Bioacoustic Recognition of Unhealthy and Intact Tones (BRUIT) Overview Report

Dr. Jacob E. Crosby V 0.1

July 2025

Contents

1 Project Objectives and Approach

This project was designed with two primary objectives in mind:

- Extract features from raw audio files. i.e. a part of your approach should identify cardiac events (S1, S2) in a .wav file.
- Develop an algorithm or a model that can differentiate between recordings of healthy heartbeats, ones with murmurs and recordings containing artifacts. Clearly document your pipeline, design decisions, and findings.

To accomplish these goals, I developed a modular and fully configurable framework called **BRUIT** — *Bioacoustic Recognition of Unhealthy and Intact Tones*. This framework is designed for extensibility, and all parameters are controlled through a single configuration file. Installation instructions and usage examples are provided in the project `README`. The repository can be cloned using:

```
git clone https://github.com/NeuroSonicSoftware/Jacob-assignment.git
```

1.1 Approach

The dataset comprises 105 audio recordings: 31 labeled as healthy, 34 with murmurs, and 40 containing artifacts. All files are in `.wav` format. Given the limited size of the dataset and the short timeframe for experimentation, the central question becomes: how can we extract the most informative data with maximum efficiency?

The preprocessing pipeline I implemented (see Section 2) is designed to isolate diagnostically relevant content by filtering noise and segmenting the recordings into smaller units centered around single heartbeat cycles. Each segment typically includes a complete S1–S2 pair, resulting in approximately 6–8 high-quality segments per file. This segmentation strategy ensures both noise suppression and preservation of temporal features critical for model learning.

For model development, I began with a Convolutional Neural Network (CNN), which is well-suited for analyzing time-frequency representations like MFCCs. Given the dataset size, CNNs strike a balance between model capacity and training stability. However, I also designed the system to support more complex temporal architectures. As the dataset grows, I envision transitioning toward more expressive models such as Graph Neural Networks (GNNs) integrated with Transformer-based attention.

To incorporate sequential dependencies present in cardiac cycles, I extended the core CNN architecture with optional recurrent components. Specifically, the framework includes configurations for:

- **CNN only**
- **CNN-LSTM**: Incorporating Long Short-Term Memory units to model long-range dependencies
- **CNN-GRU**: Using Gated Recurrent Units for efficient sequence modeling

All models are evaluated after training using multiple performance metrics. These include the training history (loss and accuracy curves), a confusion matrix, and a full classification report with precision, recall, F1-score, and support values per class. For a detailed discussion of the results, see Section X.

2 The Development of the Preprocessing Pipeline

The preprocessing steps are the most crucial in a project like this as the final extracted features are greatly coupled to the clarity of the preprocessing pipeline output. A general pipeline for audio include these following steps.

- **Raw:** Begin with the original audio files, typically in **.wav** format for lossless quality.
- **Resample:** Resample all audio files to a consistent sampling rate (e.g., 16 kHz or 44.1 kHz) to ensure comparability across the dataset.
- **Filter:** Apply a bandpass filter to isolate relevant frequency ranges (e.g., 20–2000 Hz for heartbeats or vocal ranges) and reduce noise.
- **Normalize:** Normalize amplitude levels across samples to eliminate loudness variability and stabilize training.
- **Pad/Crop/Segment:** Crop long silences or known artifacts, pad shorter clips to a uniform length (usually with zeros), or segment long recordings into shorter, fixed-size chunks.
- **Feature Extraction:** Convert the cleaned waveform into meaningful features (e.g., MFCCs, Mel-spectrograms, ZCR, RMS) that capture relevant temporal and spectral information.
- **Format for ML Input:** Structure the extracted features into consistent formats (e.g., 2D arrays, '.npz' files) suitable for input into machine learning models.

The pipeline that I created in the BRUIT framework applies these steps but they go into finer detail. An overview of my pipeline can be seen in Figure 1.

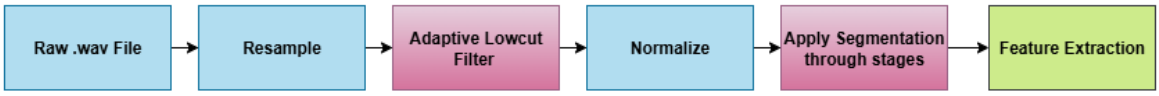

Figure 1: The macro overview of the preprocessing pipeline in the BRUIT framework. The steps shown in red are more complex and are discussed in further detail.

2.1 Adaptive Lowcut Filtering

The first step of filtering is fairly conservative, as a more aggressive filtering is later applied during segmentation. This first pass of filtering starts with a raw audio signal y sampled at rate f_s, I apply a Butterworth bandpass filter between a variable lowcut frequency f_{low} and a fixed highcut frequency f_{high}:

$$f_{\text{low}} \in \{20, 30, \ldots, 250\} \text{ Hz}, \quad f_{\text{high}} = 1200 \text{ Hz}$$

The filtered signal y_{filt} is computed via:

$$y_{\text{filt}} = \text{filtfilt}(b, a, y)$$

where (b, a) are the coefficients of a 4th-order Butterworth filter.

Next, I extract peaks in the absolute amplitude $|y_{\text{filt}}|$ using a prominence threshold. If fewer than 3 peaks are detected, the candidate is rejected.

MAD-Based Outlier Rejection

To ensure peak reliability, I remove outlier peaks using the Median Absolute Deviation (MAD). For a set of peak heights $\{x_1, \ldots, x_n\}$:

$$\text{median} = \tilde{x} = \text{median}(x_i)$$

$$\text{MAD} = \text{median}(|x_i - \tilde{x}|)$$

I retain only peaks that satisfy:

$$|x_i - \tilde{x}| < 3 \cdot \text{MAD}$$

This step ensures that only stable, regularly spaced peaks are used in the scoring process.

Scoring Function

For each valid lowcut candidate, I calculate a score S based on: - N: number of valid peaks - σ_s^2: variance of inter-peak spacing - MAD_{amp}: MAD of peak amplitudes

The score is defined as:

$$S = \frac{N}{1 + \sigma_s^2 + \text{MAD}_{\text{amp}}}$$

This formulation favors filters that produce many regularly spaced and consistently strong peaks.

Selection and Plotting

The lowcut frequency yielding the highest score is selected:

$$f_{\text{low}}^* = \arg\max_{f_{\text{low}}} S$$

Optionally, a plot of lowcut frequencies versus scores is generated for debugging and inspection (Figure 2).

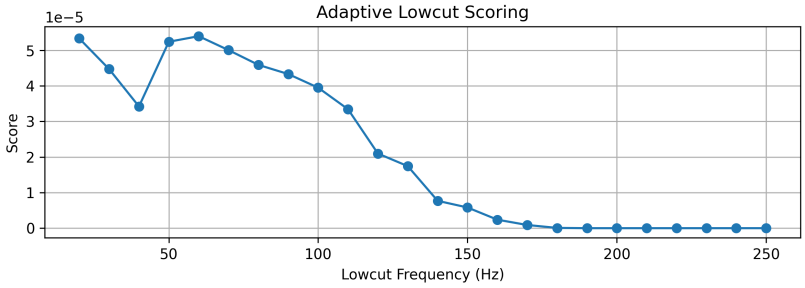

Figure 2: Lowcut frequency vs. scoring function values. The peak corresponds to the optimal lowcut value. This is from the file 201108222227.wav under the murmur label.

This process aims to automatically select the frequency range that maximizes signal clarity and rhythmic regularity.

Figures 3 and 4 show the raw and preprocessed waveforms for the same recording. While the overall structure appears similar, the preprocessed signal displays slightly sharper peaks, which are more useful for subsequent segmentation and feature extraction.

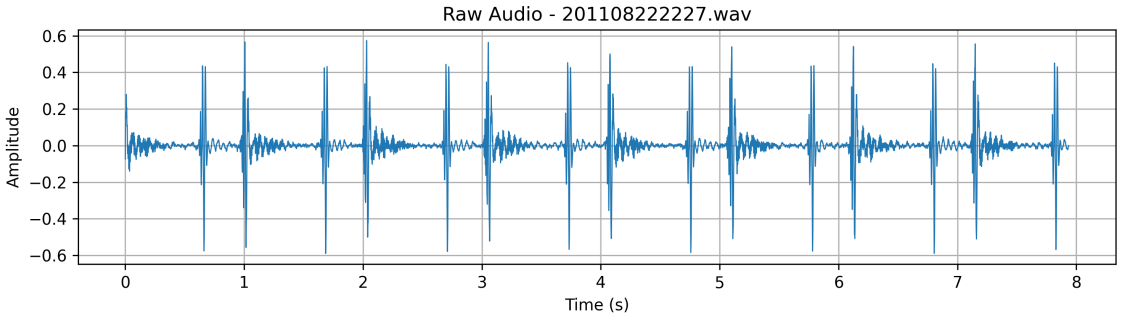

Figure 3: Raw audio waveform resampled at 16,000 Hz for visualization.

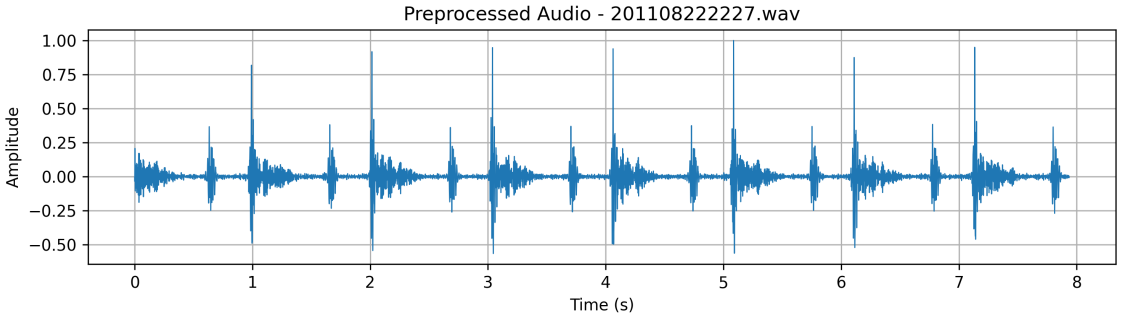

Figure 4: Preprocessed audio waveform after adaptive lowcut filtering, also resampled at 16,000 Hz.

This process aims to automatically select the frequency range that maximizes signal clarity and rhythmic regularity.

Although the visual difference between the raw and preprocessed signals is subtle, the enhanced definition of peaks improves downstream processing. Future work may explore tuning the filter parameters to allow for slightly more aggressive noise suppression, particularly in noisier datasets.

2.2 Segmentation Application

After normalization, each audio file undergoes a four-stage segmentation process designed to extract heartbeats — specifically pairs of peaks corresponding to the S1 and S2 cardiac events. The stages are designed to progressively relax constraints and apply increasingly aggressive filtering strategies to handle recordings of varying quality. If an audio file fails to yield a sufficient number of valid segments in the earlier stages, later stages fall back to more general techniques.

The goal across all stages is to extract short segments of audio that contain a clean S1–S2 pair, which can later be used for classification or feature extraction. The required number of valid segments (denoted X) per stage is a configurable setting provided in the parameter file. Figure 5 illustrates the logic flow of the segmentation process.

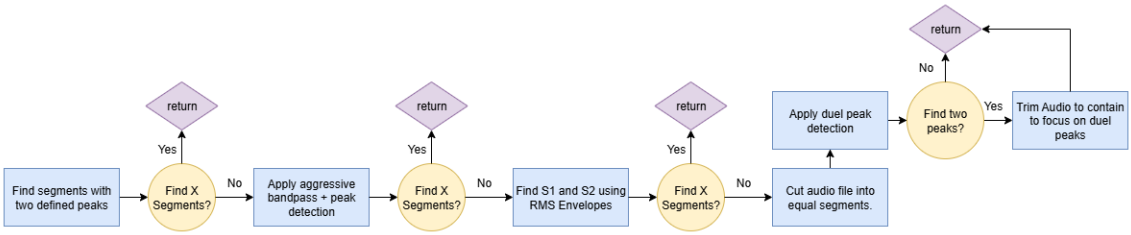

Figure 5: Diagram displaying the logic of the four segmentation stages.

The segmentation function operates as follows:

Stage 1: Peak-Based Segmentation

The first stage applies a basic peak detection on the absolute value of the filtered waveform. Candidate peaks are selected based on a prominence threshold. Intervals between adjacent peaks are evaluated to determine whether they plausibly represent an S1–S2 pair. To ensure reliability, I apply the same MAD-based outlier rejection discussed in the previous section to remove anomalous peak heights before computing inter-peak distances. If the number of valid segments falls below the threshold X, the pipeline proceeds to Stage 2.

Stage 2: Aggressive Bandpass Refinement

If Stage 1 fails, the audio undergoes an aggressive bandpass filter (typically 100–1000 Hz) to isolate a tighter frequency range where heart sounds are more prominent. Peak detection and MAD-based filtering are repeated on this bandpassed signal to extract new candidate segments. If the minimum count of valid S1–S2 pairs is still unmet, the process advances to Stage 3.

Stage 3: RMS Envelope Analysis

In Stage 3, I shift from peak detection in the waveform to using the **RMS energy envelope** of the signal. The RMS is computed over overlapping windows (e.g., 100 ms), producing a smoother amplitude representation:

$$\text{RMS}(t) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} y_i^2}$$

where N is the number of samples in each frame. Peaks in the RMS envelope are detected, and their positions are mapped back to the waveform to identify S1–S2 pairs using the same interval and context-based rules as in earlier stages.

Stage 4: Uniform Chopping + Peak Refinement

If the file still lacks sufficient segments, Stage 4 applies a brute-force fallback: the audio is divided into uniform, fixed-length windows (e.g., 1.2 seconds), using a specified stride. Segments with RMS energy below a threshold are discarded to avoid silence or noise.

Each remaining segment is further refined by detecting the first two prominent peaks (if available) and trimming the segment to include a small context window around them. If only one peak is found, the full segment is retained. This ensures every segment passed from Stage 4 contains at least one beat, and preferably a full cardiac cycle.

Outcome and Utility

The output is a list of short audio segments centered around S1–S2 peak pairs, as well as the stage number at which segmentation succeeded. This staged approach balances robustness with adaptability: higher-quality recordings tend to succeed early with minimal filtering, while noisier or ambiguous files are still recoverable through more general heuristics.

This segmentation pipeline ensures that downstream processing — such as feature extraction or classification — operates on clean, heartbeat-aligned clips that are more likely to reflect physiological patterns rather than noise or recording artifacts. At the same time, the design allows full-length, artifact-heavy audio files to pass through Stage 4, ensuring that the training dataset retains representative examples of noisy or degraded inputs. This balance supports both accurate modeling of healthy cardiac cycles and effective learning of edge cases or failure conditions.

2.3 Segmentation Examples

As each audio file undergoes segmentation, the stage at which it is successfully processed is recorded, along with the number of valid segments extracted. These results are visualized to provide insight into how different types of audio behave under the segmentation pipeline.

For recordings containing healthy or murmur cardiac events, it is expected that most files are successfully segmented within Stages 1–3, since these signals typically contain clear and regular peaks. In contrast, files dominated by artifacts — such as noise, distortion, or corrupted signals — are more likely to be segmented only at Stage 4, where uniform chopping is applied.

The distribution of segmentation stages and corresponding segment counts for each category are shown in Figures 6, 7, and 8.

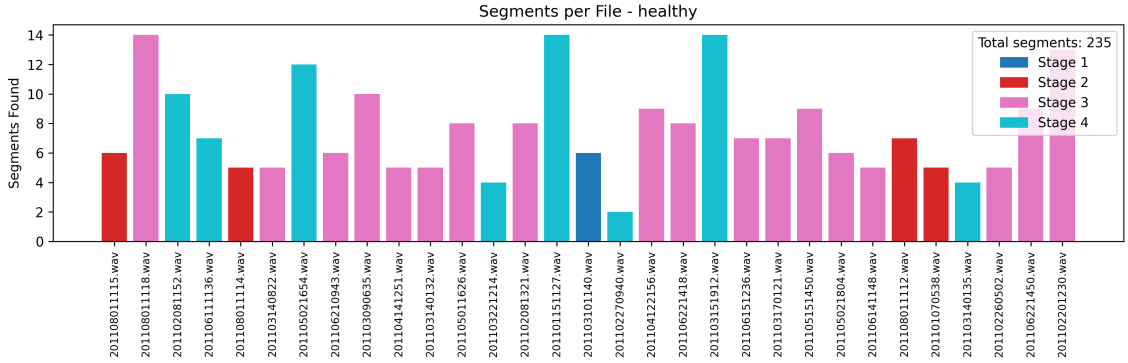

Figure 6: Segment count and the processing stage for each healthy audio file.

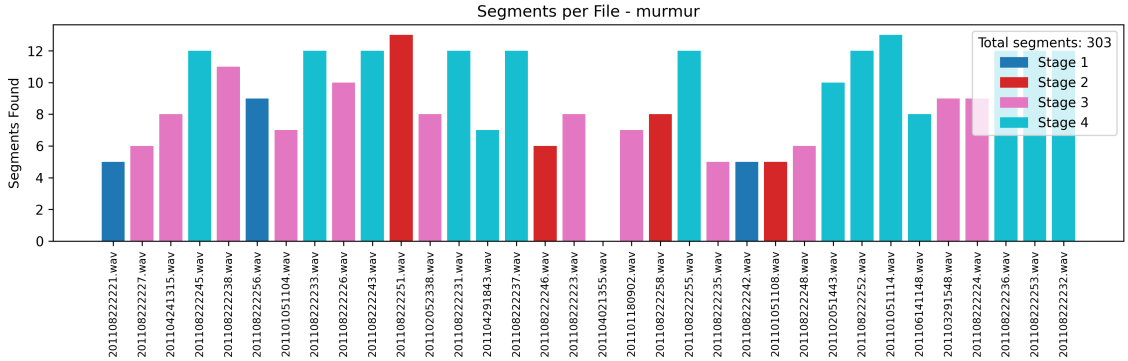

Figure 7: Segment count and the processing stage for each murmur audio file.

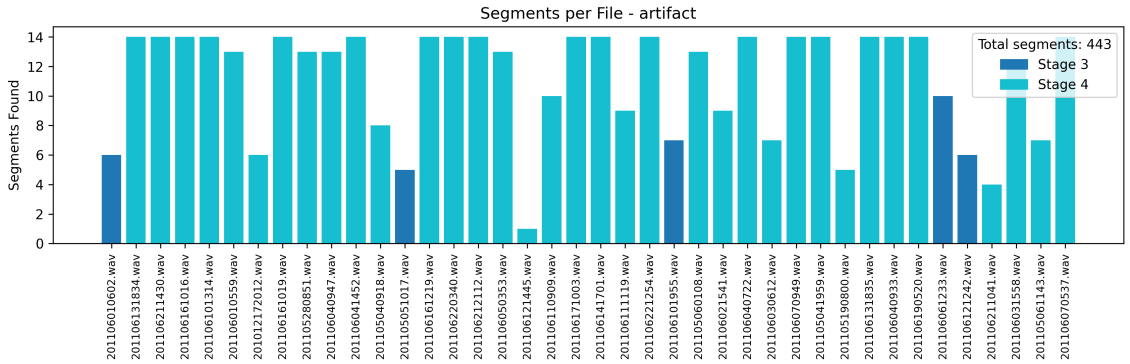

Figure 8: Segment count and the processing stage for each artifact audio file.

The following examples illustrate representative waveform and segment results for files processed at each of the four stages.

Stage 1 Example (Healthy)

Figures 9 and 10 show the full waveform and one extracted segment from the healthy audio file `201103101140.wav`, successfully segmented in Stage 1.

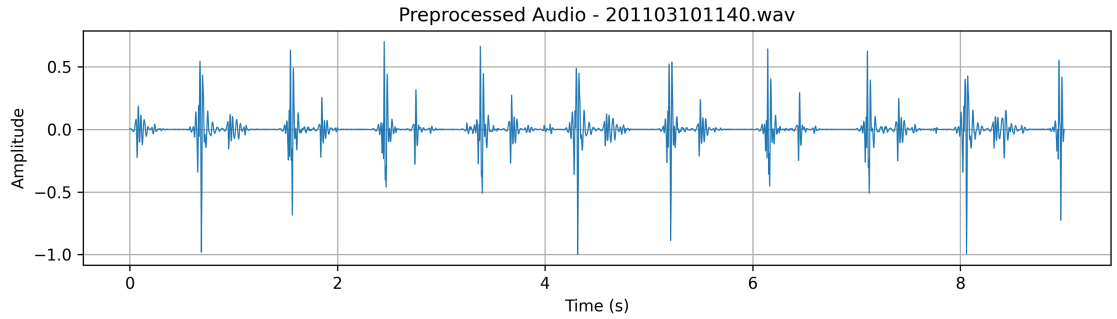

Figure 9: Raw audio waveform for the healthy file 201103101140.wav.

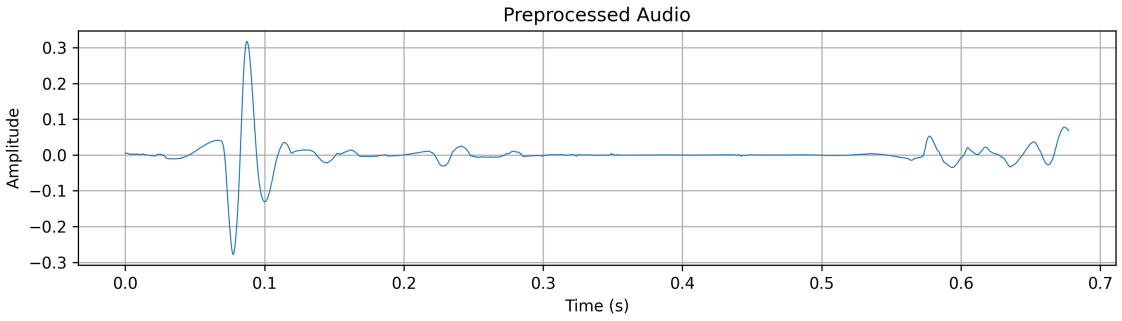

Figure 10: A single segment processed through Stage 1 for the healthy audio file 201103101140.wav.

Stage 2 Example (Murmur)

Figures 11 and 12 show the murmur audio file `201108222251.wav`, which required Stage 2's aggressive bandpass filtering to produce valid segments.

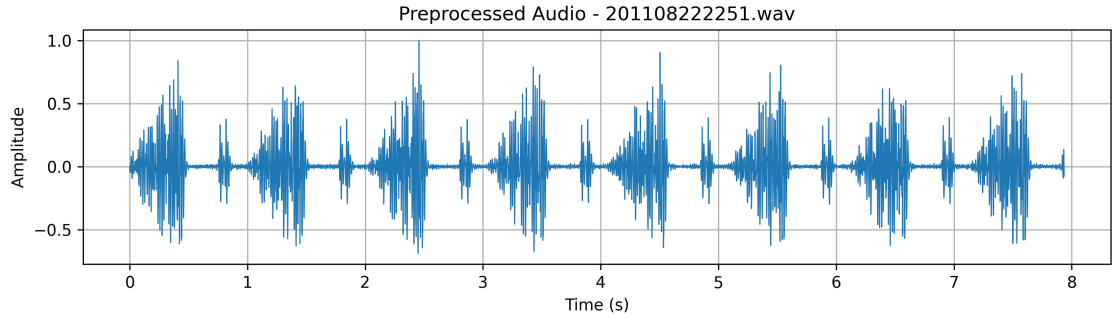

Figure 11: Raw audio waveform for the murmur file 201108222251.wav.

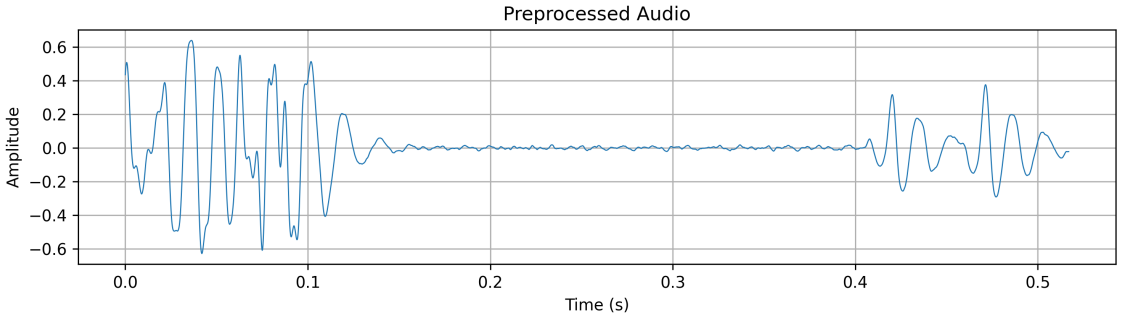

Figure 12: A single segment processed through Stage 2 for the murmur audio file 201108222251.wav.

Stage 3 Example (Healthy)

Figures 13 and 14 present a healthy file `201108011118.wav` processed using the RMS envelope method in Stage 3.

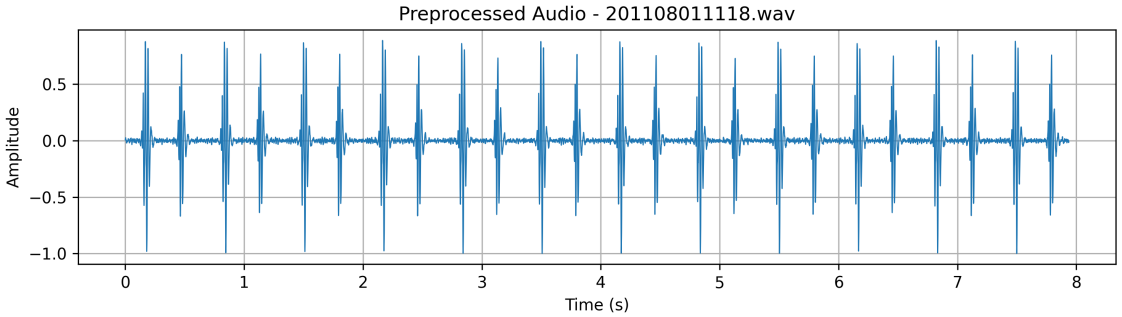

Figure 13: Raw audio waveform for the healthy file 201108011118.wav.

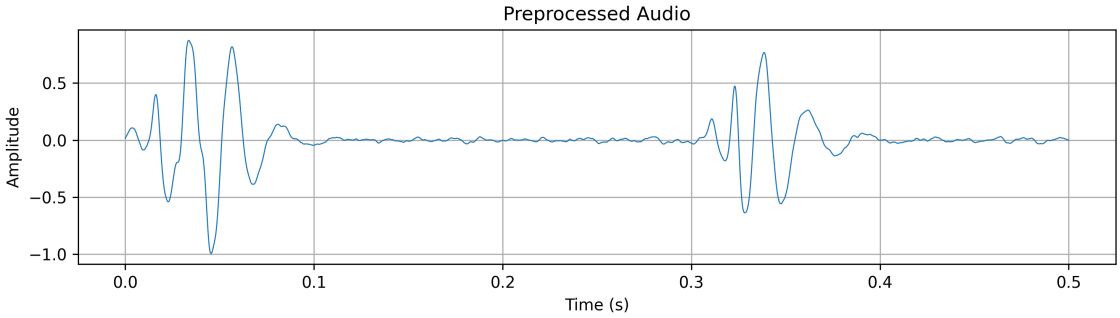

Figure 14: A single segment processed through Stage 3 for the healthy audio file 201108011118.wav.

Stage 4 Example (Healthy)

Figures 15 and 16 show the waveform and segment results from `201103151912.wav`, a healthy file processed only after falling back to uniform segmentation in Stage 4.

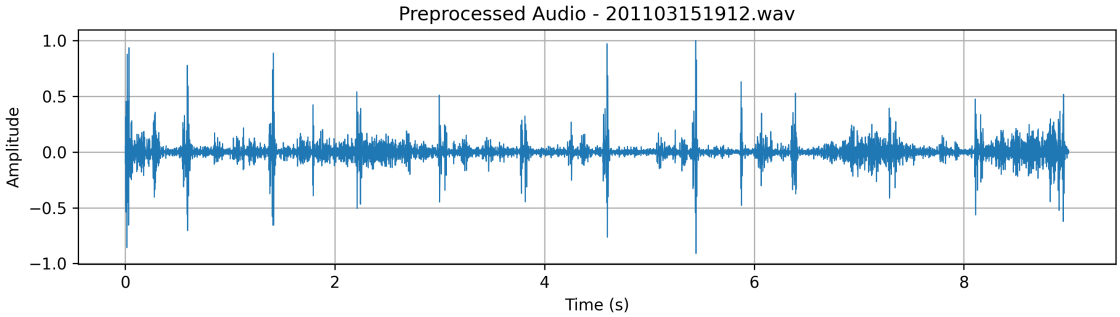

Figure 15: Raw audio waveform for the healthy file 201103151912.wav.

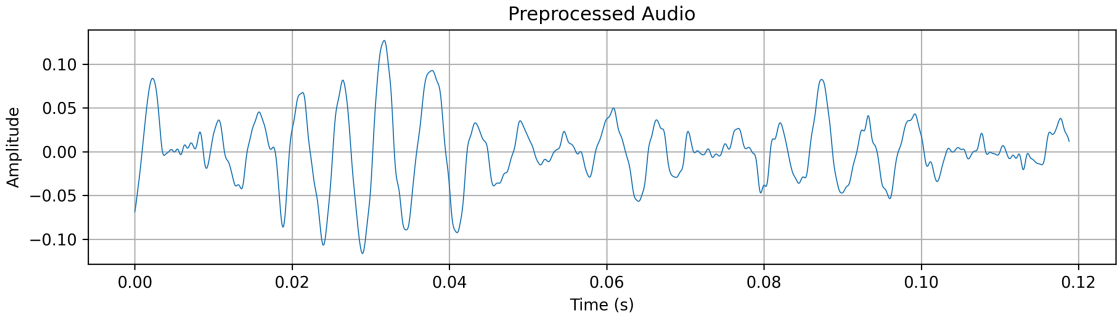

Figure 16: A single segment processed through Stage 4 for the healthy audio file 201103151912.wav.

2.3.1 Edge Cases

The segmentation process is not perfect — edge cases reveal potential improvements in parameter tuning and peak ordering logic.

Figure 17 shows an example from the same healthy file used earlier (Figure 9). Although this segment was extracted in Stage 1, the detected peaks appear in reverse order: the first has a lower amplitude (likely S2), and the second has a higher amplitude (S1). Since the current implementation does not differentiate between peak amplitudes when assigning S1 and S2, occasional misorderings can occur.

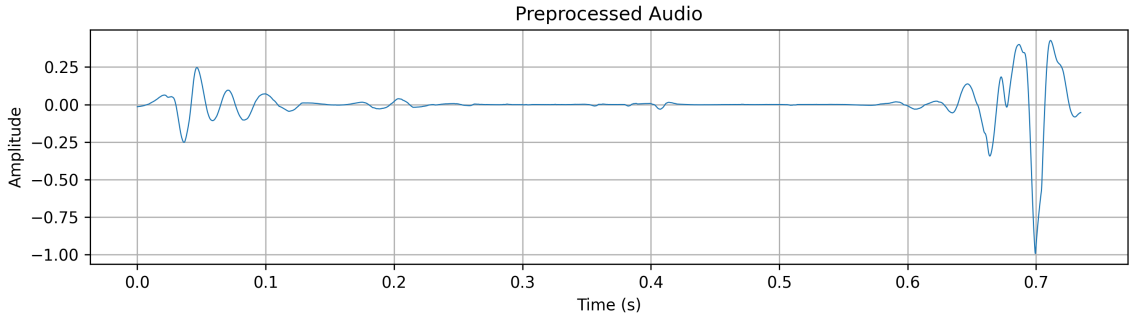

Figure 17: Segment from 201103101140.wav where the first detected peak is S2 and the second is S1.

As observed in Figure 8, most artifact-heavy files are processed only at Stage 4. Figures 18 and 19 show one such artifact file, `201012172012.wav`, and one of its extracted segments. The resulting waveform contains irregular structure and unclear peaks, but the fallback strategy still captures usable windows.

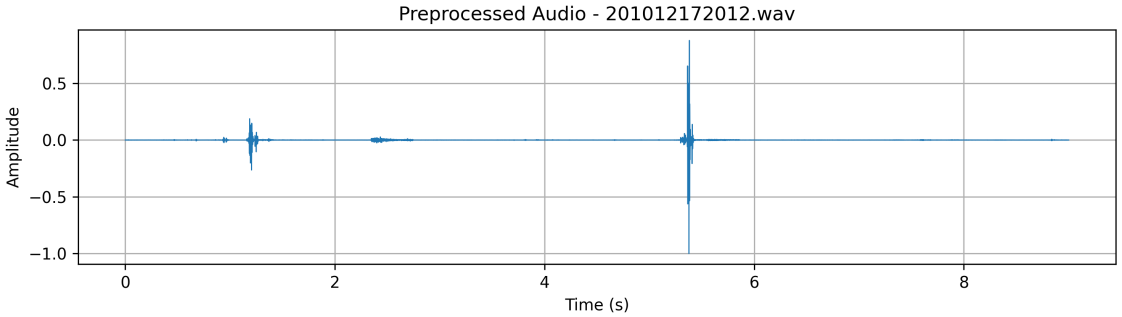

Figure 18: Raw audio waveform for the artifact file 201012172012.wav.

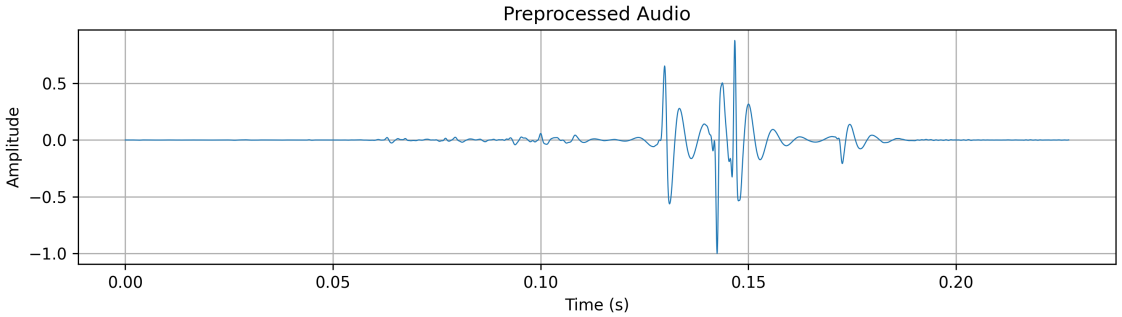

Figure 19: A single segment processed through Stage 4 for the artifact audio file 201012172012.wav.

2.4 Feature Extraction

Once heartbeat segments have been identified, a set of audio features is extracted from each segment to form the input for downstream machine learning models. Each segment is processed individually, and a feature dictionary is computed that includes multiple representations of the audio's temporal and spectral structure. The feature extraction pipeline produces the following: **Mel-Frequency Cepstral Coefficients (MFCC)**, **Mel spectrogram**, **Root Mean Square (RMS) energy**, and **Zero-Crossing Rate (ZCR)**. The extracted features are padded or truncated to uniform dimensions and saved in a compressed `.npz` archive for efficient loading during training.

Mel-Frequency Cepstral Coefficients (MFCC)

MFCCs are the primary features used for training the current classification models. They provide a compressed, perceptually-relevant representation of the signal's spectral envelope and are particularly effective for capturing rhythmic and tonal qualities in heart sounds. Importantly, MFCCs preserve *timing information* through their sequential structure, making them suitable for modeling periodic cardiac events such as S1 and S2.

To compute MFCCs, the signal is first transformed into a Mel spectrogram $M(f, t)$, where f represents Mel-scaled frequency bins and t represents time frames. Then, a discrete cosine transform (DCT) is applied to the log-scaled Mel spectrogram:

$$\text{MFCC}_k(t) = \sum_{n=1}^{N} \log(M(n, t)) \cdot \cos\left[\frac{\pi k}{N}(n - 0.5)\right]$$

where k is the cepstral coefficient index, and N is the number of Mel bands. In this implementation, the first 13 MFCCs are retained per frame. Each segment thus produces a matrix of shape $(13, T)$, where T varies with the segment's duration and sampling rate.

Mel Spectrogram

The Mel spectrogram represents energy in perceptually spaced frequency bands. Unlike linear spectrograms, Mel spectrograms mimic the human ear's sensitivity by compressing high-frequency resolution:

$$M(f, t) = \sum_{k=1}^{K} |X(k, t)|^2 \cdot H_f(k)$$

where $X(k, t)$ is the short-time Fourier transform (STFT) of the signal and $H_f(k)$ are triangular Mel filterbanks. The result is log-scaled using `librosa.power_to_db()` for improved contrast.

Root Mean Square (RMS) Energy

RMS energy provides a framewise measure of signal loudness and is calculated as:

$$\text{RMS}(t) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} y_i^2}$$

where y_i are the audio samples in frame t of length N. RMS captures amplitude dynamics, helping distinguish low-energy heartbeats from background silence or noise.

Zero-Crossing Rate (ZCR)

ZCR is the rate at which the signal waveform crosses the zero axis. It is computed as:

$$\text{ZCR}(t) = \frac{1}{2N}\sum_{i=1}^{N-1} |\text{sign}(y_i) - \text{sign}(y_{i+1})|$$

ZCR provides insight into signal noisiness or the presence of high-frequency content, which can be indicative of mechanical noise, murmurs, or artifacts.

Uniform Feature Dimensions and Saved Format

Because raw audio segments vary in length, the feature matrices must be padded or truncated along the time dimension to ensure a consistent shape for batch processing. The padding routine ensures:

- MFCCs are reshaped to $(13, T_{\max})$
- Mel spectrograms to $(64, T_{\max})$
- RMS and ZCR to $(T_{\max})$

The maximum time dimension $T_{\max}$ is inferred automatically from the longest segment for each feature type. Zero-padding is applied when the actual sequence is shorter.

All extracted features are saved into a compressed NumPy archive using `np.savez_compressed()`, producing a single `.npz` file. The archive includes:

- `segment_ids`: unique IDs for each segment
- `labels`: string labels (e.g., `"healthy"`, `"murmur"`, `"artifact"`)
- `mfcc`, `mel`, `rms`, `zcr`: arrays of padded features

This structure provides a unified, memory-efficient dataset ready for model training and validation. Currently, models are trained using only the MFCC input, but future architectures may explore combining MFCCs with secondary features to improve robustness.

3 Model Training and Architecture

With features extracted and segmented from audio recordings, the next step is to train models capable of classifying cardiac signals into diagnostic categories such as *healthy*, *murmur*, or *artifact*. The learning task is treated as a supervised classification problem, where the input is a fixed-size matrix of audio features (e.g., MFCCs) and the output is a discrete label.

The goal of this training process is to develop models that can generalize well across patient conditions and recording environments — distinguishing between meaningful physiological signals and irrelevant or corrupted data. Given the structured, time-dependent nature of the MFCC inputs, convolutional neural networks (CNNs) are a natural fit. They are efficient, spatially aware, and capable of extracting localized temporal-frequency patterns from spectrogram-like inputs.

In addition to CNNs, I also explore models that explicitly model time-dependence, including Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures. These recurrent neural networks are well-suited for sequential data and are particularly valuable for detecting temporal rhythm, repetition, or long-range dependencies in cardiac signals. By unrolling the feature sequence frame by frame (e.g., one MFCC frame at a time), these models can capture temporal context that may be diluted or missed by purely convolutional layers.

Later sections compare the performance and generalization characteristics of CNN, LSTM, and GRU-based classifiers, highlighting trade-offs in training stability, expressiveness, and inference cost. While CNNs are faster and more compact, LSTM and GRU networks offer richer temporal modeling which may prove beneficial in ambiguous or borderline cases.

3.1 CNN Architectures

Convolutional Neural Networks (CNNs) are widely used in audio classification tasks due to their ability to learn spatially local patterns across both time and frequency dimensions. In the case of MFCCs, the input to a CNN is a 2D matrix of shape $(13, T)$, where 13 corresponds to the cepstral coefficients and T is the number of time frames. This representation is conceptually similar to an image and is well-suited to 2D convolution. Figure 20 is a diagram representing the CNN architecture. Note, the matrix elements are not 1:1 to the MFCC input in this diagram.

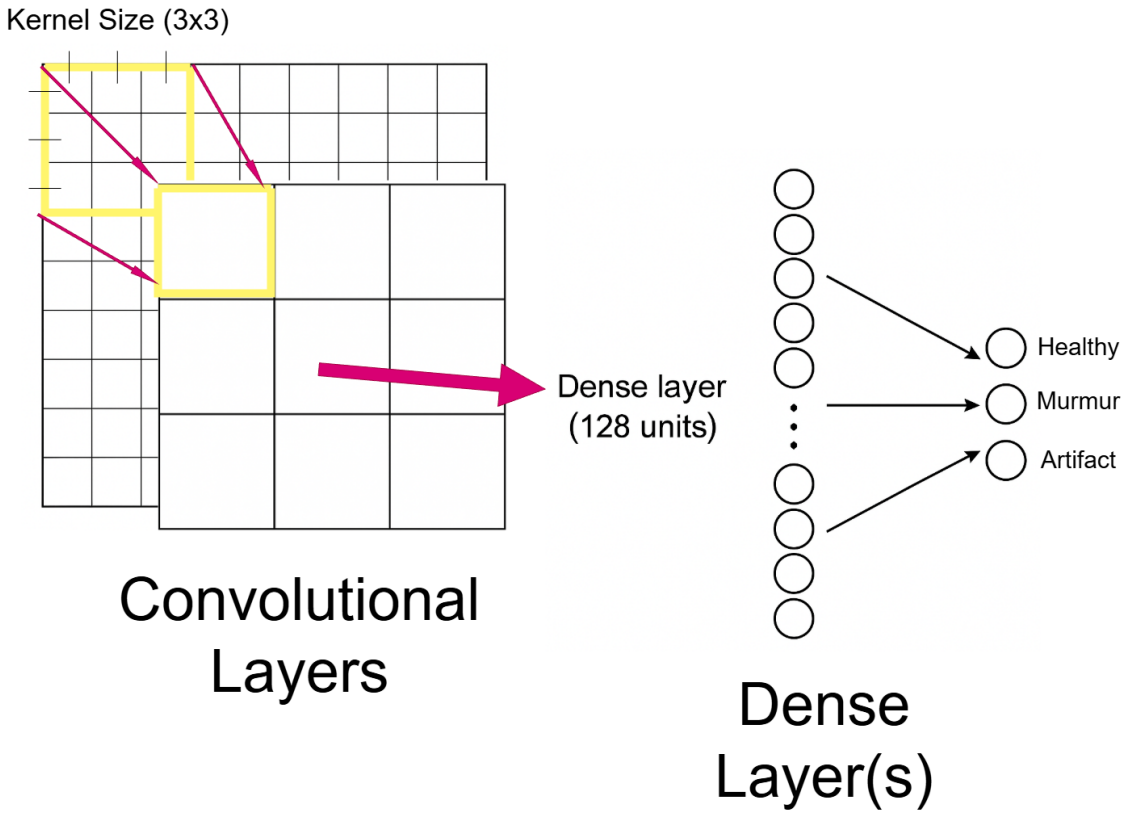

Figure 20: Two CNN blocks feeding into a flattened dense layer of 128 neurons that are then output into 3 classification nuerons.

Input Normalization and Format

Before feeding the feature matrices into the network, all MFCCs are zero-centered and optionally normalized along the feature axis. Inputs are reshaped to $(1, 13, T)$ to represent a single-channel 2D image, aligning with common CNN input conventions.

Baseline CNN Design

The baseline architecture follows a compact 2D CNN layout:

- **Conv Block 1:** `Conv2D(32, kernel=(3,3))` → `BatchNorm` → `ReLU` → `MaxPooling(2,2)`
- **Conv Block 2:** `Conv2D(64, kernel=(3,3))` → `BatchNorm` → `ReLU` → `MaxPooling(2,2)`
- **Flatten:** 2D output is flattened into a 1D vector
- **Dense Layer:** Fully connected layer with dropout
- **Output Layer:** Softmax classifier with 3 units (healthy, murmur, artifact)

This architecture is intentionally shallow to avoid overfitting on a small dataset, and dropout is used as a regularization mechanism. Kernel sizes are kept small (3x3) to capture fine-grained local structure in the MFCC representation. This architecture has proved to perform well for the objectives of this task, however, it can be changed easily within the parameters file.

Training Configuration

The model is trained using categorical cross-entropy loss and an Adam optimizer with an initial learning rate of 1×10^{-4}. Batch size, number of epochs, and early stopping patience are configurable parameters defined in the training configuration YAML file. Class imbalance is addressed either through loss weighting or dataset balancing during batch construction.

3.2 Recurrent Architectures: LSTM and GRU

While CNNs are effective at capturing spatial patterns in time-frequency representations such as MFCCs, they are inherently limited in modeling longer-term temporal dependencies. In cases where subtle changes in timing or rhythm are diagnostically relevant — such as variations in S1–S2 spacing or periodic murmur patterns — recurrent neural networks (RNNs) provide a complementary modeling strategy.

To address this, I implemented and evaluated two common recurrent architectures: **Long Short-Term Memory (LSTM)** and **Gated Recurrent Unit (GRU)** networks. Both are designed to process sequential data and preserve temporal state information over time, allowing the model to learn not just local features but also their progression across time steps.

LSTM Architecture

The LSTM cell augments a standard RNN with an internal memory state and three gates — input, forget, and output — that control the flow of information:

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Here, x_t is the input at time t, h_t is the hidden state, c_t is the cell state, and σ denotes the sigmoid activation. The cell learns when to forget previous information (f_t), when to accept new input (i_t), and how much of it to expose in the final output (o_t).

In the context of heartbeat classification, I feed each MFCC time frame as a timestep into the LSTM. The final hidden state is then passed through one or more fully connected dense layers — followed by dropout — before reaching the final softmax classification layer.

GRU Architecture

The GRU simplifies the LSTM by merging the forget and input gates into a single *update gate*, and by combining the hidden and cell states:

$$
\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\tilde{h}_t &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

GRUs are faster to train and require fewer parameters, making them an attractive choice when model size or inference time is constrained. Despite their simplicity, they often match or exceed LSTM performance in many sequence tasks.

Model Configuration

For both LSTM and GRU models, I use a single recurrent layer with 64 hidden units, followed by one or more fully connected dense layers with dropout, and finally a softmax output layer with three units. These models are trained using the same categorical cross-entropy loss and Adam optimizer configuration as the CNN models.

MFCC matrices are reshaped to sequences of shape $(T, 13)$ to match the expected input of recurrent layers. Each time step corresponds to one MFCC frame, and the temporal order of the frames is preserved.

Use Case and Trade-offs

LSTM and GRU models are particularly well-suited for capturing the dynamics of cardiac cycles over time, where slight irregularities or transient patterns may occur. While they require more computation per training step and may converge slower than CNNs, they offer the ability to model dependencies that span multiple time frames.

In this project, these architectures are evaluated alongside CNNs to better understand the trade-offs between spatial and temporal learning in heartbeat classification. Future work includes exploration of using different combinations of input data besides just using MFCC. We can also include different model architectures as they are easy to implement in this framework.

4 Model Evaluation

To assess the performance of each model architecture — including CNN, CNN-LSTM, and CNN-GRU — I evaluated both training dynamics and final classification performance on the held-out test set. The following metrics were used:

- **Training Loss and Accuracy:** Track optimization progress over epochs, allowing visual inspection of convergence and overfitting behavior.
- **Confusion Matrix:** Visualizes prediction distribution across classes, highlighting which categories are frequently misclassified.
- **Classification Report:** Provides detailed per-class metrics including precision, recall, F1-score, and support.

Each model was trained on the same preprocessed dataset using MFCC features. At the end of training, predictions were generated on the test split, and evaluation metrics were computed. Representative results are shown in the plots throughout this section.

Classification Report

The classification report offers a granular look at how each class performs. For each label — *artifact*, *healthy*, and *murmur* — the following metrics are calculated:

- **Precision**: The fraction of predicted positives that were correct.
- **Recall**: The fraction of actual positives that were correctly predicted.
- **F1-score**: Harmonic mean of precision and recall.
- **Support**: The number of true instances in the test set for each class.

An example classification report is shown below:

	precision	recall	f1-score	support
artifact	0.91	0.92	0.92	89
healthy	0.77	0.83	0.80	52
murmur	0.85	0.78	0.82	65
accuracy			0.85	206
macro avg	0.84	0.84	0.84	206
weighted avg	0.86	0.85	0.85	206

These results indicate that the model performs particularly well on the *artifact* class, with slightly lower performance on the *healthy* and *murmur* classes — which may reflect higher variability in physiological signals compared to recording artifacts.

Visual Metrics

Training curves for loss and accuracy are plotted across epochs for each model type to assess convergence behavior. Ideally, training and validation curves should converge toward similar values, indicating low variance and minimal overfitting.

The confusion matrix further highlights specific failure modes — for example, whether murmurs are frequently misclassified as healthy beats, or whether artifacts are mistaken for physiological patterns. These matrices are particularly helpful for identifying systemic class-level weaknesses.

4.1 CNN Evaluation

The performance of each model trained heavily relies on the session's hyperparameters. It's entirely possible that better results could be achieved by tuning these parameters — learning rate, number of filters, dropout rate, etc. — but the results shown here are representative enough for the current discussion.

Figure 21 shows the training history of the CNN model. Both training and validation loss stabilize quickly, with the validation loss tracking the training loss fairly closely. As expected, the validation accuracy underperforms slightly, since the model sees those samples less often and has no opportunity to directly optimize on them. However, the convergence is steady and not symptomatic of overfitting.

Figure 21: Training history of the CNN model.

Figure 22 shows the CNN model's confusion matrix. Interestingly, the model demonstrates stronger recall for murmur samples than healthy ones. This may indicate that murmur-specific spectral patterns are more consistent and therefore easier to learn. It also suggests that further refinement of the preprocessing pipeline — particularly for healthy signals — could lead to improved overall balance in classification performance.

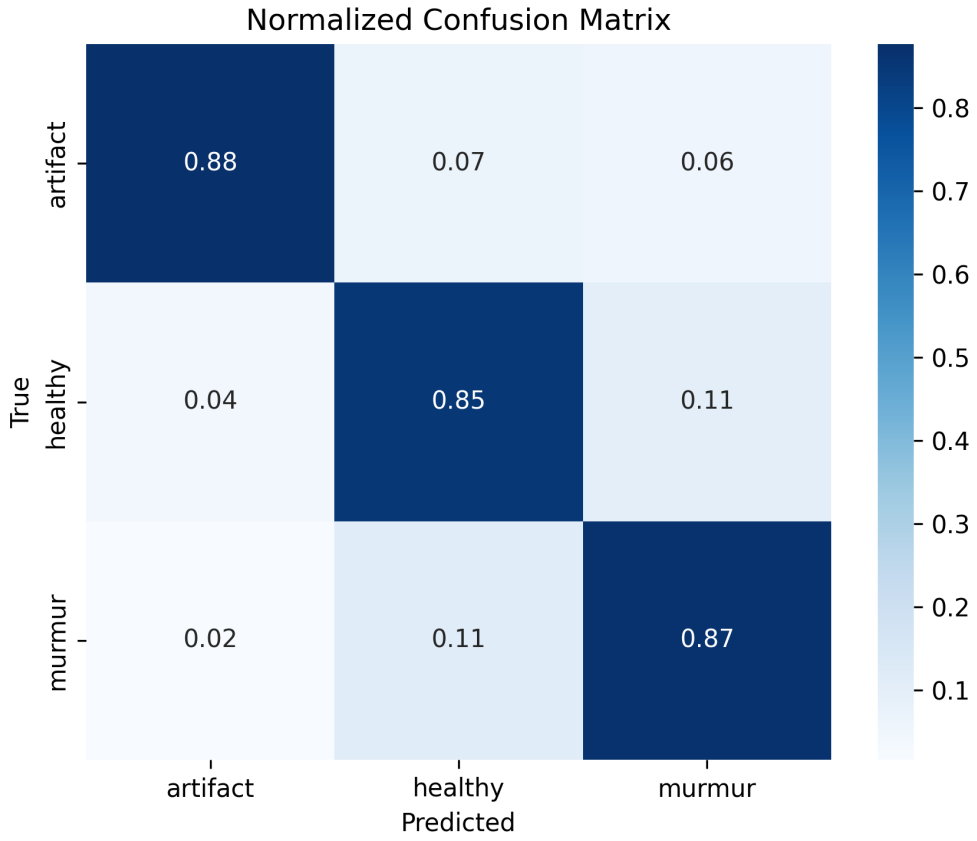

Figure 22: The confusion matrix of the CNN model.

Below is the corresponding classification report:

	precision	recall	f1-score	support
artifact	0.96	0.88	0.92	89
healthy	0.75	0.85	0.80	47
murmur	0.84	0.87	0.85	61
accuracy			0.87	197
macro avg	0.85	0.87	0.86	197
weighted avg	0.88	0.87	0.87	197

The overall accuracy of 87% is strong, with high F1-scores across all classes. The model excels at detecting artifacts, while showing slightly more variability when classifying healthy and murmur recordings.

4.2 LSTM Evaluation

Figure 23 shows the training history for the CNN-LSTM model. The trend is similar to the CNN model, with clean convergence for both loss and accuracy. The validation curve again lags slightly behind the training curve, as expected, but remains stable.

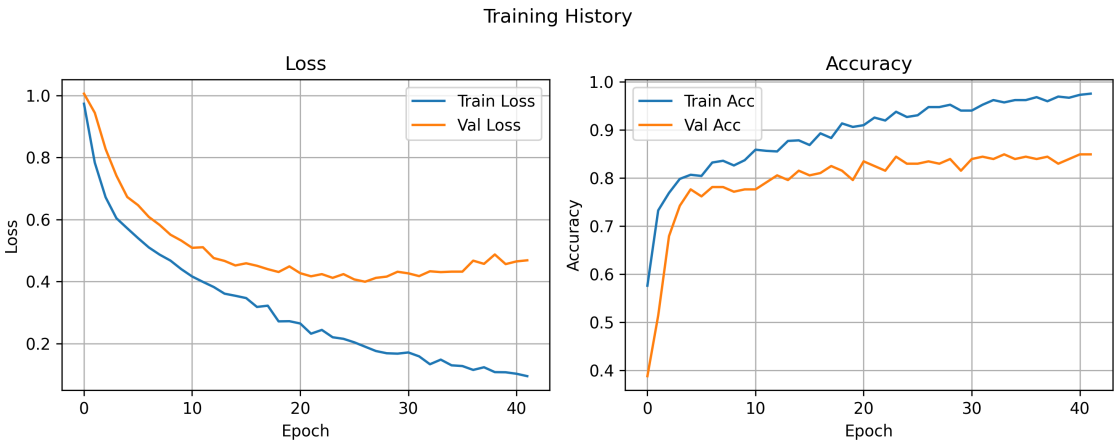

Figure 23: Training history of the CNN-LSTM model.

The confusion matrix in Figure 24 shows that the CNN-LSTM model performs slightly worse than the CNN baseline in classifying healthy and murmur files, but it improves slightly in identifying artifact signals. This could be due to the LSTM layer's ability to capture rhythm or timing **inconsistencies** that are more prevalent in artifact recordings.

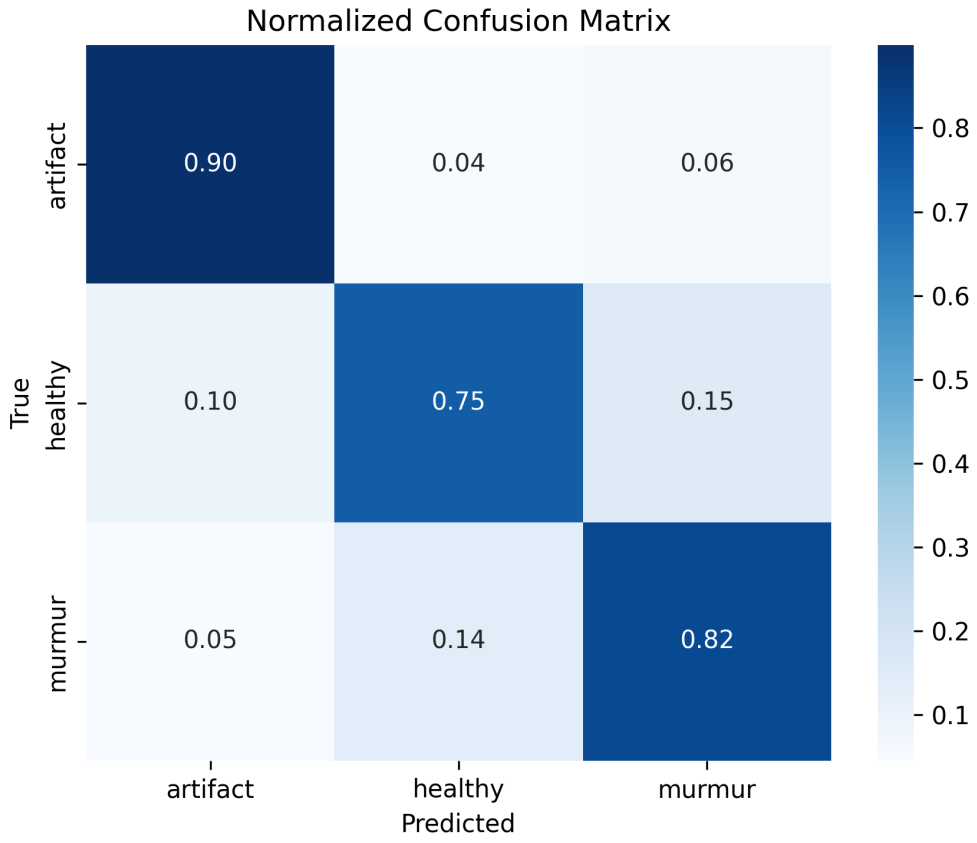

Figure 24: The confusion matrix of the CNN-LSTM model.

Classification report:

```
              precision    recall  f1-score   support
```

artifact	0.91	0.90	0.90	89
healthy	0.75	0.75	0.75	52
murmur	0.80	0.82	0.81	65
accuracy			0.83	206
macro avg	0.82	0.82	0.82	206
weighted avg	0.84	0.83	0.84	206

While performance is generally strong, the LSTM model slightly underperforms the CNN in overall accuracy and healthy-class F1-score. However, the differences are small, and the results show that the model can still learn useful temporal patterns.

4.3 GRU Evaluation

Figure 25 shows the training history of the CNN-GRU model. The convergence behavior is consistent with the other models — stable loss and steady accuracy across epochs.

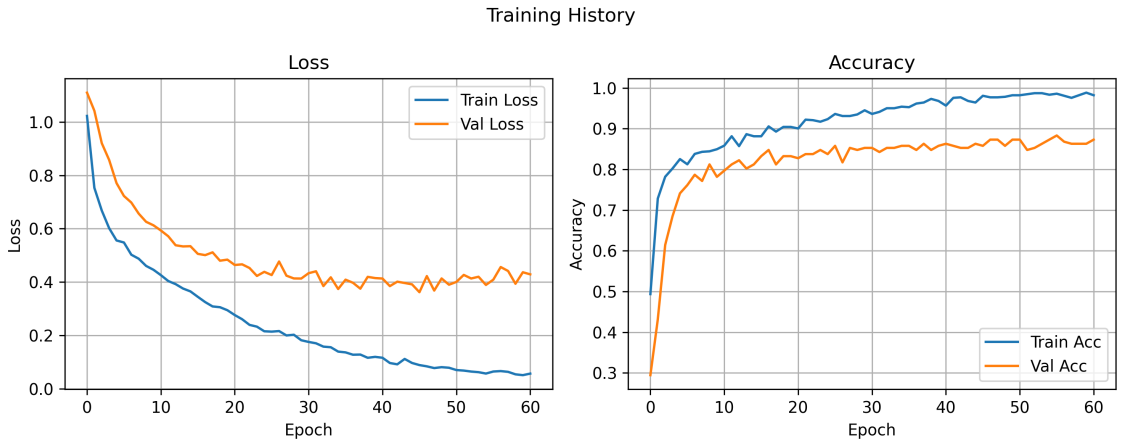

Figure 25: Training history of the CNN-GRU model.

The confusion matrix in Figure 26 shows that this model performs similarly to the CNN baseline. It improves slightly in classifying healthy files but falls just short in artifact and murmur recall. The differences are minimal, but notable.

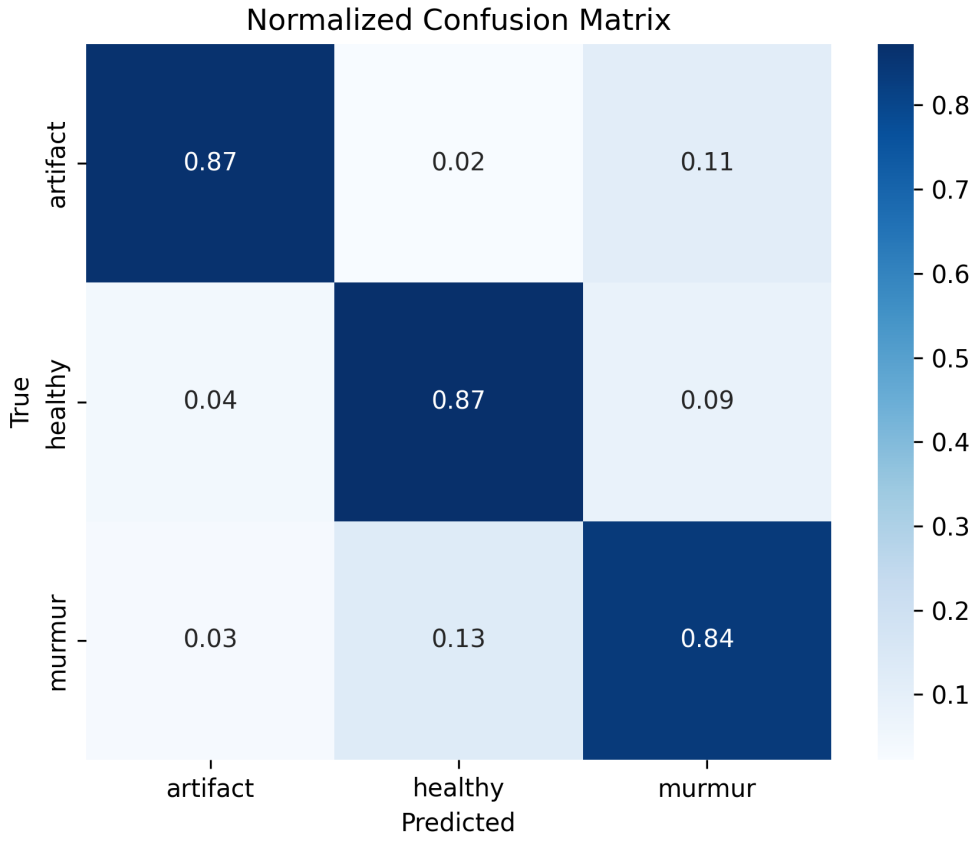

Figure 26: The confusion matrix of the CNN-GRU model.

Classification report:

	precision	recall	f1-score	support
artifact	0.95	0.87	0.91	89
healthy	0.80	0.87	0.84	47
murmur	0.78	0.84	0.81	61
accuracy			0.86	197
macro avg	0.85	0.86	0.85	197
weighted avg	0.86	0.86	0.86	197

The CNN-GRU model achieves a strong balance between all three classes, with the highest F1-score for healthy samples across all architectures. This suggests that GRUs may be better suited to capturing the short-term rhythmic patterns of clean heartbeats compared to their LSTM counterpart.

4.4 Overall Discussion

All three models perform reasonably well, with CNN leading slightly in accuracy, LSTM slightly favoring artifact detection, and GRU excelling in healthy-class classification. These results reflect how architectural differences — especially those affecting temporal modeling — influence sensitivity to different types of audio variation.

It's worth noting that none of the models show signs of severe overfitting or instability. With further hyperparameter optimization, augmentation strategies, and potentially ensembling, performance could likely be improved even further.

Future experiments might also investigate per-class confidence distributions, examine misclassified examples in detail, or explore feature ablations to understand model sensitivities.

5 Final Discussion

The BRUIT framework — *Bioacoustic Recognition of Unhealthy and Intact Tones* — was developed as a flexible, modular, and fully configurable pipeline for cardiac audio classification. Throughout this project, BRUIT served not only as a tool for experimentation, but as a foundation for reproducible and extensible research. From audio preprocessing and segmentation, to feature extraction, model training, and evaluation, every component of the system is controllable through a single YAML configuration file, making it easy to adapt the pipeline to new data, features, or architectures.

One of the key strengths of BRUIT is its end-to-end nature. Raw `.wav` files can be passed into the system, and through a series of carefully designed stages — adaptive filtering, multi-stage segmentation, feature extraction, and classification — the framework produces interpretable predictions and diagnostic metrics. It also includes rich logging and optional plotting for each stage, which has proven invaluable for debugging and analysis.

The trained models — CNN, CNN-LSTM, and CNN-GRU — were all integrated directly into BRUIT's workflow, and each demonstrated strong performance despite a relatively small dataset. The CNN model consistently performed well across all metrics, while the recurrent models showed promising gains in temporal sensitivity, particularly for distinguishing healthy cycles and identifying noisy or artifact-heavy signals.

Beyond model training, BRUIT offers a solid baseline for future work. New architectures can be added with minimal friction. Additional features (e.g., delta MFCCs, chroma, spectral contrast) can be integrated into the current format. And the existing segmentation strategy, while already effective, can be improved with more sophisticated peak detection or learned alignment methods.

In short, BRUIT is more than a project — it's a platform. With more data, time, and experimentation, it can serve as a powerful engine for clinical-grade heartbeat recognition, anomaly detection, or even broader physiological sound analysis.