Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

Institute of Control and Computation Engineering

# Master's diploma thesis

in the field of study Computer Science
and specialisation Internet Management Support Systems

## Machine learning framework for explainable artificial intelligence audio models

Maciej Falkowski, BSc
student record book number 329117

thesis supervisor
Mateusz Modrzejewski, PhD

WARSAW 2025

# Machine learning framework for explainable artificial intelligence audio models

**Abstract.**

The aim of this thesis is to design a tool that enables the execution, visualization, and comparison of Explainable Artificial Intelligence (XAI) methods for audio machine learning models. As part of this, the thesis describes the design of an audio XAI framework, which allows its user to integrate their models with the framework's API in the form of a library, which then performs explanations on the model and visualizes them in a web interface designed for the thesis. The thesis describes the developed framework architecture based on the Model-View-Presenter (MVP) pattern, which aims to ensure the solution's modularity, enabling integration of new user models and alternative views to the default one. The project integrates selected XAI methods, including Local Interpretable Model agnostic Explanations (LIME), Integrated gradients, and Layer-wise Relevance Propagation (LRP).

**Keywords:** Explainable artificial intelligence, Framework, Interpretability, Explainability, Model-View-Presenter, Integrated gradients, Layer-wise Relevance Propagation

## Framework uczenia maszynowego dla modeli audio wyjaśnialnej sztucznej inteligencji

**Streszczenie.**

Celem pracy magisterskiej jest zaprojektowanie narzędzia umożliwiającego wykonanie, wizualizację oraz zestawienie metod Wyjaśnialnej sztucznej inteligencji (XAI) dla modeli uczenia maszynowego audio. W ramach tego praca opisuje projekt frameworka wyjaśnień XAI modeli audio (audio XAI framework), który umożliwia użytkownikowi integrację jego modeli z API frameworka w postaci biblioteki, która następnie wykonuje wyjaśnienia na modelu, a następnie wizualizuje je w interfejsie webowym, który został zaprojektowany na potrzeby pracy. Praca opisuje opracowaną architekturę frameworku opartą o wzorzec Model-View-Presenter (MVP), której celem jest zapewnienie modularności rozwiązania, umożliwiającej integrację nowych modeli użytkownika oraz alternatywnych widoków względem domyślnego. Projekt integruje wyselekcjonowane metody XAI wliczając w nie Local Interpretable Model-agnostic Explanations (LIME), Integrated gradients oraz Layer-wise Relevance Propagation (LRP).

**Słowa kluczowe:** Wyjaśnialna sztuczna inteligencja, Framework, Interpretowalność, Wyjaśnialność, Model-View-Presenter, Integrated gradients, Layer-wise Relevance Propagation

# Contents


**1. Introduction** . . . 7

**2. Problem analysis** . . . 9

2.1. Black-box and white-box AI systems . . . 9

2.2. Explainable artificial intelligence . . . 9

2.3. Selection of explanation methods for framework . . . 11

2.4. Local Interpretable Model Explanation . . . 11

2.5. audioLIME . . . 12

2.6. Integrated gradients . . . 12

2.7. Layer-wise Relevance Propagation . . . 13

2.8. Sound waveform representation . . . 14

2.9. Selection of ML model for integration . . . 15

2.10. Dependency Inversion Principle . . . 16

2.11. Software framework . . . 16

2.12. Model-View-Presenter . . . 17

2.13. Audio XAI Framework architecture . . . 18

2.14. Similar solutions . . . 20

**3. Requirements and tools** . . . 21

3.1. Functional requirements . . . 21

3.1.1. Library application programming interface requirements . . . 21

3.1.2. User web interface requirements . . . 22

3.1.3. Terminal application requirements . . . 24

3.2. Non-functional requirements . . . 24

3.3. Tools specification . . . 25

**4. External specification** . . . 27

4.1. Software requirements . . . 27

4.2. Installation procedure . . . 27

4.3. Framework application programming interface . . . 27

4.4. Command-line interface . . . 28

4.5. Web interface . . . 28

**5. Framework implementation** . . . 34

5.1. Software architecture overview . . . 34

5.2. Project structure . . . 34

5.3. Model-View-Presenter implementation overview . . . 35

5.4. Interfaces for model adapters . . . 37

5.4.1. LIME method adapter . . . 37

5.4.2. Integrated gradients method adapter . . . 37

5.4.3. LRP method adapter . . . 38

5.4.4. ModelLabelProvider interface . . . 38

5.5. Model adapters . . . 38

5.6. Explainer classes . . . 41

5.6.1. Integrated gradients and LRP explainers . . . 42
5.6.2. LIME explainer . . . 44
5.7. Context class definition . . . 46
5.8. View classes . . . 48
5.8.1. Web view . . . 48
5.8.2. Debug View . . . 49
5.9. Backend Server implementation . . . 51
5.10. Web frontend implementation . . . 52
5.10.1. Frontend component overview . . . 53
5.10.2. Visualization of audio waveform . . . 54
5.10.3. Error and Notification system . . . 56
5.11. Explanation runner command-line script . . . 60
**6. Verification and validation** . . . 62
6.1. Explanation support for models . . . 62
6.2. Scope of testing . . . 64
6.3. Framework's unit tests . . . 65
**7. Conclusions** . . . 68
**References** . . . 71
**List of Symbols and Abbreviations** . . . 74
**List of Figures** . . . 76
**List of Tables** . . . 76
**List of Appendices** . . . 76

# 1. Introduction

Artificial Intelligence (AI) continues to grow rapidly, containing a broad range of applications, including audio processing. The increasing complexity of AI systems brings the challenge of understanding and interpreting their decision-making processes. Many state-of-the-art models, such as deep neural networks (DNNs) especially Convolutional neural networks [1], frequently used for audio analysis, operate as so-called *black boxes*. While these models deliver high accuracy, their inner workings often remain opaque to human observers, limiting trust, interpretability, and the ability to validate results.

In this situation, Explainable Artificial Intelligence (XAI) has emerged as a promising and quickly expanding area of research [2]. XAI aims to make the decision-making process of complex AI systems more transparent to allow developers and researchers to better understand why a model produces a given output. There are many different explanation techniques, each operating on distinct principles and offering a unique standpoint on a model's decision-making process [3]. As a result, it is often necessary to apply multiple explanation methods to the same input and then compare and contrast their results. For audio models, it is additionally valuable to visualize supplementary audio elements, such as audio playback or spectrogram representations, to provide richer context for analysis. This necessity, however, leads to repetitive and time-consuming workflows for the user, which must be repeated for each explanation method used. Developing a tool that allows the user to execute multiple explanations on their model and then visualize these results in an integrated manner would greatly reduce the amount of manual work required.

The thesis aims to design a tool that enables the execution, visualization, and comparison of XAI methods for audio machine learning models. As part of this goal, the thesis describes the design of an audio-focused XAI framework as the proposed solution to the identified problem.

The framework provides an Application Programming Interface (API) in the form of a Python library and supporting tools to allows its users to integrate their own models into the system and run various XAI techniques. The framework's architecture is based on the Model-View-Presenter (MVP) pattern to provide modularity and ensure integration of additional models and alternative presentation form to the web interface. The framework integrates selected XAI methods, including Local Interpretable Model-agnostic Explanations (LIME), Integrated Gradients, and Layer-wise Relevance Propagation (LRP). The visualization layer is implemented as a web interface based on the JavaScript React library, which enables presentation of results from multiple explanation methods alongside audio playback and spectrogram visualization. The implementation of the developed audio XAI framework is available on GitHub under the link https://github.com/m-falkowski/pylibxai.

The scope of the thesis is not to perform any research about integrated machine learning models and explanation techniques. The thesis does not evaluate any of their metrics such as performance or accuracy, nor it compares them one to another.

The content of the thesis has been split into individual chapters. Chapter 2 discusses the fundamentals of Explainable artificial intelligence, provides the analysis for the selected

methodology for the project which is a framework architecture, and discusses the selection of XAI methods and models integrated into the project. Chapter 3 describes the functional and non-functional requirements for each component of the framework as well as software requirements required for the implementation of the project of the thesis. Chapter 4 provides software requirements and the user manual of the project together with an overview of the user interface. Chapter 5 describes the internal components of the project, describing in detail the implementation of each step of interpretation. Chapter 6 describes the testing and validation procedure of the project. Chapter 7 provides a summary of the results obtained through project development, draws conclusions about them, and outlines ideas for a future research about similar topics.

# 2. Problem analysis

This chapter provides an overview of terminology and methods used in the thesis, which describes essentials of Explainable Artificial Intelligence, description of XAI methods used in the thesis, and software engineering principles used for designing an Audio XAI framework.

## 2.1. Black-box and white-box AI systems

The AI (Artificial Intelligence) systems may be classified as *black-box* or *white-box* models based on their interpretability characteristic [4]. This classification is performed based on the transparency characteristic of the neural network, which is defined as a level of understanding of an internal behavior and decision-making process by a human observer [2].

White-box models have high-level transparency, and a human may understand their internal workings. The examples of such models include decision trees or linear regression models. In contrast, black-box models lack transparency, and their internals are opaque. The examples of black-box models are Convolutional Neural Networks (CNNs) or Deep Neural Networks in general. One of the methods to introspect the internals of black-box models is post-hoc explanation methods (2.2) that allow introspection after the model processing, as shown in Fig. 2.1.

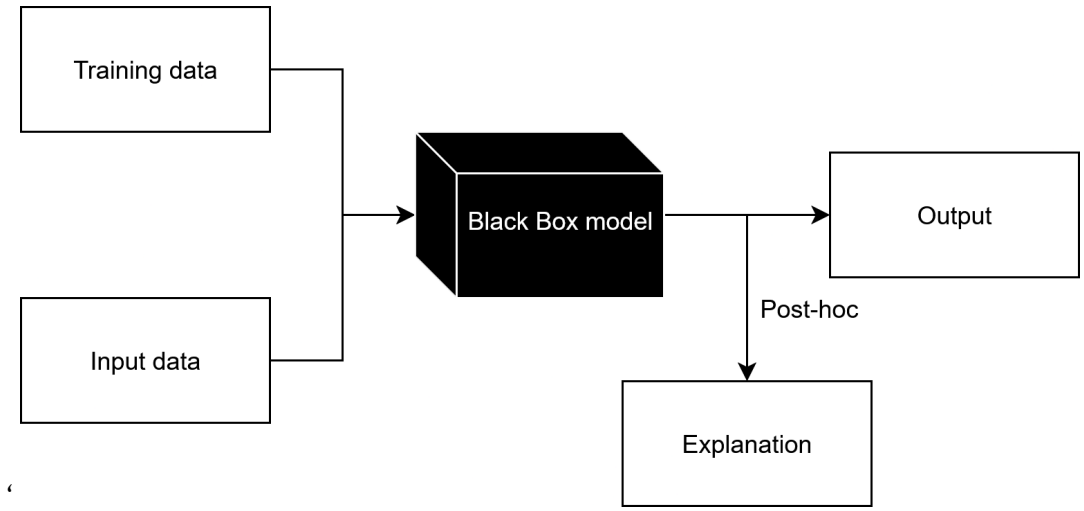


**Figure 2.1.** The diagram depicts a black-box model with post-hoc explanation techniques applied to its output.

## 2.2. Explainable artificial intelligence

Explainable Artificial Intelligence (XAI) is a collection of machine learning techniques that focuses on making the decision-making process of AI (Artificial Intelligence) systems understandable to humans [2]. XAI seeks to provide *explainability* of a model, which is a characteristic that describes any action or procedure that provides *interpretability* of its

internal decision-making process. The other XAI characteristic is *interpretability* of the model, which is the ability of the model to provide a result understandable by a human.

The main goals of Explainable artificial intelligence are to increase transparency, understandability, and trustworthiness of AI systems for humans. The other advantages of XAI are for developers of AI systems to assist them with debugging and improving their systems.

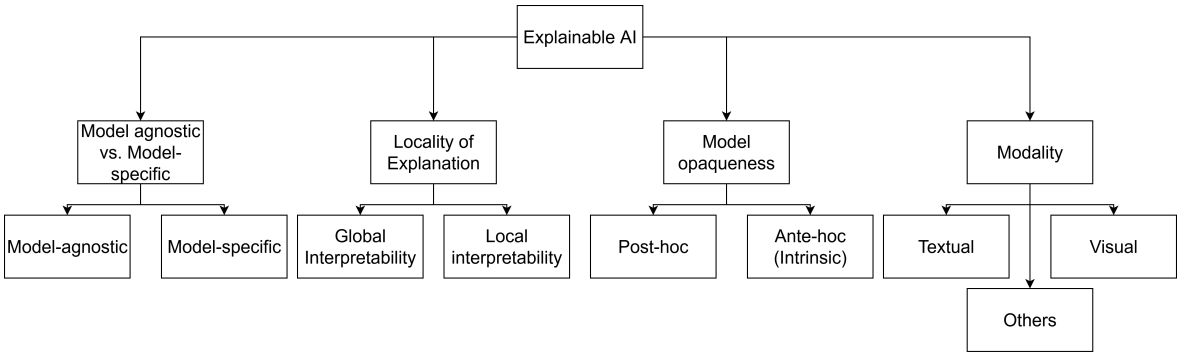


**Figure 2.2.** The taxonomy of the Explainable Artificial Intelligence [5].

The wide number of XAI techniques may be characterized into multiple categories forming a XAI taxonomy (Fig. 2.2), including:

- **Model-Specific vs. Model-Agnostic** - The model-agnostic methods can be applied to any model without taking into account its internal structure. On the other hand, model-specific techniques are tailored towards a particular type of model, for example, Convolutional Neural Networks,

- **Locality of Explanation** - Local interpretability aims at explaining a single prediction or instance to help understand why the model made a given decision. An example may be a LIME technique, which may show which audio feature or part influenced a given prediction. In contrast, global interpretability aims to explain the whole behavior of the model across the entire dataset,

- **Model opaqueness** - This criterion characterizes XAI techniques based on the transparency of models, which are black-box and white-box (section). White-box models are intrinsic, which means that they are naturally explainable, while black-box models have a complex internal structure, which means that only post-hoc methods can be used, i.e., those that explain the behavior of the model without explaining its internal implementation.

- **Modality** - It classifies the techniques based on the type or form of output in which the explanation is presented to the user. The exemplary types of modality are textual or visual.

## 2.3. Selection of explanation methods for framework

The thesis focuses on post-hoc explanation methods and provides a set of XAI methods integrated into the framework. The essential criteria for selecting these explanation methods were the interpretability of the produced explanation, simplicity of the user interface of a given XAI technique, whether it depends on a background data sample, and whether the method is model-specific or agnostic. The following XAI methods were considered as methods integrated into the framework:

- **Local Interpretable Model Explanation (LIME)** - The LIME is an XAI technique that provides local, model-agnostic explanations. The LIME implementation used for the project is *audioLIME*, which utilizes a source separation technique to provide *listenable* explanations that are interpretable for the user (Sec. 2.4),
- **Integrated Gradients** - Integrated Gradients is an attribution-based XAI technique that fulfills axiomatic requirements of sensitivity and implementation invariance, making it very reliable [6] (Sec. 2.6),
- **Layer-wise Relevance Propagation (LRP)** - The LRP is a gradient-based technique that with a similar interface to the integrated gradients which is an attribution map (Sec. 2.7),
- **SHapley Additive exPlanations (SHAP)** - The SHAP explanation method [7] was considered for the framework. It was rejected as it requires a background set of data samples, making it data-dependent [8], which complicates the implementation of that method for the project.

The XAI techniques that were selected for the method integrated into the framework are LIME, Integrated Gradients, and Layer-wise relevance propagation.

## 2.4. Local Interpretable Model Explanation

The Local Interpretable Model Explanation (LIME) is an Explainable AI technique that makes a post-hoc prediction of a given model locally and model-agnostic, by fitting a small model around the prediction [9]. The method perturbs interpretable components by creating a surrogate model $g \in G$ that finds the interpretable features around the input value to determine which interpretable features impact the model's decision.

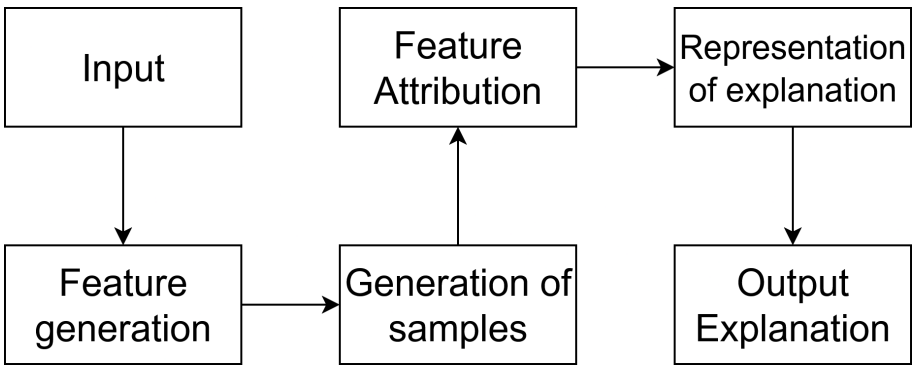


**Figure 2.3.** The diagram depicts the process of the LIME explanation [10].

For the whole LIME procedure (Fig. 2.3), the first step is generating samples where LIME creates a set of perturbed samples to locally approximate a given explained model $f$ for a given input $x \in X$. LIME defines a set of interpretable inputs $x'$, which are perturbed samples of the input and represented as a set of binary vectors $z$, and the mapping $x = h_x(x')$, which converts the vectors back to the original input space. Different mapping $h_x$ are used depending on the type of the input space, like input or text data. The LIME explanation is calculated by minimizing a loss function $\mathcal{L}(f, g, \pi_x)$:

$$\xi(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g). \tag{2.1}$$

where $\pi_x$ is defined as a proximity measure between a given input $x$ and a given binary vector $z$. The $\Omega$ is a complexity measure.

## 2.5. audioLIME

AudioLIME is an XAI technique based on the LIME framework that extends its definition to audio data. The method applies source separation to the input audio waveform, which separates sound sources such as voice or musical instruments from the original audio input [11]. The audioLIME method closely follows a pipeline of the LIME method but applies source separation to the input, as shown in Fig. 2.4.

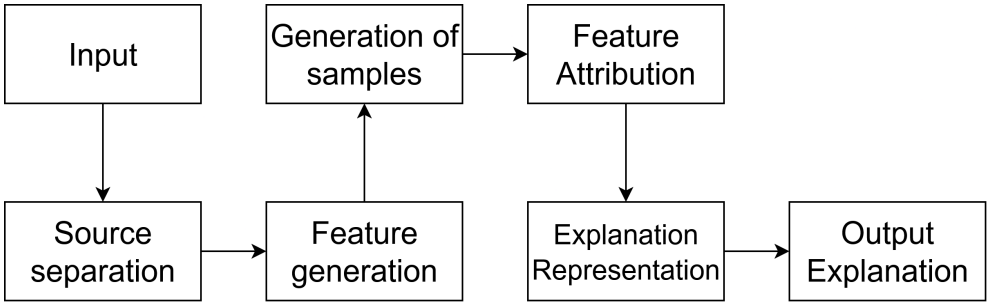


**Figure 2.4.** The diagram shows the pipeline of the audioLIME method.

For a given input, the method defines a set of interpretable features - audio segments in this scenario - and generates a set of binary vectors by performing perturbation. The surrogate model then trains on these samples originating from the decomposed input, split into multiple audio sources by segmentation, and then perturbed.

The method generates listenable explanations that provide interpretability to the user. Figure 2.5 shows an example of audioLIME's explanation, where the result contains an audio segment with the highest contribution to the model's prediction.

## 2.6. Integrated gradients

Integrated gradients (IG) is an attribution-based XAI technique that provides an explanation of the explained model by accumulating gradients from the baseline input $x'$ to the input $x$ along a path integral of the gradients between those two inputs [6]. The definition


**Figure 2.5.** The image shows the result of audioLIME's explanation.

is specified for a given input model $F : R^n \rightarrow [0,1]$, the input $x \in R^n$, and the baseline input $x' \in R^n$.

The integrated gradients in the $i^{th}$ dimension is defined for a gradient $\frac{\partial F(x)}{\partial x_i}$ as:

$$\mathrm{IG}_i(x) = (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F\left(x' + \alpha(x - x')\right)}{\partial x_i} d\alpha.$$

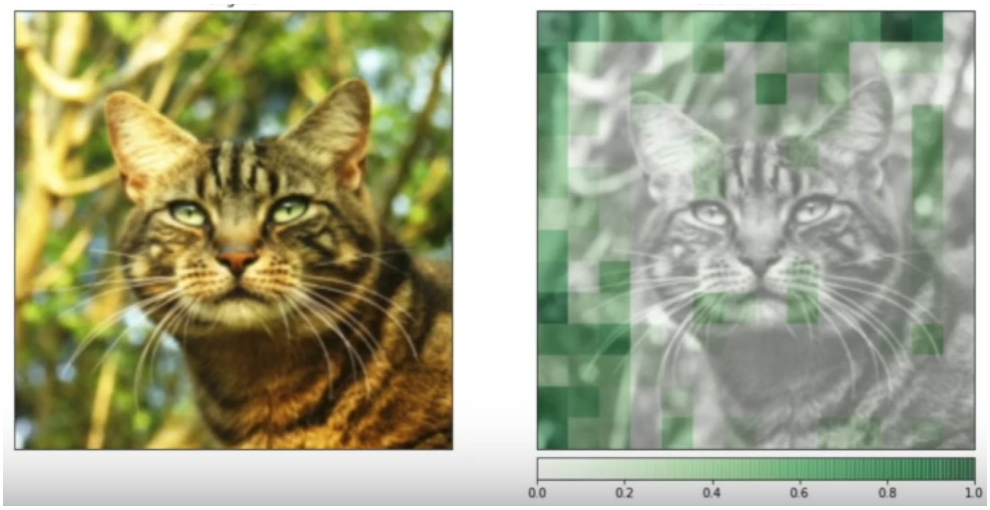


**Figure 2.6.** The image the attribution map of Integrated Gradient's explanation.

The output of the LRP method is an attribution map as shown in Fig. 2.6.

## 2.7. Layer-wise Relevance Propagation

Layer-wise Relevance Propagation (LRP) is a gradient-based XAI method that explains a given classifier's decision using decomposition and relies on a backward propagation mechanism that is applied to all the model's layers in the sequence [12].

A given prediction $f(x)$ is redistributed backwards using a given redistribution rule until a given relevance score $R_i$, like an image fragment or pixel, is calculated and assigned, ensuring that *relevance conservation* is preserved such that:

$$\sum_i R_i = \ldots = \sum_j R_j = \sum_k R_k = \ldots = f(\mathbf{x}) \tag{1}$$

Relevance conservation states that the total value of relevance is preserved at every step, such as every layer of the neural network, of the redistribution process. The relevance score $R_i$ of each input variable informs how strongly this variable contributed to the prediction.

The redistribution process from layer $l+1$ to layer $l$ can be defined for feed-forward neural networks as follows:

$$R_j = \sum_k \frac{x_j w_{jk}}{\sum_j x_j w_{jk} + \varepsilon} R_k.$$

for the relevance score $R_k$ at a layer $l+1$, the weight connection $w_{jk}$ between neuron $j$ and neuron $k$, and $x_j$ neuron activations at a layer $l$.

Different LRP redistribution rules, such as $\varepsilon$-rule or $\gamma$-rule, are defined for a given layer type. The output of the LRP method is an attribution map, similar to Integrated gradients, as shown in Fig. 2.6.

## 2.8. Sound waveform representation

The sound wave may be represented using a *waveform*, which is a graphical representation of a function of a sound in the time domain as depicted in Fig. 2.7 [13]. The digital representation of a sound wave is performed by sampling the analog waveform, usually at a regular period every $T$ seconds, which is a **sampling interval**. The measure that provides a value of average samples per second is the **sample rate** $f$, which is calculated as $f = \frac{1}{T}$.

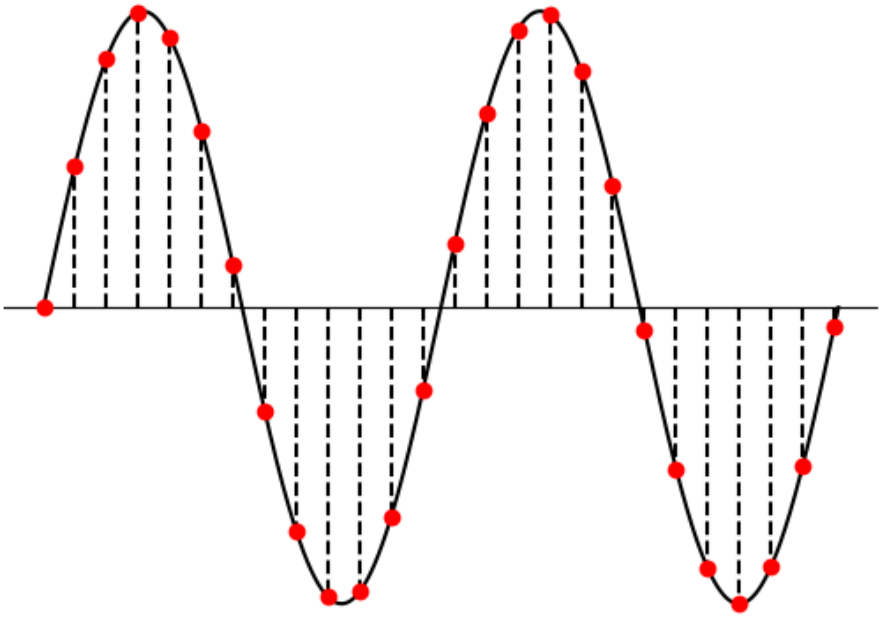

**Figure 2.7.** The process of sampling a sine audio waveform with constant interval. Sampled points are represented using red dots.

The sound may be represented in a graphical form in a time-frequency domain using a *spectrogram*. The spectrogram is a spectrum of frequencies of a sound at a given time point in the time domain, where the color at a given point is the amplitude of a given wave [14]. This representation of audio is important in Machine Learning as it allows

us to represent audio as image data and use different types of neural networks, such as Convolutional neural networks. Sound may be represented in the **Mel scale**, which is a non-linear, perceptual scale to better reflect the subjective reception of a sound level to a human rather than a standard sound scale presented in Hertz [15]. The **mel-spectrogram** is a spectrogram whose frequency scale was transformed into the mel scale. An example of a mel-spectrogram is shown in Fig. 2.8. The mel-spectrogram is frequently used in the domain of machine learning as its format in the form of an image accurately approximates how humans perceive sound [14].

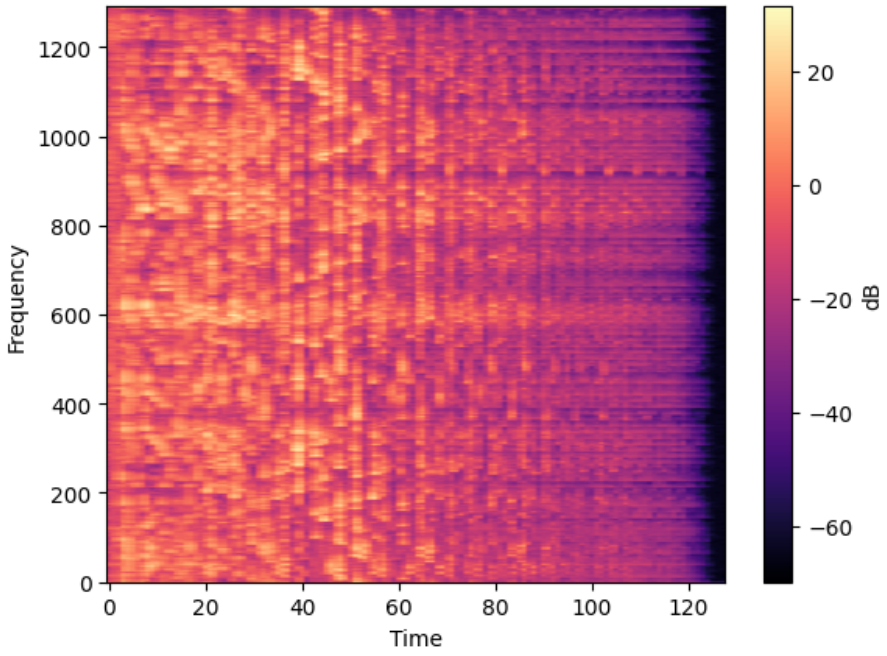


**Figure 2.8.** The picture shows an exemplary spectrogram.

## 2.9. Selection of ML model for integration

The audio XAI framework contains several machine learning models that were prematurely integrated into the framework as part of the thesis. The selection of these models focused on selecting state-of-the-art audio classifier models.

The models integrated into the framework as a part of the project include:

- **HarmonicCNN** - The HarmonicCNN model from the repository *State-of-the-art Music Tagging Models*, which is a collection of audio-tagging models [16]. The model is a novel approach to audio classification by using harmonic filters [17],
- **CNN14** - The state-of-the-art music-tagging CNN model from the *Paans* collection of models trained on the *AudioSet* dataset [18] that outperformed previous state-of-the-art Google's models in mean average precision (mAP) [19],
- **GtzanCNN** - The GtzanCNN is a handmade CNN model for the thesis based on the GTZAN dataset [20]. It was specifically designed to take a spectrogram directly as an input, in contrast to the other models, such as CNN14 and HarmonicCNN, which convert a waveform to the spectrogram in one of their layers. As later shown (Sec.

```python
class Controller:
    def __init__(self):
        self.model = MLModel()  # tightly coupled

class ControllerDecoupled:
    def __init__(self, predictor: PredictorInterface):
        self.model = predictor  # decoupled via abstraction
```

**Figure 2.9.** The example of the DIP principle implemented in the Python programming language.

6.1), this allowed the model to run the LRP method as all its layers were supported by the LRP implementation of the *captum* library.

## 2.10. Dependency Inversion Principle

Dependency Inversion Principle (DIP) is a software engineering method stemming from the SOLID principle that provides a methodology for decoupling software modules. The DIP principle states in its definition [21]:

**Definition 1.** *High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

The example of the Dependency Principle Inversion is depicted in Fig. 2.9. The first high-level class `Controller` directly uses a given machine learning model by creating its instance directly by calling `MLModel()` in its constructor. This creates a dependency of a high-level class on the implementation of the low-level one. The second class `ControllerDecoupled` has an intermediate interface `PredictorInterface`, which abstracts away the details of the low-level `MLModel` class and provides a clear interface for other low-level classes and the high-level class. This approach opens a high-level class for extension - any other class implementing the interface can be passed to the interface. The principle allows to make systems more maintainable, modular, and testable.

## 2.11. Software framework

A software framework is a software component that provides a generic, structured functionality for creating applications. It provides predefined components, tools, and API (Application Programming Interface) so that users may focus on implementing their programs [22]. It differs from a *library* by applying an Inversion of Control so that, rather than providing a certain functionality for a user to call, it provides a certain rigid program structure, calling the user's provided code at some points.

The software framework has key characteristics:

- **Inversion of Control** - A key characteristic where a framework establishes a predefined software structure and invokes user-provided components at specific points. This inversion of control distinguishes a framework from a library. This principle allows modularity, separation of concerns, and reduces boilerplate code,

- **Default structure** - A framework typically has a consistent architecture, and its core code is generally fixed. It may be structured using a specific design pattern like Model-View-Presenter to ensure modularity of the implementation. The framework may improve the organization of a code by providing a predefined structure,
- **Extensibility** - Allows for extension and customization of framework behavior. It may be offered through different solutions such as plugin systems, hooks, or abstract classes that developers can implement.

## 2.12. Model-View-Presenter

Model-View-Presenter (MVP) [23] is an architectural design pattern that is a variation of the MVC (Model-View-Controller) that aims to provide a separation of concerns between data processing (business logic) and user interface.

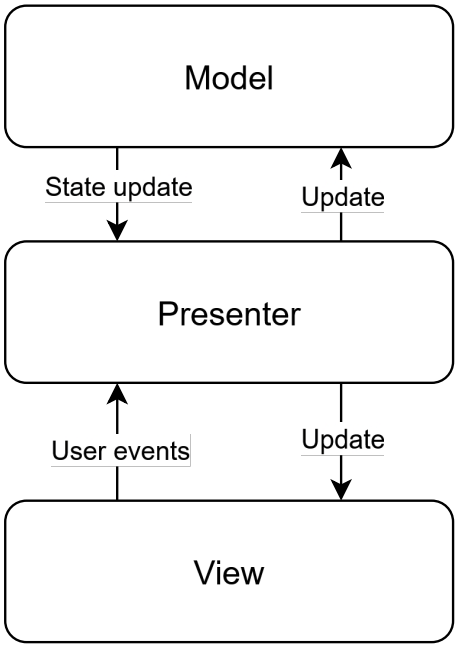


**Figure 2.10.** The diagram depicts Model-View-Presenter pattern organization.

The MVP pattern separates a program's logic into the three components, which are *Model*, *View*, and *Presenter*. The organization of the relation between components is shown in Fig. 2.10.

**Model** The model is responsible for information processing and generating a data to be displayed,

**View** It is a **passive** component that is responsible for presenting generated data results to the user. The key distinction of the MVP pattern to the classic MVC pattern is that the view does not communicate with a model in any way and rather it is immutable. The example of such a view could be a static web page,

**Presenter** The component that links the other two. Presenter retrieves model's results and formats them for the view. It manages user's input and commands and passes them to the model and view appropriately.

A key aspect identified when designing an Audio XAI framework was that data processing needs to be performed for multiple user-supplied models, and their results are subsequently visualized. This situation allowed adaptation of the MVP pattern to ensure modularity of the solution, bringing a necessary separation of concerns to the solution so that data processing and visualization are independent of each other, controlled through a presenter class. The Audio XAI framework implemented in the thesis generates a static web page which distinguish its MVP implementation from a typical one as view does not communicate completely with a presenter class.

## 2.13. Audio XAI Framework architecture

The proposed architecture for the topic of the thesis that allows generating explanations for ML models, along with visualizing them, is the *framework* architecture (2.11). This decision originates from the observation that a wide range of machine learning models are defined using various libraries and user APIs. To make a project usable, there must be a minimal amount of code created to adapt a given external model for use in the project. Consequently, the intended user of such a tool is a ML model designer or user, including programmers, data scientists, etc.

The other application architecture that was considered was a monolithic user application. This design was rejected in contrast to the framework as the project requires a minimal influence of the user to adapt their models, meaning that some form of inversion of control (Sec. 2.11) is required, which naturally suggests the framework architecture. The framework architecture is shown in Fig.2.11.

Furthermore, the framework is based on the MVP pattern (Sec. 2.12) to provide a modular and extensible implementation. This allows a framework to be extended with a new ML model, XAI technique, or presentation layer with no or minimal interaction with the rest of the implementation.

The framework is separated logically into the following parts, including:

**Library** Provides an Application programming interface for the user of the library that allows creation of an adapter for a user's model, setup of a given explanation method, its configuration including a selection of a given presentation layer, and utilities for processing and audio,

**Web interface** It is a presentation layer of the framework that allows visualization of the results of XAI techniques, the audio data including its playback and spectrogram visualization,

**Runner script** The runner script is an utility script provided by the framework to help the user to launch explanation's on libraries supported models (2.9) in a typical usage scenario where the input audio is read and preprocessed for a given model and then the explanation is launched.

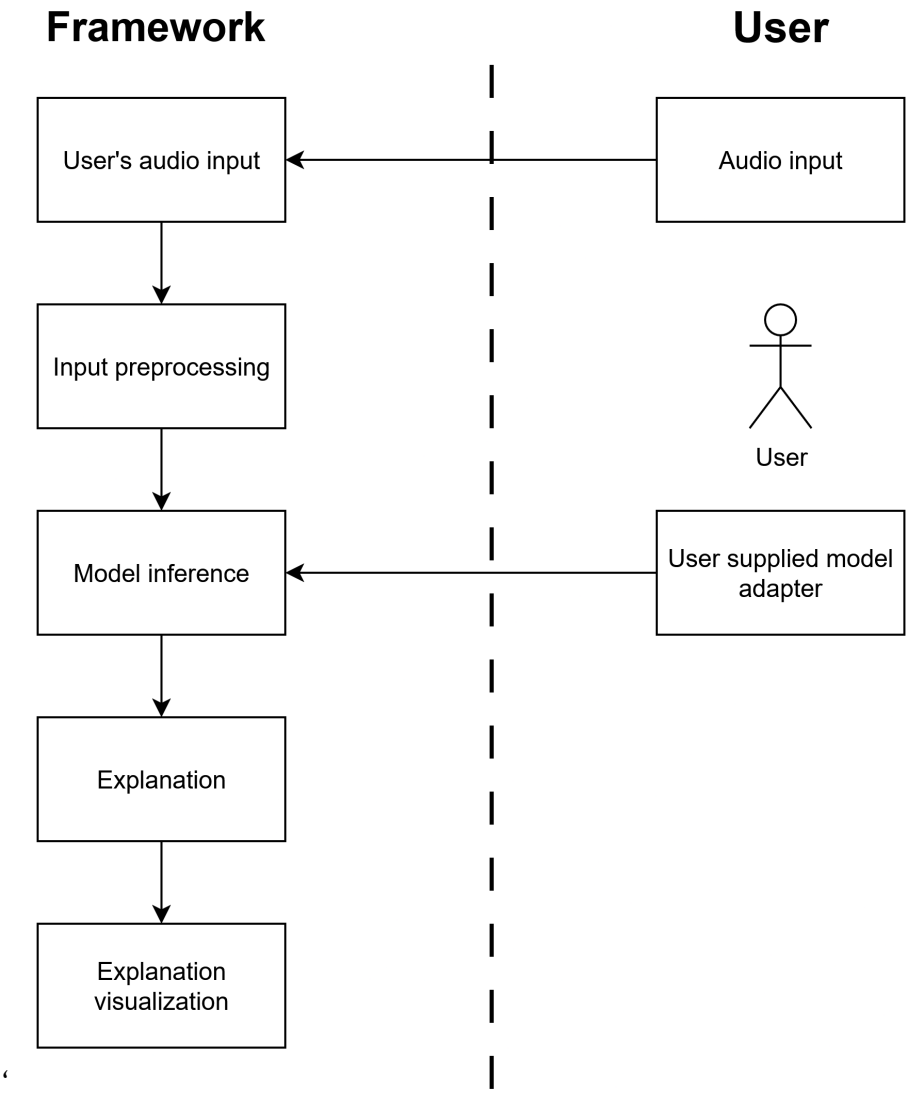


**Figure 2.11.** The Audio XAI framework architecture.

## 2.14. Similar solutions

The audio XAI framework designed and developed as part of the thesis is a unique attempt to provide an explanation framework that is tailored specifically towards the domain of audio machine learning. The framework is distinct from other XAI frameworks by providing direct support for audio features.

To the best of our knowledge, two similar XAI frameworks focused on a similar topic as the thesis, but as later explained, they do not provide direct support for audio, making them distinct from the thesis project:

- **explAIner** - The explAIner [8] is a visual analytics framework for interactive interaction with Explainable AI. It focuses on the whole XAI domain, providing support for a wide spectrum of XAI techniques supporting different data types, including text and image data. Though it is a very mature tool, it lacks a direct visual support for audio data, which distinguishes it from a project for a thesis,
- **XAI - An eXplainability toolbox for machine learning** - It is a toolbox [24] that provides collection of tools for working with Explainable AI. As with the previous example, it is distinct in that it lacks the visual support for audio data.

# 3. Requirements and tools

This chapter defines and formalizes requirements that the project must actualize to meet the objective of the thesis, including software and behavioral requirements. The chapter starts with a definition of functional and non-functional requirements, followed by an overview of software requirements and tools used in the project of the thesis.

## 3.1. Functional requirements

The functional requirements seek to define how a framework should behave for its user. For clarity, the requirements are divided based on the software component of the framework, which in the framework's case are the library, user web interface, and terminal helper tool.

### 3.1.1. Library application programming interface requirements

1. **Provide user model injection**
   The library API must enable users to create adapters for their classes, allowing them to inject operations on their data into the framework. Users may implement custom model adapters by extending the provided base classes. This enables integration of various audio model architectures with the explanation framework.

2. **Provide explanation interface for model adapters**
   The library API must offer base classes for implementing a given explanation method. Each adapter type must define specific methods required for the corresponding explanation algorithm, including LIME, Integrated gradients, or LRP, ensuring a consistent interface for different explanation methods.

3. **Explanation method Selection**
   The library API must offer users the ability to choose which explanation methods are displayed in the view. Users can select from available explainers, including LIME, Integrated gradients, and LRP, through the library API or through command-line arguments when using a helper tool. This allows users to focus on specific explanation approaches relevant to their use case.

4. **Support necessary audio format**
   The library API must support user audio input provided in `.wav` and `.mp4` formats. Audio should be automatically converted to appropriate formats for model processing.

5. **Provide interface for view classes**
   The library API must offer a base interface for implementing views that will fulfill the requirement of modularity and allow a system to support multiple view types.

6. **Context Management**

The library API must define a context that serves to store and save the current explanation progress. The context class manages working directories, file operations, and explanation state throughout the process. Context provides centralized access to resources and maintains consistency across different explanation phases.

7. **Interface for prediction class information**
   The library should offer an interface enabling users to inject prediction class information. This information is essential for proper explanation, interpretation, and visualization. The system should support label retrieval dynamically depending on whether the user provided support for their user-defined model adapters.

8. **Support for selected explanation methods**
   The library API must provide explanation methods selected for the thesis, such as LIME, Integrated Gradients, and LRP. These methods should be implemented as classes, and they also serve as controllers within the Model-View-Presenter pattern and coordinate between model adapters, explanation algorithms, and view components. They encapsulate the complete explanation workflow from input processing to result visualization.

9. **Support for web interface view**
   The library API must define web server view classes that manage file serving and a web-based user interface. Web server components handle HTTP requests and serve explanation results through a user interface.

10. **Ensure persistent result storage**
    The library API must ensure persistent storage of user results on disk to prevent data loss. The context system should provide this by saving explanation artifacts to the working directory. This will enable users to archive explanation outputs across multiple sessions.

### 3.1.2. User web interface requirements

1. **Separate tab for each explanation result**
   The user interface must aggregate the results of each explanation method in separate tabs to organize outcomes and enable switching between them. Each explanation method, which is LIME, Integrated gradients, or LRP, should get its dedicated tab. This should allow users to easily navigate different explanation approaches to compare results. The sidebar should provide a structured view of all generated explanations in a single interface and allow you to switch between them.

2. **Display original audio playback**
   The user interface must display the original audio track that serves as input for explanations across all explanation methods. The interface must enable audio playback

functionality, including play/pause controls and waveform visualization. Users can listen to the original audio while examining explanation results for better context understanding. The waveform display provides a visual representation of the audio signal structure.

3. **Visualize LIME explanation result**
   The user interface must display the LIME explanation result as playable audio. Users should be able to listen to specific audio segments that contributed most significantly to the model's prediction. This enables an intuitive understanding of which temporal parts of the audio influenced the model's decision. The width of the explained audio visualization should be aligned with the original audio.

4. **Provide version information**
   The user interface must display information about the library version so that users may verify which version of a library they are currently using. Such information may prove to be useful, for example, for troubleshooting.

5. **Display notification and error information**
   The user interface must display notifications about errors or other information in a dedicated information panel. The notification system provides user feedback for processing status, errors, warnings, and completion messages. Users receive clear communication about the system state and any issues that require attention. The dedicated panel ensures important messages are prominently displayed without disrupting the main workflow.

6. **Visualize attribution intensity**
   The user interface must show a plot of attribution intensity of the user's input audio for both Integrated gradients and LIME explanations. The attribution scores given to various audio input segments are displayed in these plots. This enables users to determine which audio signal regions had the biggest impact on the model's prediction.

7. **Display of user's input overlayed with attribution**
   The user interface must display attribution overlaid on the user's input for both Integrated gradients and LIME explanations. Attribution values are visualized as color heatmaps on the user's spectrogram, which enables a user to understand which frequency components at a given time interval influenced the model's decision. This method works very well when the input provided to the model is a spectrogram.

8. **Display prediction classes of user's model**
   The user interface must display optional model label descriptions when provided by the user's model adapter. The interpretability of explanation visualizations is improved by label information.

### 3.1.3. Terminal application requirements

The XAI framework must provide a terminal application tool for launching supported models on the framework's explanation method, with input provided by the user. The tool will then implement a typical usage scenario:

1. **Provide a selection of one or more explanation methods**
   The application must enable users to select one or more explanation methods offered by the framework, which are LIME, Integrated gradients, or LRP.

2. **Provide a selection of audio model**
   The application must enable users to select from models implemented within the framework that they intend to use.

3. **Provide a selection of visualization methods**
   The application must enable users to select the visualization method implemented in the framework. Users can choose between web-based visualization and disabled visualization.

4. **Provide a selection of working directory**
   The application must enable users to select a folder as the working directory or create a default when not provided by the user.

## 3.2. Non-functional requirements

The implementation of an audio XAI framework must meet several criteria to ensure proper behavior for the user and quality criteria. The most important criteria are modularity, usability, and persistence.

**Modularity**
The project thesis aims to provide a general and extensible audio explanation framework. The system must be designed with a modular structure, where key components such as explanation methods, model adapters, and view renderers are implemented as independent modules. For that, the system must adhere to the pattern principles of Model-View-Presenter architecture as discussed in Sec. 2.12. This modularity allows for the seamless integration of new functionalities, such as adding additional explanation algorithms or supporting new audio models, without requiring modifications to the core of the system.

**Reliability**
The system must behave stably and predictably during normal operation. During regular operation, the system has to display stable and predictable behavior. The user interface, audio processing pipelines, and explanation methods must all gracefully handle runtime errors and invalid inputs. Errors must be properly logged, and informative feedback must be sent to the user without crashing. To avoid data loss, intermediate and final outputs need to be saved consistently.

**Usability**

The framework library and web user interface must follow intuitive design principles, including consistent navigation, clear labeling, and visual clarity of explanation results. The use of separate tabs for different explanation methods ensures that users can focus on one explanation at a time while easily switching between views. Notifications and status indicators must be visible and should inform a user about the behavior of the interface. In addition, the command-line application helper tool must be a fast way of testing the library implementation and environment setup.

## 3.3. Tools specification

The developed Audio XAI Framework requires a set of tools and libraries, which were used for its building and development.

**PyTorch [25]**

PyTorch is a tensor library that is used for designing deep learning solutions. The library is used as a foundation of the framework for constructing ML models trained on the GPU. It is also used by most dependencies, especially integrated models (Sec. 2.9).

**NumPy [26]**

NumPy is a Python library for numerical computing. It efficiently implements multi-dimensional arrays and mathematical operations. In the thesis, it is used to handle numerical data, matrix operations, and preprocessing steps for audio and model inputs.

**Librosa [27]**

Librosa is a Python library for music and audio processing and analysis. It is used in the thesis for reading user's input audio, conversions into spectrogram and similar.

**torchaudio [28]**

TorchAudio is a Python library for music and audio processing and analysis. Similarly to Librosa, it is used in the thesis for reading user's input audio, conversion into spectrogram and similar.

**Pytest [29]**

Pytest is a Python testing framework. It is used in the thesis for unit testing.

**Matplotlib [30]**

Matplotlib is a Python library for data visualization. In the thesis, it is used for plotting spectrograms, training curves, and other graphs that illustrate experimental results.

**Captum [31]**

Captum is a model interpretability library for PyTorch. It provides methods such as saliency maps and attribution scores. In the thesis, it is used to analyze and explain neural network predictions on audio data.

**React [32]**

React is a JavaScript library used for developing component-based interfaces. In the thesis, it is used to implement the frontend of the application, providing interactive views and controls for audio and model outputs.

**Vite [33]**

Vite is a modern frontend build tool and development server. In the thesis, it is used for bundling the React application and enabling fast development with hot-reload support.

**Bootstrap [34]**

Bootstrap is a CSS framework providing pre-styled components and layout utilities. In the thesis, it is used to simplify styling and ensure a responsive frontend design.

**Wavesurfer [35]**

Wavesurfer is a JavaScript library for audio visualization and interaction. In the thesis, it is used to render waveform displays, allow audio playback, and enable user interactions with audio segments.

**ChartJs [36]**

ChartJs is a JavaScript charting library for visualizing data with charts and plots. In the thesis, it is used in the frontend to present experiment results and performance metrics.

# 4. External specification

The following chapter provides the external interface of the audio XAI framework implemented as part of the thesis. It discusses the project's software requirements, installation procedure, user interface, and framework usage.

## 4.1. Software requirements

The audio XAI framework is implemented mainly in the Python programming language and requires a set of tools from Python's ecosystem for correct setup. The project's software prerequisites are:

- **Python** - version 3.9.20
- **Conda** - version 24.11.0
- **curl** - version 8.11.0
- **GNU bash** - version 5.2.37

## 4.2. Installation procedure

The framework provides an installation script `setup.sh` in the project's root directory that performs a complete installation of all software dependencies, listed in Appendix 1. To install the project the users must:

- Clone the git repository of the audio XAI framework:

```
git clone https://github.com/m-falkowski/pylibxai
```

- Enter the cloned directory:

```
cd pylibxai/
```

- Initialize dependent repositories stored as git's submodules in the repository:

```
git submodule update --init --recursive
```

- Create a *conda* environment where all dependencies will be installed and activate it. The environment can be activated after the installation manually at any given:

```
conda create -y --name pylibxai_env -c conda-forge python=3.9.20 &&
↪  conda activate pylibxai_env
```

- Launch the install script that is present in the root directory:

```
chmod +x setup.sh && sudo ./setup.sh
```

## 4.3. Framework application programming interface

The most fundamental way of using the framework is through its API interface. The framework provides its API interface as a Python library that may be used by the user. The exemplary usage of the API interface is shown in Fig. 4.1, where the user imports

```
Type "help", "copyright", "credits" or "license" for more information.
>>> from pylibxai.model_adapters import HarmonicCNN
>>> from pylibxai.pylibxai_context import PylibxaiContext
>>> from pylibxai.Interfaces import ViewType
>>> from pylibxai.Explainers import LimeExplainer
>>> ctx = PylibxaiContext("/root/")
>>> ctx
<pylibxai.pylibxai_context.pylibxai_context.PylibxaiContext object at
    ↪ 0x00000140174F6A90>
>>> adapter = HarmonicCNN(device="cpu")
>>> explainer = LimeExplainer(adapter, ctx, view_type=ViewType.WEBVIEW)
>>> explainer
<pylibxai.Explainers.lime_explainer.LimeExplainer object at
    ↪ 0x00000140351E7FD0>
>>>
```

**Figure 4.1.** The exemplary usage of the framework API interface in Python's interactive mode.

in Python's interactive mode a set of classes of the framework to create a context class, instantiate an HarmonicCNN model adapter and create a `LimeExplainer` class used to launch the LIME method.

### 4.4. Command-line interface

The library provides a script, `pylibxai_explain.py` (Sec. 5.11) that allows executing the user's input audio on models and XAI methods already implemented in the framework. The interface of the script is shown in its help message, which is visible in Fig. 4.2. This command-line script contains the following options:

- **-u, –visualize** - The boolean flag that enables visualization in the web interface. If not specified, the terminal-based debug view will be used,
- **-e EXPLAINER, –explainer EXPLAINER** - The parameter is used to specify the XAI method to be used,
- **-t TARGET, –target TARGET** - Name or index of the expected class that will be returned by the prediction,
- **-i INPUT, –input INPUT** - The parameter is used for providing a path to the input file.
- **-w WORKDIR, –workdir WORKDIR** - The parameter is used for providing a path to the input file,
- **-p PORT, –port PORT** - Specifies the port number of the web interface if it is enabled. It must be used along with `-visualize` parameter,
- **-d DEVICE, –device DEVICE** - The device type on which the explanation will be launched. It is either *cpu* or *cuda*.

The exemplary usage of the explanation runner helper script is shown in Fig. 4.3.

### 4.5. Web interface

The main presentation layer of the Audio XAI framework is provided in the form of a web user interface. The web interface provides the following components to the user:

```
$ pylibxai_explain.py -h
usage: pylibxai_explain.py [-h] -m MODEL [-u] -e EXPLAINER -t TARGET -i INPUT
 ↪  -w WORKDIR [-p PORT] [-d DEVICE]

Process a model name and input path.

optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL
                        Name of the model to use [{sota_music, paans,
                         ↪  gtzan},]...
  -u, --visualize       Enable visualization of audio in browser-based UI.
  -e EXPLAINER, --explainer EXPLAINER
                        Name of the explainer to use [lime,
                         ↪  integrated-gradients, lrp].
  -t TARGET, --target TARGET
                        Name or index of the label to explain. Mapping is
                         ↪  done automatically based on the model if the
                         ↪  model provides it.
  -i INPUT, --input INPUT
                        Path to the input file or directory.
  -w WORKDIR, --workdir WORKDIR
                        Path to the workdir directory.
  -p PORT, --port PORT  Port to use for the web server.
  -d DEVICE, --device DEVICE
                        Device to use for computation [cpu, cuda]. Default is
                         ↪  'cuda' if available, otherwise 'cpu'.
```

**Figure 4.2.** The output of the help message of the command-line explanation runner.

```
pylibxai_explain.py -w ./shap_expl/ -m gtzan
 ↪  --explainer=integrated-gradients,lrp -i
 ↪  datasets/GTZAN/Data/genres_original/jazz/jazz.00050.wav
```

**Figure 4.3.** The exemplary invocation of explanation runner that runs Integrated Gradients and LRP XAI methods on GtzanCNN model.

- **The main page with XAI method's results** - The main page contains the results of a given XAI method. In case of the LIME method, there is the original audio playback provided as the result, followed by the playback of the explanation as shown in Fig. 4.4. Both are provided as a listenable result. In the case of the Integrated gradients method and LRP both methods contain the same version of the result page, as these two methods return an attribution map of a given input audio, as shown in Fig. 4.5,

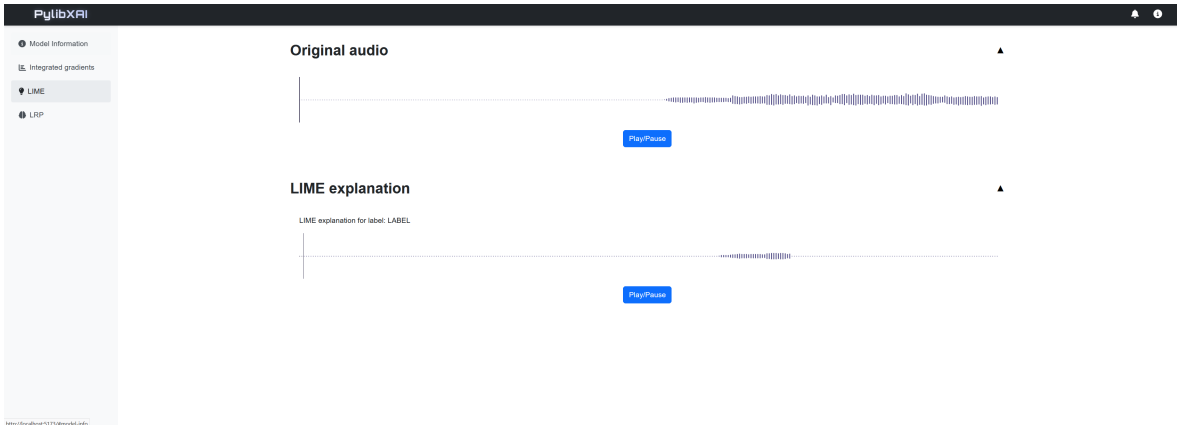


**Figure 4.4.** The picture shows the whole web interface with a main page that contains the LIME results.

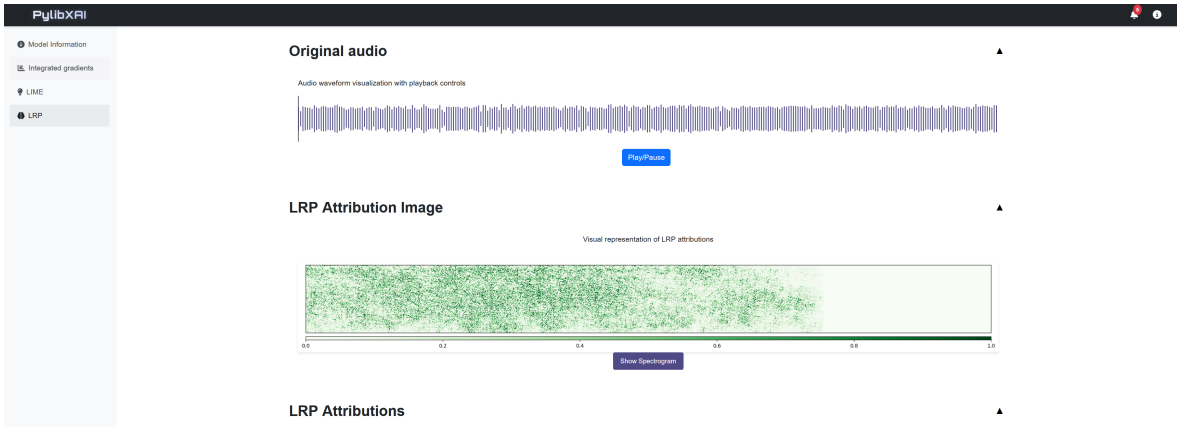


**Figure 4.5.** The picture shows the whole web interface with a main page that contains the LRP results.

- **Sidebar** - The sidebar allows switching between different subpages, defined as visible in Fig. 4.6,
- **Attribution map** - The attribution map is the XAI result of the attribution-based methods, Integrated Gradients, and LRP. For a given input, the given image of the attribution is visualized in the results page using *Chart.js*, as shown for the spectrogram model in Fig. 4.7. This section also contains the original raw input without the attribution applied, as shown in Fig. 4.8, and a button to toggle between these two representations.

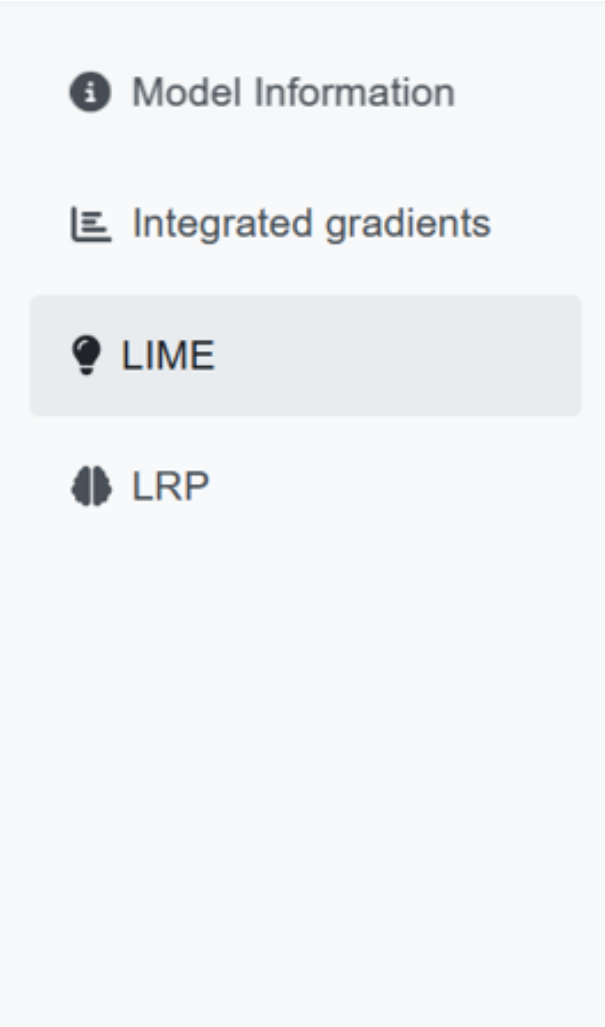


**Figure 4.6.** The picture depicts the sidebar component of the web page.

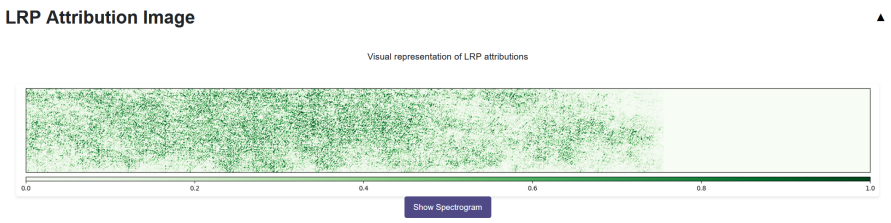


**Figure 4.7.** The picture shows the attribution map overlayed on the given input.

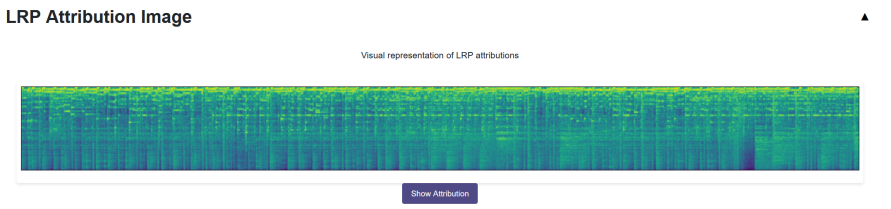


**Figure 4.8.** The picture shows the raw input without attribution overlayed.

- **Attribution intensity plot** - The attribution intensity plot visualizes the attribution over time, summing the attribution provided at a given point in time, as shown in Fig. 4.9.

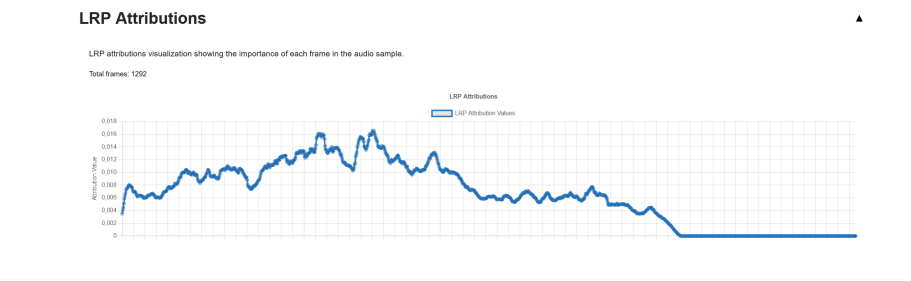


**Figure 4.9.** The picture shows the plot of attribution intensity over time.

- **Version information panel** - The version information panel displays information about software versioning, as shown in Fig. 4.10.

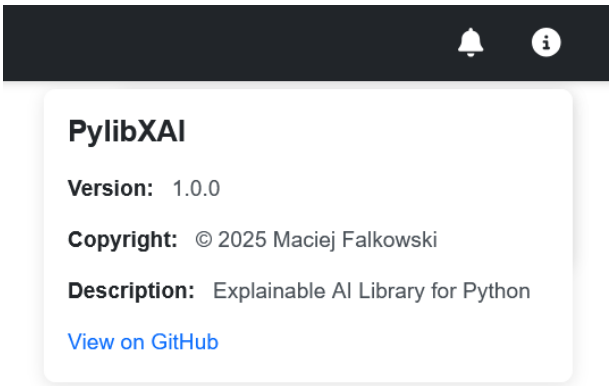


**Figure 4.10.** The image shows the version information panel of the web page.

- **Notification panel** - The notification panel contains any notifications about errors o warnings that might have happened during the execution of the web page, as shown in Fig. 4.11.
- **Model information page** - The model information page contains supplementary information about a user's model, as shown in Fig. 4.12.

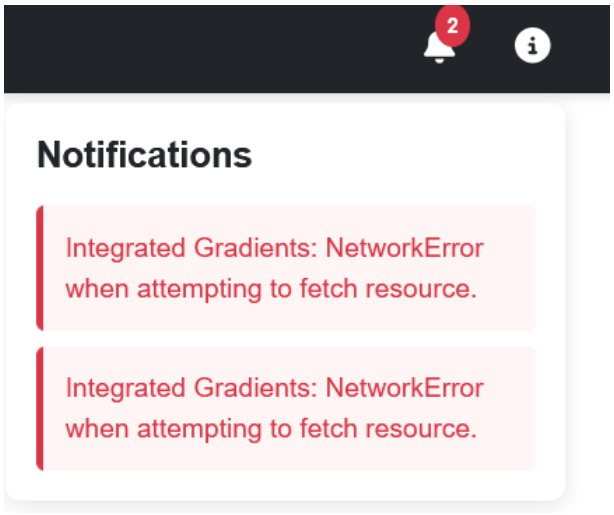


**Figure 4.11.** The image shows the notification information panel of the web page.

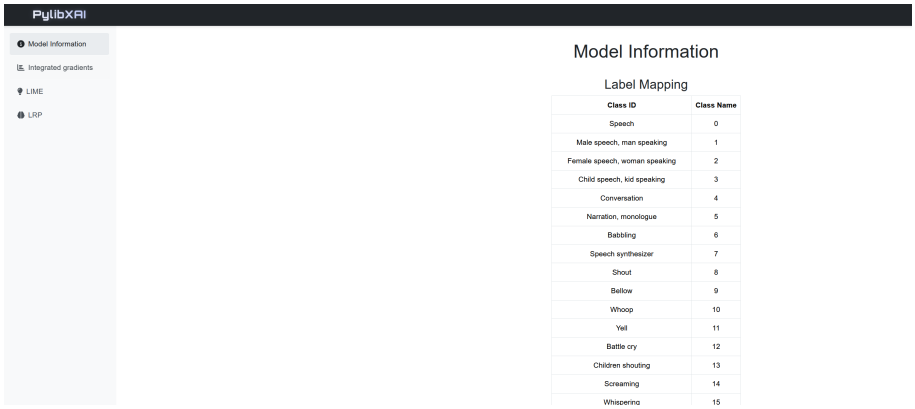


**Figure 4.12.** The image shows the model information page.

# 5. Framework implementation

The following chapter contains an overview of the architecture and implementation details of the explanation framework. It briefly discusses each of the system's components by providing a walk-through of each step of the framework's explanation pipeline. The implementation assigns a name *Pylibxai* that refers to the implementation of the audio XAI framework which is the goal of the thesis.

## 5.1. Software architecture overview

The architecture of the project consists of three main components that are library, web interface, and command-line helper script. These component implement all steps of a full-pipeline from preprocessing user's input up to visualizing explanation results as discussed in Sec. 2.13.

1. **Library** - Provides the main logic and implementation. It is interfacing directly with a user and provides the necessary tools for the execution of an explanation. This includes interfaces for the creation of adapters for users' models, logic implementing supported explanation types, visualization of explanations that include a frontend web interface, and helper utilities.
2. **Web interface** - Defines a main visual view for the observation of the explanations' results. It is composed of two parts which the first is a backend server that serves the explanation results to the frontend part through the HTTP protocol. The frontend part then requests the required data from the backend server and visualizes it in the form of a browser page. The communication process between user application and framework is shown in Fig. 5.1.
3. **Command-line explanation runner** - It is a toolchain's helper script and serves the purpose of quickly launching a typical usage type of the library, which is loading an input audio file and then launching a chosen set of explanations on the chosen library's provided model.

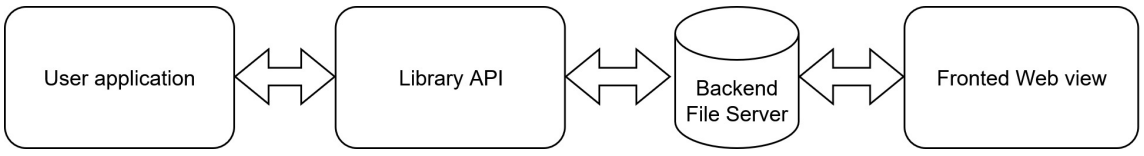


**Figure 5.1.** Communication scheme between user application and frontend.

## 5.2. Project structure

The project's source code is grouped in multiple directories grouping similar implementations. They contain an implementation of each of the framework's pipeline phases and unit test definitions for them.

- **audioLIME/** - Contains implementation of audioLIME explanation method,

- **AudioLoader/** - Provides utilities for loading and preprocessing audio files,
- **Explainers/** - Contains explainer classes for LIME, Integrated gradients, and LRP methods,
- **pylibxai_explain.py** - Main command-line entry point for running explanations on audio models,
- **Interfaces/** - Defines abstract base classes and interfaces for model adapters and views,
- **model_adapters/** - Implements adapter classes for integrating different audio models with the framework,
- **models/** - Contains pretrained model files and checkpoints for supported audio models,
- **pylibxai_context/** - Implements context management for storing explanation state and working directories,
- **Views/** - Contains implementations of different views, such as web view for serving explanation results through HTTP to the web page, or debug view for quick debugging,
- **pylibxai-ui/** - Frontend React application providing web-based user interface for visualization,

### 5.3. Model-View-Presenter implementation overview

The thesis utilizes Model-View-Presenter (MVP) architectural pattern (Sec. 2.12) to obtain extendable architecture that satisfies one of core requirements of the thesis that is modularity as discussed in Sec. 3.2. This allows a framework to be easily extendable with new explanation types, machine learning models, and views separating these three from each other so that the developer might work on one of these exclusively.

The Model-View-Presenter is implemented using the following classes visible in the class diagram 5.2. The implementation and details of these classes are discussed later in the chapter:

**Model** The function of the model in the framework is realized using adapter classes for user models, such as `HarmonicCNN` or `Cnn14` (Sec. 2.9). These classes operate on user data passed to them from controller classes, performing data manipulation, which is an inference in this case. They need to be implemented by the user to provide their

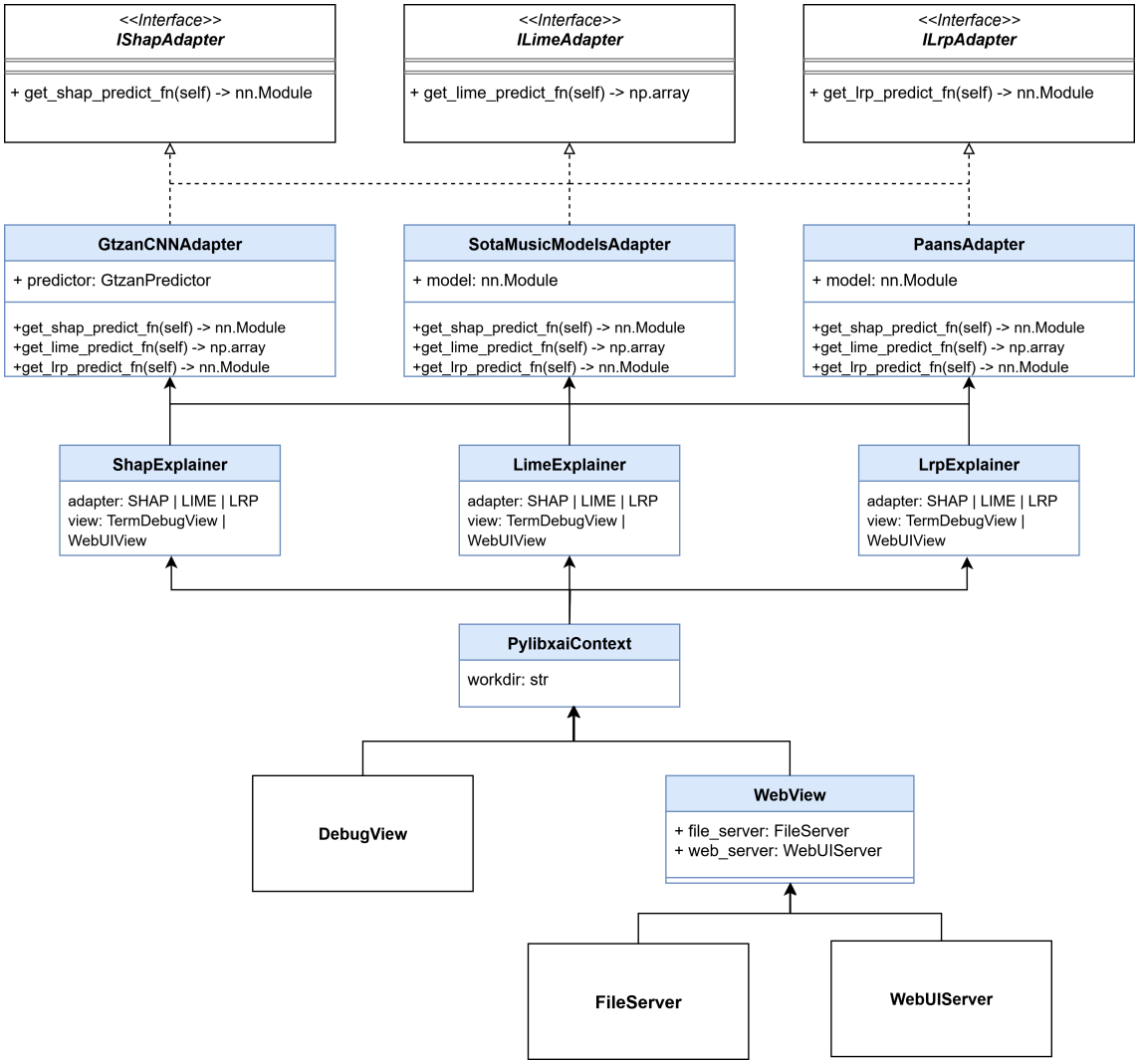


**Figure 5.2.** Library class diagram.

```python
from abc import ABC, abstractmethod
import numpy as np

class LimeAdapter(ABC):
    """Abstract base class for LIME (Local Interpretable
       Model-agnostic Explanations) adapters"""
    @abstractmethod
    def get_lime_predict_fn(self) -> Callable[[np.ndarray], np.ndarray]: pass
    """Returns a function that takes an audio input and
       returns the model's prediction for that input."""
```

**Figure 5.3.** Class definition of `LimeExplainer`.

support to the framework and they usually come in a form of an adapter class. The interface required for implementation of these classes is discussed in Sec. 5.4,

**View** The function of view in the framework is realized using classes such as `WebView` or `DebugView`. These classes receive explanation results specified in the `Context` class as required by the `ViewInterface` class and process them to display the results to the user,

**Controller** The purpose of this part is to pass the user's input into the model classes for data manipulation and to later pass the results of data manipulation into the view. This functionality is achieved by classes implementing explainers' logic, such as `LimeExplainer` or `IGradientsExplainer`.

### 5.4. Interfaces for model adapters

Model adapters require a set of interfaces that specify the required behavior that a user must fulfill to make their model work properly within the framework. For that reason, the library provides an interface for required explanation methods to perform a given explanation that specifies behavior between a given explainer class and the user's model adapter. This provides enough granularity to the interface so that a user may freely decide which explanation method they would like to provide support for. Interfaces for model adapters are implemented with the use of Python's `ABC` module [37], which allows defining abstract base classes for other classes. Every interface method is annotated `@abstractmethod`, requiring a user to define it when performing inheritance, or it throws a `TypeError` exception otherwise.

#### 5.4.1. LIME method adapter

Adapter for LIME method (2.5) defines the interface that the user's model needs to satisfy to be able to launch `audioLIME` explanation on it. This interface requires from user defining a single method `get_lime_predict_fn(self)` that must return a `np.array`.

#### 5.4.2. Integrated gradients method adapter

In the same way, adapter for Integrated gradients method (2.6) defines the interface that the user's model needs to satisfy to be able to launch Integrated gradients method on it.

```python
class IGradientsAdapter(ABC):
    """Abstract base class for Integrated gradients method adapters"""
    @abstractmethod
    def get_igrad_predict_fn(self) -> Callable[[torch.Tensor], torch.Tensor]:
        pass
    """Returns a function that takes an audio input and returns the model's
    ↪ prediction for that input."""

    @abstractmethod
    def igrad_prepare_inference_input(self, x: torch.Tensor) -> torch.Tensor:
        pass
    """Returns a function that takes an audio input and returns the model's
    ↪ prediction for that input."""
```

**Figure 5.4.** Class definition of `IGradientsAdapter`.

This interface requires from user defining a single method `get_igrad_predict_fn(self)` that must return an `np.array`.

Additionally, the `igrad_prepare_inference_input()` method is defined by the Integrated gradients explainer, which serves the purpose of additionally preprocessing data for a given model, as shown in 5.4. This is required as interface for explainers accept user's waveform input but they do expect an input which is the same as model's input which may be either waveform or a spectrogram which requires additional conversion.

### 5.4.3. LRP method adapter

The last method adapter for LRP method (2.7) also requires the user to define a single method `get_lime_predict_fn(self)` that must return `nn.Module`. This return type is required as Captum's LRP implementation takes it as an input.

**Example 1.** *An exemplary implementation of LrpAdapter for an exemplary user's adapter `MyModelAdapter` is shown in 5.5.*

### 5.4.4. ModelLabelProvider interface

The framework offers an additional interface for model adapters to pass their labels into the system. The function requires the model to return a dictionary that maps a given class name to its corresponding identifier, as shown in Fig. 5.7.

## 5.5. Model adapters

The framework requires users to provide their ML (machine learning) models in an adapter class and includes a set of interfaces that define how these adapters should be implemented (Sec. 5.4). These user-supplied classes adjust model definition and wrap around logic responsible for model inference to allow them to be used within the framework.

Adapters of users' ML models serve as model classes in a Model-View-Presenter, where they are responsible for data processing. They are used by controller classes, which are explainer classes in the case of the framework that pass the user's input to them. This

```python
from pylibxai.Interfaces import LrpAdapter

class MyModelAdapter(LrpAdapter):
    def __init__(self, model, device):
        self.model = model
        self.device = device

    def get_lrp_predict_fn(self):
        class MyModelWrapper(torch.nn.Module):
            def __init__(self, model):
                super(MyModelWrapper, self).__init__()
                self.predictor = predictor
                self.device = device

            def forward(self, x):
                self.predictor.model.eval()
                return self.predictor.model(x)

        return MyModelWrapper(self.predictor, self.device)
```

**Figure 5.5.** A sample implementation of `LrpAdapter` for the exemplary `MyModelAdapter` class.

```python
class LrpAdapter(ABC):
    """Abstract base class for LRP (Layer-wise Relevance Propagation)
       adapters"""
    @abstractmethod
    def get_lrp_predict_fn(self) -> nn.Module: pass
    """Returns a function that takes an audio input and
       returns the model's prediction for that input."""
```

**Figure 5.6.** Class definition of `LrpAdapter`.

```python
class ModelLabelProvider(ABC):
    """Abstract base class for providing labels for model predictions"""
    @abstractmethod
    def get_label_mapping(self) -> Dict[int, str]: pass
    """Returns a mapping from label IDs to label names."""

    @abstractmethod
    def map_target_to_id(self, target: str) -> int: pass
    """Maps a target label to its corresponding ID."""
```

**Figure 5.7.** Class definition of `ModelLabelProvider`.

```
class HarmonicCNN(LimeAdapter, IGradientsAdapter, ModelLabelProvider):
    def __init__(self, device='cuda'):
        # ...
        path_models = os.path.join(path_sota, 'models')

        # ...

        config = Namespace()
        config.dataset = "jamendo"  # we use the model trained on MSD
        config.model_type = "hcnn"
        config.model_load_path = os.path.join(path_models, config.dataset,
        ↪  config.model_type, 'best_model.pth')
        config.input_length = 5 * 16000
        config.batch_size = 1  # we analyze one chunk of the audio
        self.model = Predict.get_model(config)

        self.model_state = torch.load(config.model_load_path,
        ↪  map_location=self.device)
        self.model.cuda()
        self.config = config
    # ...
```

**Figure 5.8.** Model initialization in adapter `HarmonicCNN`.

way, the data manipulation is separated from the rest of the framework, and new models may be integrated without modifying the rest of the codebase, which provides modularity. The user may select individual interfaces that the framework's API provides, which allows them to implement only a subset of the functionality that a framework offers and to skip unneeded parts. The user then may freely decide which interface they would like to implement for their model between `LrpAdapter`, `LimeAdapter`, `IGradientsAdapter`, and `ModelLabelProvider`.

There are three adapters integrated into the library, which are HarmonicCNN, Cnn14, and GtzanCNN (Sec. 2.9). As their implementation is similar and long, HarmonicCNN is the model that will described with a precision in the rest of this chapter.

The given adapter first inherits the required interfaces to overload the required methods. This is shown in Fig. 5.8, where the HarmonicCNN class inherits three classes: `LimeAdapter`, `IGradientsAdapter`, and `ModelLabelProvider`:

The class initialization (Fig. 5.8) involves importing the model's class definition from the *SotaMusicModels* repository [16]. As the repository hosts a collection of models and offers the retrieval of a given model through filling `config = Namespace()` with configuration parameters, such as the model name specified as a `"hcnn"` string. The HCNN (HermeticCNN) model is then retrieved `self.model = Predict.get_model(config)`. In the last step, the model state is then loaded using `torch.load()` method.

To implement `ModelLabelProvider` interface, the class creates a mapping of model's class labels stored in global variable `TAGS` and later uses them to overload a method `get_label_mapping()` as shown in Fig. 5.9.

The adapter defines a method `get_igrad_predict_fn()` that implements Integrated

```
TAGS = ['genre---downtempo', 'genre---ambient', ...]

class HarmonicCNN(LimeAdapter, IGradientsAdapter, ModelLabelProvider):
    def __init__(self, device='cuda'):
        # ...
        self.label_to_id = {i: v for i, v in enumerate(TAGS)}
        self.id_to_label = {v: i for i, v in enumerate(TAGS)}
        # ...
    def get_label_mapping(self):
        """Returns the label mapping for the model."""
        return self.label_to_id
    # ...
```

**Figure 5.9.** Set up of label classes in adapter `HarmonicCNN`.

```
def get_igrad_predict_fn(self) -> Callable[[torch.Tensor], torch.Tensor]:
    # ...
    def predict_fn(x):
        # ...
        output_dict = self.model(x)
        output_tensor = output_dict
        return output_tensor
    return predict_fn
```

**Figure 5.10.** Supporting Integrated gradients method in adapter `HarmonicCNN`.

gradient's interface as shown in Fig. 5.10. This method must return a function with a type `Callable[[torch.Tensor], torch.Tensor]`, that takes Pytorch `torch.Tensor` both as an argument and return type. The callback then defines an inner function `predict_fn(x)` that contains the standard inference code of HarmonicCNN, and then it returns that function.

To implement `LrpAdapter` the adapter provides a definition for LRP callback as shown in Fig. 5.11. This method must return a function object that must contain the instance of Python's `torch.nn.Module` that a user must create a wrapper class around its model's implementation. The function defines a `HarmonicCNNWrapper` that inherits from a Python's type `torch.nn.Module` and provides a required `forward()` method where a HarmonicCNN inference is placed.

The last implementation is a support for the LIME method in Fig. 5.12. This method must return a function with a type `Callable[[np.ndarray], np.ndarray]` to fulfill the implementation of framework's `LimeExplainer` which is based on *audioLIME* implementation mthat operates on numpy's array types. The function then must take a numpy `np.ndarray` as an input and also return it.

## 5.6. Explainer classes

The explainer classes are the core component of the framework that is responsible for launching the explanation methods on the user input and the selected model. Their pri-

```
def get_lrp_predict_fn(self):
    class HarmonicCNNWrapper(torch.nn.Module):
        def __init__(self, model, device, model_state):
            super(SotaNNWrapper, self).__init__()
            self.model_state = model_state
            self.model = model
            self.device = device
        def forward(self, x):
            #...

            output_dict = self.model(x)
            output_tensor = output_dict
            return output_tensor
    return HarmonicCNNWrapper(self.model, self.device, self.model_state)
```

**Figure 5.11.** Support of LRP method in adapter `HarmonicCNN`.

```
def get_lime_predict_fn(self) -> Callable[[np.ndarray], np.ndarray]:
    # ...

    def predict_fn(x_array):
        # convert numpy array into tensor
        audio = torch.zeros(len(x_array), self.config.input_length)
        for i in range(len(x_array)):
            audio[i] = torch.Tensor(x_array[i]).unsqueeze(0)
        audio = audio.cuda()
        audio = Variable(audio)
        output_dict = self.model(audio)
        output_tensor = output_dict.detach().cpu().numpy()
        return np.array(output_tensor)

    return predict_fn
```

**Figure 5.12.** Support of LIME method in adapter `HarmonicCNN`.

mary purpose is to manage the explanation performed on the user's model and manipulate the resulting data based on the user's decision. They offload from the user the need to handle data between a model and an explanation, they postprocess explanation results to prepare them for visualization, and lastly, they launch the selected presentation layer, initializing it with explanation results.

These classes serve a purpose as presenter as part of the Model-View-Presenter pattern, though this name is not taken directly in their class names. As presenters, they manage the model and view classes directly, steering the flow of data between them.

Explanation methods are implemented as separate classes, including `LimeExplainer`, `IGradientsExplainer`, and `LrpAdapter`.

### 5.6.1. Integrated gradients and LRP explainers

The explainer implementations for the Integrated gradients and the LRP are identical. This is due to both of them being attribution-based methods (Sec. 2.6), which results

in their implementation in *Captum* library to have the same interface. As a result, the Integrated gradient's explainer implementation will be explained, and the LRP is defined similarly.

Both implementations' constructors initialize the classes with a user-provided model adapter, context class, and a type of view. The instance of a view class is created and managed internally, as shown in Fig. 5.13.

```
class IGradientsExplainer:
    def __init__(self, model_adapter, context, device, view_type=None,
     ↪ port=9000):
        # ...
        self.model_adapter = model_adapter
        predict_fn = model_adapter.get_igrad_predict_fn()
        self.explainer = IntegratedGradients(predict_fn)
        # ...
        self.view_type = view_type
        if view_type == ViewType.WEBVIEW:
            self.view = WebView(context, port=port)
        elif view_type == ViewType.DEBUG:
            self.view = DebugView(context)
        elif view_type == ViewType.NONE:
            self.view = None
        else:
            raise ValueError(f"Invalid view type: {view_type}. Must be one of
             ↪ WEBVIEW, DEBUG, or NONE.")
```

**Figure 5.13.** The constructor definition of the `IGradientsExplainer` class.

The `explain()` method is the top-level function used to launch the explanation process. It receives input audio and the input target as parameters and launches a set of operations, which include generating a spectrogram of the audio input and producing an attribution map as shown in Fig. 5.14. It also launches the specified view class using `self.view.start()`, which was chosen by the user during initialization.

The internal `explain_instance()` method executes the user's provided input and passes it to the explainer, which performs the explanation as shown in Fig. 5.15 The value returned from it is the attribution generated by the explanation.

The method `explain_instance_visualize()` is similar to the *explain_instance*, but as a result, it generates an image of the attribution that is overlayed on the input as shown in Fig. 5.16. The image is generated using *Matplotlib* (Sec. 3.3) library. To prevent the interactive window of the generated image, a call to the function `plt.ioff()` disables the interactive mode of Matplotlib.

The method `get_smoothed_attribution()` of the explainer is the helper function that performs a moving average over the generated attribution, which calculates the average of a given group of elements. In the case of the implementation, it is `window_size=15`, using a sliding window, as shown in Fig. 5.17.

```
class IGradientsExplainer:
    def explain(self, audio, target):
        # ...
        fig, _ = self.explain_instance_visualize(audio, target=target,
        ↪ type="original_image")
        self.context.write_plt_image(fig, os.path.join("igrad",
        ↪ "igrad_spectogram.png"))
        # ...
        attribution = self.get_smoothed_attribution()
        self.context.write_attribution(attribution, os.path.join("igrad",
        ↪ "igrad_attributions.json"))

        if self.view_type == ViewType.WEBVIEW:
            self.view.start()
        # ...
```

**Figure 5.14.** The top-level explain function of the `IGradientsExplainer` class.

```
class IGradientsExplainer:
    def explain_instance(self, audio, target, background=None):
        audio = self.model_adapter.igrad_prepare_inference_input(audio)
        attributions, delta = self.explainer.attribute(audio, target=target,
        ↪ return_convergence_delta=True)
        return attributions, delta
```

**Figure 5.15.** The implementation of `explain_instance` method of the `IGradientsExplainer` class.

```
class IGradientsExplainer:
    def explain_instance_visualize(self, audio, target, type=None,
    ↪ background=None, attr_sign='positive'):
        audio = self.model_adapter.igrad_prepare_inference_input(audio)
        attributions, delta = self.explainer.attribute(audio, target=target,
        ↪ return_convergence_delta=True)
        # ...
        plt.ioff()
        return viz.visualize_image_attr(attributions,
                                        audio,
                                        type,
                                        attr_sign,
                                        fig_size=(24,16),
                                        show_colorbar=True,
                                        outlier_perc=50)
```

**Figure 5.16.** The implementation of `explain_instance_visualize` method of the `IGradientsExplainer` class.

### 5.6.2. LIME explainer

The LIME method implementation is different from the previously described Integrated gradients and LIME methods. It is based on the *audioLIME* method (Sec. 2.5) and generates

```
class IGradientsExplainer:
    def get_smoothed_attribution(self):
        def moving_average(data, window_size=15):
            return np.convolve(data, np.ones(window_size)/window_size,
             ↪  mode='same')

        attribution = self.attribution.squeeze()
        positive_attribution = torch.clamp(attribution, min=0.0)
        summed_attribution =
         ↪  positive_attribution.sum(dim=0).detach().cpu().numpy()   # Shape:
         ↪  [1292]
        smoothed_attribution = moving_average(summed_attribution)
        return smoothed_attribution
```

**Figure 5.17.** The `get_smoothed_attribution` method of the `IGradientsExplainer` class.

an explanation in the form of a listenable audio fragment with the most relevant part of the audio.

The initialization of the class is similar to the previous explainer classes, where the explainer class is initialized with a user-provided model adapter, context class, and a type of view, as shown in Fig. 5.18.

```
class LimeExplainer:
    def __init__(self, adapter, context, view_type, port=9000):
        # ...
        self.adapter = adapter
        self.context = context
        self.view_type = view_type
        if view_type == ViewType.WEBVIEW:
            self.view = WebView(context, port=port)
        elif view_type == ViewType.DEBUG:
            self.view = DebugView(context)
        elif view_type == ViewType.NONE:
            self.view = None
        else:
            raise ValueError(f"Invalid view type: {view_type}. Must be one of
             ↪  WEBVIEW, DEBUG, or NONE.")
```

**Figure 5.18.** The initialization process of the `LimeExplainer` class.

The `LimeExplainer` contains only a top-level `explain()` method, which contains the whole implementation of the explainer in itself. The function starts with the creation of the `SpleeterFactorization` class, which performs factorization of the audio as explained in Sec. 2.5, and then creates an instance of the explanation object `LimeAudioExplainer` from the audioLIME's implementation, as shown in Fig. 5.19.

The LIME explanation instance is created by extracting the user's adapter using the `adapter.get_lime_predict_fn()` adapter handler as shown in Fig. 5.20.

The LIME explanation is performed on the selected number of listenable components of the `adapter.get_lime_predict_fn()` adapter handler, as shown in Fig. 5.21.

```
class LimeExplainer:
    def explain(self, audio, target=None):
        audio_loader = RawAudioLoader(audio)
        spleeter_factorization =
            SpleeterFactorization(audio_loader,
                n_temporal_segments=10,
                composition_fn=None,
                model_name='spleeter:5stems'
            )
        print('Creating explanation object')
        explainer = lime_audio.LimeAudioExplainer(verbose=True,
        ↪  absolute_feature_sort=False)
```

**Figure 5.19.** The top-level function `explain` of the `LimeExplainer` class.

```
class LimeExplainer:
    def explain(self, audio, target=None):
            # ...
            explanation =
            explainer.explain_instance(factorization=spleeter_factorization,
                predict_fn=self.adapter.get_lime_predict_fn(),
                top_labels=1,
                num_samples=16384,
                batch_size=16
            )
            # ...
```

**Figure 5.20.** The audioLIME explanation instance defined in `explain()` method of the `LimeExplainer` class.

```
class LimeExplainer:
    def explain(self, audio, target=None):
        # ...
        label = list(explanation.local_exp.keys())[0]
        top_components, component_indices =
            explanation.get_sorted_components(label,
                positive_components=True,
                negative_components=False,
                num_components=3,
                return_indeces=True)
        # ...
```

**Figure 5.21.** The audioLIME explanation execution in the `LimeExplainer` class.

After the LIME explanation result is generated, the explainer class saves the results into a given context and launches the given view class, as shown in Fig. 5.22.

## 5.7. Context class definition

The `PylibxaiContext` class manages the data and encapsulates the data saving process. It is an important bridge between model and view classes, formalizing how explanation

```
class LimeExplainer:
    def explain(self, audio, target=None):
        # ...
        self.context.write_audio(audio, os.path.join("lime", "original.wav"))
        self.context.write_audio(sum(top_components), os.path.join("lime",
        ↪  f"lime_explanation.wav"), 16000, 'PCM_24')

        if self.view_type == ViewType.WEBVIEW:
            self.view.start()
```

**Figure 5.22.** The end section of the `explain` function of the `LimeExplainer`, which saves the results to a given context class and lanuches a given view class.

```
def write_attribution(self, smoothed_attribution, suffix):
    path = os.path.join(self.workdir, suffix)
    with open(path, 'w') as f:
        json.dump({
            "attributions": smoothed_attribution.tolist(),
        }, f, indent=4)
```

**Figure 5.23.** The implementation of `write_attribution` method of `PylibxaiContext` class.

results are saved and accessed through a framework. The class also implements the requirement *Context Management* and *Ensure persistent result storage* as described in Sec. (3.2), fully encapsulating the process of saving the explanation results into the persistent storage. For the simplicity of design and also the purpose of serving explanation result to the web frontend as discussed later in 5.10, the implementation of this class saves all explanation results as files in a tree structure in a given working directory.

**Example 2.** *An exemplary implementation of the `write_attribution` method that is defined in the `PylibxaiContext` class. The method saves an attribution of the explanation to a file in `JSON` format as shown in Fig. 5.23.*

```
class PylibxaiContext:
    def __init__(self, workdir):
        self.workdir = workdir
        # ...

    def write_plt_image(self, fig, suffix): pass # ...

    def write_attribution(self, smoothed_attribution, suffix): pass # ...

    def write_label_mapping(self, labels, suffix): pass # ...

    def write_audio(self, audio, suffix, *args, **kwargs): pass # ...
```

**Figure 5.24.** Class definition of `PylibxaiContext`.

The `PylibxaiContext` class contains the following fields, as shown in its definition in Fig. 5.24:

- `self.workdir` - the field holds a path to the working directory in which explanation results will be saved,
- `write_plt_image(self, fig, suffix)` - it handles saving the results of Matplotlib's (Sec. 3.3) generated image,
- `write_attribution(self, smoothed_attribution, suffix)` - it handles saving explanation's attribution,
- `write_label_mapping(self, labels, suffix)` - it handles saving the mapping of labels into their indexes,
- `write_audio(self, audio, suffix, *args, **kwargs)` - it handles saving of an audio file,

### 5.8. View classes

The purpose of view classes is to present explanation results to the user in a readable form according to Model-View-Presenter pattern. The framework is implemented with modularity as a goal and allows multiple views defined implementing `ViewInterface` interface. The framework is implemented with modularity as a goal and allows multiple views defined by implementing `ViewInterface` interface that defines how explanation classes should be implemented in the framework. This interface `ViewInterface` requires a user to define the necessary methods required to initialize, start, and stop the given view as shown in Fig. 5.26.

- `__init__(self, context)` - A given view is initialized with a provided context class. The view then may utilize the accumulated results of explanation in that class and present them to the user,
- `start(self)` - The start method is responsible for launching the given view. In the case of `WebView`, it starts the backend and frontend servers. For `DebugView` it generates a terminal debugging logs showing based on explanation results,
- `stop(self)` - The stop method is responsible for stopping and cleaning the execution of a given view class. It is required though it logic may be empty when there is nothing to stop as in case of `DebugView` class.

Every view class defined in a framework must have an entry in enumeration `ViewType`. For `WebView` there is an enumeration `DEBUG` provided, for `DebugView` it is `DEBUG` as shown in Fig. 5.26.

#### 5.8.1. Web view

The `WebView` implementation provides the main visualization of the explanation results in the form of a web page. Its implementation only starts the underlying backend and frontend servers in separate processes using Python's subprocess module. The class implements the View interface to be later used within a framework. The definition of `WebView` is visible in Fig. 5.27.

```python
class ViewInterface(ABC):
    """Abstract base class for View Interface
    """
    @abstractmethod
    def __init__(self, context):
        self.context = context

    @abstractmethod
    def start(self) -> None: pass

    @abstractmethod
    def stop(self) -> None: pass
```

**Figure 5.25.** Class definition of `ViewInterface`.

```python
class ViewType(IntEnum):
    """Enum-like class for View Types"""
    WEBVIEW = 1
    DEBUG = 2
    NONE = 2
```

**Figure 5.26.** Class definition of `ViewType`.

The whole startup process of the WebView contained in the `setup()` method contains the following steps:

1. The startup procedure starts with the creation of a backend server using the function `run_file_server(self.context.workdir, self.port)`. It takes the path of the working directory from a given instance of the `Context` class and start a file serving server using Python's `socketserver.TCPServer` class. The implementation of file server is discussed later in Sec. 5.9,
2. In the next step, the frontend server is initialized with the use of the Vite [33] tool, which it is based on. The `subprocess.Popen` argument takes the Vite command startup arguments as `['npm', 'run', 'dev']` that is run in the current working directory where the implementation of the frontend source code is placed, which is `cwd=self.vite_dir`. Additional parameters for the frontend web server are passed through the environmental variables, such as a port number that is passed to the frontend through the `'VITE_PYLIBXAI_STATIC_PORT'` environmental variable.

### 5.8.2. Debug View

The purpose of the`DebugView` class is to enable quick debugging of the explanation framework, displaying information about results to a terminal without launching any external windows or software. It also shows an alternative view implementation to the `WebView` that shows the modularity of the explanation framework.

The implementation iterates over the content of the context class and displays its content in a structured manner to the user using Python's `print()` function, as shown in

```python
class WebView(ViewInterface):
    def __init__(self, context, port=9000):
        super().__init__(context, port)
        self.vite_dir = get_install_path() / "pylibxai" / "pylibxai-ui"
        self.server = None
        self.vite_process = None

    def start(self):
        # Start the file server
        self.server = run_file_server(self.context.workdir, self.port)
        # Start the Vite UI
        env = os.environ.copy()
        env['VITE_PYLIBXAI_STATIC_PORT'] = str(self.port)
        print(f"Vite UI directory: {self.vite_dir}")
        self.vite_process = subprocess.Popen(
            ['npm', 'run', 'dev'],
            cwd=self.vite_dir,
            env=env,
            stdout=sys.stdout,
            stderr=sys.stderr,
            shell=True
        )
        print(f"Vite UI launched at http://localhost:{self.port}/ (UI dev
        ↪  server running)")

    def stop(self):
        if self.vite_process:
            self.vite_process.terminate()
            self.vite_process.wait()
            print("Vite UI process terminated.")
        if self.server:
            self.server.shutdown()
            print("File server stopped.")
```

**Figure 5.27.** Class definition of `WebView`.

```python
class DebugView(ViewInterface):
    def __init__(self, context):
        super().__init__(context)
        self.context = context

    def start(self):
        print("=== DEBUG VIEW: PylibxaiContext Content ===")
        print(f"Working directory: {self.context.workdir}")
        print()

        # Display directory structure
        print("Directory structure:")
        self._print_directory_tree(self.context.workdir)
        print()

        # Display content of each subdirectory
        subdirs = ["integrated-gradients", "lrp", "lime"]
        for subdir in subdirs:
            subdir_path = os.path.join(self.context.workdir, subdir)
            if os.path.exists(subdir_path):
                print(f"=== {subdir.upper()} Directory Content ===")
                self._display_directory_content(subdir_path)
                print()

        # Display any JSON files in the root directory
        print("=== Root Directory Files ===")
        self._display_directory_content(self.context.workdir, root_only=True)
        print()

        print("=== DEBUG VIEW: Content Display Complete ===")

    def stop(self):
        pass
```

**Figure 5.28.** Class definition of `DebugView`.

the class implementation in Fig. 5.28. It does not contain any cleanup functionality, hence its `stop()` method has an empty body.

### 5.9. Backend Server implementation

The backend server implementation is implemented using Python's built-in `TCPServer` class from `socketserver` module. It allows serving files placed in a given directory provided as an input. This allows serving explanation results that are contained in an instance of `PylibxaiContext` that are passed to the view class from the framework.

Initially, the system path is changed into the working directory as shown in Fig. 5.29, so that the server is launched serving the files inside, creating endpoints for each stored resource. The advantage of such an implementation is automatic scalability with the content of a working directory, which provides simplicity of the implementation. The server is launched in a separate thread using the `threading.Thread` module over a given port provided as an input `socketserver.TCPServer(("", port), handler)`.

```python
def run_file_server(directory, port=9000):
    """Start a file server in a background thread."""
    os.chdir(directory)

    handler = CORSHTTPRequestHandler
    httpd = socketserver.TCPServer(("", port), handler)

    print(f"Serving files from {directory} at http://localhost:{port}/")

    # Run in background thread
    thread = threading.Thread(target=httpd.serve_forever, daemon=True)
    thread.start()

    return httpd  # To stop later with httpd.shutdown()

```

**Figure 5.29.** Implementation of backend server.

```python
class CORSHTTPRequestHandler(http.server.SimpleHTTPRequestHandler):
    def end_headers(self):
        # Add CORS headers
        self.send_header('Access-Control-Allow-Origin', '*')
        self.send_header('Access-Control-Allow-Methods', 'GET, OPTIONS')
        self.send_header('Access-Control-Allow-Headers', 'X-Requested-With')


        self.send_header('Access-Control-Allow-Headers', 'Content-Type')
        return super().end_headers()

    def do_OPTIONS(self):
        # Handle preflight requests
        self.send_response(200)
        self.end_headers()
```

**Figure 5.30.** Class definition of `CORSHTTPRequestHandler`.

Additionally, the server sets up a minimal CORS access control rules by defining a custom class `CORSHTTPRequestHandler` as shown in Fig 5.30.

**Example 3.** *An exemplary explanation result which is explanation's attribution saved as a file `workdir/lrp/attribution.json` will have an HTTP endpoint generated by the backend server of `/lrp/attribution.json`.*

## 5.10. Web frontend implementation

The main visualization of explanation results is created in the form of a web page displaying different results of an explanation, such as audio waveform visualization, explanation's attribution, or spectrograms graphically in a browser. The frontend implementation is built using the React JavaScript library [32] and Bootstrap CSS framework [34] and follows a

component-based architecture where each interface element is encapsulated as a reusable React component

The main objective of web page design is to offer a separate display of each explanation method to not confusing the results with each other, as discussed in Sec. 3.3. The Web page's layout is organized to emphasize the separation of different explanation methods to allow a user to quickly switch between them. For that, the web page organizes different data displayed in separate sections and provides a sidebar to switch between them.

The organization of the website's layout is shown in Fig. 5.31 and includes:

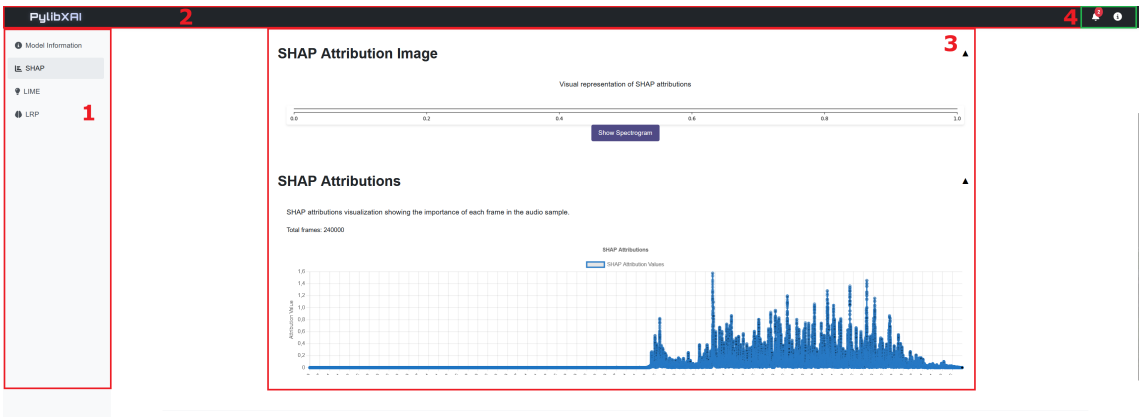


**Figure 5.31.** The organization of website. Each components is identified by a number.

- **Sidebar (1)** - Sidebar allows users to navigate between different sections, including mode information, Integrated gradients, LIME, and LRP explanations, using a list of buttons, highlighting the actively shown tab,
- **Navigation bar (2)** - The Navigation bar is a permanently visible page element that is sticky to its upper side. It contains various control icons on its left side, including a notification bell icon or a help icon,
- **Notification panel and help (3)** - these utilities elements are provided to a user to show them helper information such as framework's versioning or any errors and notifications that were encountered through the process of explanation visualization. The exemplary content of notification panel is discussed in the Sec.4.5.
- **Main content area (4)** - Main content area display currently selected section where the data displayed is organized vertically in a folded sections, allowing a user to hide unwanted information. Displayed information include model information, audio playback, attribution visualization, or explanation results.

### 5.10.1. Frontend component overview

The implementation consists of multiple components, each defined in a separate file, including:

- **App** - Main application component that manages global state, including managing the notification queue or the currently selected page to render,

- **Navbar** - Fixed navigation bar displaying supporting components, including notification bell, or version details,

- **ContentPage** - Manages rendering of the main content area and routing between different result sections of the interface,

- **ModelInfo** - Component that displays model information, such a label mappings, presenting their names along with a class identifier,

- **Lime, Igradients, Lrp** - Each of these components manages the display of its respective explanation method's result. This includes displaying audio playback, attribution visualizations, or interactive charts,

- **Footer** - Provides footer of the web page.

Frontend retrieves explanation results data from the backend server that was set up by the `WebView` class to serve collected data through the process of explanation. Both servers are set up by the view class and operate on localhost. Frontend retrieves the backend server's port from environmental variables using the `meta.env` module:

```
import.meta.env.VITE_PYLIBXAI_STATIC_PORT
```

Based on that, we define a URL of the backend server from which all frontends' requests will be made, which is stored in a variable:

```
staticBaseUrl = `http://localhost:\${staticPort}`.
```

The requests are done using JavaScript's built-in Fetch API utilizing `fetch()` method. An example of data fetching from backend which is fetching of explanation's attribution contained in a file in a JSON format as seen in Fig. 5.32.

An instance of an HTTP request is generated using the `fetch()` method. After that, the data is processed further by the frontend to create a visualization of it

```
const response = await
↪  fetch(`\${staticBaseUrl}/igrad/igrad_attributions.json`)
```

### 5.10.2. Visualization of audio waveform

The frontend provides visualization and audio playback functionality of different audio pieces that appear through the process of explanation:

- Original input - it is visualized so that a user access and listen to the original input,
- Lime method explanation - the LIME method provides a result in a form of an audio where an explanation which is the part of input that was the most important for the decision of explanation is preserved in the audio and the rest is muted.

The visualization and playback of the audio pieces is implemented using the Wavesurfer.js library [35]. It is used in the `Lime`, `Igradients`, and `Lrp` components to visualize the input

```
function Igradients() {
  // [...]

  // Get the static file server port from environment variables
  const staticPort = import.meta.env.VITE_PYLIBXAI_STATIC_PORT || '9000'
  const staticBaseUrl = `http://localhost:${staticPort}`

  // [...]

  useEffect(() => {
    // Fetch Integrated gradients attributions data
    const fetchAttributions = async () => {
      try {
        setIsLoading(true)
        const response = await
        ↪  fetch(`${staticBaseUrl}/igrad/igrad_attributions.json`)
        if (!response.ok) {
          throw new Error(`HTTP error! Status: ${response.status}`)
        }
        const data = await response.json()
        if (!data.attributions || !Array.isArray(data.attributions)) {
          throw new Error('Data is not in the expected format (object with
          ↪  attributions array)')
        }
        setAttributions(data.attributions)
        setIsLoading(false)
      } catch (error) {
        setIsLoading(false)
        pushNotification({ type: 'error', message: `Integrated gradients:
        ↪  ${error.message}` })
      }
    }
    fetchAttributions()
  }, [staticBaseUrl, pushNotification])
}
```

**Figure 5.32.** Data fetching logic defined in a `Igradients` component.

```
function Igradients() {
  // [...]

  wavesurfer.current = WaveSurfer.create({
    container: waveformRef.current,
    waveColor: '#4F4A85',
    progressColor: '#383351',
    cursorColor: '#383351',
    barWidth: 2,
    barRadius: 3,
    responsive: true,
    height: 100,
    barGap: 3
  })

  // [...]
}
```

**Figure 5.33.** Audio playback definition in a `Igradients` component.

audio, and the LIME method component also uses it for the visualization of its explanation. The advantage of the Wavesurfer.js library is that it provides interactive playback functionality meaning that a user may listen to the original audio and select any moment of the audio to listen to.

The exemplary initialization of WaveSurfer's audio playback in the `Igradients` component is shown in Fig. 5.33. The use is analogous for the `Lime` and `Lrp` components. An audio playback defined in this way will generate the following view as shown Fig. 5.34.

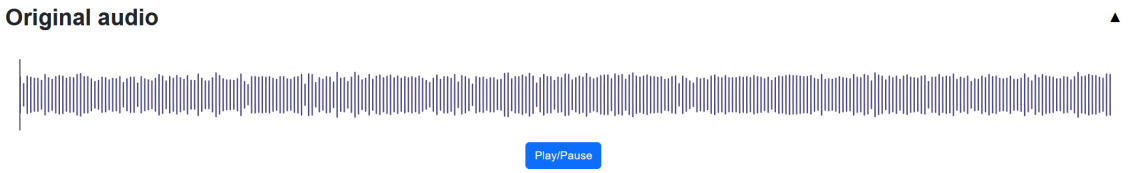


**Figure 5.34.** The visualization of audio playback.

The frontend must also provide a visualization of intensity of the attribution map (Sec. 2.6) over time. This functionality is implemented using ChartJS library [36]. The attribution intensity is a summation of all waveform frequencies over a time so it may be displayed using a simple linear plot that the ChartJS library provides. The exemplary attribution plot definition is shown in Fig. 5.36 and the generated plot corresponding to it is shown in Fig. 5.35.

### 5.10.3. Error and Notification system

The notification panel is responsible for displaying to the user any notifications and errors that appear during the execution of the frontend. It is implemented as a fixed-size bounded queue of size 10, meaning than any new errors that would exceed the maximal limit of the queue.

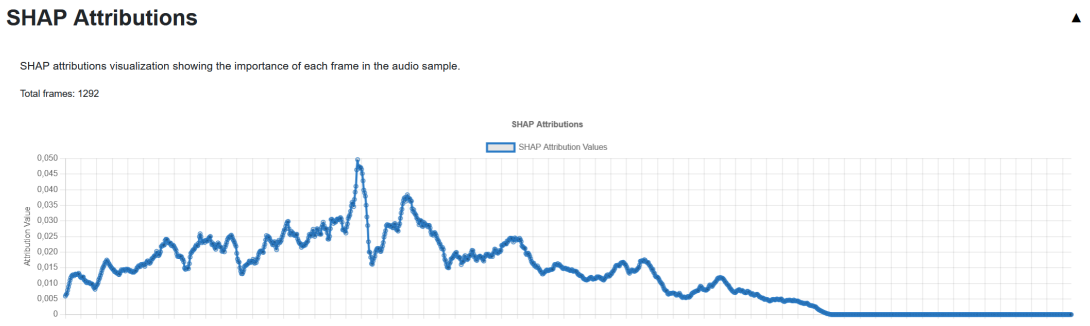


**Figure 5.35.** The linear plot displaying attribution during time of an audio.

The logic of pushing any new notification into the queue is implemented in the function `pushNotification()` as shown in Fig. 5.37. The function adds new notification at the start of the array:

```
const next = [
  { id: Date.now() + Math.random(), ...notification },
    ...prev
  ];
```

It then discards the oldest notifications when the limit exceeds the maximum size:

```
next.slice(0, maxNotifications);
```

Later, the notification queue may be used across the frontend code to handle different events that have occurred. The user provides a type of event into the `'type'` key and the message into the `'message'` key:

```
pushNotification({ type: 'error', message: `Integrated Gradients: Failed to
↪  load audio: ${error.message}` })
```

```
chartInstance.current = new Chart(ctx, {
  type: 'line',
  data: {
    labels: labels,
    datasets: [{ label: `Integrated gradients Attribution Values`, data: attributions,
      fill: false, borderColor: 'rgb(42, 123, 198)', tension: 0.1
    }]
  },
  options: {
    responsive: true,
    maintainAspectRatio: false,
    plugins: {
      title: { display: true, text: 'Integrated gradients Attributions' },
      tooltip: {
        callbacks: {
          title: (tooltipItems) => { return `Frame: ${tooltipItems[0].label}` },
          label: (tooltipItem) => { return `Value: ${tooltipItem.raw.toFixed(4)}` }
        }
      }
    },
    scales: {
      x: { title: { display: true, text: 'Frame Index' } },
      y: { beginAtZero: true, title: { display: true, text: 'Attribution Value' }
      }
    }
  }
})
```

**Figure 5.36.** Definition of attribution plot defined in the `Igradients` component.

```
// Notification context
export const NotificationContext = createContext({
  notifications: [],
  pushNotification: () => {},
});

function App() {
  const [notifications, setNotifications] = useState([]); // {id, type,
   ↪  message}
  const maxNotifications = 10;
  const handleNavigation = (section) => {
    setCurrentSection(section);
  };

  // Push notification (circular queue)
  const pushNotification = useCallback((notification) => {
    setNotifications(prev => {
      const next = [
        { id: Date.now() + Math.random(), ...notification },
        ...prev
      ];
      return next.slice(0, maxNotifications);
    });
  }, []);
}
```

**Figure 5.37.** The implementation of ring-buffer responsible for storing notifications.

### 5.11. Explanation runner command-line script

The framework provides a command-line tool script as a helper tool for quick execution of explanation methods on the user's input using models integrated into the library (Sec. 5.5). The full script implementation is shown in Appendix 2.

For this, the script sets up terminal options as described in Sec. 4.4, with a use of Python's `argparse.ArgumentParser` class:

```
parser = argparse.ArgumentParser(description="Process a model name and input path.")

parser.add_argument('-m', '--model', type=str, required=True,
                    help="Name of the model to use [{sota_music, paans, gtzan},]...")
# ...
args = parser.parse_args()
```

Then, to launch a given explanation method, an instance of the `PylibxaiContext` class is set up with a directory provided by the user through command-line options:

```
context = PylibxaiContext(args.workdir)
```

The script then initializes the appropriate model adapter provided by the user through the `-m/-model` parameter:

```
if args.model == "HCNN":
    adapter = HarmonicCNN(device=device)
elif args.model == "CNN14":
    adapter = Cnn14Adapter(device=device)
elif args.model == "GtzanCNN":
    adapter = GtzanCNNAdapter(model_path=GTZAN_MODEL_PATH, device=device)
else:
    print('Invalid value for -m/--model argument, available: [HCNN, CNN14, GtzanCNN].')
    return
```

The script then sets up the web page view if it was selected or defaults to debug view otherwise:

```
view_type = ViewType.WEBVIEW if args.visualize else ViewType.DEBUG
```

Then, the initial files are saved into the current context, which includes original audio input and a mapping of class labels into their identifiers. The mapping is performed conditionally, whether the model adapter implements `ModelLabelProvider`.

```
context.write_audio(args.input, os.path.join("input.wav"))
if issubclass(type(adapter), ModelLabelProvider):
    context.write_label_mapping(adapter.get_label_mapping(), os.path.join("labels.json"))
```

Finally, after setting up required classes and parsing all command-line options, the script invokes selected explanation methods by running their explainer class on the user's input:

```
    if "lime" in expls:
        view = view_type if expl_count == 1 else ViewType.NONE
        expl_count -= 1
        explainer = LimeExplainer(adapter, context, view_type=view)
        explainer.explain(args.input, target=None)
    if "lrp" in expls:
        # ...
    if "integrated-gradients" in expls:
        # ...
```

# 6. Verification and validation

This chapter presents testing performed to ensure the correctness of the Audio XAI framework implemented as part of the thesis. It also summarizes the results obtained during thesis implementation, particularly the support of the chosen ML models and the explanation methods. The project unit tests have been implemented using *pytest* [29], a testing framework for Python.

## 6.1. Explanation support for models

The thesis aimed to provide suitable explanation methods and integrated models to give a base functionality to work with the framework. The implemented explanation methods by the framework were implemented for each of the integrated models. The results of the supported explanation framework for the implemented models in the framework are shown in the Table. 6.1 where *supported* methods by the model are marked using the symbol ✓and *unsupported* methods using ✗. Some explanation methods had not been launched properly in some models due to various integration issues that were caused either by the development setup (Sec. 4.1) or library issues.

**Table 6.1.** Availability of Explanation Methods per Model

| Model | LIME | Integrated Gradients | LRP |
|---|---|---|---|
| CNN14 | ✓ | ✓ | ✗ |
| GtzanCNN | ✗ | ✓ | ✓ |
| HarmonicCNN | ✓ | ✓ | ✗ |

**Where:** ✓ - method has been implemented successfully; ✗ - method has not been implemented.

Firstly, the LIME method has been implemented for the *GtzanCNN* model, but launching any input on this model required more memory than the GPU on the development setup and caused a memory error, as shown in Fig. 6.1.

In the second case, the implementation of the Layer-wise relevance propagation method has been impossible for *CNN14* and *GtzanCNN* models. This is because the LRP implementation, the one used from the *Captum* framework, is based on a backward propagation mechanism that is sequentially applied to every model. *Captum* requires per-layer rules defined for each of the layers used in machine learning models. Audio models frequently contain audio-specific layers performing, for example, waveform to spectrogram conversion, as in the case of the layer `torchlibrosa.stft.LogmelFilterBank` in *CNN14* and *GtzanCNN*. This disallows the LRP algorithm from running as shown in Fig. 6.2, and such layers' enablement in the Captum library is out of scope of the thesis.

```
$ python pylibxai_explain.py -w ./gtzancnn_expl/ -m GtzanCNN --explainer=lime
 ↪ --target=jazz -i ../data/gtzan_jazz.wav
Colocations handled automatically by placer.
Creating explanation object
Starting LIME explanation
Traceback (most recent call last):
  File "pylibxai\pylibxai\pylibxai_explain.py", line 94, in <module>
    main()
  # Skipped stack trace
  File "torch\nn\functional.py", line 5209, in pad
    return torch._C._nn.pad(input, pad, mode, value)
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 5.05 GiB. GPU 0
 ↪ has a total capacity of 15.99 GiB of which 0 bytes is free. Of the
 ↪ allocated memory 25.28 GiB is allocated by PyTorch, and 77.97 MiB is
 ↪ reserved by PyTorch but unallocated. If reserved but unallocated memory
 ↪ is large try setting PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True to
 ↪ avoid fragmentation.  See documentation for Memory Management
 ↪ (https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)
```

**Figure 6.1.** The out of memory error occurring during LIME explanation on *GtzanCNN* model.

```
  File "pylibxai\captum\captum\attr\_core\lrp.py", line 202, in attribute
    self._check_and_attach_rules()
  File "pylibxai\captum\captum\attr\_core\lrp.py", line 313, in
   ↪ _check_and_attach_rules
    raise TypeError(
TypeError: Module of type <class 'torchlibrosa.stft.LogmelFilterBank'> has no
 ↪ rule defined and nodefault rule exists for this module type. Please, set
 ↪ a ruleexplicitly for this module and assure that it is appropriatefor
 ↪ this type of layer.
```

**Figure 6.2.** Missing rule error for layer `torchlibrosa.stft.LogmelFilterBank` of *CNN14* model.

```
$ ./pylibxai_test.sh
============================= test session starts ==============================
platform win32 -- Python 3.9.20, pytest-8.4.1, pluggy-1.6.0
rootdir: C:\Users\mfalk\Desktop\pylibxai
configfile: pyproject.toml
plugins: anyio-3.7.1, typeguard-4.3.0
collected 73 items

pylibxai\pylibxai_context\test_pylibxai_context.py .................
 [ 23%]
pylibxai\Interfaces\test_interfaces.py .................
 [ 46%]
pylibxai\Explainers\test_explainers.py ...................
 [ 73%]
pylibxai\Views\test_web_view.py ...................
 [100%]

============================== warnings summary ===============================
..\..\anaconda3\envs\pylibxai_env2\lib\site-packages\tensorflow\python\framework\dtypes.py:205
  tensorflow\python\framework\dtypes.py:205: DeprecationWarning: `np.bool8` is a deprecated
   alias for `np.bool_`.  (Deprecated NumPy 1.24)
    np.bool8: (False, True),

..\..\anaconda3\envs\pylibxai_env2\lib\site-packages\flatbuffers\compat.py:19
  flatbuffers\compat.py:19: DeprecationWarning: the imp module is deprecated in favour of
   importlib; see the module's documentation for alternative uses
    import imp

..\..\anaconda3\envs\pylibxai_env2\lib\site-packages\tensorboard\compat\
tensorflow_stub\dtypes.py:326
  tensorboard\compat\tensorflow_stub\dtypes.py:326: DeprecationWarning: `np.bool8`
  is a deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
    np.bool8: (False, True),

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 73 passed, 3 warnings in
 7.91s ========================
[TEST1] CNN14, LIME, Integrated Gradients, Sandman 5s
[TEST2] HarmonicCNN, LIME, Integrated Gradients, Sandman 5s
[TEST3] GtzanCNN, Integrated Gradients, LRP
```

**Figure 6.3.** The output of the testing script *pylibxai_test.sh* showing all tests passing.

## 6.2. Scope of testing

The scope of testing includes unit tests of the given framework's components and integration tests. The test setup for framework's tests is the same as for the development setup and is described in Sec. 4.1.

The library provides a script *pylibxai_test.sh* as shown in Appendix 3 that executes both pytest's unit tests and also a functional tests that utilizes the framework's explanation runner script **pylibxai_explain.py** to test functionally the entire process. The script may be launched from the command-line:

```
$ ./pylibxai_test.sh
```

The shortened testing results are visible in Fig. 6.3. The script output displays launched tests where all 73 unit tests along with functional tests performing explanation passed correctly.

### 6.3. Framework's unit tests

The unit tests were implemented using *pytest* tool and their include total of 73 tests. The implemented tests may be grouped by their testing component and summarized, including:

- **Explainers tests** - The scope of unit testing of the explainer classes includes tests for type checking, input parameter edge cases, and generated explanation results tests. An exemplary test is shown in Fig. 6.4 which tests whether the `LimeExplainer` classes properly return an error when the adapter class passed as input does not implement the expected `LimeAdapter`.

```
def test_lime_explainer_with_non_lime_adapter_raises_error(self):
    """Test LimeExplainer rejects adapters that don't implement
    ↪ LimeAdapter"""
    class NonLimeAdapter:
        def some_method(self):
            pass
    context = Mock()
    adapter = NonLimeAdapter()
    with pytest.raises(TypeError) as excinfo:
        LimeExplainer(adapter, context, ViewType.DEBUG)

    assert "LimeExplainer must be initialized with a model adapter that
    ↪ implements LimeAdapter interface" in str(excinfo.value)
```

**Figure 6.4.** The unit test checks if the adapter class passed to the explainer class has the `LimeAdapter` interface.

- **Models tests** - The testing of the integrated models is performed on the integration level with functional tests that are part of the *pylibxai_test.sh* script, as shown in Appendix 3. The exemplary test that invokes an explanation of the *GtzanCNN* model using Integrated gradients and LRP methods is shown in Fig. 6.5.

```
pylibxai_explain.py -w ./gtzancnn_expl/ -m GtzanCNN
↪ --explainer=integrated-gradients,lrp --target=jazz -i
↪ ./data/gtzan_jazz.wav
```

**Figure 6.5.** The integration tests of the *GtzanCNN* model using a jazz song from *GTZAN* dataset.

- **Interfaces tests** - The scope of unit testing of the interface classes includes tests for proper interface implementation, partial implementation detection, multi-interface support, and inheritance chain contract maintenance. The exemplary tests for interface classes that check if a partial implementation of the `ModelLabelProvider` class raises an error are shown in Fig. 6.6.
- **Context tests** - The scope of unit testing of the interface classes includes tests for directory management and creation, plot generation, attribution data JSON serialization, and audio file operations. The exemplary tests for interface classes that test JSON serialization of explanation's attribution are shown in Fig. 6.7.

```
    def test_model_label_provider_partial_implementation_raises_error(self):
        """Test that ModelLabelProvider raises TypeError when only one abstract
        ↪  method is implemented"""
        class PartialModelLabelProvider(ModelLabelProvider):
            def get_label_mapping(self) -> Dict[int, str]:
                return {0: "rock", 1: "pop", 2: "jazz"}
            # Missing map_target_to_id
        with pytest.raises(TypeError) as excinfo:
            PartialModelLabelProvider()

        assert "abstract method" in str(excinfo.value).lower()
        assert "map_target_to_id" in str(excinfo.value)
```

**Figure 6.6.** The unit test that checks if partial implementation of the `ModelLabelProvider` interface raises an error.

```
    def test_write_attribution_creates_json(self, context, temp_dir):
        """Test attribution JSON creation"""
        test_data = np.array([0.1, 0.2, 0.3, 0.4])
        suffix = "test_attribution.json"
        context.write_attribution(test_data, suffix)
        file_path = os.path.join(temp_dir, suffix)
        assert os.path.exists(file_path)
        with open(file_path, 'r') as f:
            data = json.load(f)

        assert "attributions" in data
        assert data["attributions"] == [0.1, 0.2, 0.3, 0.4]
```

**Figure 6.7**

- **Views tests** - The scope of unit testing of the interface classes includes tests for server startup and process management, error handling for subprocess failures, and resource cleanup and shutdown procedures. The exemplary test for interface classes that test frontend server initialization is shown in Fig. 6.8.

```
    @patch('pylibxai.Views.web_view.subprocess.Popen')
    @patch('pylibxai.Views.web_view.run_file_server')
    @patch('pylibxai.Views.web_view.os.environ')
    def test_start_success(self, mock_environ, mock_run_file_server, mock_popen,
     ↪  web_view):
        """Test successful start of WebView."""
        # ...
        mock_env = {'EXISTING_VAR': 'value'}
        mock_environ.copy.return_value = mock_env
        web_view.start()

        mock_run_file_server.assert_called_once_with("/test/workdir", 8000)
        assert web_view.server == mock_server
        assert mock_env['VITE_PYLIBXAI_STATIC_PORT'] == '8000'
        mock_popen.assert_called_once_with(
            ['npm', 'run', 'dev'],
            cwd=web_view.vite_dir, env=mock_env,
            stdout=sys.stdout, stderr=sys.stderr, shell=True
        )

        assert web_view.vite_process == mock_process
```

**Figure 6.8**

# 7. Conclusions

The goal of this thesis was to develop a software solution enabling XAI explanations on audio model and visualization of explanation data and audio for analysis and comparison. As part of this, the thesis proposes and describes the design of an XAI audio framework that offers its user an API interface for performing explanations and a presentation layer, within which a web interface has been developed to visualize various explanation results such as audio playback, attribution, and the like. Additionally, the framework is based on the Model-View-Presenter pattern to ensure the modularity and maintainability of the solution. The implementation of the developed audio XAI framework is available on GitHub under the link https://github.com/m-falkowski/pylibxai.

A dedicated tool for visualizing audio explanations allows for quick comparison of multiple explanation methods. The framework design relieves the user of a significant portion of the work by managing the explanation and visualization process. The user's role in this process is to define an adapter for their machine learning model and then "inject" or otherwise pass it for use in the framework classes according to the dependency inversion pattern.

The Model-View-Presenter (MVP) architectural pattern used allowed for easy extension of the project with new models and forms of presentation of results. Adding a new alternative view instead of a web page only requires defining the appropriate classes and passing them to the controller. This does not require any modification of explanation classes or adapters from models, as these classes are independent of the view according to the MVP pattern.

The interactive interface significantly supports the analysis capabilities, and the visual form of presentation makes it easier for the user to interpret the results and compare methods. Automatic visualization of audio waves with the ability to listen to them, visualization of spectrograms, and analysis of the attribution of individual methods, along with their comparison, saves the user time by automatically offering an interface that contains a ready-made set of explanations for their model.

The full scope of the thesis, which includes the development of a framework and a web view, as well as the addition of selected XAI methods and audio models, has been completed. However, it was not possible to implement full support for all methods for each model, as described in Sec. 6.1. The problems encountered during the writing of the thesis prevented the addition of support for the LRP method for HCNN and CNN14 models, as it requires the definition of calculations per model layer, and the layers responsible for conversion to spectrogram are not supported in the Captum library. In addition, support for the LIME method was implemented for the GtzanCNN model, but was not tested due to limitations of the environment in which the work was developed.

Future work on the topic of audio Explainable AI frameworks may include:

- Interactive and editable mode - the ability to dynamically select audio fragments for analysis or testing,
- Support for new types of machine learning models such as Large Language Models,

- Support for new explanation methods - integrating techniques such as Grad-CAM, SHAP, or similar XAI methods.

The XAI audio framework project demonstrates the application of classic software engineering rules in the field of Explainable Artificial Intelligence.

# References


[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[2] A. B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, *et al.*, "Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI", Oct. 2019. arXiv: 1910.10045 [cs.AI].

[3] K. Abhishek and D. Kamath, "Attribution-based XAI methods in computer vision: A review", Nov. 2022. arXiv: 2211.14736 [cs.CV].

[4] D. Castelvecchi, "Can we open the black box of ai?", *Nature*, vol. 538, pp. 20–23, Oct. 2016. DOI: 10.1038/538020a.

[5] C. Patrício, J. Neves, and T. Luís, "Explainable deep learning methods in medical diagnosis: A survey", May 2022. DOI: 10.48550/arXiv.2205.04766.

[6] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks", Mar. 2017. arXiv: 1703.01365 [cs.LG].

[7] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions", in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/8a20a8621978632d76c43dfd28b67767-Paper.pdf.

[8] T. Spinner, U. Schlegel, H. Schäfer, and M. El-Assady, "ExplAIner: A visual analytics framework for interactive and explainable machine learning", Jul. 2019. arXiv: 1908.00087 [cs.HC].

[9] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier", Feb. 2016. arXiv: 1602.04938 [cs.LG].

[10] P. Knab, S. Marton, U. Schlegel, and C. Bartelt, "Which LIME should I trust? concepts, challenges, and solutions", Mar. 2025. arXiv: 2503.24365 [cs.LG].

[11] V. Haunschmid, E. Manilow, and G. Widmer, "AudioLIME: Listenable explanations using source separation", Aug. 2020. arXiv: 2008.00582 [cs.SD].

[12] A. Binder, G. Montavon, S. Bach, K.-R. Müller, and W. Samek, "Layer-wise relevance propagation for neural networks with local renormalization layers", Apr. 2016. arXiv: 1604.00825 [cs.CV].

[13] R. Anand, *Digital Signal Processing. An Introduction to Mastering Advanced Techniques for Transforming and Analyzing Signals*. Mercury Learning and Information, 2024, ISBN: 978-18-366-4206-0.

[14] B. Zhang, J. Leitner, and S. Thornton, "Audio recognition using mel spectrograms and convolution neural networks", 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:237274283.

[15] J. Volkmann, S. Stevens, and E. Newman, "A scale for the measurement of the psychological magnitude pitch", *The Journal of the Acoustical Society of America*, vol. 8, pp. 208–208, Jun. 2005. DOI: 10.1121/1.1901999.

[16] M. Won, A. Ferraro, D. Bogdanov, and X. Serra, “Evaluation of CNN-based automatic music tagging models”, Jun. 2020. arXiv: 2006.00751 [eess.AS].

[17] M. Won, S. Chun, O. Nieto, and X. Serrc, “Data-driven harmonic filters for audio representation learning”, in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 536–540. DOI: 10.1109/ICASSP40776.2020.9053669.

[18] J. Gemmeke, D. Ellis, D. Freedman, *et al.*, “Audio set: An ontology and human-labeled dataset for audio events”, Mar. 2017, pp. 776–780. DOI: 10.1109/ICASSP.2017.7952261.

[19] Q. Kong, Y. Cao, T. Iqbal, Y. Wang, W. Wang, and M. D. Plumbley, “PANNs: Large-Scale pretrained audio neural networks for audio pattern recognition”, Dec. 2019. arXiv: 1912.10211 [cs.SD].

[20] B. L. Sturm, “The GTZAN dataset: Its contents, its faults, their effects on evaluation, and its future use”, Jun. 2013. arXiv: 1306.1461 [cs.SD].

[21] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003, pp. 127–131, ISBN: 978-0135974445.

[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Chichester: Wiley, 1996, ISBN: 978-0-471-95869-7.

[23] M. Potel, “Mvp: Model-view-presenter the taligent programming model for c++ and java", taligent inc”, May 2011.

[24] A. Saucedo and X. contributors, *Xai - explainability toolbox 0.1 documentation*, Accessed (11.08.2025): https://ethicalml.github.io/xai/index.html, 2025.

[25] P. contributors, *Pytorch 2.8.0 documentation*, Accessed (17.08.2025): https://docs.pytorch.org/docs/2.8/index.html, 2025.

[26] N. contributors, *Numpy 2.3 documentation*, Accessed (23.08.2025): https://numpy.org/doc/2.3/, 2025.

[27] L. contributors, *Librosa 0.11.0 documentation*, Accessed (17.08.2025): https://librosa.org/doc/0.11.0/index.html, 2025.

[28] T. contributors, *Torchaudio 2.8.0 documentation*, Accessed (17.08.2025): https://docs.pytorch.org/audio/stable/index.html, 2025.

[29] P. contributors, *Pytest 8.3.x documentation*, Accessed (24.07.2025): https://docs.pytest.org/en/8.3.x/, 2025.

[30] M. contributors, *Matplotlib 3.10.5 documentation*, Accessed (17.08.2025): https://matplotlib.org/stable/index.html, 2025.

[31] C. contributors, *Captum documentation*, Accessed (17.08.2025): https://captum.ai/docs/introduction, 2025.

[32] R. Contributors and I. Meta Platforms, *React v19.1 documentation*, Accessed (13.07.2025): https://react.dev/reference/react, 2025.

[33] V. Inc. and V. Contributors, *Vite v6 documentation*, Accessed (12.07.2025): https://v6.vite.dev/guide/, 2025.

[34] B. Contributors, *Bootstrap v5.0.2 documentation*, Accessed (14.07.2025): https://getbootstrap.com/docs/5.0/getting-started/introduction/, 2025.

[35] W. Contributors, *Wavesurfer.js v7.10.0 documentation*, Accessed (14.07.2025): https://wavesurfer.xyz/docs/, 2025.

[36] C. Contributors, *Chart.js 4.5.0 documentation*, Accessed (14.07.2025): https://www.chartjs.org/docs/4.5.0/, 2025.

[37] P. S. Foundation, *Python 3.9.22 documentation*, Accessed (03.07.2025): https://docs.python.org/3.9/library/abc.html, 2025.

# List of Symbols and Abbreviations

**XAI** – Explainable artificial intelligence
**SOTA** – State-of-art Music Models
**LRP** – Layer-wise relevance propagation
**MVP** – Model-View-Presenter
**AI** – Artificial Intelligence
**ML** – Machine Learning
**JSON** – JavaScript Object Notation
**CORS** – Cross-origin resource sharing
**HTTP** – Hypertext Transfer Protocol
**API** – Application Programming Interface
**GPU** – Graphics processing unit
**HCNN** – HarmonicCNN

# List of Figures


2.1 The diagram depicts a black-box model with post-hoc explanation techniques applied to its output. . . . 9
2.2 The taxonomy of the Explainable Artificial Intelligence [5]. . . . 10
2.3 The diagram depicts the process of the LIME explanation [10]. . . . 11
2.4 The diagram shows the pipeline of the audioLIME method. . . . 12
2.5 The image shows the result of audioLIME's explanation. . . . 13
2.6 The image the attribution map of Integrated Gradient's explanation. . . . 13
2.7 The process of sampling a sine audio waveform with constant interval. Sampled points are represented using red dots. . . . 14
2.8 The picture shows an exemplary spectrogram. . . . 15
2.9 The example of the DIP principle implemented in the Python programming language. . . . 16
2.10 The diagram depicts Model-View-Presenter pattern organization. . . . 17
2.11 The Audio XAI framework architecture. . . . 19
4.1 The exemplary usage of the framework API interface in Python's interactive mode. . . . 28
4.2 The output of the help message of the command-line explanation runner. . . . 29
4.3 The exemplary invocation of explanation runner that runs Integrated Gradients and LRP XAI methods on GtzanCNN model. . . . 29
4.4 The picture shows the whole web interface with a main page that contains the LIME results. . . . 30
4.5 The picture shows the whole web interface with a main page that contains the LRP results. . . . 30
4.6 The picture depicts the sidebar component of the web page. . . . 31

4.7 The picture shows the attribution map overlayed on the given input. . . . . 31
4.8 The picture shows the raw input without attribution overlayed. . . . . 31
4.9 The picture shows the plot of attribution intensity over time. . . . . 32
4.10 The image shows the version information panel of the web page. . . . . 32
4.11 The image shows the notification information panel of the web page. . . . . 33
4.12 The image shows the model information page. . . . . 33
5.1 Communication scheme between user application and frontend. . . . . 34
5.2 Library class diagram. . . . . 36
5.3 Class definition of `LimeExplainer`. . . . . 37
5.4 Class definition of `IGradientsAdapter`. . . . . 38
5.5 A sample implementation of `LrpAdapter` for the exemplary `MyModelAdapter` class. . . . . 39
5.6 Class definition of `LrpAdapter`. . . . . 39
5.7 Class definition of `ModelLabelProvider`. . . . . 39
5.8 Model initialization in adapter `HarmonicCNN`. . . . . 40
5.9 Set up of label classes in adapter `HarmonicCNN`. . . . . 41
5.10 Supporting Integrated gradients method in adapter `HarmonicCNN`. . . . . 41
5.11 Support of LRP method in adapter `HarmonicCNN`. . . . . 42
5.12 Support of LIME method in adapter `HarmonicCNN`. . . . . 42
5.13 The constructor definition of the `IGradientsExplainer` class. . . . . 43
5.14 The top-level explain function of the `IGradientsExplainer` class. . . . . 44
5.15 The implementation of `explain_instance` method of the `IGradientsExplainer` class. . . . . 44
5.16 The implementation of `explain_instance_visualize` method of the `IGradientsExplainer` class. . . . . 44
5.17 The `get_smoothed_attribution` method of the `IGradientsExplainer` class. 45
5.18 The initialization process of the `LimeExplainer` class. . . . . 45
5.19 The top-level function `explain` of the `LimeExplainer` class. . . . . 46
5.20 The audioLIME explanation instance defined in `explain()` method of the `LimeExplainer` class. . . . . 46
5.21 The audioLIME explanation execution in the `LimeExplainer` class. . . . . 46
5.22 The end section of the `explain` function of the `LimeExplainer`, which saves the results to a given context class and lanuches a given view class. . . . . 47
5.23 The implementation of `write_attribution` method of `PylibxaiContext` class. . . . . 47
5.24 Class definition of `PylibxaiContext`. . . . . 47
5.25 Class definition of `ViewInterface`. . . . . 49
5.26 Class definition of `ViewType`. . . . . 49
5.27 Class definition of `WebView`. . . . . 50
5.28 Class definition of `DebugView`. . . . . 51
5.29 Implementation of backend server. . . . . 52
5.30 Class definition of `CORSHTTPRequestHandler`. . . . . 52

5.31 The organization of website. Each components is identified by a number. . . . 53
5.32 Data fetching logic defined in a `Igradients` component. . . . . 55
5.33 Audio playback definition in a `Igradients` component. . . . . 56
5.34 The visualization of audio playback. . . . . 56
5.35 The linear plot displaying attribution during time of an audio. . . . . 57
5.36 Definition of attribution plot defined in the `Igradients` component. . . . . 58
5.37 The implementation of ring-buffer responsible for storing notifications. . . . . 59
6.1 The out of memory error occurring during LIME explanation on *GtzanCNN* model. . . . . 63
6.2 Missing rule error for layer `torchlibrosa.stft.LogmelFilterBank` of *CNN14* model. . . . . 63
6.3 The output of the testing script *pylibxai_test.sh* showing all tests passing. . . . 64
6.4 The unit test checks if the adapter class passed to the explainer class has the `LimeAdapter` interface. . . . . 65
6.5 The integration tests of the *GtzanCNN* model using a jazz song from *GTZAN* dataset. . . . . 65
6.6 The unit test that checks if partial implementation of the `ModelLabelProvider` interface raises an error. . . . . 66
6.7 . . . . . 66
6.8 . . . . . 67

# List of Tables

6.1 Availability of Explanation Methods per Model . . . . . 62

# List of Appendices

1. The installation script *setup.sh* . . . . . 77
2. Listing of the script `pylibxai_explain.py` . . . . . 79
3. The testing script *pylibxai_test.sh* . . . . . 81

# Appendix 1. The installation script *setup.sh*

```
#!/bin/env bash

# Download AudioSet class labels indices
# wget http://storage.googleapis.com/us_audioset/youtube_corpus/v1/csv/class_labels_indices.csv -O class_labels_indices.csv

if ! command -v conda &> /dev/null
then
    echo "conda could not be found"
    exit
fi

if ! command -v git &> /dev/null
then
    echo "git could not be found"
    exit
fi

if ! command -v curl &> /dev/null
then
    echo "curl could not be found"
    exit
fi

function install_regressors() {
    git clone -n https://github.com/nsh87/regressors.git
    cd regressors || exit
    git checkout HEAD -- :^".editorconfig*.py"
    git commit -m 'Remove editorconfig files'
    curl -L -o regressors_commit.patch https://github.com/nsh87/regressors/commit/717c8e7009247cfa74af09a5d5bfc592752c04ae.patch
    git am regressors_commit.patch
    python setup.py install
    cd .. || exit
    rm -rf regressors
}

function install_audiolime() {
    git clone https://github.com/CPJKU/audioLIME.git
    cd audioLIME || exit
    python setup.py install
    cd .. || exit
    rm -rf audioLIME
}

function install_captum() {
    git clone https://github.com/pytorch/captum.git
    cd captum || exit
    python setup.py install
    cd .. || exit
    rm -rf captum
}

pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

install_regressors
install_audiolime
install_captum

pip install spleeter
```

```
pip install tqdm
pip install matplotlib
pip install 'scipy>=0.9'
pip install librosa
pip install pandas seaborn 'statsmodels>=0.6.1'
pip install soundfile
pip install fire
pip install tensorboard
pip install torchlibrosa

pip install pytest
conda install -y -c conda-forge tk

# install pylibxai
pip install -e .

conda install -y -c conda-forge nodejs

cd pylibxai/pylibxai-ui
npm install
cd ../../

sudo cp pylibxai/pylibxai_explain.py /usr/local/bin/
```

# Appendix 2. Listing of the script `pylibxai_explain.py`

```
import torch
import argparse
import torchaudio
import os

from pylibxai.model_adapters import HarmonicCNN, Cnn14Adapter, GtzanCNNAdapter
from pylibxai.pylibxai_context import PylibxaiContext
from pylibxai.Explainers import LimeExplainer, IGradientsExplainer, LRPExplainer
from pylibxai.Interfaces import ViewType, ModelLabelProvider
from utils import get_install_path

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
GTZAN_MODEL_PATH= get_install_path() / "pylibxai" / "models" / "GtzanCNN" / "gtzan_cnn.ckpt"

def main():
    parser = argparse.ArgumentParser(description="Process a model name and input path.")

    parser.add_argument('-m', '--model', type=str, required=True,
                        help="Name of the model to use [{sota_music, paans, gtzan},]...")
    parser.add_argument('-u', '--visualize', action='store_true',
                        help="Enable visualization of audio in browser-based UI.")
    parser.add_argument('-e', '--explainer', type=str, required=True,
                        help="Name of the explainer to use [lime, integrated-gradients, lrp].")
    parser.add_argument('-t', '--target', type=str, required=True,
                        help="Name or index of the label to explain.\
                              Mapping is done automatically based on the model if the model
                              ↪ provides it.")
    parser.add_argument('-i', '--input', type=str, required=True,
                        help="Path to the input file or directory.")
    parser.add_argument('-w', '--workdir', type=str, required=True,
                        help="Path to the workdir directory.")
    parser.add_argument('-d', '--device', type=str, default=DEVICE,
                        help="Device to use for computation [cpu, cuda]. Default is 'cuda' if
                        ↪ available, otherwise 'cpu'.")
    args = parser.parse_args()

    device = args.device if args.device is not None else DEVICE
    assert device in ['cpu', 'cuda'], "Device must be either 'cpu' or 'cuda'."

    expls = args.explainer.split(",")
    assert all(ex in ["lime", "integrated-gradients", "lrp"] for ex in expls), \
        "Invalid explainer specified. Available options: [lime, integrated-gradients, lrp]."

    context = PylibxaiContext(args.workdir)

    if args.model == "HCNN":
        adapter = HarmonicCNN(device=device)
    elif args.model == "CNN14":
        adapter = Cnn14Adapter(device=device)
    elif args.model == "GtzanCNN":
        adapter = GtzanCNNAdapter(model_path=GTZAN_MODEL_PATH, device=device)
    else:
        print('Invalid value for -m/--model argument, available: [HCNN, CNN14, GtzanCNN].')
        return

    view_type = ViewType.WEBVIEW if args.visualize else ViewType.DEBUG
    expl_count = len(expls)

    # Attempt parsing label as an integer, if it fails then assume it's a string label
    try:
```

```
            target = int(args.target)
        except ValueError:
            target = args.target

        # copy input audio to workdir
        context.write_audio(args.input, os.path.join("input.wav"))
        if issubclass(type(adapter), ModelLabelProvider):
            context.write_label_mapping(adapter.get_label_mapping(), os.path.join("labels.json"))

        if "lime" in expls:
            view = view_type if expl_count == 1 else ViewType.NONE
            expl_count -= 1
            explainer = LimeExplainer(adapter, context, view_type=view)
            explainer.explain(args.input, target=None)
        if "lrp" in expls:
            view = view_type if expl_count == 1 else ViewType.NONE
            expl_count -= 1
            audio, _ = torchaudio.load(args.input, normalize=True)
            audio = audio.to(device)
            explainer = LRPExplainer(adapter, context, device, view_type=view)
            explainer.explain(audio, target=target)
        if "integrated-gradients" in expls:
            view = view_type if expl_count == 1 else ViewType.NONE
            expl_count -= 1
            audio, _ = torchaudio.load(args.input, normalize=True)
            audio = audio.to(device)
            explainer = IGradientsExplainer(adapter, context, device, view_type=view)
            explainer.explain(audio, target=target)

    if __name__ == '__main__':
        main()
```

# Appendix 3. The testing script *pylibxai_test.sh*

```
#!/usr/bin/env bash

GREEN="\033[32m"
CLR="\033[0m"

# Run unit tests for pylibxai
python -m pytest -v pylibxai/pylibxai_context/test_pylibxai_context.py \
                    pylibxai/Interfaces/test_interfaces.py \
                    pylibxai/Explainers/test_explainers.py \
                    pylibxai/Views/test_web_view.py


echo -e "${GREEN}[TEST1]${CLR} CNN14, LIME, Integrated Gradients, Sandman 5s"
mkdir -p ./cnn14_expl/ &&
python ./pylibxai/pylibxai_explain.py -w ./cnn14_expl/ -m CNN14
 ↪ --explainer=lime,integrated-gradients --target=0 -i ./data/sandman_5s.wav
 ↪ &&
rm -rf ./cnn14_expl/

echo -e "${GREEN}[TEST2]${CLR} HarmonicCNN, LIME, Integrated Gradients,
 ↪ Sandman 5s"
mkdir -p ./harmoniccnn_expl/ &&
python ./pylibxai/pylibxai_explain.py -w ./harmoniccnn_expl/ -m HCNN
 ↪ --explainer=lime,integrated-gradients --target=0 \
                            -i ./data/sandman_5s.wav &&
rm -rf ./harmoniccnn_expl/

echo -e "${GREEN}[TEST3]${CLR} GtzanCNN, Integrated Gradients, LRP"
mkdir -p ./gtzancnn_expl/ &&
python ./pylibxai/pylibxai_explain.py -w ./gtzancnn_expl/ -m GtzanCNN
 ↪ --explainer=integrated-gradients,lrp --target=jazz \
                            -i ./data/gtzan_jazz.wav &&
rm -rf ./gtzancnn_expl/
```