Multi-Head Binary Classification System for Synthetic Data Detection

Sabian Hibbs

Abstract

Detecting AI-generated (synthetic) data, such as AI-generated audio, is a critical challenge in modern machine learning. This white paper presents a multi-head binary classification system that distinguishes real data from synthetic data by leveraging an ensemble of neural network sub-models. The system integrates rigorous probabilistic modeling, a novel output aggregation strategy, and a comprehensive data processing pipeline. This document provides a detailed mathematical formulation of the classification model, an in-depth description of the associated scripts, and a comparative literature review, concluding with performance expectations and future directions.

1 Introduction

Detecting AI-generated (synthetic) data, such as AI-generated audio, is a critical challenge in modern machine learning. This white paper presents a **multi-head binary classification system** for distinguishing real data from synthetic data. The system is designed for AI-generated audio detection and consists of multiple neural network "heads" (sub-models) whose outputs are combined in a way that amplifies detection sensitivity. The approach integrates a rigorous probabilistic model, an ensemble of classifiers with a novel output aggregation strategy, and a comprehensive data processing pipeline. Each component—from data preprocessing to model inference—is modular, making the system flexible and suitable for inclusion in a project README or technical documentation.

The document is organized as follows: First, we provide a **mathematical formulation** of the classification model, detailing how probabilities are computed, which loss functions are used during training, how decisions are made (decision thresholds), and the rationale behind averaging the "real" logits across multiple heads. Next, we give **in-depth**

explanations of all project scripts, describing how each script contributes to the overall pipeline:

- `file_renamer.py` – ensures consistent unique filenames for audio data.
- `audio_converter.py` – normalizes audio format and sampling.
- `audio_augment.py` – generates augmented audio examples to expand the training set.
- `audio_segmenter.py` – chops audio files into fixed-length segments.
- `dataset_manager.py` – splits data into training and testing sets by class.
- `file_manager.py` – checks and fixes any overlap between training and testing sets (to prevent data leakage).
- `submodel_trainer.py` – trains individual classification sub-models (the "heads").
- `model_merger.py` – merges multiple trained sub-models into a unified multi-head ensemble model.
- `inference_runner.py` – runs the merged model on new data for inference, producing detection results.

We then present a **literature review**, comparing this multi-head approach to existing classification techniques and AI-generated audio detection frameworks. Finally, a **performance expectations** section discusses anticipated accuracy and efficiency benefits, given the system's ensemble architecture and modular pipeline design.

2 Multi-Head Classification Model: Mathematical Framework

The core of the system is a binary classifier implemented as an ensemble of neural network sub-models (multiple "heads"). Each sub-model is a deep neural network that independently attempts to classify an audio sample as **Real (authentic)** or **Synthetic (fake)**. By combining several sub-models, the system aims to improve robustness and detection accuracy through ensemble learning. In this section, we detail the model architecture and derive its probability estimates, loss function formulation, decision strategy, and explain the reasoning for the unique averaging of real logits.

2.1 Model Architecture and Probability Computation

Each sub-model is a convolutional neural network (CNN) that operates on audio features. In our implementation, each sub-model uses a *pretrained CNN backbone* (such as a ResNet-18) to extract features from an audio spectrogram, followed by a custom classifier head that outputs two logits corresponding to the two classes. Formally, for an input sample x (e.g., a log-mel spectrogram of an audio clip), a sub-model f_i (with parameters θ_i) produces a 2-dimensional output:

$$z_i = f_i(x; \theta_i) = \big(z_{i,\text{real}},\ z_{i,\text{synthetic}}\big),$$

where $z_{i,\text{real}}$ is the logit (unnormalized score) that the sample is real, and $z_{i,\text{synthetic}}$ is the logit that the sample is synthetic. (Some implementations may swap the order; what matters is consistent interpretation of each index.)

To convert logits into class probabilities for a given sub-model, we apply the softmax function. For sub-model i, the predicted probability of the sample being real or synthetic is:

$$p_{i,\text{real}} = \frac{\exp(z_{i,\text{real}})}{\exp(z_{i,\text{real}}) + \exp(z_{i,\text{synthetic}})},$$

$$p_{i,\text{synthetic}} = \frac{\exp(z_{i,\text{synthetic}})}{\exp(z_{i,\text{real}}) + \exp(z_{i,\text{synthetic}})},$$

which ensures $p_{i,\text{real}} + p_{i,\text{synthetic}} = 1$. This is a special case of the softmax for $C = 2$ classes (binary classification), where we normalize the two logits to obtain a valid probability distribution. In general, for a classifier with C classes and logits $z = (z_1, z_2, \ldots, z_C)$, the softmax defines

$$p_i = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)}.$$

Here $C = 2$, and it simplifies as above.

Our system combines N such sub-models. We construct a **multi-head ensemble model** $F(x)$ that aggregates the outputs of all sub-models. The ensemble's forward pass collates the logits from each sub-model:

- For each sub-model $i = 1, \ldots, N$, compute $z_i = (z_{i,\text{real}}, z_{i,\text{synthetic}})$.
- Separate the real and synthetic components: collect all $z_{i,\text{real}}$ into a set $\{z_{1,\text{real}}, \ldots, z_{N,\text{real}}\}$ and all synthetic logits $\{z_{1,\text{synthetic}}, \ldots, z_{N,\text{synthetic}}\}$.

- The **ensemble real logit** is defined as the average of all real logits:

$$z_{\text{ensemble, real}} = \frac{1}{N} \sum_{i=1}^{N} z_{i,\text{real}}.$$

- All the synthetic logits are kept separate (each sub-model contributes its own synthetic logit). Thus, the ensemble output is a vector of length $N + 1$:

$$F(x) = \Big(z_{1,\text{synthetic}},\ z_{2,\text{synthetic}},\ \ldots,\ z_{N,\text{synthetic}},\ z_{\text{ensemble, real}} \Big).$$

This output can be interpreted as having N "synthetic" scores (one per head) and one combined "real" score.

To obtain probabilities from the ensemble output, one approach is to apply a softmax over these $N + 1$ values. This would treat the problem as an $(N + 1)$-class classification (each synthetic head being its own class and one real class). However, since conceptually there are only two semantic classes (real vs. synthetic), we typically convert the ensemble output into a binary decision by comparing the largest synthetic score to the real score (see Decision Thresholds below). If needed, one could define an ensemble probability that the sample is real as:

$$p_{\text{ensemble, real}} = \frac{\exp(z_{\text{ensemble, real}})}{\exp(z_{\text{ensemble, real}}) + \sum_{i=1}^{N} \exp(z_{i,\text{synthetic}})},$$

and the total probability of synthetic (aggregated across all heads) as

$$p_{\text{ensemble, synthetic}} = 1 - p_{\text{ensemble, real}}.$$

In practice, the ensemble's **decision rule** (described later) can circumvent explicitly computing this combined probability by using logits directly.

2.2 Loss Function for Training

Each sub-model is trained as a binary classifier using the **cross-entropy loss** function, which is standard for classification tasks. During sub-model training, we treat "Real" as class 0 and "Synthetic" as class 1 (or vice versa, as long as it is consistent). The training data consists of input examples x with ground-truth labels $y \in \{0, 1\}$ (0 = real, 1 = synthetic).

For a single sub-model with output logits $z = (z_{\text{real}}, z_{\text{synthetic}})$ and predicted probabilities $p = (p_{\text{real}}, p_{\text{synthetic}})$ as defined above, and a one-hot encoded ground truth label $y = (y_{\text{real}}, y_{\text{synthetic}})$ (which is $(1, 0)$ for a real sample or $(0, 1)$ for a synthetic sample), the cross-entropy loss is:

$$L = -\Big[y_{\text{real}} \log(p_{\text{real}}) + y_{\text{synthetic}} \log(p_{\text{synthetic}})\Big].$$

This formula is a special case of the general cross-entropy for C classes:

$$L(\theta) = -\sum_{i=1}^{C} y_i \log p_i.$$

Here $C = 2$, and effectively if the sample is real (so $y_{\text{real}} = 1, y_{\text{synthetic}} = 0$), the loss is $-\log(p_{\text{real}})$; if the sample is synthetic, the loss is $-\log(p_{\text{synthetic}})$. Minimizing this loss trains each sub-model to output a high probability for the correct class.

It is important to note that **each sub-model is trained independently** (on the same training dataset) using this loss. We do not train the combined multi-head model in one go; instead, we train N separate models (with potentially different random initializations or other variations). This training uses standard stochastic gradient descent or a variant like the AdamW optimizer. In our implementation, we employ the AdamW optimizer (Adam with weight decay regularization) for training the sub-models. AdamW updates the model parameters θ iteratively using gradients of the loss, which helps with faster convergence. These optimization details ensure the model generalizes well and avoids overfitting through regularization.

After training, we save each sub-model's parameters. The **model merging** stage (described later) will load these parameters into a combined architecture. Notably, when merging, we do **not** further train or fine-tune the ensemble model; the sub-models remain in their trained state. The ensemble simply provides a new forward computation that aggregates their outputs.

2.3 Decision Thresholds and Inference Strategy

At inference time, given a new input sample, the merged multi-head model $F(x)$ produces N synthetic logits (one from each head) and one averaged real logit. How do we decide the final binary classification (real vs. synthetic) from these outputs? We define a decision rule that effectively sets a **threshold** for declaring a sample synthetic. The guiding principle

is that the sample should be classified as **Real only if *all* sub-models agree it looks real**, whereas a strong indication from any one head that the sample is fake should push the decision towards **Synthetic**. This is achieved by comparing the averaged real score to the maximum synthetic score among heads.

Concretely, let:

- $S_i = z_{i,\text{synthetic}}$ be the synthetic logit from sub-model i.
- $R_{\text{avg}} = z_{\text{ensemble, real}} = \frac{1}{N}\sum_{i=1}^{N} z_{i,\text{real}}$ be the averaged real logit.

Our **decision rule** is:

- Predict **"Real"** if R_{avg} is greater than all of the synthetic logits S_i (i.e., if

$$R_{\text{avg}} > \max\{S_1, S_2, \ldots, S_N\}).$$

- Otherwise, predict **"Synthetic"** (the sample is considered fake because at least one head's synthetic score exceeds the real consensus).

This can be interpreted as a threshold comparison. Essentially, we are thresholding on the difference between the highest synthetic confidence and the average real confidence. If even one model gives a logit that suggests a high probability of being synthetic (making that S_i larger than the real average), the ensemble leans towards labeling the sample as synthetic. If all synthetic logits are lower than the real average, it implies all heads are fairly confident the sample is real, so we label it real.

In practice, one could implement this rule by taking the **argmax** of the $N + 1$ outputs of $F(x)$. If the argmax corresponds to the averaged real output (the last dimension), we output "Real"; if it corresponds to any of the synthetic outputs, we output "Synthetic". This argmax rule is effectively equivalent to the threshold rule above, since whichever of $\{S_1, \ldots, S_N, R_{\text{avg}}\}$ is largest will determine the label. (If multiple heads yield high synthetic scores, the argmax will be the largest of them; if all synthetic scores are low and the real score is highest, real wins.) Therefore, the threshold is dynamically determined by the competing values of the ensemble outputs.

It is worth noting that in a standard single-model binary classifier, one would typically use a probability threshold of 0.5 (or logit 0) to decide between real or fake (i.e., label synthetic if $p_{\text{synthetic}} > 0.5$). Here, with multiple outputs, we have adopted a generalized approach: we effectively require p_{real} to be higher than all individual $p_{\text{synthetic}}$ contributions to call

something real. This ensemble decision criterion is chosen to **minimize false negatives**, ensuring that if any model finds the sample suspicious, the system flags it as synthetic.

2.4 Rationale for Averaging the Real Logits (Ensemble Strategy)

A unique aspect of our multi-head design is the asymmetric treatment of the outputs: we **average the real logits across heads** but **do not average the synthetic logits**. Instead, synthetic evidences are kept separate per head. This strategy is deliberate and rooted in the idea of "one-vs-all" consensus for authenticity. The rationale is as follows:

When detecting AI generated data, it is often more critical to catch all instances of synthetic data (high recall) even if it means occasionally raising false alarms. By averaging the real logits, we require consensus among all sub-models for a sample to be deemed real. Each sub-model must output a high real score for the average to remain high. If even one sub-model is unsure (outputting a low real logit), it will drag down the average, making it less likely that the real average surpasses the synthetic scores. On the other hand, we keep the synthetic logits separate so that a single strongly suspicious signal cannot be "washed out" by averaging with other synthetic outputs. In other words, **any one head's strong synthetic detection can override the others**.

From a probability perspective, averaging the real logits is roughly akin to multiplying the independent probabilities that each model agrees the sample is real. If we assume each sub-model's real prediction $p_{i,\text{real}}$ is independent, the probability that all N models agree on real is

$$\prod_{i=1}^{N} p_{i,\text{real}}.$$

Taking the logarithm turns that product into a sum:

$$\log \prod_{i=1}^{N} p_{i,\text{real}} = \sum_{i=1}^{N} \log p_{i,\text{real}}.$$

The log of a probability corresponds (up to a constant) to a logit. Thus, summing logits (or averaging, which is just scaling the sum by $1/N$) for the real class corresponds conceptually to combining evidence from all models in favor of real. In contrast, for synthetic detection, we are interested if *any* model provides evidence for fake, which aligns with keeping the synthetic outputs separate and taking a max.

This design is somewhat analogous to an ensemble where multiple detectors vote on an

outcome, but the voting rule is not simple majority—it is closer to a **unanimity for real** versus **any-one veto leads to fake**. Such a rule skews towards detecting fake content whenever there is doubt. This is useful AI-generated detection because missing a synthetic data (false negative) can be far more damaging than a false positive (flagging a real clip as fake), depending on the application. By averaging the real logits, the system makes it harder to mistakenly classify a fake as real: *all heads would have to be simultaneously fooled*. In contrast, a genuine real sample, being truly benign, will likely appear real to all sub-models, yielding a consistently high real average and low synthetic scores across the board, thus correctly identified as real.

In summary, **averaging the real logits** pools the confidence of all sub-models for authenticity, while treating synthetic detections with an "any sufficient evidence triggers fake" approach. This improves robustness: it reduces sensitivity to any single model's bias for calling things real, and it leverages the ensemble diversity to catch varied forms of synthetic artifacts. This approach is a novel twist on ensemble classification—rather than a straightforward averaging of all probabilities, it asymmetrically combines outputs to bias the classifier toward detection. The effectiveness of ensemble methods in improving classification performance is well-supported in machine learning literature: combining multiple models typically yields lower error rates than individual models. Our strategy specifically targets the operating point that favors detecting AI generated data with high confidence.

3 System Components and Data Pipeline

The multi-head classifier does not exist in isolation; it is the culmination of a data processing and model training pipeline composed of several scripts, each performing a focused task. This modular design follows best practices in machine learning engineering: data is preprocessed and augmented in stages, training and evaluation are clearly separated, and utilities are provided to manage the dataset and file structures. In this section, we explain each script's role in detail and how these components interoperate to form the complete synthetic data detection system. The stages are presented in a logical order (though not all steps are strictly sequential, they typically would be used in this pipeline flow).

3.1 File Renamer (`file_renamer.py`)

The pipeline often begins with a diverse collection of audio files obtained from various sources (some real, some synthetic). These files might have arbitrary names, possibly

containing metadata or inconsistent patterns. The role of **File Renamer** is to standardize filenames in a way that ensures uniqueness and anonymity of the data. This script scans a directory (optionally including subdirectories) for audio files and renames each file to a hash of its contents.

Functionality: It computes a SHA-256 hash of each audio file's binary content and uses a prefix of the hash (e.g., first 16 hex characters) as the new filename. For example, a file originally named `Interview_with_ABC.wav` might be renamed to `1f6999ff03018e9a.wav`. The script retains the original file extension (e.g., .wav, .mp3) but discards the original name. If run in recursive mode, it will traverse subfolders and rename files in each one.

Rationale: By hashing contents, we avoid name collisions (two identical audio files will hash to the same name, which could be seen as a form of deduplication) and remove any identifying information that might be embedded in filenames. This is useful for privacy (filenames might have speaker names, etc.) and for consistency. It also simplifies the next processing steps, because downstream scripts can assume a uniform naming scheme. The hash ensures that if the same audio data appears in different classes or sets, it is easily flagged by having an identical name (this plays a role later in overlap checking).

Interaction: Typically, you would run the file renamer separately on your "Real" data folder and "Synthetic" data folder, or on a combined dataset folder, before any other processing. It prints out a log of old-to-new name mappings so you can trace files if needed. This step is not strictly required, but highly recommended for maintaining a clean dataset.

3.2 Audio Converter (`audio_converter.py`)

Audio data can come in various formats (MP3, WAV, FLAC, AAC, etc.) and different sampling rates or channels. The **Audio Converter** script ensures that all audio files are in a **consistent format** optimal for training the detection model.

Functionality: It uses the FFmpeg tool (invoked via a subprocess call) to convert each input audio file to a standardized format: **WAV format, 32 kHz sampling rate, mono channel, 16-bit PCM**. It scans an input directory for audio files of allowed extensions and outputs all converted files to a specified output directory, preserving the base filename (only extension and format change). The conversion is done in parallel using multiple processes for speed, with a progress bar indicating completion status.

Rationale: A uniform audio format is crucial for reliable feature extraction. The model

(and augmentation routines) expects inputs with the same sample rate and channel count. For instance, mixing stereo and mono or different sample rates without conversion could introduce inconsistencies in the mel-spectrogram extraction. Downsampling to 32 kHz and converting to mono is a common practice to reduce data size while preserving relevant audio frequencies. Converting everything to WAV (a lossless format) ensures we do not accumulate compression artifacts from formats like MP3. Essentially, this script normalizes the data format so that subsequent processing (like augmentation and segmentation) deals with homogeneous audio data.

Interaction: This script would be run on the dataset after renaming. For example, after using `file_renamer.py` on your raw data, you could have all files in `data_raw/real/` and `data_raw/fake/` renamed. Then run `audio_converter.py` on each of those directories (or a combined one) to produce `data_converted/real/` and `data_converted/fake/` with uniform WAV audio. It prepares data for augmentation and feature processing.

3.3 Audio Augmenter (`audio_augment.py`)

Deep learning models benefit from training on diverse examples. **Audio Augmenter** expands the dataset by generating synthetic variations of each audio file through data augmentation techniques. Given an input file, the script produces multiple altered versions, which simulate different realistic distortions or variations in audio, helping the model generalize.

Functionality: For each input audio file, the script creates **10 augmented copies + 1 original copy** (so 11 files total per input). It applies a suite of augmentation techniques, each with random parameters within defined ranges:

- *Time Stretching:* randomly speed up (compression in time) or slow down (expansion in time) the audio by a factor (e.g., between $0.5\times$ and $1.5\times$ speed).
- *Pitch Shifting:* randomly shift the pitch up or down by a few semitones (e.g., ± 2 semitones) without changing speed.
- *Dynamic Range Compression:* randomly apply more or less aggressive compression, altering the audio's amplitude dynamics.
- *Additive White Noise:* inject a small amount of white noise at a random volume (within a limit) to simulate background hiss.
- *Phase Shift:* introduce a phase offset in the audio waveform.

- *Filtering:* apply a random audio filter (e.g., band-pass, high-pass, low-pass, or band-stop) to modify frequency content.
- *Time Domain Shift:* shift the audio slightly in time or apply a subtle time warping (a combination of slight time stretch and pitch shift).
- *No Augmentation (Original):* also include the original file unmodified as one of the outputs for reference.

Each technique is applied such that the output remains plausible sounding. For instance, when adding noise or applying filters, the script keeps levels reasonable to avoid destroying the audio content. The result is that for every original file, a variety of altered versions are created.

Implementation details: The script uses the `librosa` library for many effects (time stretch, pitch shift) and custom `numpy` operations for others (e.g., dynamic compression via amplitude exponentiation, noise addition). It chooses random parameters each time (e.g., a random stretch factor in $[0.5, 1.5]$). Filenames of augmented files are constructed by appending an identifier of the augmentation and its random value to the original name (e.g., if the original is `abcdef1234567890.wav`, an augmented file might be `abcdef1234567890_add_white_noise_0.0025.wav`). This makes it clear what transformation was applied, and the original hash prefix is retained to trace back to the source file group. The script leverages all CPU cores by parallelizing the augmentation of different files, and it logs progress with a TQDM progress bar.

Rationale: Augmentation greatly increases the effective size of the training set and introduces variations that help the model not to overfit to specific dataset conditions. For example, time stretching and pitch shifting help the model be invariant to speaking rate and pitch (which might vary between real and fake data differently). Adding noise or filtering simulates different recording environments. These augmentations are especially important in synthetic data detection because synthetic audio might have telltale artifacts that could disappear or become accentuated under these transformations—by training on augmented data, the model learns robust features that hold under various conditions. As noted in literature, such augmentation strategies can improve generalization to real-world data, and are commonly used in synthetic audio detection challenges.

Interaction: Typically, one would run the audio augmenter on the converted audio files. For instance, take the `data_converted/real/` directory and output to `data_augmented/real/`, and similarly for `data_converted/fake/`. The output is a much larger set of audio clips.

It is common to mix the original and augmented files together for training. The script can also output a CSV log summarizing all augmentations applied, which is useful for record-keeping.

3.4 Audio Segmenter (`audio_segmenter.py`)

Neural networks often require inputs of a fixed size. In audio tasks, this translates to audio clips of a certain duration. The **Audio Segmenter** ensures all audio samples fed to the model have a uniform length (e.g., 4 seconds). If the original or augmented files are longer than this length, they are cut into multiple segments; if shorter, they could be padded.

Functionality: The script takes either a single file or all files in a directory and splits each into consecutive fixed-duration segments (by default, 4 seconds each). It uses FFmpeg to perform the slicing. For example, a 12-second audio will be split into three 4-second WAV files. Each segment file's name is the original filename with a suffix such as `_Segment_001.wav`, `_Segment_002.wav`, etc. The script also ensures the audio is converted to mono 32 kHz during this process. Processing is done concurrently using a thread pool for efficiency, and a progress bar tracks how many files have been processed.

Rationale: Fixed-length segments are needed because our model (in `submodel_trainer.py`) processes inputs as spectrograms of a certain size. If some audio clips were significantly longer, they could be broken into many smaller ones, which effectively increases the data quantity. Importantly, segmentation also prevents very long recordings from biasing the training. By segmenting, each 4-second snippet becomes an independent training (or testing) example. Four seconds is a common choice in audio data detection—it is long enough to capture multiple phonetic units and some prosody, but short enough to assume a roughly stationary behavior and to fit memory and computation constraints.

Interaction: This step is typically applied after augmentation. You would take `data_augmented/real/` and produce `data_segmented/real/`, and similarly for fake. The segmentation script will create the output directories and populate them with segmented files. After this step, you will have a large number of short audio clips, each with a filename that still contains the original file's hash prefix.

3.5 Dataset Manager (`dataset_manager.py`)

At this point in the pipeline, we have a collection of segmented audio files, typically organized by class—for instance, a directory `data_segmented/train_data/class0/` con-

taining all real segments and `.../class1/` containing all synthetic segments. The **Dataset Manager** script automates the split of data into training and testing sets (and possibly validation) while preserving the balance between classes.

Functionality: It takes an input directory that contains subfolders for each class (e.g., `class0`, `class1`). Inside each class folder are many WAV files (the segments). The script then randomly splits the files for each class into a training set and a test set according to a specified ratio (e.g., 80% train, 20% test). It moves files into a new directory structure with `train/` and `test/` subdirectories, each containing class subfolders. For example, after running the dataset manager, you might have:

```
dataset_split/train/class0/*.wav
dataset_split/train/class1/*.wav
dataset_split/test/class0/*.wav
dataset_split/test/class1/*.wav
```

The script ensures that the proportion of files approximately matches the split ratio for each class. It uses either single-thread or multi-threaded file operations and a progress bar to indicate progress.

Rationale: Proper evaluation requires separating training data from testing data to avoid overfitting and to measure generalization. This script ensures that separation is done randomly and reproducibly. By handling each class independently, it maintains class balance. A potential pitfall is that segments from the same original file might end up in both train and test sets; this issue is addressed by the next component.

Interaction: Run this script after you have all your segmented files prepared. It creates a directory structure with clear train and test splits, which will be used by the training script (`submodel_trainer.py`).

3.6 File Manager (Overlap Checker) (`file_manager.py`)

After splitting, it is essential to ensure that no overlap exists between the training and test sets that could lead to data leakage. The **File Manager** script checks for overlapping audio segments that originated from the same source file and, if necessary, moves files from the minority subset to the majority subset.

Functionality: The script examines the filenames of WAV files in corresponding class folders of the train and test sets. It defines a "group key" as the portion of the filename

before the first underscore. Files with the same group key are assumed to come from the same original audio. The script builds dictionaries for the train and test sets and identifies overlapping groups. In **report mode**, it prints a summary of counts; in **fix mode** (using a `-fix` flag), it moves files from the smaller subset to the larger one.

Rationale: Ensuring that no fragments of the same original sample appear in both training and testing is critical for fair evaluation. This script enforces a strict separation to prevent the model from inadvertently learning features specific to an original file present in both splits.

Interaction: After running the dataset manager, run this script on the output directory to clean up any overlaps.

3.7 Submodel Trainer (`submodel_trainer.py`)

This is the core training script for the neural network sub-models (the individual heads) in the ensemble.

Functionality:

- **Data Loading:** It uses a custom `SpectrogramDataset` that reads segmented WAV files and converts each to a mel-spectrogram using torchaudio transforms. The spectrograms are resized (e.g., to 512×512) and normalized, then converted to 3-channel images suitable for a CNN backbone.
- **Model Architecture:** It employs a pre-trained ResNet (e.g., ResNet-18 from `timm`) as the feature extractor, removes the final classification layer, and adds a new fully connected head that outputs 2 logits (for Real and Synthetic).
- **Training Loop:** It trains the network using cross-entropy loss and the AdamW optimizer, possibly with gradient clipping. The script supports multi-GPU training, TensorBoard logging, and periodic evaluation on a validation set.
- **Output:** At the end of training, the model checkpoint (state_dict) is saved for later merging.

Rationale: Training each submodel independently allows the ensemble to benefit from diverse decision boundaries. Stochastic training differences yield models that make errors independently, which can be averaged out in the ensemble. The use of heavy augmentation and fine-tuning of only selected layers ensures robustness and faster convergence.

Interaction: Run this script multiple times (possibly with different random seeds) to obtain several trained sub-model checkpoints.

3.8 Model Merger (`model_merger.py`)

After obtaining several trained sub-models, the **Model Merger** script combines them into a unified multi-head ensemble model.

Functionality:

- It reads sub-model checkpoints from a specified folder and loads each into a `BinaryClassifier` instance.
- It then creates a `ModularMultiHeadClassifier` that stores these sub-models in a PyTorch `ModuleList`.
- In the forward pass, for each input x, the merged model computes each sub-model's output, concatenates all synthetic logits, and averages all real logits to form an output vector of length $N + 1$.
- The merged model, along with metadata (such as class names), is saved to a new checkpoint file.

Rationale: Merging sub-models into a single model simplifies deployment by encapsulating all components in one module. The aggregation of outputs via averaging the real logits provides a robust decision mechanism that leverages ensemble diversity while maintaining a consistent "real" criterion.

Interaction: Run this script once you have all the sub-model checkpoints ready. The output merged model is then used for inference.

3.9 Inference Runner (`inference_runner.py`)

The **Inference Runner** script applies the merged multi-head model to new audio data for synthetic detection.

Functionality:

- It loads the merged model and associated metadata.
- It preprocesses input audio (converting format, segmentation, mel-spectrogram transformation) similarly to the training pipeline.

- For each audio segment, it obtains the model's output vector, applies the decision rule (comparing synthetic logits to the averaged real logit), and determines the predicted label.
- For audio files with multiple segments, it aggregates the per-segment predictions (e.g., via averaging or majority vote) to yield an overall decision.
- The final output is formatted as a JSON structure containing the filename, segmented predictions (with start and end times), and percentage breakdowns of class probabilities.

Rationale: The inference runner abstracts the complete prediction pipeline into a single script that can be easily executed. By replicating the preprocessing used during training, it ensures consistency. The output JSON provides interpretable results for downstream applications or human review.

Interaction: Use this script to run detection on new audio files by specifying the merged model checkpoint and the input file or directory.

4 Literature Review and Comparative Analysis

Our multi-head binary classification approach builds on established principles in machine learning and synthetic audio detection. Traditional binary classifiers or single-model multi-class approaches have achieved good performance in synthetic data detection; however, they often lack the flexibility to incorporate new synthetic classes without retraining the entire model. One-vs-all classification has long been recognized as a robust method for extending binary classifiers to multi-class problems (Rifkin & Klautau, 2004). Our approach leverages a similar idea by training separate binary classifiers (one per synthetic class) and then merging them, thereby combining the advantages of ensemble learning with modularity.

Recent audio detection frameworks have employed ensemble methods to boost accuracy. For example, ensembles of CNNs have demonstrated improved robustness by averaging predictions from diverse models. Our system aligns with these findings, particularly through the unique strategy of averaging the real logits across sub-models while preserving distinct synthetic detections. This asymmetry ensures that a real sample is only classified as real when all sub-models are in agreement, reducing false negatives—a critical factor in security-sensitive applications.

Moreover, the use of extensive audio augmentation (time stretching, pitch shifting, dynamic range compression, etc.) is supported by prior research demonstrating that augmented data leads to better generalization in audio tasks. Our pipeline, which incorporates both offline augmentation and online spectrogram augmentation (e.g., SpecAugment), is designed to mimic real-world conditions and capture the variability present in both synthetic and authentic audio.

Comparatively, while some state-of-the-art methods (e.g., transformer-based models) offer slightly higher performance on specific benchmarks, they often require significantly more computational resources and data. Our approach, using a ResNet-based backbone and a modular ensemble of binary classifiers, strikes a balance between performance, efficiency, and scalability. This makes it particularly suitable for evolving deepfake detection scenarios where new synthetic methods continuously emerge over time.

5 Expected Performance and Efficiency

Based on the design and modular nature of our system, we anticipate the following performance characteristics:

- **High Detection Accuracy:** Each sub-model, trained as a binary classifier, is expected to achieve high accuracy (95–99%) on its respective task. When merged into an ensemble, overall accuracy should reach into the high 90s.
- **Robustness:** Extensive augmentation and the ensemble strategy enhance generalization, reducing both false negatives (missed fakes) and false positives (misclassifying real audio as fake).
- **Scalability:** The modular design allows for easy expansion by training additional sub-models for new synthetic classes without retraining existing ones.
- **Efficiency:** While running multiple models incurs higher computational cost than a single model, the merged model is optimized to run all sub-models in a single forward pass on a GPU. Offline preprocessing (conversion, augmentation, segmentation) is parallelized to handle large datasets efficiently.
- **Real-Time Viability:** With proper batching and GPU acceleration, the system is capable of near-real-time inference on 4-second audio segments, making it suitable for applications requiring timely detection.

6 Conclusion

This white paper presents a detailed overview of a multi-head binary classification system for synthetic data detection, specifically designed for audio deepfake detection. The system integrates a shared feature extraction backbone with multiple binary classifier heads, whose outputs are combined by averaging the real logits. This ensemble strategy enhances robustness and detection accuracy, ensuring that real audio is only classified as such when all sub-models concur. The modular pipeline, comprising scripts for file renaming, audio conversion, augmentation, segmentation, dataset management, sub-model training, model merging, and inference, reflects a rigorous, research-oriented approach to addressing the challenge of synthetic data detection.

Our approach is informed by, and contributes to, current research in ensemble learning, one-vs-all classification, and synthetic data detection. It demonstrates that a modular, extensible framework can not only match but potentially exceed the performance of traditional single-model approaches, especially in dynamic environments where new synthetic methods continuously emerge.

Future work will focus on further optimizing the ensemble (e.g., true weight-sharing among sub-models) and integrating additional modalities (such as video) to build a comprehensive synthetic content detection system.