Trabajo Fin de Grado
en Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo de un sistema de diarización e identificación de hablantes en tiempo real para la integración en un sistema conversacional.

Autor: Pablo Díaz Cotrino

Tutor: Jesús Iván Maza Alcañiz

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado
en Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo de un sistema de diarización e identificación de hablantes en tiempo real para la integración en un sistema conversacional.

Autor:

Pablo Díaz Cotrino

Tutor:

Jesús Iván Maza Alcañiz

Profesor titular

Dpto. Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado: Desarrollo de un sistema de diarización e identificación de hablantes en tiempo real para la integración en un sistema conversacional.

Autor: Pablo Díaz Cotrino

Tutor: Jesús Iván Maza Alcañiz

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2025

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

No puedo decir que esta etapa, la universitaria, haya sido la mejor de mi vida, por todo lo que le debo al colegio donde me formé, maduré y encontré a esos amigos que me acompañarán toda la vida, el Colegio Salesianos de Utrera. Pero, ésta, sin duda, ha sido una etapa muy enriquecedora tanto en lo académico como personal, la cual estoy seguro de que no olvidaré nunca.

Gracias Mamá y Papá por confiar siempre en mí y darme esa estabilidad y esos valores tan necesarios para lograr metas como estas. Gracias Marta, por ponerme siempre los pies en el suelo. Gracias abuela, por asumir mis retos como si fueran tuyos y guiarme siempre con tanto cariño. Gracias abuelo, testimonio de constancia y de lo que significa ser buena persona. Gracias Tita, o debería llamarte también mamá. Sin ustedes, nada de esto tendría sentido.

A mis compañeros, ahora, familia. Los del primer día y, contra todo pronóstico, los del último y de los que quedan. Por demostrarme la importancia de estar bien rodeado para ser mejor cada día. Especialmente, gracias a José por ser tan generoso con tu tiempo y conocimientos. Me gustaría agradecer, también, a mis amigos de "coches rata", por hacer que cada trayecto a la Cartuja fuese especial y un motivo para reír cada día.

Gracias a mi tutor, Iván Maza, por darme la oportunidad de aprender tanto con este proyecto. A la empresa 4i *Intelligent Insights*, especialmente a Lolo, por orientar mi trabajo con tanta paciencia y entusiasmo.

Aquí he aprendido a enfrentarme a lo desconocido sin miedo, y creo que esa es la clave, **vivir sin miedo**.

"Confiad en María Auxiliadora, y veréis lo que son los milagros"

1875. Don Bosco.

Pablo Díaz Cotrino

Alumno de GIERM

Resumen

Este Trabajo de Fin de Grado aborda la implementación de un sistema de diarización e identificación de hablantes en tiempo real, con aplicación directa en plataformas conversacionales. Tras un análisis comparativo de herramientas actuales, se seleccionó la librería DIART por su rendimiento en streaming, y se desarrolló un pipeline completo dividido en tres fases: diarización, segmentación e identificación, primero en modo offline y posteriormente adaptado a procesamiento en vivo desde micrófono. Para la identificación de hablantes conocidos se empleó el modelo ECAPA-TDNN de *SpeechBrain*, integrando mecanismos de verificación robustos con umbrales de confianza y gestión de silencio. Finalmente, el sistema fue integrado en la plataforma DiViVo Lite de la empresa *4i Intelligent Insights*, permitiendo filtrar en tiempo real las intervenciones de un hablante objetivo. El resultado es una solución funcional, precisa y escalable, alineada con los requisitos de procesamiento de audio en entornos de interacción humano-máquina.

Abstract

This Final Degree Project focuses on the implementation of a real-time speaker diarization and identification system, with direct application in conversational platforms. After a comparative analysis of current tools, the Diart library was selected for its streaming performance, and a complete pipeline was developed, consisting of three phases: diarization, segmentation, and speaker identification. Initially implemented in offline mode, the system was later adapted to process live audio from a microphone. For the identification of known speakers, the ECAPA-TDNN model from *SpeechBrain* was used, integrating robust verification mechanisms with confidence thresholds and silence handling. Finally, the system was integrated into the DiViVo Lite platform developed by the company *4i Intelligent Insights*, enabling real-time filtering of a target speaker's interventions. The result is a functional, accurate, and scalable solution, aligned with the requirements of audio processing in human–machine interaction environments.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xv**

Índice de Tablas **xviii**

Índice de Figuras **xx**

1 Introducción y Objetivos. 23
1.1 Planteamiento del problema. 23
1.2 Objetivos. 23
1.3 Alcance y metodología. 23

2 Diarización del Habla: Estado del Arte. 11
2.1 Evolución de las técnicas de diarización. 11
2.1.1 Preprocesado de la señal de audio. 11
2.1.2 Detección de Actividad de Habla (Speech Activity Detection, SAD). 12
2.1.3 Segmentación de locutores. 12
2.1.4 Extracción de embeddings de locutor. 12
2.1.5 Agrupamiento de segmentos (Clustering). 13
2.1.6 Postprocesado. 14
2.2 Parámetros para la evaluación de la diarización del habla. 14
2.2.1 Diarization Error Rate (DER). 14
2.2.2 Jaccard Error Rate (JER). 15
2.2.3 Exactitud por hablante (precisión y *recall*). 15
2.2.4 Tiempo de inferencia y latencia. 15
2.3 Herramientas de Diarización. 16
2.3.1 *PyAnnote*: Características y rendimiento. 16
2.3.2 DIART: Diarización en tiempo real y evaluación del desempeño. 17
2.3.3 Comparativa técnica Pyannote vs DIART. 19
2.3.4 Conclusión y justificación del uso de DIART. 20
2.4 Herramienta de Identificación de Hablantes. 20

3 Pipeline de tratamiento de audio. 22
3.1 Fase 1: Diarización de Audio. 22
3.2 Fase 2: Segmentación de Audio por Hablante. 24
3.3 Fase 3: Identificación de Hablantes. 27
3.4 Integración Secuencial del Pipeline (main.py). 29

4 Evolución del Pipeline en Tiempo Real. 31
4.1 Versión 1: Integración básica del Pipeline. 31
4.2 Versión 2: Umbral de decisión para hablantes conocidos vs. desconocidos. 32
4.3 Versión 3: Buffer de audio y fusión de intervenciones continuas. 32
4.4 Versión 4: Optimización de rendimiento y mejoras de ingeniería. 34
4.5 Versión 5: Agrupación por silencio y procesamiento al final de un turno. 38

4.6 Versión Final: Versión robusta con temporización automática y ajustes finos. *41*

5 Implementación del Sistema de Diarización e Identificación en DiViVo. **46**

5.1 Captura de Audio y Preprocesamiento (módulo speech_microphone.py). *46*

5.2 Clase CDiarization: Gestión del Pipeline de Diarización (módulo speech_diarization.py). *48*

5.2.1 process_chunk(chunk: np.ndarray) -> bool. 48

5.2.2 send_result(). 48

5.2.3 extract_intervention_chunks() -> List[np.ndarray]. 49

5.2.4 reset_buffer(). 49

5.3 Fuente de Audio por Fragmentos (módulo audiosourcechunk.py). *50*

5.4 Pipeline de Diarización en Streaming (módulo pipeline.py). *52*

5.5 Identificación del Hablante Objetivo (módulo realtime_speaker_id.py). *53*

5.5.1 Carga del modelo de identificación de hablante. 53

5.5.2 Verificación de identidad del hablante (método _flush()). 54

5.5.3 Integración y rendimiento en tiempo real. 56

5.6 Flujo Completo: Conexión de todos los componentes. *57*

6 Conclusiones y Líneas de Desarrollo Futuras. **59**

6.1 Trabajo Futuro. *59*

6.2 Conclusión. *59*

7 Referencias **61**

ÍNDICE DE TABLAS

Tabla 2.1 - Resultados de evaluación – PyAnnote.audio 17
Tabla 2.2 - Resultados de evaluación – DIART Framework 18
Tabla 2.3 - Comparativa resumida de PyAnnote vs. DIART en distintos aspectos técnicos y operativos. 20
Tabla 3.1 - Tabla comparativa de las versiones del script de diarización 24

ÍNDICE DE FIGURAS

Ilustración 2.1 - Sistema de diarización tradicional. 11

Ilustración 4.1 - Representación temporal de subsegmentos solapados (ventanas deslizantes de 1 s con solapamiento del 50%) 35

Ilustración 4.2 - Lógica de decisión en la agrupación de segmentos de un mismo hablante. 38

Ilustración 5.1- Flujo Completo: Conexión de los componentes. 58

1 INTRODUCCIÓN Y OBJETIVOS.

En los últimos años, los sistemas de reconocimiento y procesamiento automático de voz han avanzado de manera notable a nivel tecnológico siendo cada vez más presente en la sociedad. Aplicaciones como asistentes virtuales, transcriptores automáticos o sistemas de subtitulado en tiempo real requieren no solo reconocer el contenido de lo que se dice, sino también distinguir quien y cuando lo dice.

Esta capacidad es esencial en entornos multimedia con múltiples hablantes (reuniones, conferencias, debates), donde la transcripción y las tareas de análisis posterior dependen de una correcta separación de voces.

1.1 Planteamiento del problema.

Este proyecto se ha realizado en colaboración con la empresa *4i Intelligent Insights* (1), una empresa innovadora de tecnología aplicada cuya misión es lograr, a través de una interacción natural, sólida y privada, que las máquinas comprendan mejor a los humanos en beneficios de sus usuarios.

Se plantea este proyecto como una oportunidad para implementar la funcionalidad de diarización, segmentación e identificación de hablantes en la propia tecnología desarrollada por la empresa basada un sistema de diálogo hombre-máquina de última generación, multimodal, multilingüe, multiplataforma y de ejecución local, que permite mantener interacciones privadas, robustas y fluidas con robots y computadoras.

Concretamente, se hará uso de su NLP Engine (motor de Procesamiento de Lenguaje Natural) con capacidad de diálogo, visión y voz llamado DiViVo.

1.2 Objetivos.

Se comenzará analizando y comparando dos herramientas extendidas en el panorama de diarización actual: PyAnnote y Diart. Evaluando su rendimiento sobre un conjunto de datos anotado previamente, midiendo métricas estándar de diarización.

Una vez determinada la herramienta de diarización a utilizar se implementará un prototipo completo en Python que permita procesar audio. En este caso, el audio será previamente grabado y será procesado de modo offline. Este enfoque permite aprovechar al máximo los recursos disponibles, ya que el pipeline completo de diarización (lectura de audio, inferencia de segmentos de hablante y escritura de resultados), segmentación de hablantes y final identificación de *"known speakers"* (hablantes conocidos previamente fichados) opera sobre un fichero estático, eliminando las restricciones de latencia propias del procesamiento en tiempo real. Esto permite, también, comprobar el funcionamiento las tecnologías de una forma más simple y controlada.

La siguiente etapa del proyecto consiste en poner en marcha el pipeline completo en tiempo real, integrando tanto la diarización como la identificación de hablante sobre la señal que llega directamente del micrófono.

Finalmente, se integrarán estas funcionalidades en DiViVo, el cual requiere que el tratamiento del audio sea en streaming.

1.3 Alcance y metodología.

Para cumplir estos objetivos, el trabajo se basa en la revisión bibliográfica y técnica con el estudio de publicaciones y documentación oficial de las herramientas, así como de métricas y protocolos de evaluación de diarización.

Se crea un entorno en Python con el uso de Anaconda para la instalación de dependencias y configuración de scripts de prueba.

Para la implementación y pruebas se llevará a cabo el uso de librerías que permitan la captura y procesamiento de audio, ejecución de pipelines de diarización, escritura de ficheros RTTM y generación de gráficas de

resultados.

Se realiza una evaluación comparando los tiempos de ejecución y consumo de memoria de la solución propuesta. El análisis de los resultados es primordial para la identificación de los puntos fuertes y débiles de cada herramienta usada a lo largo del proyecto y propuestas de optimiza

2 DIARIZACIÓN DEL HABLA: ESTADO DEL ARTE.

Los métodos de diarización del habla usados han ido evolucionando a lo largo de la historia de la mano de los avances tecnológicos y desarrollo de herramientas matemáticas que han ido aconteciendo. Es por ello por lo que se intenta plasmar en este capítulo el estado pasado, actual y posible futuro de esta poderosa y cada vez más presente herramienta de procesamiento de audio.

2.1 Evolución de las técnicas de diarización.

La diarización de hablantes (*speaker diarization*) es el proceso de segmentar una señal de audio con varios hablantes para determinar *quién habló y cuando* (2). En otras palabras, dado un audio multihablante, el sistema identifica los intervalos de voz de cada locutor sin conocimiento previo de sus identidades.

Tradicionalmente, este problema se ha abordado en una serie de etapas bien definidas. En el pipeline clásico se incluye el preprocesado de la señal, la detección de actividad de habla, la segmentación por locutor, extracción de características o *embeddings* del locutor, el agrupamiento (*clustering*) de segmentos y, por último, un post procesamiento de refinamiento. Se puede apreciar en la Ilustración 2.1 el sistema tradicional.

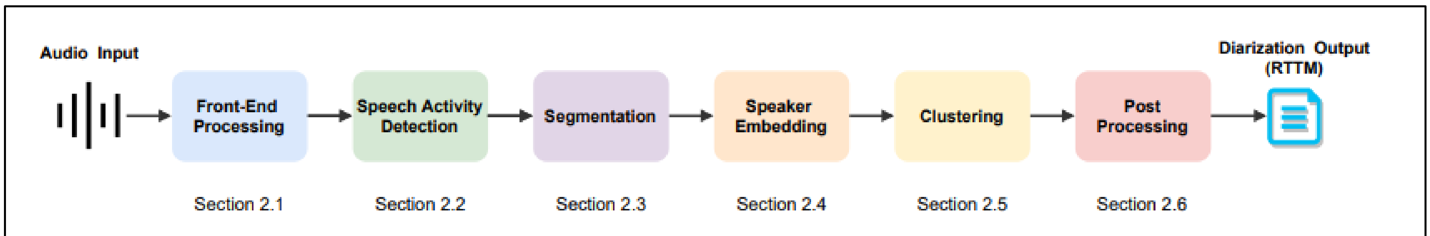

Ilustración 2.1 - Sistema de diarización tradicional.

A lo largo de los años, las técnicas empleadas en cada etapa han ido evolucionando, pasando de enfoques estadísticos convencionales a métodos modernos basados en ***Deep Learning***, lo que ha mejorado sustancialmente el desempeño de los sistemas. A continuación, se describe la función de cada etapa y cómo han ido evolucionando sus técnicas.

2.1.1 Preprocesado de la señal de audio.

Esta etapa tiene como objetivo mejorar la calidad y representación de la señal antes de intentar distinguir a los hablantes.

Se aplican técnicas de filtrado de ruido (3), normalización de volumen y eliminación de ecos o silencios prolongados. Una vez que se ha limpiado el audio, se extraen las características acústicas que representan el habla.

Típicamente se han usado los **coeficientes cepstrales de frecuencia Mel** (MFCC) como descriptor principal de la señal de voz. Los MFCC ofrecen una representación compacta del espectro de la voz humana enfatizando la información relevante para distinguir los locutores.

En las últimas décadas, el front-end ha incorporado también técnicas de aumento de datos y filtros encargados de robustecer la señal (*Speech Enhancement*) de cara a etapas posteriores (4).

2.1.2 Detección de Actividad de Habla (Speech Activity Detection, SAD).

Es fundamental saber dónde hay voz en la grabación antes de intentar separar quién habla. El módulo SAD (*Speech Activity Detection*) se encarga de distinguir entre los segmentos que contienen habla de lo que no (silencios, ruido de fondo, etc). De manera que las siguientes etapas se concentren solo en esas regiones.

En sistemas clásicos, se suele basar este procesado en umbrales de energía o en modelos estadísticos sencillos: por ejemplo, se calcula la energía a corto plazo de la señal y se determina que hay voz cuando ésta supera cierto umbral ajustado al ruido del fondo.

Con la llegada de técnicas modernas, se han desarrollado detectores de voz más robustos basados en redes neuronales profundas (DNN, *Deep Neural Network*) entrenadas para ser capaces de diferenciar el habla del ruido.

Puede ocurrir que el SAD genere un número significativo de eventos salientes falsos positivos o pasar por alto segmentos de habla. Por ello, se suele tener en cuenta el parámetro Diarization Error Rate (DER), explicado en la siguiente sección.

2.1.3 Segmentación de locutores.

Aquí se detectan los puntos de cambio de hablantes dentro de las regiones de voz previamente identificadas. Es decir, el sistema se encarga de dividir el audio continuo en segmentos más pequeños, cada uno de forma ideal manteniendo sólo a un mismo hablante.

Uno de los primeros métodos ampliamente utilizados fue el criterio de información bayesiano (BIC, *Bayesian Information Criterion*) (5). Este algoritmo, introducido a finales de los 90, evalúa mediante criterio estadístico si dos ventanas adyacentes de audio pertenecen al mismo hablante o no. Básicamente, lo que hace es comparar la probabilidad de modelar ambos segmentos con un mismo modelo gaussiano frente a modelarlos con dos modelos distintos, penalizando la complejidad para evitar la sobre segmentación. Cuando la diferencia supera cierto umbral, se decide que ha habido cambio de hablante. Este enfoque permite realizar segmentación de forma no supervisada, siendo un estándar durante años. Además, se propusieron variantes como el Delta-BIC adaptativo, que ajustaba el umbral dinámicamente, logrando detecciones aún más estables.

A pesar de que la técnica BIC funcionase razonablemente bien, presentaba ciertas limitaciones: su desempeño dependía del umbral y tenía un costo computacional elevado al evaluar múltiples particiones. Por ello, se investigaron otros métodos alternativos (6) (7) basados en distancias entre distribuciones gaussianas (GLR, *Generalized Likelihood Ratio*) y algoritmos de cambio abrupto (*Change-Point Detection*), así como enfoques mediante modelos de mezcla Gaussiana (GMM, *Gaussian Mixture Model*) entrenados para distintas ventanas deslizantes.

En la última década, con el auge del *Deep Learning*, han surgido técnicas que usan redes neuronales, integrando información contextual más amplia que los métodos puramente estadísticos. No obstante, muchos sistemas actuales combinan criterios estadísticos simples como BIC con modelos más complejos que incorporan aprendizaje automático para una detección de cambios más fiable.

2.1.4 Extracción de embeddings de locutor.

En esta etapa, las señales de voz ya segmentadas se transforman en vectores de características o *embeddings* que capturan las propiedades únicas de cada hablante. Estos vectores facilitarán la comparación y agrupamiento de segmentos según su identidad vocal.

Inicialmente, se utilizaban directamente vectores de características derivados de promediar las características acústicas (por ejemplo, promediar los MFCC del segmento). Sin embargo, la introducción del enfoque *i-vector* (8) supuso un gran avance. El método de *i-vectors*, propuesto a finales de la década de los 2000, proporciona una representación compacta de cada segmento en un espacio de dimensión fija mediante técnicas de

factorización de subespacios. Esto se logra entrenando un modelo generativo a partir de un conjunto amplio de datos de muchos hablantes: el modelo aprende una distribución general y luego representa cualquier segmento como un desplazamiento de esa distribución media en un subespacio de variabilidad total. Básicamente:

1. Se parte de un modelo que ha "escuchado" a muchas personas diferentes. A eso se le llama modelo de fondo universal o UBM (*Universal Background Model*). Es como tener una idea general de cómo suenan las voces humanas.
2. Cuando se quiere representar la voz de una persona concreta (por ejemplo, un fragmento de audio), comparamos su voz con ese modelo general.
3. A partir de esa comparación, se genera un vector corto (el *i-vector*) que resume cómo se diferencia esa voz del promedio general.

Este vector tiene siempre la misma longitud (por ejemplo, 100 o 400 valores) sin importar cuánto dure el audio. Y lo mejor es que esos valores capturan la información más importante para distinguir una voz de otra.

Una vez obtenido los *i-vectors* de todos los segmentos, es posible aplicar modelos como PLDA (*Probabilistic Linear Discriminant Analysis*) (9) para medir de forma probabilística la similitud entre los segmentos. Esta combinación permite mejorar considerablemente la capacidad de agrupar correctamente los segmentos del mismo hablante, superando los métodos basados solo en distancias en el espacio acústico original.

La siguiente gran evolución vino de la mano del *Deep Learning*. Uno de los primeros enfoques fue el denominado d-vector, donde una red neuronal profunda (DNN) se entrena en una tarea de clasificación de hablantes a gran escala. Se demostró que las redes podían aprender características más ricas que los modelos generativos tradicionales, especialmente cuando había disponibles grandes cantidades de datos de entrenamiento.

Siguiendo esta línea, en 2018 se introdujo el *x-vector* (10), ahora ampliamente utilizado. Son *embedding* extraídos mediante una arquitectura neuronal especial llamada *Time-Delay Neural Network* (TDNN) (11) entrenada con cientos de miles de segmentos de voz de muchos hablantes. Esta red aprende a mapear un segmento variable en el tiempo a un vector fijo de dimensión reducida, lo que optimiza la clasificación de hablantes.

Estudios comparativos, han mostrado que los *x-vectors* superan a los *i-vectors* en tareas tanto de identificación como de diarización de locutores, especialmente en condiciones de hablantes desconocidos para el sistema, sentando las bases de los sistemas actuales.

2.1.5 Agrupamiento de segmentos (Clustering).

Con cada segmento representado mediante un *embedding* apropiado, el siguiente paso del sistema es agrupar los segmentos según las voces, de forma que los segmentos en un mismo grupo correspondan al mismo hablante.

El método más extendido tradicionalmente ha sido el *clustering* jerárquico aglomerativo (AHC, *Agglomerative Hierarchical Clustering*). En este enfoque, cada segmento comienza su propio grupo de forma individual y se agrupan entre sí según una medida de similitud. La similitud entre segmentos o *clusters* se define en términos de distancias entre sus representaciones: en los primeros sistemas se utilizaba distancia entre centroides MFCC o puntuaciones BIC entre modelos GMM, mientras que en sistemas con *i-vectors* o *x-vectors* es común usar distancia coseno o probabilidades PLDA.

Por ejemplo, en un esquema moderno, se calcula la probabilidad de que dos *embeddings* pertenezcan al mismo hablante bajo un modelo PLDA; si esa probabilidad es alta (por encima de cierto umbral), los segmentos se fusionan en el mismo clúster. El proceso continúa hasta que no quedan pares con afinidad suficiente o hasta alcanzar un número objetivo de hablantes (siempre y cuando esa cifra sea conocida de antemano).

Es una herramienta simple y eficaz, aunque puede verse limitada por su sensibilidad al umbral de parada y su ineficiencia en contextos a gran escala.

Una evolución relevante fue la introducción del *clustering* con restricción supervisada (*Constrained AHC*), que incorpora conocimiento previo, por ejemplo, forzando a que ciertos segmentos no pertenezcan al mismo grupo. Este tipo de enfoque ha mostrado mejoras significativas en tareas como la diarización de reuniones donde ciertas etiquetas parciales están disponibles.

Más recientemente, se ha adoptado el uso de algoritmos basados en grafos y propagación de afinidades, como *Spectral Clustering*, que construyen un grafo de similitud entre *embeddings* y agrupan a partir de sus componentes espectrales. Este enfoque ha ganado popularidad gracias a su robustez frente a la variabilidad de duración y condiciones acústicas, especialmente cuando se utiliza con *embeddings* como *x-vectors.*

En paralelo, han surgido enfoques basados en *End-to-End Neural Diarization* (EEND), en los que el modelo aprende directamente la asignación de etiquetas de hablante en un marco temporal continuo. Estos sistemas requieren grandes volúmenes de datos etiquetados y no están muy extendidos en escenarios productivos, aunque son bastante prometedores.

2.1.6 Postprocesado.

Las técnicas de postprocesado tienen como objetivo mejorar la coherencia y calidad del resultado final, corrigiendo errores típicos de fases anteriores. El postprocesado tradicional incluye tareas como la fusión de segmentos muy cortos (que pueden ser a su vez fragmentos erróneamente separados), el ajuste fino de los límites de segmentación y la filtración de solapes o inconsistencias.

Es muy típico la aplicación de un collar temporal, una tolerancia que se introduce en los puntos de cambio de hablante durante la evaluación para compensar pequeñas imprecisiones.

La re-segmentación es el proceso de redefinir las fronteras entre hablantes. Se usa en sistemas modernos como los basados en *embeddings* y *clustering* jerárquico. Consiste en la aplicación de un segundo paso de asignación de hablantes más preciso utilizando modelos como PLDA. Sirve para refinar las etiquetas asignadas y mejorar la precisión.

Como otra dirección de post procesado, ha habido una serie de estudios en fusionar múltiples resultados de diarización para mejorar la precisión. Aunque generalmente, la combinación de resultados genera mejores resultados para varios sistemas (por ejemplo, *speech recognition* o *speaker recognition*), este método para la diarización puede proporcionar algunos problemas como el etiquetado de hablantes no está estandarizado entre los diferentes sistemas de diarización, el número estimado de hablantes y las fronteras estimadas de tiempo podrían diferir entre los sistemas.

Cabe destacar que algunos frameworks como *PyAnnote* y DIART permiten configurar directamente estos parámetros de postprocesamiento (collar, duración mínima de segmentos, número máximo de hablantes, etc.), lo que representa una gran ventaja para adaptarlos a diferentes contextos (reuniones, entrevistas, conversaciones abiertas).

2.2 Parámetros para la evaluación de la diarización del habla.

De alguna forma se necesita evaluar la calidad de los sistemas de diarización del habla, para ello se utilizan métricas estandarizadas que permitan medir con precisión las capacidades.

2.2.1 Diarization Error Rate (DER).

La exactitud de un sistema de diarización de hablantes está medida por el DER. Es la suma de los tres tipos de errores diferentes: Falsa Alarma (FA) de actividad vocal, omisiones (Miss) que son fragmentos donde hay voz en la referencia, pero el sistema no la detecta y confusión de hablante (SpE), aquellos fragmentos en los que se detecta actividad vocal, pero se asigna el hablante incorrecto.

Matemáticamente, se expresa como:

$$\boldsymbol{DER} = (FA + Miss + SpE) \,/\, Total\ Speech\ Time$$

El cálculo del DER suele incorporar un collar temporal dentro del cual no se penalizan las desviaciones temporales para evitar penalizaciones excesivas por errores menores en los límites de los turnos de palabra. Puede ser mucho mayor que el 100%.

2.2.2 Jaccard Error Rate (JER).

El objetivo es evaluar cada hablante con el mismo peso. Al contrario que, con el DER, el cual es estimado para toda la intervención. Se calculan primero las tasas de error por hablante y luego se promedian.

Específicamente, el JER se calcula de la siguiente manera:

$$\mathbf{JER} = \frac{1}{N} \sum_{i}^{N_{ref}} \frac{\mathrm{FA_i} + \mathrm{MISS_i}}{\mathrm{TOTAL_i}}.$$

En la ecuación, $\mathrm{TOTAL_i}$ es la unión del tiempo de habla del i-ésimo hablante en la transcripción de refencia y el tiempo de habla de i-ésimo hablante en las hipótesis. N_{ref} es el número de hablantes en el guion de referencia. El JER nunca excede el 100% ya que utiliza la operación de unión entre la referencia y la hipótesis.

Ambas métricas están altamente correlacionadas, pero si un subconjunto de hablantes es dominante, el JER tiende a ser mayor en el caso ordinario.

2.2.3 Exactitud por hablante (precisión y *recall*).

En algunos casos, como en este proyecto, es relevante saber qué porcentaje de los segmentos asignados a un hablante coinciden correctamente con su identidad real. Para ello, se usa una matriz de confusión o métricas como la precisión o *recall*. La precisión mide la proporción de segmentos que fueron etiquetados correctamente como pertenecientes a un hablante específico entre todos los segmentos que fueron clasificados como pertenecientes a ese hablante. En cambio, el *recall* se enfoca en cuántos de los segmentos reales de un hablante han sido correctamente identificados por el sistema.

Si hay alta precisión y *recall* bajo, significa que el sistema es muy estricto, lo que podría provocar que se perdieran muchos segmentos relevantes. Al contrario, significa que se identifican la mayoría de los segmentos del hablante, pero también se podrán estar asignando erróneamente segmentos a ese hablante.

2.2.4 Tiempo de inferencia y latencia.

Estas métricas son especialmente útiles en aplicaciones de tiempo real. El tiempo de inferencia es el tiempo total que tarda el sistema en procesar un segundo de audio y generar la predicción o salida. La latencia es el retraso temporal que ocurre entre la entrada, considerándose la captura del audio en tiempo real y la salida.

Estas métricas indican si un sistema puede ejecutarse de forma interactiva o si requiere procesamiento en *batch*. La latencia final depende de la arquitectura del sistema, del método de segmentación y de los tipos de *embeddings* empleados.

2.3 Herramientas de Diarización.

En este apartado se presentarán las herramientas más relevantes para la diarización del habla, con especial atención a dos *frameworks*: *Pyannote* y DIART. Se tiene como objetivo describir la arquitectura y funcionamiento de cada herramienta, así como sus capacidades de personalización e integración. Se comparará su rendimiento según métricas estándar (tasa de error de diarización, *Diarization Error Rate* o DER, y latencia). Asimismo, se discuten los casos de uso ideales de cada solución (tanto procesamiento *offline* como en tiempo real) y se revisan experimentos publicados que evalúan su desempeño.

Finalmente, se realiza una comparativa técnica directa y se justifica la elección de una de ellas para la elaboración de este proyecto, atendiendo a requerimientos de procesamiento en tiempo real, flexibilidad de configuración y facilidad de integración con otros módulos.

2.3.1 *PyAnnote*: Características y rendimiento.

PyAnnote (o *pyannote.audio*) es un *toolkit* de diarización desarrollado en Python por *H. Bredin* y colaboradores, reconocido por proporcionar un pipeline de última generación listo para usar (12). Su diseño modular sigue la arquitectura clásica de segmentación-embeddings-clustering, incorporando avances recientes para manejar trozos de habla solapados. En concreto, la versión 2.x de *PyAnnote* implementa tres etapas principales:

1. **Segmentación de hablantes** mediante un modelo neuronal *end-to-end* que opera sobre ventanas cortas deslizantes. Opera típicamente sobre ventanas de unos 5 segundos con solapamiento (paso 0.5 s) para limitar la dificultad del problema en cada fragmento y mejorar la detección de cambios de hablantes.
2. **Extracción de *embeddings*** para cada locutor local identificado.
3. **Clustering aglomerativo** global encargado de agrupar esos embeddings asignando una identidad de hablante consistente a lo largo de toda la conversación (12). Se agrupan usando clustering jerárquico (por ejemplo, aglomerativo con criterio de distancia) para producir las etiquetas finales de los hablantes.

Una ventaja es que viene con modelos pre-entrenados y configuraciones predeterminadas que logran buenos resultados. Por ejemplo, incluye modelos para la detección de voz y para embeddings de locutor basados en arquitecturas profundas ya tratadas en secciones anteriores (*x-vectors* o similares). Aunque el pipeline no requiere el ajuste manual de parámetros, ofrece opciones de personalización: fijar número de hablantes esperados (`num_speakers`) o un rango mínimo/máximo (13).

PyAnnote proporciona guías para adaptar el pipeline a datos propios mediante *fine-tuning*: procedimientos para afinar los modelos con un conjunto de entrenamiento específico y así mejorar el rendimiento en ese dominio.

No obstante, la integración en tiempo real de *Pyannote* no es trivial: la herramienta está concebida principalmente para procesar archivos de audio completos de forma offline. Aunque usa ventanas locales, el procesamiento típico está pensado para hacerse una vez leído todo el audio y la salida una vez finalizada la diarización completa. Implementar un funcionamiento en *streaming* requeriría un desarrollo adicional significativo (por ejemplo, se podría alimentar manualmente con fragmentos de audio el pipeline), algo que queda fuera del soporte estándar de la biblioteca del sistema.

En cuanto a la exportación e integración de resultados, este *toolkit* maneja formatos estándar (como RTTM), y sus objetos de anotación pueden transformarse fácilmente en Python. A diferencia con DIART, no ofrece mecanismos para enganchar observadores o módulos personalizados durante la inferencia, así que cualquier funcionalidad extra debería ser aplicada en una etapa posterior.

Rendimiento

PyAnnote ha demostrado conseguir una precisión muy alta en diarización sobre múltiples conjuntos de evaluación. En su configuración por defecto (sin *fine-tuning* específico por dataset), alcanza tasas DER competitivas. Se puede apreciar en la Tabla 2.1 los distintos resultados en varios *benchmarks* diferentes.

Benchmark	DER%	FA%	Miss%	Conf%	Expected output
AISHELL-4	14.09	5.17	3.27	5.65	RTTM
Albayzin (*RTVE 2022*)	25.60	5.58	6.84	13.18	RTTM
AliMeeting (*channel 1*)	27.42	4.84	14.00	8.58	RTTM
AMI (*headset mix, only_words*)	18.91	4.48	9.51	4.91	RTTM
AMI (*array1, channel 1, only_words)*	27.12	4.11	17.78	5.23	RTTM
CALLHOME (*part2*)	32.37	6.30	13.72	12.35	RTTM
DIHARD 3 (*Full*)	26.94	10.50	8.41	8.03	RTTM
Ego4D *v1 (validation)*	63.99	3.91	44.42	15.67	RTTM
REPERE (*phase 2*)	8.17	2.23	2.49	3.45	RTTM
This American Life	20.82	2.03	11.89	6.90	RTTM
VoxConverse (*v0.3*)	11.24	4.42	2.88	3.94	RTTM

Tabla 2.1 - Resultados de evaluación – PyAnnote.audio

Presenta aproximadamente 11.2% DER en el *benchmark VoxConverse* (conversaciones extraídas de vídeos) y tan sólo 8.2% en datos de dominio controlado como REPERE. En condiciones más desafiantes con muchos hablantes o ruido el error puede aumentar (por ejemplo, DER de unos 25-30% en reuniones AliMeeting, DIHARD III, etc), pero aun así el pipeline se mantiene cercano al estado del arte sin ajustes adicionales (13). Estas evaluaciones publicadas confirman la solidez de la herramienta en términos de exactitud. Por otro lado, resulta ser eficiente en cuanto velocidad de procesamiento se refiere: aprovechando la aceleración por GPU, el pipeline puede procesar una hora de audio en unos 1.5 minutos, lo que equivale a un factor tiempo real de ~0.025 (es decir, ~40 veces más rápido que la duración real del audio). Esto implica que para uso offline (procesamiento diferido de grabaciones) *PyAnnote* cumple sobradamente con los requisitos de rendimiento. Sin embargo, es importante notar que esta rapidez no se traduce en latencia cero para aplicaciones interactivas: aunque el cómputo sea rápido, *PyAnnote* típicamente espera a tener el audio completo y solo entonces entrega la diarización, por lo que no está optimizado para producir salida con baja latencia durante una sesión en vivo.

En síntesis, *PyAnnote* brinda excelente precisión y funcionalidades avanzadas para diarización offline, pero presenta desafíos cuando se requiere baja latencia y adaptación dinámica, que es precisamente el nicho donde entra en juego la siguiente herramienta, DIART.

2.3.2 DIART: Diarización en tiempo real y evaluación del desempeño.

DIART es un *framework* Python más reciente enfocado desde su creación en la diarización en tiempo real. Fue desarrollado por J.M. Coria, H. Bredin (2021-2024). DIART adopta la arquitectura de *PyAnnote* y la adapta para streaming, reutilizando muchos de sus componentes internos, pero añadiendo una lógica incremental de procesamiento.

La arquitectura combina un enfoque híbrido *end-to-end* con modular: por un lado, emplea un modelo neuronal entrenado de forma *end-to-end* para segmentación de hablantes en ventanas locales (como se ha visto en *PyAnnote*), lo que le permite detectar hablantes simultáneos de manera automática; y, por otro lado, usa un algoritmo de clustering incremental para ir uniendo esas etiquetas locales en identificadores de hablante globales persistentes a lo largo del tiempo.

Concretamente, DIART procesa el audio en fragmentos continuos mediante un buffer rodante de unos 5 segundos. Cada intervalo se analiza (cada 250 ms) y el modelo de segmentación produce etiquetas locales de hablantes por frame. Esas etiquetas y sus embeddings se pasan al módulo de clustering incremental, el cual asigna si se considera alguno de esos segmentos a hablantes globales ya existentes, o crea uno si no se encuentra correspondencia (esto se hace comparando la distancia del embedding local con el de cada hablante global conocido). Así se van incorporando nuevos hablantes conforme van interviniendo en la conversación. Además, gracias al modelo *end-to-end* se puede activar múltiples hablantes en un mismo frame cuando corresponda.

Se reutiliza el modelo de segmentación de hablantes de *PyAnnote* y el modelo de embeddings de locutor, por lo que se ha estudiado ya la calidad base de estos sistemas.

La diferencia radica en la orquestación del pipeline en tiempo real, por ello se emplea programación reactiva (basada en *ReactiveX*, una API para programación asíncrona con observadores (14)) para manejar el flujo de audio entrante y la producción de eventos de diarización (15).

Gracias a la API proporcionada, se puede cambiar componentes con facilidad, es posible configurar Diart para utilizar un modelo de embeddings distinto (como ECAPA o *WeSpeaker*, si se desea experimentar con ellos) simplemente cambiando la referencia del modelo y también ajustar automáticamente los hiper parámetros del sistema.

En cuanto integración, está diseñado pensando en la práctica, permite conectar directamente múltiples fuentes de audio (archivos locales, streams remotos, micrófonos en vivo, incluso *WebSockets*), abstrayendo la complejidad de leer y fragmentar el audio en tiempo real. Además, permite adjuntar observadores personalizados al pipeline, por lo que se puede registrar funciones u objetos que reciban en vivo los resultados parciales o finales de la diarización a medida que se van generando. Esto nos permitirá más adelante en el proyecto, integrar módulos adicionales como el módulo de identificación de hablantes, el cual tras cada actualización de diarización asigna IDs a los clusters de voz detectados.

DIART ya provee observadores como `RTTMWritter` (para guardar la salida en formato RTTM) o `StreamingInference` que gestiona la iteración continua.

Rendimiento.

Al igual que *Pyannote*, DIART alcanza altos niveles de precisión. Ha sido validada en entornos competitivos: por ejemplo, participó en el *Ego4D Audio-only Diarization Challenge 2022* superando con amplio margen al sistema baseline offline del desafío. Si bien DIART emplea los mismos modelos acústicos que *PyAnnote* (por defecto). Se puede observar en Tabla 2.2 el rendimiento de DIART en el estudio de referencia (16) :

rank	segmentation model	embedding model	DER	latency mean in s	latency std in s
1	pyannote/segmentation	pyannote/embedding	44.86	0.057065	0.003522
2	pyannote/segmentation-3.0	pyannote/embedding	48.79	0.060674	0.002804
3	pyannote/segmentation @Interspeech2021	pyannote/embedding	50.06	0.064204	0.007641
4	pyannote/segmentation-3.0	pyannote/wespeaker-voxceleb-resnet34-LM	48.66	0.217110	0.016487
5	pyannote/segmentation	pyannote/wespeaker-voxceleb-resnet34-LM	45.42	0.218060	0.010972
6	pyannote/segmentation @Interspeech2021	pyannote/wespeaker-voxceleb-resnet34-LM	49.89	0.284455	0.011451
7	pyannote/segmentation-3.0	speechbrain/spkrec-ecapa-voxceleb	49.16	0.461488	0.082039

Tabla 2.2 - Resultados de evaluación – DIART Framework

Podría decirse que las ventajas son claras en cuanto a latencia, ya que, DIART está optimizado para minimizar el retraso entre entrada y salida: los resultados se generan en pocos cientos de miles milisegundos. Se aprecia como la configuración de DIART con modelos de *PyAnnote* obtuvo la latencia más baja entre varios sistemas online, con un tiempo medio de alrededor de 0.006 segundos para producir la etiqueta del hablante tras recibir el audio. Es notable que incluso sistemas *end-to-end* puros como FS-EEND logran latencias similares en pruebas controladas, pero DIART ofrece esa rapidez sin requerir conocer por adelantado el número de hablantes y con la capacidad de manejar streams potencialmente infinitos de manera estable (16).

Desde una perspectiva crítica, las principales ventajas son la baja latencia y la flexibilidad. Permitiendo ajustar fácilmente parámetros o sustituir componentes (p. ej. probar distintos umbrales de similitud en el clustering, o intercambiar el modelo de embeddings por otro más ligero si se busca reducir cómputo), lo que es valioso para ajustar el rendimiento según las necesidades de la aplicación. También se hereda de *PyAnnote* el beneficio de modelos pre-entrenados sólidos, evitando tener que entrenar un sistema completo desde cero. Entre las limitaciones de DIART estaría que, al ser relativamente nuevo, no está tan ampliamente validado en tantos dominios diferentes como *PyAnnote*; la mayoría de sus evaluaciones se han centrado en escenarios de reuniones y conversaciones cortas.

Por último, DIART, al igual que *PyAnnote*, se beneficia del uso de hardware acelerado para obtener su máximo rendimiento: si bien puede correr en CPU, para mantener latencias de decenas de ms con audio continuo ha sido útil usar una GPU para la inferencia de los modelos neuronales.

2.3.3 Comparativa técnica Pyannote vs DIART.

Se ha elaborado la Tabla 2.3 que resume la comparativa directa entre *PyAnnote* y DIART, destacando sus diferencias en clave de arquitectura, rendimiento y aplicabilidad.

Características	PyAnnote.audio (Pipeline 2.x)	Diart (Diarización en tiempo real)
Arquitectura básica	Pipeline de tres etapas separadas: segmentación local + embeddings + clustering global (HAC aglomerativo).	
Modelos subyacentes	Modelos propios de *PyAnnote* (*PyTorch*)	Reutiliza modelos pre-entrenados de PyAnnote (segmentación, embeddings) por defecto; permite intercambiar modelos (e.g. ECAPA, WeSpeaker) según necesidades.
Detección de solapamiento	Soporta solapamiento a través del modelo de segmentación end-toend con resegmentación tras clustering.	Soporta solapamiento de forma nativa en cada ventana local.
Operación temporal	Offline/batch: procesa el audio completo y luego output final. No proporciona salida parcial durante la ejecución.	Online/streaming: procesa audio en tiempo real con actualización continua cada ~0.25 s; entrega segmentos y etiquetas de hablante casi inmediatamente durante la grabación.
Latencia de diarización	No optimizada para latencia (depende de duración del audio). Computación rápida: ~1.5 min por hora de audio (RTF ≈ 2.5% en GPU), pero requiere esperar a final de la conversación para el resultado	Baja latencia (diseño real-time): ~0.06 s de retraso medio desde audio entrante hasta etiqueta. Ventana local de 5 s implica ligera demora inicial, pero tras ello la diarización se actualiza prácticamente en vivo.

Precisión (DER)	Estado del arte offline: DER ~10– 20% en escenarios típicos sin ajuste; mejoras adicionales con fine-tuning específico.	Competitivo online: DER cercano a pipeline offline equivalente (usa mismos modelos).
Personalización	Parámetros ajustables (p. ej. número de hablantes conocido). Recetas para re-entrenar/ *fine-tuning* en datos propios. Menos orientado a modificación en vivo; pensado para refinar modelos fuera de línea.	Altamente personalizable en tiempo real: API OOP extensible, permite modificar componentes del pipeline fácilmente. Incluye optimización automática de hiperparámetros (Optuna) para adaptar el sistema a nuevos datasets sin re-entrenar desde cero.
Integración en otros sistemas	Enfoque investigador: típicamente se usa como herramienta autónoma (entrada: archivo, salida: RTTM). Integración en flujo streaming requiere desarrollos adicionales.	Enfoque aplicativo: diseñado para incrustarse en aplicaciones. Acepta audio de micrófono, streams web, etc. . Permite callbacks (observadores) durante la inferencia, facilitando integración con módulos externos (ej. transcripción, identificación de hablantes).
Caso de uso recomendado	Diarización offline de grabaciones conocidas (reuniones, vídeos, datasets) donde se prioriza la máxima exactitud y se puede procesar tras la captación.	Diarización en tiempo real en aplicaciones en vivo (videoconferencias, streaming de medios, asistentes virtuales) donde se requiere respuesta inmediata y adaptación dinámica a la conversación.

Tabla 2.3 - Comparativa resumida de PyAnnote vs. DIART en distintos aspectos técnicos y operativos.

2.3.4 Conclusión y justificación del uso de DIART.

Tras el análisis detallado, puede concluirse que *PyAnnote* y DIART son herramientas complementarias, cada una sobresaliendo en contextos distintos. PyAnnote representa una solución robusta y altamente precisa para diarización de audio offline, apropiada para investigaciones y escenarios batch donde la latencia no es crítica. DIART, por su parte, está específicamente diseñada para diarización con baja latencia, integrándose fácilmente en sistemas en tiempo real sin sacrificar demasiado rendimiento.

DIART ha sido seleccionada como la base del pipeline de diarización desarrollado debido a que ofrece la mejor correspondencia con las necesidades del proyecto: permite obtener una diarización de alta calidad mientras el audio ocurre, con mínima latencia, y su arquitectura abierta encaja perfectamente para incorporar los módulos adicionales (observadores personalizados, identificación de hablantes, escritura de resultados) de forma coherente y eficiente. Aunque *PyAnnote* proporciona un rendimiento muy alto en entornos offline, DIART provee las herramientas y la estructura requeridas para construir un sistema de diarización en streaming plenamente funcional, justificando técnicamente su elección para este proyecto.

2.4 Herramienta de Identificación de Hablantes.

Para la identificación de hablantes, se utiliza el modelo preentrenado ***spkrec-ecapa-voxceleb*** proporcionado por *Speechbrain*, el cual es cargado a través de la función `SpeakerRecognition.from_hparams()`.

Este modelo se basa en la arquitectura ECAPA-TDNN (*Emphasized Channel Attention, Propagation and Aggregation - Time Delay Neural Network*), una red neuronal profunda que combina tantos bloques convolucionales y residuales junto con mecanismos de atención estadística (*attentive statistical pooling*) para

extraer *embeddings* robustos de la voz de cada hablante. Esta arquitectura es una de la más empleadas por la elevada robustez que presenta en tareas de verificación de locutor al mejorar los modelo *x-vectors* tradicionales (17).

En particular, el modelo *spkrec-ecapa-voxceleb* que ha sido entrenado en el conjunto *VoxCeleb* presenta muy buena precisión (aproximadamente un 0,8% de tasa de error EER en prueba sobre *VoxCeleb1* (18)), lo que evidencia su capacidad para distinguir distintas identidades.

Finalmente, este modelo fue elegido por ofrecer equilibrio entre exactitud en la identificación y eficiencia en la inferencia, a la vez que facilita la integración al sistema gracias a tener una API sencilla de *hardware params*, evitando el hecho de tener que entrenar el modelo de cero.

3 PIPELINE DE TRATAMIENTO DE AUDIO.

El sistema completo se ha desarrollado en tres fases secuenciales: diarización, segmentación e identificación de hablantes. La fase de diarización es la que se encarga de procesar el audio de entrada para detectar segmentos temporales continuos de voz y asignarles etiquetas de hablante anónimas (por ejemplo, *Speaker0*, *Speaker1*, etc.). A continuación, en la fase de segmentación, se utiliza el resultado de la diarización para extraer los fragmentos de audio correspondientes a cada hablante detectado. Finalmente, en la fase de identificación, cada segmento de audio se compara contra una base de datos de hablantes conocidos para asignar nombres reales (u obtener "desconocido" si no coincide con ninguno).

Inicialmente, este pipeline se implementó como un proceso offline secuencial: se ejecutaban por separado un script de diarización, luego uno de segmentación y finalmente otro de identificación, intercambiando información mediante archivos intermedios (por ejemplo, un archivo de anotaciones RTTM y archivos de audio segmentados). Con el tiempo, los scripts de cada fase fueron evolucionando para ser más robustos (mejor manejo de errores, uso eficiente de recursos, umbrales para decisiones, etc.). Además, se desarrolló una versión integrada en tiempo real que combina las tres fases en un solo pipeline que procesa audio en vivo. En las siguientes secciones se detalla la estructura y evolución de cada fase, incluyendo fragmentos de código representativos y las mejoras clave introducidas, para finalmente destacar cómo se migró de un pipeline offline a uno en *streaming* en tiempo real.

3.1 Fase 1: Diarización de Audio.

En la fase de diarización, el objetivo es analizar un flujo de audio (ya sea desde un archivo WAV o un micrófono) y producir segmentos temporales etiquetados que indican cuándo habla cada persona. Para ello se emplea la librería DIART, que provee un pipeline de diarización en *streaming*. La implementación básica, que está en el script *diarize.py*, consiste en instanciar el pipeline de diarización y ejecutarlo sobre un archivo de audio, guardando los resultados en formato RTTM y visualizando la línea de tiempo de los hablantes.

Versión inicial: El script comienza configurando la fuente de audio de la entrada desde un archivo WAV y la frecuencia de muestreo. Luego, crea la instancia del pipeline de diarización DIART (`SpeakerDiarization`) y la envuelve en un proceso de inferencia en streaming (`StreamingInference`) junto con la fuente de audio de archivo (`FileAudioSource`). A este proceso se le adjunta un observador `RTTMWriter` para volcar las anotaciones de diarización en un archivo RTTM de salida. Finalmente se lanza la inferencia y se obtiene una predicción completa (`prediction = inference()`), que contiene los segmentos de diarización. El script entonces recorre estos segmentos para dibujar una gráfica temporal de la diarización, donde cada hablante detectado aparece en una línea horizontal distinta.

Al terminar, guarda la figura en un archivo PNG y notifica si el archivo RTTM fue generado correctamente. Este primer enfoque cumple su cometido, pero carece de manejo de excepciones en la inferencia y asume que todo saldrá bien.

```
# Inicializar el pipeline de diarización
pipeline = SpeakerDiarization()
audio_source = FileAudioSource(audio_path, sample_rate)

# Ejecutar la inferencia en streaming y guardar el RTTM
```

```
inference = StreamingInference(pipeline, audio_source, do_plot=False)
inference.attach_observers(RTTMWriter(audio_source.uri, output_rttm))
prediction = inference() # <- Ejecuta la diarización sobre el archivo de audio
```

La segunda versión incorpora varias mejoras para hacerlo más robusto. En primer lugar, se reorganiza la gestión de rutas de entrada y salida utilizando *os.makedirs* con *exist_ok=True*, para garantizar los directorios necesarios, como por ejemplo la carpeta de output para RTTM y figuras para la gráfica existan evitando, así, fallos si no estaban creados. Además, se encapsula la ejecución de la inferencia en un bloque *try/except* para capturar excepciones durante el procesamiento de audio. Si ocurre algún error interno en la diarización (por ejemplo, un modelo no cargado o formato de audio incorrecto), el script ahora lo reporta con un mensaje claro en lugar de colapsar silenciosamente. Abajo vemos la sección relevante donde se inicia la inferencia con manejo de errores:

```
#  Iniciar la inferencia con observadores
inference = StreamingInference(pipeline, audio_source, do_plot=False)
inference.attach_observers(RTTMWriter(audio_source.uri, output_rttm))

#  Ejecutar la diarización
try:
    prediction = inference()
except Exception as e:
    print(f" Error durante la diarización: {e}")
    prediction = None
```

Las mejoras incluyen mensajes más informativos en la consola para indicar éxito o fallo a la hora de guardar la figura PNG o el archivo RTTM. En cuanto a la funcionalidad, diarize2.py sigue utilizando la inferencia en streaming de DIART de la misma forma que la versión inicial, pero evitando parámetros no soportados. Se mantiene la generación de la gráfica, aunque con un ligero ajuste del tamaño de la figura para mayor claridad.

Paralelamente, se desarrolló un pequeño *script*, denominado *diarize_micro.py*, para probar la diarización directamente desde el micrófono en tiempo real. Se demuestra lo sencillo que es cambiar de fuente de audio, en lugar de `FileAudioSource`, usa `MicrophoneAudioSource` de DIART. El pipeline se construye de igual forma (`SpeakerDiarization()` sin parámetros) y se ejecuta con `StreamingInference`. En este caso, `do_plot = True` para visualizar en vivo la diarización mientras se va capturando el audio. Este script no realiza manejo de errores sofisticado ni guardado de figuras/ RTTM (más allá de adjuntar un `RTTMWriter` a un archivo fijo output/file.rttm), ya que su propósito es principalmente de prueba interactiva. No obstante, sirvió como paso previo para integrar la diarización en el pipeline de streaming completo.

Cabe destacar que *diarize_micro.py* imprime información del micrófono (`print(mic)`) para confirmar la selección de fuente de audio antes de iniciar la inferencia. Algo útil en entornos con mútiples dispositivos de audio.

Comparativa de versiones de diarización: En la siguiente Tabla 3.1 se resumen las diferencias clave entre las implementaciones de diarización mencionadas.

Característica	diarize.py (v1)	diarize2.py (v2)	diarize_micro.py (micro)
Fuente de audio	Archivo WAV (`FileAudioSource`)	Archivo WAV (misma fuente)	Micrófono del sistema (`MicrophoneAudioSource`)
Creación de directorios	Manual (usa `os.path.exists` + `os.makedirs`)	Automática con os.makedirs(...,`exist_ok=True`)	No aplica (usa rutas por defecto “output”)
Manejo de errores	Ninguno (asume ejecución exitosa)	Envoltorio *try/except* alrededor de inferencia	Ninguno (script de prueba simple)
Salida RTTM	Sí (archivo *diarization.rttm*)	Sí (mismo nombre de archivo RTTM)	Sí (escribe en output/ file.rttm)
Visualización gráfica	Sí (guarda y muestra plot de segmentos)	Sí (ajuste de tamaño, mensajes mejorados)	En vivo (`do_plot=True` muestra gráfico dinámico)
Comentarios/ Logs	Mensajes básicos (confirmación de guardado)	Mensajes con símbolos (/) y detalles de error	Imprime info del micrófono; sin logs avanzados

Tabla 3.1 - Tabla comparativa de las versiones del script de diarización

Esta evolución muestra una transición desde un script mínimo funcional a uno más sólido para uso automatizado, y a una variante para captura en directo.

3.2 Fase 2: Segmentación de Audio por Hablante.

Tras obtener la diarización, el siguiente paso es extraer los segmentos de audio correspondientes a cada hablante.

En un pipeline offline tradicional, se cuenta con el archivo RTTM generado en la fase anterior, que contiene líneas indicando para cada segmento: el tiempo de inicio, la duración y el identificador de hablante asignado. La tarea del script de segmentación es leer ese archivo RTTM y usarlo para cortar el archivo de audio original en trozos etiquetados por hablante.

Analizaremos primero la versión final mejorada (*segment_audio2.py*), ya que es la que se utilizó en la ejecución completa. Este script asume por defecto rutas específicas: la ubicación del archivo RTTM (diarization.rttm en la carpeta output), la ruta del audio original procesado, y define una carpeta de salida para segmentos (output/segments). Lo primero que hace es asegurarse de que la carpeta de segmentos exista, y eliminar cualquier archivo previo en ella para no mezclar resultados de ejecuciones anteriores. Se usa `os.makedirs(…, exist_ok=True)` y luego recorre `os.listdir(output_dir)` eliminado archivo a archivo, garantizando un entorno limpio.

```
# Eliminar todos los archivos de la carpeta de segmentos antes de guardar los
nuevos
for file in os.listdir(output_dir):
    file_path = os.path.join(output_dir, file)
    if os.path.isfile(file_path):
        os.remove(file_path)
```

Después, se abre el archivo RTTM y lo recorre línea a línea. Realiza un Split por espacios para extraer los campos relevantes: se asume, siguiendo el formato estándar RTTM que la posición 4 es el inicio del segmento (en segundos), la posición 5 es la duración y la posición 8 es el nombre del hablante asignado. Con esos datos calcula el final del segmento (inicio + duración). Seguidamente, agrega ese intervalo a un diccionario `segments` agrupado por hablante. Es decir, construye una estructura en memoria donde cada clave es un ID de hablante (*Speaker0*, *Speaker1*, etc.) y su valor es la lista de todos los intervalos `(start, end)` en que ese hablante habla. Ejemplo de archivo RTTM generado:

```
SPEAKER muestra3 1 0.862 1.587 <NA> <NA> speaker1 <NA> <NA>
SPEAKER muestra3 1 2.637 2.554 <NA> <NA> speaker0 <NA> <NA>
SPEAKER muestra3 1 5.408 0.100 <NA> <NA> speaker0 <NA> <NA>
SPEAKER muestra3 1 5.508 2.917 <NA> <NA> speaker1 <NA> <NA>
SPEAKER muestra3 1 8.608 2.400 <NA> <NA> speaker0 <NA> <NA>
```

Como mejora importante, esta agrupación permite luego combinar todos los fragmentos de un mismo hablante. En una posible implementación inicial más simple, se podría haber optado por cortar y guardar cada fragmento por separado. Sin embargo, *segment_audio2.py* decide concatenar todos los fragmentos de cada hablante en un solo audio, simplificando así, la identificación posterior y facilita escuchar de corrido todos lo dicho por una persona.

Una vez construido el diccionario de segmentos, se carga en memoria el audio completo (`librosa.load`) una única vez. Esto es eficiente comparado con leer el archivo desde disco varias veces. Luego se recorre el diccionario por hablante y por cada lista de intervalos se extraen los *subarrays* correspondientes. Hay que evitar índices de muestra inválidos (por ejemplo, si un cálculo excede la longitud del array de audio, o si por error el índice inicial resulta no ser menor).

Acto seguido, todos los fragmentos validados de un hablante se concatenan usando `np.concatenate`. El resultado es un array de audio único por cada hablante. Este se escribe en un archivo .wav usando `soundfile.write` en la carpeta de segmentos, nombrando el archivo con el identificador del hablante (por ejemplo, *Speaker0.wav*). Si ocurriera algún error durante la escritura del archivo WAV, se captura con una excepción y se notifica por pantalla. Así que, al finalizar, se habrán creado tantos archivos de audio como hablantes se hayan detectado en el RTTM, con todo lo hablado por cada uno.

A continuación, se muestra un extracto representativo de *segment_audio2.py* donde se realiza la extracción y guardado de los segmentos por hablantes.

```
# Leer RTTM y extraer segmentos (asumiendo formato RTTM estándar)
segments = {}
with open(rttm_file, "r") as f:
    for line in f.readlines():
        parts = line.strip().split()
        # Se asume que: parts[3] = inicio, parts[4] = duración, parts[7] =
nombre del hablante
        start, duration, speaker = float(parts[3]), float(parts[4]), parts[7]
        end = start + duration

        if speaker not in segments:
            segments[speaker] = []

        segments[speaker].append((start, end))
```

```
# Cargar el archivo de audio una sola vez
y, sr = librosa.load(audio_file, sr=16000)

# Crear y guardar un único segmento por hablante
for speaker, speaker_segments in segments.items():
    # Concatenar todos los fragmentos del mismo hablante en un solo segmento
    speaker_audio = []
    for start, end in speaker_segments:
        start_sample, end_sample = int(start * sr), int(end * sr)
        if start_sample >= end_sample or start_sample < 0 or end_sample >
len(y):
            print(f"Advertencia: Segmento con índices de muestra inválidos para
{speaker}. Omitiendo...")
            continue
        speaker_audio.append(y[start_sample:end_sample])
    # Unir todos los fragmentos del hablante en un solo array
    speaker_audio = np.concatenate(speaker_audio)

    # Guardar el segmento en un archivo único por hablante
    output_path = os.path.join(output_dir, f"{speaker}.wav")
    try:
        sf.write(output_path, speaker_audio, sr)
        print(f"Segmento para {speaker} guardado: {output_path}")
    except Exception as e:
        print(f"Error al guardar el segmento para {speaker}: {e}")
```

Aunque el análisis se ha centrado en la versión final, conviene mencionar qué problemas o limitaciones se encontraron en versiones anteriores para entender la evolución:

- En versiones anteriores no se concatenaban los segmentos por hablante, sino que se escribía un archivo por cada segmento. Lo que daba lugar a decenas de pequeños archivos WAV dificultando el manejo posterior.
- No se eliminaban archivos antiguos de la carpeta de salida, lo que causaba confusión entre resultados de diferentes ejecuciones.
- No se validaban los índices de corte. Por ejemplo, si el RTTM tenía algún tiempo mal calculado, se corría el riesgo de un error al intentar cortar el array de audio fuera de sus límites. La versión final agrega la comprobación explícita para evitarlo.
- En un enfoque inicial, se consideró usar Pandas para leer o estructurar el RTTM, pero finalmente, se optó por usar un método más directo y claro (leer líneas y hacer *split* manualmente).

En resumen, la evolución de este script se centró en robustez y eficiencia. Garantizar entorno limpio, evitar reprocesamientos innecesarias (lectura única de audio), manejar posibles inconsistencias en datos, y facilitar el siguiente paso uniendo segmentos por hablantes. Esto preparó el terreno para que la identificación de hablantes operara sobre un conjunto claro y reducido de archivos (uno por cada hablante detectado).

3.3 Fase 3: Identificación de Hablantes.

En la fase final, los segmentos de audio (ya separados por hablante anónimo) se comparan contra muestras de referencias de hablantes conocidos para intentar ponerle nombre real a cada voz. Aquí se usa la herramienta **SpeechBrain** con un modelo pre-entrenado de reconocimiento de locutor (en concreto, ECAPA VoxCeleb). El proceso general en las distintas versiones d ellos scripts de identificación es:

1. **Cargar el modelo de Speaker Recognition** de SpeechBrain (se realiza una vez al inicio dado que es pesado).
2. **Definir la base de datos de hablantes conocidos**: un diccionario con nombres y rutas de archivos WAV de referencia por cada persona que queremos reconocer. Por ejemplo, se tienen muestras de voz de "Pablo", "Marta", etc.
3. **Recorrer los segmentos de audio** generados en la fase anterior (carpeta output/segments) y, para cada archivo .wav, verificar su similitud con cada muestra conocida mediante `spk_model.verify_files(segment, ref_audio)`. Esta función devuelve típicamente un score numérico de similitud y, en algunos casos, una predicción booleana sobre si son la misma persona.
4. **Decidir la identidad** del segmento según los scores obtenidos, lo que se hace aplicando algún criterio: por ejemplo, elegir mayor score por encima de cierto umbral, o si no supera el umbral declarar "Desconocido".

En primer lugar, se realizó una versión básica, *identify_speaker.py*, la cual iteraba por todos los archivos .wav en output/ segments, y para cada uno calculaba el score contra cada hablante conocido. Lo más sencillo era que eligiese el hablante con mayor score y lo asignase, sin lógica alguna de umbral ni consideración de empate de puntuaciones. Además, no utilizaba la clase `Path` de Python ni se gestionaban errores, enfocándose así únicamente en la funcionalidad básica.

En la segunda versión, *identify_speaker2.py*, se ven protecciones adicionales. Para comenzar, se usa `pathlib.Path` para manejar la ruta de la carpeta de segmentos de forma más robusta (independiente de separadores de carpeta). Seguidamente, el script recorre los archivos de WAV usando un método propio de `Path (segment_dir.glob("*.wav"))`, lo cual es más portátil. Antes de procesar cada archivo, imprime un mensaje indicándolo y verifica que realmente existe. Luego, inicializa variables `best_match = "Desconocido"` y `best_score = -1` como punto de partida para determinar el mejor candidato.

El bucle interno evalúa cada segmento contra los audios de referencia conocidos. Se llama `spk_model.verify_files(segment_str, ref_audio)` dentro de un *try/except*. Aquí la gestión de errores mejora: si por alguna razón la verificación falla (ej. archivo corrupto, error de E/S), se captura la excepción y se continúa con el siguiente candidato en lugar de abortar todo. Si la llamada tiene éxito, devuelve un score (y un *prediction* que indica si el modelo considera misma persona). Se emplea la lógica siguiente: si el score es mayor que el mejor encontrado hasta ahora y además *prediction* es *True*, entonces actualiza `best_match` con el nombre del hablante y guarda ese `best_score`. Es decir, requiere ambas condiciones: buena puntuación y que el modelo efectivamente haya predicho que es la misma persona. Si ningún hablante produce `prediction = True`, el `best_match` se queda como "Desconocido". Tras comparar con todos, asigna ese mejor hablante al segmento actual y lo imprime.

Este enfoque asegura que no se elija un hablante solo por tener un score marginalmente mayor si el modelo no lo corrobora. Sin embargo, un posible inconveniente es que depende completamente de la señal booleana *prediction* de `verify_files`, la cual internamente podría estar usando un umbral fijo no ajustado a nuestro caso de uso. Además, no maneja la situación de empates o scores cercanos.

La tercera versión (*identify_speaker3.py*), introduce criterios adicionales y mejoras de salida. En este script, en lugar de usar la lógica de mejor score con `prediction`, se adopta un enfoque de comparación de todos los scores de forma más explícita.

- Para cada segmento, se almacena en un diccionario denominado scores la puntuación con cada hablante conocido. Todas las verificaciones se hacen dentro de *try/except* individuales para ignorar fallos como antes.
- Luego se ordena este diccionario de scores de mayor a menor.
- Se toma el mejor y segundo mejor score.
- Umbral de diferencia: aquí está la mejora: se define un `threshold` que representa cuanta diferencia debe haber entre el primero y segundo para confiar en el resultado. Si la diferencia es menor que ese umbral, significa que dos hablantes tienen puntajes muy parejos y, por tanto, el sistema decide no arriesgarse a asignar uno incorrectamente. En tal caso, clasifica el segmento como "Desconocido", aunque haya un ganador ligero. Esto reduce falsos positivos cuando el modelo no es concluyente.
- Todos los resultados se guardan en un diccionario `speaker_mapping` que mapea el nombre del archivo del segmento con el nombre asignado.
- Por último, se imprime un resumen final de las asignaciones listando cada archivo de segmento con su identidad determinada. Esto es útil para volcar los resultados de manera organizada al final de la ejecución.

He aquí una porción de *identify_speaker3.py* ilustrando la lógica descrita:

```
# Umbral de diferencia para considerar las puntuaciones como "muy similares"
threshold = 0.2  # Ajustar este valor según se desee para la diferencia mínima

# Iterar sobre los archivos .wav en el directorio de segmentos
for segment_path in segment_dir.glob("*.wav"):
    # Convertir la ruta a un string con formato POSIX (barras diagonales), lo
que evita problemas de escape
    segment_str = segment_path.resolve().as_posix()
    print(f"Procesando: {segment_str}")
    if not segment_path.exists():
        print(f"El archivo {segment_str} no existe.")
        continue

    # Almacenar las puntuaciones para comparar
    scores = {}

    # Comparar el segmento con los hablantes conocidos
    for name, ref_audio in known_speakers.items():
        try:
            score, prediction = spk_model.verify_files(segment_str, ref_audio)
        except Exception as e:
            print(f"Error al verificar {segment_str} contra {ref_audio}: {e}")
            continue

        scores[name] = score

    # Ordenar las puntuaciones en orden descendente
    sorted_scores = sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

```
    # Si las dos mejores puntuaciones son muy cercanas, lo clasificamos como
'Desconocido'
    if len(sorted_scores) > 1:
        best_match, best_score = sorted_scores[0]
        second_best_match, second_best_score = sorted_scores[1]

        # Verificar si la diferencia entre las dos puntuaciones más altas es
menor que el umbral
        if best_score - second_best_score < threshold:
            best_match = "Desconocido"
        speaker_mapping[segment_path.name] = best_match
    else:
        # Si solo hay una puntuación, asignamos el mejor hablante
        best_match, best_score = sorted_scores[0]
        speaker_mapping[segment_path.name] = best_match

    print(f" {segment_path.name} identificado como:
{speaker_mapping[segment_path.name]}")
```

En este código vemos cómo se llenan todos los scores y luego se ordenan. La condición `if best_score - second_best_score < threshold` aplica el criterio de cercanía de puntuaciones para asignar "Desconocido" si la diferencia es mínima. Finalmente, después del bucle principal, el script imprime un resumen con todas las asignaciones almacenadas en `speaker_mapping`.

3.4 Integración Secuencial del Pipeline (*main.py*).

Una vez desarrolladas las tres fases por separado con sus respectivos scripts, se creó un main para ejecutarlas en orden automáticamente: *main.py*. Este script no realiza procesamiento de señal por sí mismo, sino que lanza los otros programas en la secuencia correcta, asegurando el que el flujo de datos sea adecuado.

En su contenido, primero define la ruta del intérprete de Python a usar (por ejemplo, el ejecutable dentro del propio entorno de DIART) y el directorio donde se encuentran los scripts. Luego, simplemente utiliza llamadas al sistema operativo (`os.system`) para invocar:

1. El script de diarización: *diarize.py*.
2. El script de segmentación: en este caso llama directamente a *segment_audio2.py* (versión mejorada).
3. El script de identificación: *identify_speaker3.py* (versión final).

Cada llamada se hace pasando el ejecutable Python y la ruta del script. Dado que cada uno es un proceso por separado, *main.py* espera a que termine una etapa antes de iniciar la siguiente. Así se puede garantizar que el archivo RTTM esté listo antes de segmentar y que los archivos estén preparados antes de la identificación.

Un extracto de *main.py* muestra esta secuencia claramente:

```
# 1 Ejecutar diarización
os.system(f"{python_exec} {script_dir}\\diarize.py")

# 2 Extraer segmentos de audio
os.system(f"{python_exec} {script_dir}\\segment_audio2.py")

# 3 Identificar hablantes
os.system(f"{python_exec} {script_dir}\\identify_speaker3.py")
```

Cabe añadir que este enfoque modular facilitó pruebas y depuración, ya que cada fase puede desarrollarse por separado e incluso ejecutarse individualmente. Por otro lado, una limitación de esta integración es que depende del almacenamiento intermedio en disco: genera un archivo RTTM y múltiples WAVs temporales. Si bien esto no supuso un problema mayor en offline, abre la puerta a mejorar hacia un pipeline más directo en memoria, como veremos en la siguiente sección.

4 EVOLUCIÓN DEL PIPELINE EN TIEMPO REAL.

En este capítulo se analiza la evolución gradual de la implementación del sistema de diarización de hablantes en streaming. Ha sido desarrollado a través de sucesivas versiones del pipeline implementado en Python. A continuación, se describe cada versión ilustrando con fragmentos de código decisiones que han sido claves en el diseño. Se comparan asimismo el rendimiento (tiempo de respuesta y el uso eficiente de recursos), la robustez del sistema y la precisión de identificación de los hablantes conocidos.

4.1 Versión 1: Integración básica del Pipeline.

Consiste en la combinación de un algoritmo de diarización de hablantes en tiempo real con un modelo de reconocimiento de locutor (Speaker ID) pre-entrenado. Se utiliza la librería DIART para la diarización en streaming a partir de una fuente de micrófono, y el modelo ECAPA-TDNN de *SpeechBrain* para la verificación de hablantes.

Por cada segmento de audio detectado, el sistema extrae ese fragmento de audio, lo guarda en un archivo temporal WAV, y lo compara con un pequeño conjunto de audio de referencia de hablantes (*database* local). La comparación se realiza calculando un puntaje de similitud, seleccionando el nombre con mejor puntuación (`best_match`) y renombrando la etiqueta del segmento diarizado. En código, se aprecia en el bucle que recorre `known_speakers` y determina el mejor puntaje, renombrando la etiqueta original.

```
best_score, best_name = float("-inf"), None
for name, ref_wav in self.known_speakers.items():
    try:
        score, _ = self.spk_model.verify_files(temp_path_str, ref_wav)
        score = float(score)  # Convertir tensor a float
    except Exception as e:
        print(f"[RealTimeSpeakerID] Error con {name}: {e}")
        continue
    if score > best_score:
        best_score = score
        best_name  = name

if best_name is not None:
    annotation.rename_labels({speaker_label: best_name}, copy=False)
    print(f"[RealTimeSpeakerID] {speaker_label}({start:.2f}-{end:.2f}s) ->
{best_name}, score={best_score:.2f}")
```

En esta primera versión, se consigue una funcionalidad básica. Se asumió que siempre habría al menos un hablante conocido coincidente. Este comportamiento reduce la precisión en la clasificación de hablantes desconocidos y conocidos, generando falsos positivos de identificación. En términos de tiempo de respuesta, la versión inicial realiza la verificación de hablante de forma síncrona en cada segmento. El modelo ECAPA-TDNN es relativamente pesado, lo que añade una latencia por segmentos (lectura/ escritura de archivo temporal y cálculo del *embedding* de voz) que puede impactar la inmediatez del sistema. No obstante, con pocos hablantes conocidos el retardo es manejable. La robustez es básica, se incluyen bloques *try/ except*, imprimiendo mensaje

de error sin detener la ejecución del pipeline.

Esta versión es usada para demostrar la viabilidad de combinar diarización en tiempo real con identificación de hablantes, aunque carece de identificaciones correctas, optimizaciones y buenas prácticas.

4.2 Versión 2: Umbral de decisión para hablantes conocidos vs. desconocidos.

En esta versión, se intenta mitigar los falsos positivos de identificación. Tal y como se vio en el apartado 3.3, se aplica un umbral de similitud predefinido (`threshold`) el cual hay que superar. Si no se supera, la etiqueta genérica del segmento se deja la etiqueta genérica del segmento. De esta manera, se mejora la precisión de la identificación para casos desconocidos.

Otra modificación relevante fue ajustar la fusión de segmentos en la salida RTTM reduciendo el parámetro `path_collar` de 0.5 s a 0.2 s, lo que significa que la escritura de resultados. Este parámetro sirve para determinar a partir de que tiempo de pausa se considera separar los segmentos. Se modifica con la intención de obtener segmentos más precisos y evitar unir partes de audio separadas por silencio.

En términos de rendimiento, se mantiene un tiempo de respuesta similar, el flujo sigue siendo síncrono.

4.3 Versión 3: Buffer de audio y fusión de intervenciones continuas.

Surgieron nuevos desafíos al operar en tiempo real. El algoritmo de diarización puede emitir múltiples segmentos cortos para un mismo hablante debido a pausas breves o actualizaciones parciales del resultado conforme llega el audio en streaming. En las versiones anteriores, el sistema de identificación trataba cada segmento individualmente dando lugar a inconsistencias (por ejemplo, si un mismo hablante habla durante 10 segundos, pero con pequeñas pausas, la diarización inicial podría generar varios segmentos separados, y posteriormente trataría de identificarlos por separado).

Se aborda este problema introduciendo un buffer de audio continuo y una fusión de segmentos adyacentes antes de la identificación. Se acumulan los datos de forma progresiva en un búfer `numpy` (`self.buf`) y espera a una intervención completa del hablante antes de pasar a identificar. Para determinar cuándo ha terminado una intervención, se aplica un criterio de silencio máximo entre los segmentos (`MAX_SILENCE = 0.3` s en esta versión). Solo cuando un segmento finaliza con un silencio mayor al umbral o cambia la etiqueta, se considera concluida la intervención y se procede a identificarla.

La implementación consta de varios elementos nuevos. Se define un sink personalizado (`RTInterventionID`) que extiende la clase `Observer`. Cada vez que llega un nuevo chunk de audio y su anotación de diarización, el método `on_next` añade los datos al buffer. Luego se aplica la función `merge_turns(anotation, MAX_SILENCE)` para unir los segmentos de la anotación actual.

```
# Extraer chunk y times
data = self._mono(feat.data).astype(np.float32)
step = feat.sliding_window.step
c_start = feat.sliding_window.start
c_end = c_start + len(data) * step

# Acumular buffer
self.buf = np.concatenate([self.buf, data])

# Fusionar intervenciones
```

```
merged = merge_turns(ann, MAX_SILENCE)
for turn, _, old_lbl in merged.itertracks(yield_label=True):
    if turn.end > c_end or (turn.end - turn.start) < MIN_DUR:
        continue
    key = (round(turn.start, 2), round(turn.end, 2))
    if key in self.done: #Conjunto/set de claves (start,end) ya procesadas para
no duplicar trabajo.
        continue
```

En el modo de streaming, cada bloque de audio recibido desde el micrófono (un chunk de varios segundos) puede contener intervenciones de uno o varios hablantes. El diarizador anota esos turnos indicando sus tiempos de inicio y un final provisional. Sin embargo, cuando un turno llega justo hasta el límite del bloque, no puede considerarse cerrado todavía, ya que el hablante podría seguir hablando en el siguiente bloque. Por este motivo, el sistema solo envía a identificación los turnos cuyo final queda completamente contenido dentro del rango temporal ya procesado. Por ejemplo, si en un bloque de 0–5 segundos se detecta que *speaker0* habla de 1.0 a 2.0 s y *speaker1* de 2.0 a 4.5 s, ambos turnos se consideran completos y se identifican. En cambio, si un hablante sigue activo hasta el segundo 5.0 (el final del bloque), su turno permanece abierto y se espera a recibir el siguiente *chunk* para confirmar su cierre. Con esta estrategia, el sistema maneja correctamente la presencia de varios hablantes en un mismo bloque y evita tanto la duplicación de identificaciones como el uso de fragmentos incompletos de audio. Además, se impone una duración mínima (MIN_DUR = 1.0 s) para evitar identificar fragmentos demasiado cortos.

Para evitar duplicar el procesamiento de segmentos ya duplicados, se introdujo un conjunto `self.done` donde se almacena identificadores únicos de segmentos ya procesados (por ejemplo, una tupla con el inicio y fin del segmento). Antes de identificar un turno, se comprueba si ya fue procesado con anterioridad, y en caso afirmativo es ignorado.

Se tuvo que resolver la cuestión de convertir tiempos a índices que asociaran los distintos segmentos que se iban acumulando de manera continua en el buffer. Se calculan los índices de inicio y fin del segmento dentro de buffer teniendo en cuenta el tiempo absoluto del segmento (`turn.start` y `turn.end`) y la tasa de muestreo. Se asume que la tasa de muestro coincide con la del modelo (16 KHz) o se realiza conversión si es necesario usando `resample_poly` para garantizar la compatibilidad con el modelo Speaker ID.

Finalmente, con el segmento de audio completo en memoria (y con un margen opcional de ampliado por delante y por detrás de 0.2 s para no perder contexto), se procede a la identificación de manera análoga a versiones anteriores. En este caso, se introduce explícitamente la etiqueta de “Desconocido” cuando ningún score alcanza el umbral mínimo (`THRESHOLD = 0.25`). Se imprime por consola el resultado de la identificación acompañado por los tiempos de incio y fin, nombre asignado y score obtenido. Tras ello, el identificador de turno se agrega al conjunto done y el archivo temporal es eliminado.

Esta versión incrementó la robustez y coherencia, al conseguir que cada intervención completa del hablante se identifique solo una vez aumentando la precisión de identificación. El sistema de identificación suele producir resultados más fiables con muestras de mayor duración (>1 s).

La latencia aumenta ligeramente, ya que el sistema espera a que termine su turno o haya suficiente silencio para proceder), comportamiento que se considera aceptable en un contexto de diarización, premiando la precisión y estabilidad. El tiempo de respuesta sigue siendo en tiempo real, pues la diarización se realiza continuamente y la identificación ocurre tan pronto como es razonable.

Se introdujo mejoras en el manejo de errores y advertencias. Se manejó dexplícitamente la excepción de `WindowClosedException` que ocurre si se cierra la ventana de visualización de DIART, para que el sistema continue funcionando en segundo plano sin detener el pipeline. Cualquier otra excepción no prevista en `on_next` se captura con `traceback.print_exc()`, imprimiendo la traza de error, pero evitando que el proceso se caiga. Estas prácticas denotan un aumento de robustez.

4.4 Versión 4: Optimización de rendimiento y mejoras de ingeniería.

La cuarta versión se centró en optimizar el rendimiento, especialmente, la escalabilidad temporal y el aprovechamiento de recursos de cómputo. Múltiples mejoras fueron añadidas, entre las cuales destacan:

- Uso de un **buffer circular** limitado para el audio, evitando consumo ilimitado de memoria.
- **Paralelización del cálculo de similitud con los distintos hablantes** conocidos mediante *threads* para reducir la latencia de identificación.
- **División de segmentos largos** en subsegmentos solapados para obtener medidas de similitud más estables y robustas.
- **Parametrización** más **configurable**.
- Sustitución de los *print* por un sistema de **logging estructurado**, con niveles de severidad y registro a fichero.

En la versión anterior, el buffer self.buf crecía continuamente. En esta nueva versión, se estableció un tamaño máximo en segundos (`--max-buffer-seconds`, por defecto 60 s). Esto, implicó llevar la cuenta del tiempo de inicio que corresponde al primer índice del buffer (`self.buffer_start_time`). Cada vez que se añade un nuevo fragmento de audio al buffer, comprueba si excede ese máximo permitido de muestras, si es así, se descarta la parte más antigua actualizando `buffer_start_time` en consecuencia. El fragmento siguiente ilustra este manejo:

```
self.buffer = np.concatenate([self.buffer, wave])
            max_samples = int(self.max_buffer_seconds * self.SR)
            if len(self.buffer) > max_samples:
                excess = len(self.buffer) - max_samples
                self.buffer = self.buffer[excess:]
                self.buffer_start_time += excess / self.SR
```

Esto provoca que el uso de memoria se vea reducido incluso cuando hay ejecuciones prolongadas. El valor de unos 60 segundos de historial permite retener suficiente contexto temporal sin afectar a la identificación, ya que es improbable que se necesite audio de un minuto más atrás para deducir la etiqueta del turno presente.

Se usa un `ThreadPoolExecutor` con *workers* equivalentes al número de núcleos disponibles para lanzar verificaciones en paralelo. Al procesar un segmento, se envían tareas concurrentes para cada comparación `verify_files`, recopilando los resultados según vayan concluyendo. En código:

```
futures = {
    self.executor.submit(self.spk_model.verify_files, tmp_path, ref): name
    for name, ref in self.known_speakers.items()
}
for fut in as_completed(futures):
    name = futures[fut]
    try:
        score, _ = fut.result()
        avg_scores[name] += float(score)
    except Exception as ex:
```

```
        logging.warning(f"Fallo verificación {name} en subventana: {ex}")
```

Conforme cada *thread* termina, se extrae el resultado y se acumula el score en un diccionario de puntajes (`avg_scores`). Ahora, comparar los hablantes al mismo tiempo puede tardar lo mismo que comparar uno solo, mejorando el tiempo de respuesta global del sistema cuando el número de hablantes es significativo. Cabe destacar, que el modelo de *SpeechBrain* aprovecha `Pytorch` en su *backend*, el cual libera la GIL (*Global Interpreter Lock*) durante operaciones tensoriales, por lo que el *threading* en Python aquí puede efectivamente paralelizar cálculos pesados de forma eficiente.

Para intervenciones más largas, podría ocurrir que la similitud calculada resulte menos confiable debido a las variaciones internas. Por ello, se optó por dividir cada intervención en sub-ventanas de 1.0 s con solapamiento del 50% y promediar los resultados de verificación a lo largo de ellas. Este método, de *sliding window*, permite muestrear el audio en porciones manejables y homogéneas temporalmente.

Para un turno de 4 segundos, se evaluarían submuestras solapadas en lugar de una sola comparación de 4 s. Como se aprecia en la Ilustración 4.1:

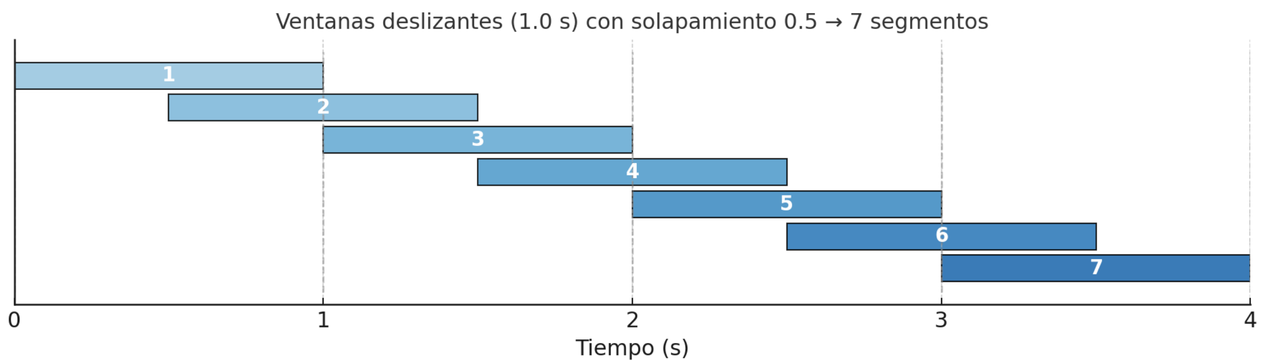

Ilustración 4.1 - Representación temporal de subsegmentos solapados (ventanas deslizantes de 1 s con solapamiento del 50%)

La lógica implementada es la siguiente:

- Fijar duración de ventana = 1.0 s, solapamiento = 0.5 (50%). Entonces el paso entre ventanas es 0.5s.
- Calcular el número de subsegmentos necesarios para cubrir el segmento:

  `num_sub = floor((duracion_segmento - 1.0) / 0.5) + 1` (al menos 1)
- Para i desde 0 hasta `num_sub - 1`: extraer subsegmento de audio desde `start + i*0.5` hasta `start + i*0.5 + 1.0` segundos (asegurando no pasar del final)
- Verificar cada subsegmento contra todos los hablantes conocidos (en paralelo como, se ha explicado anteriormente) y acumular los scores en un promedio para la intervención completa.
- Al final, se divide cada score acumulado por el número de subsegmentos y se toma la mejor coincidencia promedio.

Siendo el código:

```
window_dur, overlap = 1.0, 0.5
step_sub = window_dur * (1 - overlap)
seg_dur = len(segment) / self.SR
num_sub = max(1, int(np.floor((seg_dur - window_dur) / step_sub)) + 1)

avg_scores = {name: 0.0 for name in self.known_speakers}
for i in range(num_sub):
    sub_start = int(i * step_sub * self.SR)
    sub_end = int(min(len(segment), sub_start + window_dur * self.SR))
    subseg = segment[sub_start:sub_end]
    if len(subseg) == 0:
        continue
```

Que la decisión de identificación para un turno largo no dependa de un único valor de similitud, sino de la media de múltiples estimaciones, tiende a suavizar errores ocasionales ya dar mayor robustez frente a la variabilidad de audio. El coste computacional incrementa linealmente con la duración del segmento (un segmento de 8 s se subdividiría en unos 15 subsegmentos, con múltiples verificaciones cada uno). No obstante, la paralelización mitiga parte de este costo.

En pruebas, esta técnica mejoró la estabilidad de la identificación, el score final era más consistente y superaba el umbral de manera más confiable cuando efectivamente era el hablante registrado.

Como otras mejoras técnicas, se agregaron argumentos de líneas con el usa de agruparse para configurar parámetros como la frecuencia de muestreo objetivo, umbral de score, duraciones mínimas, longitud del búfer, etc., … con los valores por defecto. Se hizo para facilitar la experimentación con distintos parámetros sin modificar el código fuente.

```
def parse_args():
    parser = argparse.ArgumentParser(
        description="Diarización e identificación de hablantes en streaming
mejorado"
    )
    parser.add_argument("--sample-rate", type=int, default=16000,
                        help="Frecuencia de muestreo objetivo (Hz)")
    parser.add_argument("--threshold", type=float, default=0.2,
                        help="Similitud mínima para asignar hablante conocido")
    parser.add_argument("--max-silence", type=float, default=0.3,
                        help="Silencio máximo para fusionar turns (s)")
    parser.add_argument("--min-duration", type=float, default=0.5,
                        help="Duración mínima de intervención (s)")
    parser.add_argument("--margin", type=float, default=0.2,
                        help="Márgen extra en segmento de audio (s)")
    parser.add_argument("--max-buffer-seconds", type=float, default=60,
                        help="Máximo histórico de audio en buffer (s)")
    parser.add_argument("--log-file", type=Path,
default=Path("logs/stream_id.log"),
                        help="Ruta para el fichero de log")
    return parser.parse_args()
```

Se reemplazaron los comentarios basados en *print* por una configuración de *logging* (módulo *logging* de Python). Al inicio del programa, `setup_logging` configura un formateador que registra la hora, nivel y mensaje, tanto a consola como a un archivo de log. A lo largo del código, las llamadas se fueron modificando según correspondiera `logging.info()`, `logging.warning()`, etc., según corresponda. Lo que permite poder filtrar los mensajes según nivel de importancia, quedando guardados en un archivo para el análisis posterior, esencial para depuración y mantenimiento. Se puede observar el código:

```
def setup_logging(log_file: Path):
    log_file.parent.mkdir(parents=True, exist_ok=True)
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s [%(levelname)s] %(message)s",
        handlers=[
            logging.FileHandler(log_file, encoding='utf-8'),
            logging.StreamHandler()
        ]
    )
```

Adicionalmente, se implementó una captura de señales del sistema operativo para finalización limpia. Con el módulo `signal`, se interceptan *SIGINT* (Ctrl + C) y *SIGTERM*, registrando un mensaje y lanzando una excepción para romper el bucle de inferencia. Asegurando que bloque *finally* imprima mensajes finales y libere recursos de forma ordenada cuando se detenga el programa manualmente.

```
def handle_sig(signum, frame):
        logging.info(f"Señal {signum} recibida, finalizando...")
        raise KeyboardInterrupt
    signal.signal(signal.SIGINT, handle_sig)
    signal.signal(signal.SIGTERM, handle_sig)

    logging.info("Iniciando diarización e identificación...")
    try:
        inference()
    except KeyboardInterrupt:
        logging.info("Interrupción por usuario, cerrando pipeline...")
    except Exception:
        logging.exception("Error crítico en el pipeline")
    finally:
        logging.info("Proceso finalizado.")
```

En cuanto a los resultados e impacto, con estas modificaciones, se podía correr durante largos periodos sin incremento significativo de memoria gracias al buffer circular, y la latencia se mantenía baja incluso con múltiples hablantes conocidos, gracias al uso paralelo de la CPU.

Tras esta versión altamente optimizada, se experimentó con enfoques alternativos más simples para asegurar que la complejidad introducida fuera necesaria y ver si se podía usar un diseño más directo. Este análisis dio lugar a la versión 5.

4.5 Versión 5: Agrupación por silencio y procesamiento al final de un turno.

Esta versión, se planteó con el enfoque de preservar los avances logrados hasta ahora, pero con una lógica más clara y menos fragmentada. Se reestructuró el sistema de identificación para centrarse en la noción de **turno de voz completo**. Para ello, se utilizan: el silencio entre intervenciones y etiquetas genéricas proporcionadas por la diarización. Básicamente, el objetivo fue reagrupar la lógica de la versión 3 (fusión de segmentos continuos) y la versión 4 (procesamiento al final del turno) de una forma más sencilla y legible.

El *sink* de identificación (`RealTimeSpeakerID`) implementa un mecanismo de acumulación secuencial. Mantiene un estado de "intervención en curso" con variables para la etiqueta genérica actual, tiempo de inicio y fin, tasa de muestreo y un búfer de audio. Cada vez que nuevos segmentos son recibidos en `on_next`, el algoritmo recorre esos segmentos en orden temporal y decide si pertenecen a la misma intervención actual o si marcan el fin de la intervención actual y el potencial inicio de otra nueva.

La regla aplicada es la siguiente:

- Si llega un segmento con la misma etiqueta genérica que el anterior (*Speaker0*, *Speaker1*, etc.)
    - Y no ha transcurrido un tiempo prolongado entre el final del último y el inicio del nuevo → mismo turno y se agrega el audio al búfer.
    - Ha transcurrido un tiempo prolongado entre el final del último y el inicio del nuevo → turno anterior concluyó.
- Si el segmento entrante pertenece a un hablante genérico distinto → turno anterior concluyó.

La regla podría representarse con el diagrama de bloques plasmado en la Ilustración 4.2.

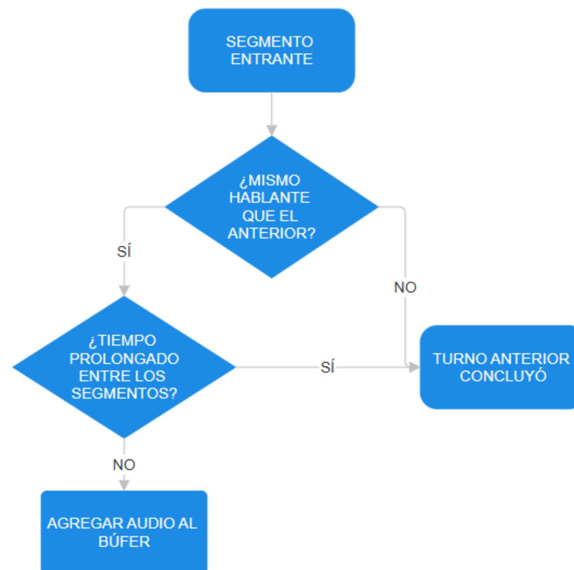

Ilustración 4.2 - Lógica de decisión en la agrupación de segmentos de un mismo hablante.

Se definió un `SILENCE_DUR` como umbral de pausa para segmentar intervenciones, más largo que en la versión 3, con la idea de tolerar pausas naturales dentro de una misma intervención (por ejemplo, tomar aire) sin fragmentarla.

El código del procesamiento sería el siguiente:

```
# Recorremos los turnos en orden cronológico
for turn, _, gen_label in ann.itertracks(yield_label=True):
    start, end = turn.start, turn.end
    # Detectar pausa o cambio de etiqueta genérica
    if (self.cur_end is not None and
        start - self.cur_end) >= SILENCE_DUR or gen_label != self.cur_generic)):
        self._flush()  # cierra intervención anterior
```

Se pude observar cómo se invoca `_flush()` para finalizar la intervención anterior. Esta función se encarga de concatenar todo el audio acumulado, verificar si se supera la duración mínima requerida (`MIN_DUR`) y entonces realizar sobre ese turno completo la identificación de hablante.

El siguiente fragmento muestra la parte donde se descartan las intervenciones muy cortas y se ejecuta la verificación:

```
def _flush(self):
        """Se llama cuando termina una intervención (silencio o cambio de
gen_label)."""
        if not self.buf_audio:
            self.reset(); return

        total_dur = sum(len(seg) for seg in self.buf_audio) / self.sr
        if total_dur < MIN_DUR:
            # Demasiado corta → descartamos
            print(f"[DEBUG] Intervención {self.cur_generic} ({total_dur:.2f}s) <
MIN_DUR; ignorada.\n")
            self.reset(); return

        # Concatenar audio y guardarlo en WAV temporal
        utter = np.concatenate(self.buf_audio)
        tmp_wav = self.tempdir / f"{uuid.uuid4().hex}.wav"
        sf.write(tmp_wav.as_posix(), utter, self.sr)

        # Verificación con SpeechBrain
        best_score, best_name = float("-inf"), None
        for name, ref in self.known.items():
            try:
                score, _ = self.model.verify_files(tmp_wav.as_posix(), ref)
                score = float(score)
            except Exception:
                continue
            if score > best_score:
                best_score, best_name = score, name

        tmp_wav.unlink(missing_ok=True)
```

```
        label = best_name if best_score >= SCORE_THRESH else "desconocido"
        print(f"[Utterance] {label}: {total_dur:.2f}s "
              f"({self.cur_start:.2f}-{self.cur_end:.2f}s) "
              f"[score={best_score:.2f}]\n")
        self.reset()
```

Tras imprimir el resultado, `reset()` borra el estado para empezar a acumular una nueva intervención cuando corresponda.

Es importante destacar algunas diferencias respecto la versión anterior (versión 4):

- Se eliminó la subsegmentación y paralelismo. Cada intervención se analiza como una sola unidad de audio (para la simplificación del proceso) y las comparaciones con hablantes conocidos se hacen secuencialmente `for name, ref in self.known.` Se podría decir que esta secuencialidad no impacta gravemente al tiempo real por dos motivos: el número de hablantes conocidos suele ser manejable, y la identificación solo ocurre al final de cada turno (no de manera continua para cada chunk).
- Ahora, los segmentos se agrupan desde la función `on_next` (en lugar de ser pre-fusionados con una función aparte), el código resulta más lineal y sencillo de seguir. Se introdujo un *sink* `PrintSegments` para imprimir todos los segmentos diarizados que fueron detectados, sin interferir en la lógica de identificación, lo que es útil para la depuración. De la siguiente forma:

```
class PrintSegments(DiarizationSink):
    #value puede ser una anotación o tupla con anotación y otros artefactos
    def on_next(self, value):
        ann = None
        if isinstance(value, tuple): #si value es una tupla
            for v in value:
                if hasattr(v, "itertracks"): #nos quedamos con la anotación
                    ann = v
        else:
            #si no es tupla, se asume que value ya es la anotación
            ann = value
        if ann is None:
            return
        for turn, _, speaker in ann.itertracks(yield_label=True):
            print(f"[PrintSegments] {turn.start:.2f}-{turn.end:.2f}s →
{speaker}")
```

- Se aumentó la tolerancia de silencio dentro de una intervención, permitiendo que el hablante pause brevemente dentro de una intervención sin desencadenar un *flush*. Para decidir este parámetro hay que tener en cuenta el compromiso de no fragmentar la voz de un mismo hablante y no retrasar demasiado la identificación desde que terminó realmente de hablar.

Esta versión ofrece resultados que son comparables en exactitud de identificación y robustez con la versión anterior, pero ofrece una implementación más concisa. Cada intervención completa seguía identificándose correctamente. El rendimiento seguía siendo en tiempo real, aunque prescindiendo del paralelismo, lo que redujo cierta sobrecarga asociada al hecho de no crear múltiples *threads* ni manejar múltiples archivos temporales por

intervención. La penalización por secuencialidad es mínima en entornos con pocos hablantes conocidos, como es el caso de esta experimentación.

La robustez permaneció alta: conservando la captura genérica de excepciones en `on_next` para imprimir cualquier error inesperado sin colapse, y se mantuvo el uso de `missing_ok = True` para evitar errores incluso cuando se intenta borrar el WAV temporal incluso si no existe.

4.6 Versión Final: Versión robusta con temporización automática y ajustes finos.

La sexta y última versión sirvió para consolidar todo lo aprendido en versiones anteriores: integrando esa simplicidad de enfoque por intervenciones completas de la versión 5 con algunas optimizaciones adicionales que se inspirarán en la versión 4. Además, se ajustarán los parámetros para refinar el comportamiento del pipeline.

Destacan las siguientes mejoras:

- **Detección automática de fin de intervención mediante temporizador**. Se introdujo un temporizador (`timer thread`) para efectuar el *flush* de la intervención actual cuando hay inactividad (es decir, silencio) durante un periodo determinado. Importante, independientemente de la llegada de un nuevo segmento o del evento `on_completed`.
- **Límite de la duración de la intervención**. El parámetro `MAX_UTTER_DUR` sirve para establecer una duración máxima de audio acumulado. Tras esta duración, se fuerza el *flush*, evitando así, acumulaciones excesivamente largas de voz de un mismo hablante.
- **Ajuste algo más fino del umbral de identificación**. El `SCORE_THRESH` fue elevado a 0.4 (40%) para exigir una mayor similitud.
- **Afinado de la configuración de diarización**. Se usaron los parámetros específicos de `SpeakerDiarizationConfig` (aplicables para una versión de DIART menor de la 0.8), como la `delta_new=0.9` y `segmentatyion_threshold=0.45`, con el fin de minimizar fragmentaciones incorrectas y detecciones espurias de cambio de locutor
- **Combinación de etiquetas en la salida RTTM más laxa**. Se aumentó el `path_collar` a 0.7 s lo que es más coherente con el incremento en el valor de SILENCE_DUR, de modo que cuando se escribe la anotación final se permitirían solapes de hasta 0.7 s para unir segmentos conjuntos (las pausas menores a ese valor se consideran dentro del mismo turno).

La estructura general del *sink* de identificación es similar a la de la versión anterior, pero con la inclusión de un manejo de temporizador. En lo que a la acumulación de audio en `on_next` refiere, después de agregar un segmento al buffer, ya no se confía solo en la detección de un nuevo segmento o en `on_completed` para decidir cuándo hacer el *flush* y considerar como terminada la intervención. Ahora, se llama a un `_schedule_timer()` para iniciar a reiniciar un temporizador del silencio cada vez que hay una nueva actividad. El temporizador ha sido implementado con `threading.Timer`, y configurado para ejecutar `_flush` automáticamente cuando transcurre un intervalo sin recibir audio (usando precisamente `SILENCE_DUR`, que se mantuvo en 1.2 s). El método `_schedule_timer` cancela cualquier temporizador previo en curso y lanza uno nuevo de duración `SILENCE_DUR`. Así, cuando un hablante deja de hablar y la diarización para de emitir segmentos para esa etiqueta, el tiempo vencerá y disparará `_flush` en segundo plano. Esto, permite terminar la intervención de manera oportuna sin la necesidad de tener que esperar a que otro evento ocurra. Se implementa de la siguiente manera.

```
def _schedule_timer(self):
        # Programar un flush tras SILENCE_DUR segundos de inactividad
        if self.timer: self.timer.cancel()
        self.timer = threading.Timer(SILENCE_DUR, self._flush)
        self.timer.start()
```

La función `_flush` en la versión 6 es semejante a la de versión 5, con la diferencia de que ahora también puede ser llamada desde el hilo del temporizador. Por ello, se asegura de cancelar el temporizador pendiente (si existe) al entrar a *flush*, y luego procede a verificar si hay audio acumulado y suficiente duración. Si la intervención es válida (≥ `MIN_DUR`), concatena el audio y calcula los scores contra los conocidos exactamente igual que antes, eliminando el archivo temporal al final. Se mantiene la decisión con umbral, pero con el nuevo valor 0.4:

```
label = best_name if best_score >= SCORE_THRESH else "desconocido"
```

El incremento del umbral a 0.4 implica exigir mayor certeza para etiquetar un hablante conocido; esto se ajustó tras analizar que en versiones previas aún podían ocurrir asignaciones incorrectas con scores marginales (~0.25-0.3) para voces algo parecidas. Con 0.4, se prioriza la precisión sobre la sensibilidad, asumiendo que, si un hablante conocido habla, la similitud con su referencia generalmente superará ese valor con el modelo ECAPA, mientras que valores menores suelen indicar que es otra persona. En pruebas finales, este ajuste prácticamente eliminó identificaciones erróneas residuales, a costa de marcar algún turno de un conocido como "desconocido" solo en casos muy cortos o con mucho ruido (lo cual es prudente). Se puede apreciar el método flush():

```
def reset(self):
        # Restablecer buffer y metadatos del turno actual
        self.cur_generic = self.cur_start = self.cur_end = self.sr = None
        self.buf_audio = []


def _flush(self):
        # Procesar el buffer acumulado al detectar silencio o al superar
MAX_UTTER_DUR
        if self.timer:
            self.timer.cancel(); self.timer = None
        if not self.buf_audio:
            self.reset(); return
        total_dur = sum(len(s) for s in self.buf_audio) / self.sr  # Duración
total acumulada
        if total_dur < MIN_DUR:
            print(f"[DEBUG] {self.cur_generic}
({total_dur:.2f}s)<MIN_DUR;ignored")
            self.reset(); return
        # Concatenar y guardar en un WAV temporal
        utter = np.concatenate(self.buf_audio)
        tmp = self.tempdir / f"{uuid.uuid4().hex}.wav"
        sf.write(tmp.as_posix(), utter, self.sr)
        # Comparar vs cada orador conocido y quedarnos con el mejor score
        best_score, best_name = float("-inf"), None
```

```
        for nm, ref in self.known.items():
            try:
                sc, _ = self.model.verify_files(tmp.as_posix(), ref)
                sc = float(sc)
            except:
                continue
            if sc > best_score: best_score, best_name = sc, nm
        tmp.unlink(missing_ok=True)  # Borrar archivo temporal
        label = best_name if best_score >= SCORE_THRESH else "desconocido"
        st = self.cur_start or 0.0; ed = self.cur_end or (st + total_dur)
        print(f"[Utterance] {label}: {total_dur:.2f}s ({st:.2f}-{ed:.2f}s)
[score={best_score:.2f}]")
        self.reset()
```

Además del uso del temporizador, se incorporó en `on_next` la lógica de *flush* forzado por duración máxima.

Tras añadir el fragmento, se comprueba si la duración de la intervención actual (`self.cur_end - self.cur_start`) alcanza `MAX_UTTER_DUR` (8 s). Si se supera, se llama a `_flush()` inmediatamente, incluso si el hablante no ha terminado para evitar acumular audios muy extensos en una sola identificación. Este valor debe tener en cuenta varios puntos: intervenciones muy largas podrían introducir latencia notable en la identificación final y riesgo de errores si el hablante en realidad cambió y la diarización no se dio cuenta.

También, segmentar forzosamente garantiza, que, en discursos prolongados, el sistema produzca actualizaciones periódicas de identificación (por ejemplo, en una conferencia donde un ponente habla durante minutos seguidos, se obtendrían identificaciones parciales cada ciertos segundos, evitando el silencio prolongado del sistema). Al hacer *flush* por tiempo máximo, el mecanismo del temporizador se reinicia para la siguiente acumulación, asegurando que no haya confusión entre segmentos anteriores y nuevos. Así se desarrolló:

```
def on_next(self, value):
        try:
            ann, wav = None, None
            # Extraer anotación y feature de wav
            if isinstance(value, tuple):
                for v in value:
                    if hasattr(v, "itertracks"): ann = v
                    if hasattr(v, "data") and hasattr(v, "sliding_window"):wav=v
            else:
                ann = value
            if not ann or not wav: return
            # Frecuencia de muestreo deducida del paso de ventana
            sr = int(1 / wav.sliding_window.step)
            # Para cada segmento detectado, extraer subarray y acumular
            for turn, _, gen in ann.itertracks(yield_label=True):
                s, e = turn.start, turn.end
                if self.cur_generic is not None and gen != self.cur_generic:
                    self._flush()
                s0 = int((s - wav.sliding_window.start) * sr)
                s1 = int((e - wav.sliding_window.start) * sr)
                if s1 <= s0: continue
                seg = wav.data[max(0, s0):min(wav.data.shape[0], s1)]
```

```
                if not self.buf_audio:
                    self.cur_generic = gen; self.cur_start = s; self.sr = sr
                self.buf_audio.append(seg); self.cur_end = e
                if (self.cur_end - self.cur_start) >= MAX_UTTER_DUR:
                    self._flush()
                else:
                    self._schedule_timer()
        except Exception as e:
            print(f"[ERROR] {e}"); traceback.print_exc()
```

Por último, se aplicó la configuración personalizada de diarización al instanciar el pipeline de DIART. Los parámetros usados:

- `delta_new` : define la cantidad de evidencia requerida el sistema para crear un nuevo ID de hablante. Cuanto mayor sea su valor, más firme debe ser la evidencia acústica de que se trata de una nueva voz; de lo contrario, se seguirá asignando los segmentos al hablante previo, evitando fluctuaciones innecesarias en las etiquetas.
- `segmentation_threshold` : controla el nivel de sensibilidad del detector de voz. Determina el umbral de probabilidad por encima del cual un fragmento se considera como voz y no como silencio. Los valores más altos hacen que el sistema sea más estricto, detectando únicamente las partes con voz clara y reduciendo falsas detecciones por ruido.
- `min_duration_off` : indica la duración mínima del silencio necesaria para que considere que un hablante ha dejado de hablar. Si este valor es demasiado pequeño, el pipeline puede fragmentar una misma intervención en varios segmentos.

```
# — Configuración de diarización para Diart ≤0.8
config = SpeakerDiarizationConfig(
    delta_new = 0.9,             # Umbral para detectar nuevo hablante
    segmentation_threshold = 0.45,  # Sensibilidad de la segmentación/VAD
    min_duration_off = 0.3,       # Rellena pausas muy breves
    max_num_speakers = 4          # Máximo de hablantes simultáneos
)
```

Es importante añadir que, aunque `min_duration_off` contribuye a reducir la fragmentación de los turnos dentro del propio modelo de diarización, no sustituirá la función del buffer y el temporizador. El diarizador procesa el audio en chunks y entrega resultados parciales y provisionales a medida que avanza la captura. Es por ello que, si un hablante continúa hablando más allá del final de un bloque, el modelo no puede confirmar que la intervención haya concluido hasta el siguiente fragmento. Esto implica que, sin un mecanismo adicional, el sistema no tendría forma de saber con certeza cuándo cerrar un turno ni cuándo ejecutar la identificación del hablante.

En consecuencia, estos parámetros permiten que las anotaciones de diarización sean más coherentes, lo que se puede traducir en un mejor funcionamiento de la lógica. Eso sí, al haber menos rupturas falsas en los segmentos diarizados, la acumulación de audio corresponde de una manera fiel a los turnos de voz reales. Esto redunda en que el *flush* ocurra exactamente en los momentos apropiados. La elección de `PATH_COLLAR = 0.7` en la escritura RTTM complementa esta configuración, ya que permite unir en la salida final segmentos cuyos límites están separados por menos de 0.7 s, de modo que la representación final de los turnos en el archivo RTTM sea consistente con la lógica interna de 1.2 s de silencio para separar intervenciones.

Los parámetros escogidos son los siguientes:

```
# — Parámetros de turno / clustering
MIN_DUR = 0.5   # Duración mínima (segundos) para considerar un fragmento válido
SILENCE_DUR = 1.2   # Tiempo de silencio (segundos) tras el cual se procesa el buffer
MAX_UTTER_DUR = 8.0   # Duración máxima de un turno antes de forzar su análisis
SCORE_THRESH = 0.4   # Umbral de score mínimo para asignar un hablante conocido
PATCH_COLLAR = 0.7   # Collaring al escribir anotaciones RTTM para solapar un poco
```

Se consideraron algunos aspectos en términos de **rendimiento** y **robustez**. El uso del temporizador de inactividad añade una capa de complejidad concurrente (ya que conlleva un hilo aparte llamando a `_flush`), por lo que se tuvo en cuenta durante el diseño: el temporizador siempre se reinicia en cada nueva actividad, y solo actúa cuando no hubo más audio en el intervalo previsto. Eso mejoró la **respuesta** del sistema en diálogos al no tener que esperar que la diarización declare un nuevo hablante para cerrar el turno anterior. El *flush* automático y el límite de duración de intervención aseguran salidas periódicas y reusar el buffer de forma controlada.

En conclusión, a lo largo de las versiones se muestra un progreso sustancial en todos los aspectos del sistema de diarización e identificación de hablantes en tiempo real. Se abordaron problemas, se incorporaron mejoras de diseño y nuevas técnicas para una mejor precisión, reducción de los errores y eficiencia. La versión final combina las estrategias de acumulación de audio por turnos, temporización basada en silencio, procesamiento optimizado y parámetros afinados para lograr un pipeline que sea robusto, rápido y fiable.

5 IMPLEMENTACIÓN DEL SISTEMA DE DIARIZACIÓN E IDENTIFICACIÓN EN DIVIVO.

Se describe la integración completa del sistema de diarización e identificación de hablantes en tiempo real dentro de DiViVo Lite, una versión ligera y modular de la plataforma DiViVo usada como entorno de prototipado. El principal objetivo es **crear un filtro de diarización que conserve solo los segmentos correspondientes al hablante deseado** (en este caso, llamado Pablo) para que prosigan por las siguientes etapas del sistema DiViVo. Se explica cómo se adaptó el pipeline desarrollado previamente y se detalla la arquitectura del sistema, donde el módulo de diarización se encuentra entre la captura de audio y las salidas de identificación y transcripción, los que operan mediante un flujo de eventos con observadores.

5.1 Captura de Audio y Preprocesamiento (módulo *speech_microphone.py*).

Este es el componente encargado de la captura de audio desde el micrófono. Utiliza la biblioteca PyAudio para acceder al dispositivo de audio en tiempo real y divide la señal en chunks de duración fija (en este caso, unos 0.5 s). Cada chunk capturado pasa por un filtro de detección de actividad de voz (VAD) para determinar si contiene habla o solo es silencio/ruido del fondo. Esto permite el ahorro de recursos, al evitar el procesamiento de segmentos sin voz.

El proceso básico de captura y preprocesamiento es el siguiente:

1. **Captura de audio en chunks.** Se abre un flujo de audio mono a 16 kHz y se leen periódicamente buffers de tamaño fijo.
2. **Detección de voz (VAD).** Cada buffer se analiza con un algoritmo VAD (ya implementado en DiViVo, el cual es modelo WebRTC VAD) para decir si contiene o no voz humana. Si el chunk en cuestión es silencio, se descarta y el bucle continúa con el siguiente fragmento.
3. **Envío a diarización.** Si se ha indicado que hay voz, el fragmento se envía al módulo de diarización mediante la llamada al método `diarization.process_chunk(chunk)`. Este método integrará el fragmento en el pipeline de diarización en streaming.
4. **Filtrado por hablante objetivo.** Una vez que el módulo de diarización procesa el audio y determina si la voz corresponde a ese hablante objetivo (en este caso, denominado Pablo).
    a. Si `process_chunk` retorna `True`: el módulo captura invoca a `diarization.extract_intervention_chunks()` para obtener solo aquellos segmentos de audio que corresponden a Pablo, y esos segmentos que han sido filtrados se transmiten a las siguientes etapas de DiViVo.
    b. Si `process_chunk` devuelve `False`, significa que el fragmento no contiene intervención por parte de Pablo y no se reenvía nada.

En el siguiente fragmento de código se ilustra la lógica descrita donde tras leer un chunk de audio, se aplica el VAD y se llama al procesador de diarización.

```
is_speech = True
if self.vad:
    LOGGER.debug("Checking VAD")
    is_speech, first_speech_chunk = self.vad.process_chunk(in_data)

    # Caso A: primer chunk tras detectar inicio de voz
    if is_speech and first_speech_chunk:
    # Reinyectar también los chunks “previos al habla” que guardó el VAD
        chunks_before = self.vad.extract_chunks_before()
        for chunk in chunks_before:
            must_extract = self.diarization.process_chunk(chunk)
            if must_extract:
                # Se ha detectado a Pablo → extraemos y entregamos
                chunks_before = self.diarization.extract_intervention_chunks()
                [self._buff.put(chunk_data) for chunk_data in chunks_before]

    # Caso B: seguimos dentro de voz (no es el primer chunk)
    elif is_speech:
        # Si hay consumidores esperando, avisamos que hay audio
        if self.barrier_available_chunks.n_waiting > 0:
            LOGGER.debug("Opening ASR barrier")
            self.barrier_available_chunks.wait()

        LOGGER.debug("Filled buffer for ASR")
        must_extract = self.diarization.process_chunk(in_data)
        if must_extract:
            chunks_before = self.diarization.extract_intervention_chunks()
            [self._buff.put(chunk_data) for chunk_data in chunks_before]

   # Guardado de audio crudo (si está activado)
   if self._write_audio:
      self.audio_chunks.put(in_data)

# Continuar el stream
return None, pyaudio.paContinue
```

En este código, el método `process_chunk` del objeto `diarization` encapsula la lógica de diarización streaming (desarrollada en `CDiarization`, que es descrita más adelante). Nótese que solo cuando `process_chunk` indica que el hablante ha sido reconocido en el segmento procesado, se extraen los fragmentos correspondientes a su voz mediante `extract_intervention_chunks()` y se envían al resto del sistema.

También hay que añadir que cuando se detecta por primera vez el habla, se añade chunks previos al habla para asegurar que no se pierde información del principio.

5.2 Clase *CDiarization*: Gestión del Pipeline de Diarización (módulo *speech_diarization.py*).

Esta clase es el núcleo del filtro que se ha implementado, responsable de orquestar la diarización en tiempo real sobre los chunks de audio que van entrando y decidir cuales contienen la voz de Pablo. Esta clase inicializa y gestiona apoyándose en observadores (*observers*). A continuación, se describen los métodos más relevantes de esta clase y su interacción con el resto del sistema:

5.2.1 process_chunk(chunk: np.ndarray) -> bool.

Método principal que recibe un fragmento de audio (normalizado a la tasa de muestreo y formato adecuados) y lo ingresa al pipeline de diarización. Internamente, este método alimenta el chunk al sistema de diarización. Durante este procesamiento, los observadores asociados (como `RealTimeSpeakerID`) analizarán por detrás el audio para determinar si corresponde al hablante Pablo. Si se detecta que el hablante efectivamente es Pablo (o en la intervención en curso que abarca dicho fragmento), este método devolverá `True`; de lo contrario, `False`. Esto permite al módulo de captura saber si debe reenviar el audio a las siguientes etapas.

```
def process_chunk(self, audio_chunk):
        """
        Alimenta un nuevo trozo de audio al sistema.
        """
        # 1) Inyecta el chunk en la cola de la fuente
        self.audio_src.feed(audio_chunk)

        # 2) Acumula para devolver trozos más tarde
        #print(f"Chunk recibido: {len(audio_chunk)} bytes")
        self.chunks_before.append(audio_chunk)

        # 3) Señala cuando se detecte Pablo
        #print(f"¿ES PABLO:?{self.has_detected_pablo}")
        has_detected_pablo = self.has_detected_pablo
        self.has_detected_pablo = False
        return has_detected_pablo
```

5.2.2 send_result().

Este método es invocado cuando se ha identificado exitosamente al hablante objetivo en el audio procesado. Su función es notificar de forma interna a `CDiarization` de que se ha encontrado una intervención de Pablo. Esto se implementa estableciendo un *flag*. El observador de identificación de hablante (descrito más adelante) llama a `send_result` una vez que reconoce a Pablo, lo que provoca que la próxima llamada a `process_chunk` indique éxito. Así, se consigue desacoplar la detección en el propio pipeline la cual es asíncrona de la respuesta del bucle principal.

```
def send_result(self):
        """
        Envío final de resultados.
        """
        # Aquí se implementaría el envío de resultados finales
        self.has_detected_pablo = True
```

5.2.3 extract_intervention_chunks() -> List[np.ndarray].

Una vez que se ha identificado a Pablo, este método es el encargado de recopilar y devolver los segmentos de audio correspondientes a esa intervención. Dado que el pipeline de diarización segmenta el audio por hablante. `CDiarization` mantiene un buffer de audio donde acumula los chunks desde el comienzo de una posible intervención de Pablo hasta su final (marcado por silencio o cambio de hablante). Cuando `send_result` indica que esa intervención es de Pablo, este método extrae del buffer solo esos datos y receta el buffer para la siguiente intervención.

```
  def extract_intervention_chunks(self):
        """
        Devuelve los chunks acumulados y vacía el buffer.
        """
        chunks = self.chunks_before.copy()
        self.chunks_before.clear()
        return chunks
```

5.2.4 reset_buffer().

Es el método auxiliar de reinicio del buffer de audio y cualquier estado interno de la clase tras procesar una intervención (ya sea válida o descartada). Esto prepara al sistema para comenzar a analizar una nueva posible intervención desde cero, evitando contaminación entre fragmentos que sean sucesivos.

```
def reset_buffer(self):
        """
        Resetea el buffer de audio y el estado VAD.
        """
        self.chunks_before.clear()
        self.has_detected_pablo = False
```

Internamente, la clase `CDiarization` inicia el pipeline de diarización en su constructor (`__init__`).

Allí, se configura la instancia con los parámetros deseados, se crea la fuente de audio y se adjuntan los observadores necesarios. En código:

```
def __init__(self):
        self.threshold = 0.3
        self.has_detected_pablo = False
        self.chunks_before = []

        # 1. Configuración
        self.cfg = get_diarization_config()
        # 2. Fuente de audio basada en trozos
        self.audio_src = AudioSource(sample_rate=16000)
        # 3. Modelo de hablante
        self.spk = SpeakerModel()
        # 4. Sinks
        self.printer    = PrintSegments()
        self.identifier = RealTimeSpeakerID(self.spk.model,
                                            self.spk.known,
                                            diarization_class=self)
```

```
        # 5. Pipeline
        self.pipeline = DiarizationPipeline(self.cfg, self.audio_src.uri)

        # # 7. Arrancar la lectura de chunks (emisión de stream)
        self._reader_thread = threading.Thread(
            target=self.audio_src.read,
            daemon=True
        )
        self._reader_thread.start()


        # 6. Arrancar inferencia (en background)
        self.pipeline.run_async(self.audio_src.src,
                                self.printer,
                                self.identifier)
```

Se puede observar cómo se crea `SpeakerDiarization` (con una configuración definida previamente en el archivo *config*) y un `AudioSourceChunk` que servirá para alimentar de audio. Luego se prepara un `StreamingInference` para ejecutar esa diarización y le añade los observadores: `PrintSegments()` para depuración y `RealTimeSpeakerID` para identificación. A este último, se le pasa, además del modelo de reconocimiento y los hablantes conocidos, una referencia (por ejemplo, un `callback on_target`) a `send_result`.

De esta forma cuando el observador detecte a Pablo, disparará a dicha función, señalizando al sistema que el fragmento en proceso contiene a ese hablante objetivo.

5.3 Fuente de Audio por Fragmentos (módulo *audiosourcechunk.py*).

Este componente es una abstracción creada para compatibilizar el flujo de audio del propio sistema de DiViVo con el pipeline implementado de diarización de la librería de DIART. En DIART, las fuentes de audio (`Diart.sources.AudioSource`) típicamente se encargan de gestionar la captura continua de los datos, ya sea desde micrófonos, archivos, etc. de forma interna, emitiendo segmentos a un *stream* reactivo. En este caso, dado que es DiViVo el que se controla la captura (y aplica VAD, etc.) se requiere una forma manual de inyectar manualmente los chunks de audio ya capturados. Se podría decir que `AudioSourceChunk` cumple este rol, actuando como un puente entre DiViVo y Diart.

Técnicamente, `AudioSourceChunk` hereda de la clase base de `AudioSource` de DIART o encapsula una instancia de `Diart.sources.ChunkAudioSource` si estuviese disponible). Implementa al menos dos métodos importantes:

- `add_chunk(chunk: np.ndarray)` : permite añadir un nuevo fragmento de audio (típicamente un array de muestras *numpy int16/float*) a la cola de procesamiento. Los chunks añadidos quedan en un búfer interno listos para ser consumidos por el pipeline de diarización.
- `read()` : método requerido por la interfaz de `AudioSource`, que inicia la emisión de audio hacia el pipeline. En nuestro diseño, `read()` extrae secuencialmente los chunks de la cola interna y los envía al *stream* reactivo del pipeline. Este método se ejecuta de forma bloqueante en un hilo separado (iniciado por `StreamingInference`) y se mantiene a la espera de nuevos datos. Cada vez que `add_chunk` inserta audio, `read()` lo toma y lo emite al pipeline en streaming. Cuando ya no se vayan a enviar más datos, se envía una señal de terminación (p. ej., un `None` o excepción para salir del bucle).

En código la clase queda así:

```
class ChunkAudioSource(AudioSource):
    """
    Fuente de audio basada en una cola de chunks externos.
    Para usarla, llama a `feed(chunk)` con un ndarray de forma (1, N)
    y luego arranca `read()` (en un hilo) para que empiece a emitir por
`stream`.
    """
    def __init__(self, uri: Text = "external_chunk_audio", sample_rate: int =
16000):
        super().__init__(uri, sample_rate)
        self._q = queue.Queue()
        self._closed = False

    def feed(self, chunk: np.ndarray):
        """Inserta un chunk (1×samples) en la cola."""
        #print(f"Inserting chunk...{self._q.qsize()} chunks in queue")
        self._q.put(chunk)

    def read(self):
        """Empieza a emitir todos los chunks insertados hasta que se cierre."""
        while True:
            chunk = self._q.get()
            #print("Stopped?",    self.stream.is_stopped)

            if chunk is None:
                print("READ IS CLOSING")
                break
            try:
                if self.stream.is_stopped:
                    print("RESTARTING STREAM")
                    self.stream = Subject()
                self.stream.on_next(chunk)
            except Exception as e:
                #self.stream.on_error(e)
                #print(f"READ IS CLOSING 2: {e}")
                pass
        # una vez salida del bucle:
        self.stream.on_completed()
        self.close()

    def close(self):
        """Marca la fuente como cerrada y desbloquea `read()` si está
esperando."""
        if not self._closed:
            self._closed = True
            # Para desbloquear un get() bloqueado
            self._q.put(None)
```

En el constructor, se inicializa la fuente con un URI identificador y la frecuencia de muestreo adecuada (16 kHz). La cola (`queue.Queue`) almacena los fragmentos pendientes de procesar. El método `read()` corre en segundo plano una vez que se inicia la inferencia en streaming; va extrayendo datos de la cola y usando `self.stream.on_next(chunk)` para entregarlos al pipeline de diarización. Cuando no hay más datos, sale del bucle y llama a `on_completed()` para indicar que ya no habrá más audio.

Es importante destacar que `AudioSourceChunk` no realiza procesamiento de audio como tal, su única función es adaptar la entrega de datos al formato esperado por DIART. Gracias a esto, se puede alimentar al diarizador con los mismos chunks que obtenemos tras el VAD, asegurando que los segmentos que son analizados son exactamente los mismos que fueron capturados. Además, se consigue que todo funcione de forma más fluida: el pipeline de diarización va recibiendo de la cola de audio conforme el sistema principal va produciendo nuevos fragmentos, manteniendo así el streaming en tiempo real.

5.4 Pipeline de Diarización en Streaming (módulo *pipeline.py*).

Este es el componente que hace efectiva la segmentación por hablante. Como se ha especificado en capítulos anteriores, para su implementación usamos la biblioteca DIART, que provee de pipeline preentrenado y facilidades para ejecutar inferencia de forma incremental (`StreamingInference`). En nuestro sistema, el pipeline se configura y lanza con las siguientes piezas:

- **Configuración del pipeline**: antes de iniciar la diarización, los parámetros específicos son definidos mediante `SpeakerDiarizationConfig`. Con los mismos parámetros y de igual manera que en la versión 6 del pipeline de streaming (apartado 4.6).
- **Inicialización de la inferencia en streaming**: con el pipeline creado y la fuente de audio `AudioSourceChunk` preparada (como se vio en la sección anterior), se configura la inferencia continua, de la siguiente manera:

```
 # Creamos un StreamingInference con los chunks de audio proporcionados
        inf = StreamingInference(self.pipeline, audio_src, do_plot=False, )

        # RTTMWriter y otros sinks
        rttm = RTTMWriter(uri=self.source_uri, path=f"{OUTPUT_DIR}/file.rttm",
patch_collar=PATCH_COLLAR)
        inf.attach_observers(rttm, *sinks) # Aquí van todos los sinks

        print("Starting inference…")
        inf()  # Este método empieza la inferencia y consume los chunks del
AudioSource
        print("Finished.")
```

En este código, se conecta la diarización con la fuente de audio. Con `attach_observers` añadimos los *sinks* u observadores que recibirán los resultados del pipeline en tiempo real. En este caso, se adjuntan dos:

- `PrintSegments()`: observador sencillo (subclase de *Observer*) que imprime en consola cada segmento de audio detectado junto con su etiqueta. Produce líneas como:

```
[PrintSegments] 0.00–2.34s → 0
[PrintSegments] 2.34–5.10s → 1
```

El código del método `on_next` de `PrintSegments` recorre las anotaciones del pipeline e imprime cada turno detectado.

- `RealTimeSpeakerID(…)`: es el observador crítico que se encarga de identificar y filtrar a Pablo. Este *sink* será detallado en la siguiente sección, aunque cabe destacar que se le pasa el modelo de reconocimiento (`spk_model`) y el diccionario de hablantes conocidos (`known_speakers`) que incluye la voz de Pablo.

Una vez que han sido adjuntados los observadores, se inicia esa inferencia continua. Tal y como se ha comprobado en capítulos anteriores, en una versión simplificada (*standalone*) se lograría simplemente invocando `inference()`, lo que bloquearía hasta que la fuente de audio terminase. En esta integración, la ejecución se gestiona en un hilo separado para no bloquear el bucle principal de captura. De este modo, el pipeline queda corriendo en segundo plano, escuchando los chunks que `AudioSourceChunk` va entregando, y generando anotaciones de diarización que son consumidas por ambos *sinks*.

Gracias a la configuración ajustada y a la infraestructura de *streaming*, el sistema puede reaccionar rápidamente a quién está hablando en cada momento. El agregado de observadores personalizados permite extender la funcionalidad básica de diarización (que solo separa voces) hacia la identificación de un hablante de interés, integrando así el filtro dentro del flujo de DiViVo.

5.5 Identificación del Hablante Objetivo (módulo realtime_speaker_id.py).

Este componente es un observador (*Observer* de DIART) diseñado para analizar cada segmento de habla detectado por el pipeline de diarización y determinar en tiempo real si corresponde a ese hablante objetivo. Se basa en técnicas de reconocimiento de locutor (*speaker recognition*), comparando características del fragmento a estudiar con vectores de referencia de los hablantes conocidos.

En esta sección se materializa la implementación de las técnicas claves desarrolladas en la versión final del capítulo anterior para la identificación basada en turnos.

5.5.1 Carga del modelo de identificación de hablante.

Para este proyecto se utilizó el modelo ECAPA-TDNN de *SpeechBrain*, obtenido a través de la clase `SpeakerRecognition` de dicha biblioteca. El código para cargar el modelo y preparar la referencia es:

```
spk_model: instancia de SpeechBrain SpeakerRecognition
known_speakers: dict con clave=nombre_hablante, valor=ruta absoluta al .wav
        """
        super().__init__()
        self.model = spk_model
        self.is_flushing = False
        if not known_speakers:
            raise ValueError("Se debe pasar `known_speakers` al constructor con
rutas absolutas")
        # Verificamos que existan los ficheros
        for name, ref in known_speakers.items():
            p = Path(ref)
            if not p.is_file():
```

```
                raise FileNotFoundError(f"Referencia no hallada para '{name}':
{ref}")
        self.known = known_speakers
        self.diarization_class = diarization_class
```

Tal y como se describió en el capítulo 4, para lograr una identificación correcta, se implementa un mecanismo de *buffering* temporal. El observador mantiene variables internas como `self.cur_generic` (identificador genérico asignado por el diarizador al hablante actual), `self.buf_audio` (lista de fragmentos acumulados del turno) y marcas de tiempo ya sean de inicio como de fin.

Cuando llega un nuevo segmento, se trata al igual que en el pipeline *standalone*, como resumen:

- Si pertenece al mismo hablante: sus muestras de audio se agregan al buffer.
- Si pertenece a hablante distinto (indica fin de turno anterior): se llama _flush(), lo que dispara el proceso de identificación del turno que acaba de terminar.

Para cada segmento:

- Se actualizan `cur_start`, `cur_end` y se comprueba si la duración acumulada excede el máximo (`MAX_UTTER_DUR`). Si se supera, se hace `_flush()` para evitar acumular turnos demasiado largos.

Paralelamente, se arma un temporizador de silencio. Cada vez que se añade un fragmento y actualiza `cur_end`, se programa (o reinicia) un *timer* que tras ciertos segundos de inactividad llamará a `_flush()` automáticamente.

Se observa, como al igual que en la versión *standalone,* se garantiza que cada intervención completa de un hablante (definida por silencio, larga duración o por cambio de hablante) sea enviada al proceso de verificación de identidad solo una vez, que está completa, evitando identificaciones parciales sobre fragmentos muy breves.

5.5.2 Verificación de identidad del hablante (método _flush()).

En este método ocurre lo siguiente:

1. Si no hay audio acumulado, es que posiblemente se llamó erróneamente, la función simplemente reinicia el estado y sale.
2. Calcula la duración total del turno acumulado (`total_dur`) en segundos. Si la duración es muy corta (por debajo del umbral `MIN_DUR`) se considera que ese fragmento es insuficiente para la identificación y se descarta. Esto se hace porque los fragmentos demasiado breves pueden ser poco confiables a la hora de la identificación. Se hace de la siguiente manera:

```
total_dur = sum(len(s) for s in self.buf_audio) / self.sr
  if total_dur < MIN_DUR:
      #print(f"[DEBUG] Duración {total_dur:.2f}s < MIN_DUR; ignoro")
      self.is_flushing = False
      return self.reset()
```

3. Si la duración es suficiente, concatena todas las porciones del audio en `buf_audio` para formar una sola señal continua (`utter`) correspondiente a la intervención completa.

```
utter = np.concatenate(self.buf_audio)
```

4. Guarda temporalmente esta señal concatenada en un archivo de audio WAV. Esto es necesario hacerlo porque la función de verificación (`verify_files`) espera rutas de archivo de entrada.

5. Para cada hablante conocido, se compara usando el modelo de `SpeakerRecognition` el audio del turno (*tmp.wav* que ha sido recién guardado) con el audio de referencia. En este caso, solo iterará por Pablo. La comparación produce un score numérico que representa la semejanza de voz (mayor score implica mayor similitud). El código recorre las entradas y guarda el mejor score y el nombre correspondiente.

```
best_score, best_name = float("-inf"), None
for name, ref in self.known.items():
    # Normalizar la ruta a POSIX para FFmpeg/torchaudio
    ref_posix = Path(ref).resolve().as_posix()
    print(f"[DEBUG] verify_files(tmp, {name}) usando {ref_posix!r}")
    try:
        sc, _ = self.model.verify_files(tmpfile.as_posix(), ref_posix)
        sc = float(sc)
        print(f"[DEBUG] verify_files → score={sc:.4f}")
    except Exception as e:
        print(f"[WARN] fallo verify_files para {name}: {e}")
        traceback.print_exc()
        continue
    if sc > best_score:
        best_score, best_name = sc, name
```

En este fragmento, `self.model.verify_files(A,B)` devuelve la puntuación de similitud entre el archivo A (el cual es la `utterance` del turno) y B (audio de referencia del hablante). Tras iterar, en `best_name` se contendrá el nombre del hablante conocido con mayor parecido a la referencia. Eso sí, si ningún score supera el umbral mínimo (`SCORE_THRESH`), se considera que el hablante es desconocido. Se procedería de la siguiente forma:

1. Se elimina el archivo WAV de la intervención para no acumular en disco.
2. Se determina la etiqueta final `label`: será el nombre del hablante reconocido con mayor puntuación si se superó el umbral. Si es conocido, `best_name == "Pablo"` y `best_score>=0.4`, con lo cual `label==Pablo`
3. Se imprime por consola el resultado indicando el `label`, la duración del turno y el score.

```
[Utterance] Pablo: 3.20s (10.50–13.70s) [score=0.85]
```

4. Finalmente, se restablece el estado interno `self.reset()`, limpiando el buffer de audio y preparando el objeto para procesar el siguiente turno.

El siguiente extracto de código muestra las partes de `_flush()` encargadas de lo anterior.

```
label = best_name if best_score >= SCORE_THRESH else "desconocido"
            st = self.cur_start or 0.0
            ed = self.cur_end   or (st + total_dur)
            print(f"[Utterance] {label}: {total_dur:.2f}s ({st:.2f}-{ed:.2f}s)
[score={best_score:.2f}]")

            print("label")
            print(label)
            if label == "pablo":
                self.diarization_class.send_result()
            else:
                self.diarization_class.reset_buffer()

            self.reset()
            print("[DEBUG] Completed _flush()")

            self.is_flushing = False
```

De esta manera, en el momento en que `_flush()` identifica positivamente a Pablo, se desencadena la llamada a `CDiarization.send_result()`, la cual, como se vio, establece un *flag* interno que hace que `process_chunk` retorne `True`. Así se cierra el ciclo de la toma de decisiones: el observador se encarga de revisar los segmentos, y cuando encuentra el objetivo en cuestión, avisa al módulo de filtrado para marcar ese audio como relevante.

En la implementación final, `RealTimeSpeakerID` se asegura de llamar a `send_result` una sola vez por cada intervención de Pablo, de esta forma se evita duplicar notificaciones si la intervención ocupó varios chunks. Tras esa notificación, el buffer se resetea y el sistema espera hasta una nueva intervención de Pablo.

Con este diseño, solo los turnos de Pablo generan una notificación, por lo que solo los chunks de las intervenciones de ese hablante son los que se encaminan a las siguientes etapas. Cualquier otro fragmento nunca activa `send_result` y, por tanto, es descartado por el sistema.

5.5.3 Integración y rendimiento en tiempo real.

Dado que este módulo opera en tiempo real, es importante comentar sobre la latencia y eficiencia. La verificación de hablantes ocurre al final de cada turno, por lo que la identificación añade un pequeño retraso (del orden de décimas de segundos) una vez que el hablante termina de hablar. Esto es aceptable en este caso de uso. Además, al utilizar un modelo optimizado como ECAPA-TDNN, la inferencia es rápida incluso con hardware modesto. El sistema de *buffering* y temporización garantiza que no se espere indefinidamente para decidir: a los 1.2 s de silencio del hablante, se fuerza la identificación. Esto significa que, en el peor de los casos, en mitad de una intervención de Pablo con pausas, podríamos partirla erróneamente; sin embargo, al ser Pablo un hablante objetivo único, incluso si se partiera, ambos fragmentos serían identificados como Pablo y enviados igualmente.

En pruebas informales, `RealTimeSpeakerID` identificó correctamente a Pablo en todos los segmentos donde hablaba, y nunca activó `send_result` para otros hablantes (que quedaron como "desconocido"), cumpliendo así la función de filtro deseada.

5.6 Flujo Completo: Conexión de todos los componentes.

Para recapitular, en esta sección se describe el flujo completo de audio a través del sistema integrando todos los componentes mencionados en este capítulo:

- El módulo de captura (*speech_microphone.py*) obtiene audio del micrófono en tiempo real, segmentándolo en chunks y filtrando silencio con VAD.
- Cada chunk con voz se envía al método `CDiarization.process_chunk`, donde es introducido en la cola de `AudioSourceChunk`. Simultáneamente, el pipeline de diarización está corriendo en streaming y consume ese audio, generando segmentaciones en vivo.
- Los observadores del pipeline reciben estas segmentaciones. `PrintSegments` imprime los segmentos y hablantes genéricos detectados. `RealTimeSpeakerID` acumula los fragmentos de audio que forman un turno de un hablante, y al terminar ese turno, compara ese audio con la voz de Pablo.
- Si el hablante de esa intervención era Pablo:
  - `RealTimeSpeakerID` activa `CDiarization.send_result`, lo que provoca que `CDiarization.process_chunk` retorne `True` al módulo de captura.
  - El módulo de captura llama a `CDiarization.extract_intervention_chunks` para obtener el audio de esa intervención de Pablo y lo reenvía a las siguientes etapas de DiViVo.
- Si el hablante no era Pablo:
  - Entonces, `send_result` no se activa, `process_chunk` devuelve `False` y el módulo de captura no envía nada, descartando ese audio. El pipeline de diarización sigue su curso, pero esos segmentos no avanzarán en el sistema.
- El buffer interno en `CDiarization` se limpia después de cada intervención procesada, para evitar mezclar el audio de distintos turnos.

Con este circuito, se logra un filtro de diarización en línea: ya que la salida consiste únicamente en la voz de Pablo (o del hablante del que se le añade la referencia), manteniendo el resto del audio (otros hablantes, ruido, silencio) fuera. La arquitectura modular permite integrar esta funcionalidad en DiViVo sin modificar drásticamente otros componentes; *speech_microphone.py* actúa como punto de unión, decidiendo en cada momento si envía audio adelante o no según la respuesta de `CDiarization`.

Se ha elaborado este diagrama en Ilustración 5.1 para poder observar de forma aproximada la relación entre los distintos módulos creados.

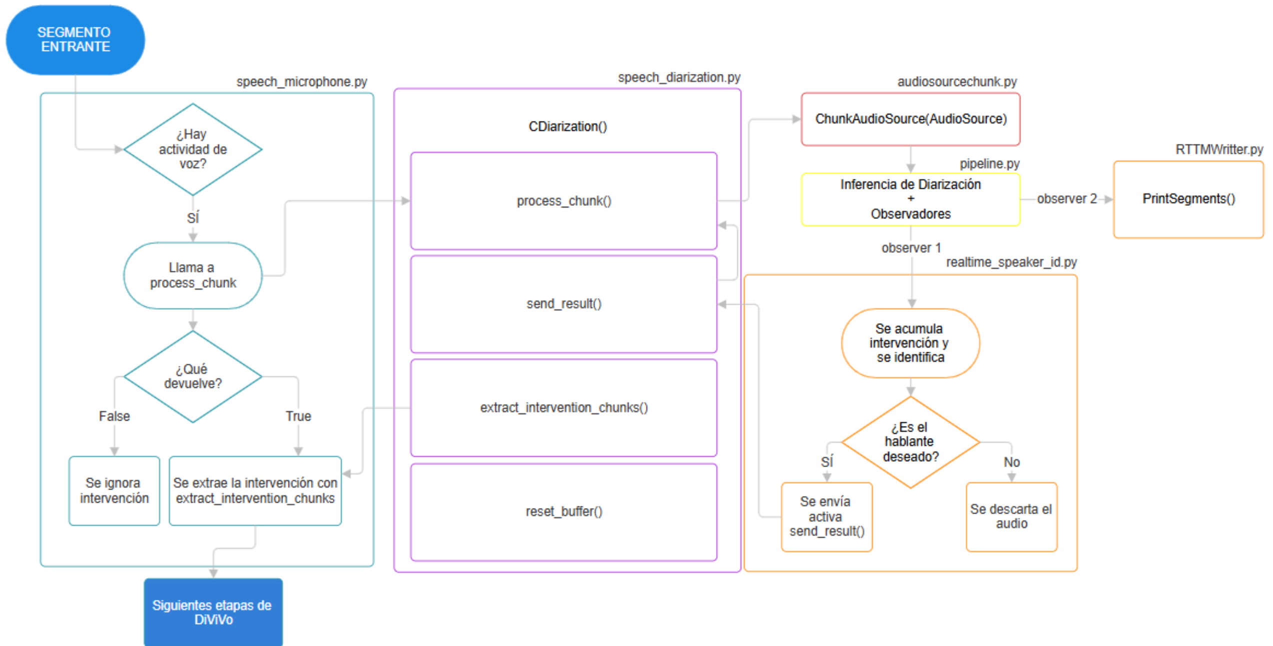

Ilustración 5.1- Flujo Completo: Conexión de los componentes.

En conclusión, se cumple el objetivo planeado de forma eficiente y escalable. Todas las partes trabajan sincronizadas para asegurar que, en el flujo final de audio, solo se escuche a quien deba escucharse.

6 CONCLUSIONES Y LÍNEAS DE DESARROLLO FUTURAS.

Si bien se ha cumplido con los objetivos propuestos inicialmente, durante su construcción surgieron varias ideas y oportunidades de mejora que quedan abiertas para el futuro. A continuación, se presentan algunas.

6.1 Trabajo Futuro.

Mejoras en la precisión y robustez de la diarización: Explorar algoritmos más avanzados o nuevas versiones de DIART que soporten situaciones de habla solapada (*speaker overlapping*) entre dos personas o entornos con elevado ruido de fondo elevado. Asimismo, ajustar dinámicamente esos parámetros como el `segmentation_threshold` o el `delta _new` según el contexto acústico de ese momento.

Optimización del rendimiento en tiempo real: Sería valioso reducir aún más la latencia y el uso de recursos para facilitar su despliegue en dispositivos con menores capacidades de cómputo como pueden ser robots, o sistemas embebidos. Implicaría simplificar modelos (usando modelos de identificación más livianos), paralelizar parte del pipeline con múltiples hilos o GPU, o incluso desarrollar una versión en C++.

Ampliación de la identificación a múltiples hablantes conocidos: Bastaría con implementar una base de datos más amplia de hablantes. También sería útil integrar algún mecanismo actualización o aprendizaje que permita añadir nuevos hablantes conocidos de forma semiautomática, alimentando el sistema con muestras de audios de nuevos hablantes que se vayan detectando.

Evaluación e implementación en entornos reales: Como paso futuro, se podrían llevar a cabo pruebas piloto del sistema en escenarios reales, como reuniones de trabajo, conferencias o entornos domésticos. Pruebas las cuales podrían proporcionar información valiosa sobre el desempeño del pipeline para refinar los parámetros (p. ej., umbrales de VAD, tiempos de collar y silencios).

Mejoras en la usabilidad del módulo: Desde la propia perspectiva de la empresa colaboradora, sería interesante convertir el pipeline desarrollado en un módulo independiente reutilizable, con una API definida, de modo que fuera aún más sencillo integrarse en las distintas soluciones que ofrece *4i Intelligent Insights* por parte de otros desarrolladores, sin necesidad de tener profundos conocimientos en diarización e identificación. También, podría contemplarse una interfaz visual o de logs más detallada para monitorear en vivo lo qué está haciendo el sistema.

6.2 Conclusión.

En este proyecto se ha podido implementar un sistema de diarización e identificación de hablantes en tiempo real con éxito. A lo largo del trabajo, se ha realizado un estudio comparativo de las dos herramientas de diarización, DIART y *PyAnnote,* concluyendo que DIART es la opción más adecuada para escenarios en tiempo real debido a su arquitectura optimizada de baja latencia y por brindar un pipeline modular y eficiente que permite diarizar sin interrupciones perceptibles.

Se elaboró un pipeline de procesamiento de audio dividido en tres fases: diarización, segmentación e identificación. Inicialmente, se implementó y probó este pipeline en modo offline sobre archivos de audios ya grabados. Lo que permitió validar la funcionalidad en un entorno controlado. En esta fase se aprendió la importancia de contemplar las métricas de diarización (DER, *False Alarm*, *Misses*, etc.) y cómo los parámetros de post-procesamiento (colas de silencio, duraciones mínimas de segmento, número estimado de hablantes, etc.) afectan la calidad de la separación de voces. Se comprobó que el modelo de identificación mediante *embeddings* de voz pre-entrenados (modelo ECAPA usado) es viable y preciso cuando se trabaja con segmentos que tienen suficiente duración y están claramente delimitados.

El procesamiento en tiempo real supuso un desafío técnico mayor. Ahora, el pipeline estaba alimentado por un micrófono en *streaming*. Por lo que fue necesario la gestión del buffer de audio y la temporización para la acumulación y fusión de segmentos pertenecientes a una intervención. Gracias a esto, se consiguió mejorar la fiabilidad de la identificación en tiempo real.

Otro aspecto técnico ha sido integrar en DiViVo Lite el filtro de diarización e identificación de hablantes conocidos. Gestionando el audio que llega en chunks, tal y como los proporciona el propio sistema y asegurando que el pipeline responda sin interrupciones. Se ha logrado mediante una lógica de observadores y considerando la importancia de sincronizar modelos asíncronos: mientras la diarización e identificación funcionaban en segundo plano, el sistema principal seguía capturando y transmitiendo audio sin bloquearse. Además, fue necesario contemplar el manejo de errores, evitar bloqueos de flujo y limpieza de buffers para no perder datos.

El contexto DiViVo, ha brindado la oportunidad de conocer sistemas de Reconocimiento Automático del Habla (ASR) para transcribir en texto lo dicho y con módulos de Procesamiento de Lenguaje Natural (NLP) para la comprensión de esas transcripciones.

Por último, al construir este pipeline de forma modular y reutilizable, se pudo integrar en el ecosistema DiViVo sin alterar su arquitectura base. Representando un aprendizaje sobre cómo trasladar la investigación y prototipos experimentales a entornos reales productivos.

7 REFERENCIAS

1. **Andalucía Aerospace Cluster.** Andalucía Aerospace. *4i Intelligent Insight: descripción del asociado.* [En línea] [Citado el: 18 de mayo de 2025.] https://andaluciaaerospace.com/asociados/4i-intelligent-insight/.

2. **Wu, Jian y al., et.** A Review of Speaker Diarization: Recent Advances with Deep Learning. *arXiv.* [En línea] 2021. [Citado el: 11 de marzo de 2025.] https://arxiv.org/abs/2101.09624. arXiv:2101.09624.

3. **Boll, S. F.** Suppression of acoustic noise in speech using spectral subtraction. *IEEE Transactions on Acoustics, Speech, and Signal Processing.* 1979, Vol. 27, 2, pp. 113–120.

4. *Speech enhancement based on a priori signal-to-noise ratio estimation.* **Scalart, P. y Filho, J. V. M.** s.l. : ICASSP'96 – IEEE International Conference on Acoustics, Speech, and Signal Processing, 1996, pp. 629–632.

5. *Speaker, environment and channel change detection and clustering via the Bayesian Information Criterion.* **Chen, S. S. y Gopalakrishnan, P. S.** s.l. : Proceedings of the DARPA Broadcast News Transcription and Understanding Workshop, 1998, pp.127–132.

6. **Anguera, X., y otros.** Speaker diarization: A review of recent research. *IEEE Transactions on Audio, Speech, and Language Processing.* 2012, Vol. 20, 2, pp. 356–370.

7. **Tranter, S. E. y Reynolds, D. A.** An overview of automatic speaker diarization systems. *IEEE Transactions on Audio, Speech, and Language Processing.* 2006, Vol. 14, 5, pp. 1557–1565.

8. *Improving speaker recognition performance in the domain adaptation challenge using deep neural networks.* **Garcia-Romero, D., y otros.** s.l. : IEEE, 2014, pp. 378–383.

9. **Ioffe, S.** Probabilistic linear discriminant analysis. *Computer Vision – ECCV 2006.* Springer : s.n., 2006, pp. 531–542.

10. *X-vectors: Robust DNN embeddings for speaker recognition.* **Snyder, D., y otros.** s.l. : 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2018, pp. 5329–5333.

11. *Time delay deep neural network-based universal background models for speaker recognition.* **Snyder, D., Garcia-Romero, D. y Povey, D.** s.l. : 2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), 2015, pp. 92–97.

12. *pyannote.audio 2.1 speaker diarization pipeline: principle, benchmark, and recipe.* **Bredin, Hervé.** Dublin, Irlanda : INTERSPEECH 2023, ISCA, 2023, pp 1983–1987. 10.21437/Interspeech.2023-105.

13. **Pyannote.** pyannote/speaker-diarization. *Hugging Face.* [En línea] [Citado el: 24 de octubre de 2025.] https://huggingface.co/pyannote/speaker-diarization.

14. **ReactiveX.** ReactiveX. [En línea] [Citado el: 24 de octubre de 2025.] https://reactivex.io/.

15. **Coria, J. M., y otros.** Diart: A Python Library for Real-Time Speaker Diarization. *Journal of Open Source Software.* 2024, Vol. 9, 99, pág. 5266.

16. **Aperdannier, R., Schacht, S. y Piazza, A.** Systematic Evaluation of Online Speaker Diarization Systems Regarding their Latency. *arXiv.* [En línea] 2024. [Citado el: 24 de octubre de 2025.] https://arxiv.org/abs/2407.04293. arXiv:2407.04293.

17. **Näreaho, M.** *Speaker diarization in challenging environments using deep networks: An evaluation of a state-of-the-art system.* Stockholm : KTH Royal Institute of Technology, 2023. Tesis de máster.

18. **Dataloop.** Spkrec Ecapa Voxceleb — model page. *Dataloop.ai.* [En línea] [Citado el: 24 de octubre de 2025.] https://dataloop.ai/library/model/speechbrain_spkrec-ecapa-voxceleb/.