Kiel University of Applied Sciences

MIP1, MIP2, MIP3

Technical Report

Implementation of a Distributed AI Driven Audio Sample Manager for Drum Sound Tagging, Clustering and Recommendations

Niklas Wantrupp (928817)

Course of Studies: Information Engineering

Supervisor:
Prof. Dr. Robert Manzke

July 15, 2021

Contents

Abbreviations

AI	Artificial Intelligence
Ajax	Asynchronous JavaScript and XML
API	Application Programming Interface
CMVN	Cepstral mean and variance normalization
CNN	Convolutional Neural Network
DCT	Discrete Cosine Transform
DNN	Deep Neural Network
JWT	Json Web Token
k-NN	k-Nearest Neighbor
MFCC	Mel Frequency Cepstral Coefficients
MSD	Million Song Dataset
MTT	MagnaTagATune
REST	Representational State Transfer
RS	Recommendation System
STFT	Short Time Fourier Transformation
UI	User Interface

1 Introduction

Audio industry professionals tend to collect enormous amounts of audio data within their production environment. Foley artists use different kinds of recorded sounds when working on film audio, music producers have libraries of different musical instruments to choose from and game sound designers also have a vast amount of audio files to choose from when designing soundscapes for games.

The process of assembling and manually tagging different audio files takes a huge amount of effort and is extremely time consuming. Audio professionals have to manually listen to all sounds within their library and then introduce meaningful tags to make the vast amount of files accessible for later usage in the production process. Furthermore, the process of finding the right sound in production also can cost a lot of time for the engineer.

To overcome these shortcomings, this project investigates the utilization of artificial intelligence for automatic audio tagging, clustering and recommendation and then the created models are used to implement a distributed audio sample manager. The required data for model training is extracted from the open source online sound library Freesound[1]. In order to curate a representative data set and to refine the scope of the project, solely drum sounds will be used to train the model. Audio sample tagging models are already well studied and thus a Transfer Learning Approach is chosen where a pre-trained MUSICNN model presented by Pons and Serra [15] is chosen and further modified to adapt to the specific use case of drum sound tagging.

The implemented recommendation system and custom audio tagger are both based on k-Nearest Neighbor classifiers (k-NN), working with normalized Mel Frequency Cepstral Coefficients (MFCC) to allow for high accuracy audio recommendation and user customized tagging.

The system is implemented and deployed, using web technologies to allow for user access via web browsers. The backend is built by using the Python Framework Flask to implement a REST-API which exposes all required model computations to the frontend which will use Angular to implement the UI.

[1] Freesound.org

2 Technical Background

To introduce the topic of musical motivated Machine Learning, this chapter introduces the main concepts which were applied during the course of the project. At first, the concept of classification problems is tackled before getting into feature extraction for audio tasks. After that, the MUSICNN models and the concept of Transfer Learning are introduced to get an idea on which concepts the projects Drumsample Base Tagger (4.2) is built on, before coming to Recommendation Systems and k-Nearest Neighbor classifiers to learn on what methodologies the Drumsample Recommendation System and Drumsample Custom Tagger (4.3) are based on.

This chapter does not claim to provide a complete introduction to the above-mentioned topics. For a more thorough introduction to Machine Learning Frochte [6] and Goodfellow, Bengio, and Courville [8] are recommended. For musical motivated Machine Learning Pons et al. [14], Pons and Serra [15], and Purwins et al. [16] are recommended.

2.1 Classification Tasks

Classification tasks try to learn a function $f : X \rightarrow Y$, where X is the set of features and Y is a discrete set of classes. The function f is learnt by feeding a model with a set of training data $D = (x_1, f(x_1)), ..., (x_n, f(x_n))$ with exemplary datapoints and their target values. Classification tasks are therefore categorized as supervised learning problems [6].

A classification problem can further be subdivided into binary and multi-class classifications, where binary models classify data into two groups, whereas multi-class models classify data into n groups.

Musical classification or tagging tasks work in the audio domain and try to provide models to automatically assign a given set of classes to a provided set of sounds. To implement audio tagging models, the provided audio data needs to be transformed into a pre-defined feature representation which fits the models design. The following section presents the most common feature representations for audio data.

2.2 Feature Representation of audio signals in Machine Learning

Feature extraction methods in the audio domain are mostly based on representations of the signal in abstract domains like filtered spectral components or cepstral components. Raw audio waveforms are rarely used and thus the following sections give an introduction into the most common methodologies, namely melspectrograms, MFCCs and normalized MFCCs [16].

Melspectrogram The first step is to take Short Time Fourier Transformations (STFT) of a given audio signal with a given window size and an overlap. The single bins are then taken and further transformed, using the Mel-scale with the formula:

$$Z = 1127 \cdot ln(1 + \frac{f}{700}), \tag{1}$$

as formulated by Stevens, Volkmann, and Newman [20]. The Mel-scale is a methodology developed to approach the problem of the logarithmic nature of the human auditory system. Frequencies in the lower spectrum tend to be easier to differentiate than frequencies on the higher spectrum [20]. The filtering process using equation 1 also results in a reduction of features. The resulting signal representation is called a melspectrogram [12].

Figure 1 shows an exemplary melspectrogram of a piano, playing the chromatic C Scale.

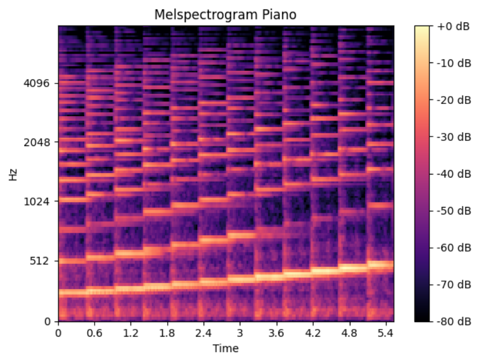

Figure 1: Melspectrogram of piano playing chromatic C scale

One can clearly observe the base frequency of the notes as most prominent in the figure, while the harmonics are decrease in level with increasing frequency.

Mel Frequency Cepstral Coefficients (MFCC) The most common feature extraction method in the audio domain are MFCCs introduced by Furui [7] which are “magnitude spectra projected to a reduced set of frequency bands, converted to logarithmic magnitudes, and approximately whitened and compressed with a discrete cosine transform (DCT)” [16].

The main purpose of using the MFCC is to further reduce the number of features required to represent an audio signal. Furthermore, the melspectrogram creation process creates correlations between adjacent frames because of using overlapping STFT windows. The application of a DCT decorrelates the feature vectors. This makes MFCC suitable for application for models which rely on the usage of covariance matrices [12]. Figure 2 shows the resulting MFCCs after performing a DCT on the melspectrogram displayed in Figure 1.

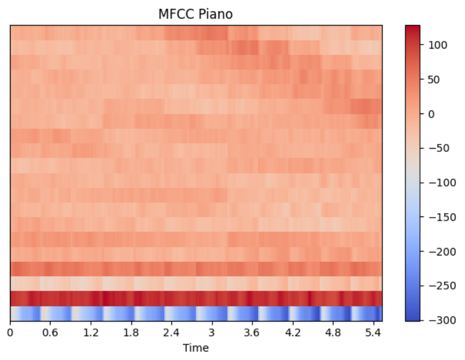

Figure 2: MFCC of a piano playing chromatic C scale

One can clearly see that the representation of the audio signal is different. The MFCC displays the spectral envelope of the signal and drops information on fine grained spectral structures [1].

Cepstral mean and variance normalization (CMVN) When a model is presented with noisy data, the performance of the model can quickly degrade, as Droppo and Acero [5] have shown. To overcome these factors, the MFCC features can be normalized by mean and variance.

For the mean normalization each normalized feature vector $\widehat{x_n}$ is retrieved by subtracting the mean vector μ_x from each feature vector x_n. For the variance normalization the mean normalized feature vector is taken and divided by the standard deviation of the vector [5]. After applying the CMVN, the mean of the cepstrum is 0 and the variance is 1. This leads to robustness against signal distortions (e.g. room accoustics, microphone transfer functions), source variability and the presence of additive noise Droppo and Acero [5]. Figure 3 shows the CMVN MFCC feature vector computed from the same audio file of figure 1 and 2.

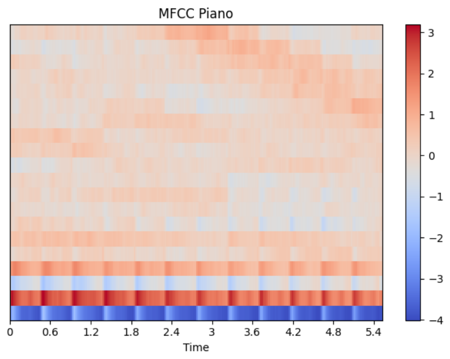

Figure 3: MFCC with CMVN applied of a piano playing chromatic C scale

2.3 MUSICNN

MUSICNN is a set of pre-trained musical motivated neural networks for audio tagging. One model is trained on the MagnaTagATune (MTT) dataset by Law et al. [11]. Another two models are trained on two different versions of the Million Song dataset (MSD) by Bertin-Mahieux et al. [3]. Those three models are trained on MTT and MSD, besides that the library also provides vgg-like models for validation purposes [15]. Figure 4 shows the network structure.

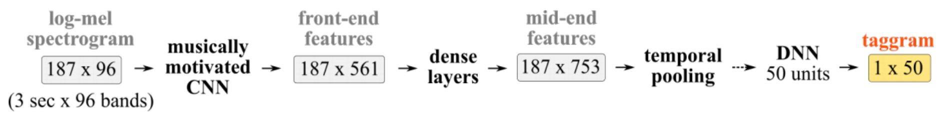

Figure 4: MUSICNN network structure, taken from Pons and Serra [15]

One can observe that the network takes a three second long melspectrogram with 96 filter banks as input and feeds those into a Convolutional Neural Network (CNN). The resulting front-end features are then fed into several dense layers before passing a temporal pooling layer which is fed into a Deep Neural Network (DNN). The DNN then outputs the resulting taggram.

The main goal of the library is to provide a pre-trained multilabel classifier which can predict the top n tags of a given song. Additionally, the library is also aimed at feature extraction, allowing for access of all intermediate representations from within the model and also for transfer learning (which will be further treated in section 2.4) to use the pre-trained model as a basis for adaption to specific use cases [15].

2.4 Transfer Learning

Transfer Learning is a methodology in Machine Learning which is often applied when sufficient data is unavailable. The idea behind the concept of Transfer Learning is not to rebuild models from scratch when the underlying feature-distribution changes, but instead utilize acquired past knowledge to solve the new task [18].

Therefore, a pre-trained model from a related domain is chosen, and a new model is created by modifying the pre-trained model, which leverages the existing knowledge i.e. features and trained weights to adapt to a new task [13]. The main idea of Transfer Learning is to remove the last layers of the given base network, freeze the weights and use the base model as a feature extractor for a domain specific model which is then trained on available data [18].

2.5 Recommendation Systems

Recommendation Systems (RS) are one of the most common use cases of Machine Learning in the modern web. They describe methodologies to provide suggestions which could be of interest to a specific user. The applied techniques for a given problem are specifically tailored to the actual domain [17].

Recommendations are often highly personalized, resulting in a wide variety of suggestions between different users of the same system. There are, however, also non-personalized recommendations although these are normally not tackled by recommendation systems research [17]. RSs propose ranked lists of item suggestions which are based on explicit user actions (e.g. ratings) or implicit user actions (e.g. time spent interacting with items of a specific type).

To propose useful information to a user, a RS must predict whether a specific item shall be recommended or not, based on available data. Burke [4] and Ricci, Rokach, and Shapira [17] distinguish between six different methodologies, which vary in addressed domain, used knowledge and utilised recommendation algorithms:

1. **Content-Based:**
    - Items are proposed, based on the similarity to favorite past items of a user.
    - Different similarity measures can be used, based on the items features.
2. **Collaborative-Filtering**
    - Recommends items to a user which have been liked by different users with similar behavior.

- Similarity is calculated between different users, based on their item interactions in the past.

3. **Demographic:**
   - Items are recommended based on the demographic profile of the user.
4. **Knowledge-Based:**
   - Recommends items, based on domain knowledge.
   - Users needs are evaluated and in a second step items are matched to personal user requirements.
5. **Community-Based:**
   - Items are recommended based on preferences of the user friends. As Sinha and Swearingen [19] have shown, users are more likely to rely on items recommended by friends than on unknown individual recommendations.
6. **Hybrid Approaches:**
   - Hybrid Approaches try to combine different already mentioned techniques to overcome shortcomings of single approaches.

2.6 k-Nearest Neighbor (k-NN)

The k-Nearest Neighbor classifier is a lazy learning algorithm. This means that most part of the computational work is done in the prediction step and not in the training stage like it is the case for eager learners [6]. This already makes clear that the k-NN algorithm has one drawback in contrast to eager learning algorithms - predictions usually take longer. The advantage of lazy learners is, however, that a model can be built locally around the feature vector at prediction time. Other advantages of the k-NN are, that the required training time is much shorter, which speeds up the development time of such a model. In addition a k-NN also has nearly no parameters. The only parameter required is k, which depicts how many datapoints from the ground truth set shall be considered in the prediction step [6].

A k-NN works by calculating distance measures around the feature vector to be classified. For this purpose the k nearest points are taken into account and the class which occurs the most determines the models output. Figure 5 shows an example of the prediction step of a k-NN classifier. The red point needs to be classified. When choosing $k = 3$, the point gets classified as *Class 2*. In contrast to this, the point gets classified as *Class 1* when choosing $k = 7$. This already shows a difficulty when determining the correct value of k. On the one hand the model is sensitive to noise and local fluctuations (overfitting) when k is too low. On the other hand the model is sensitive to underfitting issues when k is too large. Thus, choosing the right value for k highly affects the model performance [2].

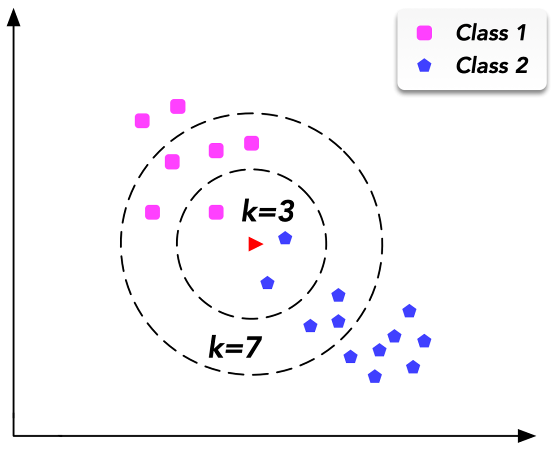

Figure 5: k-NN example

3 Development Environment

For the development environment a highly modularized approach was taken. The code editor chosen was Visual Studio Code, which offers an Extensions for developing inside Docker Containers. This results in a portable development environment, which has to be set up once and then can be executed in different environments.

To allow the usage of Nvidia graphics cards, the Nvidia Container Toolkit is utilized. This allows to share a graphic card with the development container, and thus allow to train tensorflow models on GPU, which speeds up training time. To set the development environment up, the following steps need to be taken:

1. Download and install Docker[2]
2. If Nvidia-GPU is available: Download and Install the Nvidia Container Toolkit[3]
3. Download and install Visual Studio Code[4]
4. Download and install the Extension: Visual Studio Code Remote - Containers[5]

[2] Docker
[3] Nvidia Container Toolkit
[4] VS Code
[5] Visual Studio Code Remote - Containers

4 Drumsample Models

The created Machine Learning models form the basis for the implemented distributed drum sample manager. The following sections will introduce a dataset which was curated for training models on drum sounds, a basic drum classifier and both a recommendation system and a custom drum classifier, based on a k-NN approach.

4.1 Drumsample Dataset

To be able to train a model on drum sounds at first, sufficient data needed to be collected. To achieve this, a bot was written, which automatically downloaded all sounds from freesound.org with the following tags: *Kick*, *Snare*, *Hat*, *Ride*, *Crash* and *Tom*. Because the sounds on freesound.org are normally tagged by the users, the downloaded data was audited and cleaned. Figure 6 shows a barplot of the distribution of class labels after cleaning the data.

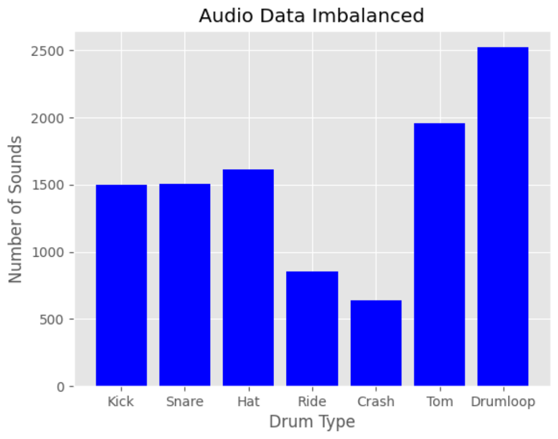

Figure 6: Imbalanced Dataset

One can observe that the data is highly imbalanced. Because imbalanced datasets lead to poorer overall model performance, the data had to be rearranged into a new class scheme [21]. *Ride*, *Crash* and *Hat* sounds typically tend to have a similar spectrum. Because *Ride* and *Crash* had the least amount of sounds, *Ride*, *Crash* and *Hat* were merged to the class *Cymbal*, with every subgroup, contributing with an equal amount of sounds. After that, the cardinality of all classes was analyzed again for every class the number of sounds was limited to 1500 sounds. In a last step, the dataset was split into train-, validation- and test-data with randomly chosen samples. The split chosen was 66.6% training data, 16.6% validation data and 16.6% test data. Figure 7 shows the final data distributions of all three sets. The Drumsample Dataset can be accessed via Google Drive[6].

[6]Drumsample Dataset

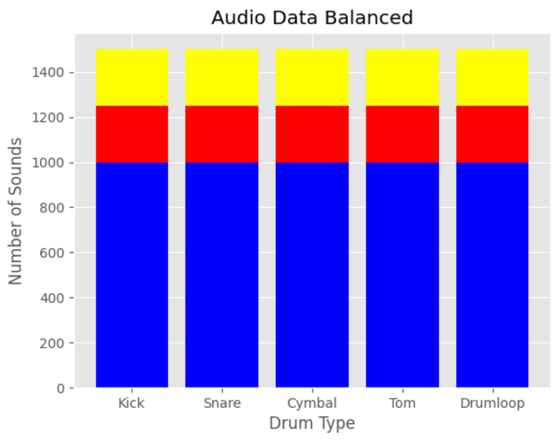

Figure 7: Balanced Dataset

4.2 Drumsample Tagger

The Drumsample Tagger is based on the MUSICNN model (described in section 2.3) which was trained on the MTT dataset. The next sections will describe how the model was modified to tackle the task of drum sound tagging rather than tagging musical genres. As a basis for the model implementation a **tensorflow.keras** implementation was used[7]. The code for the classification model can be found on Github[8].

4.2.1 Model Structure

The approach for the Drumsample Tagger was to take the basic MUSICNN model, trained on the MTT dataset, and then apply a Transfer Learning approach to use the MUSICNN model as a feature extractor. Therefore the penultimate and ultimate layer of the base model have been removed and all remaining model weights were frozen. Table 1 shows the architecture of the final Drumsample Tagger model, where *b* depicts the batch size.

Operation	Input Shape	Output Shape
Input	(b, 187, 96, 1)	(b, 187, 96, 1)
Modified MUSICNN model	(b, 187, 96, 1)	(b, 200)
Dropout	(b, 200)	(b, 200)
Dense	(b, 200)	(b, 200)
Dense	(b, 200)	(b, 5)

Table 1: Drumsample Tagger Architecture

The input layer feeds melspectrogram representation of audio data into the model which should have a minimum length of three seconds and 96 mel bands. When a sound is shorter than three

[7]musicnn_keras
[8]Drumsample Tagger Code

seconds, zero padding is applied to fit the minimum length. When a sound, however, is longer than 3 seconds, it is split into chunks of three seconds and the model is fed with a batch of successive audio frames.

After passing the input, the batch is fed into the pruned MUSICNN model for feature extraction, before being passed to the Dropout layer to minimize the chance of overfitting [9]. Finally the batch is passed to two fully connected Dense layers, where the last one represents the final output. After that, the mean over the Maximum Likelihood of classes is calculated and the maximum of this calculation determines the final model output.

4.2.2 Training and Model Selection

For the training process, the model was trained using an *Adam* optimizer with a learning rate $l_r = 0.001$ and a Categorical Crossentropy loss function. The model was trained for 50 epochs with a batch size of $b = 32$, where for every epoch a checkpoint of all models weights was saved. Figures 8 and 9 show the model accuracy and loss on both, the training and the validation set.

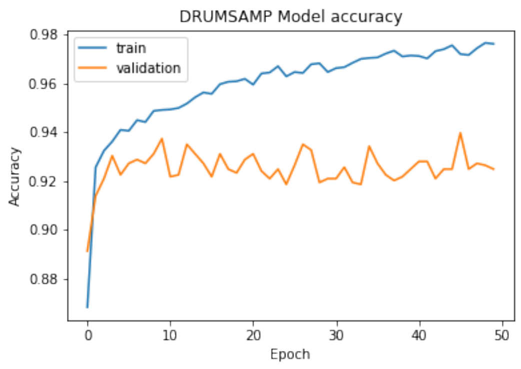

Figure 8: Model Accuracy on training and validation data

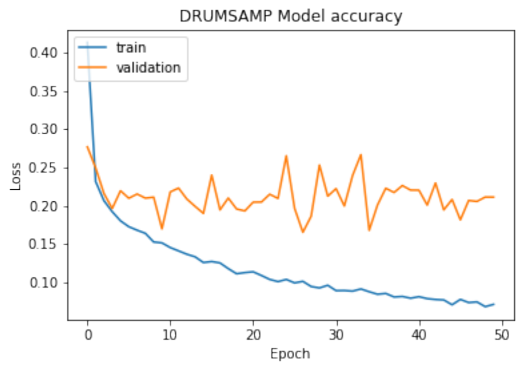

Figure 9: Model Loss on training and validation data

One can clearly see that the accuracy and loss on the training set is improving on every epoch iteration, while both metrices are quite stable in their behavior on the validation set after the first five epochs. The model shows no signs of overfitting in general, thus being an indicator of an overall good generalization performance. Looking at all epoch values for accuracy and loss, one can observe that epoch number 27 seems to be the best performing model on the validation set with an accuracy of 93.52% and an overall loss of 0.1653, making the model state during that stage a candidate for deployment. To examine the generalization performance of the model more thoroughly, section 4.2.3 investigates the model performance on the test set.

4.2.3 Validation

To examine the model performance, firstly the accuracy and loss of all model snapshots made throughout the single epochs are tested against the test set to determine the model which generalizes best on this unseen data. Figures 10 and 11 show the accuracy and loss of the single epoch snapshots on the test set respectively.

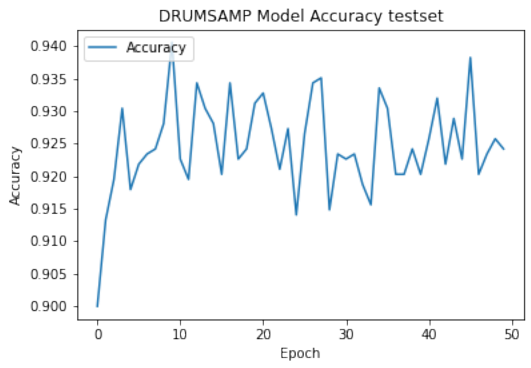

Figure 10: Model Accuracy on test data

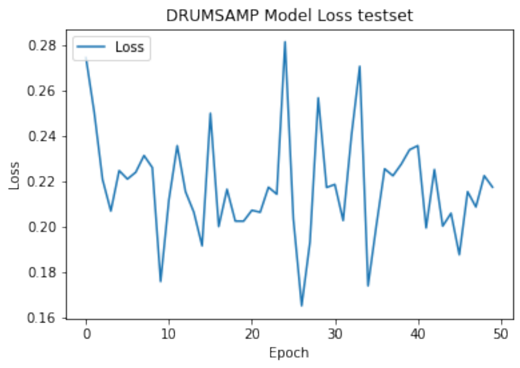

Figure 11: Model Loss on test data

Looking at both figures one can clearly see that the model at epoch 27, while performing best on the validation set, also performs best on the test set with an accuracy of 93.44% and an overall loss of 0.1654. Because this model shows the best generalization, it will be further evaluated.

Figure 12 shows the confusion matrix for the test set.

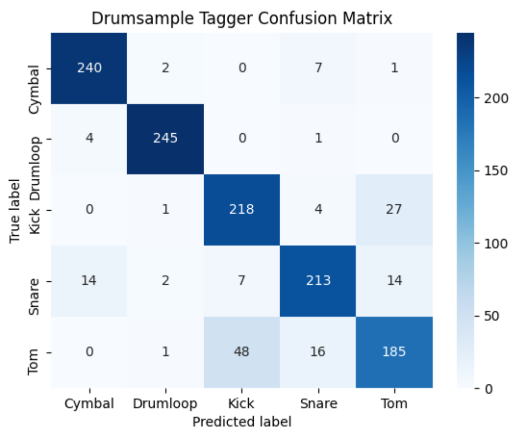

Figure 12: Drumsample Tagger Confusion Matrix

It can be seen that overall predictions seem to fit the ground trouth. However, one can observe that some sounds seem to be hard to distinguish for the model. Those combinations seem to be *Kick* and *Tom*, *Tom* and *Snare* and to some extent *Cymbal* and *Snare*. This also seems to go well with the observation that the timbre of those classes of sounds can be quite similar. For example, tom- and kick drums both tend to have some overlapping characteristics sometimes.

When feeding the model with unexpected data, it also has difficulties in recognizing those as unexpected. Because of that, a mechanism was implemented which is described in 4.4, to tag sounds below a specific threshold as *unknown*.

4.3 k-NN based Custom Tagger and Recommendation System

The custom Tagger and the RS both built upon the same concept. Both are using a k-NN approach with cepstral mean and variance normalized MFCCs. The system is described in the following sections.

4.3.1 Feature Selection

Both, the custom tagger and the RS are fed with cepstral mean and variance normalized MFCCs. Thus audio files are first transformed into MFCCs with 20 coefficients per frame. Thereafter, the CMVN described in section 2.2, is applied to the MFCCs to decrease unwanted distortions and additive noise effects. Those features are then stored to disk, to be accessible when the model requires the feature vector at prediction time. This speeds up prediction time of the k-NN as CMVN MFCC feature vectors just have to get computed once. Because sounds could be of different length, the shorter one of two compared sounds will be adjusted by zero padding to fit the other sound. This allows the usage of distance metrics like euclidean- or cosine distance.

4.3.2 Custom Tagger Structure

The custom Tagger works with a dictionary which holds sounds and related custom tags which were applied by the user. The current implementation allows every sound to have one custom tag. To calculate custom tags for a given library, the cosine distance between custom tagged sounds and all other sounds is calculated. All sounds which lie within a pre-defined range to the custom tagged sounds will be assigned the relating labels. The whole process will be further explained when the implementation of the custom tagger is discussed in section 4.4.

4.3.3 Recommendation System Structure

The RS, like the custom tagger, is also based on a k-NN approach. Because just the taste of the current user shall be considered, a content-based RS approach was chosen here. This allows the system to recommend sounds, which are similar to previous sounds the user already liked. Further modifications could be to add a collaborative-filtering approach to the RS. This would allow considering sounds which were liked by different users.

The similarity between sounds in the RS is measured using the cosine distance. The system then returns the n most similar sounds to the user, where n is an arbitrarily chosen number.

4.4 Model Package Implementation

To allow for better modularization of the code, the model library was built as a Python Wheel library. This allows for quick distribution and installation of a package via the Python Package Manager *pip*. The library code can be found on Github[9]. To install the library, the Wheel file from the repository can be installed via `pip install drumsamp_models-1.0-py3-none-any.whl`. The package is split into the submodules *tagger* (Drumsample Tagger), *ctagger* (Drumsample Custom Tagger), *recommender* (Drumsample Recommender) and *utils*.

The *utils* module contains the code required for the pre-processing of audio files. The *tagger* module contains the code for automatically tagging the sounds of a given Drumsample library. The code of the Drumsample Tagger is based on the actual implementation of Pons and Serra [15] and on the musicnn_keras library[10]. The *ctagger* and the *recommender* modules contain the code for the custom tagger and the RS. The following paragraphs explain the functionality and usage of all modules with examples.

Drumsample Tagger Example An exemplary use case of the tagger is shown in listing 1. To create the the melspectrograms for a given sample library, the `utils.save_classification_batches_to_disk()` (line 9) function has to be called. When the spectrograms are computed, a prediction can be made by calling `tagger.predict_tags_on_computed_mel()` (line 14). The function `tagger.predict_tags_on_computed_mel()` also has a possible second parameter, which is called *unknown_ threshold*. The default value of the threshold is set to 0.6 and it determines the minimum likelihood the model needs to predict a certain class. If no likelihood passes the threshold, a sound is classified as *unknown*.

```
1 from drumsamp_models import utils, tagger
2 from pathlib import Path
3 
4 audio_files = Path('./Sample_Lib/').glob('**/*.wav')
5 audio_files = [str(f) for f in audio_files]
```

[9] Drumsamp Models Package
[10] musicnn_keras

```python

classification_path = './classification_batches'

utils.save_classification_batches_to_disk(audio_files, classification_path)

classification_files = Path('./classification_batches').glob('**/*.npy')
classification_files = [str(f) for f in classification_files]

result = tagger.predict_tags_on_computed_mels(classification_files)

print(result)
```

Listing 1: Drumsample Tagger Example

Custom Tagger Example The usage of the custom Drumsample Tagger can be found in listing 2. When the recommendation batches have been successfully pre-processed (line 9), the `ctagger.get_custom_tag_nearest()`(line 23) function can be called for each successive file, where a custom prediction is required. When the distance is within the range of $cosine\_dist\_thresh = 0.002$ to one of the custom tagged files, the custom tag is applied to the sound. Notice that for the current implementation the maximum number of tags is one. The custom tags are passed as a dictionary with the filename as key and the custom tag as value (line 13-19).

```python
1  from drumsamp_models import utils, ctagger
2  from pathlib import Path
3
4  audio_files = Path('./Sample_Lib/').glob('**/*.wav')
5  audio_files = [str(f) for f in audio_files]
6
7  recommendation_path = './recommendation_batches'
8
9  utils.save_recommendation_batches_to_disk(audio_files, recommendation_path)
10
11 recommendation_files = Path('./recommendation_batches').glob('**/*.npy')
12 recommendation_files = [str(f) for f in recommendation_files]
13 custom_tag_dict = {
14   'file_name1': 'custom_tag1',
15   'file_name2': 'custom_tag1',
16   'file_name3': 'custom_tag1',
17   'file_name4': 'custom_tag2',
18   'file_name5': 'custom_tag2',
19 }
20
21 result_dict = {}
22 for file in recommendation_files:
23   custom_tags = ctagger.get_custom_tag_nearest(file, custom_tag_dict, 0.002)
24   result_dict[file] = custom_tags
25
26 print(result_dict)
```

Listing 2: Custom Drumsample Tagger Example

Drumsample Recommendation Example An example of the RS usage can be found in listing 3. Here, we also have to pre-compute the recommendation batches and save them to disk (line 9). Note that this only has to be done once per sample library for both, the RS and the custom tagger. To retrieve a list of recommendations, the recommender needs the number of recommendations to output, in this case eight, a list of favored files from a given user, the path to the pre-computed recommendation batches and a flag indicating if the favorite files are contained in the library, to exclude them for recommendation.

```
1  from drumsamp_models import utils, recommender
2  from pathlib import Path
3
4  audio_files = Path('./Sample_Lib/').glob('**/*.wav')
5  audio_files = [str(f) for f in audio_files]
6
7  recommendation_path = './recommendation_batches'
8
9  utils.save_recommendation_batches_to_disk(audio_files, recommendation_path)
10
11 recommendation_files = Path('./recommendation_batches').glob('**/*.npy')
12 recommendation_files = [str(f) for f in recommendation_files]
13
14 recommendations = recommender.get_n_most_similar_sounds_mult(8, <
       list_of_favorized_files_from_lib>, './recommendation_batches', True)
15
16 print(recommendations)
```

Listing 3: Drumsample Recommendation Example

5 System Design

The system follows a basic client-server architecture with the server running the FLASK backend which communicates with an SQLite-database and the frontend, which is a single page web application built using the Angular framework. The following sections introduce the whole system architecture while describing the implementation of the single components and the communication between them. Both, the frontend[11] and the backend[12] code can be found on Github.

5.1 System Overview

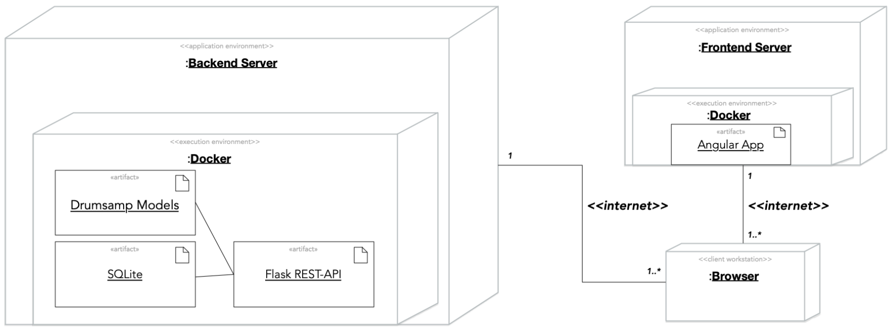

Figure 13: UML Deployment Diagram

Figure 13 shows an overview of the system architecture. The backend server runs a docker execution environment. Inside the docker environment, both the Flask REST-API and the SQLite database, are executed. The SQLite database is accessed via the SQLAlchemy Python ORM API[13] to allow for possible changes of the underlying database with minimum amount of code refactoring when required. Another external library which is accessed is the *drumsamp_ models* package, which was described in section 4.4.

It can also be seen that the Angular application is served via a second server for production purposes which also runs a docker environment to execute the app. For development purposes both, the backend server and the frontend server, are running on the same machine. When a user accesses the Drumsample Page, a HTTP-GET request is made to receive the page from the frontend server. After serving the frontend to the browser, the user can start to interact with the application. While accessing data resources, the frontend application sends Asynchronous JavaScript and XML (Ajax) requests to the backend server to retrieve the required resources.

[11] Drumsampler Frontend
[12] Drumsampler Backend
[13] SQLAlchemy

5.2 Database-Model

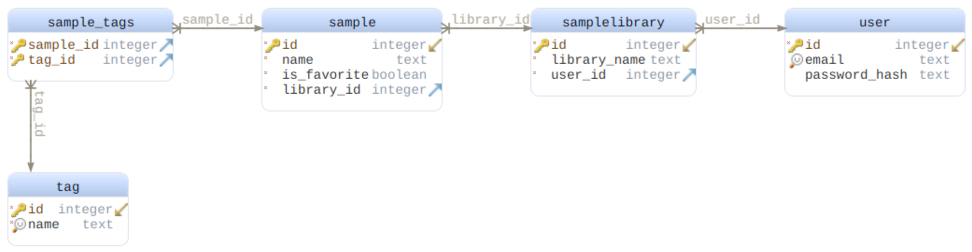

Figure 14: Database Structure

Figure 14 shows the implemented database model which holds all user specific data on the backend. As one can observe, the database holds five tables in total. The *user* table stores the user specific data and has a $1 : N$ relation to the *samplelibrary* table, meaning a user can own multiple soundlibraries. Each library can hold multiple samples, which is modeled by a $1 : N$ relationship between the tables *samplelibrary* and *sample*. Because each sample can have multiple tags associated to them and each tag can be associated with multiple samples, a $N : M$ relationship between the *sample* table and the *tag* table is required. This is done by holding the *sample_ id* and *tag_ id* in the *sample_ tags* table.

The implementation of the models can be found in the backend code repository in the file *./model/Model.py*.

5.3 User-Authentication

For the user authentication process JSON web tokens (JWT), as introduced by Jones, Bradley, and Sakimura [10] are issued when the user provides a valid email and password combination. When a user first registers for the application, a POST request with the content type `Content-Type: application/x-www-form-urlencoded` is issued to the REST-API endpoint `/register` with a chosen email and password combination. The backend then hashes the password chosen by the user and saves the data to the database. When the data has been successfully written to the database a JWT is constructed, which holds the usermail as payload and expires 30 days after creation. This token is transferred back via the response body.

When a user is already registered and wants to access a protected resource or his JWT expired, he first has to login sending, a `Content-Type: application/x-www-form-urlencoded` request to the `/login` endpoint providing the correct email and password combination. When the combination is valid, a JWT will be constructed and transferred in the response body.

The issued JWT has to be provided by all following requests to protected resources via the HTTP Authorization Bearer Header, setting the header to: `Authorization: Bearer <issued token>`.

5.4 REST-API

The REST-API controls the access to all resources saved in the database and delegates the model computations to the *drumsamp_model*. Appendix A shows the UML component diagram of the backend with all provided endpoints which are offered as interfaces. Appendix C displays all possible endpoints, which HTTP methods including content type and request data format they accept, and finally shows how the response data is formatted and encoded.

The endpoints `/register` and `/login` were already described in section 5.3, thus this section expounds the remaining endpoints. As all remaining endpoints are working with user specific data, all endpoints are protected and need to be accessed, providing the issued JWT via the HTTP Authorization Headers as explained in section 5.3.

When a user wants to list all created libraries, the frontend sends a GET request to the endpoint `/get/libraries` with no further request data added. The backend replies with a JSON array of strings which include the names of all libraries created by a user.

When a user decides to access a specific library, the `/get/library/<lib_name>` endpoint is used. The `<lib_name>` placeholder is the last path component of the endpoint and a valid library name has to be specified here. When the name is valid a JSON object, a response will be transferred back to the client, containing the library name and a list of all included sounds with the accompanying metadata added. When the name is invalid, however, the server will respond with the HTTP status code 400.

When a user creates a new library, three successive endpoint calls are made by the frontend. The first call is the POST request to `/upload/<lib_name>`. The `<lib_name>` placeholder specifies the new library name and the file parameter sent with content type `Content-Type: application/x-www-form-urlencoded` represents a file which is saved to the backend. When the library is not existent, a folder is created with the library name and the user id appended. For uploading multiple files the given endpoint has to be called for every file. Note that currently the only supported audio format is *.wav*. When this process is finished, the `/analyze/<lib_name>` endpoint is called. This starts the analysis process for the classification and recommendation pre-processing stage. Depending on the amount of audio files in the library, the analysis can take some time. When all files have been analyzed, the frontend sends the request to compute all tags with the Drumsample tagger. The endpoint called is `/tags/<lib_name>`, which invokes the classification model while saving all results to the database and responding to the client with the same formatted output as for the already described `/get/library/<lib_name>` endpoint.

For receiving recommendations based on liked sounds by the user a POST request to the `/samplefavorite/update/<lib_name>/<sample_name>` endpoint has to be issued. The request payload needs to be JSON encoded and needs to contain an array of JSON objects with the property `name` set to a favorited filename. The response of the request is JSON encoded and contains an array of JSON objects with the given filename of a sound which could be of interest to the user.

To apply custom tags to a given library, a POST request to the `/usertags/<lib_name>` endpoint has to be sent. The content type needs to be `Content-Type: application/JSON` and the data sent has to be an array of objects containing the name for a given file and an array of the assigned custom tags. After computing the result on the backend using the *drumsamp_models* package, a response is sent containing a dictionary with filenames as keys and the list of custom tags as values. Note that although the request contains a list of custom tags, solely the first element of the array is considered.

This limitation is set by the *drumsamp_ models* package.

The last two requests are both PUT requests and modify an existing sample in the library. To modify the tags of a given sound a request to the endpoint `/sampletag/update/<lib_name>/<sample_name>`, containing the library and the sample name has to be made. The request data has to be a JSON object, containing all tags of the given sounds formatted as displayed in Appendix C. To modify if a sound is favored a request to the endpoint `/samplefavorite/update/<lib_name>/<sample_name>` has to be issued. The request data has to include a JSON object, containing a flag indicating if the sound is favored by the user or not.

5.5 Frontend

The frontend is the component which is directly in interaction with the user and which is the starting point for all communication to the backend. Appendix B shows the UML component diagram of the frontend application. All components in the diagram relate directly to a user interface component of the system and every component has a service related to it which acts as a layer to handle the business logic and can be seen as the interface to the backend. Therefore, the application follows the design principles defined by the Angular Framework[14]. One special service is the `AppStateService`, which is injected into every component and which holds the complete state of the application. All other services are solely handling requests to the backend, while providing observable data structures for the relating view components to listen for backend responses.

[14] Angular Architecture

6 Conclusion and Limitations

During the course of the project, a working prototype of a distributed drum sample manager was built which utilizes machine learning methods to achieve the task of automatically tagging audio data. Therefore, a dataset was collected that contains 7500 drum sounds split equally over the 5 classes *Kick*, *Snare*, *Cymbal*, *Tom* and *Drumloop*. This dataset was then used to train a classifier based on the MUSICNN models and achieves an accuracy of 93.44% on the test data. To allow users to customize drum sound tags and let a model adapt to those tags, a k-NN approach was chosen that is also utilized for the RS.

All created models were implemented in the *drumsamp_ models* package which can be installed via the *pip* package manager. The resulting library was then used to create a distributed drum sample manager which follows a basic client server architecture. The application allows a user to utilize machine learning technologies to automatically create arbitrary drum sound libraries, tag all sounds automatically into the classes *Kick*, *Snare*, *Cymbal*, *Tom*, *Drumloop* and *Unknown*, create custom tags, let the model apply those to the rest of samples, and to retrieve recommendations based on user history.

Despite all achievements during the project, the models and the resulting application still have some limitations which can be addressed in future research. Those limitations include:

- The only supported audio format is *wave*.
- Users can apply only one custom tag to a given sound.
- JWT authentication and Flask Backend are not ready for deployment, as some security best practices have not been applied because this was beyond the scope for this work.
- User recommendations are not saved to the database, and thus have to be recomputed for every request.
- If a library is fed with new sounds or sounds are renamed, the system currently has no chance in reacting to those circumstances.
- When a dataset becomes large, tensorflow has memory allocation problems

References

[1] T. Bäckström. *Cepstrum and MFCC - Introduction to Speech Processing - Aalto University Wiki.* URL: https://wiki.aalto.fi/display/ITSP/Cepstrum+and+MFCC (visited on 07/09/2021).

[2] A. Band. *How to find the optimal value of K in KNN? | by Amey Band | Towards Data Science.* URL: https://towardsdatascience.com/how-to-find-the-optimal-value-of-k-in-knn-35d936e554eb (visited on 07/10/2021).

[3] T. Bertin-Mahieux et al. "The Million Song Dataset". In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011).* 2011.

[4] R. Burke. "Hybrid Web Recommender Systems". In: *The Adaptive Web: Methods and Strategies of Web Personalization.* Ed. by P. Brusilovsky, A. Kobsa, and W. Nejdl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 377–408. ISBN: 978-3-540-72079-9. DOI: 10.1007/978-3-540-72079-9_12. URL: https://doi.org/10.1007/978-3-540-72079-9_12.

[5] J. Droppo and A. Acero. "Environmental Robustness". In: *Springer Handbook of Speech Processing.* Ed. by J. Benesty, M. M. Sondhi, and Y. A. Huang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 653–680. ISBN: 978-3-540-49127-9. DOI: 10.1007/978-3-540-49127-9_33. URL: https://doi.org/10.1007/978-3-540-49127-9_33.

[6] J. Frochte. *Maschinelles Lernen: Grundlagen und Algorithmen in Python.* ger. 2., aktualisierte Auflage. München: Hanser, 2019. ISBN: 978-3-446-45996-0.

[7] S. Furui. "Speaker-independent isolated word recognition based on emphasized spectral dynamics". In: *ICASSP '86. IEEE International Conference on Acoustics, Speech, and Signal Processing.* Vol. 11. 1986, pp. 1991–1994. DOI: 10.1109/ICASSP.1986.1168654.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* http://www.deeplearningbook.org. MIT Press, 2016.

[9] G. E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors.* 2012. arXiv: 1207.0580 [cs.NE].

[10] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT).* RFC 7519. http://www.rfc-editor.org/rfc/rfc7519.txt. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7519.txt.

[11] E. Law et al. "Evaluation of Algorithms Using Games: The Case of Music Tagging." In: Jan. 2009, pp. 387–392.

[12] J. Lyons. *Practical Cryptography.* URL: http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/ (visited on 07/09/2021).

[13] S. J. Pan and Q. Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22 (2010), pp. 1345–1359.

[14] J. Pons et al. "End-to-end Learning for Music Audio Tagging at Scale". In: *ArXiv* abs/1711.02520 (2018).

[15] J. Pons and X. Serra. "musicnn: pre-trained convolutional neural networks for music audio tagging". In: *Late-breaking/demo session in 20th International Society for Music Information Retrieval Conference (LBD-ISMIR2019)*. 2019.

[16] H. Purwins et al. "Deep Learning for Audio Signal Processing". In: *IEEE Journal of Selected Topics in Signal Processing* 13.2 (2019), pp. 206–219. DOI: 10.1109/JSTSP.2019.2908700.

[17] F. Ricci, L. Rokach, and B. Shapira. "Introduction to Recommender Systems Handbook". In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 1–35. ISBN: 978-0-387-85820-3. DOI: 10.1007/978-0-387-85820-3_1. URL: https://doi.org/10.1007/978-0-387-85820-3_1.

[18] D. (Sarkar. *A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning*. en. Nov. 2018. URL: https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a (visited on 07/09/2021).

[19] R. R. Sinha and K. Swearingen. "Comparing Recommendations Made by Online Systems and Friends". In: *DELOS*. 2001.

[20] S. S. Stevens, J. Volkmann, and E. B. Newman. "A Scale for the Measurement of the Psychological Magnitude Pitch". In: *The Journal of the Acoustical Society of America* 8.3 (1937), pp. 185–190. DOI: 10.1121/1.1915893. eprint: https://doi.org/10.1121/1.1915893. URL: https://doi.org/10.1121/1.1915893.

[21] H. Tripathi. *What Is Balanced And Imbalanced Dataset? | by Himanshu Tripathi | Analytics Vidhya | Medium*. URL: https://medium.com/analytics-vidhya/what-is-balance-and-imbalance-dataset-89e8d7f46bc5 (visited on 07/10/2021).

Appendices

A Backend Server Component Diagram

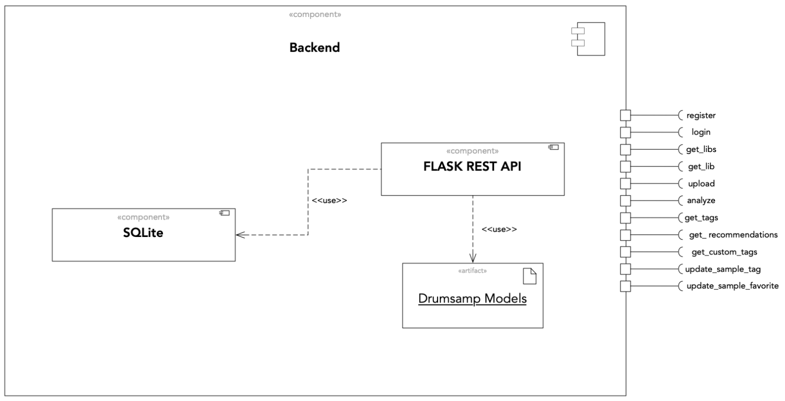

Figure 15: UML Component Diagram - Backend

B Frontend Server Component Diagram

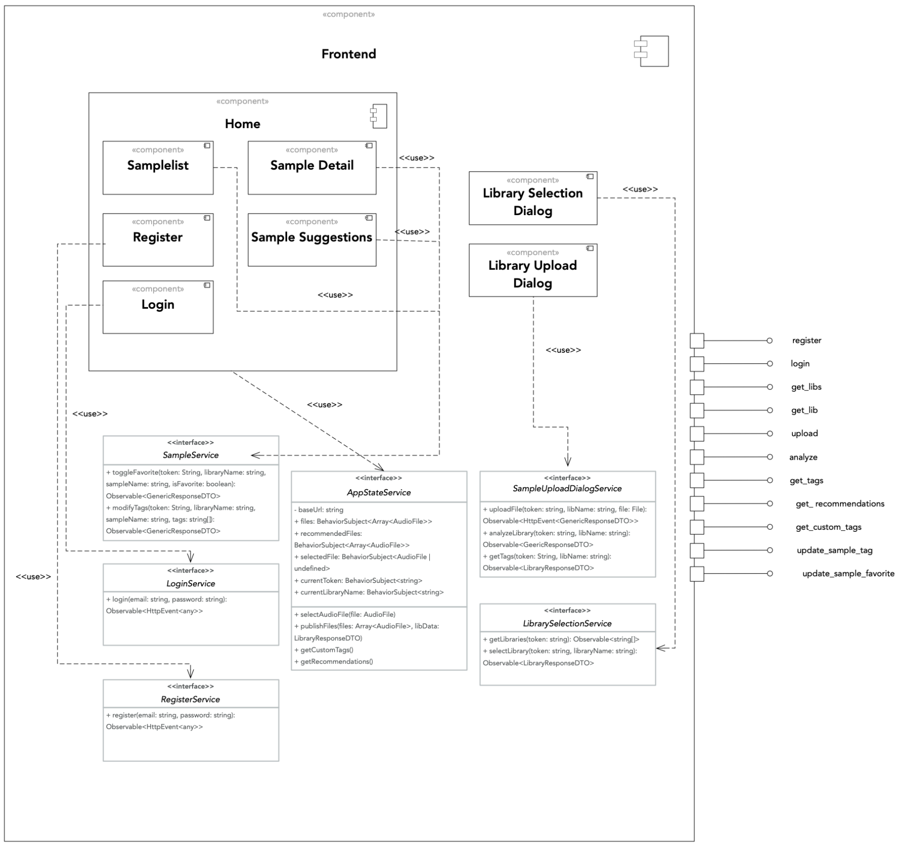
«component»
Frontend
«component»
Home
«component»
Samplelist
«component»
Sample Detail
«component»
Register
«component»
Sample Suggestions
«component»
Login
«component»
Library Selection Dialog
«component»
Library Upload Dialog
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<interface>>
SampleService
+ toggleFavorite(token: String, libraryName: string, sampleName: string, isFavorite: boolean): Observable<GenericResponseDTO>
+ modifyTags(token: String, libraryName: string, sampleName: string, tags: string[]): Observable<GenericResponseDTO>
<<interface>>
LoginService
+ login(email: string, password: string): Observable<HttpEvent<any>>
<<interface>>
RegisterService
+ register(email: string, password: string): Observable<HttpEvent<any>>
<<interface>>
AppStateService
- baseUrl: string
+ files: BehaviorSubject<Array<AudioFile>>
+ recommendedFiles: BehaviorSubject<Array<AudioFile>>
+ selectedFile: BehaviorSubject<AudioFile | undefined>
+ currentToken: BehaviorSubject<string>
+ currentLibraryName: BehaviorSubject<string>
+ selectAudioFile(file: AudioFile)
+ publishFiles(files: Array<AudioFile>, libData: LibraryResponseDTO)
+ getCustomTags()
+ getRecommendations()
<<interface>>
SampleUploadDialogService
+ uploadFile(token: string, libName: string, file: File): Observable<HttpEvent<GenericResponseDTO>>
+ analyzeLibrary(token: string, libName: string): Observable<GeericResponseDTO>
+ getTags(token: String, libName: string): Observable<LibraryResponseDTO>
<<interface>>
LibrarySelectionService
+ getLibraries(token: string): Observable<string[]>
+ selectLibrary(token: string, libraryName: string): Observable<LibraryResponseDTO>
register
login
get_libs
get_lib
upload
analyze
get_tags
get_ recommendations
get_custom_tags
update_sample_tag
update_sample_favorite

Figure 16: UML Component Diagram - Frontend

C REST-API Endpoint Description

Method	ENDPOINT	REQ-DATA	REQ Content-Type	RES-DATA	RES Content-Type
POST	`/register`	mail: string, password: password	`application/x-www-form-urlencoded`	`{token: string}`	`application/json`
POST	`/login`	mail: string, password: password	`application/x-www-form-urlencoded`	`{token: string}`	`application/json`
GET	`/get/libraries`	None	-	`string[]`	`application/json`
GET	`/get/library/<lib_name>`	None	-	`{libname : string, samples: {name: string, isFavorite: boolean, tags: string[]}[] }`	`application/json`
POST	`/upload/<lib_name>`	file: File	`application/x-www-form-urlencoded`	`{message: string}`	`application/json`
GET	`/analyze/<lib_name>`	None	-	`{message: string}`	`application/json`
GET	`/tags/<lib_name>`	None	-	`{libname : string, samples: {name: string, isFavorite: boolean, tags: string[]}[] }`	`application/json`
POST	`/recommendations/<lib_name>`	`{name: string}[]`	`application/json`	`{name: string}[]`	`application/json`
POST	`/usertags/<lib_name>`	`{name: string, customTags: string[]}[]`	`application/json`	`{string: strings[]}`	`application/json`
PUT	`/sampletag/update/<lib_name>/<sample_name>`	`{tags: string}[]`	`application/json`	`{message: string}`	`application/json`
PUT	`/samplefavorite/update/<lib_name>/<sample_name>`	`{isFavorite: boolean}[]`	`application/json`	`{message: string}`	`application/json`

Table 2: REST-API Endpoint Description