CAPÍTULO V

AUDIO SINTÉTICO Y DEEPFAKES DE VOZ: DE LA INVESTIGACIÓN DOCTORAL A LAS HERRAMIENTAS DE DETECCIÓN

Todo lo aprendido sobre ingeniería de datos, análisis exploratorio y feature engineering converge en este capítulo. Lo he reservado para el cierre del libro porque posee un significado especial dentro de mi trayectoria profesional: en él se recorre una historia que no es únicamente académica, sino también profundamente personal. Este capítulo sintetiza más de quince años de trabajo investigativo, experiencias y reflexiones que deseo dejar plasmadas como culminación natural de este recorrido.

Figura 62. Línea de tiempo del audio sintético y mi trayectoria en el campo.

Me remonto al año 2010, cuando inicié mi tesis doctoral. En ese momento, la inteligencia artificial aún no había experimentado el auge de las redes neuronales convolucionales (CNNs), y mucho menos se hablaba de GANs o de Transformers. Mientras definía la temática en la que trabajaría durante los siguientes años de mi vida académica, surgió una idea que llevaba tiempo rondando mi cabeza: transmitir señales de voz dentro de otras señales de voz.

Así nació mi tesis doctoral como un trabajo en esteganografía de voz, un campo que en ese entonces competía principalmente con métodos de enmascaramiento que prometían alta imperceptibilidad, pero cuya capacidad real de ocultamiento era todavía limitada. En ese contexto, me encontré buscando una solución de enmascaramiento eficiente, la cual decidí orientar a través de la Transformada Wavelet Discreta (DWT, *Discrete Wavelet Transform*).

Y, como ocurre con frecuencia en la investigación -y en la vida misma-, mientras intentaba resolver un problema concreto, apareció aquello que terminaría convirtiéndose en uno de los aportes más representativos de mi tesis doctoral: la capacidad de camuflaje, o imitación, de las señales de voz. Descubrí que prácticamente cualquier señal de voz podía imitar a otra, incluso cuando pertenecían a géneros o idiomas diferentes (más adelante en mi investigación definí las condiciones que se debían cumplir para poder realziar la imitación). Fue un hallazgo especialmente relevante para la época, en la que, insisto, aún no había irrumpido AlexNet en la escena del aprendizaje profundo.

Mis primeros artículos, titulados *"Highly Transparent Steganography Model of Speech Signals Using Efficient Wavelet Masking"* (2012) y, pocos meses después, *"On the Ability of Adaptation of Speech Signals and Data Hiding"*, ambos publicados en la revista *Expert Systems with Applications*, marcaron un punto de inflexión en mi trabajo doctoral.

En el primero de estos trabajos hablé, por primera vez, de la capacidad de camuflaje de las señales de voz, comparándolas con un camaleón. Allí formulé la idea de que *"any speech secret message may seem similar to a speech host message if its wavelet coefficients are sorted"*, planteando que un mensaje de voz podía manipularse mediante un proceso de ordenamiento de sus coeficientes wavelet hasta llegar a sonar perceptualmente similar a otra señal de voz.

En el segundo artículo se definieron las condiciones que debían cumplir ambas señales —la original y la señal a imitar— para que este proceso de imitación fuese posible.

Sin ser plenamente consciente en ese momento, estaba explorando un concepto que años más tarde se volvería central con la llegada de las *Generative Adversarial Networks* (GANs), alrededor de 2017–2018: la generación de un audio a partir de otro audio por imitación. En mis primeros trabajos, este proceso se realizaba de forma determinística, a partir del ordenamiento y la adaptación de coeficientes wavelet para forzar que una señal de voz adquiriera las características perceptuales de otra. Con la irrupción de las GANs, este mismo principio pasó a implementarse de manera iterativa y aprendida, donde los modelos comenzaron a extraer y reproducir patrones espectrales complejos a partir de pares de audios. Este enfoque dio origen a técnicas modernas de *Voice-to-Voice* (V2V), en las que la inteligencia artificial aprende las características de una voz objetivo y permite generar nuevos audios a partir de señales de partida, marcando un punto de inflexión en la generación de contenido de voz sintético.

Unos años después, le di un giro a mi enfoque investigativo. Ya no se trataba de generar audio a partir de audio, sino de ser capaz de identificar si una señal de voz era natural o si, por el contrario, había sido generada artificialmente, ya fuera mediante técnicas clásicas de procesamiento de señales o a través de métodos de inteligencia artificial, como *Text-to-Speech* (TTS) o *Voice-to-Voice* (V2V). De alguna manera, comencé a "combatir" aquello que, desde otra perspectiva, yo misma había contribuido a crear.

En este contexto, y en el marco de un proyecto de alto impacto de la Universidad Militar Nueva Granada, titulado *"A video forensic solution for integrity assurance, object recognition and tampering detection"*, junto con mi equipo de trabajo exploramos diferentes estrategias basadas en aprendizaje de máquina y aprendizaje profundo para identificar si un audio correspondía a una señal de voz natural o si había sido generado por procesos de imitación o por modelos de inteligencia artificial. En particular, analizamos audios generados mediante *Deep Voice*, uno de los primeros sistemas de TTS completamente basados en aprendizaje profundo, presentado en 2017, el cual marcó un punto de inflexión en la generación de voz sintética al aprender directamente representaciones acústicas y prosódicas a partir de grandes volúmenes de datos.

Finalmente, en el artículo titulado *"Deep4SNet: Deep Learning for Fake Speech Classification"*, publicado en la revista *Expert Systems with Applications* en 2021, abordamos dos soluciones para la detección de voz sintética. La primera se basó en la extracción manual de patrones, utilizando estadísticas y medidas de entropía calculadas directamente sobre las señales de audio. La segunda consistió en transformar los audios en representaciones basadas en histogramas y emplear estas imágenes como entrada para entrenar un modelo de CNN de clasificación binaria.

Los resultados fueron contundentes: las soluciones basadas en extracción manual de características no permitían distinguir de forma confiable entre audios naturales y sintéticos. En contraste, la solución basada en CNN logró identificar ambas clases con una tasa de aciertos superior al 90%, tanto para los audios generados por imitación (mi método) como para aquellos generados mediante Deep Voice. No obstante, algunos años después de la publicación de *Deep4SNet*, se evidenció que el uso de histogramas como representación para la clasificación de audios naturales y sintéticos **no** es una solución robusta frente a ataques adversarios, tal como se expone en el artículo *"Audio-deepfake detection: Adversarial attacks and countermeasures"* (2024). Este trabajo puso de manifiesto que pequeñas perturbaciones cuidadosamente diseñadas pueden degradar de forma significativa el desempeño de este tipo de enfoques, incluso cuando las métricas iniciales parecen muy altas.

Mientras tanto, en el estado del arte comenzaron a consolidarse dos grandes líneas de trabajo. Por un lado, numerosos investigadores se enfocaron en construir modelos de clasificación cada vez más complejos, con arquitecturas a medida basadas en CNNs profundas o Transformers, optimizadas para conjuntos de datos específicos. Por otro lado, un grupo más reducido exploró el impacto de diferentes representaciones espectrales (como espectrogramas, escalogramas wavelet o Constant-Q Transform (CQT)) combinadas con transferencia de aprendizaje, utilizando modelos benchmark como ResNet para la tarea de clasificación.

Sin embargo, los resultados reportados en la literatura eran difícilmente comparables entre sí. Las evaluaciones se realizaban bajo protocolos heterogéneos: distintos datasets (muchos de ellos provenientes de *challenges* que rápidamente quedaban tecnológicamente rezagados), tamaños de

experimentos dispares, ausencia de pruebas externas y, en la mayoría de los casos, sin validación frente a audios generados con herramientas comerciales recientes. Esta falta de estandarización hacía complejo extraer conclusiones sólidas sobre el verdadero desempeño y la generalización de los modelos propuestos.

Por esta razón, y como cierre natural de más de quince años de investigación en la generación e identificación de audio sintético, junto con mi equipo de trabajo desarrollamos el proyecto "FakeVoiceFinder: Sistema de identificación de voz clonada para mitigar los riesgos del mal uso de la IA", financiado también al interior de la Universidad Militar Nueva Granada. El resultado fue un framework en Python, publicado como proyecto *open-source* en GitHub, diseñado para comparar modelos benchmark y modelos personalizados bajo condiciones homogéneas de entrenamiento, validación y evaluación.

Adicionalmente, se construyeron dos datasets de audio sintético: Fake Audio Dataset (ElevenLabs & Respeecher) y TTS/V2V Audio Deepfake Dataset. En el primero, la mayoría de los audios sintéticos corresponden a escenarios Voice-to-Voice (V2V) generados con las herramientas comerciales ElevenLabs y Respeecher. En el segundo, la mayoría de los audios fueron generados mediante Text-to-Speech (TTS), utilizando voces sintéticas derivadas del ecosistema Common Voice. Ambos datasets fueron diseñados para evaluar modelos en condiciones más cercanas a escenarios reales y actuales.

En este capítulo se presenta, de forma concisa, el concepto de imitación de voz a voz y se describen las características principales de los datasets construidos. Finalmente, se introduce el framework FakeVoiceFinder como una herramienta integradora orientada a aportar rigor, comparabilidad y reproducibilidad al estudio de la detección de deepfakes de voz.

5.1. Método de Imitación

El método de imitación es una técnica de conversión de voz basada en procesamiento digital de señales. Para llevar a cabo el proceso se requieren dos audios: uno denominado ***secret*** y otro denominado ***target***. El objetivo es ordenar los coeficientes temporo-frecuenciales del audio secret, es decir, su distribución ordenada, de tal forma que el resultado perceptual suene similar

al audio target, y obtener al mismo tiempo el mapeo de las posiciones de los coeficientes espectrales que existe entre ambos.

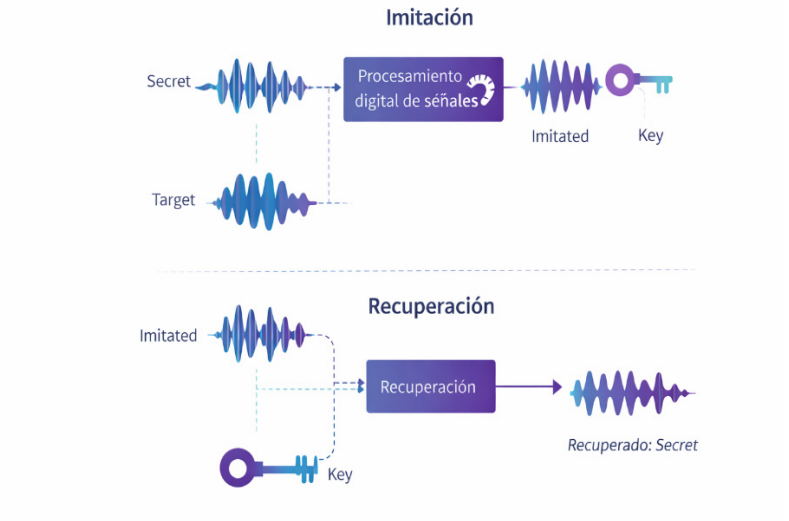

Figura 63. Diagrama del proceso de imitación y recuperación de voz.

El audio resultante de este proceso se denomina ***imitated***, mientras que el mapeo obtenido se conoce como ***key***.

La información transmitida corresponde al audio imitated y a su key, enviados a través de dos canales diferentes. De este modo, si un tercero escucha únicamente el audio ***imitated***, este sonará como el ***target***, pero no como el ***secret***. Solo quien posea la ***key*** podrá revertir el proceso de imitación y reconstruir el audio original.

Sí, suena un poco a una historia de espías… y en cierto sentido lo era. El propósito inicial de este método era precisamente transmitir información de voz de forma segura, garantizando confidencialidad a través de la manipulación estructural de la señal.

5.1.1. Código de Imitación

La implementación completa del método se encuentra disponible en el repositorio de GitHub, disponible en: https://github.com/doramariaballesteros/Imitation-using-Signal-Processing (ver Figura 64).

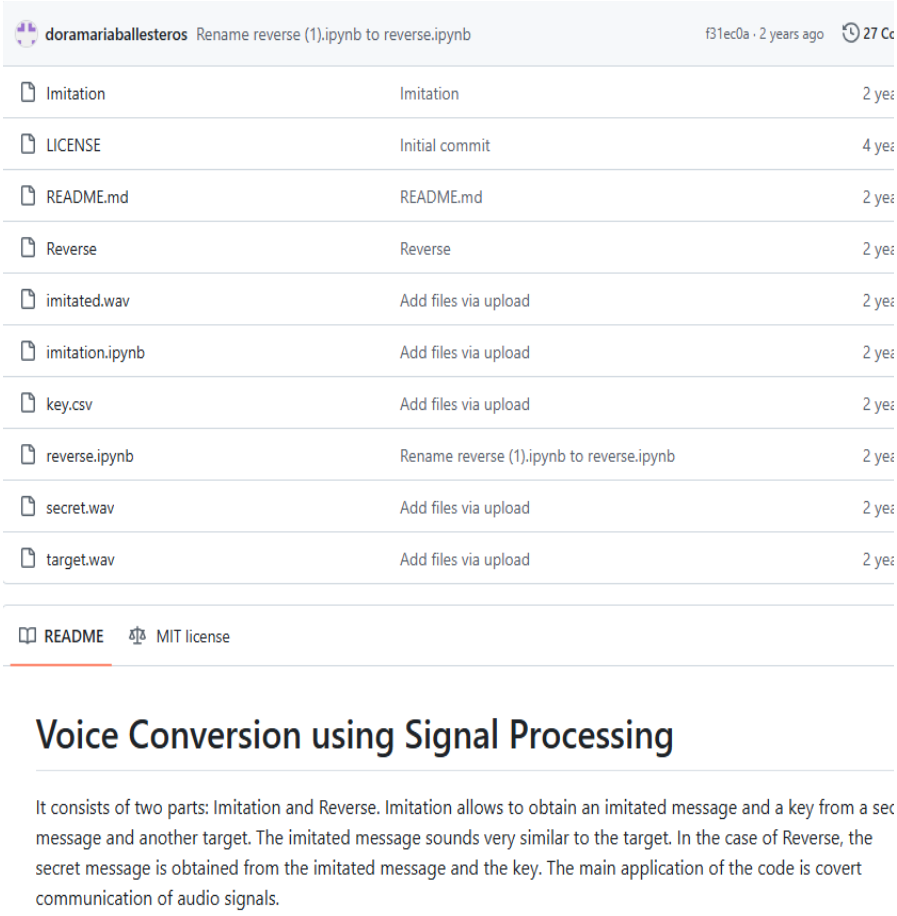

Figura 64. Vista general del repositorio GitHub que implementa el método de imitación y recuperación de voz mediante procesamiento digital de señales.

Vamos primero a trabajar con el archivo *imitation.ipynb* (descargarlo desde el repositorio). A manera de ejemplo, tenemos los audios secret.wav y target.wav, pero el lector podrá utilizar cualquier pareja de audios que cumplan con las siguientes condiciones:

1. La duración de los audios debe ser exactamente la misma.
2. La frecuencia de muestreo de los audios debe ser exactamente la misma
3. La relación de los tiempos de no silencio de los audios debe estar en un rango de [0.8 1.2]

En caso contrario, de forma previa el usuario tendrá que realizar recortes en la duración del audio mas largo, re-muestreo o cambiar alguno de los audios.

El audio secret.wav corresponde a un mensaje en inglés pronunciado por un hombre. Mientras que, el audio target.wav corresponde también a un mensaje en inglés (pero con otro plain text), pero pronunciado por una mujer. Ambos audios tienen una duración de 10 segundos, con una frecuencia de muestreo de 8 KHz.

Antes de realizar el proceso de imitación, vamos a visualizar las señales en el dominio del tiempo.

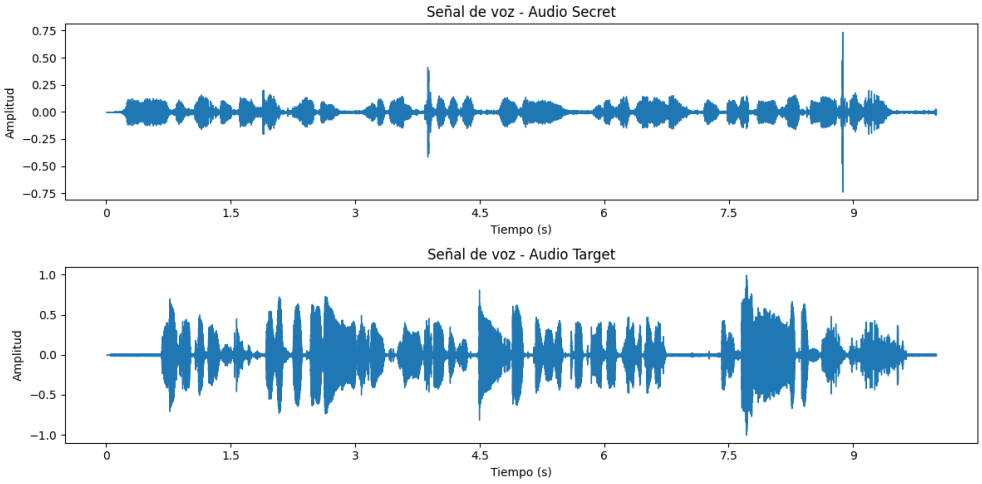

Figura 65. Visualización en el dominio del tiempo audio “secret” y “target”.

```
# Cargamos otras librerías
from scipy.io import wavfile
import IPython
import numpy as np
from numpy import savetxt
import pywt
from pywt import wavedec
```

```
# 1. Cargamos los audios y botón de display
source, sr1 = librosa.load('/content/secret.wav', sr=8000)
source=source/(np.max(abs(source)))
IPython.display.Audio(source, rate=sr1)
target, sr2 = librosa.load('/content/target.wav', sr=8000)
target=target/(np.max(abs(target)))
IPython.display.Audio(target, rate=sr2)
```

```
#2. Realizamos la descomposición con la DWT
csource = wavedec(source, 'sym4', level=2)
ctarget = wavedec(target, 'sym4', level=2)
csA2 = csource[0]
csD2 = csource[1]
csD1 = csource[2]
ctA2 = ctarget[0]
ctD2 = ctarget[1]
ctD1 = ctarget[2]
cs=np.concatenate((csA2, csD2, csD1), axis=0)
ct=np.concatenate((ctA2, ctD2, ctD1), axis=0)
```

```
#3. Aplicamos ordenamiento de coeficientes wavelet
# El ordenamiento permite transferir la estructura espectral del target
# preservando la energía global del secret
csource_sorted =np.sort(cs)
index1 = np.argsort(cs)
ctarget_sorted= np.sort(ct)
index2 = np.argsort(ct)
```

```
# 4. Re-ubicar los coeficientes wavelet de la señal “secret”
cs_m=np.zeros(len(cs))
cs_m[index2]=cs[index1]
l1 = len(csource[0])
l2 = len(csource[1])
l3 = len(csource[2])
csource_m=csource
csource_m[0] = cs_m[0:l1]
csource_m[1] = cs_m[l1:l1+l2]
csource_m[2] = cs_m[l1+l2:l1+l2+l3]

# Obtener la clave
key =np.zeros(len(cs))
key[index1]=index2
savetxt('key.csv', key, delimiter=',')
```

```
# 5. Applicamos reconstrucción wavelet a los coeficientes re-ubicados
 de “secret”

imitated = pywt.waverec(csource_m, 'sym4')
IPython.display.Audio(imitated, rate=sr1)
```

```
# 6.  Guardamos el audio resultante
import soundfile as sf
sf.write('imitated.wav', imitated, sr1, subtype='PCM_24')
```

Este código genera dos salidas: “imitated” y “key”. La primera corresponde al audio imitado, y la segunda a la clave, que es un archivo csv y guarda el mapeo entre las señales de entrada y nos permitirá en el código de *reverse.ipynb* recuperar el audio secreto.

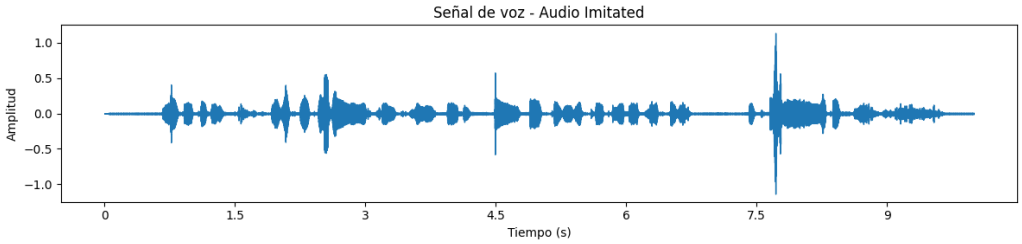

Figura 66. Visualización en el dominio del tiempo audio "imitated".

Si comparamos la Figura 65 (parte inferior, "target") con la Figura 66 ("imitated"), observamos que ambas señales son muy similares. Al escuchar el audio "imitated", este suena prácticamente igual que "target", no solamente en su contenido lingüístico, sino también conservando la entonación, el tono y el género del hablante femenino; por lo que no se sospecharía que ha sido editado. Este nivel de similitud perceptual es precisamente el tipo de comportamiento que hoy explotan los sistemas modernos de generación de audio sintético y deepfakes de voz.

5.1.2. Código de Recuperación (reverse)

En esta segunda parte, el propósito consiste en "revelar" el mensaje secreto oculto en el archivo imitated.wav. Recordemos que esta señal suena como target.wav, pero proviene originalmente de secret.wav. Para ello, necesitaremos dos entradas: imitated.wav y key.csv. Es importante aclarar que cada proceso de imitación genera una clave única, por lo que el uso de una clave diferente no permitirá recuperar el mensaje secreto original.

El código que se utilizará para este proceso también se encuentra disponible en el repositorio de GitHub y se denomina reverse.ipynb.

```
from scipy.io import wavfile
import IPython
import numpy as np
import pywt
import csv
from pywt import wavedec
from pywt import waverec
```

```
import librosa
import librosa.display
```

```
# 1. Cargar el audio y aplicar DWT
imitated, sr3 = librosa.load('/content/imitated.wav', sr=8000)
IPython.display.Audio(imitated, rate=sr3)
cImitated = wavedec(imitated, 'sym4', level=2)
cIA2 = cImitated[0]
cID2 = cImitated[1]
cID1 = cImitated[2]
cI=np.concatenate((cIA2, cID2, cID1), axis=0)
```

```
# 2. Cargar la clave
data_path = '/content/key.csv'
with open(data_path, 'r') as f:
    reader = csv.reader(f, delimiter=',')
    key = np.array(list(reader)).astype(float)
key = np.transpose(key)
key = np.reshape(key,(len(cI)))
key = key.astype(int)
```

```
# 3. Re-organizar los coeficientes a partir de la clave

cI=cI[key]
cIm = cImitated
l1 = len(cImitated[0])
l2 = len(cImitated[1])
l3 = len(cImitated[2])
cIm[0] = cI[0:l1]
cIm[1] = cI[l1:l1+l2]
cIm[2] = cI[l1+l2:l1+l2+l3]
```

```
# 4. Re-construir la señal y obtener el mensaje secreto revelado

reverse = pywt.waverec(cIm, 'sym4')
IPython.display.Audio(reverse, rate=sr3)
```

El audio reverse.wav obtenido corresponde al mismo mensaje contenido originalmente en secret.wav.

¿Qué es lo importante de este método y por qué hace parte de este capítulo? Porque utiliza un principio muy similar al empleado por los modelos generativos modernos, como las GANs, para la generación de voz a partir de voz: aprender los patrones del audio target para que el audio secret los imite. Estos patrones no incluyen únicamente el contenido lingüístico, sino también características prosódicas como la entonación, el timbre y el estilo del hablante.

En el caso de las GANs, el proceso de aprendizaje se realiza típicamente sobre representaciones espectrales, como espectrogramas o espectrogramas Mel. En contraste, en el método de imitación propuesto originalmente en 2012, este aprendizaje se lleva a cabo utilizando coeficientes espectrales obtenidos mediante la Transformada Wavelet Discreta (DWT).

5.2. Deep4SNet: identificación de audio sintético

Deep4SNet marcó un hito fundamental en mi trayectoria investigativa en el área de identificación de audio sintético. No solo por los altos niveles de desempeño obtenidos con los audios evaluados, sino porque permitió, por primera vez en mi línea de trabajo, evidenciar de manera clara qué enfoques resultaban prometedores y cuáles presentaban limitaciones para la detección de audio sintético generado tanto mediante técnicas clásicas de procesamiento digital de señales como mediante modelos basados en inteligencia artificial.

Esta investigación se desarrolló durante el año 2019 y fue publicada en 2021 en *Expert Systems with Applications*, una de las revistas más reconocidas a nivel mundial en el área de inteligencia artificial. Más allá del logro académico, este trabajo me dejó una de las lecciones más valiosas en investigación aplicada: ninguna solución es definitiva y todo modelo es susceptible de ser mejorado, cuestionado o superado.

Inicialmente, el problema de clasificación entre audios naturales y sintéticos lo abordamos como un problema de datos estructurados, empleando modelos clásicos de aprendizaje de máquina (Figura 67). Para ello, realizamos ingeniería manual de características basada, por un lado, en estadísticas globales del audio (como el promedio y la desviación estándar) y, por otro, en medidas de

entropía calculadas por segmentos. Sin embargo, en ambos casos observamos una correlación extremadamente baja entre las características extraídas y la salida del clasificador, lo que nos llevó a descartar este enfoque.

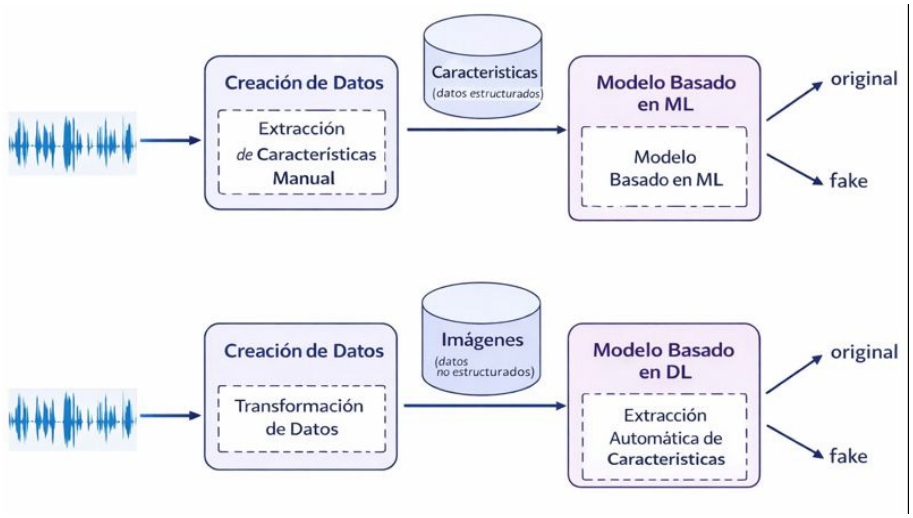

Figura 67. Enfoques dentro del proyecto Deep4SNet.

A partir de estos resultados, centramos nuestros esfuerzos en una estrategia distinta: la transformación del audio en representaciones gráficas mediante histogramas, con el objetivo de entrenar modelos de clasificación binaria basados en redes neuronales convolucionales 2D. Para ello, construimos un conjunto de imágenes a partir de audios naturales y sintéticos -provenientes tanto de un método propio de imitación como del sistema DeepVoice-, dataset que fue publicado en Mendeley bajo el título *"Fake voice histograms (Imitation + DeepVoice)"*.

Los resultados obtenidos con este enfoque fueron altamente satisfactorios en el contexto experimental considerado, alcanzando tasas de precisión y *recall* cercanas al 99 % para audios sintéticos por imitación, y superiores al 94 % para audios generados con DeepVoice. En ese momento, estos resultados evidenciaban claramente el potencial de las representaciones basadas en histogramas para la detección de audio sintético.

Sin embargo, como ocurre de manera natural en un campo tan dinámico como el de los deepfakes de voz, la rápida evolución de las técnicas

de generación de audio comenzó a poner en evidencia las limitaciones de este tipo de representaciones, particularmente en términos de robustez frente a perturbaciones no perceptuales y de capacidad de generalización a escenarios no vistos durante el entrenamiento.

Tres años después de la publicación de Deep4SNet, la revista ***Expert Systems with Applications*** publicó el artículo "Audio-deepfake detection: Adversarial attacks and countermeasures", en el cual se analizaron de forma sistemática las debilidades de los enfoques basados en histogramas frente a distintos tipos de ataques adversarios.

Lejos de representar un revés, este trabajo constituyó para mí un punto de inflexión conceptual. Que una investigación independiente, publicada en una de las revistas más relevantes del área, se dedicara a estudiar las limitaciones de un modelo desarrollado por mi equipo de trabajo, fue una confirmación del impacto y la visibilidad alcanzados por Deep4SNet. Los resultados mostraron que, ante perturbaciones relativamente simples -como la adición de ruido a los histogramas-, la tasa de acierto del clasificador podía degradarse de forma drástica.

Fue precisamente a partir de este análisis crítico que se encendió en mí una nueva idea sobre el camino a seguir en el ámbito de los audios *deepfake*: la construcción de un *framework* más flexible, robusto y sistemático, capaz de comparar múltiples representaciones y modelos bajo condiciones controladas, incluyendo escenarios con datos externos y manipulados. De esta necesidad nace **FakeVoiceFinder**, como una evolución natural y madura del camino iniciado con Deep4SNet.

5.3. FakeVoiceFinder: una librería para identificar audio sintético

Cierro este libro con mi investigación más reciente, desarrollada durante el año 2025, de la cual se derivó la publicación del artículo *"FakeVoiceFinder: An Open-Source Framework for Synthetic and Deepfake Audio Detection"* en la revista *Big Data and Cognitive Computing*. Este trabajo representa la consolidación natural del camino recorrido a lo largo de los capítulos anteriores y recoge, de manera estructurada, las lecciones aprendidas en más de una década de investigación en audio sintético y detección de *deepfakes* de voz.

FakeVoiceFinder es un marco de trabajo experimental que permite al usuario explorar, de forma sistemática y flexible, distintas estrategias para la clasificación de audios sintéticos. En particular, el *framework* ofrece las siguientes capacidades:

- **Experimentación centrada en el modelo, en los datos y de tipo híbrido**, facilitando el análisis comparativo de enfoques bajo un mismo escenario experimental para la clasificación de audio sintético a partir de representaciones espectrales.

- **Evaluación eficiente de un espacio de búsqueda**, en el cual el usuario puede seleccionar hasta cuatro tipos de representaciones espectrales (espectrograma en escala logarítmica, espectrograma en escala mel, escalograma y CQT) junto con diferentes arquitecturas *benchmark* de clasificación de imágenes (16 en total). Esto permite identificar, bajo las mismas condiciones de entrenamiento y validación, qué combinación arquitectura/representación espectral resulta más adecuada para un daraset específico.

- **Comparación justa entre soluciones personalizadas y arquitecturas de referencia**, tanto basadas en redes neuronales convolucionales como en Transformers, garantizando condiciones homogéneas de experimentación y evaluación.

- **Módulo de inferencia**, que permite evaluar si un audio externo es natural o sintético utilizando modelos entrenados dentro del framework o modelos externos compatibles, siempre que cumplan con el mismo tipo de entrada y formato de datos.

A nivel de arquitecturas *benchmark*, FakeVoiceFinder incorpora modelos secuenciales -como la familia VGG-, arquitecturas residuales -como ResNet-, modelos de múltiples ramas -como Inception, arquitecturas ligeras -como MobileNet-, enfoques más modernos -como ConvNeXt- y modelos basados en Transformers -como Vision Transformer (ViT)-. De esta manera, el usuario puede identificar inicialmente qué tipo de arquitectura resulta más adecuada para la detección de audios sintéticos generados por una herramienta específica, de acuerdo con el dataset utilizado. Esta información se convierte

en una guía valiosa al momento de diseñar arquitecturas personalizadas o de adaptar modelos existentes a nuevos escenarios de detección.

5.3.1. Datasets: Fake Audio Dataset (ElevenLabs & Respeecher) y TTS/V2V Audio Deepfake Dataset

Dentro del proyecto de investigación FakeVoiceFinder se construyeron dos conjuntos de audios sintéticos generados mediante herramientas de inteligencia artificial. Ambos datasets fueron publicados en Mendeley Data bajo los nombres *Fake Audio Dataset (ElevenLabs & Respeecher)* y *TTS/V2V Audio Deepfake Dataset*.

La motivación para desarrollar dos conjuntos independientes fue garantizar que presentaran características diferenciadas, particularmente en aspectos como la proporción de audios generados mediante Voice-to-Voice (V2V) y Text-to-Speech (TTS), así como en las herramientas empleadas durante el proceso de generación. Esta diversidad permite evaluar los modelos de detección bajo escenarios más realistas y heterogéneos.

A continuación, se presenta un resumen de los principales metadatos de ambos conjuntos de datos.

Tabla 6. Metadatos de los datasets Fake Audio Dataset (ElevenLabs & Respeecher) y TTS/V2V Audio Deepfake Dataset, incluyendo herramienta de generación, tipo de síntesis y tamaño del conjunto.

Dataset	Herramienta	V2V	TTS	Total	Duración (s)
Fake Audio Dataset (ElevenLabs & Respeecher)	ElevenLabs Respeecher	492	108	600	8 - 10
TTS/V2V Audio Deepfake Dataset	Minimax	40	603	643	8 - 10

De esta manera, uno de los datasets se utilizó para el entrenamiento y validación de los modelos de clasificación de audios naturales y sintéticos (específicamente Fake Audio Dataset), mientras que el otro se reservó para la prueba externa (es decir, TTS/V2V Audio Deepfake). Esta estrategia permitió no solo analizar el impacto de la proporción de audios generados mediante V2V y TTS en la capacidad de detección de cada tecnología, sino también evaluar la capacidad de generalización de los modelos cuando la inferencia se realiza sobre audios generados con herramientas distintas a las empleadas durante el entrenamiento. Estas herramientas pueden dejar diferentes “rastros” o huellas en el audio sintético, lo que aproxima la fase experimental a un entorno de evaluación más realista.

5.3.2. Experimento básico dentro de FakeVoiceFinder

FakeVoiceFinder está diseñado para facilitar el trabajo del investigador que desea realizar múltiples experimentos, combinando diferentes **arquitecturas**, **tipos de transformaciones de datos**, esquemas de **transferencia de aprendizaje** o **entrenamiento desde cero (scratch)** y, si se desea, compararlos con modelos desarrollados a medida.

Dentro del framework, publicado en GitHub en https://github.com/DEEP-CGPS/FakeVoiceFinder, se dispone de una carpeta denominada notebooks, y específicamente para el experimento básico se encuentra el archivo 3-Experiment_All_models_and_transforms.ipynb, el cual puede descargarse y ejecutarse directamente en Google Colaboratory**.**

A continuación, se describen brevemente las secciones principales de este notebook.

Sección 1: Environment & Imports.

En esta sección se cargan las librerías de trabajo y se importan, desde el framework, las funciones necesarias para la ejecución del experimento.

Se debe tener disponibles dos archivos zip, uno correspondiente a los audios naturales y el otro a los audios sintéticos, con los nombres real.zip y fake.zip, respectivamente.

Sección 2: Experiment Configuration.

Aquí se configuran los hiperparámetros**,** tanto de los datos como de las arquitecturas utilizadas. Para lectores no familiarizados con este concepto, en las Secciones 1.4.1 **y** 1.4.2 se presenta una explicación detallada de los hiperparámetros en cada contexto.

Dentro de *FakeVoiceFinder* es posible ajustar el tipo de representación espectral**,** seleccionando una, dos, tres o las cuatro opciones disponibles:

```
# Transforms to generate
cfg.transform_list = ["mel","dwt","log","cqt"]
```

Para cada representación se definen hiperparámetros específicos, como se describe a continuación:

Espectrograma en escala mel (mel):

```
cfg.mel_params = {
    "n_mels": 68,
    "n_fft": 2048,
    "hop_length": 512,
    # "win_length": None,
    # "fmin": 0,
    # "fmax": None,
    }
```

En este caso, n_mels corresponde al número de bandas en la escala mel; n_fft define el tamaño de la ventana de la FFT, y hop_length representa el desplazamiento, en número de muestras, entre ventanas consecutivas del análisis espectral. La superposición entre ventanas se obtiene como la diferencia entre el tamaño de la ventana (n_fft) y el valor de hop_length.

Espectrograma en escala logarítmica (log):

```
cfg.log_params = {
    "n_fft": 2048,
    "hop_length": 256,
    # "win_length": None,
```

```
}
```

En esta representación se utilizan los hiperparámetros n_fft y hop_length, cuya interpretación es análoga a la descrita en el caso del espectrograma en escala mel.

Escalograma (dwt):

```
cfg.dwt_params = {
    "wavelet": "db6",
    "level": 5,
    "mode": "symmetric",
}
```

Esta representación espectral se basa en la **Transformada Wavelet Discreta (DWT)**, en lugar de la **STFT** empleada en los casos anteriores. Los hiperparámetros asociados son: wavelet, que define la base wavelet utilizada (incluyendo la familia y el tamaño de sus filtros); level, que indica la cantidad de niveles de descomposición; y mode, que controla el **método de extensión de la señal en los bordes** durante el cálculo del escalograma.

La función de mode es gestionar la falta de muestras cuando la wavelet se aplica cerca del inicio o del final de la señal, influyendo principalmente en la estimación de energía en las regiones extremas del dominio temporal. Entre las opciones más comunes se encuentran: constant (relleno con ceros), reflect (reflexión de la señal en los bordes), symmetric (simetría estricta con bordes suaves), periodic (asume periodicidad de la señal) y nearest (repetición del último valor). Para aplicaciones de audio, se recomienda utilizar los modos reflect o symmetric.

Constant-Q Transform (cqt):

```
cfg.cqt_params = {
    "hop_length": 256,      # good time–frequency tradeoff
    "n_bins": 96,           # recommended range ~84–120
    "bins_per_octave": 24,  # 12 or 24 → 24 gives more detail on formants
    "scale": True,          # more stable spectral distribution
    # "fmin": 32.70319566,  # C1 (this is the default in the code)
    # "fmin": 65.40639133,  # C2 if you want to shift focus to vocal range
}
```

La Transformada Q Constante (CQT) se diferencia de la STFT en que sus bandas de frecuencia están espaciadas logarítmicamente, imitando la percepción auditiva humana. En este contexto, los parámetros bins_per_octave y scale controlan cómo se discretiza y normaliza el eje frecuencial. El parámetro bins_per_octave define el número de bandas de frecuencia por octava, controlando la resolución frecuencial del análisis. Por su parte, el parámetro scale determina si los coeficientes se normalizan para compensar la diferente duración temporal de las ventanas asociadas a cada banda, permitiendo una distribución más equilibrada de la energía a lo largo del espectro.

Por otro lado, en términos de la arquitectura, tenemos los siguientes hiperparámetros.

Arquitectura:

```
cfg.models_list = [
    “alexnet”,
    “resnet18”,
    “resnet34”,
    “resnet50”,
    “vgg16”,
    “vgg19”,
    “densenet121”,
    “mobilenet_v2”,
    “efficientnet_b0”,
    “squeezenet1_0”,
    “vit_b_16”,
    “googlenet”,
    “inception_v3”,
    “convnext_tiny”,
    “convnext_small”,
    “convnext_base”
]
```

FakeVoiceFinder dispone de **16 arquitecturas** predefinidas. No obstante, el usuario puede seleccionar libremente cuáles desea entrenar y evaluar, simplemente eliminando del listado aquellas que no requiera.

Hiperparámetros de entrenamiento:

```
cfg.type_train = “both”   # ‘scratch’ | ‘pretrain’ | ‘both’
cfg.epochs = 10
cfg.batch_size = 8
cfg.learning_rate = 0.0001
cfg.patience = 5
```

El parámetro cfg.type_train define el esquema de entrenamiento: transferencia de aprendizaje (pretrain), entrenamiento desde cero (scratch) o la evaluación de ambas estrategias (both). En este último caso, se entrenan dos modelos por cada combinación de arquitectura y representación espectral: uno inicializado desde cero y otro inicializado con pesos preentrenados.

Por otro lado, cfg.epochs corresponde al número de épocas de entrenamiento; cfg.batch_size define el tamaño del lote (se recomiendan valores de hasta 64); cfg.learning_rate representa la tasa de aprendizaje (con valores típicos entre 0.0001 y 0.001); y cfg.patience indica el número de épocas consecutivas sin mejora en la métrica de validación antes de aplicar un parado anticipado (early stopping).

Esta configuración permite comparar, bajo condiciones controladas, el efecto de la arquitectura, la representación espectral y el esquema de entrenamiento sobre el desempeño de detección.

Sección 3: Clip length.

```
# Select the audio window (clip_seconds)
min_sec = int(shortest_audio_seconds(cfg))
print(f”Duración mínima detectada en los zips: {min_sec}”)

# Option A: use exactly the minimum detected
cfg.clip_seconds = min_sec

# Option B: use a fixed value of your choice (e.g., 3.0 s)
# cfg.clip_seconds = 3.0

# Note: if you set a value greater than many audio files, it will be filled with
padding (as the pipeline already does).
```

En esta sección se ajusta la longitud de los audios, ya que todas las representaciones espectrales deben generarse bajo condiciones homogéneas para evitar sesgos en los datos de entrada a los modelos. Esto evita introducir sesgo por padding excesivo, así como garantiza que todas las representaciones espectrales se generen bajo la misma ventana temporal.

La selección entre la opción A o la opción B se realiza simplemente comentando la alternativa que no se desea utilizar. Tal como se muestra en el código, se encuentra activa la opción A, que establece la duración del clip igual a la longitud mínima detectada en el conjunto de audios.

Sección 4: Create Experiment Layout.

Inicializar el experimento. Se definen los metadatos del experimento.

Sección 5: Prepare Dataset.

En esta sección se realiza la partición del dataset en conjuntos de entrenamiento y validación. Adicionalmente, se generan las transformaciones espectrales definidas previamente, de acuerdo con la configuración del experimento.

```
summary = exp.prepare_data(train_ratio=0.8, seed=cfg.seed, transforms=cfg.
transform_list)
print("Data prep summary:")
pprint(summary)

print("Manifest:", (exp.root / "experiment.json").as_posix())
```

Por defecto, el dataset se divide utilizando una proporción del 80 % para entrenamiento y el 20 % restante para validación. Si se desea modificar esta relación, basta con ajustar el valor del parámetro train_ratio.

Como resultado de este proceso, el framework genera un resumen con la distribución de los datos por clase y por partición, así como el número de muestras asociadas a cada representación espectral seleccionada. Además, se crea un archivo de configuración (experiment.json) que actúa como manifiesto del experimento, almacenando de forma reproducible los parámetros utilizados.

```
Data prep summary:
{'load': {'fake': 600, 'real': 600},
 'save_original': {'test': 240, 'train': 960},
 'split': {'test': {'fake': 120, 'real': 120, 'total': 240},
           'train': {'fake': 480, 'real': 480, 'total': 960}},
 'transforms': {'cqt': {'test': 240, 'train': 960},
                'dwt': {'test': 240, 'train': 960},
                'log': {'test': 240, 'train': 960},
                'mel': {'test': 240, 'train': 960}}}
Manifest: D:/FakeVoiceFinder/outputs/exp_all_v1/experiment.json
```

Figura 68. Ejemplo de resultado de la Sección 5 de FakeVoiceFinder.

Al utilizar en este ejemplo el dataset Fake Audio Dataset (ElevenLabs & Respeecher), que contiene 600 audios sintéticos, junto con un conjunto de 600 audios naturales, y aplicar una partición del 80 % para entrenamiento, se obtienen 480 audios por clase para entrenamiento y 120 audios por clase para validación (o test interno). Esto da como resultado un total de 960 audios destinados al entrenamiento y 240 a la validación. Estos valores se mantienen de forma consistente para cada una de las representaciones espectrales seleccionadas en la Sección 2.

Sección 6: Prepare Models.

```
loader = ModelLoader(exp)
bench = loader.prepare_benchmarks(add_softmax=False, input_
channels=getattr(cfg, “input_channels”, 1))
print(“Benchmarks saved under models/loaded:”)
pprint(bench)

# User models (if any .pt/.pth under cfg.models_path)
user = loader.prepare_user_models(add_softmax=False, input_channels=cfg.
input_channels)
print(“User models saved:”)
pprint(user)
```

Inicialmente, mediante la clase ModelLoader, el framework carga y prepara las arquitecturas predefinidas incluidas en FakeVoiceFinder, manteniendo la salida binaria en forma de logits y sin aplicar la capa softmax durante el entrenamiento. Posteriormente, se identifican y cargan de manera automática los modelos personalizados del usuario, almacenados en formato .pt o .pth en la ruta definida por cfg.models_path, los cuales se integran al mismo flujo experimental y se evalúan bajo condiciones homogéneas.

Sección 7: Sanity Checks.

```
def print_tree(root: Path, max_depth: int = 3, prefix: str = ""):
    if max_depth < 0:
        return
    try:
        entries = sorted(root.iterdir(), key=lambda p: (p.is_file(), p.name.lower()))
    except FileNotFoundError:
        return
    for e in entries:
        print(prefix + ("📄 " if e.is_file() else "📂 ") + e.name)
        if e.is_dir():
            print_tree(e, max_depth - 1, prefix + "    ")

print_tree(exp.root, max_depth=3)
```

En esta sección se utiliza una función auxiliar para visualizar la estructura de directorios generada por el experimento. La función print_tree recorre de forma recursiva la carpeta raíz del experimento (exp.root) y muestra en consola los archivos y subdirectorios hasta una profundidad máxima definida por el parámetro max_depth. Esta visualización permite al investigador verificar de manera rápida y ordenada la organización de los datos, modelos y resultados producidos durante la ejecución del pipeline.

Podemos observar que se generan tres carpetas principales: datasets, models y reports.

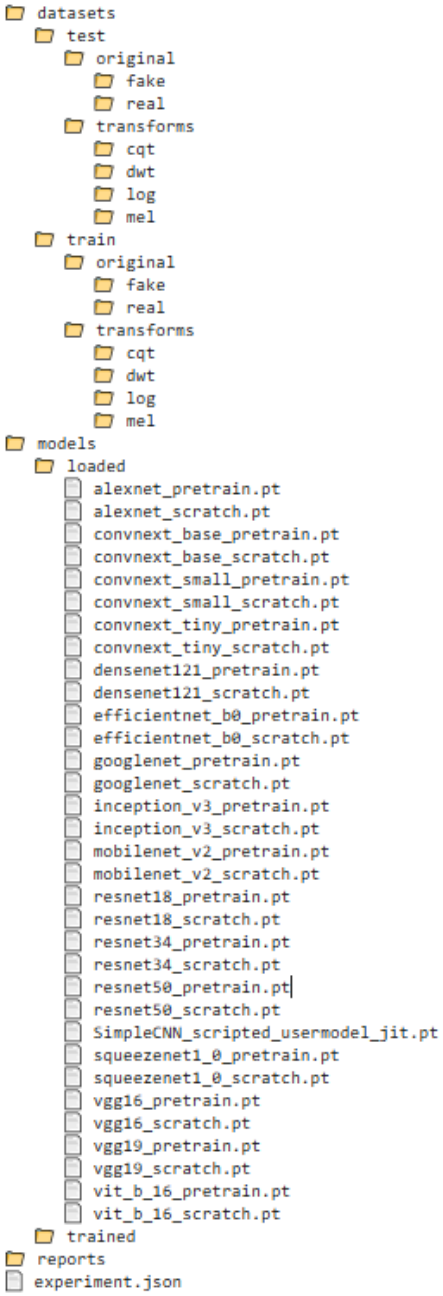

Figura 69. Ejemplo de resultado de la Sección 7 de FakeVoiceFinder.

Sección 8: Train & Evaluate.

```
exp.loaded_models
trainer = Trainer(exp)
train_results = trainer.train_all()
print("Training results (repo-relative paths):")
pprint(train_results)

print("Best checkpoints stored in:", (exp.trained_models).as_posix())
```

Cuando se desea ejecutar el experimento de forma completa, es decir, cuando no se cuenta con modelos previamente entrenados, es necesario ejecutar esta sección. En caso contrario, si únicamente se busca comparar el desempeño de modelos ya entrenados, se puede avanzar directamente a la siguiente sección.

Cada una de las arquitecturas definidas en la Sección 2 se entrena utilizando cada una de las representaciones espectrales seleccionadas en esa misma sección. De este modo, todos los modelos quedan entrenados bajo condiciones experimentales homogéneas, lo que permite realizar comparaciones justas y consistentes entre arquitecturas y tipos de representación espectral.

```
[densenet121][mel][pretrain] Start training for 10 epochs
[densenet121][mel][pretrain] Epoch 1/10 - loss=0.2609 acc=0.9792
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 234, FP=   6],
 [FN=   4, TP= 236]]
[densenet121][mel][pretrain] ✅ New best acc=0.9792 at epoch 1
[densenet121][mel][pretrain] Epoch 2/10 - loss=0.0843 acc=0.9833
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 236, FP=   4],
 [FN=   4, TP= 236]]
[densenet121][mel][pretrain] ✅ New best acc=0.9833 at epoch 2
[densenet121][mel][pretrain] Epoch 3/10 - loss=0.0391 acc=0.8667
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 178, FP=  62],
 [FN=   2, TP= 238]]
[densenet121][mel][pretrain] Epoch 4/10 - loss=0.0598 acc=0.9875
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 234, FP=   6],
 [FN=   0, TP= 240]]
[densenet121][mel][pretrain] ✅ New best acc=0.9875 at epoch 4
[densenet121][mel][pretrain] Epoch 5/10 - loss=0.0197 acc=0.9917
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 238, FP=   2],
 [FN=   2, TP= 238]]
[densenet121][mel][pretrain] ✅ New best acc=0.9917 at epoch 5
[densenet121][mel][pretrain] Epoch 6/10 - loss=0.0646 acc=0.9958
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 238, FP=   2],
 [FN=   0, TP= 240]]
[densenet121][mel][pretrain] ✅ New best acc=0.9958 at epoch 6
[densenet121][mel][pretrain] Epoch 7/10 - loss=0.0174 acc=1.0000
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 240, FP=   0],
 [FN=   0, TP= 240]]
[densenet121][mel][pretrain] ✅ New best acc=1.0000 at epoch 7
[densenet121][mel][pretrain] Epoch 8/10 - loss=0.0680 acc=1.0000
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 240, FP=   0],
 [FN=   0, TP= 240]]
[densenet121][mel][pretrain] Epoch 9/10 - loss=0.0174 acc=1.0000
[densenet121][mel][pretrain] Confusion matrix (test):
[[TN= 240, FP=   0],
 [FN=   0, TP= 240]]
```

Figura 70. Ejemplo de resultado parcial de la Sección 8 de FakeVoiceFinder.

Por ejemplo, la arquitectura densenet121 con transferencia de aprendizaje (pretrain) y representación espectral en escala mel se entrena a lo largo de las 10 épocas definidas en la Sección 2. Cada vez que se obtiene una mejora en el valor de accuracy, el modelo se almacena como un nuevo checkpoint. Adicionalmente, en cada época se puede visualizar la matriz de confusión correspondiente, lo que permite hacer un seguimiento detallado del proceso de entrenamiento.

Sección 9: Metrics & Plots

```
from fakevoicefinder.config import ExperimentConfig
from fakevoicefinder.experiment import CreateExperiment
from fakevoicefinder.metrics import MetricsReporter

EXP_NAME = cfg.run_name

cfg = ExperimentConfig(); cfg.run_name = EXP_NAME
exp = CreateExperiment(cfg, experiment_name=cfg.run_name)

rep = MetricsReporter(exp)                # toma reports/ del manifest
df = rep.evaluate_all("metrics_summary.csv") # guarda CSV en outputs/<exp>/reports/
df
```

Obtenemos una previsualización de los resultados numéricos de los modelos entrenados. La información queda almacenada en un dataframe con el nombre del modelo, la variane, el tipo de representación espectral utilizada, el accuracy, el f1, el f1_macro, el f1_micro, la precisión, el recall y el check.

	model	variant	transform	accuracy	f1	f1_macro	f1_micro	precision	recall	check
0	alexnet	pretrain	cqt	92.50	91.96	92.47	92.50	99.04	85.83	outputs/exp_all_v1/models/trained/alexnet_
1	alexnet	scratch	cqt	94.17	94.31	94.16	94.17	92.06	96.67	outputs/exp_all_v1/models/trained/alexnet_
2	convnext_base	pretrain	cqt	97.92	97.94	97.92	97.92	96.75	99.17	outputs/exp_all_v1/models/trained/convnext_
3	convnext_base	scratch	cqt	78.75	79.35	78.73	78.75	77.17	81.67	outputs/exp_all_v1/models/trained/convnext_
4	convnext_small	pretrain	cqt	93.75	93.39	93.73	93.75	99.07	88.33	outputs/exp_all_v1/models/trained/convnext_
...	...	...	...	...	...	...	...	...	...	
127	vgg16	scratch	mel	94.17	94.12	94.17	94.17	94.92	93.33	outputs/exp_all_v1/models/trained/vgg16_sc
128	vgg19	pretrain	mel	96.67	96.64	96.67	96.67	97.46	95.83	outputs/exp_all_v1/models/trained/vgg19_pr
129	vgg19	scratch	mel	50.00	33.33	33.33	50.00	0.00	0.00	outputs/exp_all_v1/models/trained/vgg19_sc
130	vit_b_16	pretrain	mel	50.00	33.33	33.33	50.00	0.00	0.00	outputs/exp_all_v1/models/trained/vit_b_16_
131	vit_b_16	scratch	mel	85.83	83.65	85.58	85.83	98.86	72.50	outputs/exp_all_v1/models/trained/vit_b_16

132 rows × 10 columns

Figura 71. Ejemplo de resultado de dataframe con métricas de desempeño de FakeVoiceFinder.

Posteriormente, podemos visualizar todos los modelos generados con un mismo tipo de representación.

```
        # Figuras (se guardan en 'reports/' al pasar out_name)
        rep.plot_architectures_for_transform(df, transform="mel",
metric="accuracy",
                                             y_min=0, y_max=100, out_name="fig_arch_
mel_acc.png"
```

Para este caso, hemos escogido la representación espectral “mel” (recordemos que FakeVoiceFinder soporta “log”, “mel”, “dwt” y “cqt”), cuya figura de salida se presenta a continuación:

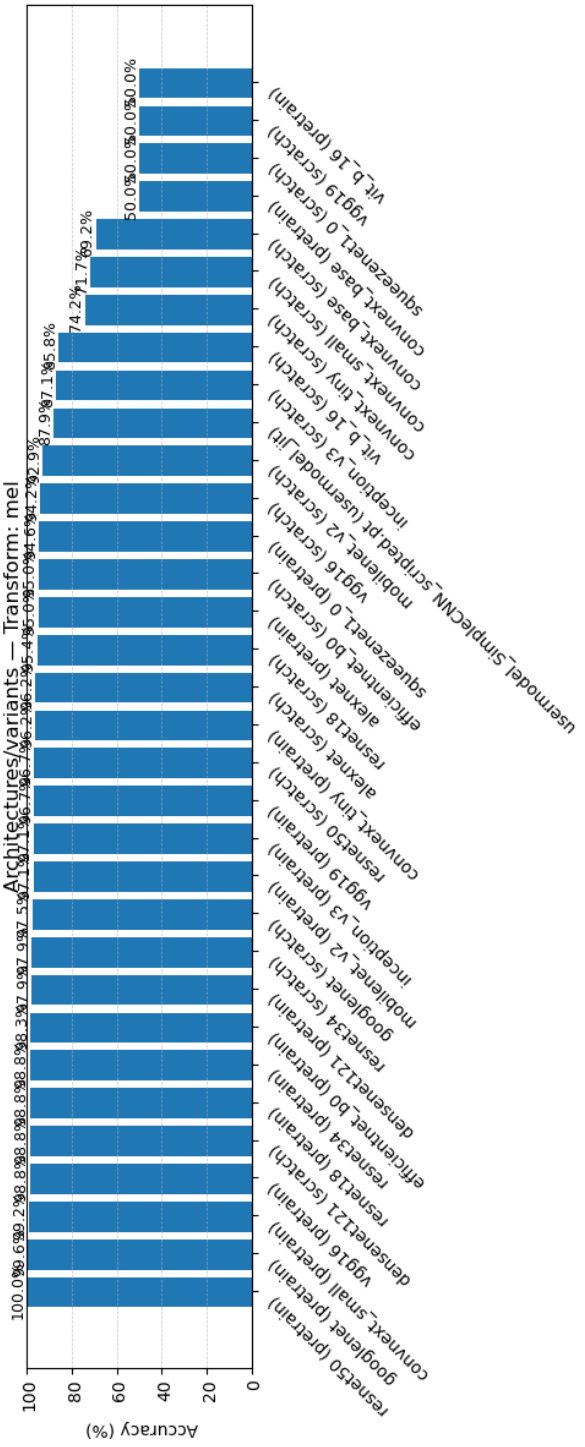

Figura 72. Ejemplo de gráfica model-center de FakeVoiceFinder: comparación de 33 modelos con el tipo de representación espectral espectrograma en escala Mel.

De esta forma, podemos evaluar cuál es el mejor modelo para el tipo de dataset utilizado, bajo las mismas condiciones de entrenamiento y tipo de datos de entrada.

Posteriormente, utilizamos el siguiente código para análisis data-centric:

```
rep.plot_variants_for_model(df, model="alexnet", variant="pretrain", metric="accuracy",
                            y_min=0, y_max=100, out_name="fig_alexnet_pretrain_accuracy.png")
```

Demos configurar dos hiperparámetros en este caso, el modelo y el tipo de entrenamiento. Para el ejemplo, se ha utilizado "alexnet" (recordmeos que FakeVoiceFinde soporta 16 arquitecturas diferentes tipos benchmark) y "pretrain" como variante lo que signidfica que hemos realizado transferencia de aprendizaje.

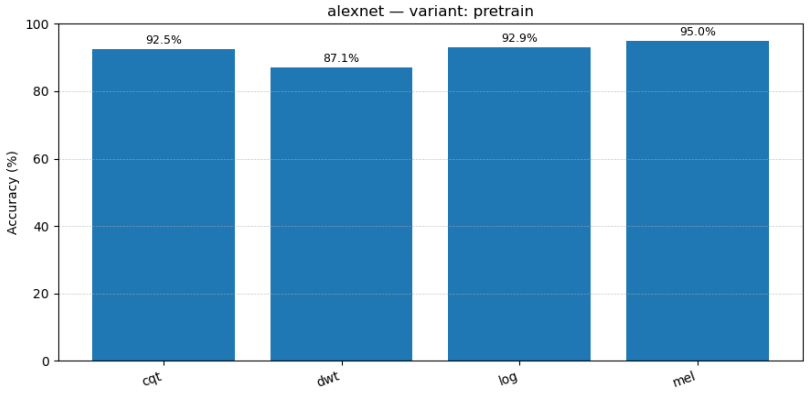

Figura 73. Ejemplo de gráfica data-center de FakeVoiceFinder: comparación de 4 modelos con diferente tipo de representación espectral, para un mismo tipo de arquitectura.

De esta forma, podemos evaluar para un mismo tipo de arquitectura qué tanto impacta el tipo de representación espectral utilizado. Para el ejemplo de la Figura 73, la diferencia entre el peor modelo obtenido y el mejor está alrededor del 8% (87.1% para dwt vs. 95% para mel).

Finalmente, es posible realizar un análisis híbrido, en el cual se evalúa de forma simultánea el impacto de la arquitectura seleccionada, el tipo de

entrenamiento y el tipo de representación espectral utilizada a la entrada de los modelos.

```
rep.plot_heatmap_models_transforms(df, metric="accuracy", vmin=50,
vmax=100,
                                   out_name="fig_all.png")
```

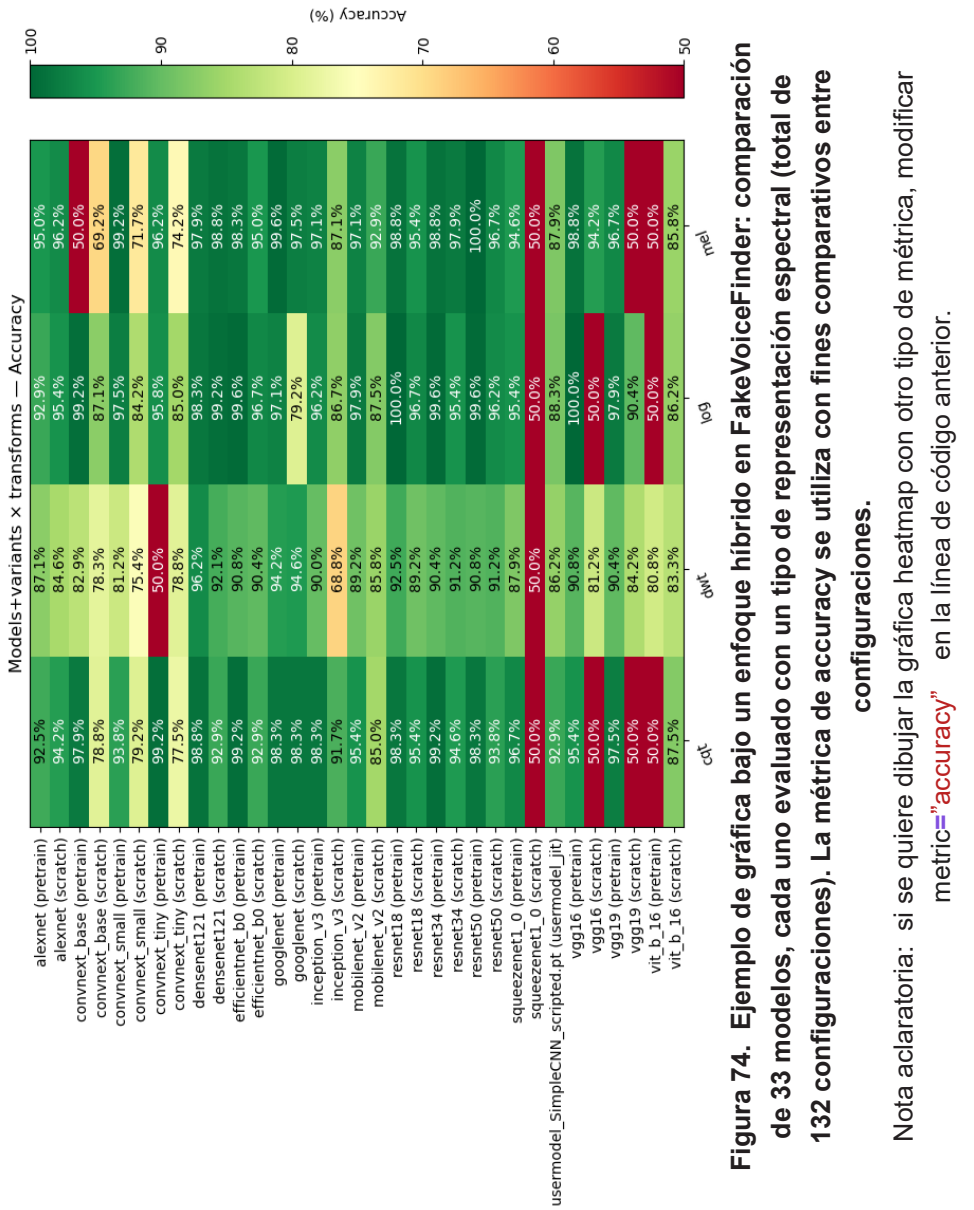

Figura 74. Ejemplo de gráfica bajo un enfoque híbrido en FakeVoiceFinder: comparación de 33 modelos, cada uno evaluado con un tipo de representación espectral (total de 132 configuraciones). La métrica de accuracy se utiliza con fines comparativos entre configuraciones.

Nota aclaratoria: si se quiere dibujar la gráfica heatmap con otro tipo de métrica, modificar metric="accuracy" en la línea de código anterior.

Este tipo de gráfica, a diferencia de las dos anteriores, no es de barras, sino que corresponde a un heatmap. En la barra de color utilizada y el valor correspondiente en términos de accuracy. Algunas arquitecturas llegan a funcionar muy bien para un tipo de representación espectral específico, y muy mal para otro. Por ejemplo, convnext_base (pretrain) alcanza un accuracy del 99.2% para la representación de espectrograma en escala logarítmica, pero de tan solo el 50% para escala mel. También, podemos fácilmente observar que, para un mismo tipo de representación espectral, por ejemplo “mel”, existen arquitecturas que permiten obtener valores de accuracy muy altos -ej. googlenet(pretrain), resnet50(pretrain)-, mientras que con otras los valores son muy bajos, ej. convnext_base (pretrain), squeezenet1_0(scratch), vgg19(scratch), vit_b_16 (pretrain)-.

Finalmente, de esta misma gráfica podemos observar qué tanto impacta el tipo de entrenamiento, si se realiza desde cero los pesos (scratch), o se realiza transferencia de aprendizaje (pretrain). Aunque existen arquitectuas como AlexNet en las que los valores de accuracy de scratch superan a los de su contraparte pretrain, en la mayoría de las arquitecturas benchmark para el dataset de entrenamiento utilizado (recordemos que solamente tenía 600 audios fake y 600 naturales), los resultados con pretrain son superiores a los de scracth. Adicionalmente, el modelo hecho a medida, denominado “usermodel_ simpleCNN_scripted.pt” en este experiemento, obtiene valores relativamente buenos (accuracy superior al 85% para todas las representaciones espectrales), pero por debajo de los mejores modelos obtenidos, ej. resnet50(pretrain).

Lo anterior, nos lleva a pensar que, si tenemos un modelo a medida, necesitaremos un conjunto de audios de mucho mayor tamaño que el del dataset utilizado, para que sea competitivo contra modelos obtenidos con arquitecturas benchmark y con trasnferencia de aprendizaje, dado que esos modelos aprendieron muchos patrones por el gran tamaño del dataseto utilizado en su entrenamiento inicial (es decir, con el dataset Imagenet). Y si bien, el tipo de patrones de Imagenet a priori pueden ser muy diferentes a los del problema de clasificación de audio natural/sintético, al transferirlos algunos de ellos puedes ser de gran utilidad para nuestro problema en cuestión.

5.3.3. Inferencia dentro de FakeVoiceFinder

Una vez entrenados y evaluados los modelos dentro de FakeVoiceFinder, el siguiente paso natural consiste en analizar su comportamiento frente a audios externos, es decir, señales que no han sido utilizadas durante ninguna de las fases de entrenamiento o validación. Este proceso, denominado inferencia, permite evaluar de manera más realista la capacidad del modelo para identificar audios sintéticos en escenarios cercanos a su uso práctico.

El ejemplo de inferencia dentro del framework está disponible en: https://github.com/DEEP-CGPS/FakeVoiceFinder/blob/main/notebooks/4-%20inference_demo_all.ipynb

Sección 1: Carga de dependencias y módulos de inferencia

```
!pip install tqdm

# 1) Rutas y pathing del proyecto (este notebook vive en notebooks/)
import sys, os
from pathlib import Path

lib_path = os.path.abspath(os.path.join(os.getcwd(), ".."))
if lib_path not in sys.path:
    sys.path.insert(0, lib_path)
print("Project root added to sys.path:", lib_path)

from fakevoicefinder.inference import InferenceRunner, FakeProbabilityGauge
```

En primer lugar, se instalan las dependencias auxiliares y se configura el entorno de ejecución. Dado que el notebook de inferencia se ejecuta desde la carpeta *notebooks*, se añade dinámicamente el directorio raíz del proyecto al *Python path*, garantizando la correcta importación de los módulos internos de *FakeVoiceFinder*. Finalmente, se cargan las clases principales encargadas de gestionar el proceso de inferencia y la interpretación de los resultados.

Sección 2: Estructura de datos y parámetros de inferencia

```
import os
from pathlib import Path
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from fakevoicefinder.inference import InferenceRunner

# -------------------------
# PATHS
# -------------------------
real_folder = "real_sample"
fake_folder = "fake_sample"
models_folder = "models"

csv_output = "inference_results_v9.csv"

device = None
```

A continuación, se definen las rutas de trabajo del proceso de inferencia, incluyendo las carpetas que contienen audios naturales y sintéticos, así como la ubicación de los modelos previamente entrenados. Asimismo, se especifica el archivo de salida donde se almacenan los resultados y se configura el dispositivo de ejecución, permitiendo una selección automática entre CPU y GPU según la disponibilidad del entorno.

Sección 3: Definición de transformaciones espectrales de entrada

```
# TRANSFORM CONFIGURATIONS
# -------------------------
transform_dict = {
    "mel": {"sample_rate":16000, "clip_seconds":4.0, "image_size":224,
"n_mels":68, "n_fft":2048, "hop_length":512},
    "log": {"sample_rate":16000, "clip_seconds":4.0, "image_size":224,
"n_fft":2048, "hop_length":256},
    "dwt": {"sample_rate":16000, "clip_seconds":4.0, "image_size":224,
```

```
"wavelet":"db6", "level":5, "mode":"symmetric"},
                "cqt": {"sample_rate":16000, "clip_seconds":4.0, "image_size":224,
"hop_length":256,
                    "n_bins":96, "bins_per_octave":24, "scale":True},
            }
```

Se define el diccionario transform_dict, el cual especifica las configuraciones necesarias para generar las representaciones espectrales utilizadas durante la inferencia. Cada entrada corresponde a un tipo de transformación e incluye tanto los parámetros comunes de preprocesamiento como los hiperparámetros propios de cada representación.

En el experimento presentado, el diccionario contempla cuatro configuraciones asociadas a los espectrogramas en escala Mel y logarítmica, al escalograma basado en la Transformada Wavelet Discreta y a la Transformada Q Constante. Todas comparten una frecuencia de muestreo de 16 kHz, una duración fija del clip de 4 segundos y un tamaño de imagen de 224 por 224 píxeles, garantizando condiciones homogéneas de evaluación durante la inferencia.

Para un experimento más específico, en el cual se desea realizar inferencia únicamente con un modelo entrenado con una sola representación espectral, el diccionario puede simplificarse dejando activa únicamente la configuración correspondiente. En este caso, es fundamental que los parámetros definidos coincidan exactamente con los utilizados durante el entrenamiento, con el fin de evitar desajustes en la distribución de los datos de entrada y asegurar la validez de los resultados obtenidos.

Sección 4: Interpretación de nombres de modelos entrenados

```
# --------------------------
# PARSE MODEL NAMES
# --------------------------
def parse_model_filename(fname: str):
    """
    Example filenames:
    alexnet_scratch_cqt_seed23_epoch010_acc0.94.pt
    squeezenet1_0_scratch_dwt_seed23_epoch001_acc0.67.pt
```

```
    vgg16_pretrain_mel_seed23_epoch004_acc0.99.pt
    """
    base = os.path.splitext(fname)[0]
    parts = base.split("_")

    # model name can include numbers e.g., "squeezenet1_0"
    # strategy: model name = everything until we hit "scratch" or "pretrain"
    variant_idx = None
    for i, p in enumerate(parts):
        if p in ("scratch", "pretrain"):
            variant_idx = i
            break

    if variant_idx is None:
        return None, None, None

    model_name = "_".join(parts[:variant_idx])
    variant = parts[variant_idx]
    train_transform = parts[variant_idx + 1]  # mel/log/dwt/cqt

    return model_name, variant, train_transform
```

La función parse_model_filename se utiliza para extraer información clave a partir del nombre de los archivos que contienen los modelos entrenados. A partir de un esquema de nombrado predefinido, la función identifica automáticamente la arquitectura del modelo, el tipo de entrenamiento empleado y la representación espectral utilizada durante el entrenamiento, incluso en casos donde el nombre del modelo incluye números o subrayados. Para ello, elimina la extensión del archivo, segmenta el nombre en sus componentes y localiza la posición correspondiente a la estrategia de entrenamiento, utilizando esta referencia para reconstruir correctamente cada atributo. En caso de que el archivo no siga el formato esperado, la función devuelve valores nulos, evitando inconsistencias durante el proceso de inferencia y permitiendo una asociación correcta y automatizada entre modelos y configuraciones espectrales.

Sección 5: Recolección de audios y modelos para inferencia

```
# -------------------------
# FILENAME PARSER FOR AUDIO
# -------------------------
def parse_audio_filename(fname: str):
    base = os.path.splitext(fname)[0]
    parts = base.split("_")
    label = parts[0] if len(parts) > 0 else ""
    audio_id = parts[1] if len(parts) > 1 else ""
    tool = parts[2] if len(parts) > 2 else ""
    return label, audio_id, tool

# -------------------------
# COLLECT AUDIO FILES
# -------------------------
valid_exts = {".wav", ".mp3", ".flac", ".m4a", ".ogg"}
audio_files = []

for folder in [real_folder, fake_folder]:
    p = Path(folder)
    if p.exists():
        for f in p.iterdir():
            if f.suffix.lower() in valid_exts:
                audio_files.append(str(f.resolve()))

# -------------------------
# COLLECT MODELS + METADATA
# -------------------------
models_info = []
pmodels = Path(models_folder)
for f in pmodels.iterdir():
    if f.suffix == ".pt":
        m_name, m_variant, m_train_t = parse_model_filename(f.name)
        if m_name is not None:
            models_info.append({
                "path": str(f.resolve()),
```

```
            "file": f.name,
            "model_name": m_name,
            "variant": m_variant,
            "train_transform": m_train_t
        })
```

En esta sección se definen los procedimientos para identificar y recolectar los audios y modelos utilizados durante la inferencia. A partir del nombre de los archivos de audio, se extraen metadatos básicos como la etiqueta de clase, un identificador y, cuando está disponible, la herramienta de generación. Posteriormente, se recorren las carpetas de audios naturales y sintéticos, filtrando únicamente los archivos con extensiones válidas. De manera análoga, se explora el directorio de modelos entrenados y, a partir de su nombre, se recupera información clave sobre la arquitectura, el tipo de entrenamiento y la representación espectral, permitiendo organizar automáticamente los insumos necesarios para el proceso de inferencia.

Sección 6: Bucle de inferencia y registro de resultados

```
# -------------------------
# INFERENCE LOOP (FILTERED)
# -------------------------
rows = []

for transform_name, transform_params in transform_dict.items():

    # Only use models trained with the same transform
    models_filtered = [
        m for m in models_info
        if m["train_transform"] == transform_name
    ]

    print(f"\n=== Transform: {transform_name} — Using {len(models_filtered)} matching models ===")

    for model in tqdm(models_filtered, desc=f"Models ({transform_name})", ncols=100):
```

```
    runner = InferenceRunner(
        model_path=model["path"],
        transform=transform_name,
        transform_params=transform_params,
        device=device,
    )

    for audio_path in audio_files:
        fname = os.path.basename(audio_path)
        label, audio_id, tool = parse_audio_filename(fname)

        try:
            out = runner.predict(audio_path)
        except:
            continue

        pred_label = out.get("pred_label")

        rows.append({
            "audio": fname,
            "tool": tool,
            "label": label,
            "model_name": model["model_name"],
            "variant": model["variant"],
            "train_transform": model["train_transform"],
            "infer_transform": transform_name,
            "real_score": out.get("real"),
            "fake_score": out.get("fake"),
            "pred_label": pred_label,
            "confidence": out.get("confidence"),
            "correct": 1 if pred_label == label else 0,
        })
```

Se ejecuta el bucle principal de inferencia de forma controlada y consistente con el entrenamiento. Para cada representación espectral definida en transform_dict, primero se filtran los modelos disponibles y se conservan únicamente aquellos que fueron entrenados con esa misma representación,

evitando desajustes entre el tipo de entrada esperado por el modelo y la transformación aplicada durante la inferencia. Luego, para cada modelo filtrado se inicializa un objeto InferenceRunner con la ruta del checkpoint y los parámetros de transformación correspondientes, y se evalúan todos los audios recolectados. Para cada archivo de audio se extraen metadatos desde el nombre, se ejecuta la predicción y se almacenan los resultados en una lista de registros que incluye la etiqueta predicha, los puntajes de clase, la confianza y un indicador de acierto calculado al comparar la predicción con la etiqueta esperada, construyendo así una tabla lista para exportación y análisis posterior.

Sección 7: Almacenamiento y exportación de resultados de inferencia

```
# --------------------------
# SAVE RESULTS
# --------------------------
df = pd.DataFrame(rows)
df.to_csv(csv_output, index=False)
print("Saved:", csv_output)
print(df.head())
```

Se consolidan los resultados obtenidos durante el proceso de inferencia en una estructura tabular. A partir de los registros almacenados en la lista rows, se construye un *dataframe* que organiza de manera sistemática la información asociada a cada audio evaluado, incluyendo el modelo utilizado, la representación espectral y las predicciones obtenidas. Posteriormente, estos resultados se exportan a un archivo en formato CSV, lo que facilita su análisis posterior, comparación entre modelos y trazabilidad de los experimentos realizados.

Sección 8: Visualización de resultados mediante mapas de calor

```
# ------------------------------------------------------------
#                    HEATMAP PLOTTING
# ------------------------------------------------------------
def plot_heatmap(df: pd.DataFrame, title: str, outname: str):
    pivot = df.pivot_table(
        index=df["model_name"] + " (" + df["variant"] + ")",
```

```
    columns="train_transform",
    values="correct",
    aggfunc="mean"
) * 100

A = pivot.to_numpy(dtype=float)

plt.figure(figsize=(12, 7))
im = plt.imshow(
    A,
    aspect="auto",
    interpolation="nearest",
    cmap="RdYlGn",
    vmin=np.nanmin(A),
    vmax=100.0,
)
plt.colorbar(im, label="Accuracy (%)")

plt.yticks(range(len(pivot.index)), pivot.index)
plt.xticks(range(len(pivot.columns)), pivot.columns, rotation=20, ha="right")

plt.title(title)

for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        value = A[i, j]
        txt = "—" if np.isnan(value) else f"{value:.1f}%"
        plt.text(j, i, txt, ha="center", va="center", fontsize=10, color="black")

plt.tight_layout()
plt.savefig(outname, dpi=150)
plt.show()
print("Heatmap saved:", outname)


# ------------------------
```

```
# GLOBAL HEATMAP
# -------------------------

df_final = df.copy()
plot_heatmap(
    df_final,
    title="Overall Accuracy (models × transforms)",
    outname="heatmap_overall.png"
)

# -------------------------
# HEATMAP PER (label, tool)
# -------------------------
unique_labels = df_final["label"].unique()
unique_tools = df_final["tool"].unique()

for lab in unique_labels:
    for tl in unique_tools:
        subset = df_final[(df_final["label"] == lab) & (df_final["tool"] == tl)]
        if len(subset) == 0:
            continue

        fig_name = f"heatmap_{lab}_{tl}.png"
        title = f"Accuracy — label={lab}, tool={tl}"

        plot_heatmap(subset, title, fig_name)
```

En esta sección se generan visualizaciones tipo *heatmap* para analizar de forma comparativa el desempeño de los modelos durante la inferencia. A partir del *dataframe* de resultados, se construyen tablas pivote que resumen el porcentaje de aciertos de cada modelo para cada tipo de representación espectral utilizada durante el entrenamiento. Estas tablas se representan gráficamente mediante mapas de calor, donde los colores permiten identificar de manera inmediata combinaciones modelo–transformación con mejor o peor desempeño. Además de un heatmap global que resume el comportamiento general del sistema, se generan mapas específicos filtrados por etiqueta y por

herramienta de generación, lo que permite analizar con mayor detalle cómo varía la capacidad de detección según el tipo de audio evaluado.

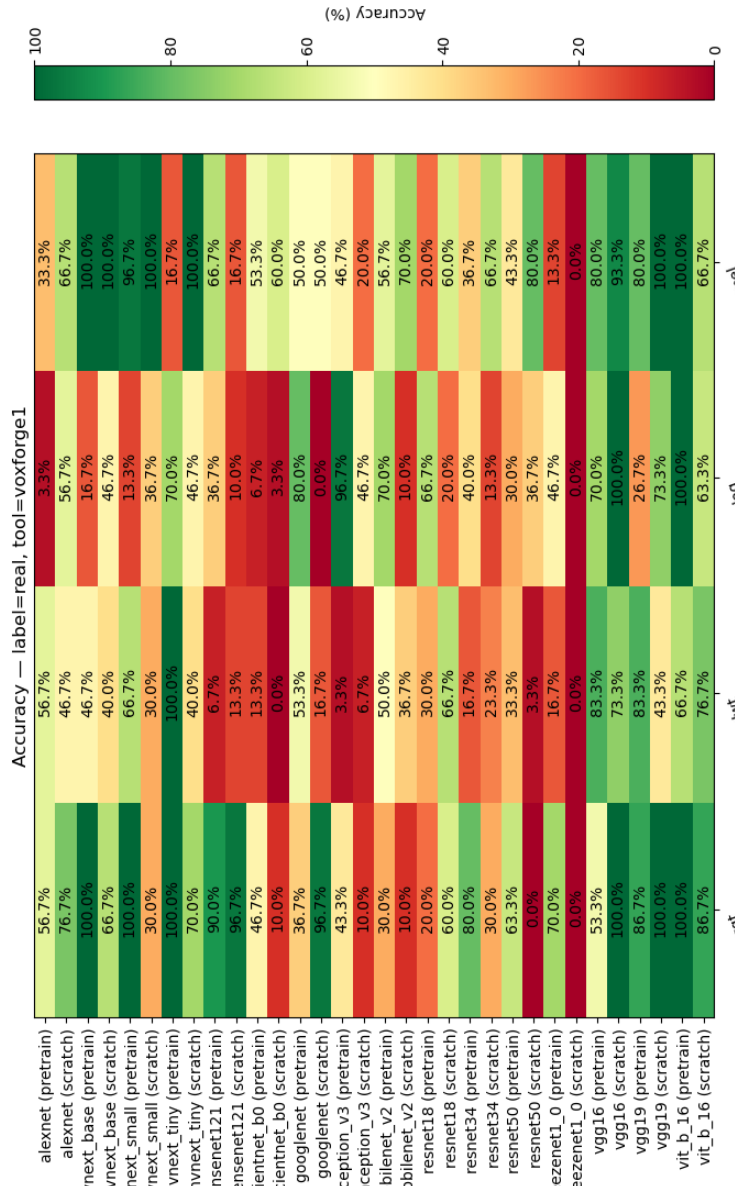

Figura 75. Ejemplo de Inferencia de FakeVoiceFinder: herramienta de gemneración de audio sintético Voxforge1.

La Figura anterior muestra un mapa de calor del desempeño de los modelos durante la inferencia sobre audios naturales generados con la herramienta *voxforge1*. Cada celda representa el porcentaje de acierto para una combinación específica de arquitectura y representación espectral, permitiendo identificar de manera visual diferencias significativas en el rendimiento. Los resultados evidencian que el desempeño del modelo depende fuertemente de la coherencia entre la representación espectral utilizada en el entrenamiento y la aplicada en la inferencia, así como del esquema de entrenamiento empleado, resaltando la importancia de evaluaciones comparativas bajo condiciones controladas.

5.1.1. Cierre del capitulo

Este capítulo final no solo cierra un libro, sino que recoge un trayecto de más de quince años de preguntas, búsquedas y aprendizajes alrededor del audio sintético y la voz humana. Desde los primeros experimentos con procesamiento digital de señales y esteganografía, pasando por la imitación de voz mucho antes de que el término *deepfake* se instalara en el lenguaje cotidiano, hasta el desarrollo de modelos de detección basados en aprendizaje profundo, este recorrido evidencia que la investigación no es una sucesión lineal de resultados, sino un proceso vivo que evoluciona junto con la tecnología y con quien la investiga.

A lo largo del capítulo se ha mostrado que cada avance trae consigo nuevas preguntas y nuevos riesgos, y que la sofisticación de los modelos generativos exige, al mismo tiempo, mayor rigor metodológico, mayor capacidad crítica y una mirada responsable sobre el impacto de estas tecnologías. En este sentido, FakeVoiceFinder surge como una respuesta madura a esa necesidad, no como una solución cerrada, sino como un marco abierto que permite comparar, cuestionar y comprender el comportamiento de los modelos de detección bajo condiciones controladas y reproducibles.

Este libro, y en particular este capítulo, no pretende ofrecer verdades absolutas, sino compartir una experiencia investigativa real, con aciertos, limitaciones y aprendizajes acumulados a lo largo del tiempo. Es una invitación a entender la ciencia de datos y la inteligencia artificial no solo como un conjunto de herramientas, sino como un ejercicio de criterio, ética y responsabilidad. Si el

lector llega hasta aquí con una comprensión más profunda del problema del audio sintético, con nuevas preguntas por explorar y con la motivación para investigar con rigor y sentido crítico, entonces el propósito de este libro se ha cumplido. Porque al final, más allá de los modelos y los algoritmos, lo que permanece es la capacidad de pensar, cuestionar y construir conocimiento con honestidad y pasión.