Instrument Recognition Software

CSULB 491B Computer Science Senior Project II

Architecture and Design Document

Convolutional Neural Network Architecture and Design

Application use case

Sound classification

The CNN ML model uses Power Spectral Density (PSD) graphs to predict sounds. The user can input a sound file and the ML model would classify the sound with one of the pre-trained labels. The model in the current state can handle 6 monophonic instruments: flute, guitar, saxophone, trumpet, tuba, violin.

Machine Learning 10 Steps

Business Case

The Instrument Recognition Software team began as a group of college senior programmers passionate about music that were looking for new ways to sort data. A prototype of the model was developed that focused on individual usage of the application to sort songs by instrument for entertainment purposes. As research continued, the team pivoted to a product model targeted toward enterprises that showcased sound recognition for custom, unique audio signals. Our current deployment uses monophonic instrument data to demonstrate a proof of concept however the product holds great potential for wide-spread application through transfer learning.

Data Sources

Philharmonic Orchestra URL: https://www.philharmonia.co.uk/explore/sound_samples

Import and Explore Data

Philharmonic Orchestra provides an adequate amount of audio recordings that we can use as our main source of training and testing datasets.

There are a total of 57 instruments, each with an adequate amount of samples with varying pitches, lengths, dynamics, and articulation.

For our approach, we are planning on only using a small fraction of the samples (2-3 instruments) as the process will become convoluted. Each instrument should have at least 150 sound samples. We need a close to equal amount of samples per instrument to train the model correctly.

Preprocess Data

The audio files are in a lossy MPEG Audio Layer-3 format (MP3), which we will be converting into a lossless Waveform audio file format (WAV) in order to enhance the clarity and readability of the audio file for better analysis.

Each of the WAV sound files then gets converted into a Power Spectral Density graph using a fast Fourier transform algorithm in python. We save the graphs in the JPEG format.

Before the data can be used in the model, we resize the PSD JPEG images to 360 x 1440 pixels. That step is necessary to ensure the data in the model does not differ in size as it could alter the training and testing phase. If the PSD graph axis do not align for all samples it would yield incorrect results.

We also convert all the graphs to gray scale, so the model does not train, or test based on graph color. This is another precaution we take to ensure all data aligns and no other factors get in the way of getting correct results.

We then convert the image to a three-dimensional array of numbers that our model will train and test on.

Split Data (Train vs Test)

The testing data is no different from the training data. We use the same pre-processing for both. In fact, our testing data is taken out of the 150-200 samples per instrument to ensure that the model is training on the data correctly. Using test data from a different source would not be as reliable for us in the early stages of training the model because we might not be able to tell the correct predictions apart from incorrect predictions.

Select and Create Models

Our team decided to implement a Convolutional Neural Network to handle the classification of our data. This was a logical conclusion as our input was in the form of an image and we required label outputs. Through trial and error we settled on the use of 14 hidden layers to digest the data into higher level features.

Train Model

The model was trained on 200 Power Spectral Density graphs made of one to three second audio clips from singular instruments. Through experimentation we determined 14 hidden layers were ideal and resulted in a consistent training accuracy rating of over 95%.

Evaluate Results

On each training session for the model our program set aside 30 graphs per instrument to test against for validation data. This data was explicitly excluded from the training data to maintain the integrity of the evaluation and consistently scored over 90% on the final iteration of the model.

Make Predictions and Evaluate Results

Our initial model was expected to achieve +40% accuracy and ended up scoring exactly 33%, the equivalent of random chance when using three labels. Through reformatting the data to include more essential information we were able to achieve our previous estimate of 40%. In another endeavor to achieve a higher accuracy, we reduced our number of hidden layers from 22 to 14. This resulted in a jump to over 90% consistent accuracy. The change was inspired by the idea that our model may be simplifying our data into obscurity and removing the relevant information accidently.

Deploy Model, Monitor, and Reuse

Our initial model deployment premiered with three instrument labels and was quickly updated to include three more for a total of six. Our current trajectory has the model branching in three possible directions, those being the addition of complimentary instruments for heightening precision, shifting to polyphonic sound for a deeper breadth of application, or adventuring further into other auditory mediums besides music.

Deployment Architectural Choices

Local on user's device

The trained CNN model file along with the server and client files could be distributed as executables. The flaws to this deployment choice include no remote monitoring of services (application or model updates) and Python library dependencies.

When considering this deployment for a large audience we must keep in mind any updates that happen in the system. We would need to add a remote service to update any necessary components over time.

We also need to be mindful of the user device's storage restrictions. The trained CNN model is a large file. It also requires some Python libraries like tensorflow and keras which have system requirements.

Remote on Heroku and local on user's device

The trained CNN model file and potentially the server file could be hosted on Heroku and used as an API. The client and potentially the server file could be run locally on

user's device as they do not have system requirements as extensive as the trained CNN model. This is beneficial as we could upload updates to Heroku as needed and users would get the updated files within hours. The challenging side of this deployment choice would be ensuring Heroku services stay available. If Heroku services are down, so is our application's ML feature. We would also have to account for data transfer and possible data loss over the internet as well as ensure secure connections.

User Interface Architecture and Design

Trade Off Analysis

Client:

The client will be the users' data side of the software in which it contains the user menu for the user to pick and scroll through their options. Which will allow them to interact with the software in order to choose their song files as well as pick out all the different ways in which they want to categorize their songs. Assumingly, they will search through their favorite music based on how their favorite instrument which our algorithm will go through and determine for the users.

Server:

This is the server side and it will contain all of the data necessary to process the information we will be using in our software. This will include our login system information which contains the users username and password so that they may login to their own private account which contain their favorite songs.

Controller:

The controller will be used to manage through all of the updates and user inputs that occur within our program. It will serve as the main process for managing through all of the actions Model and View does.

View:

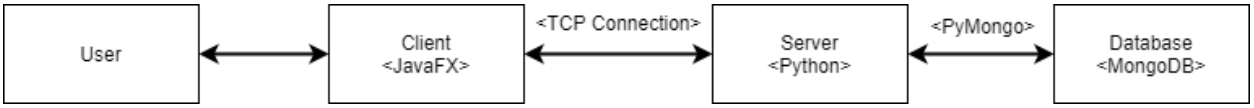
This shall be used to allow the user to see what is being processed so in a way it is a graphical user interface that our users will be using to interact with the software and allows ease of access.

Model:

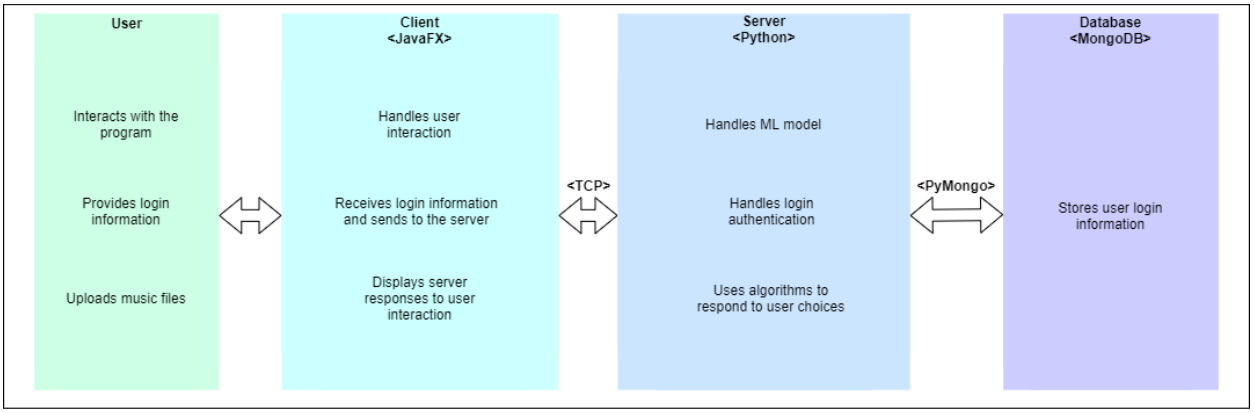
This shall contain the user login information as well as their music file so that we can notify the controller.

Criterion	Weight	Python	Java
Object Oriented	25%	0	1
Data Management	50%	1	0
Ease of Access	10%	1	0
Libraries	15%	1	0

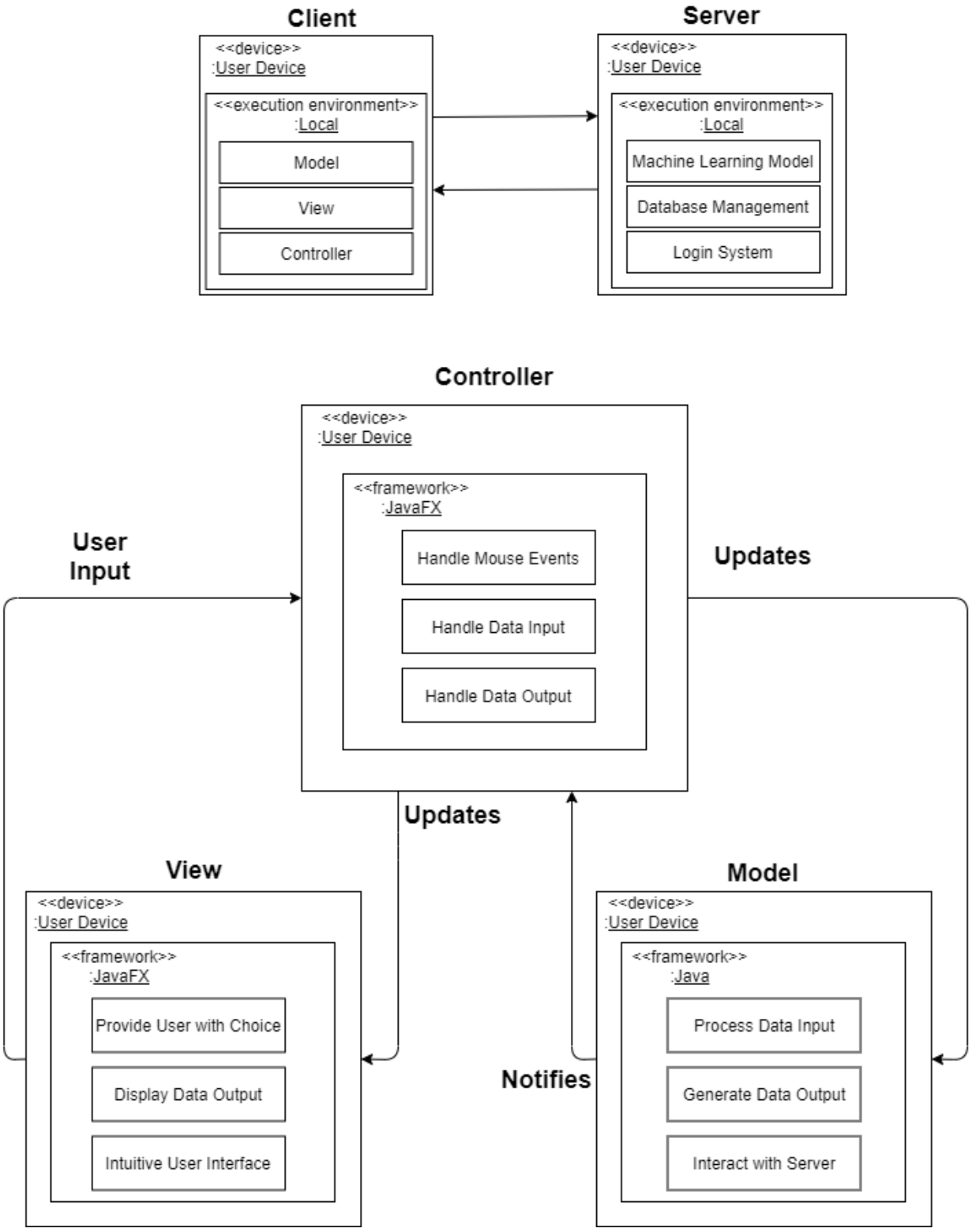
High Level Architecture Design



Low Level Architecture Design



Instrument Recognition Architectural Design



Technology Innovation and Depth

Note: Click on the component title to be linked to the code file on GitHub for physical evidence

Instrument Recognition Software consists of:

- Data pre-processing functions
- CNN model trained on 6 monophonic instruments
- A login and registration system
  - Store usernames and hashed passwords in non-relational database (lines 5 - 7)
  - Authentication handled in a Python server (lines 44 - 63)
  - User input handled in a Java client (lines 64 - 99)
  - Data transferred using a TCP connection
- Server
  - Handles login and registration
  - Perform fast Fourier transform on a WAV file to get a PSD graph
  - Handles input data pre-processing (preprocess(img) function)
  - Connects to non-relational database (lines 5 - 7)
  - Uses trained CNN model to make predictions (line 25 and 30)
  - Uses TCP connection to send and receive data from the UI
- User Interface
  - Written using Java, JavaFX, and FXML
  - Handles user interaction (sample controller file)
  - Provides simple UX (sample layout file)
  - Handles file upload/deletion (lines 87 - 147)
  - Plays an uploaded sound file (lines 186 - 198)
  - Uses TCP connection to send and receive data from the Server
  - Displays generated PSD graphs for files (lines 163 - 184)
  - Displays ML model prediction (lines 149 - 161)