Multiple Predominant Instrument Classification — Technical Documentation

Lumen 2023. — Data Science

Contents

1. Introduction

This document is a high-level technical description of models, processes and application developed for Lumen 2023 Data Science student competition. The goal of the competition was to develop a model for recognition of musical instruments in polyphonic music. Besides the model itself, the task was to develop an API by which the model can be used.

2. Technologies and Processes

We used a version control system (Git) to keep track of changes to make it easier to collaborate with others and to revert to previous versions if necessary. While developing models, we relied on virtual environments in `conda` to manage our project dependencies. This ensured that the project dependencies are isolated from other projects on our local machines.

For application development, we used `pip` and `requirements.txt` file to keep all the needed dependencies in one place. Furthermore, we used Docker for building the image of the application and Make for defining some commands for easier building and running of application when using Docker.

Finally, we structured both the API part and the machine learning part in a modular way so that new functionalities such as preprocessing and augmentations can be added extremely simply.

3. Project organization

On a project of this magnitude, with many qualitatively distinct parts, it is of paramount importance to organize the code and workflow properly, otherwise simple additions such as new architectures or new augmentation/mixing methods would become increasingly more difficult and intractable. Our complete project's structure, as it looks on local machines, is given in Figure 3.1.

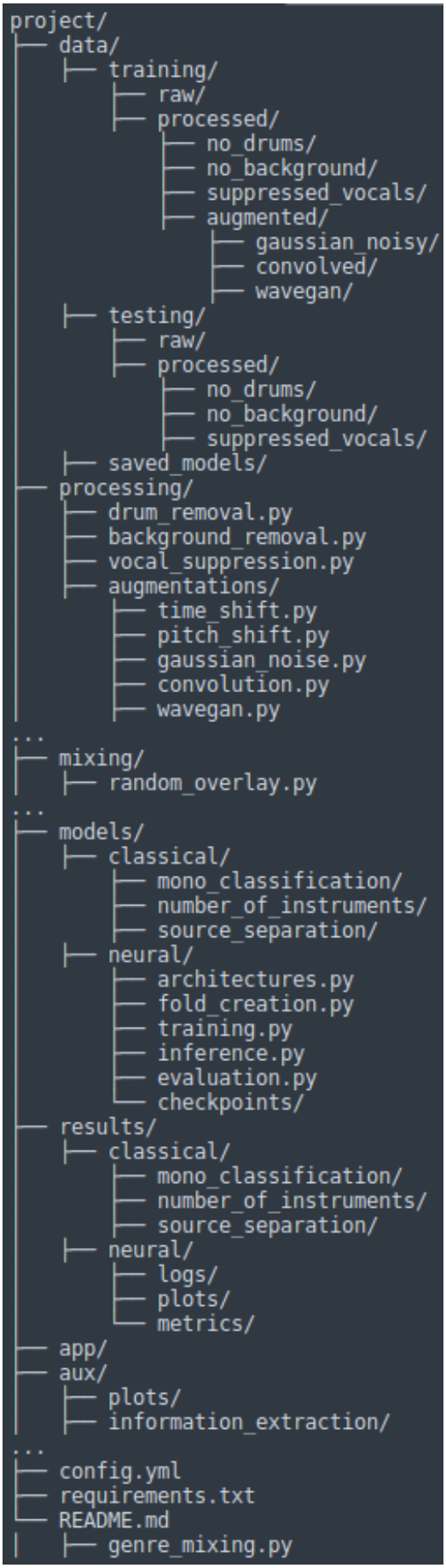


Figure 3.1.: Hierarchical organization of the project.

The most important aspect of our organization is the aforementioned separation of different parts of the model (e.g. machine learning and application parts) — we abide by the principle that these parts should communicate on a higher level, while knowing very little about each other's details. For example, neural network will produce some predictions based on loaded data and then pass them to API which will output them, without knowing anything about the process of inference.

4. Application

4.1. Back end

Back end of application is built using Python and Flask framework. Although the back end is technically a standalone API, Flask offers usage of HTML templates which we will mention later. There are two endpoints in the API, `/analyze_files/single_model` and `/analyze_files/multi_model`. Single model mode is used when we want to use only one model for prediction of instruments. We can choose which model we want to use by specifying its name. Of course, there is a default model in case we don't choose any. On the other hand, Multi model mode is used to get predictions of all the models we have loaded in application, if their weight is set to be greater than zero. We send these weights for each model to the back end endpoint where we weigh each prediction appropriately. There are four files regarding the models. First file is "Model" which is parent class to two specific models classes, which was introduced to avoid repetition of code in these two specific classes.

Last file in regards to the models is "model_runner.py" — it is used to consolidate differences between two endpoints and to parallelize execution — we have done so by using thread pool so we can utilize all the threads on the machine. Parallelization was done with regard to different songs, so if we send multiple songs for predictions to back end, each song is processed in a separate thread.

4.2. Front end

Front end is developed using HTML, CSS, JS and Flask's template engine called Jinja. When you run the application using Docker, there will be a web application running on *localhost:8080/*. It is a simple UI that lets user upload one or more .wav files that are sent to the back end for predictions, as can be seen in Figure 4.1.

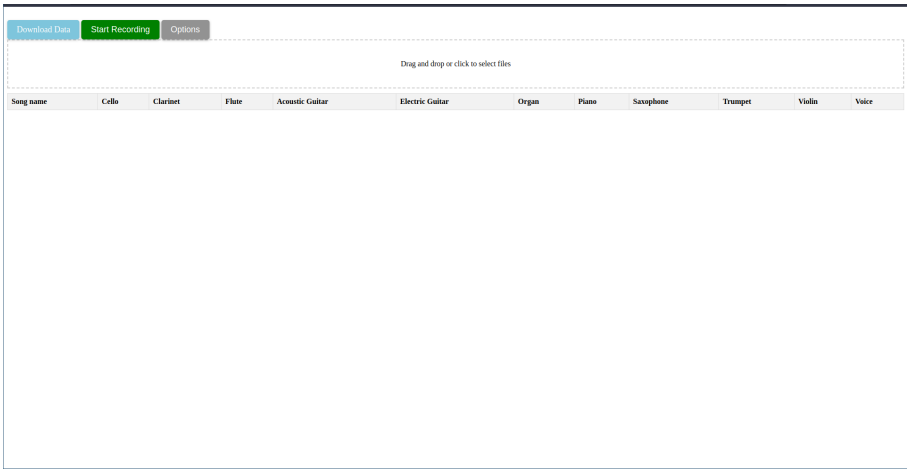


Figure 4.1.: Landing page of our application.

After they have been uploaded, files are sent automatically and evaluated by the model or a mix of models that can neatly be regulated by sliders (Figure 4.2).

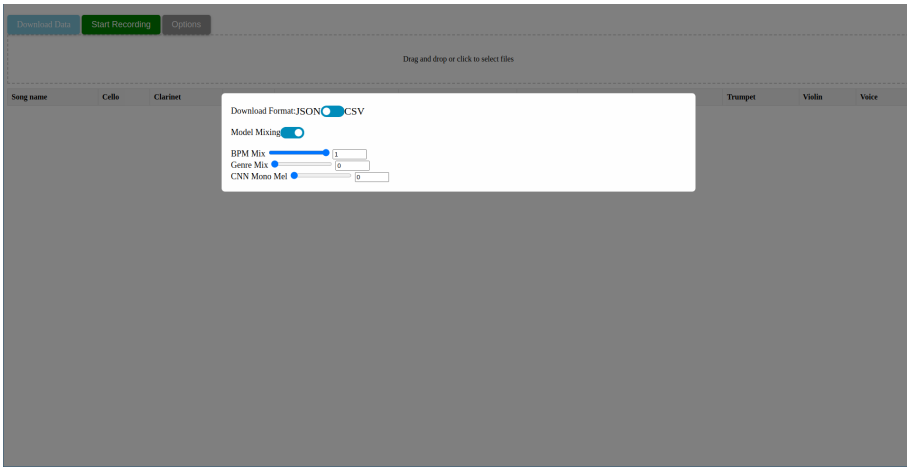
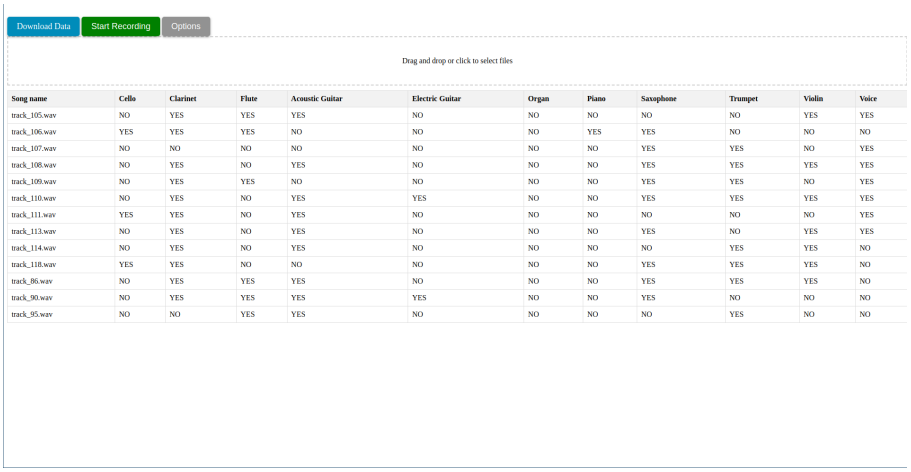


Figure 4.2.: Choices of weights of different models in final predictions, as well as the format in which we wish to download the data.

Calculated predictions are outlined in tabular form (as in Figure 4.3), Download button becomes available and we can download predictions locally as a file of name "wav-files-instruments-predictions", with file extension being either CSV or JSON.



Song name	Cello	Clarinet	Flute	Acoustic Guitar	Electric Guitar	Organ	Piano	Saxophone	Trumpet	Violin	Voice
track_105.wav	NO	YES	YES	YES	NO	NO	NO	NO	NO	YES	YES
track_106.wav	YES	YES	YES	NO	NO	NO	YES	YES	NO	NO	NO
track_107.wav	NO	NO	NO	NO	NO	NO	NO	YES	YES	NO	YES
track_108.wav	NO	YES	NO	YES	NO	NO	NO	YES	YES	YES	YES
track_109.wav	NO	YES	YES	NO	NO	NO	NO	YES	YES	NO	YES
track_110.wav	NO	YES	NO	YES	YES	NO	NO	YES	YES	YES	YES
track_111.wav	YES	YES	NO	YES	NO	NO	NO	NO	NO	NO	YES
track_113.wav	NO	YES	NO	YES	NO	NO	NO	YES	NO	YES	YES
track_114.wav	NO	YES	NO	YES	NO	NO	NO	NO	YES	YES	NO
track_118.wav	YES	YES	NO	NO	NO	NO	NO	YES	YES	YES	NO
track_86.wav	NO	YES	YES	YES	NO	NO	NO	YES	YES	YES	NO
track_90.wav	NO	YES	YES	YES	YES	NO	NO	YES	NO	NO	NO
track_95.wav	NO	NO	YES	YES	NO	NO	NO	NO	YES	NO	NO

Figure 4.3.: Tabular form of output predictions.

4.3. Application — Future Improvements

There are several possible application improvements for the future.

In the application there is a "Start Recording" button which records the sound from your microphone. In scope of this project, the infrastructure needed to implement this way of input is mostly done, however we suppose that it would be more appropriate to add some preprocessing steps since our model is trained on high quality data (albeit with augmentations such as noise addition).

Similarly, although the UI is intuitive and easy to use, it could be improved aesthetically as well as ordered in more application-like manner so as to make it easily migrated to mobile platforms as well.

5. Preprocessing the data and training the models

Using the `config.yml` file in root directory, one can easily change the number and names of instruments in the system and only with few additional changes repurpose our neural networks to classify less or more instruments, and different ones at that. To be able to do this, one must also install the pip requirements contained in `requirements.txt` and then set the `PROJECT_ROOT` (inside current environment) to root directory of the project on local computer. After this, simply choosing the desired paths for data, augmented datasets are generated on local machine.

Following this, using the notebook for generating training folds, five of them are obtained — this is done so that each file contributes both to training and validating; in this way a more robust and balanced models are obtained.

As was discussed in scope of the project documentation, there are also several ways in which our neural models could be improved on a machine learning level, so at this moment

we only reiterate that due to parallelization, our application is efficient when mixing several models — thus, it is well-suited for use on personal computers, as well as mobile phones.

Regarding the computational requirements to train any of listed models, it is highly recommended to use GPU based computing since CPU based ones are usually significantly slower. We trained most of our models on 3 different GPU-s. We used GTX1070, RTX3090, and V100 which is available with pro Colab computing. All of them achieved results within margin of error, thus making our work reproducible. Training of model2 architecture occasionally crashed, probably due to insufficient RAM memory at the time. model3 architecture is lightweight and may be best for day to day usage. It consists of around 2M parameters whereas model2 architecture has more than 10M parameters. We did not tune several model parameters such as mini batch size so as to achieve more robust results, as mentioned in project documentation.

WaveGan architecture is not listed in our models since we used public implementation from chrisdonahue's GitHub repository. Please note that the stock architecture available on that repo is not adjusted to support newer versions of Tensorflow, so simple adjustments in code have been made . In the Issues section, one can find GitHub repository with mentioned modifications. Training WaveGan is highly computationally expensive, but also expensive in the traditional way since training 4000 epochs for each instrument lasts for around 2 days and Colab offers only around 6 hours of V100 use for 12 euros. For that purpose, we selected optimal parameters so that our training performs as quickly as possible. In readme document, which will be available on cloud[1], we listed those settings.

An important notice must be made on a required disc space for each of described methods. Some augmentation methods require more than 50GB of disc space, but also on a CPU which must appropriately support GPU to accelerate loading of batches. We used Ryzen5 3600 and Colab multicore CPU for that purpose. RAM memory is also an important aspect since we load our data on RAM, obviously. We used 48GB ddr4 and 83.5GB from our config and Cloud config respectively. For a comparison, standard machine which is available in Kaggle notebook was not sufficient for handling our models.

All the models which we used are provided in cloud, for which the link will be provided.

Finally, our proposed model is built as an ensemble of 2 architectures, 3 datasets and 2 different augmentation strategies. We achieved best overall scores for weights 0.36, 0.46, 0.18 when applied to predictions of model2 with genre mixing and BPM mixing and model3, along with score level fusion. We show the complete processing procedure in Figure 5.1 below.

[1]The link to open Drive folder is given in separate txt file in root directory.

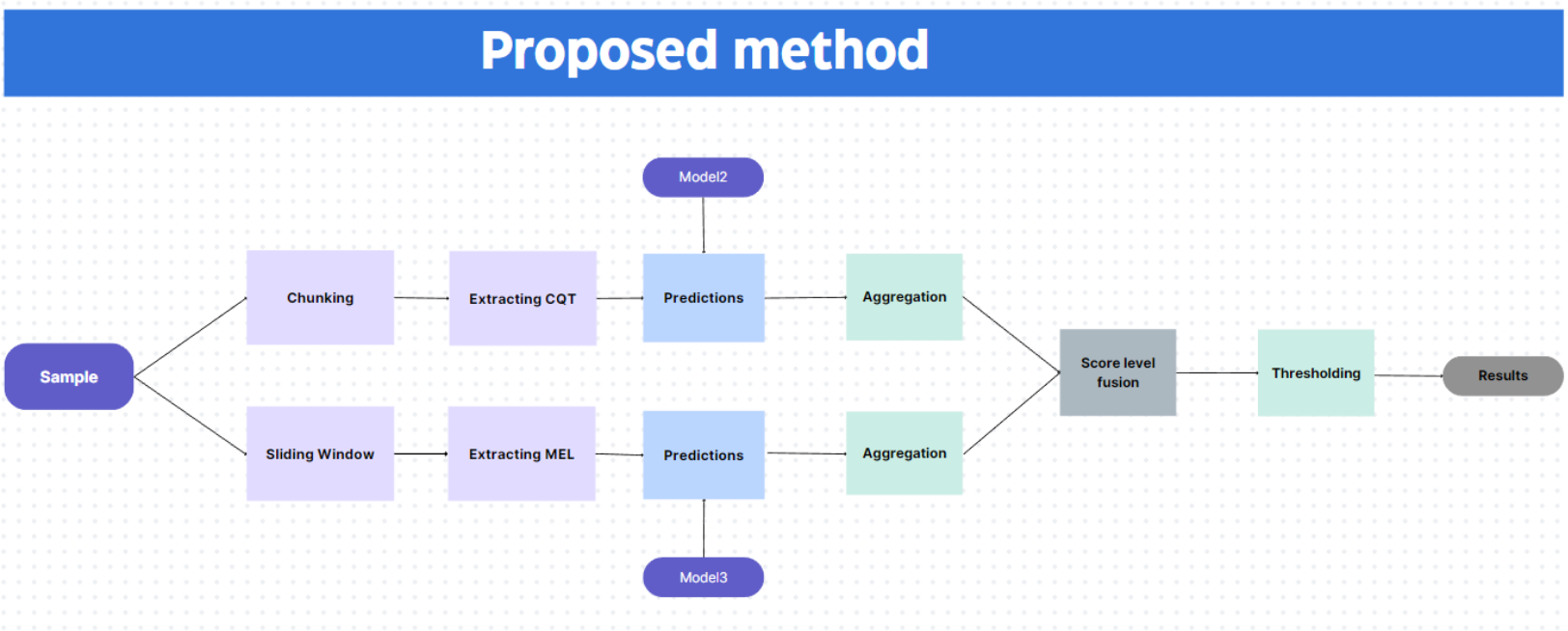


Figure 5.1.: Proposed method.