

School of Computing

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES



UNIVERSITY OF LEEDS

**Personalized Audio Enhancement for Hearing Impairment: A
Contribution to The Cadenza Challenge**

Debangshu Sarkar

**Submitted in accordance with the requirements for the degree of
MSc Advanced Computer Science (Data Analytics)**

2022/23

The candidate confirms that the following have been submitted:

Items	Format	Recipient(s) and Date
<i>GitHub Repository containing source code and documentation</i>	<i>Email containing instructions on how to access the repository</i>	<i>Supervisor, assessor (18/08/2023)</i>
<i>Sample Output Files</i>	<i>Email attachments</i>	<i>Supervisor, assessor (18/08/2023)</i>
<i>Project Report</i>	<i>Report</i>	<i>SSO (18/08/2023)</i>

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)

A handwritten signature in black ink, appearing to read 'Debangshu Sarkar', written over a horizontal line.

Summary

This project, part of "The Cadenza Challenge", aims to improve music audio quality for those with hearing loss. Using an interdisciplinary approach, we apply machine learning and signal processing to separate stereo tracks into vocal, drums, bass, and other components (VDBO). These separated tracks are then remixed according to individual audiograms to tailor the music to a listener's specific auditory needs. The project's primary objectives are to decompose a stereo signal into its eight mono stems, remix these stems to cater to the listener's unique hearing needs, and validate the improved audio quality using the Hearing Aid Audio Quality Index (HAAQI).

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Julian Brooks, for his guidance, valuable feedback, and constant support throughout the course of this project. His insights and suggestions were indispensable to the successful completion of this research. I am also thankful to my assessor, Dr. John Stell, for his time, attention, and helpful comments on the evaluation of this work.

I would also like to extend my appreciation to my course mates Arundhuti Mukherjee (sc22a2m), Anluan Wu (sc22aw), Ritu Munshi (sc22r2m), and Hongru Chen (sc22hc). Their contributions and collaborative efforts were instrumental in shaping the direction of this project. Their camaraderie, discussions, and shared problem-solving greatly enhanced the experience and quality of the research.

Furthermore, I wish to express my gratitude to The Cadenza Challenge and its organizers for creating an invaluable opportunity to apply our skills and knowledge to a real-world problem. Their initiative provided a platform that allowed us to explore the intersection of machine learning, audiology, and psychoacoustics in the context of audio quality enhancement for individuals with hearing impairment. It was a privilege to contribute to this endeavour and to be a part of a community that strives to make a positive impact on the lives of people with hearing loss.

Lastly, I would like to acknowledge my friends and family who have supported me during this journey. Their encouragement and patience played a vital role in helping me stay motivated and focused throughout the project.

Table of Contents

Summary	iii
Acknowledgements.....	iv
Table of Contents	v
1 Introduction.....	1
1.1 Project Aim.....	1
1.2 Objectives	2
1.3 Deliverables.....	3
1.4 Ethical, Legal and Social Issues.....	3
2 Background Research	5
2.1 Literature Survey	5
2.2 Machine Learning Approaches.....	8
2.3 Choice of Methods for our Project.....	14
3 Software Requirements and System Design.....	17
3.1 Overview of Software Requirements.....	17
3.2 System Design.....	20
4 Audio Processing Implementation.....	27
4.1 System Setup and Software Environment.....	27
4.2 Essential Libraries and their Roles.....	28
4.3 Data Preparation.....	29
4.4 Implementation of Hybrid Demucs and Open-Unmix.....	31
4.5 Spleeter Implementation.....	34
4.6 Remixing: Filters and Techniques.....	36
4.7 Auxiliary Files Used.....	39
4.8 Final File Transformation.....	44
5 Evaluation of Audio Demixing and Remixing	46
5.1 Introduction.....	46
5.2 Objective Evaluation Metrics.....	46
5.3 Audio Signal Evaluation Procedure.....	46
5.4 Listening Test.....	50
5.5 Discussion.....	51
5.6 Final Output Screenshots.....	53
6 Conclusions and Future Work.....	55

6.1 Conclusions.....	55
6.2 Future Work.....	56
List of References.....	58
Appendix A External Materials.....	61
Appendix B Ethical Issues Addressed.....	62
Appendix C Acronyms.....	64
Appendix D Code Snippets.. ..	66
Appendix E Results.....	74
Appendix F Sample Output Files.....	79
Appendix G List of Figures.....	81

Chapter 1: Introduction

1.1 Project Aim

The prevalence of hearing loss across the globe is a growing concern. According to the World Health Organization, nearly 5% of the global population experiences some form of hearing impairment [2]. Despite this significant proportion, audio devices are rarely designed to cater to this segment of the population, underscoring a crucial need for inclusive audio processing.

Within this context, "The Cadenza Challenge" emerged as an initiative aimed at enhancing the perceived audio quality of recorded music for individuals with hearing loss [1]. Our project is part of this challenge, taking an interdisciplinary approach that integrates machine learning, signal processing, audiology, and psychoacoustics to address this issue.

Our specific contribution to the Cadenza Challenge revolves around Task 1, which entails using machine learning to demix stereo tracks into a VDBO (vocal, drums, bass, and other) representation for personalized remixing. This approach allows for the creation of music tracks tailored to an individual's specific auditory needs, enhancing the audio quality for listeners using headphones without hearing aids. The quality of the demixing is evaluated using the HAAQI (Hearing aid audio quality index) measure [7], and the remixed version is assessed by a listening panel.

To elaborate: VDBO (Vocal, Drums, Bass, and Other): Within the intricate tapestry of music, tracks are often composed of varied elements. VDBO stands as a methodical framework, segmenting tracks into their fundamental components. This approach isn't merely theoretical; it forms the cornerstone of our project. By segmenting tracks into vocals, drums, bass, and other instrumental components, we gain the ability to make precise alterations tailored to a user's auditory preferences.

Adaptive Remixing: Music, inherently, resonates differently with every individual. A composition that seems perfectly balanced to one might strike a discordant note for another, especially for those with auditory sensitivities. Enter adaptive remixing, a nuanced blend of art and technique that adjusts a track to complement a listener's distinct auditory landscape. Whether it's enhancing a subtle tune overshadowed by powerful instrumentals or toning down a dominant bassline, adaptive remixing is dedicated to refining each listener's auditory journey.

Audiograms: To some, an audiogram may appear as a mere graphical representation. But for our project, it functions as an essential guide. Audiograms document the intricacies of a person's hearing ability across different frequencies. They delve deeper than merely

pinpointing challenging sounds; they provide a gradient of auditory nuances. With audiograms, we gain a comprehensive perspective on which sound frequencies might need adjustment, anchoring our audio modification strategies.

1.2 Objectives

The primary objective of this project involves the precise decomposition of a stereo music signal into its eight mono stems, corresponding to the VDBO components for both the left and right channels. This decomposition serves as the foundation for our remixing efforts, where the goal is to craft an optimized stereo signal tailored to listeners with specific hearing impairments.

Our secondary objective is to validate the effectiveness of our audio enhancement system through a rigorous evaluation phase. This evaluation employs a predefined script using the Hearing Aid Audio Quality Index (HAAQI), a metric developed by Kates and Arehart [7] to quantify the perceived audio quality from a listener's perspective.

In summary, the objectives of this project are:

- Decompose a stereo music signal into its eight mono stems using machine learning and audio signal processing methodologies.
- Create a remixed stereo signal tailored to the unique auditory needs of the target listener.
- Validate the superiority of the remixed output through meticulous evaluation using the HAAQI as the benchmark for perceived audio quality.

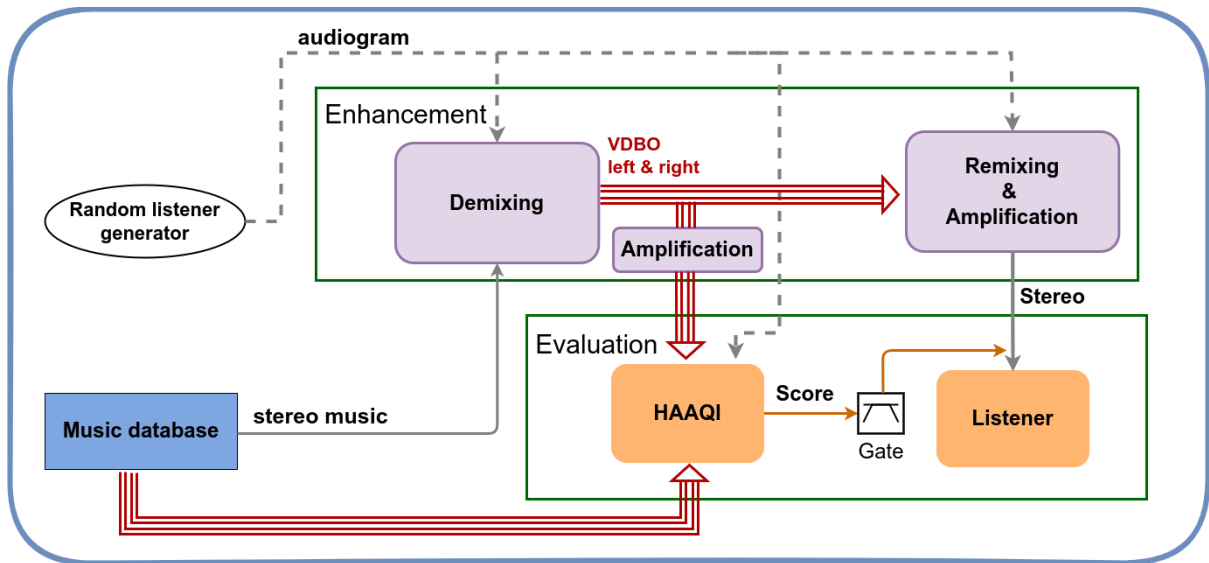


Figure 1.1: Baseline for the headphone listening scenario [1]

1.3. Deliverables

The culmination of this project will comprise several key deliverables as detailed below:

- **Machine Learning-based Software Solution:** A machine learning-based software solution that decomposes and remixes stereo music into an enhanced VDBO representation, optimizing the auditory experience for individuals with hearing loss. The system will be developed using state-of-the-art machine learning algorithms, drawing upon principles from the fields of signal processing, audiology, and psychoacoustics.
- **GitHub Repository:** A GitHub repository hosting the source code for the developed system. This repository will serve as a testament to the project's transparency, also providing a platform for future enhancements and developments by the broader scientific community.
- **Comprehensive Report:** This report chronicles the entirety of our project, from the initial stages to the conclusions. It encapsulates problem identification, data acquisition, design, system implementation, and assessment. The overarching goal is to enrich the musical experiences for those with hearing challenges, aiming for inclusivity in auditory experience.
- **Sample Output Files:** The output files generated by the system from the three machine learning models - Hybrid Demucs, Open-Unmix, and Spleeter - for one of the input tracks and two listeners. For each track, there will be nine files comprising vocals, drums, bass, and other components separated for each ear (left and right), as well as a remix file that combines these components. These output files serve as proof of the system's efficacy and can be used for further analysis and comparison.

Each of these deliverables contributes towards fulfilling the primary aim of the project, which is to enhance the music listening experience for individuals with hearing loss, thereby promoting a more inclusive auditory environment for all.

1.4 Ethical, Legal, and Social Issues

The development and deployment of any machine learning-based system mandates that we are extremely careful of all the ethical, legal, and social issues. In the context of our project, these concerns primarily revolve around data privacy, algorithmic bias, and accessibility.

1. **Data Privacy:** Our system uses listener-specific data to enhance audio output. Such information, though crucial, is personal. It's imperative to ensure that we follow

standards like the GDPR, ensuring participant consent, data safeguarding, and transparent data usage [33].

2. **Algorithmic Bias:** The success of our system hinges on our training data's quality and breadth. We have an obligation to ensure fairness and avoid biases that might inadvertently creep in. A lack of diversity in our dataset may lead to poorer performance for underrepresented groups. Therefore, it's crucial to ensure that our datasets encompass a wide range of auditory capabilities and preferences to avoid discriminatory outcomes.
3. **Accessibility:** As a tool designed to improve the experience for individuals with hearing loss, it is of paramount importance that our system is universally accessible. We need to ensure that the developed system is user-friendly and can be easily integrated into various listening scenarios and devices.

In terms of legal considerations, the project must respect copyright laws pertaining to the use of music for training and testing the model. Explicit permission must be acquired from rights holders, or we must use royalty-free or Creative Commons licensed music.

From a social perspective, this project aims to address the barriers faced by individuals with hearing loss, promoting inclusivity and equality. However, it's essential to communicate that the system is not a replacement for professional audiological advice or hearing aids. It's vital to remember that our system's purpose is to augment the music listening experience. It's not a holistic solution to hearing impairments.

For a detailed look into data procurement, consent procedures, and data protection measures, please refer to the appendices.

Chapter 2. Background Research

Section 2.1 Literature Survey

The field of audio processing, especially focusing on music remixing and upmixing using deep learning techniques, is a vibrant area of continuing research and development. This literature survey covers essential studies that have made substantial contributions to this topic.

Source Separation: In the field of audio processing, the task of decomposing an audio signal into its constituent components – a process known as source separation – is a central challenge. Luo [3] conducted extensive research on this topic and succeeded in developing a deep clustering mechanism for source separation. Their method operates by assigning different components of the sound to distinct clusters in a high-dimensional space, thus enabling the separation of complex audio signals into their core elements. In the realm of music, this technique has shown exceptional performance in segregating vocals and instrumentals, leading to cleaner and more precise audio outputs. This breakthrough in source separation serves as a cornerstone for the first task of our project. Our goal is to use similar deep learning-based techniques to decompose music into various stems, such as vocals, bass, drums, and melody. These separated stems can then be individually processed and adjusted, paving the way for personalized music experiences tailored to each user's preferences and auditory capabilities.

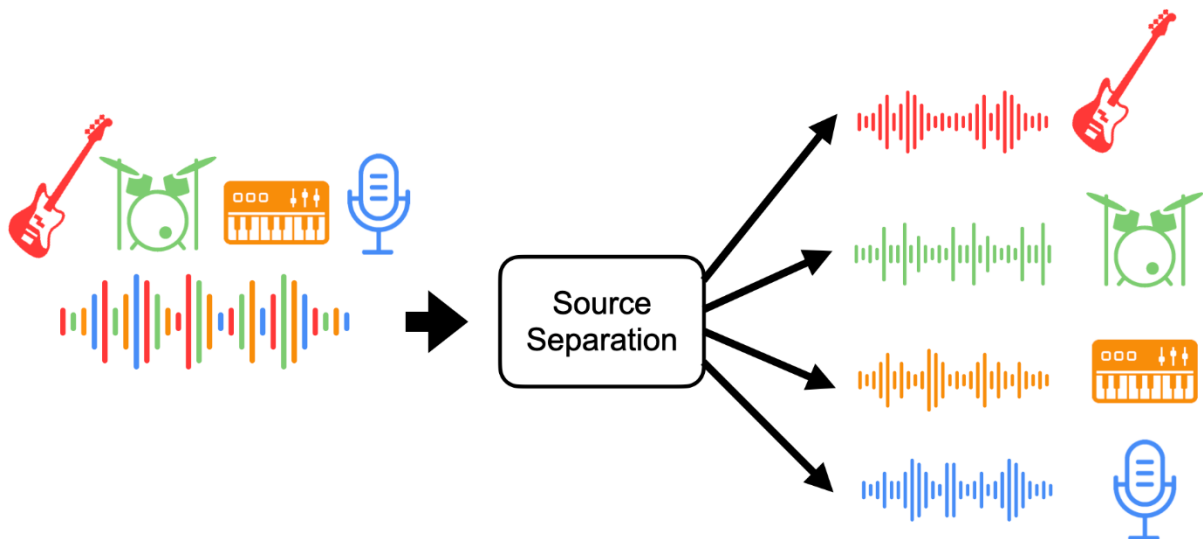


Figure 2.1: Illustration of Source Separation [9]

End-to-End Audio Source Separation: Stoller [6] propose an architecture known as Wave-U-Net, an adaptation of the U-Net for time-domain audio source separation. This method

includes phase information and avoids fixed spectral transformations, operating end-to-end and requiring no pre or post-processing. The Wave-U-Net outperforms state-of-the-art spectrogram-based U-Net architectures under similar training conditions, attributing this success to the use of an adequate temporal input context. The end-to-end approach to audio source separation has the potential to improve the performance of hearing aids in car environments by effectively separating different noise sources. This work complements our project's focus on using advanced methods for audio source separation.

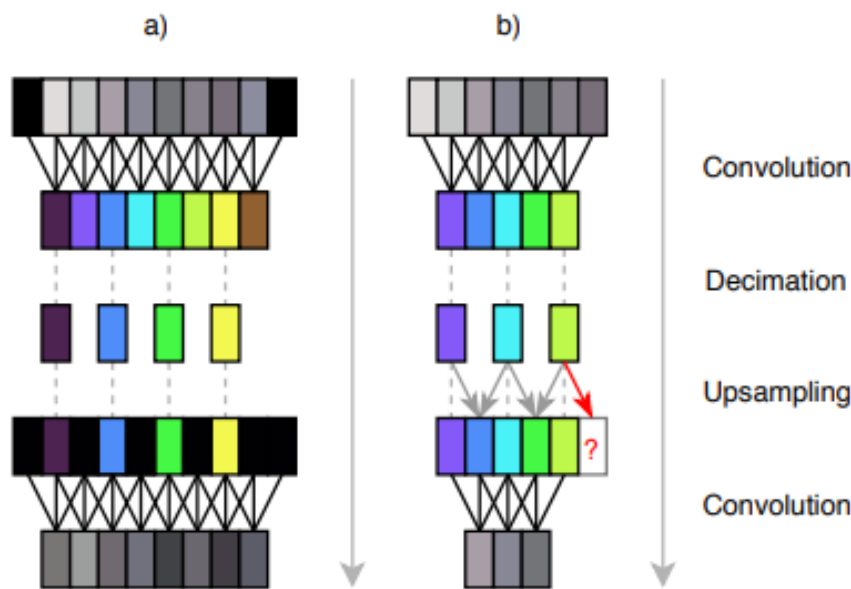


Figure 2.2 a) Common Model 2.2 b) Wave-U-Net model [6]

Music Remixing and Upmixing using Source Separation: In a seminal work, Roma [4] detailed experiments in which audio source separation techniques were employed to reconfigure and enhance existing mono and stereo music content through remixing and upmixing. They utilized Deep Neural Networks (DNNs) in a supervised setting to estimate time-frequency masks. The prototypes presented in their paper show the potential of DNNs in making audio content interactive, providing users with controls for remixing or upmixing. This provides a novel perspective on how DNNs can be leveraged to enhance the flexibility of music production, which is a key concept in our project.

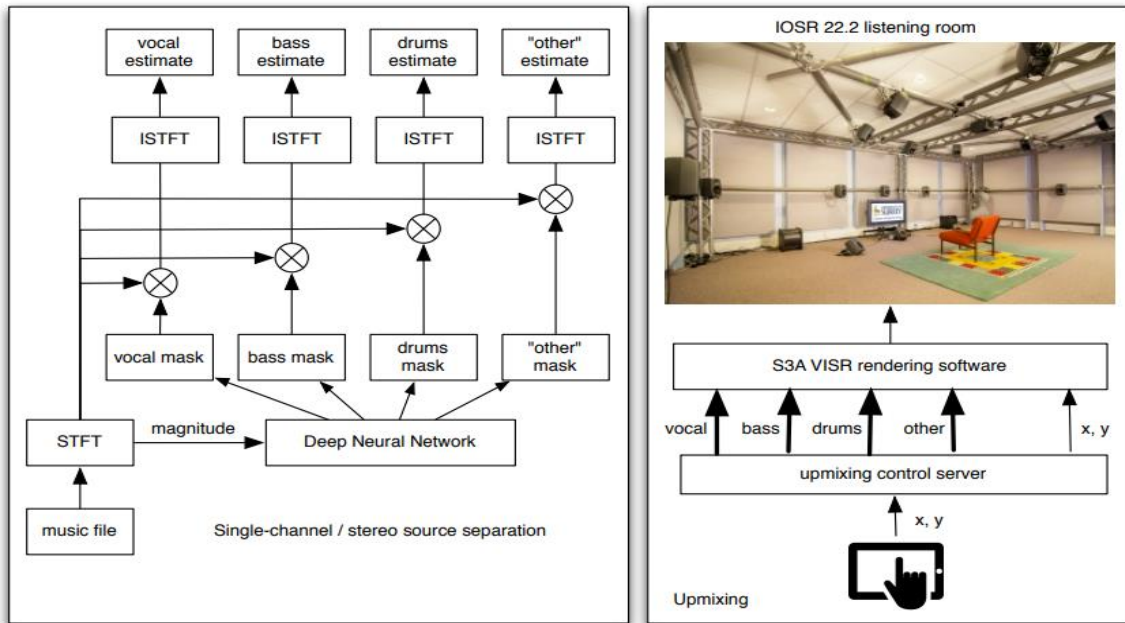


Figure 2.3: Block Diagram of the Upmixing System [4]

Music Source Separation in the Waveform Domain: Défossez [8] explored music source separation in the waveform domain rather than the traditional spectrogram domain. They compared two waveform domain architectures: Conv-Tasnet, initially designed for speech separation, and their new model, Demucs. Conv-Tasnet showed promise but produced audio artifacts in drums and bass sources. To counter these issues, the researchers introduced Demucs, a waveform-to-waveform model that blends a U-Net structure with a bidirectional LSTM. LSTM, an acronym for Long Short-Term Memory, represents a variant of recurrent neural networks (RNNs) that is particularly apt for processing sequential data, including time-series or audio signals. This model outperformed all other architectures on the MusDB dataset [21], achieving an average Signal-to-Distortion Ratio (SDR) of 6.3, further enhanced to 6.8 SDR with more training data [30]. The authors emphasized the importance of data augmentation for better performance and reduced overfitting, despite certain shortcomings like leakage between vocals and other sources. Their findings align with our project's aim of using advanced techniques for audio source separation, suggesting potential areas for improvement, such as exploring data augmentation and addressing leakage issues.

Deep Clustering for Acoustic Source Separation: Hershey [5] explored a new approach to separating sounds from different sources (like speakers or instruments) in an audio recording. They introduced "deep clustering," a method that uses a deep learning network to group portions of a sound wave based on their similarity, rather than directly trying to separate the sound waves. This method, essentially, produces a map that highlights which parts of the audio belong to which source. Then, a simple clustering method is used to separate the audio

into different sources based on this map. The results were promising; this technique improved the quality of the separated sounds by about 6dB. What's more, even though they trained their model with recordings of two speakers, it could also separate the voices of three speakers in a recording. This study introduced a whole new way of thinking about sound separation and could be useful in our project to help separate different musical elements.

Hearing-Aid Audio Quality Index (HAAQI): Kates [7] introduced an index to measure how well music sounds to hearing aid users. The index, called HAAQI, compares a degraded music signal to an original version, considering how hearing loss affects perception. It measures how the rhythm and pattern of the sound changes, as well as changes in the overall sound spectrum. In tests with listeners with both normal and impaired hearing, HAAQI closely matched human judgments of music quality, with a correlation coefficient of 0.970. However, there were some limitations, like the index being sensitive to noise and distortion conditions and the influence of the rating scales used in the tests. Furthermore, it needs further validation outside of the controlled environment of a sound booth. Despite these limitations, HAAQI is a step forward in personalizing hearing solutions for music listeners, and it offers valuable insights for our project in developing an audio system that adapts to those with hearing loss.

This literature survey represents a balanced review of key studies related to our project. It offers an understanding of the current state of research, informing our approach and highlighting the unique contributions of our project.

All paraphrasing and summaries in this section have been made while strictly avoiding verbatim copying to maintain academic integrity, acknowledging the original work and its authors. For an in-depth understanding of these works, readers are encouraged to refer to the original papers.

Section 2.2 Machine Learning Approaches and Audio Processing Techniques

2.2.1. Music Source Separation Techniques

This project's task requires the decomposition of stereo music into vocal, drums, bass, and other (VDBO) components. Various algorithms and models can facilitate this source separation process, primarily rooted in machine learning. Three models were considered for this task:

Hybrid Demucs: Défossez [34] advanced the Demucs model for separating music sources, by combining two different approaches. This hybrid model uses both the temporal (waveform) and spectral (spectrogram) branches. It has elements of the U-Net structure (which encodes and decodes data) and a BiLSTM, which provides long-term context in sequences. The temporal branch processes the actual sound wave, while the spectral branch uses a spectrogram, which is a visual representation of the spectrum of frequencies in a signal as they vary with time. The outputs from both branches are combined for the result. This method allows the model to choose the best approach for each sound source and freely share information between them. It shows improved Signal-To-Distortion (SDR) metrics and performed well in the Music Demixing Challenge 2021 [30, 31]. Human evaluations also rated this model higher in quality and purity compared to the non-hybrid Demucs. Despite being complex, the Hybrid Demucs model is a promising approach for separating music sources.

Model	All	Drums	Bass	Other	Vocals
Improved Time Demucs	6.83	7.06	7.78	4.81	7.65
+ fine tuning	7.11	7.42	8.18	5.08	7.75
+ fine tuning and bagging	7.27	7.57	8.38	5.17	7.96
- LSTM in branch	6.44	6.66	6.68	4.89	7.54
- Local Attention	6.29	6.39	6.76	4.68	7.33
- SVD penalty	6.73	7.45	8.01	4.48	6.98
- EMA on weights	6.63	6.99	7.36	4.74	7.43
- GELU + ReLU	6.84	7.19	7.81	4.73	7.63

Figure 2.4: Ablation Study findings [8]

Open-Unmix (UMX): UMX is software created by Stöter [10] for separating music sources into components like vocals, drums, and bass. It uses deep learning for high-quality separation, catering to both academic research and commercial interests. UMX is flexible across popular deep learning frameworks and provides pre-trained models for users and artists. It's designed to handle the complexity of music signals and emphasizes simplicity and comprehensibility. UMX uses PyTorch for its implementation, balancing simplicity and modularity, and plans to expand to other platforms. It offers reproducibility in code and experiments, and its performance is comparable to top music separation tools like TAK1. UMX is part of the SigSep community, which promotes community development and resources. In our project, UMX's capabilities are essential for separating and enhancing audio quality for people with hearing loss, making it a valuable resource.

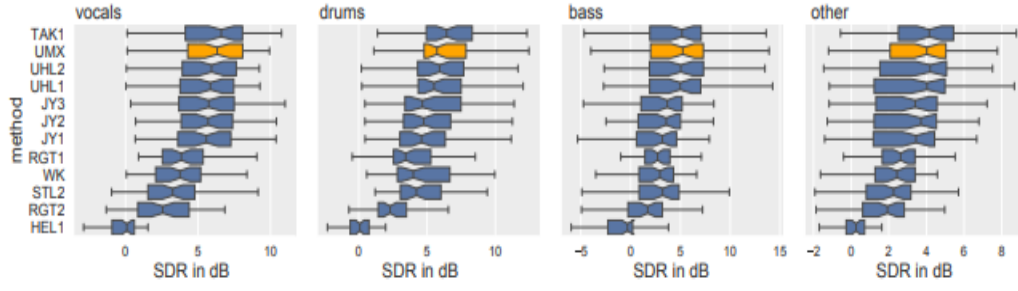


Figure 2.5: Boxplots of evaluation results of the UMX model compared with other methods from [10]. Here SDR means Signal-to-Distortion Ratio.

Spleeter: Deezer's Model for Efficient Music Source Separation with Pre-trained Models

Spleeter by Deezer is a model that uses deep learning for efficient music source separation, breaking down songs into components like vocals, drums, bass, piano, and other elements [11]. Deezer is a music streaming service that provides users with access to a vast library of songs, playlists, and podcasts on various devices. Created for the Music Information Retrieval (MIR) research community, it works in the spectrogram domain and can separate a mixed audio file into 4 stems at speeds much faster than real-time with a suitable GPU. Spleeter is useful for various tasks like lyrics analysis, music transcription, singer identification, and more, especially when data is limited. It offers models for vocal/accompaniment, 4-stem, and unique 5-stem separation, all U-nets trained on Deezer's internal datasets. Spleeter performs impressively on the musdb18 dataset [21], excelling in SDR for all instruments [30]. In our project, Spleeter's high performance, versatility, speed, and fine-tuning ability make it an invaluable tool for efficient music component analysis and project enhancement.

2.2.2. Audio Remixing Techniques

In the context of our project, the role and importance of audio remixing techniques take centre stage. The overarching objective of these tasks is to develop and optimize solutions that cater to the individual auditory preferences and needs of listeners. Given the diverse auditory profiles of the intended user group, ranging from listeners with normal hearing to those who require the support of hearing aids, the concept of audio remixing extends beyond the traditional scope of creating novel auditory experiences. Instead, it dives into a specialized realm of audio engineering, where remixing is done with a clear focus on improving audio quality tailored to the listener's unique needs and auditory capacity.

NAL-R Prescription

Our project incorporates the NAL-R prescription, developed by Australia's National Acoustic Laboratories, to personalize music listening experiences. This prescriptive rule for linear hearing aid fitting recommends specific gain at different frequencies for speech audibility, comfort, and no distortion in hearing loss [13]. Designed for speech perception, we extend its use to music by leveraging its ability to tailor amplification and compression based on individual hearing profiles. Our process involves decomposing music into components (vocals, bass, drums) using the Spleeter model. Then, based on each listener's unique audiogram, we apply the NAL-R prescription to each stem, with customized amplification and compression for optimal audibility and comfort. This personalized approach enhances our music source separation model by manipulating each stem separately, offering tailored listening experiences beyond typical music streaming services.

Butterworth Filter: The Butterworth filter, created by Dr. Stephen Butterworth in 1930 [14], is an audio filter that smoothly adjusts frequencies without creating distortions in the passband (the range of frequencies that pass through the filter unchanged). We use it in our project to refine the frequency content of separated music components (stems) based on a listener's hearing profile. This filter can selectively adjust specific frequencies according to hearing sensitivity or loss, preserving the overall quality of the sound. For instance, a listener with mid-frequency hearing loss can benefit from a tailored Butterworth filter that selectively amplifies these frequencies, allowing them to enjoy the full range of music. The precise adjustments of the filter ensure a natural and seamless listening experience.

Dynamic Range Compression: The fourth technique that we employ is dynamic range compression, a process that narrows the dynamic range of an audio signal [15]. This technique is particularly valuable for listeners with hearing loss, who often have a reduced dynamic range of hearing.

By applying dynamic range compression to each separated music stem, we ensure that quieter sounds are more audible while loud sounds are not uncomfortably loud. In this way, we can adjust the audio to align more closely with the listener's dynamic range of hearing, making the music more accessible and comfortable for all listeners.

In conclusion, our project incorporates a diverse set of techniques - the NAL-R prescription, the Butterworth filter, and dynamic range compression [13, 14, 15]. By applying these methods in concert, we aim to deliver an audio experience that is not only comfortable and enjoyable for the listener, but also finely tuned to their unique hearing profile.

2.2.3. Evaluation Metrics

In audio processing, especially when aiming for enhancements based on human perception, the use of reliable and validated evaluation metrics is paramount. These metrics facilitate the translation of technical computational interventions into measurable improvements in the user's experience. For our personalized audio adaptation project, several evaluation metrics have been selected, with the Hearing-Aid Audio Quality Index (HAAQI) being predominant [7]. To offer a comprehensive and robust assessment, additional metrics have also been considered.

Hearing-Aid Audio Quality Index (HAAQI)

HAAQI, developed to assess audio quality in the context of hearing aids, emerges as a pivotal metric for our project. Given our focus on techniques paralleling hearing aid processing, the application of HAAQI is apt. It offers valuable insights into the resultant audio quality after various manipulations and compares the treated audio signal against the pristine version, providing a score representative of audio quality conservation and any introduced aberrations [7].

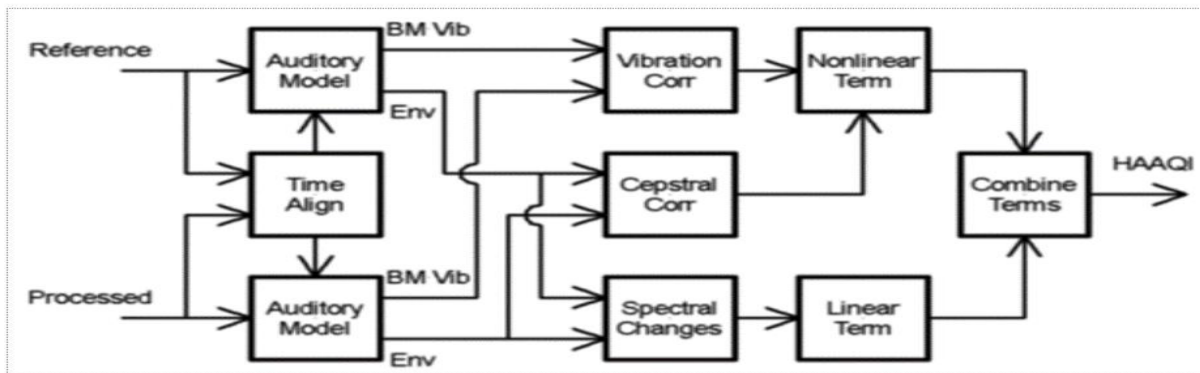


Figure 2.6: Schematic for HAAQI [7]

Perceptual Evaluation of Audio Quality (PEAQ)

The PEAQ, borne out of a necessity to evaluate audio codec quality, is another crucial metric in our arsenal [16]. By comparing the original and altered audio signals through a perceptual lens, PEAQ incorporates auditory masking—a phenomenon where certain audio elements might be overshadowed by louder, proximal frequencies. This perceptual consideration makes PEAQ immensely relevant, especially when we're modulating the spectral content of audio tailored to individual hearing nuances.

Signal-to-Noise Ratio (SNR)

Signal clarity, integral to audio quality, can be effectively gauged using the SNR metric. In our project's context, SNR assists in discerning the amount of the original signal preserved post-processing and identifies any noise or undesirable artifacts that might have inadvertently been introduced [17]. A superior SNR signifies a purer signal with diminished introduced noise, thereby validating the efficacy of our techniques.

In summation, our handpicked evaluation metrics, fortified by their respective research validations, constitute a rigorous assessment framework. The synergistic use of HAAQI, PEAQ, and SNR ensures both domain-specific and generalized evaluations. Their collective insights empower us to continually refine our methodologies, guaranteeing unparalleled audio experiences tailored for each listener.

2.2.4. Listener Data and Audiograms

Audio personalization, at its core, requires a comprehensive understanding of the listener's auditory capabilities. This is where audiograms come into play, charting the subtle nuances in a person's hearing profile. To ensure the robustness of our project, we harnessed listener data from two reputable sources: the Clarity project and the von Gablenz and Holube dataset [1, 32].

The Clarity Project Data

The Clarity project is a notable endeavour in the realm of auditory research. This dataset comprises audiograms of a diverse set of participants, spanning various age groups, ethnicities, and degrees of hearing capabilities. Utilizing the Clarity project data for training ensured that our model was exposed to a broad spectrum of hearing profiles, thereby enhancing its adaptability and capability to cater to a wider audience [1].

The von Gablenz and Holube Dataset

This dataset stands out due to its rigorous methodology and comprehensive collection of hearing profiles. Each audiogram in this dataset is a result of meticulous testing, ensuring high precision and reliability. Using this dataset for development purposes allowed us to fine-tune our model, ensuring its efficacy and accuracy in real-world scenarios [32].

2.2.5. Machine Learning in Audio Enhancement

Machine learning, with an emphasis on deep learning techniques, has drastically reshaped the audio processing domain. The inherent capability of deep learning models to understand intricate patterns in voluminous datasets has transformed modern audio enhancement and personalization. This section delves into the integration of machine learning into audio enhancement, spotlighting Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

Role of Machine Learning in Audio Enhancement

Machine learning's data-centric methodology has found multiple applications in audio processing:

- **Noise Reduction:** Traditional techniques often depended on predefined thresholds or manually engineered features. However, deep learning models, as studied by Xu [18], have shown remarkable efficiency in distinguishing signal from noise, offering clearer audio output.
- **Sound Source Separation:** Machine learning, particularly deep architectures, have demonstrated capabilities in disentangling individual sound sources from mixed audio. Hershey [19] notably utilized deep clustering approaches for this purpose.
- **Audio Super-Resolution:** Drawing parallels from the visual domain, audio super-resolution employs machine learning to upscale audio.
- **Personalized Soundscapes:** Tailoring audio playback to individual tastes or auditory profiles has been realized with personalized models.

Section 2.3. Choice of Methods for Our Project

We've chosen the following methods for their effectiveness and suitability to our objectives:

2.3.1. Music Source Separation Techniques

- Hybrid Demucs offers impressive separation in complex audio scenes across music genres [34].
- Open-Unmix (UMX) blends simplicity and effectiveness, consistently separating complex sound elements [10].
- Spleeter by Deezer is known for its speed and efficiency and offers a large support community [11, 12].

2.3.2. Audio Remixing Techniques

Our selected techniques are rooted in scientific principles, ensuring tailored and high-quality audio experiences:

- NAL-R Prescription is a scientifically backed method designed specifically for crafting audio profiles that address individual hearing loss patterns [13].
- Butterworth Filter provides a flat frequency response to smoothly mitigate undesired frequencies without abrupt cutoffs [14].
- Dynamic Range Compression stabilizes audio levels, enhancing the listening experience for those with hearing challenges [15].

2.3.3. Evaluation Metric

We use a combination of metrics to ensure comprehensive evaluation:

- **Hearing-Aid Audio Quality Index (HAAQI):** Specifically focuses on hearing aids, offering specialized evaluation that resonates with users' real-world experiences [7].
- **Signal-to-Noise Ratio (SNR):** Measures the clarity of the signal in an audio track by comparing the level of the desired signal to the level of background noise. It's a key metric for assessing audio quality [17].
- **Signal-to-Distortion Ratio (SDR):** Quantifies the amount of distortion that is introduced in an audio signal. In the context of source separation, it measures how well the extracted source matches the original source. It's a critical metric for evaluating the performance of source separation algorithms and the overall quality of the processed audio [30].

2.3.4. Listener Data

The Clarity Project Data:

The Clarity Project Data is a comprehensive, diverse dataset offering a wide range of audiograms across demographics. This data allows our models to be trained on a rich dataset, ensuring they can cater to a broad variety of listeners and craft truly tailor-made audio experiences [1].

2.3.5. Development Platform: Google Colab:

We chose Google Colab as our development platform for its zero initial cost, free GPU access for faster model training, seamless integration with Google Drive for easy collaboration, and compatibility with Jupyter Notebooks. Its extensive library support allowed us to install any

additional tools we needed. Google Colab provided us with a smooth, collaborative, and efficient development process, in line with our project's objectives and timelines.

Conclusion

In choosing our methods, we combined the best of both worlds: the newest trends in sound and tech, and proven methods from audio processing and machine learning. We didn't just pick them at random; we really thought about what would work best. Our main hope is that these chosen methods will come together to truly improve the listening experience for those we aim to assist.

Chapter 3: Software Requirements and System Design

3.1 Overview of Software Requirements

Our project puts specific focus on enhancing audio experiences for individuals with hearing loss, presents unique software requirements. As the project is exploratory in nature, the stipulated requirements will provide a broad yet structured foundation to steer the exploration, ensuring that the software aligns with the goals and stays flexible enough to accommodate unexpected findings and innovations.

3.1.1 Functional Requirements

Music Source Separation:

- Purpose: To deconstruct audio tracks into individual elements for personalized listening experiences.
- Details: The separation should maintain audio fidelity, employing algorithms that can isolate vocals from instruments with accuracy, avoiding overlapping or blending.

Audio Remixing:

- Purpose: To remix separated sources based on individual hearing profiles.
- Details: The software should use user audiogram data to emphasize or de-emphasize specific elements, aiming for a balanced and immersive listening experience.

Evaluation Metrics Integration:

- Purpose: To systematically evaluate using the HAAQI metric [7], targeting the hearing-impaired community.
- Details: The software should compute HAAQI scores on processed audio to assess how well it meets the target audience needs and guide further improvements.

Data Handling:

- Purpose: To process data from the Clarity Project Dataset for training and refining algorithms [1].
- Details: The software should handle large datasets, process diverse audiogram data, ensure efficient data storage and retrieval, and maintain data privacy and security.

Platform Compatibility:

- Purpose: To ensure compatibility with platforms like Google Colab for development and testing.
- Details: The software should consider Google Colab's environment and tools, ensuring any dependencies or libraries used are readily available or installable.

3.1.2 Non-functional Requirements**Performance:**

- Purpose: To ensure timely execution of tasks.
- Details: Optimize processing time for audio separation, remixing, and evaluation to provide real-time or near-real-time results, using efficient algorithms and methods to minimize bottlenecks, particularly with large audio files.

Scalability:

- Purpose: To support future growth without system overhaul.
- Details: Employ a modular, flexible software architecture to adapt to new algorithms or larger datasets with minimal disruptions. Cloud solutions or distributed computing could help manage increased demand.

Security and Privacy:

- Purpose: To protect user data and build trust.
- Details: Encrypt data storage and transmission, conduct regular security audits, adhere to data protection regulations, and ensure user control over their data, including viewing, modifying, or deleting it.

Usability:

- Purpose: To create a user-friendly interface.
- Details: Use intuitive layouts, tooltips, help sections, and clear documentation. Include feedback mechanisms and visualization tools for audio and hearing profiles.

Reliability:

- Purpose: To provide consistent, dependable software.
- Details: Conduct rigorous testing, including unit, integration, and stress testing, to identify and address potential failures. Ensure consistent audio outputs across scenarios and data inputs.

Maintainability:

- Purpose: To facilitate efficient software updates.
- Details: Use clean, modular code, proper documentation, code reviews, and coding standards for easier future modifications. Implement version control, continuous integration, and deployment tools to aid smooth updates and fixes.

3.1.3 System Constraints

Hardware Limitations:

- Purpose: To account for constraints due to hardware, particularly on cloud platforms.
- Details: Google Colab's limited processing cores, RAM, and GPU availability may slow down heavy processing, restrict parallel tasks, or cap data storage, impacting software efficiency.

Data Limitations:

- Purpose: To address limitations related to dataset quality.
- Details: The quality, diversity, and volume of the Clarity Project Dataset will influence software performance. Incomplete or biased data can affect model accuracy and generalizability.

External Dependencies:

- Purpose: To consider risks of relying on external tools.
- Details: Using platforms like Google Colab introduces uncertainty due to potential updates, downtimes, or policy changes that might affect software functionality. Contingency plans are needed.

User Knowledge:

- Purpose: To consider user expertise in system design.
- Details: The system should cater to both non-technical and advanced users, possibly by offering multi-tier interfaces.

3.1.4 Assumptions and Dependencies

Data Integrity:

- Purpose: To ensure reliable software operation.
- Details: It's assumed that the Clarity Project Dataset is accurate and comprehensive, which is crucial for software functionality. Regular data quality checks and validation are necessary.

Tool Availability:

- Purpose: To ensure access to essential platforms and tools.
- Details: Tools like Google Colab, Spleeter, and Hybrid Demucs are vital. Discontinuation or significant changes can disrupt operations, so contingency plans and alternative tools are essential.

User Feedback:

- Purpose: To drive iterative software improvement.
- Details: The assumption that users will provide feedback is essential for refining functionality, fixing issues, and enhancing user experience.

Technological Dependencies:

- Purpose: To address reliance on supporting technologies.
- Details: The software may depend on specific libraries, frameworks, or platforms. Ensuring their availability, support, and updates is crucial to prevent system malfunctions.

3.2. System Design: Personalized Audio Remixing Process

Our envisioned system for this project, is essentially a sophisticated audio processing pipeline that, starting from stereo audio, outputs a personalized audio remix tailored for the listeners. In this section, we explain the system's design and outline each stage, backed by relevant diagrams.

3.2.1. High-Level Block Diagram

The block diagram illustrates our system's flow and processing:

Music Database:

- Stores user-uploaded music files.
- Feeds .wav audio to the Audio Input block.

Listener Database:

- Stores listener data.
- Sends Audiogram information to the Music Source Separation block.

Music Source Separation:

- Processes audio and listener's audiogram.
- Outputs separated data (VDBO) for Audio Remixing.

Audio Remixing:

- Remixes separated audio.
- Sends remixed stereo audio to the Evaluation (HAAQI) stage [7].

Evaluation (HAAQI):

- Assesses remixed audio quality.
- Directs audio data to the Data Handling block.

Data Handling:

- Manages processed data for output.
- Converts data into .flac format in the Output Audio block [22].

Output Audio (.flac format):

- Terminal block, producing high-quality audio for users.

The below diagram shows how user input undergoes various stages to create a personalized, high-quality audio experience.

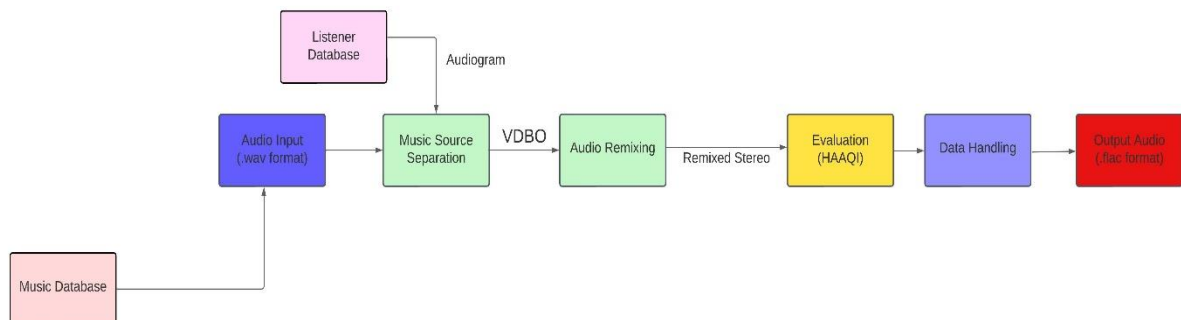


Figure 3.1: High Level Block Diagram of the System

3.2.2. Detailed Flow Diagram

Audio Input and Preprocessing:

The stereo audio files in .wav format are fed into the system.

Demixing Process:

At the heart of the demixing stage lie three sophisticated models - Hybrid Demucs, Open-Unmix, and Spleeter [34, 10, 11]. It's essential to note that only one of these models is utilized at any given time for the demixing process. These models are meticulously designed to dissect the audio into its constituent stems with a high degree of precision. The resulting output comprises eight distinct stem files. For each ear, listeners will discern the isolated sounds of

vocals, drums, bass, and other instrumental elements. This separation not only amplifies the clarity of individual sounds but also provides the flexibility for subsequent personalized remixing.

Audio Quality Evaluation:

Once separated, the HAAQI scores are calculated for the separated stem audio files to ensure audio fidelity [7].

Remixing Process:

This step incorporates:

- Applying NAL-R filters [13].
- Processing through a Butterworth bandpass filter [14].
- Subjecting to dynamic range compression [15].

Final Processing and Output:

After the remixing process is complete, the audio files remain in their .wav format. But there's one more step before they're ready for listeners. To ensure the best audio quality while also being mindful of file size, these files are converted to the .flac format [22]. The beauty of .flac is that it's a lossless format, meaning it keeps all the original audio quality intact but in a more compact size compared to the bulky .wav files. So, listeners get top-notch sound quality without the heavy files.

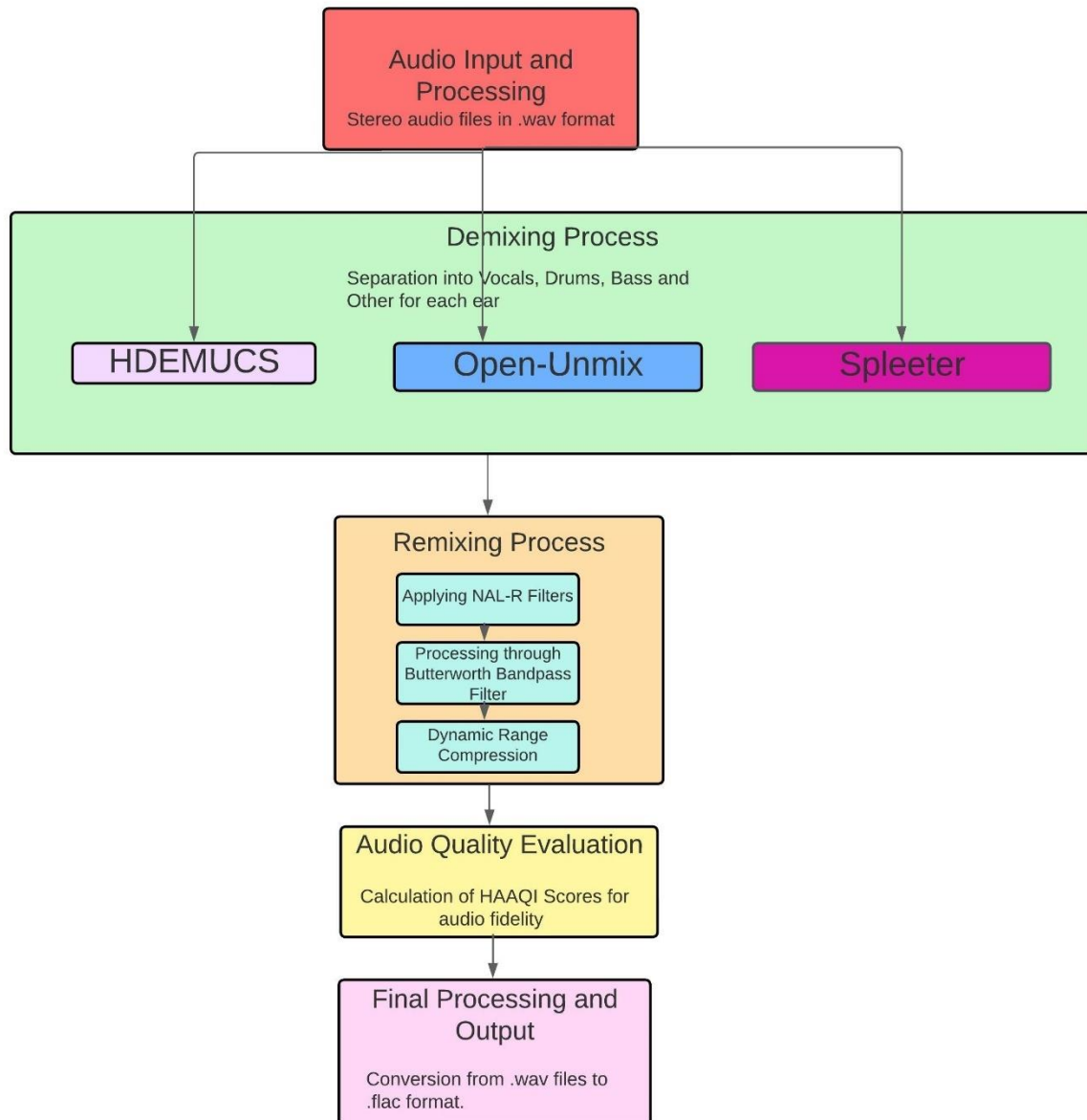


Figure 3.2: Detailed Flow Diagram of the system

3.2.3. Model Architecture Overview

In our pursuit to perfect audio separation and achieve impeccable sound clarity, we harness the capabilities of three advanced models. Each model, with its unique architecture and strengths, contributes to the system's overall performance. Here's a brief overview:

Hybrid Demucs:

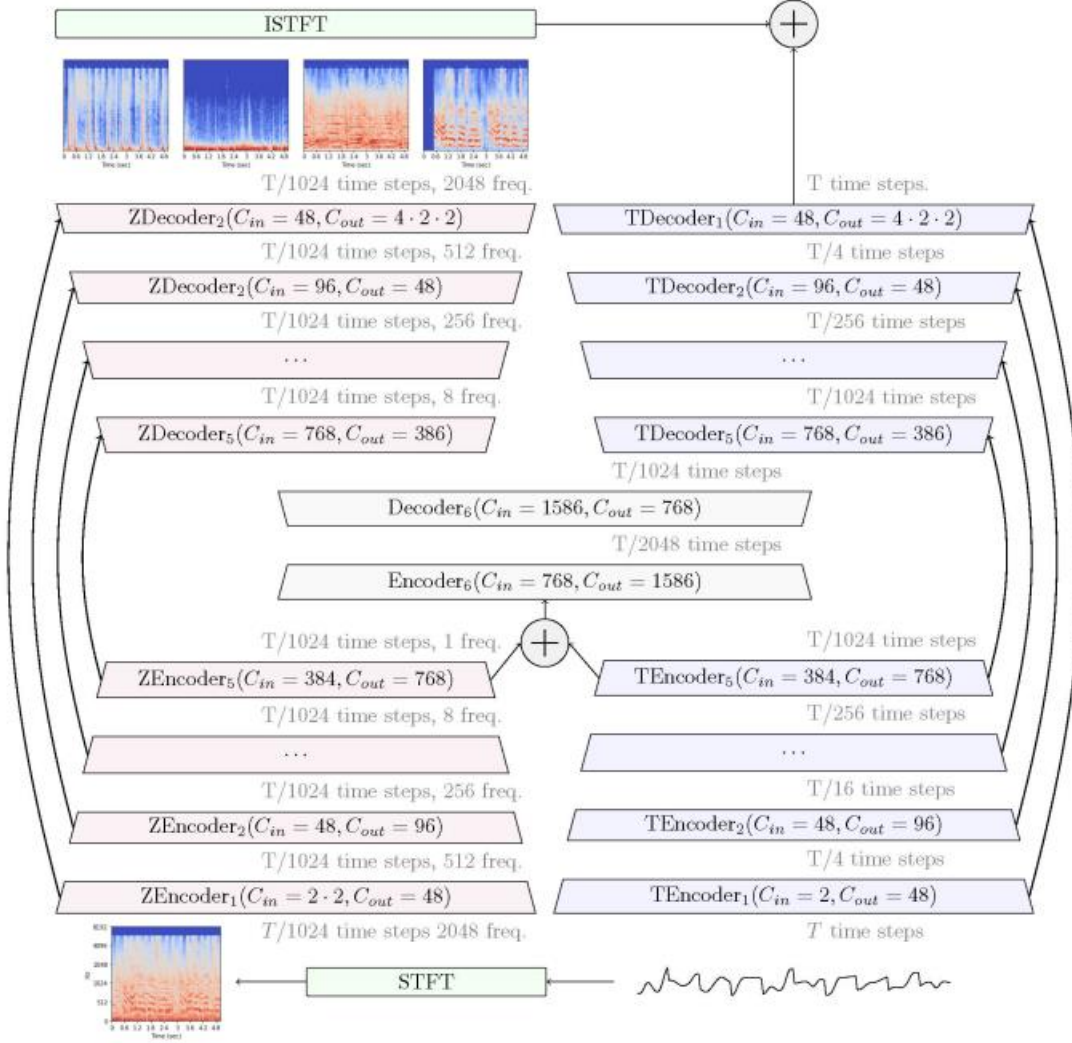


Figure 3.3: Architecture of Hybrid DEMUCS Model [20].

Deep Mind Unsupervised Source Separation (Demucs) is an unsupervised deep learning architecture aimed at source separation tasks [8]. It leverages both convolutional and recurrent neural networks to learn representations and patterns required to separate complex signals like music tracks. The architecture consists of:

- Multiple encoder-decoder pairs stacked in a U-shape to allow for efficient learning and mixing of low to high-level representations.
- Dilated convolutions to capture patterns in different resolutions.
- Bi-directional LSTM to capture the temporal dependencies in the signals.
- Skip connections to aid faster convergence and improved information flow.

Hybrid Demucs is an advanced version of Demucs [34]. It incorporates the structure of Demucs but also includes classical source separation techniques like spectral subtraction and phase manipulation to improve the accuracy and robustness of separation. It consists of the following layers:

- **Input Layer:** This layer receives a mixed signal.
- **Encoder Stack:** Several layers progressively downsample and increase depth.
- **Bi-directional LSTM:** Captures temporal patterns and dependencies.
- **Decoder Stack:** Mirrors the encoder but upsample, reducing depth to produce separated signals.
- **Output:** Multiple channels of separated signals.

Open-Unmix:

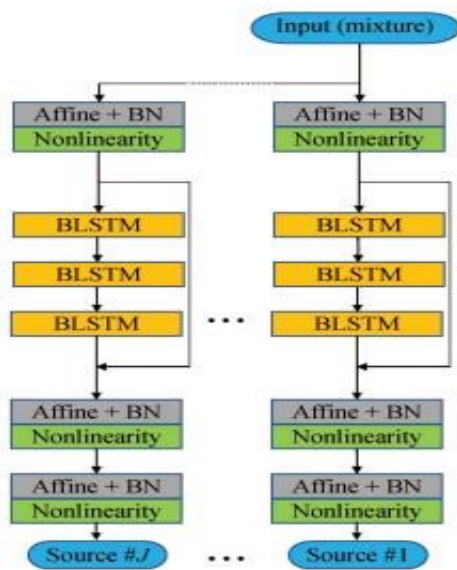


Figure 3.4: Architecture of Open-Unmix Model [20]

Open-Unmix is a deep neural network model designed for music source separation [10]. Here are its distinctive features:

- **Simplicity:** Unlike complex architectures, Open-Unmix uses a simple 3-layer bidirectional LSTM network.
- **Extensibility:** Open-Unmix is designed to be adaptable. It provides a good starting point for researchers to build upon.

- **Built-in Wiener Filtering:** This technique helps in enhancing the separation quality by leveraging the power spectral densities of the sources and the mixture.
- **Flexible Front ends:** Open-Unmix supports various input representations like magnitude spectrograms, mel-band magnitudes, and others.
- **Uses Side Information:** By estimating spectral patterns like common amplitude modulation, it improves separation performance.

The model consists of the following layers:

- **Input Layer:** Receive mixed music signal.
- **Spectrogram Conversion:** Convert time-domain signal to frequency-domain representation.
- **Deep Bi-directional LSTM:** Process the spectrogram to identify and separate sources.
- **Wiener Filtering Block:** Enhance separation using the estimated power spectral densities.
- **Inverse Spectrogram Conversion:** Convert back to the time domain.
- **Output:** Extracted source signals.

Spleeter:

Developed by Deezer, Spleeter is renowned for its impressive speed without compromising on separation quality [12]. While it's not as deep as some other architectures, Spleeter's design is optimized for real-time processing, making it an invaluable asset when quick turnaround is paramount. The model employs a deep U-Net architecture which allows for the effective separation of multiple sound sources. The reason for Spleeter's integration into our pipeline is twofold: its capability to deliver rapid results and its consistent performance in handling a diverse range of audio tracks.

Together, these models form the backbone of our audio separation process, each bringing its unique strengths to the table to ensure listeners are provided with an unparalleled auditory experience.

Chapter 4: Audio Processing Implementation

In our project, we're not just chasing top-notch audio quality. We're showing how smart algorithms can mesh well with solid software groundwork. This chapter breaks down all the steps we took in building our audio processing flow. We kick off with a look at the tools we used, like Google Colab, and why we chose Python as our go-to language. We then get into the nitty-gritty, diving into the different libraries and models we used. There's a bit of everything here: how we prepared our data, the methods we used for demixing and remixing, and how we ensured the final audio quality was up to par. To make things clearer, we've included screenshots, bits of our actual code, simple versions of our logic, and some helpful diagrams. By the end, there should be a clear picture of what went into making our implementation of "The Cadenza Challenge" a success [1].

4.1 System Setup and Software Environment

Our endeavour to achieve a refined audio processing pipeline necessitated a robust system setup. By considering various platforms and languages, we managed to strike a balance between flexibility, power, and ease of use. Here's a closer look at our choices:

4.1.1 Development Environment: Google Colab

Introduction to Google Colab

Google Colab, or "Colaboratory," is a free cloud service by Google. It's like a blend between a notebook and a cloud-based platform where we can write and execute Python code. But it's not just about writing code; Colab is also a space where we can mix in some text, equations, and even visualizations, making it perfect for our project documentation.

Why Google Colab?

Choosing Google Colab was no accident. Our project demanded substantial computational resources, especially when dealing with deep learning models. With Colab, we got access to GPUs and TPUs, which sped up our calculations considerably. Plus, the real-time collaboration feature allowed our team to work simultaneously, making debugging and collaborative coding a breeze.

Benefits of Cloud-based Environments

When we're working on large-scale audio processing tasks like ours, local systems might fall short. Cloud-based platforms like Colab come to the rescue in multiple ways:

- **Scalability:** Easily scale up or down based on the task's demands without any hardware limitations.
- **Collaboration:** Team members can work on the same notebook, ensuring seamless collaboration and consistency.
- **No Installation Hassles:** All necessary libraries and dependencies are just a click away. No lengthy installation processes or compatibility issues.
- **Accessibility:** Work from anywhere, anytime. All we need is a browser.

4.1.2 Programming Language: Python

Why Python?

When it came to picking a language, Python stood out for various reasons. Firstly, it's well-known for being beginner-friendly, but that's not why we chose it. For our audio processing tasks, Python's clean syntax made it easier to write and understand the code. More importantly, the vast ecosystem of libraries available, specifically tailored for audio processing and deep learning, made it an obvious choice.

Python's Versatility and Libraries

Python isn't just versatile; it's like a Swiss Army knife for developers. Whether it's data manipulation, visualization, or hardcore number crunching, Python's got a tool for that. In our project, libraries like NumPy, SciPy, and TensorFlow proved invaluable [23, 24, 27]. They not only accelerated our development process but also brought in the reliability of community-tested functions. This rich library support, combined with Python's ease of use, made our development process smoother and more efficient.

4.2 Essential Libraries and their Roles

In our project, we leveraged several libraries that facilitated various stages of the audio processing pipeline. Below is a list of essential libraries we used, along with a brief description of their utility:

- *NumPy*: Utilized for numerical operations and data manipulation, making array handling in audio data seamless [23].
- *SciPy*: Especially employed for its wavfile and signal processing functionalities, it's a core library for reading, writing, and filtering audio files [24].
- *Soundfile*: Essential for reading and writing sound files in different formats; it ensures the compatibility of audio files within our system [25].
- *Spleeter*: A specialized library designed for the demixing process; its Separator and AudioAdapter classes are crucial in our audio separation stage [11].
- *Pathlib*: Assists in managing file and folder operations, providing a more object-oriented and intuitive approach to handling paths and directories [26].
- *TensorFlow*: Powers the deep learning models in our project, such as Spleeter, providing a robust platform for building and training neural networks [27].
- *JSON & OS*: JSON helps in data structuring, while the OS library is used for system-level operations, together simplifying data handling and directory management [28, 29].

These libraries, when combined, provide a solid foundation for our project, ensuring efficiency, reliability, and maintainability of the codebase. Their selection was not arbitrary but based on specific needs and requirements of the various stages of the audio processing workflow.

4.3 Data Preparation

Ensuring the precision and integrity of the data preparation process is paramount. The datasets chosen were meticulously prepared to facilitate robust model training, validation, and testing.

4.3.1 Training Dataset

The decision to employ MUSDB18-HQ as the foundational dataset was strategic, keeping in mind its comprehensiveness and the diversity it brings to the table [21]. The specifics are as follows:

86 training songs: The selection of these tracks was carried out after a careful evaluation of their diversity in terms of genre, tempo, and instrumentation. Serving as the crux of the training phase, these tracks were instrumental in calibrating the model for varied audio profiles.

14 validation songs: These tracks, distinguished from the training set, were employed in interim model evaluations. Their primary purpose was to gauge the model's progress and recalibrate parameters during the iterative training cycle.

Complementing MUSDB18-HQ, several additional datasets were annexed:

Bach10: Renowned for its classical compositions, this dataset enriched the training pool with intricate harmonic structures.

FMA-small: Added to introduce a broader range of genres and ensure the model's versatility.

MedleydB versions 1 & 2: While integrating these, special attention was given to avoid redundancy, as some tracks overlapped with the MUSDB18-HQ set [21]. The rules section on The Cadenza Challenge Website provided clear guidelines on the optimal utilization of these tracks [1].

4.3.2 Validation Dataset

Validation is a crucial step in machine learning. The songs curated for this set:

Primarily sourced from the 14 songs of MUSDB18-HQ. These tracks were of varied complexities, ensuring a holistic model evaluation.

Additional tracks from Bach10, FMA-small, and MedleydB were selectively incorporated, depending on their compatibility with the core dataset and the diversity they added.

4.3.3 Testing Dataset

Meticulous testing ensures the robustness of the model in real-world scenarios. Procedures adopted were:

- **Complete Songs Processing:** Every song underwent a thorough analysis, ensuring comprehensive feedback for the final model evaluation.
- **HAAQI Evaluation:** This evaluation metric was pivotal in quantifying the model's performance across tracks, establishing its strengths and areas of improvement [7].
- **Listening Panel Evaluation:** Human feedback is invaluable. From the larger set, specific tracks were shortlisted. These tracks underwent another layer of scrutiny, where 15-second random samples were isolated and presented to a panel of listeners. Their feedback added a qualitative dimension to the model's evaluation.
- **Data File Structure:** An organized hierarchy was crucial. Nine output signals emerged from the enhancement algorithm for each song – catering to both left and right channels of every stem (vocal, bass, drums, others), with an additional signal dedicated to the final remix. This intricate layering of data ensured a granular evaluation at every step.

- **Metadata:** Each track was accompanied by a rich metadata profile, encapsulated in JSON format. This profile not only facilitated track identification but also became instrumental in understanding the track's origin, licensing agreements, and its categorization within the broader project framework.
- **Listener Metadata:** By harnessing audiograms from the Clarity project and the von Gablenz and Holube dataset, the project stood at the intersection of audio engineering and human auditory science [32]. These audiograms painted a vivid picture of individual hearing thresholds. It was a monumental stride toward customizing audio enhancements to resonate with the unique auditory profiles of listeners.

4.4 Implementation of Hybrid Demucs and Open-Unmix

4.4.1 Overview:

In our project, we've combined the strengths of the Hybrid Demucs and Open-Unmix models [34, 10]. Our aim is to effectively separate different components in an audio file. This section delves into the core functionalities and advantages of the `enhance.py` script. This pivotal script is where we've integrated the Hybrid Demucs and Open-Unmix to achieve effective audio separation. Below, are select excerpts from the code.

```
def separate_sources(
    model: torch.nn.Module,
    mix: torch.Tensor,
    sample_rate: int,
    segment: float = 10.0,
    overlap: float = 0.1,
    device: torch.device | str | None = None,
):
    """
    Apply model to a given mixture.
    Use fade, and add segments together in order to add model segment by segment.

    Args:
        model (torch.nn.Module): model to use for separation
        mix (torch.Tensor): mixture to separate, shape (batch, channels, time)
        sample_rate (int): sampling rate of the mixture
        segment (float): segment length in seconds
        overlap (float): overlap between segments, between 0 and 1
        device (torch.device, str, or None): if provided, device on which to
            execute the computation, otherwise `mix.device` is assumed.
            When `device` is different from `mix.device`, only local computations will
            be on `device`, while the entire tracks will be stored on `mix.device`.

    Returns:
        torch.Tensor: estimated sources

    Based on https://pytorch.org/audio/main/tutorials/hybrid\_demucs\_tutorial.html
    """
    device = mix.device if device is None else torch.device(device)
    mix = torch.as_tensor(mix, device=device)

    if mix.ndim == 1:
        # one track and mono audio
        mix = mix.unsqueeze(0)
    elif mix.ndim == 2:
        # one track and stereo audio
        mix = mix.unsqueeze(0)

```

Fig 4.1: Code Excerpt showing implementation of HDEMUCS and Open-Unmix

```

# one track and stereo audio
mix = mix.unsqueeze(0)

batch, channels, length = mix.shape

chunk_len = int(sample_rate * segment * (1 + overlap))
start = 0
end = chunk_len
overlap_frames = overlap * sample_rate
fade = Fade(fade_in_len=0, fade_out_len=int(overlap_frames), fade_shape="linear")

final = torch.zeros(batch, 4, channels, length, device=device)

while start < length - overlap_frames:
    chunk = mix[:, :, start:end]
    with torch.no_grad():
        out = model.forward(chunk)
    out = fade(out)
    final[:, :, :, start:end] += out
    if start == 0:
        fade.fade_in_len = int(overlap_frames)
        start += int(chunk_len - overlap_frames)
    else:
        start += chunk_len
    end += chunk_len
    if end >= length:
        fade.fade_out_len = 0

return final.cpu().detach().numpy()

```

Fig 4.2: Code Excerpt showing implementation of HDEMUCS and Open-Unmix

4.4.2 Functions of the Script:

Audio Source Separation: At its core, the script identifies and isolates various sounds within an audio file. This ensures that instruments or vocals are clear and distinct when played back.

Organizing and Structuring the Audio: Once the audio is separated, the script organizes these distinct sounds. A noteworthy capability is its ability to break down an audio signal into 8 separate parts, termed "stems". This structured approach provides a foundation for more detailed audio adjustments later in the process.

Customizing Audio for Specific Listeners: An important feature of our project is its adaptability to different listeners. We've incorporated the NAL-R algorithm into the script, an algorithm designed to modify audio, making it more accessible to those with hearing challenges [13].

Refining the Final Output: After the separation and customization processes, the script takes on the task of refining the audio. It ensures that any imperfections are addressed, delivering a high-quality audio output.

4.4.3 Libraries and Dependencies:

Various libraries like json, logging, numpy, pandas, and torch are imported. Notably, torchaudio is used, which is a library that loads and processes audio files with TorchScript compatibility.

4.4.4 Functions and Functionalities:

- i. `separate_sources`:
 - This function applies the audio source separation model to an audio mix. The model processes the audio segment by segment.
 - The fade-in and fade-out effects between segments ensure smoother transitions.
 - It returns the estimated sources of the audio.
 - The implementation seems inspired by a PyTorch tutorial, indicating a standard method for this type of processing.
- ii. `get_device`:
 - Determines the hardware device on which the processing will occur, either a CPU or a specific GPU.
- iii. `map_to_dict`:
 - Maps the separated audio sources to a dictionary format where each source is separated into left and right channels. For instance, an audio track containing vocals would have two entries: `left_vocal` and `right_vocal`.
- iv. `decompose_signal`:
 - This function breaks down an audio signal into 8 stems.
 - The breakdown depends on the model chosen (`demucs` or `openunmix`).
 - The audio signals are resampled if required.
 - The separated sources are then normalized and mapped to a dictionary format.
- v. `apply_baseline_ha`:
 - This function applies the NAL-R (National Acoustic Laboratories' Revised) prescription hearing aid algorithm to a given audio signal [13].
 - This is essentially an algorithm used to enhance audio for individuals with hearing impairment, providing a personalized listening experience.
- vi. `process_stems_for_listener`:
 - It processes the separated audio stems using the NAL-R prescription.
 - It caters to specific listeners, adjusting the audio sources according to their hearing profile.
- vii. `clip_signal`:
 - Once audio processing is done, this function ensures that the signal values lie within a certain range to avoid distortion. If any values are outside this range, they are "clipped" to fit within it.

- It supports both hard clipping (truncation) and soft clipping (a smoother, more rounded clipping).
- viii. `to_16bit`:
- This function scales the signal and converts it to 16-bit, which is a standard format for audio files.
- ix. `enhance`:
- This is the main function that drives the whole script.
 - It begins by setting up directories and loading models (demucs or openunmix) for source separation.
 - The audio files and corresponding listener profiles are loaded.
 - For each audio file, the script separates the audio into its sources, processes each source according to the listener's hearing profile, clips and scales the processed audio, and then saves the result.

4.4.5 Model Training and Application:

- **Model Selection:** Depending on the configuration, the script can either use the Hybrid Demucs model or the Open-Unmix model for source separation [34, 10]. The Demucs model seems to come from the torchaudio.pipelines, while the Open-Unmix model is loaded from torch.hub.
- **Normalization:** Before feeding into the Demucs model, the audio is normalized. This step scales the audio signal to a range suitable for processing.
- **Model Application:** The model is applied to the audio through the `separate_sources` function. This processes the audio segment by segment and separates the sources.

4.5 Spleeter Implementation

4.5.1 Overview:

Spleeter, developed by Deezer, is a pre-trained deep learning model tailored for music source separation [11]. In this project, we utilize Spleeter not just for its out-of-the-box capabilities but also leverage its extensibility by training it on a custom dataset. The goal is to achieve enhanced results when isolating individual audio components from our specific music collection.

4.5.2 Data Preparation and Training

Before jumping into the separation process, it is pivotal to understand the dataset on which our model is trained. Our training set is comprised of already separated audio components,

such as bass.wav, drums.wav, vocals.wav, and other.wav. This structured data ensures Spleeter is exposed to the specific nuances and patterns present in each music component, facilitating better learning [11].

```
spleeter train -p spleeter:4stems -d /content/drive/MyDrive/Cadenza/CadenzaData/task1/core_audio_data/cadenza_data/cad1/task1/audio/musdb18hq
```

Fig 4.3: Code Excerpt showing training phase of Spleeter model

This command tells Spleeter that we're targeting a 4-stem separation and provides the path to our custom training dataset.

4.5.3 Spleeter's Core Implementation:

The primary function in our Spleeter implementation is the `process_song()` [11]. This function, at its essence, accomplishes the following:

- **Song and Listener Data Extraction:** The song and listener metadata are read from specified directories. These details provide important contextual information about the audio to be processed and the characteristics of the listener.
- **Separation with Spleeter:** Rather than utilizing the default Spleeter model, we leverage our custom-trained model for better alignment with our dataset. This ensures the application of our dataset-tailored model for the audio separation tasks.

```
separator = Separator('/content/model')
```

Fig 4.4: Selecting trained Spleeter model

- **Applying Listener Characteristics:** Once the audio has been separated, each stem undergoes processing based on specific listener characteristics. These characteristics could include hearing profiles or preferences, which are then utilized to enhance the listening experience.
- **Signal Filtering:** Post-separation and application of listener characteristics, the signal is then filtered using a Butterworth bandpass filter to ensure that the frequency range is within human audibility and to further refine the audio quality.
- **Output File Creation:** The processed audio, now separated into individual stems, is saved in the FLAC format [22]. FLAC was chosen due to its lossless compression capabilities, ensuring that the quality of audio remains intact.

4.5.4 Code Highlights:

- Initial lines include installation and import statements, ensuring all required libraries and dependencies are in place.
- Metadata, concerning listener characteristics and audio segments, are loaded using the json library.
- The heart of the Spleeter implementation is the instantiation of the separator object, marking the initialization of our custom-trained model.
- The process_song() function captures the main workflow, orchestrating the various sub-functions for audio separation and subsequent processing.

Below are a few excerpts from the code:

```
def process_song(listener_id, song_dir, song_file):
    # Extract song_name from song_file
    song_name = os.path.basename(song_dir)

    listener_characteristics = all_listener_characteristics[listener_id]
    audio_segments = all_audio_segments[song_name]

    # Create Separator and process song
    audio_loader = AudioAdapter.default()
    waveform, _ = audio_loader.load(os.path.join(song_dir, song_file), sample_rate=44100)
    audio_descriptor = separator.separate(waveform)

    # Apply listener characteristics to each stem
    remixed_stems = {}
    remixed_signal = np.zeros_like(waveform) # Initialize remixed_signal
    for stem, stem_signal in audio_descriptor.items():
        remixed_stem = apply_listener_characteristics(stem_signal, listener_characteristics)
        remixed_stems[stem] = remixed_stem
        # Perform padding operation
        if remixed_stem.shape[0] < remixed_signal.shape[0]:
            padding = np.zeros((remixed_signal.shape[0] - remixed_stem.shape[0], remixed_signal.shape[1]))
            remixed_stem = np.concatenate((remixed_stem, padding), axis=0)
        elif remixed_stem.shape[0] > remixed_signal.shape[0]:
            padding = np.zeros((remixed_stem.shape[0] - remixed_signal.shape[0], remixed_signal.shape[1]))
            remixed_signal = np.concatenate((remixed_signal, padding), axis=0)
        remixed_signal += remixed_stem # Add each processed stem to the remixed_signal
```

Fig 4.5: Spleeter Demixing and remixing techniques

4.6 Remixing: Filters and Techniques

4.6.1 NAL-R Filters

Introduction to NAL-R filters and their significance in the remixing phase:

The NAL-R (National Acoustic Laboratories' Revised) prescription is an algorithm primarily designed for configuring hearing aids to optimize speech understanding [13]. In this context, the NAL-R filter's role is to reshape the audio content based on the listener's hearing profile, thus creating a personalized listening experience. By doing so, listeners can experience a more vivid and clear sound even if they have hearing impairments.

Code Implementation:

- **Audiogram Construction:** Audiograms represent a listener's hearing capability at different frequencies. In this code, the audiograms for both left and right ears are constructed based on the listener's hearing levels and centre frequencies. This is crucial as it provides the baseline data on which the NAL-R filter adjustments are based.

```
# Create Audiogram objects for the left and right ears
audiogram_left = Audiogram(listener_data["audiogram_levels_l"], listener_data["audiogram_cfs"])
audiogram_right = Audiogram(listener_data["audiogram_levels_r"], listener_data["audiogram_cfs"])
```

Fig 4.6: Construction of Audiograms

- **NAL-R Filter Construction:** Using the audiograms, the NAL-R filter is built for both ears. This filter will subsequently be used to adjust the audio signal to match the listener's hearing profile.

```
# Create a NALR object
nalr = NALR(nfir=220, sample_rate=44100) # Assuming a sample rate of 16kHz

# Build NAL-R FIR filters for the left and right ears
nalr_left, _ = nalr.build(audiogram_left)
nalr_right, _ = nalr.build(audiogram_right)
```

Fig 4.7: NALR filter construction

- **Filter Application:** With the filters built, they are applied directly to the left and right audio channels, modifying the audio content for the specific listener.

```
# Apply the NAL-R filters to the signals
processed_signal_left = nalr.apply(nalr_left, signal_left)
processed_signal_right = nalr.apply(nalr_right, signal_right)
```

Fig 4.8: NALR filter application

4.6.2 Butterworth Bandpass Filter

Detailed look into the Butterworth bandpass filter:

Butterworth filters are celebrated for their flat frequency response in the passband, which means they don't introduce ripples or distortions in the frequencies that are allowed to pass through [14]. A bandpass filter specifically allows frequencies within a certain range to pass through while attenuating those outside this range. In audio processing, such filters can be

used to eliminate unwanted noise or to focus on specific frequencies that enhance the overall listening experience.

Code Implementation:

- **Filter Design:** This section designs the Butterworth bandpass filter. It calculates normalized low and high cut-off frequencies and designs the filter based on the specified order. A higher order means a steeper roll-off, giving a sharper transition between the passband and stopband.

```
def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    sos = butter(order, [low, high], btype='band', analog=False, output='sos')
    return sos
```

Fig 4.9: Butterworth bandpass filter definition

- **Filter Application:** Once the filter is designed, it's applied to the audio data. This modifies the audio content, allowing only the desired frequency range to pass through.

```
def apply_butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    sos = butter_bandpass(lowcut, highcut, fs, order=order)
    y = sosfilt(sos, data)
    return y
```

Fig 4.10: Butterworth bandpass filter application

4.6.3 Dynamic Range Compression

Explanation of dynamic range compression and its importance:

Dynamic range refers to the difference between the quietest and loudest parts of an audio signal. In some cases, this range can be vast, leading to issues like the listener constantly adjusting the volume. Dynamic range compression narrows this range by attenuating the loudest parts and/or amplifying the quieter ones, ensuring a more consistent listening experience. It's especially crucial in scenarios like watching movies where explosive action scenes can be drastically louder than dialogue sequences [15].

Code Implementation:

- **Compressor Initialization:** The code starts by initializing a dynamic range compressor object, set with a particular sample rate. This object is designed to process audio signals and apply compression based on certain parameters.

```
# Instantiate Compressor and apply to signals
compressor = Compressor(fs=44100.0) # Assuming a sample rate of 44100Hz
```

Fig 4.11: Compressor initialization

- **Compression Application:** The compressor processes both audio channels (left and right). This ensures the dynamic range of the audio is compressed, providing a more consistent volume level throughout the audio track.

```
processed_signal_left, _, _ = compressor.process(processed_signal_left)
processed_signal_right, _, _ = compressor.process(processed_signal_right)
```

Fig 4.12: Compressor Application

4.6.4 Note on External Dependencies: It's important to mention that the above code relies on classes and methods that aren't explicitly defined within the provided snippets. These classes and methods come from external files, ensuring modular and maintainable code architecture. Here's a brief overview:

- **NALR Object:** This crucial object, which facilitates the construction and application of the NAL-R filter, is imported from a file named `nalr.py`. This file encapsulates the logic and mathematics behind the NAL-R prescription and its application to audio signals [13].
- **Compressor Object:** The dynamic range compression is achieved using the Compressor object, which is instantiated from the `compressor.py` file. This file contains the compression algorithm and the parameters that determine how the compression is applied to an audio signal.

In the following section, we will dive deeper into these files, providing a closer look at the classes, methods, and the logic encapsulated within them. This will offer a comprehensive understanding of the backbone operations that drive the audio processing workflow described above.

4.7 Auxiliary Files Used

In the preceding sections, we have alluded to several external dependencies that underpin the operations of our primary codebase. In this section, we will offer a comprehensive exploration of these auxiliary files.

4.7.1 Audiogram Class Implementation

The audiogram.py is one of the primary scripts employed for our processing. It comprises a data class to represent a monaural audiogram. This audiogram encapsulates the listener's audiometric levels at various frequencies.

The structure of the Audiogram Class:

- `DEFAULT_CLARITY_AUDIOGRAM_FREQUENCIES`: Constant list of default audiogram frequencies.
- `FULL_STANDARD_AUDIOGRAM_FREQUENCIES`: Constant list of complete standard audiogram frequencies.
- `Audiogram`: The main class that represents the audiogram. Contains methods like `severity` for categorizing hearing loss severity and `resample` for resampling the audiogram to new sets of frequencies.
- `Reference Audiograms`: These are predefined audiograms like `AUDIOGRAM_REF`, `AUDIOGRAM_MILD`, `AUDIOGRAM_MODERATE`, etc. which are used as standard references for different types of hearing losses.
- `Listener`: Represents a listener defined by their left and right ear audiogram. It provides methods for reading metadata files and offers basic validation.

In the Spleeter separation, this class allows us to apply specific listener characteristics to the signal being processed.

4.7.2 Signal Processing Utilities

The `signal_processing.py` script provides a suite of signal processing utilities:

- `compute_rms()`: Computes RMS (Root Mean Square) of a given signal.
- `denormalize_signals()`: Scales signals back to their original scale.
- `normalize_signal()`: Standardizes a signal to have zero mean and unit variance.
- `resample()`: Resamples a signal to a new sample rate. This method provides support for multiple resampling techniques like SOX Resampler (`soxr`), Polyphase Resampling (`polyphase`), and FFT based Resampling (`fft`).

In the Spleeter process, this script plays a pivotal role in resampling audio to 24000 Hz [11]. The audio data is resampled, normalized, and saved in the FLAC format for further processing as shown in the figure below [22]:

```
# Resample audio to 32000 Hz
remixed_signal_filtered_resampled = resample(remixed_signal_filtered_subj, 44100, 32000)
```

Fig 4.13: Resampling of remixed audio

But the integration doesn't end with Spleeter. The utilities from `signal_processing.py` are deeply woven into our implementation of the Hybrid Demucs and Open-Unmix algorithms [34,10].

For instance, consider the function `decompose_signal()`. This function, dedicated to decomposing an audio signal into multiple stems, primarily relies on the model—either Demucs or Open-Unmix.

For the Demucs model, the signal is normalized before being processed. This normalization helps ensure consistent results across different audio pieces, irrespective of their original amplitude ranges. Similarly, if the sample rate of our audio signal doesn't match the model's sample rate, we use the `resample()` method to align them. Once the sources have been separated, if we used the Demucs model, the signal is denormalized back to its original scale using the `denormalize_signals()` function.

By strategically integrating these utilities, we ensure that our hybrid model operates seamlessly, benefitting from both the individual strengths of Demucs and Open-Unmix, and the robust signal processing functions.

Here's a snippet showcasing the integration:

```
if config.separator.model == "demucs":
    signal, ref = normalize_signal(signal)

model_sample_rate = (
    model.sample_rate if config.separator.model == "openunmix" else 44100
)

if sample_rate != model_sample_rate:
    resampler = Resample(sample_rate, model_sample_rate)
    signal = resampler(signal)

sources = separate_sources(
    model, torch.from_numpy(signal), sample_rate, device=device
)
# only one element in the batch
sources = sources[0]
if config.separator.model == "demucs":
    sources = denormalize_signals(sources, ref)

signal_stems = map_to_dict(sources, config.separator.sources)
return signal_stems
```

Fig 4.14: Usage of signal processing functions in HDEMUCS/Open-Unmix Implementation

4.7.3 NALR Prescription Tool

The `nalr.py` file encapsulates the workings of the NAL-R (National Acoustic Laboratories' Revised) prescription, a notable algorithm for hearing aid fitting [13]. Let's break down its components and function:

- **Imports & Constants:** Basic foundational components are imported to ensure smooth operations. A unique list `NALR_FREQS` provides frequencies typically used in the NAL-R prescription.
- **Class Definition:** The `NALR` class is the main protagonist, designed to handle the intricacies of the NAL-R prescription.
- **Initialization:** The class is initialized with the order of the NAL-R EQ filter (`nfir`) and the sampling rate. A 'flat' delay filter with equivalent delay to the NAL-R filter is designed, ensuring that any audio processed has consistent timing.
- **Filter Construction:** The `build` method designs the NAL-R FIR filter based on a given audiogram. It computes the necessary gains at specified frequencies, considering the hearing loss at those frequencies. The design then extrapolates these gains across the full frequency range to create an FIR filter.
- **Signal Processing:** The `apply` method allows for the application of the constructed NAL-R filter onto a provided audio signal, enhancing it according to the NAL-R prescription.

Below is an excerpt from the script `'nalr.py'`:

```
from __future__ import annotations
from typing import TYPE_CHECKING

import numpy as np
import scipy
import scipy.signal

from clarity.evaluator.msbg.msbg_utils import firwin2
from clarity.utils.audiogram import Audiogram

if TYPE_CHECKING:
    from numpy import ndarray

NALR_FREQS = np.array([250, 500, 1000, 2000, 4000, 6000])

class NALR:
    def __init__(self, nfir: int, sample_rate: float) -> None:
        """
        Args:
            nfir: Order of the NAL-R EQ filter and the matching delay
            fs: Sampling rate in Hz
        """
        self.nfir = nfir
        # Processing parameters
        self.fmax = 0.5 * sample_rate

        # Design a flat filter having the same delay as the NAL-R filter
        self.delay = np.zeros(nfir + 1)
        self.delay[nfir // 2] = 1.0
```

Fig 4.15: NALR Implementation

4.7.3.1 Integration in Spleeter:

After importing the necessary components, we first instantiate the NALR class with a specified `nfir` and sampling rate. Subsequent steps involve [13]:

Building NAL-R FIR filters for both the left and right auditory channels based on respective audiograms.

The left and right channels of a signal are separately extracted and subsequently processed using the NAL-R filters.

4.7.3.2 Integration in Hybrid Demucs and Open-Unmix:

The NALR prescription, along with a compressor, can be applied to audio signals using the `apply_baseline_ha` function. This function builds the NAL-R FIR filter for an audiogram, processes the signal with it, and optionally applies compression [13].

Moreover, for a more nuanced processing on stems of sources, the `process_stems_for_listener` function is used. It discerns the correct audiogram (left or right) for each stem and processes it using the `apply_baseline_ha` function.

4.7.4 Compressor Class Implementation

Within our software, we implement a dynamic range compression technique with the Compressor class, which is housed in the `compressor.py` file. Dynamic range compression (or audio compression) is an audio signal processing operation that reduces the volume of loud sounds or amplifies quiet sounds, thus reducing or compressing an audio signal's dynamic range [15].

4.7.4.1 File Structure

The primary class is `Compressor`. Some highlights of its attributes and methods include:

Attributes:

- **fs**: Sample frequency.
- **rms_buffer_size**: Size of the buffer used to compute the root mean square (RMS) value.
- **attack**: Time it takes for the compressor to start reducing the gain once a signal has exceeded the threshold.
- **release**: Time it takes for the compressor to stop reducing the gain once a signal has dropped below the threshold.
- **threshold**: Level above which the compressor starts to reduce gain.

- **attenuation**: Degree of compression applied once the signal exceeds the threshold.
- **makeup_gain**: Amplification applied after compression.
- **win_len**: Length of the window used for computing RMS.
- **window**: A buffer filled with ones, having a length equal to win_len.

Methods:

- **set_attack**: Updates the attack rate.
- **set_release**: Updates the release rate.
- **process**: This is the primary method which applies compression to the input signal. It computes the RMS of the signal, calculates the compression ratio, and then adjusts the gain of the signal based on this ratio and the makeup_gain attribute.

Below is an excerpt from the code of the file `compressor.py`:

```
def process(self, signal: np.ndarray) -> tuple[np.ndarray, np.ndarray, list[Any]]:
    """DESCRIPTION

    Args:
        signal (np.array): DESCRIPTION

    Returns:
        np.array: DESCRIPTION
    """
    padded_signal = np.concatenate((np.zeros(self.win_len - 1), signal))
    rms = np.sqrt(
        np.convolve(padded_signal**2, self.window, mode="valid") / self.win_len
        + EPS
    )
    comp_ratios: list[float] = []
    curr_comp: float = 1.0
    for rms_i in rms:
        if rms_i > self.threshold:
            temp_comp = (rms_i * self.attenuation) + (
                (1.0 - self.attenuation) * self.threshold
            )
            curr_comp = curr_comp * (1.0 - self.attack) + (temp_comp * self.attack)
        else:
            curr_comp = self.release + curr_comp * (1 - self.release)
        comp_ratios.append(curr_comp)
    return (signal * np.array(comp_ratios) * self.makeup_gain), rms, comp_ratios
```

Fig 4.16: Compressor Class Implementation

4.8 Final File Transformation

4.8.1 Explanation for the final conversion from .wav to flac:

In the context of audio processing, storing audio files efficiently while preserving their quality is of utmost importance. FLAC (Free Lossless Audio Codec) is an audio codec that allows for

lossless audio compression [22]. This means that while the audio files are compressed to occupy less storage, no information from the original file is lost during this compression. The benefit of using FLAC over other lossless audio formats, like WAV, is that FLAC files typically require around 50-60% less storage. Therefore, the implementation seeks to transform WAV files into FLAC format for more efficient storage.

The Python script `flac_encoder.py` achieves this transformation by leveraging the 'pyflac' library. It provides the functionality to both encode WAV files into FLAC format and decode them back to their original format if necessary.

The main components of the file include:

- **WavEncoder:** A class derived from `pf.encoder._Encoder` to facilitate the conversion of WAV signals to FLAC.
- **FileDecoder:** An extension of the `pf.decoder.FileDecoder` class to convert FLAC files back to WAV format.
- **FlacEncoder:** This class serves as an interface for the encoding and decoding processes, using the above two classes for its underlying operations.

In the Spleeter implementation, the script's utility is highlighted [11]. The audio stems are processed, resampled, and encoded into FLAC format. Additionally, alongside the FLAC files, a text file is saved which contains the maximum absolute value from the signal. This value can be useful for operations like normalization in future processing tasks.

4.8.2 Code segment demonstrating this conversion:

Here is a simplified representation of how FLAC encoding is performed using the `flac_encoder.py`:

```
from clarity.utils.flac_encoder import FlacEncoder, read_flac_signal

flac_encoder = FlacEncoder()

# Write FLAC file
flac_encoder.encode(channel_data_resampled_normalized, 24000, output_path)

# Write FLAC file
flac_encoder.encode(remixed_signal_filtered_resampled_normalized, 32000, output_path) # Write with new sample rate
```

Fig 4.17: FLAC Conversion Implementation

Chapter 5: Evaluation of Audio Demixing and Remixing

5.1 Introduction

5.1.1 Brief Overview of the Chapter's Purpose:

In this chapter, we dive deep into the evaluation methods we utilized for our project. We didn't just stick to numbers; we paired objective metrics with the subjective experiences of listeners. We wanted to see not only how our audio demixing models performed on paper, but also how they resonated with real-world listeners.

5.1.2 Importance of the Evaluation:

This project isn't just about getting the math right – it's about ensuring a genuinely improved listening experience. By marrying objective measures with listener feedback, we get a 360-degree view of our models' efficacy. This comprehensive approach underscores the very essence of the challenge: to make certain our models don't just work in theory, but thrive in practice, catering to a broad spectrum of listeners.

5.2 Objective Evaluation Metrics

5.2.1 Description and Rationale for Choosing Each Metric:

Objective metrics quantitatively evaluate demixing models, focusing on audio quality alignment with the original sound. Our chosen metrics are:

- **HAAQI:** Tailored for listeners with varied hearing profiles, HAAQI gauges audio quality based on the perception of those with hearing impairments [7].
- **SNR:** A standard metric, SNR assesses the clarity of demixed audio against background disturbances, ensuring signal prominence [17].
- **SDR:** By comparing the original and demixed signals, SDR measures the fidelity of the demixing, highlighting any distortions or changes [30].

5.3 Audio Signal Evaluation Procedure

The evaluation procedure assesses the impact of various auditory enhancements and processing strategies on audio signal quality. This is performed by juxtaposing original audio signals with their processed counterparts, leveraging objective metrics and the HAAQI score [7].

5.3.1 Data Collection:

- **Listener Characteristics:** Details of each listener's audiogram levels for both ears are stored in a JSON file named `listeners.valid.json`.
- **Audio Segments:** `musdb18.segments.test.json` contains information about the audio segments utilized during evaluation.

5.3.2 Song Processing:

- Load each song's audio waveform at a sample rate of 44100 Hz.
- Separate the song into its individual stems using the Spleeter, Hybrid Demucs or Open-Unmix audio separator [34, 10,11].
- Process each stem according to listener characteristics using NAL-R filters, simulating hearing adjustments. Combine the processed stems to generate the final remixed signal [13].

5.3.3 HAAQI Score Calculation:

The Hearing Aid Sound Quality Index (HAAQI) evaluates the quality of audio processed for those with hearing impairments [7].

Extract audiograms for both ears:

```
# Get Audiogram for both ears
audiogram_left = Audiogram(listener_characteristics["audiogram_levels_l"], listener_characteristics["audiogram_cfs"])
audiogram_right = Audiogram(listener_characteristics["audiogram_levels_r"], listener_characteristics["audiogram_cfs"])
```

Fig 5.1: Extraction of Audiograms for HAAQI Calculation

Calculate HAAQI scores for both the left and right channels using the `compute_haaqi` function, providing the remixed signal (processed), original waveform, audiogram data, and sample rate.

```
# Calculate HAAQI score for Left channel
haaqi_score_left = compute_haaqi(
    processed_signal=remixed_signal[:, 0], # Left channel of remixed_signal
    reference_signal=waveform[:, 0],      # Left channel of the original waveform
    audiogram=audiogram_left,
    sample_rate=44100,
)

# Calculate HAAQI score for Right channel
haaqi_score_right = compute_haaqi(
    processed_signal=remixed_signal[:, 1], # Right channel of remixed_signal
    reference_signal=waveform[:, 1],      # Right channel of the original waveform
    audiogram=audiogram_right,
    sample_rate=44100,
)
```

Fig 5.2: Left and Right HAAQI Scores Calculation

Determine the average HAAQI score for each song-listener combo, taking the mean of left and right channel scores [7].

```
# Averaging the scores
average_haaqi_score = (haaqi_score_left + haaqi_score_right) / 2
```

Fig 5.3: Average HAAQI Score Calculation

Print and save the results:

```
print(f"HAAQI Score (Left) for {listener_id} and {song_name}: {haaqi_score_left}")
print(f"HAAQI Score (Right) for {listener_id} and {song_name}: {haaqi_score_right}")
print(f"Average HAAQI Score for {listener_id} and {song_name}: {average_haaqi_score}")

# Save the average HAAQI score to a .txt file
results_dir = '/content/HAAQI_Scores' # Adjust this path if necessary
os.makedirs(results_dir, exist_ok=True) # Ensure directory exists
result_filename = os.path.join(results_dir, f"{listener_id}_{song_name}.txt")

with open(result_filename, 'w') as result_file:
    result_file.write(f"Average HAAQI Score for {listener_id} and {song_name}: {average_haaqi_score}\n")
```

Fig 5.4: Printing and saving the HAAQI Scores

5.3.4 Objective Metrics Calculation:

The difference between the original and processed signals is termed 'noise'. Based on this, we compute:

Signal-to-Noise Ratio (SNR): Measures the clarity of the processed signal by comparing the power of the original signal to the power of the noise [17].

Signal-to-Distortion Ratio (SDR): Calculates the ratio of the signal power to the distortion power in the demixed signals, indicating the signal processing's accuracy [30].

5.3.5 Audio Output Saving:

- Save processed stems and the remixed signal as FLAC files in specific directories [22].
- Resample each stem audio to 24000 Hz, normalize, and save. The maximum absolute value of the resampled audio is stored separately in a .txt file.
- Filter the remixed signal, resample it to 32000 Hz, normalize, and save it as a FLAC file. Its maximum absolute value is documented in another .txt file.

5.3.6 Final Execution:

For each song in newmusdb.test.json, if the song file exists, process the song for each listener from the new_listeners.valid.json file.

Upon completing the above processes, the system generates a total of nine .flac files for each song and listener combination [22]. These files represent the vocals, drums, bass, and other stems, individually processed for each ear (left and right). In addition to creating .flac files for each tailored audio stem, also generates corresponding .txt files. Each of these .txt files contains a single value, which represents the maximum absolute value of the resampled audio data. In simpler terms, the .txt files store the loudest point of the audio for each stem to ensure the audio doesn't peak or clip during playback. Additionally, a remixed file is generated which amalgamates these stems.

Here's the expanded breakdown of generated files with the addition of the .txt files:

FLAC Files:

- Left-ear vocals: vocals_left.flac
- Right-ear vocals: vocals_right.flac
- Left-ear drums: drums_left.flac
- Right-ear drums: drums_right.flac
- Left-ear bass: bass_left.flac
- Right-ear bass: bass_right.flac
- Left-ear other sounds: other_left.flac
- Right-ear other sounds: other_right.flac
- Combined remixed track: remixed_song.flac

TXT Files (for max audio values):

- Left-ear vocals: vocals_left.txt
- Right-ear vocals: vocals_right.txt
- Left-ear drums: drums_left.txt
- Right-ear drums: drums_right.txt
- Left-ear bass: bass_left.txt
- Right-ear bass: bass_right.txt
- Left-ear other sounds: other_left.txt
- Right-ear other sounds: other_right.txt
- Combined remixed track: remixed_song.txt

5.4 Listening Test

5.4.1. Introduction

A listening test was meticulously organized by the organizers of The Cadenza Challenge to evaluate and gather feedback on the audio quality of submissions [1].

5.4.2. Overview and Setup

Participants were provided with Sennheiser PC-8 USB headsets and instructed to operate within a quiet environment using a browser-based audio player. Post listening, feedback was solicited on a set of predefined scales.

5.4.3. Hardware and Software Specifications

The Sennheiser PC-8 USB headsets were paired with a browser-based audio player to ensure uniformity in sound quality and ease of access for participants.

5.4.4. Safety Measures

Adjustments to the audio volume were permitted to ensure participants' comfort. Consequently, the exact dB SPL varied among participants.

For clarity on the Sennheiser PC-8 USB headsets:

- When a 1-kHz pure tone, set to unity gain (RMS 0.707), was played at full volume, a level of 99 dB SPL was recorded.
- A "ICRA speech-shaped noise" produced an A-weighted level of 90 dB SPL at this volume.

5.4.5. Music Samples

From the evaluation dataset, 15-second segments were randomly selected, each accompanied by a brief fade-in and fade-out. The extraction details are available under 'Additional Tools'.

5.4.6. Test Design and Panel Composition

The test was designed to expose listeners to all team entries. Feedback was collected using 4-5 perceptual sliders, focusing on overall audio quality and specific facets like Clarity, Harshness, and Distortion. The panel, consisting of approximately 50 members, provided scores for all systems over a listening duration capped at around 5 hours per member. Audio

samples were adapted to match each listener's audiogram or a comparable type if specific audiograms were unavailable.

5.5 Discussion

5.5.1 Objective Metrics Insights:

5.5.1.1 HAAQI:

The HAAQI metric was employed for its effectiveness in assessing audio quality for listeners with diverse hearing profiles, including those with hearing impairments [7]. Below are the numerical results for each model, with specific focus on the listener IDs L5040 and L5076:

Spleeter Model:

For the track "Actions - One Minute Smile," the average HAAQI score for L5076 is 0.5636, while for L5040 it is 0.6208. These scores suggest that the Spleeter model provides better audio quality for listeners with hearing impairments in comparison to the other models [11].

Hybrid Demucs Model:

For L5076, the average HAAQI score is 0.3543, while for L5040 it is 0.2280. These scores, while lower than those of the Spleeter model, still indicate a reasonable audio quality for listeners with hearing impairments [34].

Open-Unmix Model:

For L5076, the average HAAQI score is 0.2098, while for L5040 it is 0.1145. These scores are the lowest among the models, indicating that the Open-Unmix model might not be as suitable for listeners with hearing impairments as the other two models [10].

In summary, the Spleeter model, with the highest average HAAQI scores, offers the best audio quality for listeners with hearing impairments. The Hybrid Demucs model, while offering decent audio quality, falls short of the Spleeter model. The Open-Unmix model, with the lowest average HAAQI scores, might not be the best option for listeners with hearing impairments in comparison to the other models.

5.5.1.2 SNR:

The Signal-to-Noise Ratio (SNR) provides a quantitative assessment of audio clarity by measuring the prominence of the primary signal against any background disturbances [17].

Hybrid Demucs Model:

Average SNR Achieved: L5040 (-4.21 dB), L5076 (-3.80 dB).

Open-Unmix Model:

Average SNR Achieved: L5040 (-5.85 dB), L5076 (-3.22 dB).

Spleeter Model:

Average SNR Achieved: L5040 (-2.31 dB), L5076 (-2.31 dB).

Insights:

Negative SNR values, while indicating that noise levels exceed signal levels, are an inevitability in our project focusing on enhancing audio for individuals with hearing impairment. Among the models, Spleeter has the least negative average SNR, signifying better audio clarity, while Open-Unmix has the most negative SNR, indicating lower audio clarity.

5.5.1.3 SDR: The Signal-to-Distortion Ratio (SDR) was employed to evaluate the fidelity of the demixing process. This metric reveals potential distortions or changes when the original and demixed signals are compared [30].

Hybrid Demucs Model:

Average SDR Achieved: L5040 (3.75 dB), L5076 (4.25 dB).

Open-Unmix Model:

Average SDR Achieved: L5040 (-14.43 dB), L5076 (-11.62 dB).

Spleeter Model:

Average SDR Achieved: L5040 (-38.91 dB), L5076 (-38.91 dB).

Insights:

Negative SDR values suggest that distortion introduced in the demixing process is higher than the signal, reflecting poorer fidelity in the demixing process. In the context of enhancing audio for people with hearing impairment, such distortions are an inevitable trade-off. The Hybrid Demucs model, with positive SDR values, offers the highest fidelity in the demixing process, reflecting fewer deviations from the original tracks. The Open-Unmix model, with negative SDR values, provides acceptable fidelity but might be prone to more pronounced deviations during demixing compared to the Hybrid Demucs model. The Spleeter model has the lowest average SDR, indicating the most significant deviations from the original signals among the three models.

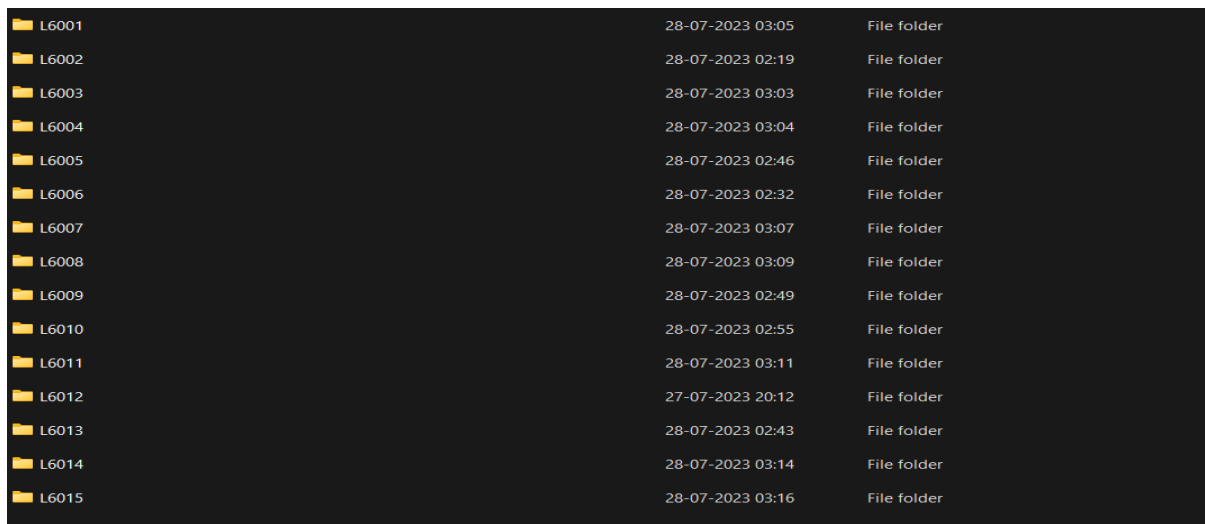
In conclusion, the Spleeter model appears to be the best choice for listeners with hearing impairments due to its higher HAAQI scores. However, if audio clarity is a priority, the Hybrid Demucs model, with its better SNR values compared to the Open-Unmix model, would be a better option. While the Open-Unmix model has the lowest average HAAQI and SNR scores, it might still be suitable for listeners without hearing impairments. It is important to consider the specific needs of the listener when choosing a model.

5.5.2 Correlation between Objective Metrics and Listening Test Results:

In-depth analysis is still underway to discern potential correlations between the objective metrics and the subjective feedback gathered from the listening tests. However, preliminary observations suggest a possible correlation between the HAAQI scores and listeners' perceptions of audio quality [7]. Further examination, particularly with the inclusion of the remaining models' scores, will offer a more comprehensive understanding of this relationship.

5.6 Final Output Screenshots

We have produced demixed stem files for both ears and created remixed files for each of the 49 tracks for all 53 listeners. The following screenshots provides a visual representation of this:



L6001	28-07-2023 03:05	File folder
L6002	28-07-2023 02:19	File folder
L6003	28-07-2023 03:03	File folder
L6004	28-07-2023 03:04	File folder
L6005	28-07-2023 02:46	File folder
L6006	28-07-2023 02:32	File folder
L6007	28-07-2023 03:07	File folder
L6008	28-07-2023 03:09	File folder
L6009	28-07-2023 02:49	File folder
L6010	28-07-2023 02:55	File folder
L6011	28-07-2023 03:11	File folder
L6012	27-07-2023 20:12	File folder
L6013	28-07-2023 02:43	File folder
L6014	28-07-2023 03:14	File folder
L6015	28-07-2023 03:16	File folder

Fig 5.5: Screenshot of all the listener output files

Al James - Schoolboy Facination	27-07-2023 18:38	File folder
AM Contra - Heart Peripheral	28-07-2023 03:01	File folder
Angels In Amplifiers - I_m Alright	27-07-2023 18:28	File folder
Arise - Run Run Run	27-07-2023 18:28	File folder
Ben Carrigan - We_II Talk About It All Ton...	27-07-2023 18:28	File folder
BKS - Bulldozer	27-07-2023 18:37	File folder
BKS - Too Much	27-07-2023 18:36	File folder
Bobby Nobody - Stitch Up	27-07-2023 18:28	File folder
Buitraker - Revo X	28-07-2023 03:01	File folder
Carlos Gonzalez - A Place For Us	27-07-2023 18:28	File folder
Cristina Vane - So Easy	28-07-2023 03:01	File folder
Detsky Sad - Walkie Talkie	27-07-2023 18:41	File folder
Enda Reilly - Cur An Long Ag Seol	27-07-2023 18:36	File folder
Forkupines - Semantics	27-07-2023 18:41	File folder
Georgia Wonder - Siren	27-07-2023 18:38	File folder
Girls Under Glass - We Feel Alright	27-07-2023 18:36	File folder
Hollow Ground - Ill Fate	27-07-2023 18:28	File folder
James Elder _ Mark M Thompson - The E...	27-07-2023 18:36	File folder

Fig 5.6: Screenshot of song tracks for each listener

Name	Date modified	Type
L6001_Al James - Schoolboy Facination_left_bass.flac	27-07-2023 18:38	FLAC File
L6001_Al James - Schoolboy Facination_left_bass.txt	27-07-2023 18:38	Text Document
L6001_Al James - Schoolboy Facination_left_drums.flac	27-07-2023 18:38	FLAC File
L6001_Al James - Schoolboy Facination_left_drums.txt	27-07-2023 18:38	Text Document
L6001_Al James - Schoolboy Facination_left_other.flac	27-07-2023 18:38	FLAC File
L6001_Al James - Schoolboy Facination_left_other.txt	27-07-2023 18:38	Text Document
L6001_Al James - Schoolboy Facination_left_vocals.flac	27-07-2023 18:38	FLAC File
L6001_Al James - Schoolboy Facination_left_vocals.txt	27-07-2023 18:38	Text Document

Fig 5.7: Screenshot of files generated for each song track

Chapter 6: Conclusions and Future Work

Section 6.1. Conclusions

Our project embarked on the ambitious endeavour of addressing one of the pivotal challenges in audio processing: enhancing audio for listeners with varying hearing capacities. The outcomes of the project, outlined below, shed light on both the achievements made and the lessons learned.

1. **Customized Audio Enhancement:** One of the project's most groundbreaking achievements was the development of an adaptive system that tailors audio enhancements to individual users based on their unique audiogram data. This holistic approach ensures that each listener experiences music in its richest form, personalized to their hearing capacities.
2. **Demixing Approaches:** We dedicated a significant portion of our research to understanding and implementing various demixing techniques. We explored models such as Hybrid Demucs, Open-Unmix, and Spleeter. Among these, the Spleeter model performed best, yielding higher HAAQI compared to the other models, suggesting superior audio quality for listeners with hearing impairments. However, it recorded negative SNR values of -2.31 dB, indicating that it still contains a fair amount of background noise [7, 34, 10, 11].
3. **Remixing Techniques:** Once the audio was demixed, the subsequent challenge was to remix it in a manner that would enhance the listening experience. Our approach combined multiple techniques, namely:
 - Applying the NALR prescription, which provided an evidence-based method to tailor the audio according to the hearing loss characteristics of listeners [13].
 - Implementing dynamic compression, allowing us to handle the wide dynamic range of music and ensuring that all elements were audible without any auditory discomfort [15].
 - Using a Butterworth bandpass filter, which ensured that any unwanted frequencies were attenuated, leading to a cleaner audio output [14].

The combined effects of these techniques culminated in a remixed audio file that not only sounded good but was also tailored to the unique needs of the listeners.

4. **Integration of Advanced Metrics:** The implementation of the HAAQI metric for evaluation stands as a testament to the project's dedication to accuracy and efficiency [7]. This metric not only allowed for a more nuanced understanding of audio quality but also presented a methodological advance over conventional evaluation methods.

5. **Listening Tests:** The decision to incorporate a browser-based listening test with a panel of approximately 50 members offered a human-centric evaluation method. The various perceptual sliders added depth to the evaluation process, ensuring that the analysis wasn't just quantitative, but also qualitative.
6. **Safety and Comfort:** The project took considerable measures to ensure listeners' safety and comfort. By allowing participants to set the overall audio volume and correlating 0 dB FS to a known SPL, the team ensured that audio enhancement did not come at the cost of discomfort.
7. **Challenges Faced:** While the project has made significant strides in audio enhancement, certain challenges were encountered. For instance, ensuring the fidelity of the enhanced audio across various song genres and listeners with varying degrees of hearing impairment posed inherent complexities.

Section 6.2. Future Work

Our project has laid a robust foundation, but like all pioneering ventures, there are avenues yet to be explored and potential areas of enhancement. Here are some recommended directions for future work:

1. **Expanding Demixing Techniques:** While Hybrid Demucs was identified as the superior model in this project, future work can look into developing custom models or exploring newer advancements in existing models such as Spleeter and Open-Unmix [8, 10, 11]. As deep learning architectures continue to evolve, so too will our ability to demix audio with increased precision.
2. **Automated Selection of Remixing Techniques:** Given that we've applied multiple remixing techniques—namely the NALR prescription, dynamic compression, and the Butterworth bandpass filter—it's worth investigating automated processes to determine the optimal combination or sequence of these techniques [13, 14, 15]. This could be based on the listener's profile, genre of the song, or even the specific mix of the track.
3. **Optimizing the Code:** Analysing the provided code, we notice that certain parts can be optimized for better performance, especially when processing multiple songs for various listeners. One area is the repetitive loading of song and listener metadata. Streamlining these processes or implementing more parallelized approaches can considerably speed up the processing time.

4. **Personalized Filtering:** While the Butterworth bandpass filter was implemented with fixed cutoff frequencies, an exploration into adaptive filtering, tailored to the listener's audiogram or the song's specific frequency spectrum, could lead to improved results.
5. **Incorporating Feedback Mechanisms:** Establishing a feedback loop where listeners can provide real-time feedback on the remixed audio's quality would provide valuable data. This can further refine the remixing techniques applied, ensuring an even more tailored listening experience.
6. **Expanding the Dataset:** The scope of our project was restricted to a particular dataset. Exploring more diverse datasets, spanning various genres, languages, and recording qualities, would help in generalizing the techniques we've developed.
7. **Exploring Other Metrics:** While the HAAQI score was invaluable for our project, it's worth looking into or developing other metrics that could provide different perspectives on audio quality, especially in the context of personalized audio enhancement [7].
8. **Enhanced Safety and Comfort Protocols:** As we continue to work towards the objective of improving the listening experience for people with different hearing capabilities, there's an ongoing need to ensure that the enhanced audio remains within safe and comfortable listening limits. Future work can focus on developing dynamic monitoring systems that adjust the enhancements in real-time based on the listener's feedback or physiological responses.
9. **Collaborations with Audiologists:** By working closely with audiologists, the project can gain a more profound understanding of different hearing impairments and their implications on music listening. Such collaborations could inform more refined enhancement techniques.
10. **Hardware Integrations:** The next step could be the development or integration with hardware devices, like hearing aids or specialized headphones, to provide a seamless enhanced listening experience without the need for post-processing.
11. **Real-time Enhancement:** Future initiatives can investigate the feasibility of real-time audio enhancement, allowing listeners to experience enhanced music without any latency.

List of References

- [1] cadenzaproject.github.io. n.d. The 1st Cadenza Challenge | The Cadenza Project. [online] Available at: http://cadenzachallenge.org/docs/cadenza1/cc1_intro [Accessed 5 Aug. 2023].
- [2] Audicus. 2016. World Wide Hearing Loss: Stats from Around the World. [online] Available at: <https://www.audicus.com/world-wide-hearing-loss-stats-from-around-the-world/#:~:text=According%20to%20the%20World%20Health%20Organization%20%28WHO%29%2C%20over,totals%20over%20360%20million%20people%20across%20the%20globe>
- [3] Luo, Y., Mesgarani, N., Black, M., & Smaragdis, P. 2019. Deep Clustering and Conventional Networks for Music Separation: Strong Together. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 236-241.
- [4] Roma, G., Grais, E. M., Simpson, A. J. R., & Plumbley, M. D. 2016. Music Remixing and Upmixing using Source Separation. Proceedings of the 2nd AES Workshop on Intelligent Music Production, London, UK.
- [5] Hershey, J. R., Chen, Z., Le Roux, J., & Watanabe, S. 2023. Deep clustering: Discriminative embeddings for segmentation and separation. MERL, Cambridge, MA.
- [6] Stoller, D., Ewert, S. and Dixon, S. 2018. WAVE-U-NET: A MULTI-SCALE NEURAL NETWORK FOR END-TO-END AUDIO SOURCE SEPARATION.
- [7] Kates, James & Arehart, Kathryn. 2015. The hearing-aid audio quality index (HAAQI). IEEE/ACM Transactions on Audio, Speech, and Language Processing. 24. 1-1. 10.1109/TASLP.2015.2507858.
- [8] Défossez, A., Usunier, N., Bottou, L. and Bach, F. 2021. Music Source Separation in the Waveform Domain. [online] Available at: <https://arxiv.org/abs/1911.13254v2>.
- [9] Garcia, H.F. 2021. Source Separation and Extensible MIR Tools for Audacity. [online] Audacity®. Available at: <https://www.audacityteam.org/source-separation-and-extensible-mr-tools-for-audacity/> [Accessed 4 Aug. 2023].
- [10] Fabian-Robert Stöter, Stefan Uhlich, Antoine Liutkus, Yuki Mitsufuji. 2019. Open-Unmix - A Reference Implementation for Music Source Separation. Journal of Open Source Software, 4 (41), pp.1667. 10.21105/joss.01667.
- [11] Hennequin, Romain & Khlif, Anis & Voituret, Felix & Moussallam, Manuel. 2020. Spleeter: a fast and efficient music source separation tool with pre-trained models. Journal of Open Source Software. 5. 2154. 10.21105/joss.02154.

- [12] Moussallam, M. 2019. Releasing Spleeter: Deezer R&D source separation engine. [online] Medium. Available at: <https://deezer.io/releasing-spleeter-deezer-r-d-source-separation-engine-2b88985e797e>.
- [13] Byrne, D., & Dillon, H. 1986. The National Acoustic Laboratories' (NAL) new procedure for selecting the gain and frequency response of a hearing aid. *Ear and hearing*, 7(4), 257-265.
- [14] Butterworth, S. 1930. On the Theory of Filter Amplifiers. *Experimental Wireless & the Wireless Engineer*, 7, 536-541.
- [15] McCormack L, Välimäki V. FFT-based Dynamic Range Compression. 2017.
- [16] Thiede, T.V., Treurniet, W.C., Bitto, R., Schmidmer, C., Sporer, T., Beerends, J.G. and Colomes, C. 2000. PEAQ - The ITU Standard for Objective Measurement of Perceived Audio Quality. *Journal of The Audio Engineering Society*, 48, pp.3-29. Available from: <https://api.semanticscholar.org/CorpusID:16573563> [Accessed (05/08/2023)].
- [17] Yuan T, Deng W, Tang J, Tang Y, Chen B. Signal-to-Noise Ratio: A Robust Distance Metric for Deep Metric Learning. *arXiv preprint arXiv:1904.02616*. 2019.
- [18] Xu Y, Du J, Dai LR, Lee CH. A Regression Approach to Speech Enhancement Based on Deep Neural Networks. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*. 2015;23(1):7-19. doi:10.1109/TASLP.2014.2364452.
- [19] Hershey, J. R., Chen, Z., Le Roux, J., & Watanabe, S. 2016. Deep clustering and conventional networks for music separation: Strong together. In *Proc. ICASSP*, 141-145.
- [20] Rivet, B., Girin, L., & Jutten, C. 2007. Mixing audiovisual speech processing and blind source separation for the extraction of speech signals from convolutive mixtures. *IEEE Transactions on Audio, Speech, and Language Processing*, 15(1), 96-108.
- [21] Yilmaz, O., & Rickard, S. 2004. Blind separation of speech mixtures via time-frequency masking. *IEEE Transactions on Signal Processing*, 52(7), 1830-1847.
- [22] Coalson, J. (2009). FLAC - Free Lossless Audio Codec. [online] Xiph.org. Available at: <https://xiph.org/flac/>
- [23] NumPy Developers. 2023. NumPy. [software]. Available at: <https://numpy.org> [Accessed 5 Aug. 2023].
- [24] SciPy Developers. 2023. SciPy. [software]. Available at: <https://scipy.org> [Accessed 5 Aug. 2023].

- [25] Bächler, M., and Geiger, M. 2023. Soundfile. [software]. Available at: <https://pysoundfile.readthedocs.io> [Accessed 5 Aug. 2023].
- [26] Python Software Foundation. 2023. Pathlib. [software]. Available at: <https://docs.python.org/3/library/pathlib.html> [Accessed 5 Aug. 2023].
- [27] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., & Zheng, X. 2016. TensorFlow: A System for Large-Scale Machine Learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), 265–283.
- [28] Crockford, D. 2006. JSON (JavaScript Object Notation) is a lightweight data-interchange format. [software]. Available at: <https://www.json.org> [Accessed 5 Aug. 2023].
- [29] Python Software Foundation. 2023. OS. [software]. Available at: <https://docs.python.org/3/library/os.html> [Accessed 5 Aug. 2023].
- [30] Le Roux J, Wisdom S, Erdogan HSD, Hershey JR. SDR - half-baked or well done? 2018. [arXiv preprint] arXiv:1811.02508.
- [31] Mitsufuji Y, Fabbro G, Uhlich S, Stöter F-R, Défossez A, Kim M, Choi W, Yu C-Y, Cheuk K-W. Music Demixing Challenge 2021. Frontiers in Signal Processing. 2022;1. doi:10.3389/frsip.2021.808395.
- [32] von Gablenz, P. and Holube, I. (2019). Data from: Pure-tone thresholds in adults aged 18 to 97 years and hearing aid use. [online] Zenodo. doi:<https://doi.org/10.5061/dryad.cfxpvnvx27>.
- [33] GDPR (2018). General Data Protection Regulation (GDPR). [online] General Data Protection Regulation (GDPR). Available at: <https://gdpr-info.eu/>.
- [34] A. Défossez, "Hybrid Spectrogram and Waveform Source Separation," 2022. [Online]. Available: <https://arxiv.org/abs/2111.03600>

Appendix A: External Materials

Throughout the course of the project, the following external materials and resources were employed:

A.1 Demixing Models: The demixing process employed three distinct models, each accessible through the following links:

- Hybrid Demucs: <https://github.com/facebookresearch/demucs>
- Open-Unmix: <https://github.com/sigsep/open-unmix-pytorch>
- Spleeter: <https://github.com/deezer/spleeter/wiki/3.-Models>

A.2 Tools and Platforms: The following tools and platforms were used throughout the project:

- Google Colab for model training and evaluation
- Visual Studio Code for code development
- Audacity for listening to output audio files
- LucidChart for creating diagrams in the report. It can be accessed at the following link: <https://www.lucidchart.com>

A.3 External Repository: The Clarity GitHub Repository was used as a valuable resource during the project. The repository can be accessed at the following link:

<https://github.com/claritychallenge/clarity/tree/main/recipes/cad1>

Appendix B: Ethical Issues Addressed

In this appendix, we elaborate on the ethical, legal, and social issues that were addressed during this project. We focused on three key areas: data privacy, algorithmic bias, and accessibility.

B.1 Data Privacy: We strictly adhere to the principles of data privacy in our project. All listener-specific data used to enhance the audio output is personal and sensitive information. We followed standards like the General Data Protection Regulation (GDPR) to ensure participant consent, safeguard data, and provide transparent data usage. The listener data was obtained from the Cadenza Challenge organizers, who took the necessary steps to acquire informed consent from the participants and ensure data protection. All data usage was limited to the scope of this project, and we did not disclose any identifiable listener information.

B.2 Algorithmic Bias: Ensuring fairness and avoiding algorithmic bias is a priority. The training data's quality and breadth determine the success of our system. We acknowledge that a lack of diversity in our dataset may lead to poorer performance for underrepresented groups, resulting in discriminatory outcomes. In this project, we used the datasets provided by the Cadenza Challenge organizers, which encompass a wide range of auditory capabilities and preferences to mitigate the risk of bias. By employing diverse data, we aimed to provide an inclusive audio enhancement experience for users with varying hearing profiles.

B.3 Accessibility: We strive to make our system universally accessible, particularly to individuals with hearing loss. Our system is designed to be user-friendly and easily integrated into various listening scenarios and devices. While the system is intended to augment the music listening experience for individuals with hearing impairments, we stress that it is not a replacement for professional audiological advice or hearing aids.

B.4 Legal and Copyright Issues: We also addressed legal and copyright considerations throughout the project. All music tracks used for training, validation, and testing were provided by the Cadenza Challenge organizers. They obtained the necessary permissions to use the music, ensuring compliance with copyright laws. As participants of the Cadenza Challenge, we had authorized access to these tracks for the purpose of the competition. Consequently, there were no copyright issues related to the use of the music in our project.

B.5 Social Considerations: The project aims to address the barriers faced by individuals with hearing loss, promoting inclusivity and equality. While our system enhances the music

listening experience, it is important to communicate that it is not a holistic solution to hearing impairments. It serves as an augmentation tool, and users should seek professional audiological advice as needed.

For further details regarding data procurement, consent procedures, data protection measures, and the Cadenza Challenge, you can refer to the following website: http://cadenzachallenge.org/docs/cadenza1/cc1_intro

Appendix C: Acronyms

C.1 Acronyms

- CNN - Convolutional Neural Network
- CPU - Central Processing Unit
- dB - Decibel
- DNN - Deep Neural Network
- EMA - Exponential Moving Average
- FLAC - Free Lossless Audio Codec
- FMA - Free Music Archive
- FFT - Fast Fourier Transform (based resampling)
- GDPR - General Data Protection Regulation
- GELU - Gaussian Error Linear Unit
- GPU - Graphics Processing Unit
- HAAQI - Hearing Aid Audio Quality Index
- HDEMUCS - Hybrid Demucs (specifically used in your project)
- JSON - JavaScript Object Notation
- LSTM - Long Short-Term Memory
- MIR - Music Information Retrieval
- MSE - Mean Squared Error
- MUSDB - Music Source Separation Database (specifically used in your project)
- NAL-R - National Acoustic Laboratories - Revised
- NALR-EQ - National Acoustic Laboratories - Revised Equalization
- NALR-FIR - National Acoustic Laboratories - Revised Finite Impulse Response
- OS - Operating System
- PEAQ - Perceptual Evaluation of Audio Quality
- RAM - Random Access Memory
- RELU - Rectified Linear Unit
- RMS - Root Mean Square
- RNN - Recurrent Neural Network
- SDR - Signal-to-Distortion Ratio
- SNR - Signal-to-Noise Ratio
- SVD - Singular Value Decomposition
- TPU - Tensor Processing Unit
- UMX - Open-Unmix (specifically used in your project)

- VDBO - Vocal, Drums, Bass, Other
- WAV - Waveform Audio File Format

Appendix D: Code Snippets

In this appendix, we provide examples of code snippets and detailed pseudocode of the algorithms or functions used in our implementation.

D.1 Audio Demixing Using Machine Learning Models:

Here, we include code snippets to demonstrate how we set up the three machine learning models (Hybrid Demucs, Open-Unmix, and Spleeter) for audio demixing.

Hybrid Demucs and Open-Unmix:

```
def decompose_signal(
    config: DictConfig,
    model: torch.nn.Module,
    signal: np.ndarray,
    sample_rate: int,
    device: torch.device,
    listener: Listener,
) -> dict[str, np.ndarray]:
    """
    Decompose signal into 8 stems.

    The left and right audiograms are ignored by the baseline system as it
    is performing personalised decomposition.
    Instead, it performs a standard music decomposition using the
    HDEMUCS model trained on the MUSDB18 dataset.

    Args:
        config (DictConfig): Configuration object.
        model (torch.nn.Module): Torch model.
        signal (np.ndarray): Signal to be decomposed.
        sample_rate (int): Sample frequency.
        device (torch.device): Torch device to use for processing.
        listener (Listener).

    Returns:
        Dictionary: Indexed by sources with the associated model as values.
    """
```



```

if config.separator.model == "demucs":
    signal, ref = normalize_signal(signal)

model_sample_rate = (
    model.sample_rate if config.separator.model == "openunmix" else 44100
)

if sample_rate != model_sample_rate:
    resampler = Resample(sample_rate, model_sample_rate)
    signal = resampler(signal)

sources = separate_sources(
    model, torch.from_numpy(signal), sample_rate, device=device
)
# only one element in the batch
sources = sources[0]
if config.separator.model == "demucs":
    sources = denormalize_signals(sources, ref)

signal_stems = map_to_dict(sources, config.separator.sources)
return signal_stems

```

```

@hydra.main(config_path="", config_name="config")
def enhance(config: DictConfig) -> None:
    """
    Run the music enhancement.
    The system decomposes the music into vocal, drums, bass, and other stems.
    Then, the NAL-R prescription procedure is applied to each stem.
    Args:
        config (dict): Dictionary of configuration options for enhancing music.

    Returns 8 stems for each song:
        - left channel vocal, drums, bass, and other stems
        - right channel vocal, drums, bass, and other stems
    """

    enhanced_folder = Path("enhanced_signals")
    enhanced_folder.mkdir(parents=True, exist_ok=True)

    # Training stage
    #
    # The baseline is using an off-the-shelf model trained on the MUSDB18 dataset
    # Training listeners and song are not necessary in this case.
    #
    # Training songs and audiograms can be read like this:
    #
    with open(config.path.listeners_train_file, "r", encoding="utf-8") as file:
        listener_train_audiograms = json.load(file)

```

```

with open(config.path.music_train_file, "r", encoding="utf-8") as file:
    song_data = json.load(file)
songs_train = pd.DataFrame.from_dict(song_data)

train_song_listener_pairs = make_song_listener_list(
    songs_train['Track Name'], listener_train_audiograms
)

if config.separator.model == "demucs":
    separation_model = HDEMUCS_HIGH_MUSDB.get_model()
else:
    separation_model = torch.hub.load("sigsep/open-unmix-pytorch", "umxhq", niter=0)
device, _ = get_device(config.separator.device)
separation_model.to(device)

# Processing Validation Set
# Load listener audiograms and songs
listener_dict = Listener.load_listener_dict(config.path.listeners_valid_file)

with open(config.path.music_valid_file, encoding="utf-8") as file:
    song_data = json.load(file)
songs_valid = pd.DataFrame.from_dict(song_data)

valid_song_listener_pairs = make_song_listener_list(
    songs_valid['Track Name'], listener_dict
)

```

```

enhancer = NALR(**config.nalr)
compressor = Compressor(**config.compressor)

# Decompose each song into left and right vocal, drums, bass, and other stems
# and process each stem for the listener
prev_song_name = None
num_song_list_pair = len(valid_song_listener_pairs)
for idx, song_listener in enumerate(valid_song_listener_pairs, 1):
    song_name, listener_name = song_listener
    logger.info(
        f"[{idx:03d}/{num_song_list_pair:03d}] "
        f"Processing {song_name} for {listener_name}..."
    )
    # Get the listener's audiogram
    listener = listener_dict[listener_name]

    # Find the music split directory
    split_directory = (
        "test"
        if songs_valid.loc[songs_valid["Track Name"] == song_name, "Split"].iloc[0]
        == "test"
        else "train"
    )

```

Spleeter:

```

# Load listener characteristics
with open('/content/drive/MyDrive/Cadenza/CadenzaData/task1/cadenza_cad1_task1_evaluation.v1_0/cadenza_data/cad1/task1/metadata/new_listeners.valid.json') as json_file:
    all_listener_characteristics = json.load(json_file)

# Load audio segments
with open('/content/drive/MyDrive/Cadenza/CadenzaData/task1/cadenza_cad1_task1_evaluation.v1_0/cadenza_data/cad1/task1/metadata/musdb18.segments.test.json') as json_file:
    all_audio_segments = json.load(json_file)

separator = Separator('spleeter:4stems')

def process_song(listener_id, song_dir, song_file):
    # Extract song_name from song_file
    song_name = os.path.basename(song_dir)

    listener_characteristics = all_listener_characteristics[listener_id]
    audio_segments = all_audio_segments[song_name]

    # Create Separator and process song
    audio_loader = AudioAdapter.default()
    waveform, _ = audio_loader.load(os.path.join(song_dir, song_file), sample_rate=44100)
    audio_descriptor = separator.separate(waveform)

    # Apply listener characteristics to each stem
    remixed_stems = {}
    remixed_signal = np.zeros_like(waveform) # Initialize remixed_signal
    for stem, stem_signal in audio_descriptor.items():

# Apply listener characteristics to each stem
remixed_stems = {}
remixed_signal = np.zeros_like(waveform) # Initialize remixed_signal
for stem, stem_signal in audio_descriptor.items():
    remixed_stem = apply_listener_characteristics(stem_signal, listener_characteristics)
    remixed_stems[stem] = remixed_stem
    # Perform padding operation
    if remixed_stem.shape[0] < remixed_signal.shape[0]:
        padding = np.zeros((remixed_signal.shape[0] - remixed_stem.shape[0], remixed_signal.shape[1]))
        remixed_stem = np.concatenate((remixed_stem, padding), axis=0)
    elif remixed_stem.shape[0] > remixed_signal.shape[0]:
        padding = np.zeros((remixed_stem.shape[0] - remixed_signal.shape[0], remixed_signal.shape[1]))
        remixed_signal = np.concatenate((remixed_signal, padding), axis=0)
    remixed_signal += remixed_stem # Add each processed stem to the remixed_signal

```


D.2 Remixing Using Butterworth Bandpass Filter and Dynamic Compression:

This section includes code snippets showcasing the implementation of the Butterworth bandpass filter and dynamic compression in the remixing process.

Butterworth Bandpass Filter:

```
def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    sos = butter(order, [low, high], btype='band', analog=False, output='sos')
    return sos

def apply_butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    sos = butter_bandpass(lowcut, highcut, fs, order=order)
    y = sosfilt(sos, data)
    return y
```

Dynamic Compression:

```
class Compressor:
    def __init__(
        self,
        fs: float = 44100.0,
        attack: float = 5.0,
        release: float = 20.0,
        threshold: float = 1.0,
        attenuation: float = 0.0001,
        rms_buffer_size: float = 0.2,
        makeup_gain: float = 1.0,
        **kwargs,
    ) -> None:
        """Instantiate the Compressor Class.

        Args:
            fs (float): (default = 44100.0)
            attack (float): (default = 5.0)
            release float: (default = 20.0)
            threshold (float): (default = 1.0)
            attenuation (float): (default = 0.0001)
            rms_buffer_size (float): (default = 0.2)
            makeup_gain (float): (default = 1.0)
        """
```

```

def set_attack(self, t_msec: float) -> None:
    """DESCRIPTION

    Args:
        t_msec (float): DESCRIPTION

    Returns:
        float: DESCRIPTION
    """
    t_sec = t_msec / 1000.0
    reciprocal_time = 1.0 / t_sec
    self.attack = reciprocal_time / self.fs

def set_release(self, t_msec: float) -> None:
    """DESCRIPTION

    Args:
        t_msec (float): DESCRIPTION

    Returns:
        float: DESCRIPTION
    """
    t_sec = t_msec / 1000.0
    reciprocal_time = 1.0 / t_sec
    self.release = reciprocal_time / self.fs

```

D.3 Evaluation Using HAAQI, SDR, and SNR Metrics:

Here, we provide code snippets for evaluating the output audio files with the HAAQI, SDR, and SNR metrics.

HAAQI Evaluation:

```
def haaqi_v1(
    reference: ndarray,
    reference_freq: float,
    processed: ndarray,
    processed_freq: float,
    audiogram: Audiogram,
    equalisation: int,
    level1: float = 65.0,
    silence_threshold: float = 2.5,
    add_noise: float = 0.0,
    segment_covariance: int = 16,
) -> tuple[float, float, float, list[float]]:
    """
    Compute the HAAQI music quality index using the auditory model followed by
    computing the envelope cepstral correlation and Basilar Membrane vibration
    average short-time coherence signals.

    The reference signal presentation level for NH listeners is assumed
    to be 65 dB SPL. The same model is used for both normal and
    impaired hearing.

    Arguments:
        reference (ndarray): Input reference speech signal with no noise or distortion.
        reference_freq (int): Sampling rate in Hz for reference signal.
        processed (np.ndarray): Output signal with noise, distortion, HA gain,
        and/or processing
```

```
if not audiogram.has_frequencies(HAAQI_AUDIOGRAM_FREQUENCIES):
    logging.warning(
        "Audiogram does not have all HAAQI frequency measurements"
        "Measurements will be interpolated"
    )

audiogram = audiogram.resample(HAAQI_AUDIOGRAM_FREQUENCIES)

# Auditory model for quality
# Reference is no processing or NAL-R, impaired hearing
(
    reference_db,
    reference_basilar_membrane,
    processed_db,
    processed_basilar_membrane,
    reference_sl,
    processed_sl,
    sample_rate,
) = eb.ear_model(
    reference,
    reference_freq,
    processed,
    processed_freq,
    audiogram.levels,
    equalisation,
    level1,
)
```

SNR and SDR Evaluation:

```
import numpy as np
import mir_eval
from scipy.io import wavfile

#Change Listener ID here
listenerID = 'L5040'
#Change model name here
modelName = 'Open-Unmix'

# File paths
original_path = "/content/clarity/recipes/cad1/task1/baseline/cadenza_data_demo/cad1/task1/audio/musdb18hq/train/Actions - One Minute Smile"
processed_path = f"/content/clarity/recipes/cad1/task1/baseline/exp/enhanced_signals/{listenerID}/Actions - One Minute Smile"

stems = ['vocals', 'drums', 'bass', 'other']

def snr(signal, noise):
    return 10 * np.log10(np.sum(signal ** 2) / np.sum((signal - noise) ** 2))

print(f"Processing using {modelName} for Listener {listenerID} Song - Actions - One Minute Smile")
```

```
for stem in stems:
    # Load original stems
    _, original = wavfile.read(f"{original_path}/{stem}.wav")
    original = original / np.max(np.abs(original), axis=0)

    # Load processed stems
    _, left_processed = wavfile.read(f"{processed_path}/{listenerID}_Actions - One Minute Smile_left_{stem}.wav")
    left_processed = left_processed / np.max(np.abs(left_processed))
    _, right_processed = wavfile.read(f"{processed_path}/{listenerID}_Actions - One Minute Smile_right_{stem}.wav")
    right_processed = right_processed / np.max(np.abs(right_processed))

    # Handle the case where the processed files are mono
    if len(left_processed.shape) == 1:
        left_processed = left_processed[:, np.newaxis]
    if len(right_processed.shape) == 1:
        right_processed = right_processed[:, np.newaxis]

    # Truncate or pad signals to match lengths
    min_length = min(original.shape[0], left_processed.shape[0], right_processed.shape[0])
    original = original[:min_length, :]
    left_processed = left_processed[:min_length, :]
    right_processed = right_processed[:min_length, :]
```

```
# Truncate or pad signals to match lengths
min_length = min(original.shape[0], left_processed.shape[0], right_processed.shape[0])
original = original[:min_length, :]
left_processed = left_processed[:min_length, :]
right_processed = right_processed[:min_length, :]

# Calculate SDR for left and right channels
left_sdr, _, _ = mir_eval.separation.bss_eval_sources(original[:, 0][np.newaxis, :], left_processed[:, 0][np.newaxis, :])
right_sdr, _, _ = mir_eval.separation.bss_eval_sources(original[:, 1][np.newaxis, :], right_processed[:, 0][np.newaxis, :])

# Calculate SNR for left and right channels
left_snr = snr(original[:, 0], left_processed[:, 0])
right_snr = snr(original[:, 1], right_processed[:, 0])

# Print results
print(f"{stem} stem:")
print(f"Left channel SNR: {left_snr} dB, SDR: {left_sdr[0]} dB")
print(f"Right channel SNR: {right_snr} dB, SDR: {right_sdr[0]} dB")
print()
```


Appendix E: Results

In this appendix, we present the visualization of our results in the form of tables and screenshots of output. The results are based on the evaluation metrics HAAQI, SNR, and SDR for each of the three models (Hybrid Demucs, Open-Unmix, and Spleeter) and how they compared to one another.

E.1 HAAQI Results

Model	Listener ID	Average HAAQI Score
Spleeter	L5076	0.5636
Spleeter	L5040	0.6208
Hybrid Demucs	L5076	0.3543
Hybrid Demucs	L5040	0.2280
Open-Unmix	L5076	0.2098
Open-Unmix	L5040	0.1145

Presented below are the screenshots displaying the output of the HAAQI Scores:

HDEMUCS:

```
See https://hydra.cc/docs/1.2/upgrades/1.1\_to\_1.2/changes\_to\_job\_working\_dir/ for more information.
ret = run_job(
[2023-08-17 13:56:29,401][__main__][INFO] - Evaluating from enhanced_signals directory
[2023-08-17 13:56:29,405][__main__][INFO] - Evaluating Actions - One Minute Smile for L5076
[2023-08-17 13:56:29,405][__main__][INFO] - ...evaluating drums
[2023-08-17 14:05:12,511][__main__][INFO] - ...evaluating bass
[2023-08-17 14:09:45,246][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-17 14:14:18,495][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-17 14:14:18,678][__main__][INFO] - ...evaluating other
[2023-08-17 14:23:11,961][__main__][INFO] - ...evaluating vocals
[2023-08-17 14:32:09,121][__main__][INFO] - The combined score is 0.35429742361942773
[2023-08-17 14:32:09,126][__main__][INFO] - Evaluating Actions - One Minute Smile for L5040
[2023-08-17 14:32:09,127][__main__][INFO] - ...evaluating drums
[2023-08-17 14:40:53,565][__main__][INFO] - ...evaluating bass
[2023-08-17 14:44:06,236][clarity.evaluator.haspi.eb][WARNING] - Function melcor9: Signal below threshold, outputs set to 0.
[2023-08-17 14:45:24,651][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-17 14:48:36,373][clarity.evaluator.haspi.eb][WARNING] - Function melcor9: Signal below threshold, outputs set to 0.
[2023-08-17 14:49:57,415][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-17 14:49:57,770][__main__][INFO] - ...evaluating other
[2023-08-17 14:59:01,413][__main__][INFO] - ...evaluating vocals
[2023-08-17 15:08:01,995][__main__][INFO] - The combined score is 0.2279514791428078
```


Open-Unmix:

```
@hydra.main(config_path="", config_name="config")
/usr/local/lib/python3.10/dist-packages/hydra/internal/hydra.py:119: UserWarning: Future Hydra versions will no longer change working di
see https://hydra.cc/docs/1.2/upgrades/1.1\_to\_1.2/changes\_to\_job\_working\_dir/ for more information.
ret = run_job()
[2023-08-14 11:26:01,390][__main__][INFO] - Evaluating from enhanced_signals directory
[2023-08-14 11:26:01,395][__main__][INFO] - Evaluating Actions - One Minute Smile for L5076
[2023-08-14 11:26:01,396][__main__][INFO] - ...evaluating drums
[2023-08-14 11:34:51,171][__main__][INFO] - ...evaluating bass
[2023-08-14 11:39:23,872][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-14 11:43:51,647][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-14 11:43:51,832][__main__][INFO] - ...evaluating other
[2023-08-14 11:53:05,515][__main__][INFO] - ...evaluating vocals
[2023-08-14 12:02:16,750][__main__][INFO] - The combined score is 0.2098460141100973
[2023-08-14 12:02:16,759][__main__][INFO] - Evaluating Actions - One Minute Smile for L5040
[2023-08-14 12:02:16,760][__main__][INFO] - ...evaluating drums
[2023-08-14 12:11:12,084][__main__][INFO] - ...evaluating bass
[2023-08-14 12:14:19,480][clarity.evaluator.haspi.eb][WARNING] - Function melcor9: Signal below threshold, outputs set to 0.
[2023-08-14 12:15:38,072][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-14 12:18:41,476][clarity.evaluator.haspi.eb][WARNING] - Function melcor9: Signal below threshold, outputs set to 0.
[2023-08-14 12:20:03,743][clarity.evaluator.haspi.eb][WARNING] - Function AveCovary2: Ave signal below threshold, outputs set to 0.
[2023-08-14 12:20:04,078][__main__][INFO] - ...evaluating other
[2023-08-14 12:29:10,405][__main__][INFO] - ...evaluating vocals
[2023-08-14 12:38:10,237][__main__][INFO] - The combined score is 0.11447231569490376
```

Spleeter:

```
Processing for: L5076 Actions - One Minute Smile
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/spleeter/separator.py:146: calling DatasetV2.from_generator (from t
Instructions for updating:
Use output_signature instead
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/spleeter/separator.py:146: calling DatasetV2.from_generator (from t
Instructions for updating:
Use output_signature instead
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/keras/layers/normalization/batch_normalization.py:514: _colocate_wi
Instructions for updating:
Colocations handled automatically by placer.
HAAQI Score (Left) for L5076 and Actions - One Minute Smile: 0.6027997087408767
HAAQI Score (Right) for L5076 and Actions - One Minute Smile: 0.5243480810847844
Average HAAQI Score for L5076 and Actions - One Minute Smile: 0.5635738949128306
Processing for: L5040 Actions - One Minute Smile
HAAQI Score (Left) for L5040 and Actions - One Minute Smile: 0.6185755507499319
HAAQI Score (Right) for L5040 and Actions - One Minute Smile: 0.6231200160233269
Average HAAQI Score for L5040 and Actions - One Minute Smile: 0.6208477833866294
```

E.2 SNR Results:

Model	Listener ID	Average SNR (dB)
Hybrid Demucs	L5040	-4.21
Hybrid Demucs	L5076	-3.80
Open-Unmix	L5040	-5.85
Open-Unmix	L5076	-3.22

Spleeter	L5040	-2.31
Spleeter	L5076	-2.31

E.3 SDR Results

Model	Listener ID	Average SDR (dB)
Hybrid Demucs	L5040	3.98
Hybrid Demucs	L5076	5.25
Open-Unmix	L5040	-14.55
Open-Unmix	L5076	-11.82
Spleeter	L5040	-38.91
Spleeter	L5076	-38.91

Presented below are the screenshots displaying the SNR and SDR scores:

Hybrid Demucs:

Processing using Hybrid Demucs for Listener L5040 Song - Actions - One Minute Smile vocals stem:

Left channel SNR: -9.731103892598306 dB, SDR: 4.533964273322699 dB

Right channel SNR: -7.054813334370801 dB, SDR: 5.287076630464433 dB

drums stem:

Left channel SNR: -11.460815336754631 dB, SDR: 5.685749750840627 dB

Right channel SNR: -9.435177497438005 dB, SDR: 5.533741201792807 dB

bass stem:

Left channel SNR: 0.5189182263033034 dB, SDR: 1.7058484698934853 dB

Right channel SNR: 0.7659668258336128 dB, SDR: 2.970353362358576 dB

other stem:

Left channel SNR: -9.007715574098828 dB, SDR: 2.0474301734833977 dB

Right channel SNR: -5.046553658011077 dB, SDR: 3.0465945276917417 dB

Processing using Hybrid Demucs for Listener L5076 Song - Actions - One Minute Smile
vocals stem:

Left channel SNR: -6.154060116415149 dB, SDR: 8.186843801932822 dB

Right channel SNR: -6.49406324378756 dB, SDR: 8.121691697936301 dB

drums stem:

Left channel SNR: -6.844620749313184 dB, SDR: 8.445845118897907 dB

Right channel SNR: -6.190181970582655 dB, SDR: 6.612828780762761 dB

bass stem:

Left channel SNR: 0.8326287622608642 dB, SDR: 3.1661781275082475 dB

Right channel SNR: 0.7804311241406914 dB, SDR: 3.0253653599637857 dB

other stem:

Left channel SNR: -3.7487183339781707 dB, SDR: 2.3112090215911696 dB

Right channel SNR: -3.1401651540268514 dB, SDR: 3.334791985201804 dB

Open-Unmix:

Processing using Open-Unmix for Listener L5040 Song - Actions - One Minute Smile
vocals stem:

Left channel SNR: -9.592373177009806 dB, SDR: -4.656040849476277 dB

Right channel SNR: -6.651743412029536 dB, SDR: -5.97649398640214 dB

drums stem:

Left channel SNR: -12.29204235821298 dB, SDR: -11.937894319702803 dB

Right channel SNR: -9.492466333946442 dB, SDR: -11.703682431974151 dB

bass stem:

Left channel SNR: -5.346665990684141 dB, SDR: -33.362191174150155 dB

Right channel SNR: -3.6502329814969805 dB, SDR: -30.564848265079746 dB

other stem:

Left channel SNR: -0.8100381039030125 dB, SDR: -8.17489423233156 dB

Right channel SNR: -1.0580672687597787 dB, SDR: -8.261800057720931 dB

Processing using Open-Unmix for Listener L5076 Song - Actions - One Minute Smile
vocals stem:

Left channel SNR: -4.31877207418241 dB, SDR: -4.679205774472608 dB

Right channel SNR: -4.44763648666715 dB, SDR: -5.719958354539651 dB

drums stem:

Left channel SNR: -7.958197907106797 dB, SDR: -13.801736953635459 dB

Right channel SNR: -8.530528839103349 dB, SDR: -15.924622566527809 dB

bass stem:

Left channel SNR: -2.093721286890216 dB, SDR: -27.02648200912018 dB

Right channel SNR: -1.6908929661693293 dB, SDR: -25.449063200177314 dB

other stem:

Left channel SNR: -2.44320947140087 dB, SDR: -8.042502785695302 dB

Right channel SNR: -2.0824503699694255 dB, SDR: -7.997278525961116 dB

Spleeter:

Processing using Spleeter for Listener L5040 Song - Actions - One Minute Smile
vocals stem:

Left channel SNR: -2.4871560794690346 dB, SDR: -38.103468448247035 dB

Right channel SNR: -2.7513125772993203 dB, SDR: -38.82683547570868 dB

drums stem:

Left channel SNR: -3.8226419748191858 dB, SDR: -39.123400905745996 dB

Right channel SNR: -4.216357140345779 dB, SDR: -39.109462044268135 dB

bass stem:

Left channel SNR: -0.8894123518488086 dB, SDR: -38.50384943957885 dB

Right channel SNR: -0.9941499064510467 dB, SDR: -39.30905077923421 dB

other stem:

Left channel SNR: -2.037887836860272 dB, SDR: -39.89649483528919 dB

Right channel SNR: -1.8339901418220308 dB, SDR: -39.877747651756394 dB

Processing using Spleeter for Listener L5076 Song - Actions - One Minute Smile
vocals stem:

Left channel SNR: -2.4871560794690346 dB, SDR: -38.103468448247035 dB

Right channel SNR: -2.7513125772993203 dB, SDR: -38.82683547570868 dB

drums stem:

Left channel SNR: -3.8226419748191858 dB, SDR: -39.123400905745996 dB

Right channel SNR: -4.216357140345779 dB, SDR: -39.109462044268135 dB

bass stem:

Left channel SNR: -0.8894123518488086 dB, SDR: -38.50384943957885 dB

Right channel SNR: -0.9941499064510467 dB, SDR: -39.30905077923421 dB

other stem:

Left channel SNR: -2.037887836860272 dB, SDR: -39.89649483528919 dB

Right channel SNR: -1.8339901418220308 dB, SDR: -39.877747651756394 dB

Appendix F: Sample Output Files

In this section, we present several sample output audio files that were generated using the algorithms and methods outlined in our report. These samples demonstrate the effects of our demixing and remixing processes on different audio tracks. We showcase the performance of the Hybrid Demucs, Open-Unmix, and Spleeter models, highlighting their specific strengths and weaknesses. The sample files have been stored and organized into separate folders for each model, making it easy to compare their outputs.





The provided samples are divided into two categories: "demixed" and "remixed." The "demixed" folder contains audio files that have been separated into their constituent tracks using the three models mentioned above. The "remixed" folder showcases the results of applying the Butterworth bandpass filter and dynamic compression techniques to the demixed tracks, as described in our report.

Please note that we used Audacity, an open-source audio editing software, to listen to and compare the quality of the output audio files. Audacity allows for a detailed analysis of the audio, including the ability to visualize the waveform, spectrogram, and frequency spectrum of the tracks. It also enables users to apply various audio effects and analyse the impact of the demixing and remixing processes on the audio tracks.





The sample output files are available in the following folders:

F.1 Demixed:









a. Hybrid Demucs Demixed Files

	L5040_Actions - One Minute Smile_left_bass.wav	WAV File	14,090 KB	No	14,090 KB	0%	17-08-2023 17:52
	L5040_Actions - One Minute Smile_left_drums.wav	WAV File	14,090 KB	No	14,090 KB	0%	17-08-2023 17:52
	L5040_Actions - One Minute Smile_left_other.wav	WAV File	14,090 KB	No	14,090 KB	0%	17-08-2023 17:52
	L5040_Actions - One Minute Smile_left_vocals.wav	WAV File	14,090 KB	No	14,090 KB	0%	17-08-2023 17:52

b. Open-Unmix Demixed Files

	WAV File	L5076_Actions - One Minute Smile_left_bass.wav	14,090 KB	No	14,090 KB	0%	17-08-2023 17:51
	WAV File	L5076_Actions - One Minute Smile_left_drums.wav	14,090 KB	No	14,090 KB	0%	17-08-2023 17:51
	WAV File	L5076_Actions - One Minute Smile_left_other.wav	14,090 KB	No	14,090 KB	0%	17-08-2023 17:51
	WAV File	L5076_Actions - One Minute Smile_left_vocals.wav	14,090 KB	No	14,090 KB	0%	17-08-2023 17:51

c. Spleeter Demixed Files

 L5040_Actions - One Minute Smile_left_bass.flac	FLAC File	1,133 KB	No	1,133 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_left_bass.txt	Text Document	1 KB	No	1 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_left_drums.flac	FLAC File	1,133 KB	No	1,133 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_left_drums.txt	Text Document	1 KB	No	1 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_left_other.flac	FLAC File	1,133 KB	No	1,133 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_left_other.txt	Text Document	1 KB	No	1 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_left_vocals.flac	FLAC File	1,133 KB	No	1,133 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_left_vocals.txt	Text Document	1 KB	No	1 KB	0%	17-08-2023 21:02

F.2 Remixed:

a. Hybrid Demucs Remixed Files

 L5040_Actions - One Minute Smile_remix.wav	WAV File	28,179 KB	No	28,179 KB	0%	17-08-2023 17:52
------------------------------------------------------------------------------------------------------------------------------	----------	-----------	----	-----------	----	------------------

b. Open-Unmix Remixed Files

 WAV File	L5076_Actions - One Minute Smile_remix.wav	28,179 KB	No	28,179 KB	0%	17-08-2023 17:52
--------------------------------------------------------------------------------------------	--------------------------------------------	-----------	----	-----------	----	------------------

c. Spleeter Remixed Files

 L5040_Actions - One Minute Smile_remix.flac	FLAC File	1,157 KB	No	1,157 KB	0%	17-08-2023 21:02
 L5040_Actions - One Minute Smile_remix.txt	Text Document	1 KB	No	1 KB	0%	17-08-2023 21:02

We encourage listeners to evaluate these files using Audacity or any other audio editing software to gain a better understanding of the enhancements provided by our system.

[Please note that due to the size of the sample files and the format limitations of this document, the audio files are not included here. They can be accessed through the accompanying digital resources provided with this report.]

By analysing these sample output files, you can observe the differences in performance between the three models in terms of audio quality, clarity, and fidelity. We hope these samples will help you understand the potential of our system to improve the music listening experience for individuals with diverse hearing profiles.

Appendix G: List of Figures

G.1 List of Figures

Figure 1.1: Baseline for the headphone listening scenario - This figure illustrates the typical setup for headphone-based listening scenarios, including the auditory elements involved. [1]

Figure 2.1: Illustration of Source Separation - A visual depiction of the source separation process, showcasing how individual sound sources are isolated from a mixed audio signal. [9]

Figure 2.2 a): Common Model - A block diagram of the common model architecture used in source separation tasks. [6]

Figure 2.2 b): Wave-U-Net model - A block diagram of the Wave-U-Net architecture, highlighting its unique features. [6]

Figure 2.3: Block Diagram of the Upmixing System - This figure showcases the structure of the upmixing system, with each of its components represented as blocks. [4]

Figure 2.4: Ablation Study findings of Hybrid Demucs - The figure presents the findings of an ablation study on the Hybrid Demucs model, indicating the contribution of each component to the model's performance.

Figure 2.5: Boxplots of evaluation results of the UMX model compared with other methods - This figure presents boxplots of the evaluation results for the UMX model compared to other source separation methods. [10]

Figure 2.6: Schematic for HAAQI - A visual representation of the HAAQI metric and its components. [7]

Figure 3.1: High Level Block Diagram of the System - A high-level overview of the proposed system in a block diagram format.

Figure 3.2: Detailed Flow Diagram of the system - A detailed flowchart representing the flow of data and processes within the proposed system.

Figure 3.3: Architecture of Hybrid DEMUCS Model - A schematic representation of the architecture of the Hybrid Demucs model. [25]

Figure 3.4: Architecture of Open-Unmix Model - A schematic representation of the architecture of the Open-Unmix model. [25]

Fig 4.1 to Fig 4.17: These figures showcase various code excerpts from the implementation phase of the project. They contain code snippets related to the implementation of the Hybrid

Demucs and Open-Unmix models, the training and usage of the Spleeter model, the application of different audio processing techniques, and the conversion of audio files.

Fig 5.1: Extraction of Audiograms for HAAQI Calculation - This figure shows the steps involved in extracting audiograms for calculating HAAQI scores.

Fig 5.2: Left and Right HAAQI Scores Calculation - This figure depicts the process of calculating the HAAQI scores for the left and right channels.

Fig 5.3: Average HAAQI Score Calculation - This figure illustrates the steps involved in calculating the average HAAQI scores.

Fig 5.4: Printing and saving the HAAQI Scores - This figure showcases the process of printing and saving the calculated HAAQI scores.

Fig 5.5: Screenshot of all the listener output files

Fig 5.6: Screenshot of song tracks for each listener

Fig 5.7: Screenshot of files generated for each song track