High Fidelity Prototype

Grupo 1 - Mafalda Dinis e Martinho Figueiredo

This report presents an overview of our implementation of a high fidelity prototype, developed to demonstrate the functionality of our adaptive streaming multimedia system.

System Design

We started by outlining the design, operation, and dependencies of our application, which handles video and audio processing, real-time server-client interaction, and machine learning-based audio inference.

Our application consists of a Flask-based server that integrates three main functionalities: video processing, audio clip processing and analysis, and real-time interaction with a web client. This architecture is designed to handle these tasks efficiently while remaining adaptable to evolving requirements. One of its standout features is the integration of a zero-shot audio classification model, which provides great flexibility in defining and modifying classification categories dynamically.

Our application relies on a combination of Python libraries and external tools to achieve its functionality. Flask and Flask-SocketIO form the foundation of the server, supporting HTTP routes and real-time WebSocket communication. NumPy and SciPy handle numerical and audio file operations, while Torch integrates the MSCLAP model for machine learning-based audio inference. Tomli is used for reading configuration files in TOML format, enabling easy customization of the application.

Externally, the application relies on FFmpeg for media conversion, providing robust support for processing both video and audio files. The Contrastive Language-Audio Pretraining (CLAP) model, a machine learning framework for audio-text similarity, is central to the audio inference process. Its ability to compute embeddings for audio and text and measure their similarity powers the dynamic zero-shot classification feature.

Our application monitors a directory for new MP4 files. When it detects an unprocessed video, FFmpeg converts it into HLS format by generating an .m3u8 playlist file and segmenting the video into smaller .ts files. These processed files are stored in a structured directory, enabling efficient retrieval and streaming.

When an audio clip is received from the client, it is temporarily stored in WebM format before being converted to WAV using FFmpeg. These WAV files are then passed to the CLAP model for analysis.

Additionally, Flask-SocketIO enables real-time communication between the server and client. The server listens for events, such as the submission of audio clips, and emits responses, such as inference results or error messages. This ensures that users receive immediate feedback, enhancing interactivity and usability.

The design of our system focuses on scalability, flexibility, and maintainability. Scalability is achieved through Flask-SocketIO's asynchronous capabilities, enabling the server to handle multiple concurrent client connections efficiently. The use of FFmpeg and CLAP ensures high performance in media processing and inference tasks.

Flexibility is a key advantage of the application, particularly in its audio inference module. The zero-shot classification model empowers users to redefine categories at any time, making it highly adaptable to diverse use cases.

Maintainability is supported through modular design, separating video processing, audio processing, and real-time interaction into distinct components. The configuration-driven approach further simplifies customization, allowing users to adjust server behavior without modifying the codebase.

Sound Classification

Focusing on the sound classification model implemented, we integrated the CLAP model for zero-shot inference. As previously mentioned, CLAP is a machine learning model that computes embeddings for audio and text inputs, enabling the system to measure the similarity between them. This capability forms the backbone of the application's zero-shot classification feature.

Unlike traditional models with preset categories, this zero-shot model allows for categories to be defined dynamically through textual prompts. For instance, a caption such as *"this is the sound of a cough"* can be used to classify audio, and by simply changing the text to *"this is the sound of a sneeze"*, the model can adapt to a new category without retraining. This flexibility allows for rapid adaptation to new use cases or scenarios.

The efficiency of the system is influenced by the number of sound classes and the computational cost of generating embeddings and similarity scores. While CLAP is designed for high performance, certain trade-offs exist. For instance, increasing the

number of sound classes requires the system to compute and compare more embeddings, potentially impacting processing time. This can be mitigated by optimizing configurations, such as limiting the number of classes to be compared.

Despite these considerations, the ability to define and update sound classes dynamically eliminates the need for retraining, significantly reducing overhead compared to traditional classification models.

The flexibility of defining sound classes on the fly introduces minimal system complexity. The CLAP model seamlessly handles new categories by embedding textual prompts at runtime. However, careful management is required to avoid ambiguous or overly broad categories, which could affect classification accuracy. Additionally, users must strike a balance between the granularity of classes and the system's ability to distinguish between similar sounds.

From an implementation perspective, the system's reliance on a robust configuration file simplifies user interaction, making it easier to manage dynamic sound classes without modifying the application code.

Streaming Protocol

The application utilizes HTTP Live Streaming (HLS) as its streaming protocol to deliver video content to clients. HLS, developed by Apple, is designed to adapt to varying network conditions, ensuring uninterrupted video playback. It achieves this by segmenting video files into smaller chunks and delivering them over standard HTTP. These segments are listed in a manifest file, or .m3u8 playlist, which guides the client in sequentially downloading and playing the video.

A distinctive feature of HLS is its support for adaptive bitrate streaming. The server generates multiple versions of the video, each encoded at a different bitrate. This allows the client to dynamically switch between versions depending on available bandwidth, maintaining smooth playback even when network conditions fluctuate. Additionally, because HLS operates over HTTP, it is compatible with existing web infrastructure, including firewalls and content delivery networks (CDNs).

This capability ensures that the application can handle a large number of simultaneous users without compromising performance. The segmentation of video files also enhances fault tolerance; even if a segment is delayed or corrupted during transmission, the client can continue playback by accessing other segments.

In the application, the video processing module relies on FFmpeg to convert MP4 files into HLS format. The resulting .m3u8 playlist and associated .ts segments are

stored in a structured directory, making them easily accessible for client playback. This setup ensures that video content can be delivered efficiently while adapting to the user's network capabilities.

A key strength of HLS lies in its ability to adapt to changing network conditions, which enhances user experience and reduces buffering. While much of this adaptability is handled client-side, the server's implementation plays a crucial role in enabling and optimizing these capabilities.

Higher-bitrate streams are delivered when network bandwidth is sufficient, providing users with a high-resolution experience. Conversely, when bandwidth is limited, the client switches to lower-bitrate streams, ensuring uninterrupted playback. This dynamic adjustment is guided by the .m3u8 playlist, which provides information about the available bitrate levels.

The server also incorporates additional strategies to optimize streaming. The duration of each video segment, configurable in the application, influences the balance between responsiveness and overhead. Shorter segments allow the client to adapt more quickly to changes in network conditions, while longer segments reduce the frequency of playlist updates and improve server efficiency. The application's use of chunked transfer encoding further enhances responsiveness by transmitting data in manageable chunks, which is particularly beneficial in high-latency networks.

Despite its many advantages, the implementation of HLS is not without challenges. The segmentation process introduces a small latency between the server and the client, as each segment must be generated before it can be streamed. Reducing the segment duration could help minimize this latency, but this would also increase the frequency of playlist updates and server load.

Another consideration is the resource cost of generating multiple bitrate versions of a video. This approach, while essential for adaptive streaming, increases storage and bandwidth usage. Employing more efficient video encoding techniques or dynamically generating bitrate versions on demand could alleviate this issue.

The client-side adaptation process also introduces variability in performance, as different media players may implement HLS features differently. Ensuring compatibility and optimizing the behavior across diverse devices is essential to maintaining a consistent user experience. For this we used Video.js, an open-source HTML5 player framework, which is an industry standard and widely adopted.

Adaptation Logic

The system's adaptation logic is designed to be responsive to the user's context, driven by real-time data from the environment and user preferences. Upon accessing the website, the system evaluates whether it has permission to use the device's microphone. This decision dictates whether the system operates in a manual mode or an adaptive mode that utilizes environmental sound classification to adjust streaming parameters automatically. By blending automation with user control, the system ensures flexibility while enhancing usability.

If microphone access is granted, the system relies on environmental sound classification, using CLAP, in order to adapt streaming settings to suit the user's surroundings. In this mode, environmental audio data is analyzed in real time to identify contextual sounds, such as speech, background noise, or silence.

On the other hand, when microphone access is denied, the system falls back to a manual adjustment mode where users can directly control parameters such as audio volume and video quality. This dual-mode design ensures that the system respects user preferences while remaining capable of providing advanced functionality when permitted.

These insights drive automated adjustments to the streaming settings, optimizing the user experience. For instance, if the system detects a noisy environment, it might reduce video quality to conserve bandwidth and minimize playback interruptions.

Conversely, in quiet environments, it can enhance audio clarity and maintain higher video resolutions for an immersive experience. By tailoring its behavior to the user's environment, the system enhances the overall multimedia experience.

If microphone access is denied, however, the system operates in manual mode, allowing users to make adjustments to streaming settings on their own. Users retain control over essential features like audio volume, video quality, and playback. This approach ensures that users can still interact effectively with the system, even without granting full permissions.

The system's adaptation logic significantly enhances the user experience by creating a streaming environment that aligns with individual preferences and contexts. Users in noisy environments benefit from automatic adjustments that reduce disruptions, while those in quieter settings enjoy a high-quality playback experience. By dynamically responding to environmental sounds, the system ensures that users can focus on the content without being distracted by inappropriate playback settings.

Code

Our high fidelity prototype implementation can be found in the following GitHub repository:

CAMM_App Github Repository - https://github.com/martinhofigueiredo/CAMM_app