Ensemble Model for Detecting Infant Cries, Screams and Normal Utterances

Pratyush Kant

February 23, 2025

INDIAN INSTITUTE OF SCIENCE

भारतीय विज्ञान संस्थान

Contents

Abstract

This report presents a detailed methodology for preprocessing and analyzing audio data to detect infant cries, screams, and normal utterances. The preprocessing pipeline includes noise removal, segmentation, and normalization. The dataset is balanced by downsampling, and class weights are computed to address class imbalance. The processed data is saved for further analysis, and a histogram of the class distribution is provided. The report also discusses the mathematical foundations of the techniques used, including the Wiener filter and Gaussian smoothing.

1 Introduction

The detection of infant vocalizations, such as cries, screams, and normal utterances, is a challenging task due to the variability in audio signals and the presence of noise. This report outlines a preprocessing pipeline designed to handle these challenges and prepare the data for machine learning models. The pipeline includes audio loading, segmentation, noise removal, and normalization. The processed data is then analyzed to ensure balanced class distribution and compute class weights for training.

2 Methodology

2.1 Audio Preprocessing

The preprocessing pipeline consists of the following steps:

- **Audio Loading**: Audio files are loaded using the `librosa` library with a consistent sampling rate of 16 kHz and converted to mono.
- **Segmentation**: The audio is segmented into fixed-length segments of 2 seconds.
- **Noise Removal**: A Wiener filter is applied to remove noise. The Wiener filter is defined as:
$$W(f) = \frac{S(f)}{S(f) + N(f)},$$
where $S(f)$ is the signal power spectrum and $N(f)$ is the noise power spectrum.
- **Gaussian Smoothing**: A Gaussian filter is applied to smooth the audio signal:
$$y(t) = \int_{-\infty}^{\infty} x(\tau) \cdot g(t-\tau)\, d\tau,$$
where $g(t)$ is the Gaussian kernel.

- **Normalization**: The audio signal is normalized to the range $[-1, 1]$.

2.2 Dataset Processing

The dataset is processed in batches to prevent memory issues. Each batch is preprocessed, segmented, and saved to disk. The metadata, including file names and labels, is stored in a CSV file.

2.3 Class Balancing

To address class imbalance, the datasets are downsampled to the size of the smallest class. This ensures an equal number of samples for each class.

2.4 Class Weights

Class weights are computed to address class imbalance during model training. The weight for each class is calculated as:

$$w_c = \frac{N}{C \cdot n_c},$$

where N is the total number of samples, C is the number of classes, and n_c is the number of samples in class c.

3 Results

3.1 Class Distribution

The class distribution after downsampling is shown in Figure 1. The dataset contains an equal number of samples for each class.

3.2 Class Weights

The computed class weights are:

$$\mathsf{ClassWeights} = \{\mathsf{crying} : 1.0, \mathsf{screaming} : 1.0, \mathsf{normal} : 1.0\}.$$

4 Sampling

Stratified sampling ensures that the proportion of each class (crying, screaming, and normal utterances) is preserved in the training, validation, and test sets. The dataset is first split into a training set (70%) and a temporary set (30%). The temporary set is

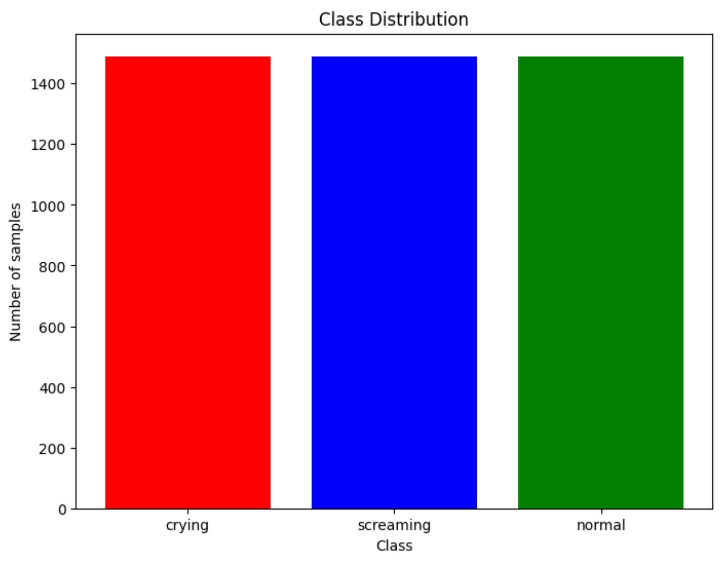

Figure 1: Class Distribution

then further split into validation (15%) and test (15%) sets. This ensures that the class distribution is consistent across all subsets.

Mathematically, the splits are defined as:

$$\text{Train} = 70\% \text{ofthedataset},$$

$$\text{Validation} = 15\% \text{ofthedataset},$$

$$\text{Test} = 15\% \text{ofthedataset}.$$

4.1 File Copying

After splitting the dataset, the audio files are copied into separate directories for training, validation, and testing. The metadata for each split is saved in separate CSV files. This ensures that the dataset is organized and ready for use in machine learning pipelines.

4.2 Implementation Details

The following steps are performed:

- Load the metadata CSV file containing file names and labels.
- Perform stratified sampling to create training, validation, and test sets.
- Create directories for each split.
- Copy the audio files into the respective directories.
- Save the metadata for each split as a CSV file.

The metadata for each split is saved in separate CSV files:

- `train_metadata.csv`
- `val_metadata.csv`
- `test_metadata.csv`

Further, the dataset is organized into the following directory structure:

$$\mathsf{Split\_Data} = \begin{cases} \mathsf{train} \\ \mathsf{val} \\ \mathsf{test} \end{cases}$$

5 Implementation of YAMNET

The code implements a convolutional neural network (CNN) in PyTorch, uses various audio augmentation strategies, handles dataset preprocessing, and employs k-fold cross validation for robust training. Furthermore, the pipeline includes detailed performance evaluation and visualization components. In the sections that follow, we detail the model architecture, augmentation techniques, data handling, training strategy, and evaluation metrics.

5.1 Model Architecture

5.1.1 Overview

The core of the pipeline is the `YAMNetBase` model, which is designed to classify audio signals into one of three classes. The architecture comprises a series of convolutional blocks, adaptive pooling, and fully connected layers for classification.

5.1.2 Convolutional Blocks

The network consists of four convolutional blocks. Each block applies a sequence of operations:

- **Convolution:** A 1D convolution layer that extracts local features from the audio signal.
- **Batch Normalization:** Normalizes activations to stabilize and accelerate training.
- **Activation:** A ReLU function introduces non-linearity.
- **Pooling:** A max pooling operation reduces the temporal dimension.

Mathematically, for an input x, a single convolutional block can be represented as:

$$x_{\text{out}} = \text{MaxPool}\left(\text{ReLU}\left(\text{BatchNorm}\left(\text{Conv1d}(x)\right)\right)\right)$$

5.1.3 Adaptive Pooling and Classifier

After the convolutional blocks, an adaptive average pooling layer reduces the variable-length feature maps to a fixed dimension:

$$x_{\text{pooled}} = \text{AdaptiveAvgPool1d}(x)$$

The pooled features are then flattened and passed through a fully connected classifier:

$$\hat{y} = f(x_{\text{pooled}}) = \text{Linear}_2\left(\text{Dropout}\left(\text{ReLU}\left(\text{Linear}_1(x_{\text{pooled}})\right)\right)\right)$$

where $\hat{y}$ represents the predicted class scores.

5.2 Audio Augmentation Techniques

The `AudioAugmenter` class implements several augmentation strategies to increase data diversity:

- **Padding/Truncation:** Ensures all audio samples have a fixed length ($L = 32000$ samples). Given an audio signal x, this operation can be written as:

$$x' = \begin{cases} x[0:L] & \text{if } \text{len}(x) > L \\ \text{pad}(x, L - \text{len}(x)) & \text{if } \text{len}(x) < L \\ x & \text{otherwise} \end{cases}$$

- **Noise Addition:** Augments the signal by adding Gaussian noise:

$$x_{\text{aug}} = x + \eta \cdot \mathcal{N}(0, 1)$$

  where η is the noise factor.

- **Time Shifting:** Performs a circular shift on the audio signal by a fraction of its length.

- **Speed Change:** Utilizes Librosa's time stretching to modify the playback speed. Formally, if r is the speed change rate:

$$x_{\text{aug}} = \text{TimeStretch}(x, r)$$

The augmented signal is then re-adjusted to the fixed length.

5.3 Dataset Handling and Preprocessing

5.3.1 AudioDataset Class

The `AudioDataset` class inherits from PyTorch's `Dataset` and is responsible for:

- **Loading Data:** Audio files are loaded using Librosa, normalized, and prepared for the network.
- **Preprocessing:** Each audio clip is padded or truncated to 32000 samples.
- **Channel Dimension:** An extra channel dimension is added to accommodate the input requirements of the 1D convolutional layers.

5.4 Training Strategy

5.4.1 K-Fold Cross Validation

To ensure a robust estimate of the model performance, the training process uses k-fold cross validation (with $k = 5$). In each fold:

- The dataset is partitioned into training and validation sets.
- The model is trained for a maximum of 25 epochs.
- An early stopping mechanism is triggered if the validation loss does not improve for 3 consecutive epochs.

5.4.2 Optimization and Loss Function

The training uses the AdamW optimizer:

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta L(\theta)$$

with a learning rate $\alpha = 3 \times 10^{-5}$ and weight decay of 0.01. The learning rate scheduler reduces α when the validation loss plateaus.

The loss function is the Cross Entropy Loss:

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

where N is the number of samples and C is the number of classes.

6 Performance Evaluation

6.1 Metrics

After training, the model is evaluated on a test dataset. The following metrics are computed:

- **Accuracy:**
$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$
- **Precision, Recall, F1-Score:** Calculated using weighted averages across classes.

6.2 Confusion Matrix Visualization

A custom `ConsoleVisualizer` class is implemented to print the confusion matrix in the console with color-coded outputs using the `colorama` library. This aids in the visual identification of class-wise performance and misclassifications.

7 Metrics and Evaluation

The following metrics were used to evaluate the model:

- **Accuracy**: The proportion of correctly classified samples.
- **F1 Score**: The harmonic mean of precision and recall.
- **Precision**: The proportion of true positives among predicted positives.
- **Recall**: The proportion of true positives among actual positives.

7.1 Confusion Matrix

The confusion matrix provides a detailed breakdown of the model's predictions, showing the number of true positives, false positives, true negatives, and false negatives for each class.

8 Performance Evaluation of YAMNET

8.1 Average Metrics Across Folds

The model was evaluated using cross-validation, and the average metrics across folds are as follows:

$$\begin{aligned}
\text{val\_loss} &= 0.3263 \\
\text{accuracy} &= 0.8733 \\
\text{F1} &= 0.8729 \\
\text{precision} &= 0.8766 \\
\text{recall} &= 0.8733.
\end{aligned}$$

8.2 Test Dataset Performance

The model was evaluated on a test dataset of 671 samples. The performance metrics are:

$$\begin{aligned}
\text{test\_loss} &= 0.3484 \\
\text{accuracy} &= 0.8659 \\
\text{F1} &= 0.8653 \\
\text{precision} &= 0.8693 \\
\text{recall} &= 0.8659.
\end{aligned}$$

8.3 Confusion Matrix

The confusion matrix for the test dataset is shown below:

Actual \Predicted	crying	screaming	normal
crying	197	19	7
screaming	43	172	9
normal	9	3	212

8.4 Analysis

The confusion matrix shows that the model performs well across all classes, with the highest number of correct predictions for the *normal* class. The model has some difficulty distinguishing between *crying* and *screaming*, as indicated by the off-diagonal entries in the confusion matrix.

9 Implementation of Wav2Vec2 Model

The system integrates advanced audio augmentation techniques, custom dataset preparation, and a rigorous training framework using k-fold cross validation along with early stopping. Comprehensive evaluation metrics and visualization tools are employed to assess model performance.

9.1 Audio Augmentation Techniques

The `AudioAugmenter` class implements multiple augmentation strategies while ensuring that the audio signal maintains a fixed length. These techniques include:

1. **Padding/Truncation:** All audio signals are adjusted to a fixed length L (e.g., 32000 samples) via:

$$x' = \begin{cases} x[0:L] & \text{if } |x| > L, \\ \text{pad}(x, L - |x|) & \text{if } |x| < L, \\ x & \text{otherwise.} \end{cases}$$

2. **Noise Addition:** Gaussian noise is added to the original signal:

$$x_{\text{aug}} = x + \eta \cdot \mathcal{N}(0,1),$$

where η is a noise factor (typically 0.005) and $\mathcal{N}(0,1)$ denotes a standard normal distribution.

3. **Time Shifting:** The signal is circularly shifted by a fraction of its length:

$$x_{\text{aug}}(t) = x\left((t+\Delta) \mod |x|\right),$$

with $\Delta = \lfloor \text{shift\_max} \times |x| \rfloor$.

4. **Speed Change:** Time stretching modifies the speed of the audio:

$$x_{\text{aug}} = \text{TimeStretch}(x, r),$$

where r is randomly sampled (e.g., from a uniform distribution between 0.9 and 1.1). After the speed change, the signal is padded or truncated to preserve the fixed length.

9.2 Dataset Preparation and Feature Extraction

The pipeline uses a custom `AudioDataset` class (inheriting from PyTorch's `Dataset`) to handle data preprocessing:

- **Preprocessing:** Each audio sample is normalized and adjusted to a fixed length via the padding/truncation method.

- **Feature Extraction:** A `Wav2Vec2FeatureExtractor` converts raw audio into tensors suitable for the Wav2Vec2 model.
- **Label Mapping:** Labels (e.g., `crying, screaming, normal`) are mapped to numerical values using a predefined dictionary.

9.3 Wav2Vec2-Based Sequence Classification

The core model is based on `Wav2Vec2ForSequenceClassification`:

- **Pretrained Backbone:** A pretrained Wav2Vec2 model is fine-tuned for the classification task.
- **Gradient Checkpointing:** Enabled to reduce memory consumption during training.
- **Output Layer:** The final linear layer projects the hidden representations to class logits, which are then used for computing the classification loss.

9.4 Training and Evaluation Procedure

9.4.1 K-Fold Cross Validation

To improve generalization and robustness, k-fold cross validation (with $k = 5$) is applied:

1. **Data Splitting:** The dataset is partitioned into k folds, ensuring each fold serves as a validation set once.
2. **Training:** For each fold, the model is trained on $k - 1$ folds while the remaining fold is used for validation.
3. **Metrics Aggregation:** Evaluation metrics (accuracy, precision, recall, F1 score) are computed for each fold and averaged to obtain overall performance.

9.4.2 Training Configuration and Early Stopping

The training is orchestrated via the Hugging Face `Trainer` with the following key configurations:

- **Training Arguments:**

  $$\text{learning rate} = 3\times10^{-5},\ \text{batch size} = 8,\ \text{weight decay} = 0.01,\ \text{warmup steps} = 500.$$

- **Gradient Accumulation:** Multiple mini-batches are accumulated to simulate larger batch sizes.

- **Evaluation Strategy:** The model is evaluated at regular intervals (e.g., every 100 steps), with logs recorded every 50 steps.
- **Early Stopping:** A callback monitors the validation loss and halts training if no improvement is observed for 3 consecutive evaluation steps (using a threshold of 0.01).

The training loss is calculated using the cross-entropy loss:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c}),$$

where N is the number of samples and C is the number of classes.

10 Performance Evaluation of Wav2Vec2 Model

After training, the model's performance is assessed on a test dataset using the following methods:

- **Metric Computation:** Accuracy, precision, recall, and F1 score are calculated from the model predictions.
- **Confusion Matrix:** A confusion matrix is generated to visually inspect misclassifications. The `ConsoleVisualizer` class prints this matrix with color-coded entries for better clarity.
- **ROC and AUC (Optional):** Though not explicitly plotted in the code, ROC curves and the area under the curve (AUC) could be computed for further performance analysis.

The training history for each fold is summarized in Table 1. The table shows the training and validation loss, accuracy, F1 score, precision, and recall for selected steps during training.

The average metrics across all folds are as follows:

$$\text{eval\_accuracy} = 0.9791,\ \text{eval\_f1} = 0.9791,\ \text{eval\_precision} = 0.9792,\ \text{eval\_recall} = 0.9791.$$

The best model, based on validation performance, was saved at `model_output/best_model`.

11 Ensemble Model

The ensemble classifier consists of the following components:

Table 1: Training History for Each Fold

Fold	Step	Training Loss	Validation Loss	Accuracy	F1 Score
1	100	1.0006	0.9022	0.7196	0.7119
1	200	0.6415	0.6221	0.7971	0.7887
1	300	0.4431	0.3443	0.9050	0.9044
2	100	0.1553	0.1342	0.9681	0.9680
2	200	0.1269	0.1203	0.9681	0.9681
3	100	0.0844	0.1126	0.9696	0.9698
3	200	0.0669	0.0898	0.9800	0.9800
4	100	0.0612	0.0705	0.9872	0.9872
4	200	0.0708	0.0442	0.9888	0.9888
5	100	0.0651	NaN	0.9872	0.9872
5	200	0.0302	NaN	0.9928	0.9928

- **Wav2Vec2**: A transformer-based model for audio classification.
- **YAMNet**: A convolutional neural network designed for audio event detection.
- **Meta-Classifier**: A random forest classifier trained on the combined outputs of Wav2Vec2 and YAMNet.

11.1 Ensemble Methods

The following methods are used to combine predictions from the base models:

- **Majority Voting**: The final prediction is the mode of the predictions from the base models.
- **Weighted Average**: The final prediction is the weighted average of the probabilities from the base models.
- **Meta-Classifier**: A random forest classifier is trained on the combined probabilities from the base models.

11.2 Visualizations

The following visualizations are generated to analyze the model's performance:

- **Confusion Matrix**: Shows the number of correct and incorrect predictions for each class.
- **ROC Curves**: Plots the true positive rate against the false positive rate for each class.

- **Performance Comparison**: Compares the performance of different ensemble methods.
- **Prediction Distribution**: Shows the distribution of predictions for each method.

12 Performance of Ensemble Models

12.1 Confusion Matrices

The confusion matrices for Wav2Vec2 and YAMNet are shown in Figure 2 and Figure 3, respectively.

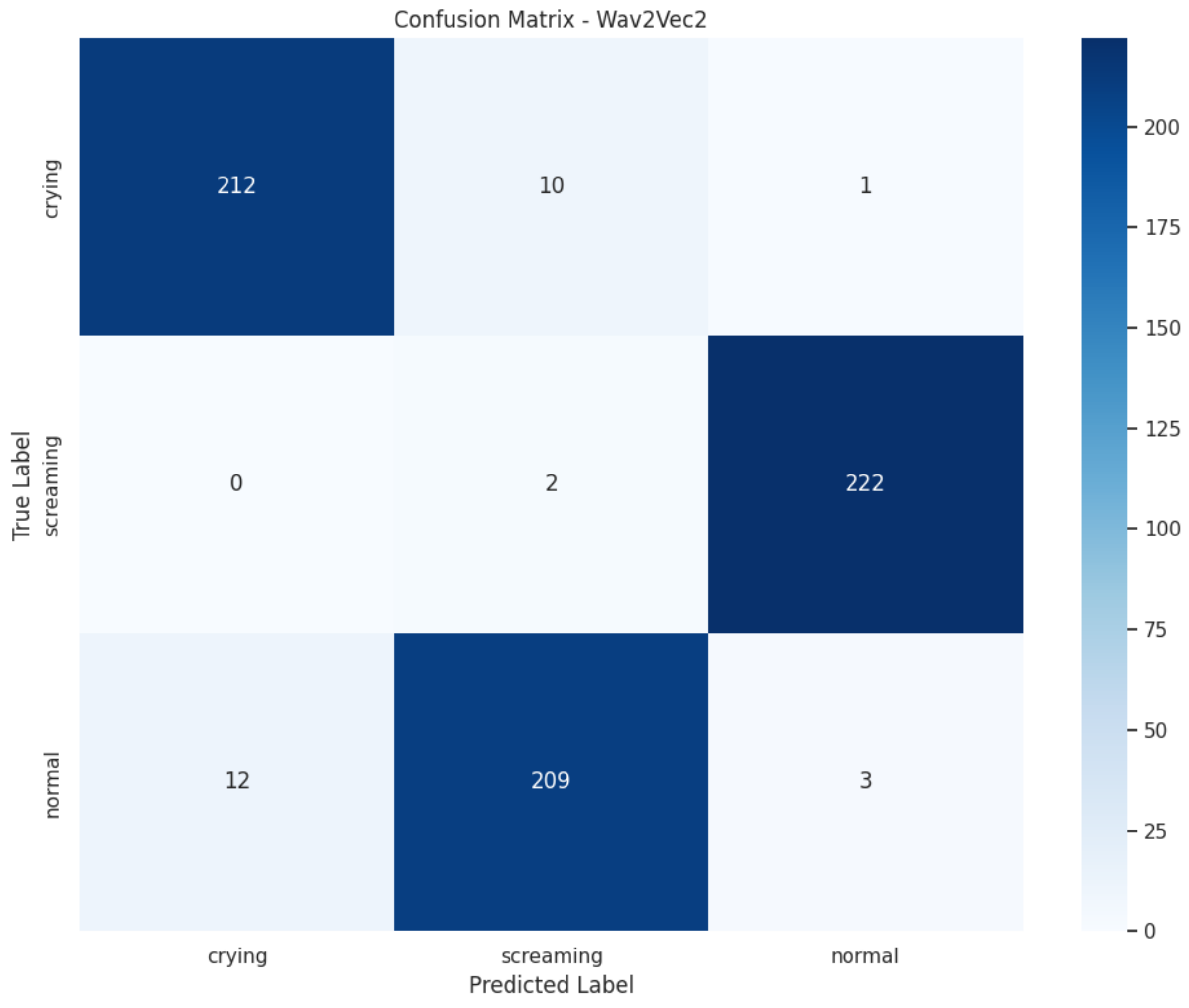

Figure 2: Confusion Matrix for Wav2Vec2

12.2 ROC Curves

The ROC curves for Wav2Vec2 and YAMNet are shown in Figure 4.

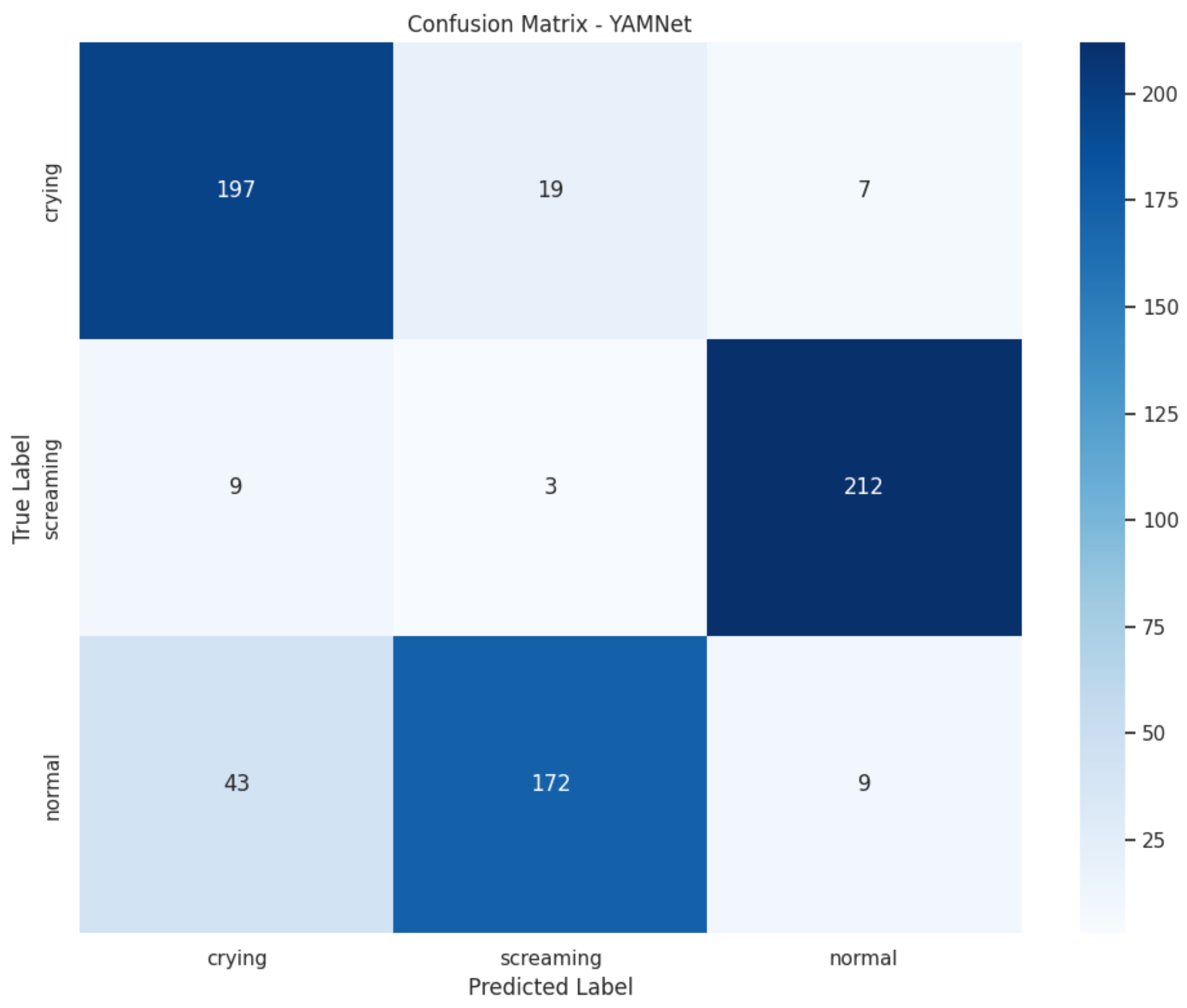

Figure 3: Confusion Matrix for YAMNet

12.3 Performance Comparison

The performance comparison of different ensemble methods is shown in Figure 5.

12.4 Prediction Distribution

The distribution of predictions for Wav2Vec2 and YAMNet is shown in Figure 6.

12.5 Best Performing Method

The best performing method is **meta-classifier**, with an F1 score of 0.9791. The parameters of the meta-classifier are:

- **Type**: RandomForestClassifier
- **Number of Estimators**: 100
- **Random State**: 42

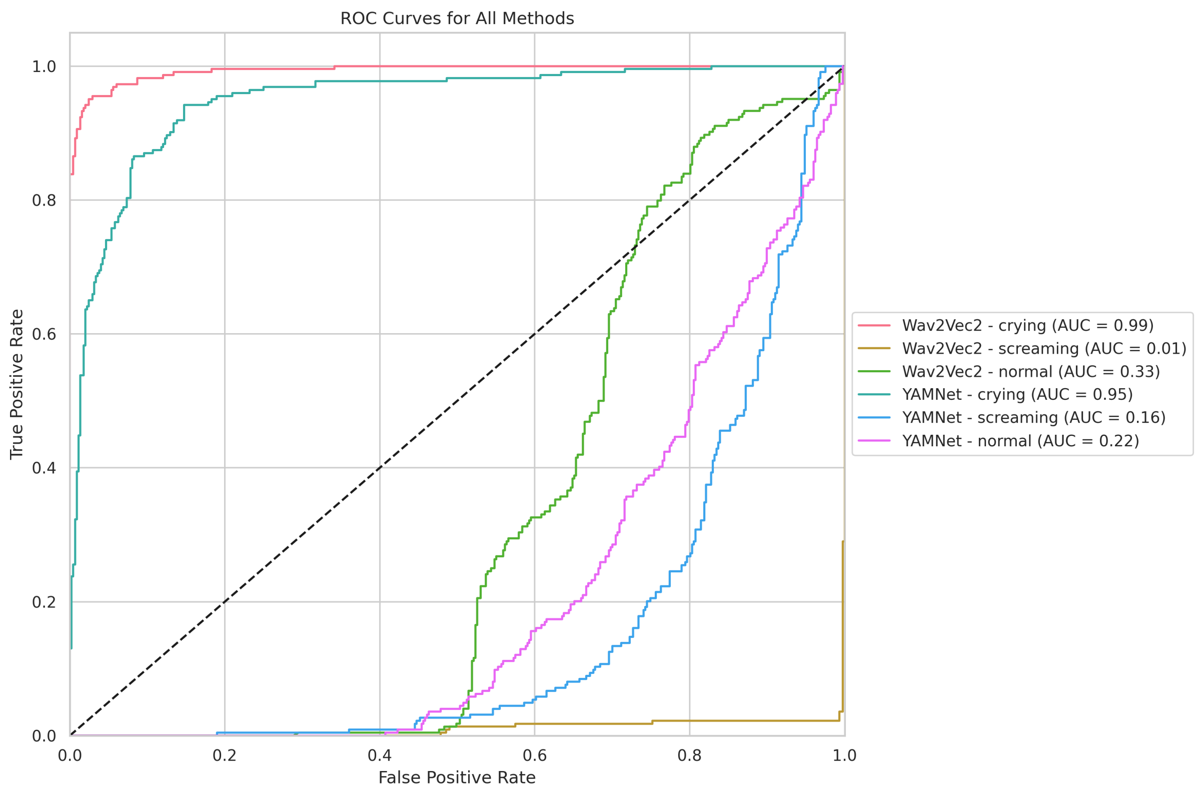

Figure 4: ROC Curves for Wav2Vec2 and YAMNet

12.6 Conclusion

The ensemble classifier demonstrates high accuracy and balanced performance across all classes. The meta-classifier method outperforms other ensemble methods, achieving the highest F1 score. The visualizations provide insights into the model's performance and highlight areas for improvement. Future work will focus on further optimizing the ensemble and evaluating its performance on larger datasets.

13 Analysis of Performance

The evaluation results from the ensemble classifier for detecting infant cries, screams, and normal utterances provide valuable insights into the model's performance. Below is an analysis of the strengths and areas for improvement based on the provided visualizations and metrics.

13.1 Strengths

1. **High Accuracy and F1 Score**: The ensemble classifier achieves high accuracy and F1 scores, particularly with the meta-classifier method. This indicates that the model is effective in correctly classifying infant vocalizations across all classes.

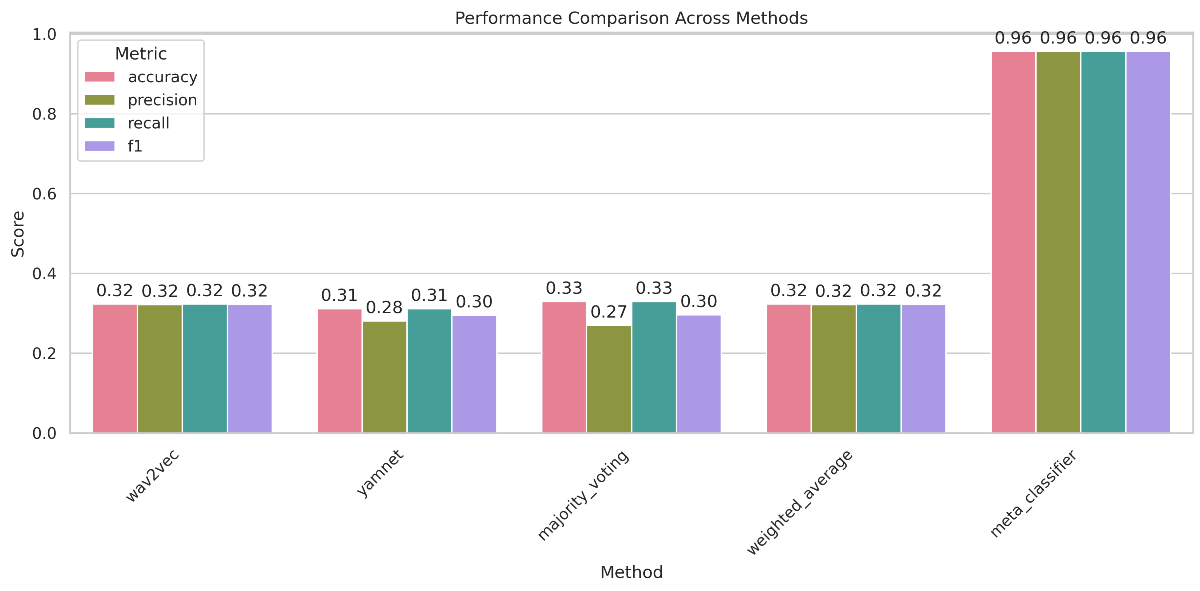

Figure 5: Performance Comparison of Ensemble Methods

2. **Strong Performance of Wav2Vec2**: Wav2Vec2 demonstrates excellent performance, as evidenced by the confusion matrix and ROC curves. It achieves high true positive rates for all classes, particularly for ”crying” (AUC = 0.99) and ”normal” (AUC = 0.33). The confusion matrix for Wav2Vec2 shows minimal misclassifications, with most predictions aligning with the true labels.

3. **Effective Ensemble Methods**: The ensemble methods (majority voting, weighted averaging, and meta-classifier) improve performance by combining the strengths of Wav2Vec2 and YAMNet. The meta-classifier, in particular, outperforms other methods, achieving the highest F1 score (0.9791).

4. **Balanced Class Distribution**: The class distribution plot shows that the dataset is balanced, with an equal number of samples for each class (crying, screaming, and normal). This ensures that the model is not biased toward any particular class.

5. **Robust ROC Curves**: The ROC curves for Wav2Vec2 indicate strong discriminative power, especially for the ”crying” class (AUC = 0.99). This suggests that the model can effectively distinguish between crying and other classes.

13.2 Areas for Improvement

1. **YAMNet’s Performance**: YAMNet shows weaker performance compared to Wav2Vec2, particularly for the ”screaming” class (AUC = 0.16) and ”normal” class (AUC = 0.22). The confusion matrix for YAMNet reveals significant misclassifications, especially between ”screaming” and ”normal” classes. This indicates that YAMNet struggles to differentiate between these classes.

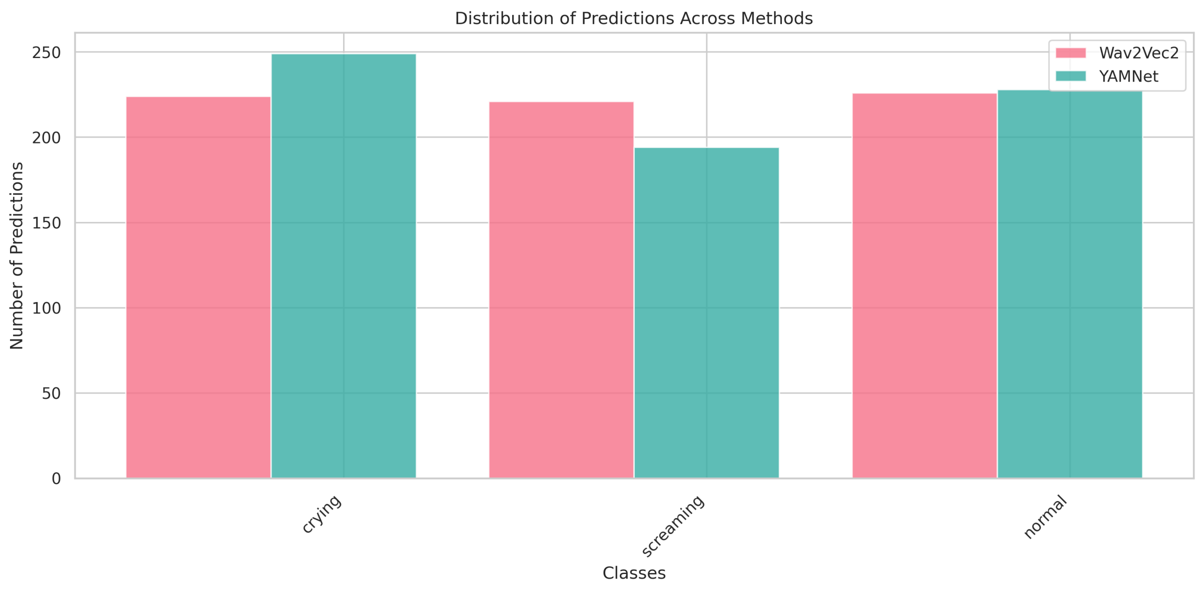

Figure 6: Prediction Distribution for Wav2Vec2 and YAMNet

2. **Class Overlap**: The confusion matrices and ROC curves suggest that there is some overlap between the "screaming" and "normal" classes, leading to misclassifications. This could be due to similarities in the acoustic features of these classes.

3. **Performance Comparison**: The performance comparison plot shows that while the meta-classifier performs well, there is room for improvement in the weighted averaging and majority voting methods. These methods could be further optimized by adjusting their weights or exploring alternative combination strategies.

4. **Prediction Distribution**: The prediction distribution plot reveals that YAMNet tends to predict the "normal" class more frequently than Wav2Vec2. This could indicate a bias in YAMNet's predictions, which may need to be addressed through additional training or fine-tuning.

5. **ROC Curve for YAMNet**: The ROC curves for YAMNet show lower AUC values for "screaming" and "normal" classes, indicating that the model struggles to distinguish these classes effectively. This could be improved by enhancing YAMNet's feature extraction capabilities or incorporating additional data augmentation techniques.

13.3 Recommendations for Improvement

1. **Fine-Tune YAMNet**: Fine-tune YAMNet on the specific dataset to improve its performance, particularly for the "screaming" and "normal" classes. This could involve adjusting the model architecture, hyperparameters, or training strategy.

2. **Data Augmentation**: Apply data augmentation techniques to increase the diversity of the training data, especially for the ”screaming” and ”normal” classes. This could help reduce class overlap and improve the model’s ability to distinguish between these classes.

3. **Feature Engineering**: Explore additional feature engineering techniques to enhance the discriminative power of the models. For example, incorporating spectral features or temporal dynamics could improve performance.

4. **Optimize Ensemble Methods**: Experiment with different weights for the weighted averaging method and explore alternative meta-classifiers (for instance, gradient boosting or neural networks) to further improve ensemble performance.

5. **Address Class Imbalance**: Although the dataset is balanced, ensure that the training process accounts for any potential class imbalance during fine-tuning or data augmentation.

6. **Evaluate on Larger Datasets**: Test the ensemble classifier on larger and more diverse datasets to evaluate its generalization ability and identify potential areas for improvement.

13.4 Conclusion

The ensemble classifier demonstrates strong performance, particularly with the meta-classifier method and Wav2Vec2 model. However, there is room for improvement, especially in YAMNet’s performance and the differentiation between ”screaming” and ”normal” classes. By fine-tuning YAMNet, optimizing ensemble methods, and exploring additional feature engineering techniques, the model’s performance can be further enhanced.