity is to construct sets of data consisting of timbrally differentiating groups of sound. Although distinction by timbre might be subjective in itself, user-derived groups of sound from Freesound, together with the author choosing the groups, will act as the judge in this case. Freesound supports sound pack configurations defined by their users [32], which the test cases, in turn, take advantage of as building blocks. The examples utilize various packs to either model a more closely resembled sound landscape, with say similar-sounding synthesizer sounds, or a more contrasting one, e.g., with drums and piano sounds. Most of the packs have descriptive tags, meaning the labels of the annotated sounds are what the sound represents: ‘gunshot’, ‘guitar sound’, or similar. Even though the application’s primary goal is to deal with subjective tags, the descriptive tags are applicable for test data, as they are, in a sense, generally agreed upon subjective tags. Some sound waves classify as piano sounds to humans while others do not, but the infinite amount of interlying sounds keeps the separating border open to debate.

5.2 Measurement

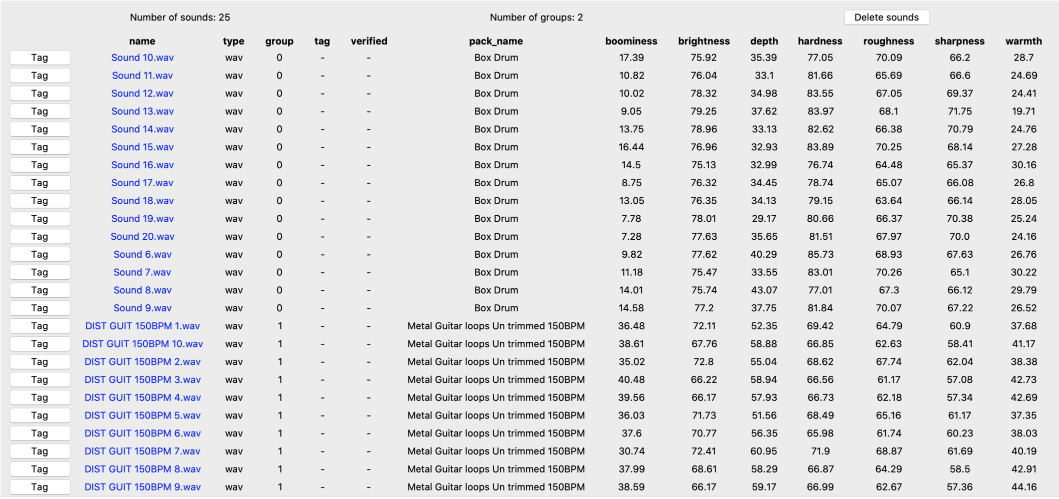
The unit used for grading in the evaluation will be the number of user interactions needed to reconstruct the test data interpretation until each sample resides in its appointed group. Such rearrangements occur every time the user chooses to retag a sound, thus creating more groups and changing the structure of existing ones (see section 4.6.2 for more details). This strategy implies that the best-case scenario will be with zero corrections made by the user, which occurs when the starting *Mean Shift* algorithm (section 4.4.1) finds the right number of groups immediately. The more modifications needed from the user, the worse the performance rating becomes, signifying that the clustering logic only discovering a single group when the user wants each sound to be in its group is the worst-case scenario.

When correcting sounds with erroneous tags, the order of which this is performed matters. Similar to how a chess opening alters the course of the game, the sound initially revised affects the rest of the modification process, which may conclude in a path with fewer or more total adjustments needed. Choosing to verify sounds as they become tagged or leaving them be also plays a vital role in how the application functions further on. This variation makes it difficult to measure all potential outcomes in complex use cases and settle on a final score.

5.3 Case 1 – Rudimentary

In this introductory example, sounds from two unconnected packs will make up the test data. The first pack of sound, *Box Drum* [55], consists of 15 samples produced by hitting a large metallic box under a bridge. *Metal Guitar loops Un trimmed 150BPM* [46] is the second pack used and contains ten, as hinted by the name, distorted guitar sounds all played at the same speed.

The goal in this example will be to have the sounds separated into two groups, having the box drum sounds tagged as **Box**, and the metal guitar sounds labeled with **Guitar**. Loading the test data into the application, one can see that opening groupings done by the program coheres with the wanted result; all the box sounds are in one group, and the guitar sounds in another.



Number of sounds: 25

Number of groups: 2

	name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	Sound 10.wav	wav	0	-	-	Box Drum	17.39	75.92	35.39	77.05	70.09	66.2	28.7
Tag	Sound 11.wav	wav	0	-	-	Box Drum	10.82	76.04	33.1	81.66	65.69	66.6	24.69
Tag	Sound 12.wav	wav	0	-	-	Box Drum	10.02	78.32	34.98	83.55	67.05	69.37	24.41
Tag	Sound 13.wav	wav	0	-	-	Box Drum	9.05	79.25	37.62	83.97	68.1	71.75	19.71
Tag	Sound 14.wav	wav	0	-	-	Box Drum	13.75	78.96	33.13	82.62	66.38	70.79	24.76
Tag	Sound 15.wav	wav	0	-	-	Box Drum	16.44	76.96	32.93	83.89	70.25	68.14	27.28
Tag	Sound 16.wav	wav	0	-	-	Box Drum	14.5	75.13	32.99	76.74	64.48	65.37	30.16
Tag	Sound 17.wav	wav	0	-	-	Box Drum	8.75	76.32	34.45	78.74	65.07	66.08	26.8
Tag	Sound 18.wav	wav	0	-	-	Box Drum	13.05	76.35	34.13	79.15	63.64	66.14	28.05
Tag	Sound 19.wav	wav	0	-	-	Box Drum	7.78	78.01	29.17	80.66	66.37	70.38	25.24
Tag	Sound 20.wav	wav	0	-	-	Box Drum	7.28	77.63	35.65	81.51	67.97	70.0	24.16
Tag	Sound 6.wav	wav	0	-	-	Box Drum	9.82	77.62	40.29	85.73	68.93	67.63	26.76
Tag	Sound 7.wav	wav	0	-	-	Box Drum	11.18	75.47	33.55	83.01	70.26	65.1	30.22
Tag	Sound 8.wav	wav	0	-	-	Box Drum	14.01	75.74	43.07	77.01	67.3	66.12	29.79
Tag	Sound 9.wav	wav	0	-	-	Box Drum	14.58	77.2	37.75	81.84	70.07	67.22	26.52
Tag	DIST GUIT 150BPM 1.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	36.48	72.11	52.35	69.42	64.79	60.9	37.68
Tag	DIST GUIT 150BPM 10.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	38.61	67.76	58.88	66.85	62.63	58.41	41.17
Tag	DIST GUIT 150BPM 2.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	35.02	72.8	55.04	68.62	67.74	62.04	38.38
Tag	DIST GUIT 150BPM 3.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	40.48	66.22	58.94	66.56	61.17	57.08	42.73
Tag	DIST GUIT 150BPM 4.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	39.56	66.17	57.93	66.73	62.18	57.34	42.69
Tag	DIST GUIT 150BPM 5.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	36.03	71.73	51.56	68.49	65.16	61.17	37.35
Tag	DIST GUIT 150BPM 6.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	37.6	70.77	56.35	65.98	61.74	60.23	38.03
Tag	DIST GUIT 150BPM 7.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	30.74	72.41	60.95	71.9	68.87	61.69	40.19
Tag	DIST GUIT 150BPM 8.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	37.99	68.61	58.29	66.87	64.29	58.5	42.91
Tag	DIST GUIT 150BPM 9.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	38.59	66.17	59.17	66.99	62.67	57.36	44.16

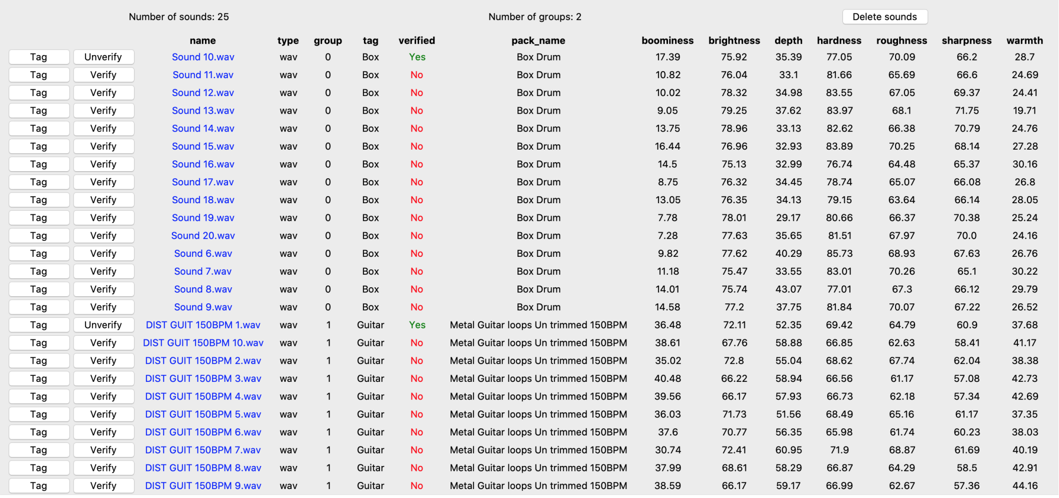
Figure 5.1: Case 1 test data initially loaded into the application.

Worth noting for both packs is that the sound source remains constant throughout all samples, hence the sounds being comparatively similar sounding in themselves, mostly varying in rhythm, intensity, and placement. The calculated timbral characteristics reflect this auditory closeness, with the average range between the maximum and minimum values shown in table 5.1 being a relatively small value.

This example falls under the best-case scenario, as the user needs to make **no group modifications** to give each sound its proper tag. Labeling one sound from each pack with the corresponding tag correctly tags the remaining sounds as well, via the program's automatic tagging procedure.

Table 5.1: Case 1 timbral characteristics maximum and minimum value ranges

	Box Drum	*Metal Guitar loops Un trimmed 150BPM*
boominess	10.11	9.74
brightness	4.12	6.63
depth	13.89	9.39
hardness	8.98	5.92
roughness	6.63	7.70
sharpness	6.66	4.96
warmth	10.51	6.81
Average	8.70	7.31



Number of sounds: 25

Number of groups: 2

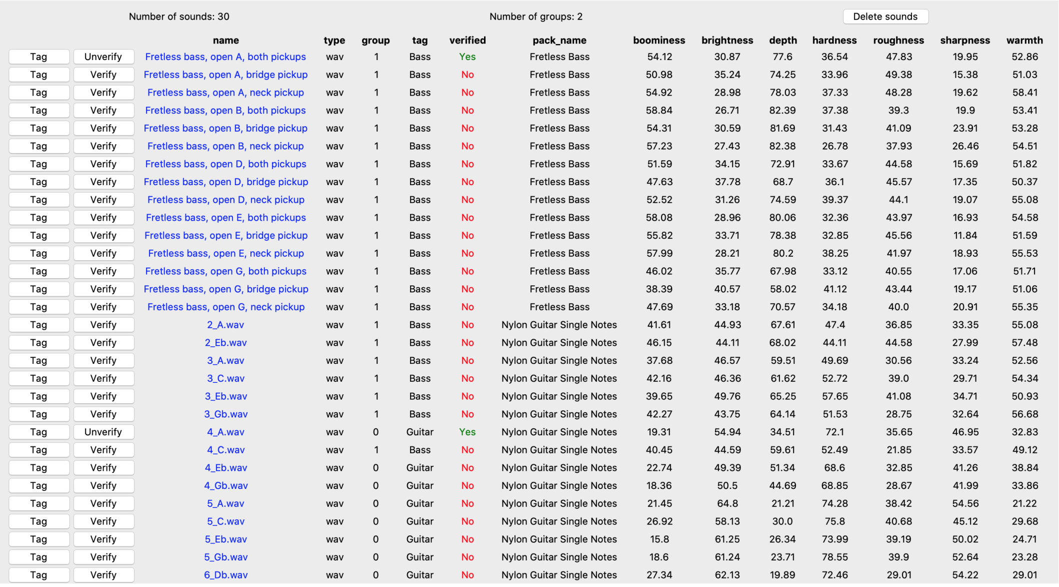
		name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	Unverify	Sound 10.wav	wav	0	Box	Yes	Box Drum	17.39	75.92	35.39	77.05	70.09	66.2	28.7
Tag	Verify	Sound 11.wav	wav	0	Box	No	Box Drum	10.82	76.04	33.1	81.66	65.69	66.6	24.69
Tag	Verify	Sound 12.wav	wav	0	Box	No	Box Drum	10.02	78.32	34.98	83.55	67.05	69.37	24.41
Tag	Verify	Sound 13.wav	wav	0	Box	No	Box Drum	9.05	79.25	37.62	83.97	68.1	71.75	19.71
Tag	Verify	Sound 14.wav	wav	0	Box	No	Box Drum	13.75	78.96	33.13	82.62	66.38	70.79	24.76
Tag	Verify	Sound 15.wav	wav	0	Box	No	Box Drum	16.44	76.96	32.93	83.89	70.25	68.14	27.28
Tag	Verify	Sound 16.wav	wav	0	Box	No	Box Drum	14.5	75.13	32.99	76.74	64.48	65.37	30.16
Tag	Verify	Sound 17.wav	wav	0	Box	No	Box Drum	8.75	76.32	34.45	78.74	65.07	66.08	26.8
Tag	Verify	Sound 18.wav	wav	0	Box	No	Box Drum	13.05	76.35	34.13	79.15	63.64	66.14	28.05
Tag	Verify	Sound 19.wav	wav	0	Box	No	Box Drum	7.78	78.01	29.17	80.66	66.37	70.38	25.24
Tag	Verify	Sound 20.wav	wav	0	Box	No	Box Drum	7.28	77.63	35.65	81.51	67.97	70.0	24.16
Tag	Verify	Sound 6.wav	wav	0	Box	No	Box Drum	9.82	77.62	40.29	85.73	68.93	67.63	26.76
Tag	Verify	Sound 7.wav	wav	0	Box	No	Box Drum	11.18	75.47	33.55	83.01	70.26	65.1	30.22
Tag	Verify	Sound 8.wav	wav	0	Box	No	Box Drum	14.01	75.74	43.07	77.01	67.3	66.12	29.79
Tag	Verify	Sound 9.wav	wav	0	Box	No	Box Drum	14.58	77.2	37.75	81.84	70.07	67.22	26.52
Tag	Unverify	DIST GUIT 150BPM 1.wav	wav	1	Guitar	Yes	Metal Guitar loops Un trimmed 150BPM	36.48	72.11	52.35	69.42	64.79	60.9	37.68
Tag	Verify	DIST GUIT 150BPM 10.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	38.61	67.76	58.88	66.85	62.63	58.41	41.17
Tag	Verify	DIST GUIT 150BPM 2.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	35.02	72.8	55.04	68.62	67.74	62.04	38.38
Tag	Verify	DIST GUIT 150BPM 3.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	40.48	66.22	58.94	66.56	61.17	57.08	42.73
Tag	Verify	DIST GUIT 150BPM 4.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	39.56	66.17	57.93	66.73	62.18	57.34	42.69
Tag	Verify	DIST GUIT 150BPM 5.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	36.03	71.73	51.56	68.49	65.16	61.17	37.35
Tag	Verify	DIST GUIT 150BPM 6.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	37.6	70.77	56.35	65.98	61.74	60.23	38.03
Tag	Verify	DIST GUIT 150BPM 7.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	30.74	72.41	60.95	71.9	68.87	61.69	40.19
Tag	Verify	DIST GUIT 150BPM 8.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	37.99	68.61	58.29	66.87	64.29	58.5	42.91
Tag	Verify	DIST GUIT 150BPM 9.wav	wav	1	Guitar	No	Metal Guitar loops Un trimmed 150BPM	38.59	66.17	59.17	66.99	62.67	57.36	44.16

Figure 5.2: Case 1 test data finalized.

5.4 Case 2 – Divergence

The test data in the next use case incorporates sounds from two related packs, both having 15 sound snippets each played on a stringed instrument. The goal will be the same as in the first example: tag the two packs according to the object producing the sounds.

The first pack, *Fretless Bass*, has tracks that are all played on a Harley Benton B-550FL fretless bass without any effects added, but with the selected pickup altering. The *Nylon Guitar Single Notes* pack lacks a description, but the sounds are what the title describes, and make up the second portion of the sounds used in this example. Adding the sounds to the application categorizes most sounds accurately, but some of the guitar sounds end up in the bass group. Labeling the two groups with their relevant tag – **Bass** for the first pack and **Guitar** for the latter – illustrates this divergence more clearly.



Number of sounds: 30

Number of groups: 2

		name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	Unverify	Fretless bass, open A, both pickups	wav	1	Bass	Yes	Fretless Bass	54.12	30.87	77.6	36.54	47.83	19.95	52.86
Tag	Verify	Fretless bass, open A, bridge pickup	wav	1	Bass	No	Fretless Bass	50.98	35.24	74.25	33.96	49.38	15.38	51.03
Tag	Verify	Fretless bass, open A, neck pickup	wav	1	Bass	No	Fretless Bass	54.92	28.98	78.03	37.33	48.28	19.62	58.41
Tag	Verify	Fretless bass, open B, both pickups	wav	1	Bass	No	Fretless Bass	58.84	26.71	82.39	37.38	39.3	19.9	53.41
Tag	Verify	Fretless bass, open B, bridge pickup	wav	1	Bass	No	Fretless Bass	54.31	30.59	81.69	31.43	41.09	23.91	53.28
Tag	Verify	Fretless bass, open B, neck pickup	wav	1	Bass	No	Fretless Bass	57.23	27.43	82.38	26.78	37.93	26.46	54.51
Tag	Verify	Fretless bass, open D, both pickups	wav	1	Bass	No	Fretless Bass	51.59	34.15	72.91	33.67	44.58	15.69	51.82
Tag	Verify	Fretless bass, open D, bridge pickup	wav	1	Bass	No	Fretless Bass	47.63	37.78	68.7	36.1	45.57	17.35	50.37
Tag	Verify	Fretless bass, open D, neck pickup	wav	1	Bass	No	Fretless Bass	52.52	31.26	74.59	39.37	44.1	19.07	55.08
Tag	Verify	Fretless bass, open E, both pickups	wav	1	Bass	No	Fretless Bass	58.08	28.96	80.06	32.36	43.97	16.93	54.58
Tag	Verify	Fretless bass, open E, bridge pickup	wav	1	Bass	No	Fretless Bass	55.82	33.71	78.38	32.85	45.56	11.84	51.59
Tag	Verify	Fretless bass, open E, neck pickup	wav	1	Bass	No	Fretless Bass	57.99	28.21	80.2	38.25	41.97	18.93	55.53
Tag	Verify	Fretless bass, open G, both pickups	wav	1	Bass	No	Fretless Bass	46.02	35.77	67.98	33.12	40.55	17.06	51.71
Tag	Verify	Fretless bass, open G, bridge pickup	wav	1	Bass	No	Fretless Bass	38.39	40.57	58.02	41.12	43.44	19.17	51.06
Tag	Verify	Fretless bass, open G, neck pickup	wav	1	Bass	No	Fretless Bass	47.69	33.18	70.57	34.18	40.0	20.91	55.35
Tag	Verify	2_A.wav	wav	1	Bass	No	Nylon Guitar Single Notes	41.61	44.93	67.61	47.4	36.85	33.35	55.08
Tag	Verify	2_Eb.wav	wav	1	Bass	No	Nylon Guitar Single Notes	46.15	44.11	68.02	44.11	44.58	27.99	57.48
Tag	Verify	3_A.wav	wav	1	Bass	No	Nylon Guitar Single Notes	37.68	46.57	59.51	49.69	30.56	33.24	52.56
Tag	Verify	3_C.wav	wav	1	Bass	No	Nylon Guitar Single Notes	42.16	46.36	61.62	52.72	39.0	29.71	54.34
Tag	Verify	3_Eb.wav	wav	1	Bass	No	Nylon Guitar Single Notes	39.65	49.76	65.25	57.65	41.08	34.71	50.93
Tag	Verify	3_Gb.wav	wav	1	Bass	No	Nylon Guitar Single Notes	42.27	43.75	64.14	51.53	28.75	32.64	56.68
Tag	Unverify	4_A.wav	wav	0	Guitar	Yes	Nylon Guitar Single Notes	19.31	54.94	34.51	72.1	35.65	46.95	32.83
Tag	Verify	4_C.wav	wav	1	Bass	No	Nylon Guitar Single Notes	40.45	44.59	59.61	52.49	21.85	33.57	49.12
Tag	Verify	4_Eb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	22.74	49.39	51.34	68.6	32.85	41.26	38.84
Tag	Verify	4_Gb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	18.36	50.5	44.69	68.85	28.67	41.99	33.86
Tag	Verify	5_A.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	21.45	64.8	21.21	74.28	38.42	54.56	21.22
Tag	Verify	5_C.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	26.92	58.13	30.0	75.8	40.68	45.12	29.68
Tag	Verify	5_Eb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	15.8	61.25	26.34	73.99	39.19	50.02	24.71
Tag	Verify	5_Gb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	18.6	61.24	23.71	78.55	39.9	52.64	23.28
Tag	Verify	6_Db.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	27.34	62.13	19.89	72.46	29.01	54.22	29.01

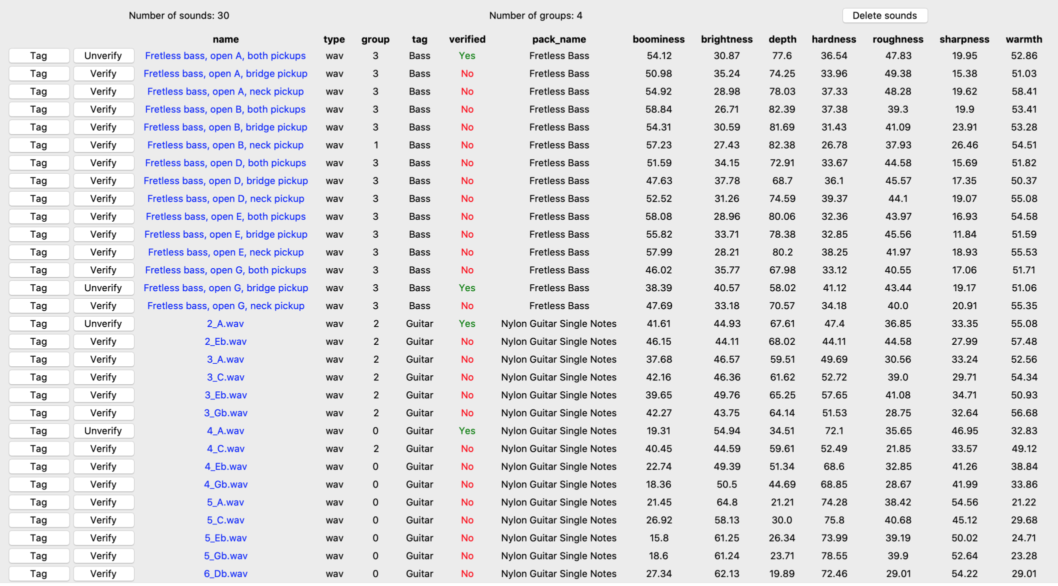
Figure 5.3: Case 2 test data loaded and initially tagged.

This unfinished distribution is a by-product of the groups' correlation to one another and how the sounds themselves are situated. By comparing the ranges for the timbral characteristic values listed in table 5.2 with the ones from table 5.1 in the first example, it is evident that the test data for the current use case consists of sounds more spread out, at least according to how the program distributes them.

Table 5.2: Case 2 timbral characteristics maximum and minimum value ranges

	Fretless Bass	*Nylon Guitar Single Notes*
boominess	20.45	30.35
brightness	13.86	21.05
depth	24.36	48.13
hardness	14.35	34.45
roughness	11.46	22.73
sharpness	14.62	26.57
warmth	8.04	36.26
Average	15.31	31.36

Finishing this example can be done in several ways, but resolving an average score involves trying every path available. Because the sound pool is quite small in this example, it is possible to check each path individually. Coincidentally, all alternatives take **two corrections** to achieve the goal of one tag per sound pack.



Number of sounds: 30

Number of groups: 4

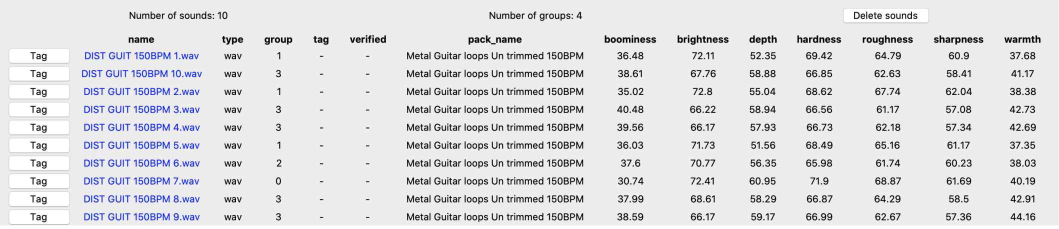
		name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	Unverify	Fretless bass, open A, both pickups	wav	3	Bass	Yes	Fretless Bass	54.12	30.87	77.6	36.54	47.83	19.95	52.86
Tag	Verify	Fretless bass, open A, bridge pickup	wav	3	Bass	No	Fretless Bass	50.98	35.24	74.25	33.96	49.38	15.38	51.03
Tag	Verify	Fretless bass, open A, neck pickup	wav	3	Bass	No	Fretless Bass	54.92	28.98	78.03	37.33	48.28	19.62	58.41
Tag	Verify	Fretless bass, open B, both pickups	wav	3	Bass	No	Fretless Bass	58.84	26.71	82.39	37.38	39.3	19.9	53.41
Tag	Verify	Fretless bass, open B, bridge pickup	wav	3	Bass	No	Fretless Bass	54.31	30.59	81.69	31.43	41.09	23.91	53.28
Tag	Verify	Fretless bass, open B, neck pickup	wav	1	Bass	No	Fretless Bass	57.23	27.43	82.38	26.78	37.93	26.46	54.51
Tag	Verify	Fretless bass, open D, both pickups	wav	3	Bass	No	Fretless Bass	51.59	34.15	72.91	33.67	44.58	15.69	51.82
Tag	Verify	Fretless bass, open D, bridge pickup	wav	3	Bass	No	Fretless Bass	47.63	37.78	68.7	36.1	45.57	17.35	50.37
Tag	Verify	Fretless bass, open D, neck pickup	wav	3	Bass	No	Fretless Bass	52.52	31.26	74.59	39.37	44.1	19.07	55.08
Tag	Verify	Fretless bass, open E, both pickups	wav	3	Bass	No	Fretless Bass	58.08	28.96	80.06	32.36	43.97	16.93	54.58
Tag	Verify	Fretless bass, open E, bridge pickup	wav	3	Bass	No	Fretless Bass	55.82	33.71	78.38	32.85	45.56	11.84	51.59
Tag	Verify	Fretless bass, open E, neck pickup	wav	3	Bass	No	Fretless Bass	57.99	28.21	80.2	38.25	41.97	18.93	55.53
Tag	Verify	Fretless bass, open G, both pickups	wav	3	Bass	No	Fretless Bass	46.02	35.77	67.98	33.12	40.55	17.06	51.71
Tag	Unverify	Fretless bass, open G, bridge pickup	wav	3	Bass	Yes	Fretless Bass	38.39	40.57	58.02	41.12	43.44	19.17	51.06
Tag	Verify	Fretless bass, open G, neck pickup	wav	3	Bass	No	Fretless Bass	47.69	33.18	70.57	34.18	40.0	20.91	55.35
Tag	Unverify	2_A.wav	wav	2	Guitar	Yes	Nylon Guitar Single Notes	41.61	44.93	67.61	47.4	36.85	33.35	55.08
Tag	Verify	2_Eb.wav	wav	2	Guitar	No	Nylon Guitar Single Notes	46.15	44.11	68.02	44.11	44.58	27.99	57.48
Tag	Verify	3_A.wav	wav	2	Guitar	No	Nylon Guitar Single Notes	37.68	46.57	59.51	49.69	30.56	33.24	52.56
Tag	Verify	3_C.wav	wav	2	Guitar	No	Nylon Guitar Single Notes	42.16	46.36	61.62	52.72	39.0	29.71	54.34
Tag	Verify	3_Eb.wav	wav	2	Guitar	No	Nylon Guitar Single Notes	39.65	49.76	65.25	57.65	41.08	34.71	50.93
Tag	Verify	3_Gb.wav	wav	2	Guitar	No	Nylon Guitar Single Notes	42.27	43.75	64.14	51.53	28.75	32.64	56.68
Tag	Unverify	4_A.wav	wav	0	Guitar	Yes	Nylon Guitar Single Notes	19.31	54.94	34.51	72.1	35.65	46.95	32.83
Tag	Verify	4_C.wav	wav	2	Guitar	No	Nylon Guitar Single Notes	40.45	44.59	59.61	52.49	21.85	33.57	49.12
Tag	Verify	4_Eb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	22.74	49.39	51.34	68.6	32.85	41.26	38.84
Tag	Verify	4_Gb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	18.36	50.5	44.69	68.85	28.67	41.99	33.86
Tag	Verify	5_A.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	21.45	64.8	21.21	74.28	38.42	54.56	21.22
Tag	Verify	5_C.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	26.92	58.13	30.0	75.8	40.68	45.12	29.68
Tag	Verify	5_Eb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	15.8	61.25	26.34	73.99	39.19	50.02	24.71
Tag	Verify	5_Gb.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	18.6	61.24	23.71	78.55	39.9	52.64	23.28
Tag	Verify	6_Db.wav	wav	0	Guitar	No	Nylon Guitar Single Notes	27.34	62.13	19.89	72.46	29.01	54.22	29.01

Figure 5.4: Case 2 finalized in one of several different ways.

5.5 Case 3 – Scope

An intriguing quality of the ML algorithms used to tag sounds automatically in the application is how they execute differently depending on the scope of the data points in a given input series. Stripping the sound pool in the first example – where the application immediately groups the loaded samples correctly – to a single pack, one could expect the program to identify that pack as a single group.

That is not quite the case, as can be seen in fig. 5.5, where the *Metal Guitar loops Un trimmed 150BPM* pack is inserted by itself. This perchance unexpected event happens due to the nature of how the clustering algorithms function when handling differently sized and spread out data sets. One way to potentially regulate this action would be to fine-tune the precision at which the application generates the sound groupings. The tolerance value is such a calibration but, although publicly available for reconfiguration as a function argument labeled *tol* [54, 53], remains untouched and set to its default behavior in this implementation. Tweaking this attribute could help the program adjust to certain situations better and possibly improve the performance for specific cases; the further work segment (section 7.1) examines this proposal further.

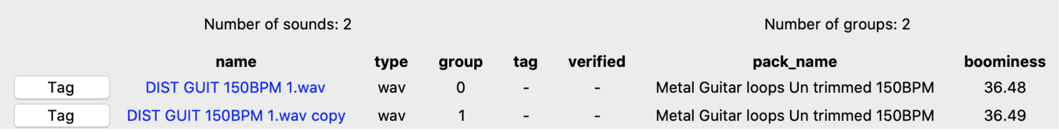


Number of sounds: 10 Number of groups: 4

	name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	DIST GUIT 150BPM 1.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	36.48	72.11	52.35	69.42	64.79	60.9	37.68
Tag	DIST GUIT 150BPM 10.wav	wav	3	-	-	Metal Guitar loops Un trimmed 150BPM	38.61	67.76	58.88	66.85	62.63	58.41	41.17
Tag	DIST GUIT 150BPM 2.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	35.02	72.8	55.04	68.62	67.74	62.04	38.38
Tag	DIST GUIT 150BPM 3.wav	wav	3	-	-	Metal Guitar loops Un trimmed 150BPM	40.48	66.22	58.94	66.56	61.17	57.08	42.73
Tag	DIST GUIT 150BPM 4.wav	wav	3	-	-	Metal Guitar loops Un trimmed 150BPM	39.56	66.17	57.93	66.73	62.18	57.34	42.69
Tag	DIST GUIT 150BPM 5.wav	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	36.03	71.73	51.56	68.49	65.16	61.17	37.35
Tag	DIST GUIT 150BPM 6.wav	wav	2	-	-	Metal Guitar loops Un trimmed 150BPM	37.6	70.77	56.35	65.98	61.74	60.23	38.03
Tag	DIST GUIT 150BPM 7.wav	wav	0	-	-	Metal Guitar loops Un trimmed 150BPM	30.74	72.41	60.95	71.9	68.87	61.69	40.19
Tag	DIST GUIT 150BPM 8.wav	wav	3	-	-	Metal Guitar loops Un trimmed 150BPM	37.99	68.61	58.29	66.87	64.29	58.5	42.91
Tag	DIST GUIT 150BPM 9.wav	wav	3	-	-	Metal Guitar loops Un trimmed 150BPM	38.59	66.17	59.17	66.99	62.67	57.36	44.16

Figure 5.5: Pack from Case 1 loaded separately into the application.

Expanding on this topic: when duplicating a single sound and adding both the original and copied one to the application, the two sounds will end up belonging to the same group. Furthermore, as accomplished in fig. 5.6 by copying and modifying an individual Freesound metadata file, if two sounds differ even in a single magnitude for one of the timbral attributes, the resulting sum of groups will be two.



Number of sounds: 2 Number of groups: 2

	name	type	group	tag	verified	pack_name	boominess
Tag	DIST GUIT 150BPM 1.wav	wav	0	-	-	Metal Guitar loops Un trimmed 150BPM	36.48
Tag	DIST GUIT 150BPM 1.wav copy	wav	1	-	-	Metal Guitar loops Un trimmed 150BPM	36.49

Figure 5.6: A single sound plus its modified clone loaded into the application.

One can think of this concept as a 'zoom effect' in the dimensional space where the resulting picture is the same whether the parameters are large or small as long as the proportions remain constant. Reducing the dimensions of predicted timbral characteristics from seven to two enables a relatively effective visualization of this type of proportion-based groupings relative to the volume of the sounds added to the program. In two dimensions, this means that the size of the x and y values (for example, *boominess* and *brightness*) is redundant when distributing the groups, and only the relative distance between the data points on the coordinate axes is of importance. This phenomenon is also, in some instances, comparable to the *Droste effect*, where a picture appears within itself one or multiple times, as shown in fig. 5.7.

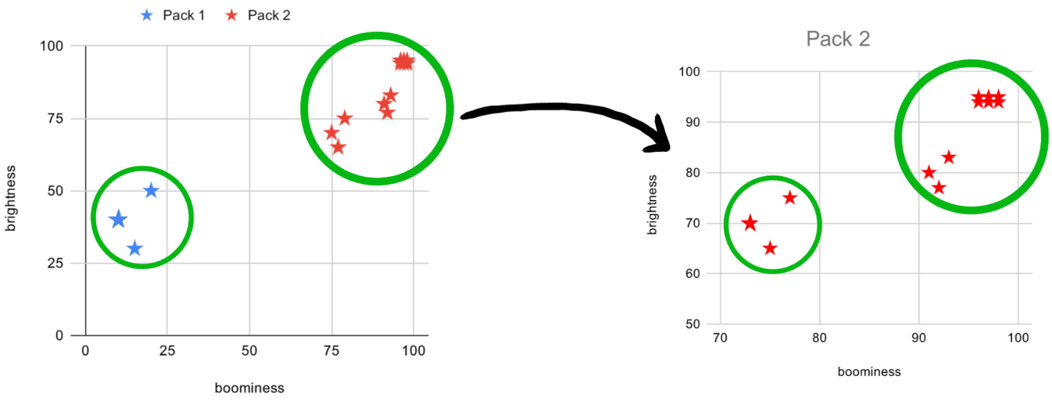


Figure 5.7: An illustration of the *Droste effect* in the application when changing the scope for sounds added to the program. The green circles represent the determined groups found by the application. Pack 2 has the same layout of sounds in itself as it has with Pack 1, hence the groupings being the same when processed together or Pack 2 by itself, even though the scale changes in the two pictures.

5.6 Case 4 – Scalability

The last example mimics a more realistic usage of the application, with six different packs on a smaller scale, forming a total of 30 sounds. The test data is made up of instrumental, electronic, and ambient sounds, stretching over a broader sound domain than previous examples. As before, the goal is to give every sound in each pack a descriptive group tag that corresponds to its sounds' source.

The first pack, *A Harpsicord Dream* [64], consists of harp-like glissandos played on a Roland XP-10 keyboard. The two next packs used, *Basic Tech-Trance* [9] and *Bass Bars 1* [20], have beats that would suit as a foundation for loops in techno songs. *Sharpening Knives* [59] and *Static and Radio Sounds* [18] contain samples of audio from everyday life. *Tom-Tom Grooves* [23] is the last pack included in the test data and has short loop-friendly drum rhythms. It is noticeable when examining the averages in table 5.3 that the estimated timbral characteristics map some of the chosen packs as more compact and other as more spread out.

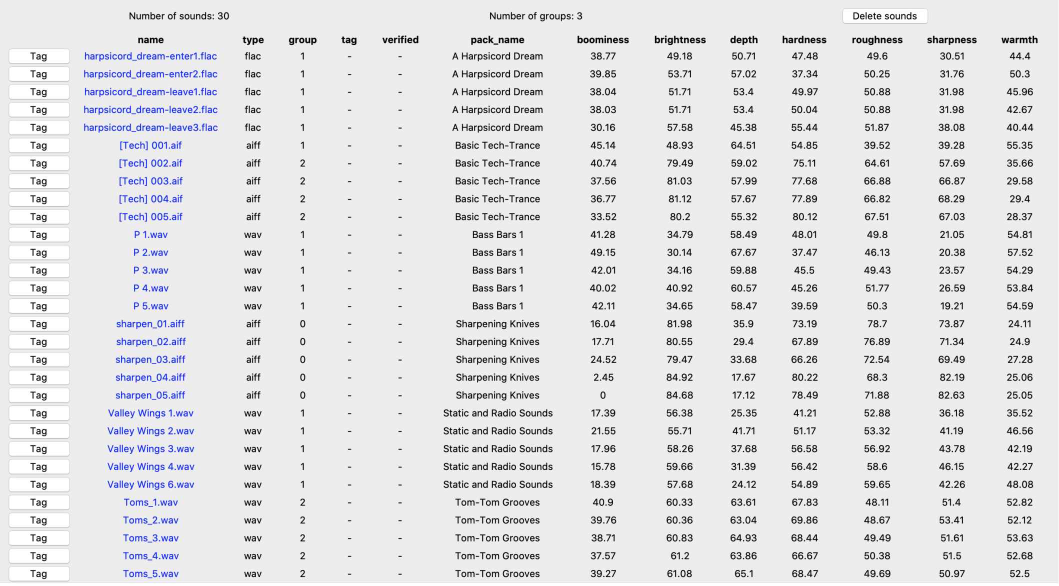
When handing the test data to the application for analysis, it does a decent job of organizing the sounds; fig. 5.8 displays this initial state. The number of groups predicted is slightly modest, consequently mixing and joining some packs together. Reaching a finalized state of fig. 5.9 takes an unfixed amount of user interactions – explained in section 5.2. Because the number of revisions has grown compared to other use cases, it is laborious to cover every alternative and, therefore, difficult to work out an exact median value for the number of alterations needed.

Table 5.3: Case 4 timbral characteristics maximum and minimum value ranges

	Instrumental	
	A Harpsicord Dream	*Tom-Tom Grooves*
boominess	9.69	3.34
brightness	8.40	0.87
depth	11.64	2.06
hardness	18.10	3.19
roughness	2.27	2.27
sharpness	7.57	2.44
warmth	9.87	1.51
Average	9.65	2.24

	Electronic	
	Basic Tech-Trance	*Bass Bars 1*
boominess	11.63	9.13
brightness	32.19	10.78
depth	9.18	9.21
hardness	25.27	10.53
roughness	27.99	5.64
sharpness	29.00	7.38
warmth	26.97	3.68
Average	23.18	8.05

	Ambient	
	Sharpening Knives	*Static and Radio Sounds*
boominess	24.52	5.77
brightness	5.45	3.95
depth	18.77	17.58
hardness	13.96	15.38
roughness	10.40	6.76
sharpness	13.14	9.97
warmth	3.17	12.56
Average	12.77	10.28

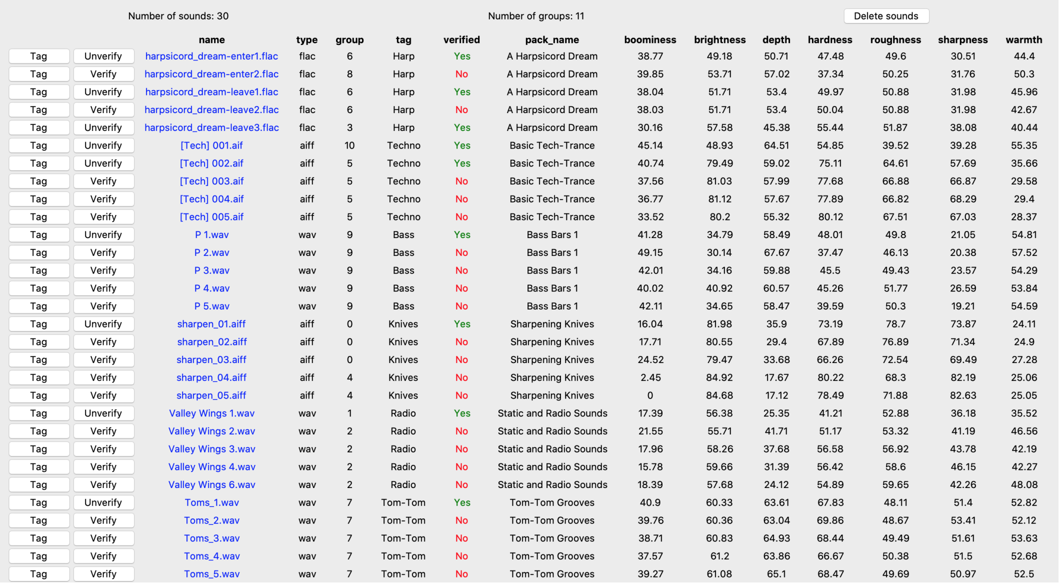


Number of sounds: 30

Number of groups: 3

	name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	harpsicord_dream-enter1.flac	flac	1	-	-	A Harpsicord Dream	38.77	49.18	50.71	47.48	49.6	30.51	44.4
Tag	harpsicord_dream-enter2.flac	flac	1	-	-	A Harpsicord Dream	39.85	53.71	57.02	37.34	50.25	31.76	50.3
Tag	harpsicord_dream-leave1.flac	flac	1	-	-	A Harpsicord Dream	38.04	51.71	53.4	49.97	50.88	31.98	45.96
Tag	harpsicord_dream-leave2.flac	flac	1	-	-	A Harpsicord Dream	38.03	51.71	53.4	50.04	50.88	31.98	42.67
Tag	harpsicord_dream-leave3.flac	flac	1	-	-	A Harpsicord Dream	30.16	57.58	45.38	55.44	51.87	38.08	40.44
Tag	[Tech] 001.aif	aiff	1	-	-	Basic Tech-Trance	45.14	48.93	64.51	54.85	39.52	39.28	55.35
Tag	[Tech] 002.aif	aiff	2	-	-	Basic Tech-Trance	40.74	79.49	59.02	75.11	64.61	57.69	35.66
Tag	[Tech] 003.aif	aiff	2	-	-	Basic Tech-Trance	37.56	81.03	57.99	77.68	66.88	66.87	29.58
Tag	[Tech] 004.aif	aiff	2	-	-	Basic Tech-Trance	36.77	81.12	57.67	77.89	66.82	68.29	29.4
Tag	[Tech] 005.aif	aiff	2	-	-	Basic Tech-Trance	33.52	80.2	55.32	80.12	67.51	67.03	28.37
Tag	P 1.wav	wav	1	-	-	Bass Bars 1	41.28	34.79	58.49	48.01	49.8	21.05	54.81
Tag	P 2.wav	wav	1	-	-	Bass Bars 1	49.15	30.14	67.67	37.47	46.13	20.38	57.52
Tag	P 3.wav	wav	1	-	-	Bass Bars 1	42.01	34.16	59.88	45.5	49.43	23.57	54.29
Tag	P 4.wav	wav	1	-	-	Bass Bars 1	40.02	40.92	60.57	45.26	51.77	26.59	53.84
Tag	P 5.wav	wav	1	-	-	Bass Bars 1	42.11	34.65	58.47	39.59	50.3	19.21	54.59
Tag	sharpen_01.aiff	aiff	0	-	-	Sharpening Knives	16.04	81.98	35.9	73.19	78.7	73.87	24.11
Tag	sharpen_02.aiff	aiff	0	-	-	Sharpening Knives	17.71	80.55	29.4	67.89	76.89	71.34	24.9
Tag	sharpen_03.aiff	aiff	0	-	-	Sharpening Knives	24.52	79.47	33.68	66.26	72.54	69.49	27.28
Tag	sharpen_04.aiff	aiff	0	-	-	Sharpening Knives	2.45	84.92	17.67	80.22	68.3	82.19	25.06
Tag	sharpen_05.aiff	aiff	0	-	-	Sharpening Knives	0	84.68	17.12	78.49	71.88	82.63	25.05
Tag	Valley Wings 1.wav	wav	1	-	-	Static and Radio Sounds	17.39	56.38	25.35	41.21	52.88	36.18	35.52
Tag	Valley Wings 2.wav	wav	1	-	-	Static and Radio Sounds	21.55	55.71	41.71	51.17	53.32	41.19	46.56
Tag	Valley Wings 3.wav	wav	1	-	-	Static and Radio Sounds	17.96	58.26	37.68	56.58	56.92	43.78	42.19
Tag	Valley Wings 4.wav	wav	1	-	-	Static and Radio Sounds	15.78	59.66	31.39	56.42	58.6	46.15	42.27
Tag	Valley Wings 6.wav	wav	1	-	-	Static and Radio Sounds	18.39	57.68	24.12	54.89	59.65	42.26	48.08
Tag	Toms_1.wav	wav	2	-	-	Tom-Tom Grooves	40.9	60.33	63.61	67.83	48.11	51.4	52.82
Tag	Toms_2.wav	wav	2	-	-	Tom-Tom Grooves	39.76	60.36	63.04	69.86	48.67	53.41	52.12
Tag	Toms_3.wav	wav	2	-	-	Tom-Tom Grooves	38.71	60.83	64.93	68.44	49.49	51.61	53.63
Tag	Toms_4.wav	wav	2	-	-	Tom-Tom Grooves	37.57	61.2	63.86	66.67	50.38	51.5	52.68
Tag	Toms_5.wav	wav	2	-	-	Tom-Tom Grooves	39.27	61.08	65.1	68.47	49.69	50.97	52.5

Figure 5.8: Case 4 test data initially loaded into the application.



Number of sounds: 30

Number of groups: 11

		name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	Unverify	harpsicord_dream-enter1.flac	flac	6	Harp	Yes	A Harpsicord Dream	38.77	49.18	50.71	47.48	49.6	30.51	44.4
Tag	Verify	harpsicord_dream-enter2.flac	flac	8	Harp	No	A Harpsicord Dream	39.85	53.71	57.02	37.34	50.25	31.76	50.3
Tag	Unverify	harpsicord_dream-leave1.flac	flac	6	Harp	Yes	A Harpsicord Dream	38.04	51.71	53.4	49.97	50.88	31.98	45.96
Tag	Verify	harpsicord_dream-leave2.flac	flac	6	Harp	No	A Harpsicord Dream	38.03	51.71	53.4	50.04	50.88	31.98	42.67
Tag	Unverify	harpsicord_dream-leave3.flac	flac	3	Harp	Yes	A Harpsicord Dream	30.16	57.58	45.38	55.44	51.87	38.08	40.44
Tag	Unverify	[Tech] 001.aif	aiff	10	Techno	Yes	Basic Tech-Trance	45.14	48.93	64.51	54.85	39.52	39.28	55.35
Tag	Unverify	[Tech] 002.aif	aiff	5	Techno	Yes	Basic Tech-Trance	40.74	79.49	59.02	75.11	64.61	57.69	35.66
Tag	Verify	[Tech] 003.aif	aiff	5	Techno	No	Basic Tech-Trance	37.56	81.03	57.99	77.68	66.88	66.87	29.58
Tag	Verify	[Tech] 004.aif	aiff	5	Techno	No	Basic Tech-Trance	36.77	81.12	57.67	77.89	66.82	68.29	29.4
Tag	Verify	[Tech] 005.aif	aiff	5	Techno	No	Basic Tech-Trance	33.52	80.2	55.32	80.12	67.51	67.03	28.37
Tag	Unverify	P 1.wav	wav	9	Bass	Yes	Bass Bars 1	41.28	34.79	58.49	48.01	49.8	21.05	54.81
Tag	Verify	P 2.wav	wav	9	Bass	No	Bass Bars 1	49.15	30.14	67.67	37.47	46.13	20.38	57.52
Tag	Verify	P 3.wav	wav	9	Bass	No	Bass Bars 1	42.01	34.16	59.88	45.5	49.43	23.57	54.29
Tag	Verify	P 4.wav	wav	9	Bass	No	Bass Bars 1	40.02	40.92	60.57	45.26	51.77	26.59	53.84
Tag	Verify	P 5.wav	wav	9	Bass	No	Bass Bars 1	42.11	34.65	58.47	39.59	50.3	19.21	54.59
Tag	Unverify	sharpen_01.aiff	aiff	0	Knives	Yes	Sharpening Knives	16.04	81.98	35.9	73.19	78.7	73.87	24.11
Tag	Verify	sharpen_02.aiff	aiff	0	Knives	No	Sharpening Knives	17.71	80.55	29.4	67.89	76.89	71.34	24.9
Tag	Verify	sharpen_03.aiff	aiff	0	Knives	No	Sharpening Knives	24.52	79.47	33.68	66.26	72.54	69.49	27.28
Tag	Verify	sharpen_04.aiff	aiff	4	Knives	No	Sharpening Knives	2.45	84.92	17.67	80.22	68.3	82.19	25.06
Tag	Verify	sharpen_05.aiff	aiff	4	Knives	No	Sharpening Knives	0	84.68	17.12	78.49	71.88	82.63	25.05
Tag	Unverify	Valley Wings 1.wav	wav	1	Radio	Yes	Static and Radio Sounds	17.39	56.38	25.35	41.21	52.88	36.18	35.52
Tag	Verify	Valley Wings 2.wav	wav	2	Radio	No	Static and Radio Sounds	21.55	55.71	41.71	51.17	53.32	41.19	46.56
Tag	Verify	Valley Wings 3.wav	wav	2	Radio	No	Static and Radio Sounds	17.96	58.26	37.68	56.58	56.92	43.78	42.19
Tag	Verify	Valley Wings 4.wav	wav	2	Radio	No	Static and Radio Sounds	15.78	59.66	31.39	56.42	58.6	46.15	42.27
Tag	Verify	Valley Wings 6.wav	wav	2	Radio	No	Static and Radio Sounds	18.39	57.68	24.12	54.89	59.65	42.26	48.08
Tag	Unverify	Toms_1.wav	wav	7	Tom-Tom	Yes	Tom-Tom Grooves	40.9	60.33	63.61	67.83	48.11	51.4	52.82
Tag	Verify	Toms_2.wav	wav	7	Tom-Tom	No	Tom-Tom Grooves	39.76	60.36	63.04	69.86	48.67	53.41	52.12
Tag	Verify	Toms_3.wav	wav	7	Tom-Tom	No	Tom-Tom Grooves	38.71	60.83	64.93	68.44	49.49	51.61	53.63
Tag	Verify	Toms_4.wav	wav	7	Tom-Tom	No	Tom-Tom Grooves	37.57	61.2	63.86	66.67	50.38	51.5	52.68
Tag	Verify	Toms_5.wav	wav	7	Tom-Tom	No	Tom-Tom Grooves	39.27	61.08	65.1	68.47	49.69	50.97	52.5

Figure 5.9: Case 4 finalized in one of several different ways.

6 Discussion

The developed application fulfills the goal of enabling subjective tagging of sounds while minimizing the manual effort needed to do so, hence proving that the proposed system is a viable and working solution. Since there exist no prior comparable solutions, at least not found by the author, it is impossible to measure the performance of the program to any previous values.

The end product is a standalone application capable of tagging sounds according to different opinions and continuously improves with usage. The solution is in itself quite simple and relatively easy to adapt. When narrowing down the codebase to units responsible for actual calculations related to sound similarity, tagging, and grouping, the result is only a couple of hundred lines of code. Still, the intended purpose of the solution is to function as a complement to other existing systems. This expansion could be in the form of a plugin, an extension, or some other similar variation.

Many of the components in the implemented program are competent and well-functioning. The sound similarity system and the automatic tagging functionalities perform well in all of the use cases evaluated in chapter 5. Regardless of how specific or generic the desired tag distribution might be, it should always be possible to achieve the sought after result using the application as a platform in its current form.

As described in the evaluation process, there are many ways of reaching a finalized state where all sounds are correctly labeled, depending on what order sounds are tagged. Each available path requires a specific amount of corrections, meaning that some paths need more manual effort to complete than others. Therefore, the course taken by the user might end up being the most effortless one, the most tedious one, or something in between.

The application is deterministic in the sense that repeating the same actions always result in the same outcome, which makes it stable and possibly easier to integrate with other systems. However, the program is inconsistent with tag distributions and groups, as the corresponding sum of groups and how they encompass the sounds can vary in a fixed arrangement of labels. Tagging multiple sounds in one way might conclude in a few groups, while in another, the number of groups could be doubling.

While developing and formulating the proposed solution, the author of this thesis realized that many different kinds of elements play a role when forming a subjective categorization and management system for sounds. The fundamental dilemma concerning subjectivity is the most difficult to understand and define fully. Because subjectivity is abstract and can change from situation to situation, it is difficult to make assumptions around it and define rules. Subjectivity is also presumptively exposed to peer pressure; if someone describes a sound as, e.g., 'sad', someone else will probably perceive the sound the same way. If the latter person heard the sound in isolation, they might have labeled the sound differently, unaffected by the other person's predefined description. These kinds of human factors can make it challenging to pinpoint problems when measuring the effectiveness of categorization and management systems for sounds, as the user could have trouble understanding or describing the problem themselves. When annotating sounds, adapting one's perception to someone else's might work for some cases, but resisting this tendency and creating a personal collection of labels instead will hypothetically be beneficial in the long run.

7 Conclusion

Designing sounds and building sound landscapes is a challenging yet inspiring occupation. Dealing with large data sets and having numerous influencing factors are some of the concerns making the situation more demanding. Utilizing sound libraries containing SFX is one way to make the process more approachable and productive. However, many aspects still need improvement to make the means of working with the libraries more gratifying and compelling. No standard exists for categorizing and classifying sounds, meaning that most libraries have a custom labeling system, making it harder to develop programs capable of handling all variations. The tools used for finding and managing sounds among the libraries are also far from perfect.

Extensive research has taken place in automatic sound identification and objective audio labeling, which could help classify sounds more accurately and expressly. The research results and developed systems thereof could assist the library creators in the tagging process, making related sounds across multiple libraries labeled similarly. This procedure could aid in establishing a standard of how to describe sounds concretely, such as the sound source of the audio.

Far less experimentation has ensued with systems aimed towards individual sound annotation with personal descriptors, capable of adjusting to different users' perception. Humans often describe signals perceived through the senses with subjective words, e.g., that something is scary. This impression holds for audio as well: one person might interpret a sound as 'happy' when another defines it as 'smooth.'

The purpose of this thesis is to present and evaluate an approach capable of categorizing sounds with subjective tags. The developed solution is a flexible stand-application application, which continuously improves with usage and minimizes the manual effort needed to label sounds utilizing a sound similarity and automatic tagging system. Although the program functions well on its own, the eventual intent is to have the system integrated with existing categorization methods and management tools. This procedure could hopefully improve the work process for sound designers using sound libraries and also spark ideas for new programs or extensions to systems regarding the subject.

7.1 Further work

There is plenty of room for improvement and optimization concerning the application described in this thesis, as the program in its current stage only ranks as a prototype. Many of these advancements relate to the performance or usability aspects of the application, while others propose enhancements to the underlying logic.

One of the main issues that prevent the present version of the design from becoming a distributable application or integrated into existing systems is the lack of functionality for multiple tags per sound. Multi-labeling coheres substantially more with real-world use cases than only allowing individual tags, as a single label is seldom enough to describe a sound fully incorporating all its qualities. However, managing multiple tags require a different approach capable of handling advanced situations, such as overlapping and intertwining labels. Extending the described implementation to support this feature would probably lead to a complete rewrite, possibly preserving some of the current system design.

Another central subject needing further work is the sound analysis part of the application regarding both performance and accuracy. The sound similarity concept described in section 4.4 relies on its building blocks being accurate, which are the seven predicted timbral characteristics used throughout the program. The original purpose of the values is to assist in automatic labeling of sounds, and adapting them to serve as subjectivity measurement variables could lead to inconsistent and inadequate results. Also, the absence of dimensionality reduction and other forms of optimization procedures means some of the characteristics might be redundant.

Furthermore, when processing extensive data sets, the analysis step takes a considerable amount of time to finish. As of now, the application processes sounds one after the other, resulting in the operation taking longer for each additional sound added to the pool of sounds to analyze. This issue is a significant bottleneck performance problem preventing the system from scaling. Some form of parallelization, changing the single-threaded sequence of events, would cut down the execution time significantly. Massive data sets also slow down the program considerably, preventing any form of testing when the number of included sounds is high. This issue makes it difficult to check the performance scalability of the implemented ML algorithms.

Both the ML algorithms used for grouping sounds accomplish their shared goal of categorizing sounds satisfactory in their current form. Nevertheless, as touched upon in section 5.5, the algorithms could benefit from revision, since the working configurations incidentally use the default values defined in the package. One advantageous or disadvantageous outcome – depending on the situation – of using the predefined versions of the algorithms is how they directly adapt to changing the scale of how widespread added sounds are. Adding numerous closely resembling sounds could result in the same initial group predictions as a collection of more spread out sounds. Despite its situational usefulness, this behavior is currently persistent at all times. Having no control over this dynamical tolerance means that it activates for situations where it is a hindrance, as well as instances where it is beneficial. Tweaking and controlling the precision at which the algorithms draw their group borders might improve the performance in many cases. The degree at which this transpires and what the optimal tolerance comes out as is probably dependant on each user's sound perception.

How sounds are tagged and retagged might also be slightly confusing to the user. The tagging process currently depends on the sounds' verification status and decides the appropriate choices accordingly on its own in encapsulation. This implementation is a working solution in its own right but has its pros and cons. The program takes responsibility for most of the choices concerning tagging at the cost of the user having less control. Eliminating the verification property and implementing separate functionalities for creating and modifying tags could make the application more approachable. Some form of internal verification system would still need to be present to keep track of the confirmed tagged sounds. However, this means handling edge cases appropriately and redesigning the UI accordingly. Regarding updating the visuals, this would naturally occur when implementing the proposed solution into an existing system. The design built with the *tkinter* package works fine for presenting the prototype stage of the program but will need an overhaul to meet the industry standard.

Svensk sammanfattning

Subjektiva taggar vid personlig kategorisering av ljud

Inledning

Människor intar dagligen en stor dos av ljud i dagens värld. Större helheter av ljud byggs upp med ljudeffekter, jinglar, loopar och andra ljudfragment. Tillsammans används de för att skapa ljudlandskap avsedda för olika ändamål. Skapandeprocessen och förfaringssättet bakom alla de här ljuden är mindre påtagligt i vardagen och få är helt insatta i hur ljuden blir till.

Yrket som ljuddesigner innebär att skapa och hantera ljud avsedda för ett visst ändamål eller en viss produkt. Arbetet förutsätter mycket kreativitet och fantasi för att få fram de ljud som situationen kräver. Verktygen som ljuddesigner använder sig av för att hantera ljud måste vara så lätta som möjligt att förstå för att arbetsprocessen ska kännas inspirerande och effektiv. Annars kan det uppstå frustrerande situationer där motivationen lätt försvinner.

I huvudsak finns det två olika tillvägagångssätt för att skapa ljud ämnade för film, spel och andra medier. Det första alternativet är att skapa alla ljud från grunden genom att spela in dem i en studio med lämplig utrustning. Man har då möjlighet att utforma exakt sådana ljud man vill ha, men processen kan bli både dyr och tidskrävande eftersom flera ljud ofta behövs för att tillsammans kombineras till ett tillfredsställande resultat.

Den andra varianten, som är i fokus i den här avhandlingen, är att använda sig av befintliga ljudbibliotek med färdigt inspelade och kategoriserade ljud. Biblioteken kan innehålla ett fåtal eller flera tusen ljud, där likheten mellan ljuden varierar. Problemet med det här alternativet är att det inte finns en etablerad standard för hur ljuden ska indelas och klassificeras. Indelningarna är vanligen gjorda manuellt med olika former av konkreta taggar som redogör för ljudkällan, vad som händer i ljudklippet, eller liknande beskrivningar. En rad sökverktyg finns tillgängliga för att hitta ljud bland ljudbiblioteken, men på grund av variationen mellan de olika samlingarnas kategoriseringssystem blir sökningsresultatet ofta mediokert.

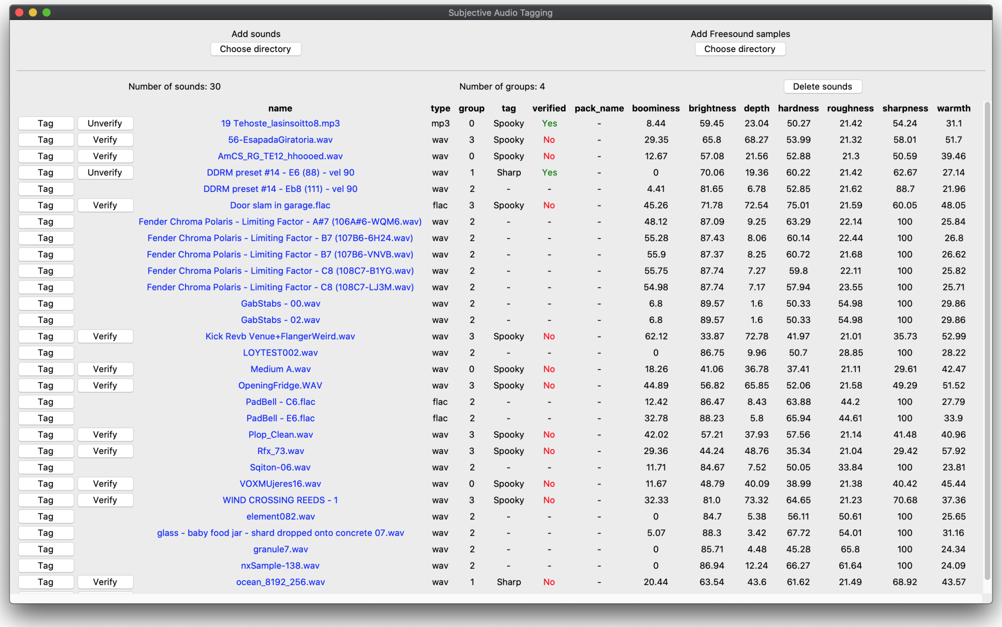
En genomgående brist i sökverktygen och ljudbibliotekens kategoriseringssystem är att de inte tar subjektivitet i beaktande. Människor beskriver ofta ljud i subjektiva ord, vilket tyder på att personliga taggar skulle kunna vara till stor hjälp för att klassificera och hantera ljud. Problemet med att implementera subjektiva taggar i ett kategoriseringssystem är att de är individuella; samma ljud kan ha olika beskrivningar beroende på vem man frågar. Processen att förse ljud med subjektiva beskrivningar borde således göras skilt för varje användare, men att manuellt tagga ljud är både tidskrävande, arbetsdrygt och felbenäget.

Mål

Målet med den här avhandlingen är att ta fram ett lösningsförslag på hur subjektivitet kan implementeras i sökverktyg ämnade för ljudbibliotek. Lösningsförslaget som utvecklats är ett fungerande program, men tjänar mera som ett utkast till hur kategorisering med subjektiva beskrivningar kan se ut. Programmet ger användare möjligheten att med sina egna subjektiva ljudupplevelser kategorisera ljud genom att märka dem med subjektiva taggar. Varje användare kan alltså klassificera ljud enligt eget tycke och smak. För att underlätta det manuella arbetet som krävs söks liknande ljud automatiskt upp med hjälp av maskininlärning och taggas enligt en användares preferenser.

Avsikten med att inkorporera subjektivitet i sökverktygen är att underlätta det dagliga arbetet som exempelvis en ljuddesigner gör genom att erbjuda ett alternativt sätt att organisera och söka fram ljud. Inkorporeras lösningsförslaget vidare och utvecklas till en färdig produkt kan helheter av ljud förhoppningsvis byggas upp på ett bättre och smidigare sätt, vilket därmed också sparar tid och pengar för de som anställer ljuddesigners.

Tillämpning



Number of sounds: 30

Number of groups: 4

		name	type	group	tag	verified	pack_name	boominess	brightness	depth	hardness	roughness	sharpness	warmth
Tag	Unverify	19 Tehoste_lasinsoitto8.mp3	mp3	0	Spooky	Yes	-	8.44	59.45	23.04	50.27	21.42	54.24	31.1
Tag	Verify	56-EsapadaGiratoria.wav	wav	3	Spooky	No	-	29.35	65.8	68.27	53.99	21.32	58.01	51.7
Tag	Verify	AmCS_RG_TE12_hhoooed.wav	wav	0	Spooky	No	-	12.67	57.08	21.56	52.88	21.3	50.59	39.46
Tag	Unverify	DDRM preset #14 - E6 (88) - vel 90	wav	1	Sharp	Yes	-	0	70.06	19.36	60.22	21.42	62.67	27.14
Tag		DDRM preset #14 - Eb8 (111) - vel 90	wav	2	-	-	-	4.41	81.65	6.78	52.85	21.62	88.7	21.96
Tag	Verify	Door slam in garage.flac	flac	3	Spooky	No	-	45.26	71.78	72.54	75.01	21.59	60.05	48.05
Tag		Fender Chroma Polaris - Limiting Factor - A#7 (106A#6-WQM6.wav)	wav	2	-	-	-	48.12	87.09	9.25	63.29	22.14	100	25.84
Tag		Fender Chroma Polaris - Limiting Factor - B7 (107B6-6H24.wav)	wav	2	-	-	-	55.28	87.43	8.06	60.14	22.44	100	26.8
Tag		Fender Chroma Polaris - Limiting Factor - B7 (107B6-VNVB.wav)	wav	2	-	-	-	55.9	87.37	8.25	60.72	21.68	100	26.62
Tag		Fender Chroma Polaris - Limiting Factor - C8 (108C7-B1YG.wav)	wav	2	-	-	-	55.75	87.74	7.27	59.8	22.11	100	25.82
Tag		Fender Chroma Polaris - Limiting Factor - C8 (108C7-LJ3M.wav)	wav	2	-	-	-	54.98	87.74	7.17	57.94	23.55	100	25.71
Tag		GabStabs - 00.wav	wav	2	-	-	-	6.8	89.57	1.6	50.33	54.98	100	29.86
Tag		GabStabs - 02.wav	wav	2	-	-	-	6.8	89.57	1.6	50.33	54.98	100	29.86
Tag	Verify	Kick Revb Venue+FlangerWeird.wav	wav	3	Spooky	No	-	62.12	33.87	72.78	41.97	21.01	35.73	52.99
Tag		LOYTEST002.wav	wav	2	-	-	-	0	86.75	9.96	50.7	28.85	100	28.22
Tag	Verify	Medium A.wav	wav	0	Spooky	No	-	18.26	41.06	36.78	37.41	21.11	29.61	42.47
Tag	Verify	OpeningFridge.WAV	wav	3	Spooky	No	-	44.89	56.82	65.85	52.06	21.58	49.29	51.52
Tag		PadBell - C6.flac	flac	2	-	-	-	12.42	86.47	8.43	63.88	44.2	100	27.79
Tag		PadBell - E6.flac	flac	2	-	-	-	32.78	88.23	5.8	65.94	44.61	100	33.9
Tag	Verify	Plop_Clean.wav	wav	3	Spooky	No	-	42.02	57.21	37.93	57.56	21.14	41.48	40.96
Tag	Verify	Rfx_73.wav	wav	3	Spooky	No	-	29.36	44.24	48.76	35.34	21.04	29.42	57.92
Tag		Sqiton-06.wav	wav	2	-	-	-	11.71	84.67	7.52	50.05	33.84	100	23.81
Tag	Verify	VOXMUjeres16.wav	wav	0	Spooky	No	-	11.67	48.79	40.09	38.99	21.38	40.42	45.44
Tag	Verify	WIND CROSSING REEDS - 1	wav	3	Spooky	No	-	32.33	81.0	73.32	64.65	21.23	70.68	37.36
Tag		element082.wav	wav	2	-	-	-	0	84.7	5.38	56.11	50.61	100	25.65
Tag		glass - baby food jar - shard dropped onto concrete 07.wav	wav	2	-	-	-	5.07	88.3	3.42	67.72	54.01	100	31.16
Tag		granule7.wav	wav	2	-	-	-	0	85.71	4.48	45.28	65.8	100	24.34
Tag		nxSample-138.wav	wav	2	-	-	-	0	86.94	12.24	66.27	61.64	100	24.09
Tag	Verify	ocean_8192_256.wav	wav	1	Sharp	No	-	20.44	63.54	43.6	61.62	21.49	68.92	43.57

Figur 7.1: Bild på det utvecklade programmet i användning.

Som lösningsförslag till de nämnda problemen med subjektivitet vid ljudklassificering beskrivs ett utvecklat program som är kapabelt att justera sig enligt en användare. Programmet är gjort i Python och syftet är att demonstrera hur man kan minska det manuella arbete som krävs för att kategorisera ljud efter de preferenser och ljuduppfattningar varje enskild användare har. Upplägget och designen kan ses i fig. 7.1. Valda ljud som läggs till i programmet analyseras och grupperas, varefter de sparas i en databas och visas upp i en tabell där användaren kan tagga och verifiera ljuden. Viktigt att notera är att endast en beskrivning kan ges åt varje ljud. Ett ljud kan exempelvis inte ha både taggen ”skrämmande” och ”lugnande”, men ljud i olika grupper kan dela samma definition; två grupper av ljud kan alltså båda ha beskrivningen ”värmande”.

Kärnan i programmet som hjälper användaren med kategoriseringen är ett automatiskt klassificeringssystem som utformats med hjälp av maskininlärning. Systemet delar internt in liknande ljud i grupper baserat på deras klangfärg, eller timbre, genom att analysera ljuden med biblioteket timbral_models. Ljuden processeras och får sju stycken tillhörande numeriska värden utmärkta, som varierar från 0 till 100. De här värdena används i sin tur när programmet söker efter motsvarande ljud; ju fler värden som ligger nära varandra mellan två ljud, desto mer påminner ljuden om varandra.

Logiken som grupperar in ljuden enligt relevans är uppbyggd av två olika maskininlärningsalgoritmer som sekventiellt får fram en inledande och en justerbar lösning. Båda algoritmerna är av typen icke-väglett lärande (unsupervised learning) och är tagna ur scikit-learn-biblioteket. Den första metoden som uppskattar hur många grupper det finns i en samling av ljud är gjord med Mean shift-algoritmen. Den söker fram naturligt förekommande grupper av data baserade på de attribut den får inmatade, i det här fallet de uträknade värdena på klangfärg. Summan som det resulterar i och som produceras är en estimerad utgångspunkt för hur många grupper det kommer att finnas i slutet när användaren kategoriserat klart.

Den andra metoden är K-means-algoritmen som används för att kontinuerligt uppdatera grupperingarna i enlighet med hur användaren använder programmet. Till en början fungerar uppdelningarna som gjorts av Mean shift-algoritmen som startpunkt, men allteftersom användaren gör flera kategoriseringar kan det behövas nya indelningar och omorganisering i de existerande grupperna. Antalet grupper med liknande ljud kan aldrig minska, endast öka i antal – förutom när nya ljud läggs till. Den övre gränsen för antalet grupper är lika många som det finns ljud i programmet. Om en användare vill ge en specifik benämning till vart och ett av de inkluderade ljuden är det således möjligt.

När ljud blivit tillagda, analyserade och grupperade av maskininlärningsalgoritmerna kan de bli tilldelade taggar – en singulär beskrivning per ljud – definierade av antingen användaren själv eller automatiskt av programmet. Kategoriseringar som görs av användaren antas alltid vara korrekta. Taggar som programmet automatiskt tilldelar ljud förblir obekräftade till en början, då de kan vara inkorrekta. Det här beror på att programmet inte kan vara säkert på att klassificeringarna är rätt och är sålunda pessimistiskt i sitt handlingssätt. För att hålla reda på de här två möjligheterna har varje ljud en associerad boolesk datatyp, med benämningen verified, som klarlägger om den givna taggen ett ljud har är korrekt eller inte. Programmatiskt taggade ljud kan verifieras av användaren om hen anser att de är korrekta. Verifieringen kan också återställas och ändras för vart och ett av de redan kategoriserade ljuden när som helst under användning, vilket är behändigt om man av misstag verifierat ett ljud.

Verified-värdet bestämmer också hur programmet ska gå till väga när motsvarande ljud ska taggas. Om ett ljud klassificeras när alla ljud i dess grupp är overifierade – till exempel i början när inga ljud ännu har blivit kategoriserade – eller om ett verifierat ljud taggas om, hålls de fastlagda grupperingarna intakta. Då kommer alla ljud som tillhör samma grupp bli tilldelade den benämning som gavs till det interagerade ljudet. Det här scenariot är i princip detsamma som att döpa om taggen som hör ihop med gruppen.

Det andra händelseförloppet inträffar när användaren väljer att kategorisera om ljud som programmet redan taggat. Då bildas och modifieras grupper ända tills ett resultat nåtts där grupperingarna överensstämmer med de kategoriseringar som användaren begär. Den tilldelade gruppen kan ändras för alla ljud, med ljud som redan är verifierade kommer behålla sina taggar. Overifierade ljud kan å andra sidan bli märkta med en ny potentiell benämning, även om de tillhör en annan grupp än det korrigerade ljudet.

När ytterliga ljud läggs till i programmet behövs logik som ser till att de framsteg användaren redan gjort i klassificeringar bibehålls. Eftersom de nya ljuden kan befinna sig var som helst i relation till de som redan är inkluderade, kommer all gruppering att göras om och programmet applicerar alla de definierade taggarna på nytt. I och med det här kan antalet grupper ändra, men allt som användaren åstadkommit förblir oförändrat. De nyinlagda ljuden har möjligen fått en automatiskt genererad tagg om de påminner om något av de existerande ljuden.

Evaluering

För att evaluera det utvecklade programmet används fyra konstruerade exempelfall som påvisar fungerande, men också bristfälliga utfall. Valda grupper av ljud tagna från Freesound, som någon av deras användare skapat, utgör testdata i de olika scenarierna. Målet i de olika fallen är att tagga alla ljud med samma etikett i enlighet med de bestämda grupperna de tillhör. Kategoriseringarna är objektiva benämningar på ljud enligt deras klangfärg, till exempel att trumpetljud låter ”trumpetiga”. Även om det förekommer meningsskiljaktigheter om indelning baserad på klangfärg kan den här typen av benämningar också ses som gemensamt överenskomna subjektiva taggar, och fungerar också därför för att bedöma programmet.

Enheten som mäts i testerna är antalet omorganiseringar, eller korrigeringar, som behövs för att uppnå det bestämda målet. Utgående från resultatet i de olika exemplen blir det tydligt att det behövs olika antal korrigering beroende på hur nära besläktade grupperna av ljud är. Gitarr- och basljud blir lättare ihopblandade och behöver rättas till än ljud av exempelvis trummor och piano. Det blir också uppenbart att det finns flera olika sätt att nå fram till slutresultatet och att olika alternativ kan kräva olika antal korrigeringar.

Sammanfattning

Det behandlade lösningsförslaget i form av ett program visar möjligheten att få fram ett mottagligt kategoriseringssystem med subjektiva taggar genom att kombinera existerande metoder till en fungerande produkt. Programmet minskar tidskrävande manuellt arbete, är flexibelt med avseende på olika användares uppfattningsförmåga och förbättras kontinuerligt ju mera det utnyttjas. Lösningen är likväl ändå inte en färdig produkt; bland annat utseendet, prestandan och användarupplevelsen bör förbättras. Programmet i sig är inte heller det väsentliga i den här avhandlingen, utan hur idén och upplägget på lösningsförslaget kan utvecklas vidare. Genom att integrera systemet med existerande sökverktyg kunde sökträffarna förbättras och sökningarna tillämpas för att exempelvis i realtid hitta liknande ljud enligt en specifik användares tolkning.

Bibliography

[1] AudioCommons. *Audio Commons.* Accessed on April 23, 2020. URL: https://www.audiocommons.org/.

[2] AudioCommons. *Audio Commons Audio Extractor.* Accessed on February 13, 2020. URL: https://github.com/AudioCommons/ac-audio-extractor.

[3] AudioCommons. *Audio Commons Audio Extractor Web Demonstrator.* Accessed on February 13, 2020. URL: http://www.audiocommons.org/ac-audio-extractor/web_demonstrator/.

[4] Dmitry Bogdanov et al. *ESSENTIA: An Audio Analysis Library for Music Information Retrieval.* 2013.

[5] Pedro Cano et al. *Knowledge and Content-Based Audio Retrieval Using Wordnet.* 2004.

[6] Pedro Cano et al. *Nearest-Neighbor Automatic Sound Annotation with a WordNet Taxonomy.* 2005.

[7] Eugene Cherny et al. *An Approach for Structuring Sound Sample Libraries Using Ontology.* 2016.

[8] François Chollet. *Deep Learning with Python.* Manning, 2018. URL: https://livebook.manning.com/book/deep-learning-with-python/.

[9] ErrorCell. *Freesound - pack: Basic Tech-Trance by ErrorCell.* Accessed on March 20, 2020. URL: https://freesound.org/people/ErrorCell/packs/1292/.

[10] Frederic Font. *Audio Commons - Freesound.* Accessed on February 13, 2020. URL: https://www.audiocommons.org/2017/08/01/freesound.html/.

[11] Frederic Font et al. *Extending Tagging Ontologies with Domain Specific Knowledge.* 2014.

[12] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow.* O'Reilly, 2017.

[13] Yann Geslin, Pascal Mullon, and Max Jacob. *Ecrins an audiocontent description environment for sound samples*. 2002.

[14] Thomas Grill. *Constructing high-level perceptual audio descriptors for textural sounds*. 2012.

[15] Thomas Grill. *On Automated Annotation of Acousmatic Music*. 2012.

[16] Thomas Grill, Arthur Flexer, and Stuart Cunningham. *Identification of perceptual qualities in textural sounds using the repertory grid method*. 2011.

[17] Hideo Hattori. *A tool that automatically formats Python code to conform to the PEP 8 style guide*. Accessed on February 7, 2020. URL: https://pypi.org/project/autopep8/.

[18] hello_flowers. *Freesound - pack: Static and Radio Sounds by hello_flowers*. Accessed on March 20, 2020. URL: https://freesound.org/people/hello_flowers/packs/1867/.

[19] David M. Howard and Andy M. Tyrrell. *Psychoacoustically informed spectrography and timbre*. 1997.

[20] Hypnosomnia. *Freesound - pack: Bass Bars 1 by Hypnosomnia*. Accessed on March 20, 2020. URL: https://freesound.org/people/Hypnosomnia/packs/1254/.

[21] Institute of Sound Recording (IoSR). *Python scripts for modelling timbral attributes*. Accessed on February 6, 2020. URL: https://github.com/AudioCommons/timbral_models/.

[22] Kristoffer Jensen. “The Timbre Model”. Unpublished.

[23] jesuswaffle. *Freesound - pack: Tom-Tom Grooves by jesuswaffle*. Accessed on March 20, 2020. URL: https://freesound.org/people/jesuswaffle/packs/2918/.

[24] JetBrains. *Python Developers Survey 2018 Results*. Accessed on February 5, 2020. 2018. URL: https://www.jetbrains.com/research/python-developers-survey-2018/.

[25] Olivier Lartillot and Petri Toiviainen. *A Matlab Toolbox For Musical Feature Extraction From Audio*. 2007.

[26] Brian McFee et al. *LibROSA — librosa 0.7.2 documentation*. Accessed on April 23, 2020. URL: https://librosa.github.io/.

[27] Brian McFee et al. *librosa: Audio and Music Signal Analysis in Python*. 2015.

[28] Eduardo Reck Miranda. *Machine Learning and Sound Design.* 1997.

[29] David Moffat, David Ronan, and Joshua D. Reiss. *Unsupervised Taxonomy of Sound Effects.* 2017.

[30] MTG (UPF). *Browseable Freesound APIv2.* Accessed on February 13, 2020. URL: https://freesound.org/apiv2/.

[31] MTG (UPF). *Freesound - Freesound.* Accessed on February 13, 2020. URL: https://freesound.org/.

[32] MTG (UPF). *Freesound - Packs.* Accessed on March 5, 2020. URL: https://freesound.org/browse/packs/.

[33] MTG (UPF). *Freesound Annotator - Index.* Accessed on April 27, 2020. URL: https://annotator.freesound.org/.

[34] MTG (UPF). *Homepage — Essentia 2.1-beta6-dev documentation.* Accessed on April 23, 2020. URL: https://essentia.upf.edu/.

[35] Andy Pearce, Tim Brookes, and Russell Mason. *D5.1: Hierarchical ontology of timbral semantic descriptors.* 2016.

[36] Andy Pearce, Tim Brookes, and Russell Mason. *D5.2: First prototype of timbral characterisation tools for semantically annotating non-musical content.* 2017.

[37] Andy Pearce, Tim Brookes, and Russell Mason. *D5.3: Evaluation report on the first prototypes of the timbral characterisation tools.* 2017.

[38] Andy Pearce et al. *D5.6: Second prototype of timbral characterisation tools for semantically annotating non-musical content.* 2018.

[39] Andy Pearce et al. *D5.7: Evaluation report on the second prototypes of the timbral characterisation tools.* 2018.

[40] Andy Pearce et al. *D5.8: Release of timbral characterisation tools for semantically annotating non-musical content.* 2019.

[41] Python Code Quality Authority. *python code static checker.* Accessed on February 7, 2020. URL: https://pypi.org/project/pylint/.

[42] Python Software Foundation. *tkinter — Python interface to Tcl/Tk.* Accessed on February 7, 2020. URL: https://docs.python.org/3.8/library/tkinter.html.

[43] Python Software Foundation. *Welcome to Python.org.* Accessed on February 6, 2020. URL: https://www.python.org/.

[44] Kenneth Reitz. *NumPy — NumPy.* Accessed on February 6, 2020. URL: https://numpy.org/.

[45] Kenneth Reitz. *Pipenv: Python Dev Workflow for Humans.* Accessed on February 4, 2020. URL: https://pipenv.kennethreitz.org/.

[46] REVEREND.BLACK. *Freesound - pack: Metal Guitar loops Un trimmed 150BPM by REVEREND.BLACK.* Accessed on March 11, 2020. URL: https://freesound.org/people/REVEREND.BLACK/packs/19793/.

[47] Julien Ricard and Perfecto Herrera. *Morphological sound description computational model and usability evaluation.* 2004.

[48] Ali Gholami Rudi. *a python refactoring library...* Accessed on February 7, 2020. URL: https://pypi.org/project/rope/.

[49] Mihir Sarkar, Cyril Lan, and Joe Diaz. *Words that Describe Timbre A Study of Auditory Perception Through Language.* 2009.

[50] scikit-learn developers. *2.3.2. K-means.* Accessed on February 14, 2020. URL: https://scikit-learn.org/stable/modules/clustering.html#k-means.

[51] scikit-learn developers. *2.3.4. Mean Shift.* Accessed on February 14, 2020. URL: https://scikit-learn.org/stable/modules/clustering.html#mean-shift.

[52] scikit-learn developers. *scikit-learn: machine learning in Python.* Accessed on February 4, 2020. URL: https://scikit-learn.org/.

[53] scikit-learn developers. *sklearn.cluster.KMeans.* Accessed on March 19, 2020. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn-cluster-kmeans.

[54] scikit-learn developers. *sklearn.cluster.MeanShift.* Accessed on March 19, 2020. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MeanShift.html#sklearn-cluster-meanshift.

[55] scuzzpuck. *Freesound - pack: Box Drum by scuzzpuck.* Accessed on March 11, 2020. URL: https://freesound.org/people/scuzzpuck/packs/1904/.

[56] Markus Siemens. *Welcome to TinyDB!* Accessed on February 6, 2020. URL: https://tinydb.readthedocs.io/.

[57] Dagobert Soergel. *WordNet. An Electronic Lexical Database.* 1998.

[58] Sound and Video Understanding teams at Google. *AudioSet.* Accessed on April 27, 2020. URL: https://research.google.com/audioset/.

[59] tim.kahn. *Freesound - pack: Sharpening Knives by tim.kahn.* Accessed on March 20, 2020. URL: https://freesound.org/people/tim.kahn/packs/2280/.

[60] Pantelis N. Vassilakis. *SRA: A Web-based Research Tool for Spectral and Roughness Analysis of Sound Signals*. 2007.

[61] Sanae Wake and Toshiyuki Asahi. *Sound Retrieval with Intuitive Verbal Descriptions*. 2001.

[62] David H. Wolpert. *The Lack of A Priori Distinctions Between Learning Algorithms*. 1996.

[63] Nina Zakharenko. *About Python*. Accessed on February 6, 2020. 2019. URL: https://www.learnpython.dev/02-introduction-to-python/010-best-practices/02-brief-history/.

[64] zerolagtime. *Freesound - pack: A Harpsicord Dream by zerolagtime*. Accessed on March 20, 2020. URL: https://freesound.org/people/zerolagtime/packs/1793/.

Appendix

Repository

https://github.com/AndreasNasman/masters-thesis/

Python source code

timbral_models.py

```python
from json import loads
from pathlib import Path
from uuid import uuid1

from timbral_models import timbral_extractor

timbral_characterstics = [
    "boominess",
    "brightness",
    "depth",
    "hardness",
    "roughness",
    "sharpness",
    "warmth",
]


def process_freesound_samples(directory):
  """Processes Freesound samples to a common format."""
  result = []
  relevant_properties = ["ac_analysis", "id", "name", "pack_name", "type"]

  paths = Path(directory).rglob("*.json")
  for path in paths:
    try:
      with open(path, "r") as reader:
        metadata = loads(reader.read())
        converted_metadata = {
            "path": str(path),
            "tag": "",
            "verified": False
        }

        for rp in relevant_properties:
          value = metadata[rp]

          if rp == "ac_analysis":
            for tc in timbral_characterstics:
              converted_metadata[tc] = value["ac_" + tc]
          else:
            converted_metadata[rp] = value

        result.append(converted_metadata)
    except:
      continue
  return result


def process_sounds(directory):
  """Processes sounds using AudioCommons Timbral Models."""
  result = []

  paths = Path(directory).rglob("*")
  for path in paths:
    try:
      path_str = str(path)
      sound = timbral_extractor(path_str, clip_output=True)
      del sound["reverb"]

      sound["id"] = uuid1().hex
      sound["name"] = Path(path_str).resolve().stem
      sound["pack_name"] = ""
      sound["path"] = path_str
      sound["tag"] = ""
      sound["type"] = Path(path_str).suffix.replace(".", "")
      sound["verified"] = False

      result.append(sound)
    except:
      continue

  return result
```

machine_learning.py

```
from copy import deepcopy

import numpy
from sklearn.cluster import KMeans, MeanShift

from logic.timbral_models import timbral_characterstics


def build_X(sounds):
  """Builds X, i.e. coordinates for all samples' features."""
  X = []

  for sound in sounds:
    x = []
    for tc in timbral_characterstics:
      x.append(sound[tc])
    X.append(x)

  return X


def find_groups(sounds, X, centroids):
  """Finds groups of sound using K means."""
  k = len(centroids)
  km = KMeans(init=centroids, n_clusters=k, n_init=1).fit(X)

  updated_sounds = deepcopy(sounds)
  for sound, group in zip(updated_sounds, km.labels_):
    sound["group"] = int(group)

  updated_centroids = km.cluster_centers_.tolist()
  updated_centroids.sort()

  return updated_sounds, updated_centroids


def init_groups(sounds, X):
  """Searches for initial groups of sound using Mean shift."""
  ms = MeanShift().fit(X)
  centroids = ms.cluster_centers_.tolist()
  centroids.sort()

  return find_groups(sounds, X, numpy.array(centroids))


def add_group(sounds, X, centroids, modified_sound):
  """Adds a new group of sound."""
  new_centroid = []
  for tc in timbral_characterstics:
    new_centroid.append(modified_sound[tc])

  new_centroids = deepcopy(centroids)
  new_centroids.append(new_centroid)
  new_centroids.sort()

  return find_groups(sounds, X, numpy.array(new_centroids))


def modify_groups(sounds, X, centroids, modified_sound, tag, db_sounds):
  """Modifies groups until sounds have their associated tag."""
  updated_sounds, updated_centroids = add_group(
      sounds, X, centroids, modified_sound)

  # Finds the new group.
  for updated_sound in updated_sounds:
    if updated_sound["id"] == modified_sound["id"]:
      new_group = updated_sound["group"]
      break

  # Tags sounds in the new group.
  for updated_sound in updated_sounds:
    if updated_sound["group"] == new_group:
      updated_sound["tag"] = tag

  # Checks that all verified sounds kept their tag.
  for db_sound, updated_sound in zip(db_sounds, updated_sounds):
    if (db_sound["verified"] and
        db_sound["tag"] != updated_sound["tag"] and
            db_sound["id"] != modified_sound["id"]):
      return modify_groups(updated_sounds, X, updated_centroids,
                           db_sound, db_sound["tag"], db_sounds)

  return updated_sounds, updated_centroids


def sort_sounds(sounds):
  """Sounds need to be sorted for some of the logic to work correctly."""
  updated_sounds = sorted(sounds, key=lambda s: (s["group"], not s["tag"]))
  updated_X = build_X(updated_sounds)

  return updated_sounds, updated_X


def review_groups(sounds, X, centroids):
  """Makes sure that there is only one tag per group."""
  updated_sounds, updated_X = sort_sounds(sounds)
  updated_centroids = centroids

  for a, b in zip(updated_sounds, updated_sounds[1:]):
    if (a["group"] == b["group"] and a["tag"] != b["tag"]
            and a["tag"] and b["tag"]):
      updated_sounds, updated_centroids = add_group(
          updated_sounds, updated_X, updated_centroids, b)
      return review_groups(
          updated_sounds, updated_X, updated_centroids)

  return updated_sounds, X, updated_centroids


def tag_groups(sounds):
  """Tags sounds that are missing a group's tag."""
  updated_sounds, updated_X = sort_sounds(sounds)

  for a, b in zip(updated_sounds, updated_sounds[1:]):
    if a["group"] == b["group"] and a["tag"] and not b["tag"]:
      b["tag"] = a["tag"]

  return updated_sounds, updated_X
```

main.py

```python
from operator import itemgetter
from os import path
from pathlib import Path
from subprocess import run
from tkinter import (HORIZONTAL, VERTICAL, Canvas, E, N, S, Tk, Toplevel, W,
                     filedialog, messagebox, ttk)

from tinydb import TinyDB, where

from logic.machine_learning import (build_X, init_groups, modify_groups,
                                    review_groups, tag_groups)
from logic.timbral_models import (process_freesound_samples, process_sounds,
                                  timbral_characterstics)

# Database setup
db = TinyDB("db.json")
ml = TinyDB("ml.json")


def update_databases(sounds, centroids, X=None):
  """Updates databases with modified data."""
  for sound in sounds:
    db.upsert(sound, where("id") == sound["id"])

  ml.upsert({"data": centroids, "id": "centroids"},
            where("id") == "centroids")
  if X:
    ml.upsert({"data": X, "id": "X"}, where("id") == "X")


def purge_databases():
  """Clears all data from the databases."""
  if not len(db.all()):
    messagebox.showerror(message="There␣are␣no␣sounds␣to␣delete!")
    return

  ok = messagebox.askokcancel(
      message="This␣will␣remove␣all␣currently␣added␣sounds.")

  if ok:
    db.purge()
    ml.purge()
    draw_table()


def check_database_occurrence(sounds):
  """Checks if new sounds are already in the database."""
  new_sounds = []

  for sound in sounds:
    db_sound = db.search(where("id") == sound["id"])
    if not db_sound:
      new_sounds.append(sound)

  return new_sounds


def choose_directory(mode=None):
  """Loads files from a directory and handles them appropriately."""
  directory = filedialog.askdirectory(initialdir=Path.cwd())
  if not directory:
    return

  if mode == "freesound":
    sounds = process_freesound_samples(directory)
    if not sounds:
      messagebox.showerror(
          message=("No␣valid␣metadata␣files␣(.json)␣" +
                   "found␣in␣chosen␣directory!"))
      return
  else:
    sounds = process_sounds(directory)
    if not sounds:
      messagebox.showerror(
          message=("No␣supported␣sound␣files␣(.wav)␣" +
                   "found␣in␣chosen␣directory!"))
      return

  # Checks if chosen sounds already exist in the database.
  sounds = check_database_occurrence(sounds)
  if not sounds:
    messagebox.showerror(
        message="No␣new␣sounds␣to␣analyze␣found␣in␣chosen␣directory!")
    return

  # Finds initial groups for all sounds.
  sounds += db.all()
  X = build_X(sounds)
  sounds, centroids = init_groups(sounds, X)

  # Checks that all groups has a single unique tag and
  # makes sure new sounds are tagged according to the group they belong to.
  sounds, X, centroids = review_groups(sounds, X, centroids)
  sounds, X = tag_groups(sounds)

  update_databases(sounds, centroids, X)
  draw_table()


def tag(dialog, input, sound):
  """Tags a sound, plus potentially related sounds."""
  errorMessage = None

  if not input or input.isspace():
    errorMessage = "Input␣cannot␣be␣empty!"
  elif input == sound["tag"]:
    errorMessage = "The␣input␣cannot␣be␣the␣same␣as␣the␣current␣value!"

  if errorMessage:
    messagebox.showerror(message=errorMessage)
    return

  # Keep current groupings.
  group_sounds = db.search(where("group") == sound["group"])
  if (sound["verified"] or all([not gs["verified"] for gs in group_sounds])):
    db.update({"tag": input}, where("group") == sound["group"])
    db.update({"verified": True}, where("id") == sound["id"])
  else:  # Modify groups.
    db.update(
        {"tag": input, "verified": True},
        where("id") == sound["id"]
    )
    modified_sound = db.search(where("id") == sound["id"])[0]

    db_sounds = db.all()
    X = ml.search(where("id") == "X")[0]["data"]
    centroids = ml.search(where("id") == "centroids")[0]["data"]

    sounds, centroids = modify_groups(
        db_sounds,
        X,
        centroids,
        modified_sound,
        input,
        db_sounds
    )
    sounds, X, centroids = review_groups(sounds, X, centroids)
    sounds, X = tag_groups(sounds)
    update_databases(sounds, centroids, X)

  draw_table()
  dialog.destroy()


def tag_dialog(sound):
  """Dialog to enter tag input."""
  dialog = Toplevel()
  dialog.title("Tag")

  frame = ttk.Frame(dialog)
  frame.grid(column=0, row=0, sticky=(N, W, E, S))

  input = ttk.Entry(frame)
  input.grid(column=0, row=0)
  input.focus()
  button = ttk.Button(frame, text="Tag", command=lambda dialog=dialog,
                      input=input: tag(dialog, input.get().strip(), sound))
  button.grid(column=0, row=1)


def verify(sound):
  """Toggles verification of a sound."""
  db.update({"verified": not sound["verified"]}, where("id") == sound["id"])
  draw_table()


def listen(event, sound):
  """Opens a sound in the system default application."""
  sound_path = f"{path.splitext(sound['path'])[0]}.{sound['type']}"
  run(["open", sound_path])


def draw_table():
  """(Re)draws the table of sounds."""
  lower_frame = ttk.Frame(main_frame)
  lower_frame.grid(column=0, row=2, sticky=(N, E, S, W))
  lower_frame.columnconfigure(0, weight=1)
  lower_frame.rowconfigure(1, weight=1)

  # Info
  info_frame = ttk.Frame(lower_frame, padding=(0, 0, 0, 10))
  info_frame.grid(column=0, row=0, sticky=(N, E, S, W))
  info_frame.columnconfigure(0, weight=1)
  info_frame.columnconfigure(1, weight=1)
  info_frame.columnconfigure(2, weight=1)

  sounds = db.all()
  label = ttk.Label(info_frame, text=f"Number␣of␣sounds:␣{len(sounds)}")
  label.grid(column=0, row=0)

  try:
    centroids = ml.search(where("id") == "centroids")[0]["data"]
  except:
    centroids = []
  label = ttk.Label(
      info_frame, text=f"Number␣of␣groups:␣{len(centroids)}")
  label.grid(column=1, row=0)

  button = ttk.Button(info_frame, command=purge_databases,
                      text="Delete␣sounds")
  button.grid(column=2, row=0)

  # Scrollbars
  canvas = Canvas(lower_frame, bg="#e6e6e6", highlightthickness=0)
  canvas.grid(column=0, row=1, sticky=(N, E, S, W))

  scrollbar_y = ttk.Scrollbar(
      lower_frame, orient=VERTICAL, command=canvas.yview, bg="#e6e6e6")
  scrollbar_y.grid(column=1, row=1, sticky=(N, E, S, W))

  scrollbar_x = ttk.Scrollbar(
      lower_frame, orient=HORIZONTAL, command=canvas.xview, bg="#e6e6e6")
  scrollbar_x.grid(column=0, row=2, sticky=(N, E, S, W))

  scrollable_frame = ttk.Frame(canvas, padding=(10, 0))
  scrollable_frame.bind("<Configure>", lambda e: canvas.configure(
      scrollregion=canvas.bbox("all")))

  canvas.create_window((0, 0), window=scrollable_frame, anchor="nw")
  canvas.configure(yscrollcommand=scrollbar_y.set)
  canvas.configure(xscrollcommand=scrollbar_x.set)

  # Table
  fields = ["name", "type", "group", "tag",
            "verified", "pack_name"] + timbral_characterstics
  sounds = db.all()
  sounds.sort(key=itemgetter("pack_name", "name", "id"))

  for row, sound in enumerate(sounds, start=1):
    # Headers
    if row == 1:
      for column, field in enumerate(fields, start=2):
        label = ttk.Label(scrollable_frame,
                          font="-weight␣bold", text=field)
        label.grid(column=column, row=0)
        scrollable_frame.columnconfigure(column, weight=1)

    button = ttk.Button(scrollable_frame, command=lambda sound=sound:
                        tag_dialog(sound), text="Tag")
    button.grid(column=0, row=row)

    if sound["tag"]:
      if sound["verified"]:
        text = "Unverify"
      else:
        text = "Verify"
      button = ttk.Button(scrollable_frame, command=lambda sound=sound:
                          verify(sound), text=text)
      button.grid(column=1, row=row)

    for column, field in enumerate(fields, start=2):
      label = ttk.Label(scrollable_frame)
      value = sound[field]

      if value == "":
        value = "-"
      elif isinstance(value, bool):
        if sound["tag"]:
          if value:
            value = "Yes"
            label.configure(foreground="green")
          else:
            value = "No"
            label.configure(foreground="red")
        else:
          value = "-"
      elif isinstance(value, float):
        value = round(value, 2)

      label.configure(text=value)
      label.grid(column=column, row=row)

      if field == "name":
        label.bind("<Button-1>", lambda event,
                   sound=sound: listen(event, sound))
        label.configure(cursor="hand2", foreground="blue")

  # Removes old render.
  if len(main_frame.grid_slaves()) > 3:
    main_frame.grid_slaves()[1].destroy()

  root.update()
  if(scrollable_frame.winfo_width() < canvas.winfo_width()):
    canvas.itemconfig(1, width=canvas.winfo_width())


# Root
root = Tk()
root.title("Subjective␣Audio␣Tagging")
root.grid_columnconfigure(0, weight=1)
root.grid_rowconfigure(0, weight=1)
root.geometry(f"{root.winfo_screenwidth()}x{root.winfo_screenheight()}")

# Main frame
main_frame = ttk.Frame(root)
main_frame.grid(column=0, row=0, sticky=(N, E, S, W))
main_frame.grid_columnconfigure(0, weight=1)
main_frame.grid_rowconfigure(2, weight=1)

# Lower frame
upper_frame = ttk.Frame(main_frame, padding=10)
upper_frame.grid(column=0, row=0, sticky=(N, E, S, W))
upper_frame.columnconfigure(0, weight=1)
upper_frame.columnconfigure(1, weight=1)

label = ttk.Label(upper_frame, text="Add␣sounds")
label.grid(column=0, row=0)

button = ttk.Button(upper_frame, command=choose_directory,
                    text="Choose␣directory")
button.grid(column=0, row=1)
```

```
label = ttk.Label(upper_frame, text="Add␣Freesound␣samples")
label.grid(column=1, row=0)

button = ttk.Button(upper_frame, command=lambda mode="freesound":
                    choose_directory(mode), text="Choose␣directory")
button.grid(column=1, row=1)

# Middle frame
middle_frame = ttk.Frame(main_frame, padding=10)
middle_frame.grid(column=0, row=1, sticky=(N, E, S, W))
middle_frame.columnconfigure(0, weigh=1)

separator = ttk.Separator(middle_frame)
separator.grid(column=0, row=0, sticky=(N, E, S, W))

# Lower frame
draw_table()

# Enter event loop
root.mainloop()
```